Learning Program for the Design of Layout Geometry of Rectilinear Looped Water Distribution Networks

by

James William Davidson

A Thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree

Doctor of Philosophy

Department of Civil and Geological Engineering
University of Manitoba
Winnipeg, Manitoba

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

LEARNING PROGRAM FOR THE DESIGN OF LAYOUT GEOMETRY OF RECTILINEAR LOOPED WATER DISTRIBUTION NETWORKS

BY

JAMES WILLIAM DAVIDSON

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree

of

DOCTOR OF PHILOSOPHY

Acknowledgments

The author extends special thanks to Dr. Slobodan Simonovic for his support and helpful suggestions.

Many of the ideas presented in this thesis grew out of work on rule based programming undertaken from 1992 to 1994 at Central Queensland University, Rockhampton, Australia. Prof. Ian Goulter supervised that work and the Australian Research Council provided funding.

Abstract

This thesis describes a computer program for optimizing the layout geometry of looped water pipe networks. The program uses production rules that it learns through autonomous experimentation with sample problems. The program consists of three components, an agitator, a rule formulator and a production system. The agitator and rule formulator enable the program to derive new rules thereby extending the ability of the production system to improve solutions. The thesis also describes an evolution based method for solving the same problem. Comparison of the two methods demonstrates the advantages of the learning program's ability to utilize previous experience.

Table of Contents

List of Figures

List of Tables

Notation

This thesis adopts a convention in which the total number of nodes in a network is N and the total number of links is M. The subscript i refers to the number of a node where the nodes number from 1 to N and the subscript j refers to the number of a link where the links number from 1 to M.

This document uses the following symbols:

C	=	cost constant
C_{HW}	=	Hazen-Williams friction coefficient for pipe material (m^2/s)
C_{m_j}	=	cost of pipe per unit length in link j ($/m)
C_t	=	cost per unit length of pipe of type t ($/m)
C_{total}	=	total cost of pipe components in a network ($)
C^*	=	constant for pipe material
C'	=	constant used to determine unit cost of pipe ($/m^{n+1})
D	=	set of all commercially available pipe sizes, $D = \{d_1,\ldots,d_T\}$
d_j	=	diameter of pipe in link j (assumes continuous diameters) (m)
d_t	=	diameter of pipe type t, $d_t \in D$
G_ℓ	=	gradient for each natural loop or path between nodes with fixed heads ℓ ($s/m^3)
H	=	maximum allowable head loss (m)
h_{f_j}	=	frictional head loss in link j (m)
h_i	=	head at node i (m)
h_{in_j}	=	head at inlet of link j (m)
h_ℓ	=	head loss between nodes with fixed heads or 0 for loop
h_{min}	=	minimum allowable head (m)
h_{out_j}	=	head at outlet of link j (m)
h_s	=	head at source (m)
i	=	subscript indicating node number, $i \in \{1,\ldots,N\}$
in_j	=	number of node at inlet of link j

J_j = head gradient in pipe j (m/m)

j = subscript indicating link number, $j \in \{1,\ldots,M\}$

K_1 = friction coefficient (s/m^2)

K_2 = friction coefficient ($m^{1.08}/s^{0.54}$)

L_j = length of link j (m)

L_{jt} = length of pipe of diameter d_t in link j (m)

L_{max} = maximum length of any path for which the gradient is the binding constraint (m)

ℓ = subscript used to designate a path between nodes with fixed heads or a natural loop

M = total number of links in a network

N = total number of nodes in a network

n = constant dependent on pipe network type (water, sewer, natural gas)

out_j = number of node at outlet of link j

P_i = the set of all links on the path connecting node I and the source

P_ℓ = set of all links on a path between nodes of fixed heads or the set of all links in a natural loop

p = constant ($1.7 \le p \le 2.0$)

Q = vector of assumed flows Q_ℓ

Q_i = demand at node i (m^3/s)

Q_j = flow in link j (m^3/s)

Q_ℓ = assumed flow in a path between nodes with fixed heads ℓ or flow in a natural loop ℓ (m^3/s)

r = constant ($4.7 \le r \le 5.0$)

S = denotes all paths other than ℓ

s = a path of the set S

T = total number of discrete pipe sizes available commercially

t = subscript for commercially available pipe diameter, $t \in \{1,\ldots,T\}$

U_i = set of upstream links adjacent to node i

V_i = set of downstream links adjacent to node i

w_ℓ = dual variable corresponding to head constraint with fight hand side equal to h_ℓ ($/m)

1. Introduction and Literature Review

1.1. Introduction

This thesis describes a new method for the design of layout geometry of rectilinear looped water distribution networks. The method utilizes a computer program that is capable of autonomous learning by the automatic formulation and testing of production rules.

The proposed method is the continuation of an approach developed by Davidson and Goulter (1991a) in which a rule based production system optimized layout geometry of branched natural gas systems. The new work applies the rule based approach to the design of looped water distribution systems. The primary focus of the new work is the selection of the appropriate layout geometry. The hydraulic design of the network components is not an objective of the thesis.

Previous work extended the rule based approach from branched natural gas systems to tree structured water distribution systems (Davidson and Goulter, 1992). Looped water distribution systems, however, represent a more complex problem than either branched gas or water networks due to differences in the required network geometry and not differences in the material properties of natural gas and water. The difficulties result from the stipulation that the networks must provide sufficient reliability in the event of pipe failure. A geometry comprised of loops has the potential to provide sufficient redundancy in the form of alternative pathways between the source and demand nodes to ensure that the system meets the demands during a link failure or planned shut down.

The literature review that follows spans several fields including engineering, management science, graph theory, computer science and artificial intelligence. Section 1.2 of the literature review, "Component Optimization of Tree Networks," explains the basic hydraulic principles of pipe networks and optimization methods for the design of components of branched (or tree) networks.

Section 1.3, "Component Optimization of Looped Networks," discusses various approaches to the optimization of looped networks and the difficulties encountered with these techniques. The problems encountered with optimization of looped networks were motivating factors for research on reliability of pipe networks. No review of the work on optimizing looped water distribution systems would be complete without covering the issue of reliability. Section 1.4, "Reliability Measures and Optimization Models", reviews some of the significant work done on pipe network reliability.

Reliability of pipe networks is highly dependent on the redundancy provided by the network layout geometry. To this point the literature review has covered the optimization methods that consider only the issue of sizing of pipe components. Although the selection of optimal layout geometry is a significant issue there has been very little success in determining what constitutes "good" layout geometry for looped pipe networks. Section 1.5, "The Problem of Layout Design," explains some of the issues and approaches to design of layout geometry of branched pipe networks.

Evolution based methods, genetic algorithms and evolution strategies, have gained popularity in recent years and researchers have applied these methods to both the component sizing and layout selection aspects of pipe network problems. Section 1.6, "Search and Adaptation" reviews evolution based methods. The method presented

in this thesis, the learning program, uses the combination of an evolution strategy and a production system. Section 1.7, "Rule Based Programming Approach" reviews previous work that applied the production system approach to pipe network problems.

The final section of this chapter, "Summary of the Literature Review and Objectives of the Research," summarizes the literature review and describes the scope of the work presented in the remainder of the thesis.

1.2. Component Optimization of Tree Networks

The load (or flow) on each link in a tree structured water distribution network depends entirely on the demand of the downstream nodes. Unlike looped distribution systems, the size of pipe components in a tree network has no effect on the load distribution. Looped networks may consist of several sources, whereas, tree networks considered in this work have only one source. Looped distribution systems provide several paths between any demand node and the source node (or source nodes in the case of multiple sources). The selection of pipe diameters for a looped network affects the distribution of loads among the various paths available. Looped networks are more difficult to design because the choice of diameters and the load distribution are interdependent. The techniques for the design and optimization of looped networks have evolved from those used for the simpler problem of designing and optimizing tree networks. This section explains techniques for the optimization of tree networks.

1.2.1. Tree Network Hydraulics

The goal in optimizing the components in a pipe network is the selection of the appropriate pipe diameter for each link in the network. The tradeoff is between cost and

performance. Larger diameter pipes cost more but result in smaller pressure losses per unit length for a given discharge. The objective is to minimize the cost of the network while providing the loads at the required heads (or pressures).

The load on each link in a tree network is easy to determine since it is simply the sum of all the downstream loads. The problem of designing a tree network consists of selecting a pipe diameter for each link that will accommodate the load on that link with adequate outlet head. Equation 1 (Fujiwara and Dey, 1988) expresses the relationship between load, pipe diameter and head gradient:

$$J_j = K_1 Q_j^{p} d_j^{-r} \tag{1}$$

where

d_j = diameter of pipe in link j (m);
J_j = head gradient in pipe j (m/m);
K_1 = friction coefficient (s/m^2);
p = constant ($1.7 \leq p \leq 2.0$);
Q_j = flow in link j (m^3/s); and
r = constant ($4.7 \leq r \leq 5.0$).

One common form of the Equation 1 is the Hazen-Williams equation (Jeppson, 1976), Equation 2, commonly used in the design of pressurized water supply networks in which $p = 1.852$ and $r = 4.87$:

$$h_{f_j} = \frac{10.7\ Q_j^{1.852}\ L_j}{C_{HW}^{1.852}\ d_j^{4.87}} \tag{2}$$

where

C_{HW} = Hazen-Williams friction coefficient for pipe material (m^2/s);
h_{f_j} = frictional head loss in link j (m); and
L_j = length of link j (m).

Equation 3 provides the outlet head of a link from the inlet head and frictional losses:

$$h_{out_j} = h_{in_j} - h_{f_j} \tag{3}$$

where

h_{in_j} = head at inlet of link j (m); and
h_{out_j} = head at outlet of link j (m).

For an existing pipe network, one can calculate the outlet head of any link from the load on the link and the inlet head. Prior to performing any calculations, the only known head values consist of the inlet heads of the links connected to the source node. Calculating the outlet heads for these links using Equations 2 and 3 provides the inlet heads for the links immediately downstream. Repeating the calculations radiating outward from the source provides outlet heads for all the links in the tree.

Optimizing the cost of pipe for a network consists of selecting pipe diameters for each of the links in a manner that produces the lowest total cost. The optimization is subject to the constraint that all outlet heads are above a required minimum under the specified load, or loads. The subsections that follow present two methods for selecting

pipe diameters. The first method is a computationally efficient technique that assumes that the network must maintain a constant head (or pressure) gradient in every link in the network. The second method, which is more computationally intensive, uses linear programming.

1.2.2. Constant Head Gradient Approach

A simple method for selecting pipe diameters consists of utilizing a constraint that operates in addition to the loads and minimum outlet heads. A maximum head gradient limits the allowable drop in head per unit length of pipe as expressed in Equation 4:

$$J_{max} \geq J_j \qquad \forall\ j \qquad (4)$$

where

J_j = head gradient in pipe j (m/m); and
J_{max} = maximum head gradient (m/m).

Equation 5 is the formula for the head gradient:

$$J_j = \frac{h_{out_j} - h_{in_j}}{L_j} \qquad (5)$$

where

h_{in_j} = head at inlet of link j (m);
h_{out_j} = head at outlet of link j (m); and
L_j = length of link j (m).

An important parameter used in the Constant Head Gradient method is L_{max}, obtained from Equation 6. L_{max} is the maximum length of a path from the source to a sink for which the gradient constraint, Equation 4, is the binding constraint. Equation 6 has the following form:

$$L_{max} = \frac{h_s - h_{min}}{J_{max}} \tag{6}$$

where

h_{min} = minimum allowable head (m);
h_s = head at source (m); and
L_{max} = maximum length of any path for which the gradient is the binding constraint (m).

If the length of the longest path from the source to a demand node is less than L_{max} then the maximum pressure gradient is a binding constraint for all links in the network, and J_j equals J_{max} for all links in the network. The head at the outlet of every link along any path is above the allowable minimum, h_{min}, if the total length of the path is shorter than L_{max}. Any further reduction in cost by reducing pipe diameters is not possible without violating the maximum gradient constraint. The Constant Head Gradient method selects the diameter of every link in a path from the source to a sink such that the gradient of every link equals the allowed maximum, J_{max}. Combining Equation 1 and Equation 4 under these conditions produces Equation 7. Equation 7 has the following form:

$$J_{max} = K_1 Q_j^{p} d_j^{-r} \tag{7}$$

where

d_j = diameter of pipe in link j (m);
K_1 = friction coefficient (s/m^2);
p = constant ($1.7 \leq p \leq 2.0$);
Q_j = flow in link j (m^3/s); and
r = constant ($4.7 \leq r \leq 5.0$).

The method assumes that continuous pipe diameters are available rather than the discrete diameters normally offered by pipe manufacturers. Equation 8 provides the optimal diameter for link j with flow, d_j.

$$d_j = \left(\frac{K_1}{J_{max}} Q_j^p \right)^{\frac{1}{r}} \tag{8}$$

Walters and Lohbeck (1993) were first to use this approach for the design of natural gas networks. That work used the pressure gradient rather than the head gradient and the product of two friction constants C* and λ replaced the K_1 term. Walters and Lohbeck (1993) applied their model to British gas networks. In Britain unit costs of various kinds of pipe were found to fit Equation 9:

$$C_{m_j} = C' \cdot d_j^n \tag{9}$$

where

C_{m_j} = cost of pipe per unit length in link j (£/m or $/m);
C' = constant used to determine unit cost of pipe (£/m^{n+1} or $/$m^{n+1}$); and

n = constant dependent on pipe network type (water, sewer, natural gas).

The list below provides the values of n for various pipe systems (Walters and Lohbeck, 1993):

sewer system n = 0.72
water mains n < 0.91
gas distribution n = 1.2

Equation 10 results from combining Equation 8 with Equation 9.

$$C_{m_j} = C'\left(\frac{K_1}{J_{max}} Q_j^{p}\right)^{\frac{1}{r-n}} \qquad (10)$$

The Darcy Weisbach equation is the governing equation in the low pressure natural gas systems modeled by Walters and Lohbeck (1993). The Darcy Weisbach equation is a variation of Equation 1 in which the value of the constant p equals 2 and the value of r is 5. Walters and Lohbeck (1993) approximated the value of n as 1.25. Substituting these values in Equation 10 produces Equations 11 and 12.

$$C_{m_j} = C\sqrt{Q_j} \qquad (11)$$

where

$$C = C'\left(\frac{K_1}{J_{max}}\right)^{0.25} \qquad (12)$$

The conclusion drawn from Equation 11 is that the cost of any link is proportional to the product of the length of the link and the square root of the flow. Equation 11 is particularly useful as a fitness function to rapidly evaluate the cost of many layout alternatives for genetic algorithms as Walters and Lohbeck (1993) accomplished. The total cost of the pipe in a network is expressed in Equation 13:

$$C_{total} = C\sum_{j=1}^{M} L_j\sqrt{Q_j} \qquad (13)$$

where

C_{total} = total cost of pipe components in a network (£ or $); and
M = total number of links in the network.

One can derive an equation similar to Equation 13 for tree networks other than low pressure natural gas systems using Equation 1 as the governing equation. The equation is valid if continuous pipe diameters are an acceptable approximation to actual costs, and the problem includes a maximum gradient constraint.

1.2.3. Linear Programming Approach

There are two problems with Equation 8 in the previous section. The first problem occurs if the head gradient is not a binding constraint. A given design problem might not specify a maximum head gradient or the specified gradient may not be binding due to one or more of the paths in the network being longer than L_{max}. The second problem is that pipe manufactures do not make pipes available over a continuous range of diameters, but rather, offer a set of discrete sizes.

A model formulated by Karmeli et al. (1968) based on linear programming overcomes both problems. A difficulty with linear programming is that the objective function and constraints must be linear functions of the decision variables. Equations 1 and 9 indicate that neither the minimum head constraints nor the objective cost function are linear with respect to the pipe diameter. Karmeli et al. (1968) recognized that any link in a network does not have to consist of a single diameter of pipe but can consist of a combination of lengths of different diameter pipes joined together by reducers, fittings that couple pipes of different diameter. Assuming that link j has length L_j, the link j could consist of several lengths of pipe L_{jt} where t represents one of the pipe diameters in the set of all available diameters, D. The lengths of these segments L_{jt} for link j must total to L_j as expressed in Equation 14:

$$\sum_{t \in D} L_{jt} = L_j \qquad \forall j \qquad (14)$$

where

D = set of all commercially available pipe sizes, $D = \{d_1,\ldots,d_T\}$;
L_j = length of link j (m);
L_{jt} = length of pipe of diameter d_t in link j (m); and
t = subscript for commercially available pipe diameter, $t \in \{1,\ldots,T\}$.

The decision variables in this approach are the lengths of pipe of a given diameter that comprise a link and both the cost (objective function) and the head losses (constraints) vary linearly with respect to the length of pipe. The Hazen-Williams formula, Equation 2, provides the head loss for any section of pipe in a pressurized water

network. Summing the contributing head losses associated with each pipe diameter provides the head loss, h_{f_j}, for the entire link j as expressed in Equations 15 and 16:

$$\sum_{t \in D} K_1 \cdot Q_j^{p} \cdot L_{jt} \cdot d_t^{-r} = h_{f_j} \tag{15}$$

where

$$K_1 = \frac{10.7}{C_{HW}} \tag{16}$$

and

C_{HW} = Hazen-Williams friction coefficient for pipe material (m^2/s);
d_t = diameter of pipe type t, $d_t \in D$;
h_{f_j} = frictional head loss in link j (m);
K_1 = friction coefficient (s/m^2);
L_{jt} = length of pipe of diameter d_t in link j (m);
p = constant for Hazen-Williams formula (1.852);
Q_j = flow in link j (m^3/s); and
r = constant for Hazen-Williams formula (4.87).

The constraints for the minimum allowable pressure require the maximum head loss, H, which is calculated from Equation 17:

$$H = h_s - h_{min} \tag{17}$$

where

H = Maximum allowable head loss (m);
h_s = head at source (m); and

h_{min} = minimum allowable head (m).

The constraints for minimum allowable head for any node i is the sum of the head loss for each link j found on the path P_i connecting the source to the node i.

The final problem formulation follows:

minimize

$$C_{total} = \sum_{j=1}^{M} \sum_{t \in D} C_t \cdot L_{jt} \qquad (18)$$

subject to

$$\sum_{j \in P_i} \sum_{t \in D} K_1 \cdot Q_j^{p} \cdot L_{jt} \cdot d_t^{-r} \leq H \qquad \forall i \qquad (19)$$

$$\sum_{t \in D} L_{jt} = L_j \qquad \forall j \qquad (20)$$

$$L_{jt} \geq 0 \qquad \forall j, \forall t \qquad (21)$$

where

C_t = cost per unit length of pipe of type t ($/m);
C_{total} = total cost of pipe components in a network ($);
M = total number of links in a networks; and
P_i = the set of all links on the path connecting node i and the source.

Fujiwara and Dey (1987) later showed that this formulation selects at most two adjacent diameters per link at optimality. At optimality most of the L_{jt} variables equal zero.

1.3. Component Optimization of Looped Networks

This section explains how methods developed for the optimization of tree networks, in particular Karmeli et al. (1968), evolved to include multiple sources and loops.

1.3.1. Linear Programming Gradient Method

Alperovits and Shamir (1977) introduced the use of the Linear Programming Gradient (LPG) method for designing components of looped networks. The difficulty in optimizing looped systems stems from the distribution of flows in the network. Unlike tree systems the layout geometry does not strictly determine the loads in the links. The Alperovits and Shamir (1977) model consists of two parts, the gradient component and the linear programming (LP) component. The method assumes an initial set of flows which satisfy the continuity constraints. The method finds the optimal size of components for the set of assumed flows using the LP component. The gradient component then adjusts the flows based on the results of the LP component producing a new set of flows. The new set of flows should produce an improved cost with the next LP model. The method performs the two components iteratively until it reaches an optimal solution where no further improvement is possible.

The LP component is the Karmeli et al. (1968) model with modifications to accommodate loops. The modifications consist of two types of constraints, constraints

that result from multiple sources and constraints that result from loops. The new formulation adds an instance of Equation 22 in each case:

$$\sum_{j \in P_\ell} \sum_{t \in D} \left[K_1 \cdot Q_j^{\,p} \cdot L_{jt} \cdot d_t^{\,-r} \right] = h_\ell \qquad (22)$$

where

- D = set of all commercially available pipe sizes, $D = \{d_1,\ldots,d_T\}$;
- d_t = diameter of pipe type t, $d_t \in D$;
- h_ℓ = head loss between nodes with fixed heads or zero for a natural loop;
- K_1 = friction coefficient (s/m^2);
- ℓ = subscript used to designate a path between nodes with fixed heads or a natural loop;
- L_{jt} = length of pipe of diameter d_t in link j (m);
- Q_i = demand at node i (m^3/s);
- p = constant for Hazen-Williams formula (1.852);
- r = constant for Hazen-Williams formula (4.87);
- t = subscript for commercially available pipe diameter, $t \in \{1,\ldots,T\}$; and
- P_ℓ = set of all links on a path between nodes with fixed heads or the set of all links in a natural loop.

The value of the right hand side of Equation 22, h_ℓ, is the head loss associated with either a loop or a path between two sources. For each pair of sources (nodes with fixed heads) the method formulates an instance of Equation 22 by proceeding along a path that connects the two sources starting at the source with the higher head so h_ℓ is greater than zero. For each natural loop the method formulates an instance of Equation 22 by proceeding around the natural loop with h_ℓ equal to zero. (Natural loops are non-overlapping loops, sometimes referred to as basic or free loops. This document uses the

term natural loop.) The additional constraints enable the LP model to determine the optimal cost solution for a set of specified flows. For each path between nodes with fixed heads and each natural loop the equations of continuity do not strictly determine the flows. An arbitrary flow Q_ℓ is assumed for each path connecting a pair of sources or each natural loop, ℓ. The method regards the set of flows Q_ℓ as a vector, **Q**. The LP component produces an optimal cost as a function of **Q**, as expressed in Equation 23:

$$C_{total} = LP(\mathbf{Q}) \tag{23}$$

where

C_{total} = total cost of pipe components ($);
Q = vector of assumed flows Q_ℓ; and
Q_ℓ = assumed flow in a path between nodes of fixed heads ℓ or assumed flow in a natural loop ℓ (m^3/s).

The gradient component of the model seeks to find an incremental change in **Q**, $\Delta\mathbf{Q}$, that produces an improved cost as expressed in Equation 24.

$$LP(\mathbf{Q} + \Delta\mathbf{Q}) < LP(\mathbf{Q}) \tag{24}$$

Equation 25 determines $\Delta\mathbf{Q}$.

$$\frac{\partial(C_{total})}{\partial(\Delta Q_\ell)} = \frac{\partial(C_{total})}{\partial h_\ell} \cdot \frac{\partial h_\ell}{\partial(\Delta Q_\ell)} = w_\ell \cdot \frac{\partial h_\ell}{\partial(\Delta Q_\ell)} \tag{25}$$

The value of each w_ℓ is the value of the dual variable for the corresponding constraint of the type in Equation 22 from the linear program. Equation 26 expresses the other term in Equation 25. The term is the partial derivative of h_ℓ with respect to ΔQ_ℓ

calculated from Equation 22. The method assumes the value of p to be 1.852, consistent with the Hazen-Williams equation.

$$\frac{\partial h_\ell}{\partial(\Delta Q_\ell)} = \frac{\partial h_\ell}{\partial Q_\ell} = \frac{\partial}{\partial Q_\ell} \sum_{j \in P_\ell} \sum_{t \in D} \left[K_1 \cdot Q_j^{1.852} \cdot L_{jt} \cdot d_t^{-r} \right] \qquad (26)$$

$$= 1.852 \sum_{j \in P_\ell} \frac{h_{f_j}}{Q_j}$$

The gradient, G_ℓ for each natural loop or path between nodes with fixed heads, ℓ, is the relative change in cost with respect to an incremental change in flow, ΔQ_ℓ. The method requires only the relative magnitude of the G_ℓ values. Therefore, the formulation eliminates the 1.852 term from Equation 26. Equation 27 provides the value of G_ℓ.

$$G_\ell = w_\ell \sum_{j \in P_\ell} \frac{h_{f_j}}{Q_j} \qquad (27)$$

The method requires a maximum step size to calculate the values for ΔQ_ℓ. Alperovits and Shamir (1977) do not provide a simple method to determine this value but rely on a "heuristic approach". The method calculates the various ΔQ_ℓ values by assigning the step size to ΔQ_ℓ for the loop or path, ℓ, with the largest G_ℓ value and assigning to the other ΔQ_ℓ variables the value of the step size in relative proportion to the corresponding G_ℓ value. Through successive iterations the step size value remains the same or decreases on success or failure of the previous step in generating an improved solution. Failure to generate an improved solution results in the previous iteration being repeated with a smaller step size. The algorithm terminates when the step size has reduced to some minimum value indicating the solution has converged on an optimum.

Later, Quindry et al. (1979) suggested that Equation 25 required additional terms to accommodate the interactions of all the paths in the network. Equation 28 provides the improved gradient:

$$\frac{\partial(C_{total})}{\partial(\Delta Q_\ell)} = -W_\ell \sum_{j \in P_\ell} \frac{h_{f_j}}{Q_j} + \sum_{s \in S} \pm W_s \sum_{\substack{j \in P_\ell \\ j \in s}} \frac{h_{f_j}}{Q_j} \qquad (28)$$

where

S = denotes all paths other than ℓ; and

s = a path of the set S.

The term w_s is the value of the dual variable of the constraint corresponding to the path s. The sign of the additional term depends on whether the path s uses the link j in the same direction as the loop or path ℓ.

There are two major difficulties with the Alperovits and Shamir (1977) method. The paper clearly explains the first.

> Experience has shown that when a network is designed for a single loading, unless a minimum diameter is specified for all pipes the optimal network will have a branching configuration. (Alperovits and Shamir, 1977)

Alperovits and Shamir (1977) recognized that optimizing a looped network results in a tree network in the single load case scenario. One method to ensure that optimization does not remove loops is to specify a minimum flow or minimum pipe diameter for each link. However, the reason for having loops in a network is to provide

alternate paths from the source to the sinks in the event of pipe failure. These paths must also provide adequate capacity. Specifying an arbitrary minimum flow or minimum diameter to ensure that loops remain, does not ensure that alternate pathways provide adequate capacity. If the minimum flow (or the minimum diameter) is too small, capacity will not be adequate to supply nodes during a failure event. If the minimum is too large the solution will not be optimal. Even in the case where multiple loads produce looped systems at optimality there is no assurance that the system will provide adequate capacity in the event of a failure either.

The other drawback to the Alperovits and Shamir (1977) method is that every spanning tree that is a subset of the initial looped network constitutes a local optimum. The method is not guarantied to produce the global optimum but rather proceeds by steepest descent to the nearest local optimum. The initial flows that the method sets arbitrarily determine the local optimum the method produces. To solve this problem, Eiger et al. (1994) developed a branch and bound method that finds the global optimum. However, branch and bound techniques can exhibit non-polynomial computational effort when pursuing the global optimum. To avoid a lengthy search, the Eiger et al. (1994) algorithm terminates when its solution and the bound on the global optimum are within a specified tolerance.

Despite the drawbacks the Alperovits and Shamir (1977) paper is a highly significant work in the field of optimization of water distribution systems. The paper points clearly to the key issue that research must resolve before optimal component sizing of looped networks is possible, a need for a clearly defined criterion for reliability.

> Forcing the network to have a fully looped configuration is not a satisfactory way of defining reliability. A more intrinsic definition is needed,

> one which depends on a performance criterion for specified emergency situations. More work should be done in this area. (Alperovits and Shamir, 1977)

1.3.2. Other Methods for Component Sizing of Looped Networks

Quindry et al. (1981) proposed a method similar to Alperovits and Shamir (1977) based on the linear programming formulation of Lai and Schaake (1969) rather than Karmeli et al. (1988). In the Lai and Schaake (1969) model the decision variables are the pipe diameters rather than lengths and the method makes the assumption that continuous pipe diameters are available. Rather than assuming flows in loops as in the Alperovits and Shamir (1977) model, the Quindry et al. (1981) model assumes known heads at all the nodes. Since fluid flow is always in the direction of decreasing head, the method can find the direction of flow in each link once it has assigned assumed heads to all the nodes. The method formulates constraints in terms of the equations of continuity of flow at each node.

The Quindry et al. (1982) paper demonstrates the optimization model on the New York City Water Supply Tunnels problem considered in the original Lai and Schaake (1969) paper. The cost of the solution in the original Lai and Schaake paper is $78.1 million and Quindry et al. (1981) obtain a cost of $63.6 million.

Templeman (1982) has drawn attention to difficulties resulting from the linearization of the problem. Templeman (1982) argues the objective is a non-convex function that when minimized over a linearly constrained region can have several local minima, the global minimum being one of these minima. The linear version of the objective function in Quindry et al. (1981) method is convex having a single global

minimum at a constrained vertex. There is nothing included in the method to guarantee that global optimum for the linear problem corresponds to the global optimum for the original non-linear problem.

Rowell and Barnes (1982) proposed a two level model for sizing components in looped systems in which the first level designs a minimum cost spanning tree. The second level of the method introduces redundant pipes to interconnect branches of the tree to produce loops. Goulter and Morgan (1984) have shown that assumptions used in this approach did not guarantee that head loss around each loop is equal to zero in the final solution, a necessary condition for hydraulic consistency.

Morgan and Goulter (1985) presented a model that combines a linear programming component with a Hardy-Cross network solver. The linear program generates a set of modifications to pipe sizes that minimize cost while maintaining the specified demands at the required pressures. The Morgan and Goulter (1985) approach used the same decision variables (i.e., the length of pipe in a link assigned a given pipe diameter) and the same objective function (cost) as the Alperovits and Shamir (1977) model. This solution space is known to be non-convex having local optima at solutions corresponding to each of the spanning tree subsets of the looped graph. The Morgan and Goulter (1985) model tends to produce tree networks in a single load case scenario similar to the method of Alperovits and Shamir (1977). However, Morgan and Goulter (1985) designed the algorithm to remove no more than one link from the layout in a single iteration.

The Morgan and Goulter (1985) model may be useful in view of a proposed approach to the problem of sizing redundant links. The proposed approach sizes the

redundant components using multiple load cases. Each load case corresponds to a worst case scenario involving combinations of fires flows and critical pipe failures. Gradient search methods can solve formulations involving multiple demand patterns. However, the number of constraints increases geometrically with the number of demand patterns. Morgan and Goulter (1985) claim that their model can accommodate a large number of demand patterns more easily than the gradient search techniques. However, the technique requires simplifications which do not guarantee optimality.

Morgan and Goulter (1985) applied their model to the New York City Water Supply Tunnels problem. The model produced a solution cost of $38.9 million which was lower than best solution obtained for this problem to that time, $41.8 million, obtained by Gessler (1982) subsequent to the work by Quindry et al. (1981).

The Morgan and Goulter (1985) model combined a simulation program with an optimization technique. There are many other models that combine simulation with optimization. Many of these models use the KYPIPE (Wood, 1980) analysis package in combination with the Generalized Reduced Gradient (GRG2) (Lansdon et al., 1984) non-linear optimization technique. The models include Su et al. (1987), Duan et al.(1990) and Cullinane et al. (1992). Models that use a simulation component can incorporate reliability by simulating worst case scenarios. Duan et al.(1990) incorporates the RAPS (Reliability Analysis of Pumping Systems) model developed by Duan and Mays (1990) and can include pumps and tanks in the optimization as well.

1.4. Reliability Measures and Optimization Models

The previous section showed that optimizing looped networks for cost removes redundant links and leaves tree networks. The reason for including loops is to provide

alternative paths to enable the network to service demand nodes during the time of a pipe failure or during planned repairs. Researchers have proposed many different approaches to quantify reliability and described optimization models that incorporate reliability since Alperovits and Shamir (1977) identified the need to incorporate reliability into optimization models. However, there is very little agreement on the proper approach. Tanyimboh and Templeman (1994) describe the current situation as follows:

> The difficulties introduced by reliability considerations are twofold. Conceptually , there is still some uncertainty about the exact meaning of the term "reliability" in the context of water supply. On a practical level the more realistic and useful a candidate measure of reliability is the more difficult and time consuming it is to measure quantitatively. (Tanyimboh and Templeman, 1994)

This section briefly reviews some of the approaches to reliability. The approaches fall into two categories, those that assess the reliability of existing networks and those models that optimize networks and include a reliability constraint or objective.

In regard to assessing the reliability of existing pipe networks there are essentially two approaches, analytical methods and simulation based approaches. In a set of companion papers Wagner et al. (1988a, 1988b) examined examples of both approaches. Generally analytical approaches are more efficient computationally but simulation based approaches can potentially yield more information such as the duration of the longest period of failure at a node, the duration of the longest period of reduced service, and the failure event in which the greatest shortfall occurred.

A logical choice for optimization models that incorporate reliability is to use the LPG method of Alperovits and Shamir (1977) with reliability constraints included. Many of the optimization models for looped networks take this approach. Unfortunately many

of these models often make simplifying assumptions regarding the behavior of networks in formulating the reliability constraints. Examples of LPG based models include Goulter and Coals (1986), Goulter and Bouchart (1990), Bouchart and Goulter (1991), Fujiwara and DeSilva (1990) and Fujiwara and Tung (1991).

The model proposed by Goulter and Coals (1986) assessed the reliability of a network by assessing the reliability of each node. However, only the effects of failure of the links immediately adjacent to each node contributed to this measure of reliability. The Goulter and Bouchart (1990) model constrained to a minimum the level of a heuristic criterion based on a fixed flow pattern in the network. However, flow patterns change when failures occur. Bouchart and Goulter (1991) formulated a reliability constraint based on expected volume deficit to consumers. The method is unique in considering the placement of valves in a network. Bouchart and Goulter (1991) showed that adding more valves rather than more pipes could improve the reliability of a network.

Fujiwara and DeSilva (1990) proposed a three stage heuristic model. The first stage assumes a set of flows in the network and designs the components for that set of flows. The second stage assesses the network under all conditions in which a single link fails. The third stage is a heuristic technique that modifies the set of flows based on the results of the second stage. The method repeats until it reaches an optimum. Fujiwara and Tung (1991) improved the technique to incorporate a non-linear maximum flow model and to directly operate on pipe sizes rather than modify flows.

1.4.1 Stochastic Loading

While much of the work on reliability concerns pipe failure, stochastic loading approaches recognize that the nature of the demands on the system are probabilistic. Very high demands on the system can cause pressures and flows to fall below acceptable levels in a manner similar to pipe failure.

Both the analytical and simulation based approaches offer perspectives on stochastic loading. Research has investigated the consequences of stochastic loading in isolation and in combination with pipe failure probability. Bao and Mays (1990) assessed the hydraulic reliability of networks under stochastic demands using Monte Carlo techniques in conjunction with network simulation over a broad range of loading conditions. That model did not consider the probability of pipe failure and resulting consequences. The main difficulty with Monte Carlo simulation techniques is that they are extremely time consuming. Tung et al. (1987) proposed a simpler model for incorporating probabilistic demands.

Goulter and Bouchart (1987) proposed an optimization model based on both stochastic loads and the probability of pipe failure. Lansey et al. (1989) proposed a model based on chance constraints to incorporate uncertainties in consumers demands and pipe roughness coefficients. Like Bao and Mays (1990) this model does not consider the consequences of pipe failure. The model uses the Generalized Reduced Gradient (GRG2) non-linear optimization technique.

1.4.2. Graph Theory Based Approaches

Tung (1986) used the concept of "cut sets" to determine the probability of nodes being isolated from the source by multiple failure events. Su et al. (1987) developed this

approach further in a model that considered the capacity of links in determining node isolation rather than the connectivity alone. Jacobs and Goulter (1991), recognizing that complete analysis of network reliability is computationally intractable, used the concept of cut sets to estimate the confidence in a measure of reliability based on a reduced number of calculations. Jacobs and Goulter (1991) found the probability of simultaneous failure events to be extremely rare.

> The probabilities of system failure ... are small to begin with and decrease very rapidly with increase in the number of simultaneous failures. (Jacobs and Goulter, 1991)

Kessler et al. (1990) proposed an approach to designing reliable looped networks that involved selecting two complementary spanning tree subsets of a looped network and sizing the components to meet the requirements of both trees simultaneously. An algorithm by Itai and Rodeh (1984) serves as the basis of the method. According to the theory, a single link failing disrupts one tree at most. The complementary tree remains intact and has sufficient capacity to satisfy demands during the failure event. Networks produced by this method are only reliable in the event of a single link failing. The method does not make any attempt to address the problem of simultaneous failure events. In view of the findings of Jacobs and Goulter (1991) neglecting simultaneous failures is arguably a reasonable approach.

The Kessler et al. (1990) approach involves sizing components on the basis of trees and therefore the problem is readily solvable by the techniques discussed for designing trees such as Karmeli et al. (1968). Kessler et al. (1990) did not claim that the technique produces the optimal solutions but rather that the model produces a solution that will meet the design loads in the event of a single failure. There can be many sets of

complementary spanning trees for a looped network. The number grows roughly exponentially with the number of loops in the network. No obvious method exists at this time to select the best set of complementary trees (Ostfeld and Shamir, 1993).

1.4.3. Entropy Based Approaches

The computational intractability of probability based measures of reliability has largely motivated the work on developing entropy based measures of reliability. The hypothesis is that looped networks that distribute flows with a high entropy are inherently reliable. According to the theory, changes in the design of a network that increase entropy increase reliability. Awumah et al. (1991, 1992) and Tanyimboh and Templeman (1993) presented work on entropy. Assuming that the entropy measures developed are computationally tractable and correspond well with reliability there are still some difficulties with entropy. Entropy measures are difficult for engineers to interpret meaningfully in terms of the trade-off between cost and reliability. Furthermore, the method to incorporate entropy in an optimization model, either as a constraint or an objective, is still unclear.

Awumah and Goulter (1992) presented a model that maximized the entropy of a network subject to a maximum cost constraint. However, this was not a network optimization model. The purpose of the model was not to produce the optimal solution but rather to illustrate the trade-off between cost and entropy and the relationship between entropy and a reliability measure, Nodal Pair Reliability. Awumah and Goulter (1992) maximized the entropy of an example network over a range of constrained costs and computed the Nodal Pair Reliability of the resulting networks. Awumah and Goulter (1992) found their entropy measure corresponds well with Nodal Pair Reliability.

However, the study was inconclusive in terms of recommending the optimal level of entropy to be used in pipe networks.

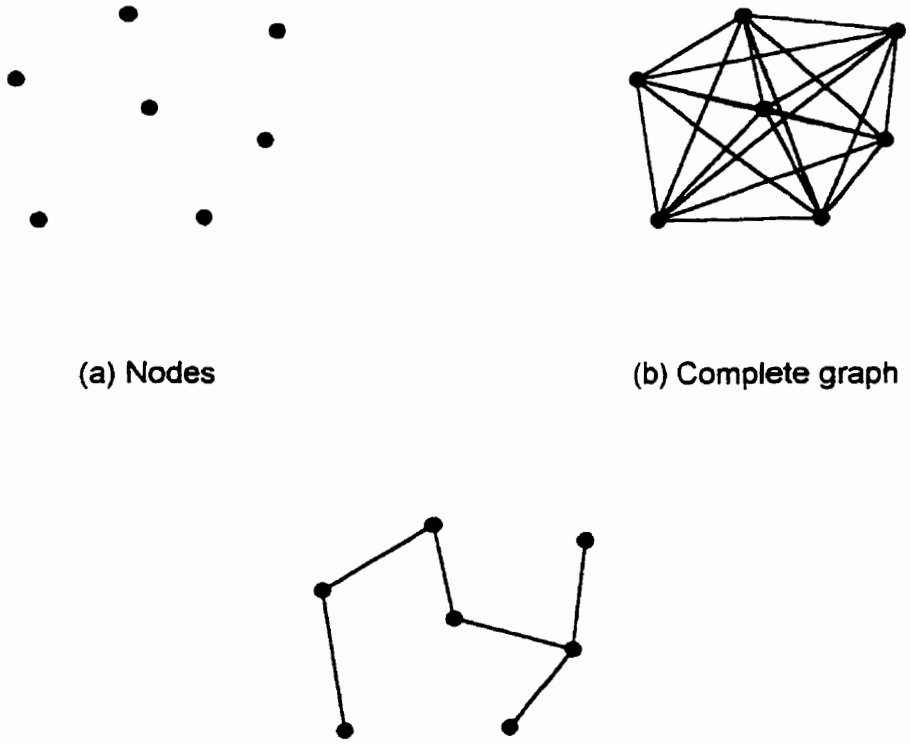
1.5. The Problem of Layout Design

The methods discussed in the previous section give an indication of the importance of layout geometry in providing reliability. This section considers issues involved in the selection of layout geometry. As in the previous sections on component sizing, this section explains the issues associated with the simpler problem of selection of layout geometry for tree networks. However, unlike the previous section these explanations do not develop into a similar discussion for looped networks. There is very little research on the geometry of looped pipe networks. Many of the techniques mentioned in the previous section on reliability indirectly address the layout geometry problem. However, these techniques assume the layout geometry of a base graph, from which the methods select a subset of links. Research has not addressed the problem of the selection of layout geometry for looped systems without any such pre-selected base graph.

1.5.1. Kruskal's Algorithm - The Minimal Spanning Tree

Kruskal's algorithm (Kruskal, 1956) is a simple method for designing optimal tree networks. Another common name for Kruskal's algorithm is the Minimal Spanning Tree algorithm. Kruskal's algorithm selects the least cost subset of links from a graph that spans (connects) all the nodes in a graph. The algorithm (Bondy and Murty, 1976) that follows assumes e_i represents any link j and $w(e_i)$ represents the cost associated with link j. The algorithm also assumes "$E\{ e_1, e_2,\ldots, e_i\}$" is the set of all links excluding e_1, $e_2,\ldots, e_i$ and "$G[\{e_1, e_2,\ldots, e_{i+1}\}]$" represents the graph formed by links $e_1, e_2,\ldots, e_{i+1}$.

1. Choose a link e_1 such that $w(e_1)$ is as small as possible.
2. If edges $e_1, e_2, \ldots, e_i$ have been chosen, then choose a link e_{k+1} from $E\{e_1, e_2, \ldots, e_i\}$ in such a way that:
   (i) $G[\{e_1, e_2, \ldots, e_{i+1}\}]$ is acyclic;
   (ii) $w(e_{i+1})$ is as small as possible subject to (i).
3. Stop when step 2 cannot be implemented further.



(a) Nodes

(b) Complete graph

(c) Minimum length spanning tree

Figure 1-1: Minimum length spanning tree from a complete graph

Figure 1-1a shows the location of a source node and several sink nodes. Figure 1-1b is a complete graph, constructed by connecting every node to every other node by

a straight line. The complete graph forms the base graph (or supergraph) from which Kruskal's algorithm chooses a subset of links that spans all the nodes.

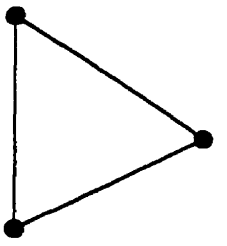
One of the major difficulties in using Kruskal's algorithm to obtain minimum cost layouts lies in obtaining the cost of the links. As the equations developed in Section 1.2 indicate the cost of a link results from the diameter of pipe selected which is a function of the load, Q, and possibly the available inlet head. However, knowing the load and inlet head for a link requires knowledge of the layout geometry.

One approach to break the vicious circle formed by link costs and layouts is to use the length of each link in place of the cost of the link in Kruskal's algorithm. The resulting tree network is the minimal length subset of the complete graph, shown in Figure 1-1c. A component sizing algorithm can use the least length layout. However, there is no guarantee that this produces the least cost solution. While the algorithm in the form presented here cannot solve the least cost network problem, several researchers (Rowell and Barnes, 1982, Davidson and Goulter, 1991b and Walters and Lohbeck, 1993) have incorporated it, and variations of it, in more complex network optimization algorithms.

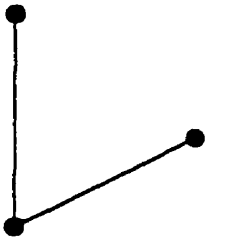
1.5.2. The Steiner Graph Problem

Another important problem with Kruskal's algorithm results from the fact that adding more nodes to the problem can actually reduce the total length of the resulting graph. This fact may seem impossible. However, limiting the final layout to only those links in a complete graph of the type in Figure 1-1b imposes unnecessary restrictions on the final layout geometry and adding more nodes can overcome these restrictions. Figure 1-2a illustrates the complete graph for a simple problem involving three nodes.

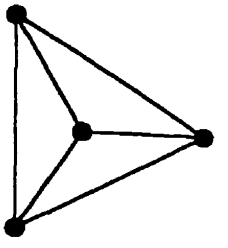
Figure 1-2b shows the minimal length spanning tree solution. Figure 1- 2c introduces a new node and the resulting layout in Figure 1-2d clearly has less total length than the layout in Figure 1-2b without the additional node.



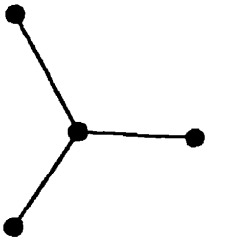
(a) Complete graph



(b) Minimum length spanning tree



(c) Complete graph with node added



(d) New minimum length spanning tree

Figure 1-2: Steiner node improves layout

For the problem illustrated in Figure 1-2 the location of the additional node that produces the optimal solution is the location where the internal angles formed by the links adjacent to the center node are all equal to 120 degrees (Bern and Graham, 1989).

However, for problems involving more than three nodes no simple solution exists. The Steiner Graph Problem is the name given to the problem, which is NP-complete (Garey, Graham and Johnson, 1977). NP-complete problems are those problems for which there are no known algorithms that find the optimal solution in a reasonable length of time for anything but trivially small instances.

The additional nodes that can form part of the layout are neither sources nor sinks. They do not contribute any additional load on the network but increase the choice of optional links that are available. The term Steiner nodes refers to these nodes. Allowing Steiner nodes into the problem makes the layout problem NP-complete. Algorithms that eliminate Steiner nodes from consideration, such Kruskal's algorithm, do not guarantee the final network solution will be optimal.

An important variation of the Steiner Graph Problem is the Rectilinear Steiner Graph Problem. This problem requires that solution layouts consist of links oriented in orthogonal directions. The requirement restricts Steiner nodes to locations that produce right angles in the final layout. Steiner nodes in this case have the additional purpose of creating the ninety degree elbows, tees and intersections needed to form rectilinear layouts. Most of the Steiner nodes increase the total length of the layout when added to transform non-rectilinear layouts to rectilinear layouts. However, sometimes rectilinear Steiner nodes will reduce the total length of the layout.

Although the possible locations of Steiner nodes are restricted in the rectilinear problem the Rectilinear Steiner Graph Problem is known to be an NP-complete problem just as the non-rectilinear variation (Garey and Johnson, 1977).

1.5.3. Work on Determining Layout of Pipe Networks

The sizing of pipe components is the primarily concern of the methods for optimization of water networks that the thesis has described to this point. Many optimization models presume to know the topology or layout of the network that produces the optimal solution before performing the optimization of pipe components. Problems occur if the optimal layout is unknown. An optimization of components sizes using a sub-optimal layout necessarily results in a sub-optimal solution. The optimization models presented in the previous sections are representative of the progress that research has made on the component sizing issue. However, research has largely neglected the optimization of layout geometry.

Liebman (1967) developed a method for optimizing layout of tree networks for the design of sewer systems. The approach was an interactive program that allowed the user to manually generate and automatically test tree systems by removing links from a base graph containing all the potential options for link locations. Liebman (1967) referred to the method as a heuristic because it does not necessarily generate the optimal solution.

The method Liebman (1967) used selected a subset of links from a base graph as does Kruskal's algorithm. Techniques of this type are similar to the component optimization methods in the previous chapters in that they provide no systematic method for generating the base graph and no systematic method for locating Steiner nodes if the methods include Steiner nodes at all. Walters (1985) described an approach that included a method for locating Steiner nodes. Walters (1985) referred to the additional nodes as junction nodes rather than Steiner nodes. The method used dynamic programming to select the location of junction nodes from a set of candidates.

Section 1.7, "Rule Based Programming Approach", describes techniques developed by Davidson and Goulter (1989, 1991b) for locating Steiner nodes in rectilinear tree networks.

1.6. Search and Adaptation

Genetic algorithms and evolution strategies are relatively new techniques for engineering optimization. The concept of biological evolution serves as the basis for both genetic algorithms and evolution strategies. Both techniques encode the salient features of solutions to a design problem in a format considered to be analogous to genetic information encoded in DNA. The techniques operate on populations of solution codings using procedures that are similar to reproduction, mutation and natural selection. A properly designed algorithm will improve the quality of solutions in a population over a series of iterations.

Of the two techniques, evolution strategy and genetic algorithms, evolution strategy is the older. Rechenberg (1965) first demonstrated the evolution strategy technique in a physical experiment that minimized the drag created by a series of plates placed at variable angles of incidence in a wind tunnel. Rechenberg (1973) and Schwefel (1977) promoted evolution strategy as an optimization technique for hard combinatorial problems that were not amenable to more traditional optimization techniques. Parallel work by Holland (1975) began later on the genetic algorithm technique which Goldberg (1989) helped popularize. Michalewicz (1992) has argued for the use of a more general term, evolution program, since researchers often modify the genetic algorithm and evolution strategy algorithms to suit their needs rather than using the algorithms in their original form.

Cembrowicz and Krauter (1977) did the earliest application of evolution based techniques to pipe networks. That work used an evolution strategy in combination with linear programming to design tree networks. The evolution strategy selected the links in the network and linear programming sized the components and evaluated costs by the method of Labye (1966), which is similar to Karmeli et al. (1968). Cembrowicz and Krauter (1992) extended the work to design looped systems with the incorporation of graph theory principles. The method partitions a looped network into two sets of links, one set that forms a tree and the other set that forms the corresponding cotree. A cotree is not itself a tree but rather a set of "cords" that are links. When the cotree combines with the corresponding tree the result is a looped network. Although the technique produces looped networks, Cembrowicz and Krauter (1992) did not directly address the issue of reliability.

Goldberg and Kuo (1986) did the first application of genetic algorithms to pipeline problems. Their genetic algorithm optimized operation of a serial pipeline. Goldberg and Kuo (1986) demonstrated the method on an example problem with 40 pumps. The results of the genetic algorithm were very nearly optimal when compared with those obtained using branch and bound integer programming.

Walters and Lohbeck (1993) formulated a genetic algorithm to design tree networks for natural gas distribution. To overcome difficulties imposed by the possibility of generating infeasible solutions the method creates new solutions by combining all the links from two parent solutions. Then a variation of Kruskal's algorithm selects a subset of links that form a tree network at random from the combined links from both parents.

Davidson and Goulter (1995) developed an evolution program for the design of rectilinear networks. The method utilized a principle proven by Hanan (1966) that the optimal Rectilinear Steiner Graph is the subset of the Hanan grid, the graph formed by extending orthogonal lines from the source node and each of the demand nodes. Potential Steiner nodes occur at the intersections of these lines. Davidson and Goulter (1995) used a coding scheme that was incompatible with the conventional genetic algorithm operators of crossover and mutation, producing infeasible solutions at an unacceptably high rate. The program divided the solution codings into separate chromosomes each consisting of a path between the source and a demand node, or a portion of that path. Rather than use a crossover operator, the method recombined chromosomes from parent solutions. A new operator, perturbation, replaced mutation to perform small modifications that preserved the feasibility of the resulting solutions.

None of the evolution-based techniques mentioned in this section with the exception of Davidson and Goulter (1995) explicitly address the issue of including Steiner nodes in the process of layout selection. However, the Walters and Lohbeck (1993) model can design optimal Rectilinear Steiner Tree networks more efficiently than the Davidson and Goulter (1995) algorithm when a Hanan grid forms the base graph. Another disadvantage of the Davidson and Goulter (1995) method is that it uses the total length of solutions as a surrogate for total cost to improve computational efficiency of the algorithm. As mentioned previously, Walters and Lohbeck (1993) demonstrated that the cost of links in a gas network with a constant pressure gradient is proportional to the product of the length of the link and the square root of the flow in the link (Equation 11). This finding enabled solutions to be evaluated rapidly on the basis of cost in the Walters and Lohbeck (1993) algorithm, rather than a surrogate measure.

One of the major advantages of evolution based techniques and genetic algorithms in particular is their ability to overcome epistasis, the tendency for an optimization technique to become trapped at local optima. Simpson et al. (1994) and Simpson and Goldberg (1994) present examples in which genetic algorithms optimize components in the non-convex solution space associated with the component sizing of looped networks. Dandy et al. (1996) demonstrated the advantages of a genetic algorithm on the New York City Water Supply Tunnels problem previously investigated by Lai and Schaake (1969) and Quindry (1981). In addition, Savic and Walters (1997) demonstrated their genetic algorithm on the New York City Water Supply Tunnels problem and two other problems from previous literature, a small six node problem (Alperovits and Shamir, 1977) and the Hanoi trunk network (Fujiwara and Khang, 1990). Like Dandy et al. (1996), Savic and Walters (1997) compared their solutions with the results obtained in the previous literature. In both papers, the results obtained by the genetic algorithm are either superior or comparable to the best results obtained by other techniques.

In contrast, Davidson and Goulter (1995) compared the performance of their evolution program for the design of layout geometry of rectilinear networks with a heuristic technique that combined procedural and rule based approaches (Davidson and Goulter, 1991a, 1991b). The evolution program occasionally produced better layouts than those of the heuristic technique but did so at a cost of increased computation time. Davidson and Goulter (1995) considered the possibility of combining the two techniques to form a single hybrid approach that combines the best features of both methods.

> This ... suggests that there is value in the development of hybrid techniques for both this and similar problems that exploit and combine the domain knowledge characteristics of time-efficient heuristic methods with

> unbiased random search aspects of evolution programs, which can identify optimal solutions that do not conform to assumptions of the heuristic process. (Davidson and Goulter, 1995)

1.7. Rule Based Programming Approach

The subsections that follow present two examples of the rule based approach to locate Steiner nodes. The learning program that this thesis describes derives from these works. The examples illustrate how rules perform optimization of network layout problems.

1.7.1. Use of Rules to Locate Steiner Nodes

Davidson and Goulter (1989) used rules to assist in the design of layout geometry of rectilinear natural gas networks. That approach used a rule base in conjunction with an interactive program using Kruskal's algorithm to design the layout of the networks. Initially Kruskal's algorithm generated minimum length layouts from a complete graph with many diagonally oriented links. The method required the user to interact with the program by providing the location of additional Steiner nodes that forced the algorithm to generate rectilinear layouts. The program could display diagonal links enclosed in rectangles to assist in the selection of Steiner node locations by an approach referred to as the "boxplot" method. Figure 1-3 shows a diagonally oriented link connecting two demand nodes. The demand nodes are shown as solid circles. A rectangle similar to that provided by the boxplot method encloses the link. Circles indicate the two potential locations for Steiner nodes at the two vertices of the rectangle that are not demand nodes. If a Steiner node exists at either of these two positions on a further iteration of Kruskal's algorithm, the algorithm selects the two sides of the

rectangle adjacent to the Steiner node in preference over the diagonal in the current layout.

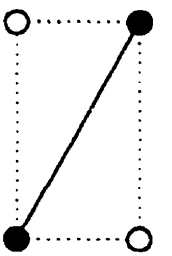


Figure 1-3: Diagonal link enclosed in a "box"

Davidson and Goulter (1989) did not present the method as an optimization technique, but rather the intent was to automate the tasks of layout design, component sizing and quantity take-off. However, leaving the selection of the Steiner nodes entirely in the hands of the user is a weakness of the method. Some of the Steiner nodes, if properly located, could reduce both the total length and the total cost of the network. The approach also included a rule based program to search for potential locations of cost improving Steiner nodes and select those locations prior to the execution of the boxplot procedure.

Figure 1-4 illustrates two examples of the types of pattern sought by the rule based program. In each case the pattern consists of two links. Each of the links forms the hypotenuse of a right triangle and the right angle vertices of the two triangles must overlap in the case of the pattern in Figure 1-4a but do not overlap in the pattern in Figure 1-4b. If the program finds patterns similar to either of those in Figure 1-4 among

any two links in the network a rule recommends including the Steiner nodes at the positions designated by the circles in the figure.

(a) Pattern with overlapping triangles (b) Alternative pattern

Figure 1-4: Example patterns for rules

Table 1-1 is a simplified version of the rule based program written in an English-like syntax, yet similar to the Prolog code of the original program. A program of this type uses a declarative form requiring a control strategy or inference engine to determine the procedural process required to solve the problem. As illustrated in Table 1-1 the program consists of three parts, a goal, a rule base, and a data base. The control strategy attempts to apply the rules in the rule base to the facts asserted in the data base in order to determine if it can accomplish the goal. In this instance the goal is to find the location of the Steiner nodes as defined by the rules.

Table 1-1: Example rule based program to locate Steiner nodes

Data Base

There is a segment connecting (4, 8) and (1, 5).
There is a segment connecting (5, 1) and (8, 4).

Rule Base

If

there is a segment a segment connecting (X1, Y1) and (X2, Y2)

and

there is a segment connecting (X3, Y3) and (X4, Y4)

and

$X1 < X2 < X3 < X4$

and

$Y3 < Y4 < Y1 < Y2$

then

the dummy nodes are located at (X2, Y1) and (X3, Y4).

If

there is a segment a segment connecting (X1, Y1) and (X2, Y2)

and

there is a segment connecting (X3, Y3) and (X4, Y4)

and

$X2 < X1 < X3 < X4$

and

$Y3 < Y4 < Y2 < Y1$

then

the dummy nodes are located at (X1, Y2) and (X3, Y4).

Goal

Find the location of the dummy nodes

The advantage of this approach is the separation of the rules, which describe what the program can accomplish, from the control strategy, which defines the sequence in which the program is to use the rules. The control strategy is hidden from the programmer, leaving the programmer free to concentrate on what the program is to accomplish rather than concentrate on how the program is to achieve this. A similar

approach forms the basis of the learning program. The learning program, using a similar type of architecture, learns by acquiring new rules. The method for applying the rules, the control strategy, remains unchanged by the learning process.

The rules used to identify patterns in the network take the form of "if ... then ..." statements or predicates. The "if" clause of a rule considers the relationship between the coordinates of the end nodes of links described in the data base. The rules represent the coordinates as variables. The control strategy assigns values to the variables in every possible combination that can result from the set of links in the data base. Any set of coordinate values that satisfies all the relations of an "if" clause causes the corresponding statements in the "then" clause to return the values of coordinates of the Steiner nodes.

For simplicity the rule base in Table 1-1 consists of only two rules. However, the patterns in Figure 1-4 require sixteen rules to completely define in all the different possible variations and orientations. Similarly the data base in Table 1-1 contains entries for only two links. During the solution of an actual problem the method requires an entry for every link in the network.

The rule based program demonstrated by Davidson and Goulter (1989) was relatively primitive. Davidson and Goulter (1989) considered the possibility of developing a larger set of patterns similar to those in Figure 1-4 and testing those patterns on a large set of networks formed from randomly generated nodes. The tests would assess the relative effectiveness of each of these patterns based on the average change in cost that results when the program uses rules based on a particular pattern.

Davidson and Goulter (1989) speculated that a future version of the rule based program would search for patterns anticipated to be most effective based on their history and only resort to less effective patterns if the initial search fails. The future rule base proposed in Davidson and Goulter (1989) is similar to the rule based learning system proposed in this document. However, the learning system takes this concept further. The learning program automates the process for generating new patterns and derives the patterns from successful instances of random manipulation of solutions.

While the rule based program of Davidson and Goulter (1989) has features that serve as the basis for a learning program there are some serious limitations on the architecture of this program that require resolution. The shortcomings are in the manner in which the program defines rules and the nature of the control strategy.

The first of these shortcomings is that the consequent, or "then", clause of the rule does not completely define the desired modification to the network. The rules add Steiner nodes but they do not explicitly describe how the Steiner nodes connect to the rest of the network. The procedure requires the execution of Kruskal's algorithm to connect the new nodes to the network. The results of the subsequent execution of Kruskal's algorithm may not be the desired layout. There is no way to rate the anticipated effect prior to the execution of Kruskal's algorithm in terms of change in cost or change in length with certainty.

The problems with the control strategy are twofold. First, goal driven control strategies terminate a search when the goal is satisfied, or when the strategy has tested all the rules and the goal is not satisfied. Alternatively, goal driven systems can enumerate all instances that satisfy the goal rather than terminating after the first

instance. In either case this type of strategy is not very effective if the rules create incremental changes in the data. Incremental changes are likely to occur with a large rule base where modifications to the network made by a rule can create the conditions in which another rule will apply. A simple solution might be to run the goal driven system repeatedly in multiple passes until the goal fails to be satisfied, indicating that no more rules apply. While this simple strategy may enable incremental improvements it will not overcome the second problem with the control strategy, the difficulties encountered when different rules act on the same part of the network. The control strategy must incorporate some mechanism for resolving conflicts. The type of control strategy that accommodates iterative application of rules and conflict resolution is a data driven or pattern directed strategy (Bratko, 1986).

1.7.2. An Improved Rule Base and Control Strategy

The Davidson and Goulter (1989) method was inconsistent due to the heavy reliance on user decision making in the selection of the location of Steiner nodes. To improve the technique Davidson and Goulter (1991b) developed a method that combined Kruskal's algorithm, Dijkstra's algorithm, and heuristic approaches to automate the selection of Steiner nodes. However, experience with the new algorithm demonstrated that certain deficiencies appeared in the layouts the algorithm generated. Attempts to modify the algorithm to correct these problems proved to be unsuccessful. Modifications to the procedure to correct a particular problem invariably resulted in the production of layouts that were deficient in some other aspect.

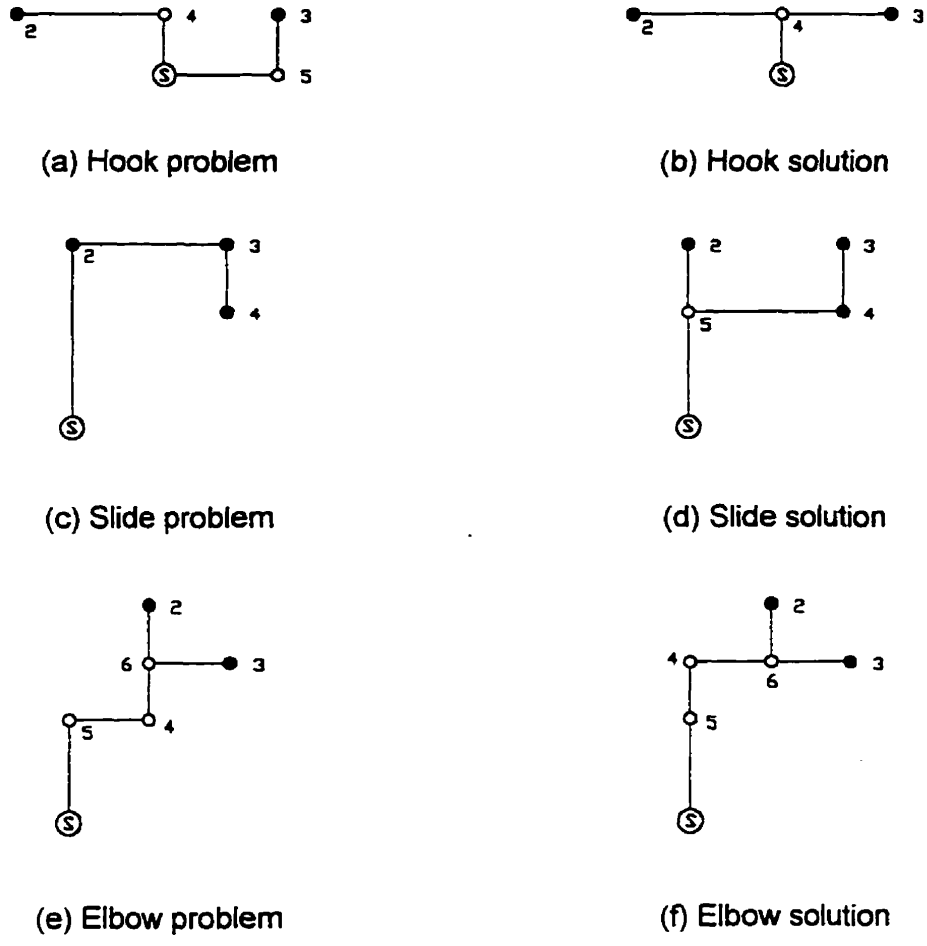


Figure 1-5: Improved patterns for rules

Davidson and Goulter (1991a) developed a rule based program to diagnose and correct the known deficiencies in the new algorithm. The new rule base consisted of three categories of rules referred to as "hooks", "slides" and "elbows." Figure 1-5 shows examples of each rule. "Hooks" reduce the total length of the system. "Slides" improve the hydraulics of the system by moving branches as close to the source as possible without affecting the total length. "Elbows" remove unnecessary bends in the system.

All three categories of rules produced a rule base that was substantially larger than the previous rule base consisting of approximately 70 rules. The new system addressed conflict resolution and problems caused by incremental changes. The new rule base incorporated rules that made explicit modifications to the network geometry rather than merely adding Steiner nodes. This eliminated the need for repeated iterations of a layout generation algorithm such as Kruskal's algorithm.

The conflict resolution strategy employed by Davidson and Goulter (1991a) activates a rule in two stages. The first stage searches the data to find every possible opportunity to apply every rule. The unification and backtracking algorithms that are part of Prolog's inference engine accomplish this in a manner similar to the older Davidson and Goulter (1989) program. Once the program has found a combination of data that fulfills a rule's antecedent clauses the rule has "instantiated". The program records a copy of the name of the rule and the data that instantiates the rule in an area of the computer memory designated as the conflict set. The rule has now "triggered". A triggered rule may alter the data but at this time the rule does not do so. The program holds the rule on "stand by" while the control strategy investigates other instantiations of this and other rules.

The second stage in the activation of rules begins once the program has recorded all the successful instantiations of the rules in the conflict set. The program resolves conflicts by rating the different instantiations of the rules according to the predefined criteria. The program fires the instantiation of the rule found to be the best in the conflict set. Firing the rule simply means that the program executes the procedures described in the rule's consequent clause. The consequent clauses modify the data in the data base.

Most systems that follow this design fire only one instantiation in the conflict set. These programs normally delete the remaining rule instantiations in the conflict set and repeat the entire process from the beginning of the triggering phase. The program will find a whole new set of rule instantiations since the data have changed by the firing of a rule. The program repeats the process of triggering and firing rules until it has modified the data to the point at which no further improvement is possible and no rules instantiate.

For a large network rules may instantiate in parts of the network that are separated far enough from one another that conflict between rules does not occur. Rather than fire only one rule in the conflict set, the strategy devised by Davidson and Goulter (1991a) has the potential to enable all rules to fire in one iteration if no conflicts occur. The method involves firing all the rules in the conflict set in order of priority but instantiating each rule a second time immediately prior to firing. If the rule can successfully instantiate the second time, those rules at a higher priority that have fired previously have not altered the data that the rule requires. If a rule cannot instantiate successfully a second time, the rules that have fired previously have altered the data required by the rule to fire successfully. The program deletes the rule from the conflict set and the rule does not fire in this case. Figure 1-6 shows a flowchart of the control strategy.

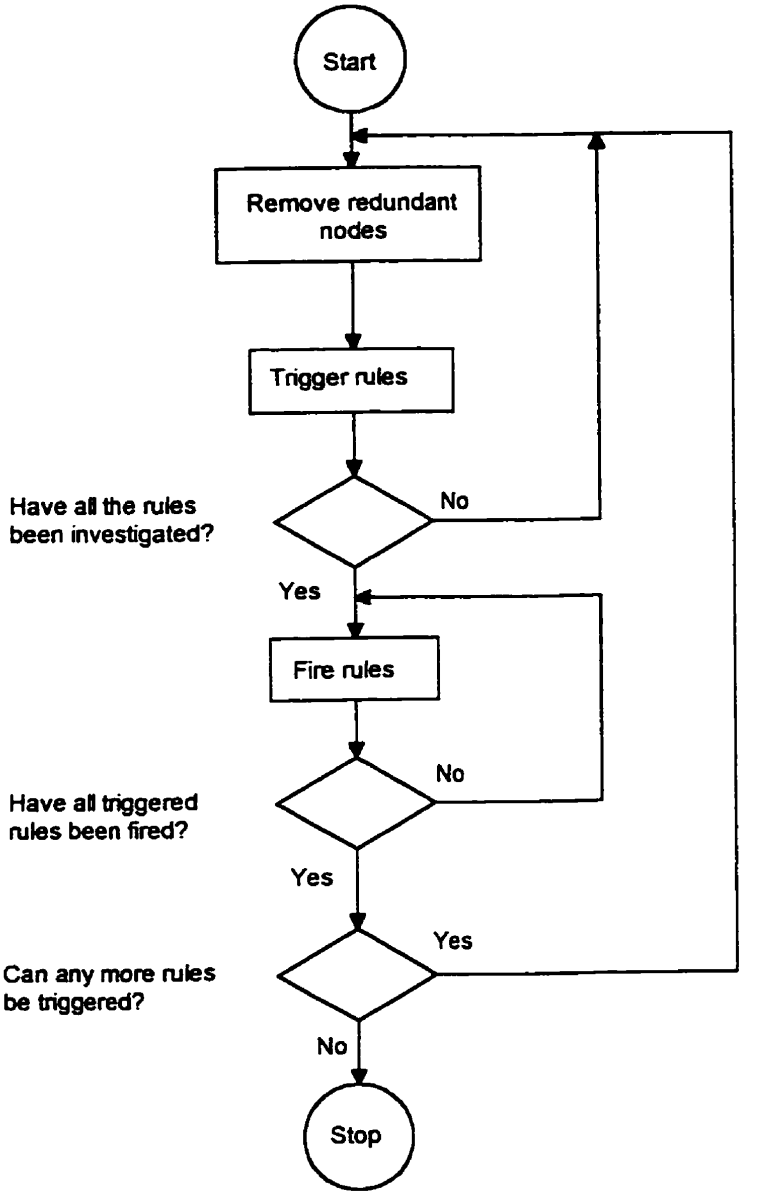


Figure 1-6: Control strategy with conflict resolution

Another aspect of the Davidson and Goulter (1991a) rule based program is the use of a focused search. The program organizes rules in a tree structure based on common features. For example, in a rectilinear network the first link examined by a rule may be either an east-west oriented link or a north-south oriented link. The method partitions rules in a rule base into two sets, rules in which the first clauses describing a link describe an east-west oriented link, and rules in which the first clauses describing a link describe a north-south oriented link. The effect of this partitioning is that comparing every link in the network to every rule in the rule base is no longer necessary. The program only needs to compare half of the rules in the rule base with each link, depending on the orientation of the link.

The method further partitions each of the partitioned rule bases. In the Davidson and Goulter (1991a) rule base the partitioning continues until the rule base took the form of a tree structure. With this approach, the number of rules contained in the tree is proportional the breadth of the tree and the time required to search the tree is proportional to the depth of the tree. Since the breadth of the tree is an exponential function of the depth, the search time increases sub-linearly (logarithmically) with the number of rules. If a method uses this type of structuring of the rule base, very large rule bases can be very efficient from a computational standpoint.

The difficulty with large rule bases is not the computational efficiency but the time and effort required to formulate and test large numbers of rules. The rule based learning program described in this thesis automates this process.

1.8. Summary of Literature Review and Objectives of the Research

1.8.1. Summary of Literature Review

The literature review has described methods that successfully optimize tree networks. However, there are problems with optimization methods for looped networks. Looped networks tend to lose their redundancy and become trees when optimized. The problem has motivated the research on reliability in the hope of developing a new objective or constraint for optimization models. However, attempts to define reliability have only revealed the complex and unstructured nature of the reliability problem.

Most of the work on reliability has shown that redundancy and therefore layout geometry is a critical issue. Most of the optimization techniques reviewed in this chapter do not consider network geometry, or select a subset of links from an initial base graph. None of the optimization techniques reviewed concerning looped water distribution systems present any method for designing the base graph or consider the issue of introducing Steiner nodes.

Evolution based techniques and rule based techniques are effective tools for solving the layout geometry for tree networks with the inclusion of Steiner nodes. Both the evolution based and rule based techniques have shortcomings. The learning program presented in this thesis combines both approaches into a single hybrid technique.

1.8.2. Objectives of the Research

The learning program is a new method for the design of layout geometry of rectilinear looped water distribution systems. The method uses production rules to

optimize layout geometry similar to previous work done on rectilinear tree gas networks. However, unlike the previous work there is an absence of domain knowledge to serve as the basis for formulating production rules. There appears to be no readily available source of information on the characteristic features of reliable and efficient geometry for looped networks. The learning program is an experiment in the design of rule based programs in the absence of domain knowledge. The learning program acquires rules autonomously by learning from the successful application of evolution based searches on representative sample problems.

The approach is new and untried in any other problem domain. For this reason the principal objective of the thesis is to establish if the proposed method can solve a simplified version of the layout design problem. The formulation of the simpler layout problem takes into consideration the need to reduce the amount of the computer programming to a manageable level, to reduce the computational effort of the programs, and the need to clearly assess the success or failure of methods in view of the unstructured and potentially conflicting objectives of the real problem.

The simplifications consist of the four following changes:

1. The sample problems are small networks.
2. The layout geometry is strictly rectilinear based on Hanan grids, as in the Davidson and Goulter (1995) model.
3. A network is reliable if it provides adequate capacity to all demand nodes at the required pressure for a single demand pattern in the event that any single link fails. Multiple loading conditions, stochastic loading, or the consequences of multiple link failures are outside the scope of the work.
4. The program optimizes the networks on the basis of least length rather than least cost. This simplification reduces computational effort of the program and eliminates the need to develop a component sizing algorithm that does not remove redundant links. Research on incorporating a cost optimization method should only

proceed if the program successfully demonstrates that learning on the basis of the simpler objective of length minimization is possible.

The learning program required several new algorithms. The chapters that follow document these algorithms and the problems they address. The list that follows includes five of the more significant algorithms:

1. The program incorporates an algorithm that efficiently tests the feasibility of solutions on the basis of the requirement of providing full connectivity in the event of any single link failing.
2. The program includes an efficient method to randomly generate feasible solutions in response to a solution space dominated by infeasible solutions.
3. Traditional approaches to random modification (mutation) produced infeasible solutions. The learning program incorporates a method that performs random manipulations on feasible solutions that preserves feasibility.
4. Learning rules requires a method that automatically derives and generalizes rules from successful instances of random search.
5. The learning process incorporates a method that tests the validity of rules that the program has acquired.

The organization of the remainder of the thesis is as follows. Chapter 2, "Data Representation Scheme", describes the method of representing rectilinear networks. Algorithms described in all the chapters that follow utilize the representation scheme. Chapter 3, "The Feasibility Testing Algorithm", describes the method for testing the feasibility of networks that use the representation scheme in Chapter 2. Chapter 4, "Generating Feasible Solutions", introduces a method for generating solutions that are feasible on the basis of the test described in Chapter 3. Chapter 5, "An Evolution Program", describes an evolution based optimization method that uses the concepts in Chapters 2 through 4. The performance of the evolution program serves as a basis of

comparison for the learning program, which Chapter 6, "The Learning Program", describes. Chapter 7, "Experimental Trials", documents a series of tests that evaluate and compare the methods of Chapters 4 through 6. The final chapter is Chapter 8, "Conclusions". This chapter lists the contributions made by the thesis and several possible directions for future research.

The thesis includes four appendices. The first three, Appendices A through C provide detailed information on the internal mechanisms of the learning program. Appendix D provides a listing of the source code of the program along with listings of the sample problems used in Chapter 7.

2. Data Representation Scheme

2.1. The Hanan Grid

All of the network problems considered in the thesis consist of a single source and multiple sink nodes. Together the source and sink nodes constitute a set referred to as basic vertices. All links in the networks are straight line segments. Therefore, rectilinear layouts require nodes in addition to the basic vertices to produce elbows, tees and intersections. The term "Steiner node" designates a node of this type that is neither a source nor a sink.

(a) Set of basic vertices (b) Hanan grid

Figure 2-1: Basic vertices and Hanan grid

According to Hanan (1966) the optimal rectilinear tree network is a subgraph of the set of lines formed by drawing two lines, one north-south and one east-west, through each basic vertex. Figure 2-1 illustrates the grid formed by the lines, known as the Hanan grid. Figure 2-1a shows the set of basic vertices and Figure 2-1b shows the same set of nodes with a north-south and an east-west line drawn through each of the nodes. Candidate locations for Steiner nodes are the intersections of the north-south and east-west lines.

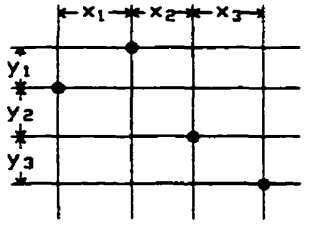
Although Hanan (1966) established that the optimal rectilinear tree network is a subset of the Hanan grid, a similar proof for the optimal rectilinear looped network does not exist. The methods presented in the thesis use the Hanan grid in the absence of the proof that the optimal looped network is a subset of the grid. The methods search for the minimum length, fully looped spanning subset of the Hanan grid.

2.2. Representation of Nodes Data

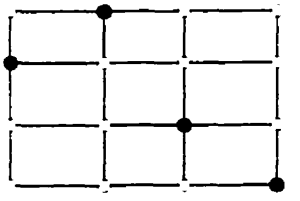
This section explains the primary data representation scheme used by the program. The representation scheme consists of four matrices designated nodes, east links, south links, and cells. In addition to the four matrices the representation scheme includes data that specify the spacing of nodes in the grid. Two vectors, one for east-west spacing (**X**) and one for north-south spacing (**Y**), represent the lengths of the line segments in Figure 2-2a. Figure 2-2a shows the elements of these vectors (**X** = $<x_1, x_2, x_3, \ldots>$ **Y** = $<y_1, y_2, y_3, \ldots>$) as dimensions.

With a few modifications the graph in Figure 2-2a produces Figure 2-2b. Figure 2-2b shows all potential Steiner nodes as circles that are not solid. Lines that pass through the nodes at the extreme east, west, north and south create the outer rectangle of the Hanan grid. The grid in Figure 2-2b eliminates all segments that extend beyond the outer rectangle. In Figure 2-2c a matrix represents the set of nodes in Figure 2-2b. Each row in the matrix corresponds to an east-west line in the Hanan grid and each column in the matrix corresponds to a north-south line. Therefore, each element of the matrix represents the position of a node. A value of 1 for an element in the matrix designates a basic vertex and a value of zero designates a potential or actual Steiner

node. A matrix similar to Figure 2-2c and two length vectors, **X** and **Y**, are all the data required to represent any rectilinear network problem.



(a) Hanan grid showing X and Y vectors



(b) Hanan grid with Steiner nodes

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(c) Matrix representation of nodes

Figure 2-2: Length vectors and nodes matrix

Unique north-south and east-west grid lines are associated with each basic vertex. The grid lines create the rows and columns of the matrices. Therefore, in matrices similar to Figure 2-2c there is necessarily one, and only one, element equal to 1 in each row and each column. Unique north-south and east west grid lines still exist for each node even if two or more basic vertices have identical x or y coordinates. However, the grid lines in this case separate by a gap of zero distance.

2.3. The Modified Hanan Grid

This section explains two additional modifications to the Hanan grid representation scheme. The first modification is the inclusion of a buffer region of links and cells. The second modification attributes to the Hanan grid the "wrap-around" topology of a two-torus. The two modifications impart properties to the Hanan grid that prove useful to the feasibility testing algorithm. Chapter 3 describes the feasibility testing algorithm and explains the benefits of the modifications.

2.3.1. Convention for Representation of Links Data

More than one matrix represents the links in the Hanan grid. Using several matrices to represent the grid requires the adoption of a convention concerning the use of indices. The convention precisely defines the relationship between elements of different matrices with similar indices. For example, without the convention the relationship between the node element at row i column j and a link with the same indices is unclear. The position of the link in relation to the node is not immediately obvious. This subsection explains the convention concerning the representation of links. The algorithms explained in the following chapters and the appendices make use of the convention.

The convention for the indices of links derives from the indices of the nodes matrix. Each node at row i column j has four links immediately adjacent to it, one to the north, east, south and west. The convention refers to the links using the indices of the node, as north link (i, j), east link (i, j), south link (i, j), west link (i, j). Figure 2-3a illustrates the four links adjacent to a node. The arrows do not represent the direction of flow in the

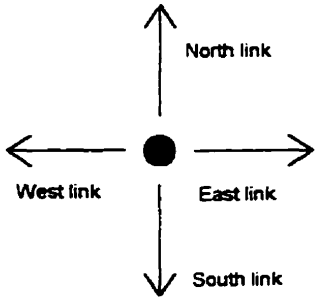
link, but rather, represent the relationship of the link to the node having the same index values.

Storing the links data in four separate matrices (north, east, south and west) is doubly redundant. The north link of node (i, j) is the south link of node (i - 1, j) and the west link of node (i, j) is the east link of node (i, j - 1). The representation of link data requires only one of either the east or west links and one of either the north or south links for every node. The convention adopted by the thesis represents the Hanan grid using only the east and the south adjacent links for every node.

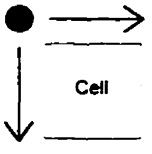
Figure 2-3b shows the same node and adjacent links as Figure 2-3a without the north and west links. Figure 2-3b also shows a rectangular region referred to as a "cell". Chapter 3 explain the use of the cell. Each of the four elements in Figure 2-3b, the node, the east link, the south link and the cell, have the same index values. Figure 2-3c demonstrates that a representation of the entire Hanan grid requires the east and south adjacent links only.

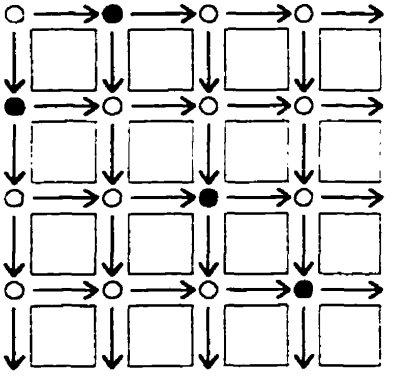
Using the convention the link to the south of node (i, j) is south link (i, j). However, the link to the north of node (i, j) is the link to the south of node (i - 1, j), or south link (i - 1, j). A similar relationship exists for links in the east-west direction. The link to the west of node (i, j) is east link(i, j - 1).

The extreme rows and columns of the Hanan grid play an important role in the algorithms that follow. The explanations that follow do not use the modifier "most" in reference to the extreme rows and columns as in the expression "east-most column". Instead the term "east column" designates the column at the extreme east of the grid.



(a) Links adjacent to a node

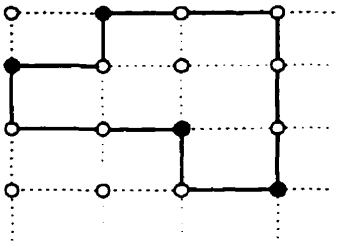




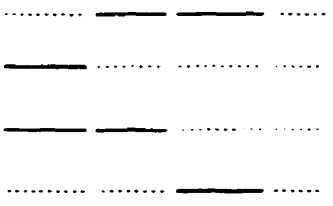
(b) Components of a single grid element (c) Schematic representation of 4x4 grid

Figure 2-3: Index convention for the Hanan grid

For grids similar to Figure 2-3c the east links in the east column and the south links in the south row play an important role in the feasibility testing algorithm. There can be no east links selected from the east column and no south links selected from the south column. These links extend beyond the outer rectangle of the Hanan grid and therefore cannot constitute part of a layout solution. These particular east and south links are dummy links and have no associated length.



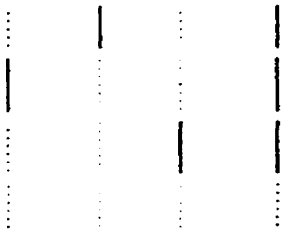
(a) Example solution layout



(b) East links

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

(c) Matrix representation of east links



(d) South links

$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

(e) Matrix representation of south links

Figure 2-4: East and south links matrices

Figure 2-4a shows a looped network that consists of a subset of links arbitrarily selected from the graph in Figure 2-3c. This section does not discuss the selection process. The selected subset serves only to illustrate the representation scheme. Chapter 4 explains the procedures for selecting links.

Figure 2-4a shows the links selected as part of the network as solid lines and the links not selected as dotted lines. East links in the east column appear as dotted lines since they cannot be part of the layout solution. For the same reason south links in the south row appear as dotted lines. Figure 2-4b shows only the east links using the same representation scheme as Figure 2-4a. Figure 2-4c is a matrix representation of the east links in which each element in the matrix corresponds to an east link. A value of 1 for that element indicates that the corresponding link constitutes part of the network and 0 indicates that the corresponding link is not a layout component. Figure 2-4d shows only the south links and Figure 2-4e is the corresponding matrix representation. A value of 1 for a matrix element similarly indicates that the corresponding link is a component of the network and a value of 0 indicates that the link is not a network component.

The representation scheme includes the cells in a matrix similar to the nodes and links. Initially all elements in the cells matrix are equal to 0. Figure 2-5 shows the four matrices, nodes, east links, south links and cells, for the layout in Figure 2-4a.

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

nodes east links south links cells

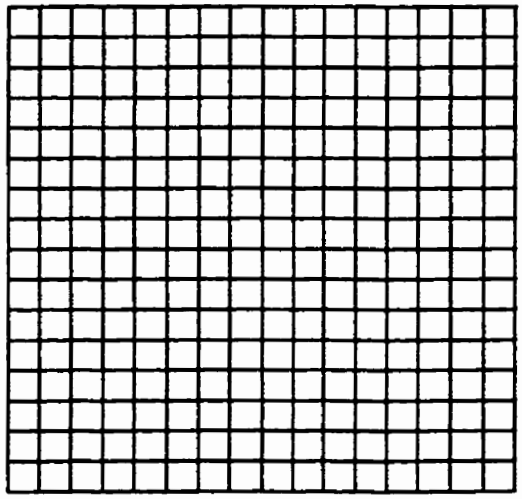
Figure 2-5: Four matrices representing problem and solution

2.3.2. The Two-Torus Representation Scheme and Cursors

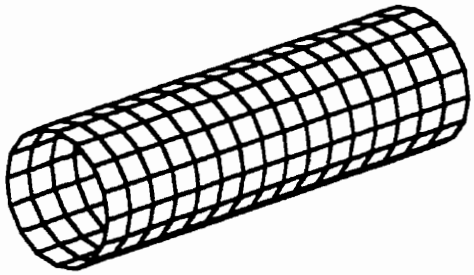
The feasibility testing algorithm makes use of a cursor that points to the grid element on which the algorithm is currently operating. The cursor is a variable that records the location (row and column) of the elements of the grid, namely, the node, east link, south link or cell.

Each of the four matrices operates as a two-torus. Figure 2-6 illustrates a grid plane that is transformed to a two torus. On a two-torus the eastern edge "wraps around" to the western edge and the northern and southern edges similarly connect together. A cursor moving east from the east column remains in the same row but reappears on the other side of the grid in the west column. Conversely, moving west from the west column will position the cursor in the east column. The same is true in the north-south directions. A cursor moving north off the northern edge of the grid will reappear at the southern edge of the grid and vice versa.

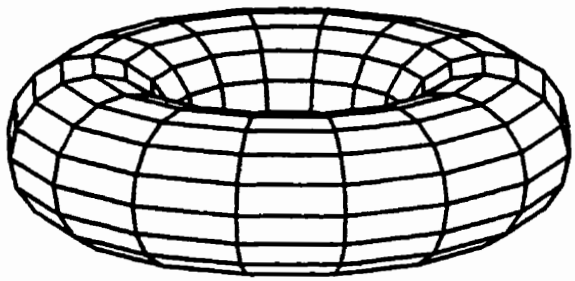
The transformation of the Hanan grid to a two-torus is conceptual only. Physically the Hanan grid still remains within a flat plane. The wrap-around properties of the two-torus and the dummy links in the matrices for the east and south links play an important role in testing the feasibility of solutions. The next chapter describes the algorithm for testing feasibility.



(a) Grid



(b) Grid wraps around to form tube



(c) Tube wraps around to form two-torus

Figure 2-6: Development of a two-torus from a flat grid

3. The Feasibility Testing Algorithm

A network layout is feasible if it connects all basic vertices and the disconnection of one or more basic vertices requires the removal of at least two links. A layout is not feasible if the removal, or failure, of a single link can disconnect basic vertices. To be feasible all the components of a layout that connect the basic vertices together must consist entirely of loops. Any branched components that connect basic vertices or interconnect looped components do not meet the feasibility criterion. The algorithm that tests the feasibility of layouts is a two stage process. The first stage determines if all basic vertices are part of a loop. The second stage determines if all the loops form a contiguous network.

Any loop forms a closed polygon that encloses an area. Natural loops uniquely enclose a single contiguous area. The first stage of the feasibility testing algorithm identifies the natural loops. The stage consists of two parts. The first part marks the space enclosed by each polygon with a unique identifier. The procedure is similar to the "flood-fill" routines used by computer drawing programs to fill polygons with color. The elements of the cells matrix store the identifiers that mark each enclosed area.

Once the first part of stage one has identified all the polygon areas the second part finds each natural loop in the network. A natural loop consists of the nodes and links that form the boundaries of the enclosed areas, and only those nodes and links. After identifying the boundary nodes and links, the algorithm separates all the nodes and links in the network into two sets:

1. Branched components - nodes and links that are not part of a natural loop;

2. Looped components - nodes and links that are part of a natural loop.

The flood-fill algorithm comprises the first part of the first stage of the feasibility testing algorithm and the loop marking algorithm forms the second part. The next two sections explain these two algorithms.

3.1. The Flood-Fill Algorithm

3.1.1. The Area Outside the Natural Loops

The procedure for identifying polygon areas begins by identifying the cells that are not within any natural loop. Initially all elements of the cells matrix have a value equal to 0. The algorithm assigns a value of 1 to all cell elements that are not within any of the natural loops of the network. Two modifications made to the representation scheme for the Hanan grid guarantee that the cells outside the natural loops comprise one contiguous region. The two modifications, explained in the previous chapter, are (1) the absence of south links in the south row and the absence of east links in the east column and (2) the "wrap-around" properties of the two-torus.

The absence of south links in the south row guarantees that natural loops do not enclose any of the cells in the south row. Similarly, the absence of east links in the east column guarantees that natural loops do not enclose any of the cells in the east column. The east column exists on the west side of the west column due to the two-torus wrap around properties of the grid. Similarly, the south row exists on the north side of the north row. The effect of wrap around is that the east column and south row surround the Hanan grid with a single contiguous region of cells. Natural loops do not enclose any of the cells in the surrounding region. Any other cells not enclosed in natural loops are

necessarily contiguous with the surrounding region. A group of cells can be separate from the surrounding region only if links enclose that group of cells on all sides. Links that enclose a group of cells on all sides create a natural loop. Therefore, the region outside all of the natural loops is a single contiguous region.

The cells in the east column and the south row are part of the region outside the natural loops, as explained previously. The flood-fill algorithm identifies the area outside the natural loops if the algorithm begins at cells in either the east column or the south row. The algorithm uses the cell in the northeast corner as the starting cell.

3.1.2. Identifying Contiguous Regions

In addition to the cursor described previously, the flood-fill routine requires a stack to record cursor positions. Any time the algorithm must return to a position to complete a task the cursor location is "pushed" on the stack. The algorithm "pops" cursor positions from the stack in the reverse order in which they are pushed on the stack.

The process of identifying a region of cells enclosed in a natural loop begins by selecting a single cell from the region and marking it with an identifier value. The identified region "spreads out" by marking any adjacent contiguous unmarked cells found on any of the four sides of the cell. There is potential for very inefficient use of recursion because the expansion of the identified region proceeds in four directions. The algorithm can only explore one direction at a time while others are placed on the stack to return to later. The danger is that the algorithm will return repeatedly to cells that are already marked. Appendix C (Section C.1) presents details of an algorithm that makes use of the concept of "closed rows" as a means of limiting recursion.

3.1.3. Marking All Areas with the Flood-Fill Algorithm

The feasibility testing algorithm performs iterations of the flood-fill algorithm then iterations of the loop marking algorithm. The first iteration the flood-fill algorithm marks all the cells that are not within the boundaries of any natural loop. The starting point for the first iteration is the cell at the northern end of the east column. Throughout the iteration the area identifier value is equal to 1. The iterations of the flood-fill algorithm that follow are different. They begin by scanning the Hanan grid in row column order for any cells equal to zero. The location of the first cell equal to zero serves as the starting point for that iteration. Before the execution of each iteration the algorithm increases the area identifier value by 1. The increase guarantees that each contiguous region of cells has a unique identifier value. This process numbers each of the regions enclosed within natural loops sequentially starting at 2. The iterations of the flood-fill algorithm cease when the scan fails to find any remaining cells equal to zero.

3.2. The Loop Marking Algorithm

3.2.1. Rules and Conventions of the Loop Marking Algorithm

The loop marking algorithm is a procedure for finding links and nodes that are components of natural loops. The procedure consists of moving a cursor in a clockwise direction around the boundaries of polygons. A "polygon" in this context refers to a group of contiguous cells set to a common identifier value. The cells in the grid identify polygons upon the completion of the iterations of the flood-fill algorithm described in the previous section. The boundaries of the polygons are the natural loops the feasibility testing algorithm must find. A link that exists at the edge between two adjacent cells from different polygon areas is the link found in the gap between the two cells. However, the

feasibility testing algorithm must find the nodes that make up the natural loops in addition to the boundary links. The loop marking algorithm uses a convention to include a node with every link. The convention guarantees that the algorithm finds all nodes in a natural loop if it finds all the links that comprise the loop. The convention follows the clockwise movement of the cursor around the edge of the polygon. The convention associates a link to the node on the counter-clockwise side of the link.

Figure 3-1 illustrates the association between links and nodes for a polygon that consists of only one cell. The four links and four nodes that encircle the cell define the boundary of the polygon as illustrated in Figure 3-1a. The clockwise arrow in Figure 3-1a illustrates the direction of motion of the cursor. Figure 3-1b through Figure 3-1e illustrate that the node associated with each link exists on the counter-clockwise side of the link.

(a) One cell polygon (b) North boundary (c) East boundary (d) South boundary (e) West boundary

Figure 3-1: Node and link association

The cursor may point in one of four directions, (i.e., north, south, east or west). The action of the cursor depends on its direction and the content of neighboring cells. Four rules define the behavior of the cursor. Which of the four rules that apply depends on the cursor's direction.

Appendix C (Section C.4) presents a detailed description of the four rules that control the movement of the cursor. The procedure described here is a generalization of the four rules of the loop marking algorithm. In each of the rules the cursor moves one position in the forward direction. The algorithm tests the cell at the new position to determine if its identifier value is different from the cell at the previous cursor location. A change in identifier value indicates that the cursor has crossed a boundary. The cursor rotates 90° counter-clockwise if it did not cross a boundary. If the cursor crossed a boundary the algorithm marks the node and link combination that exists at the boundary. The node and link combination will be one of the four illustrated in Figures 3-1b to 3-1e. The cursor returns to its previous position and rotates 90° clockwise after marking the boundary.

The procedure used to mark a node differs from the procedure used to mark a link. The node's value changes from 1 to -2 if the marked node is a basic vertex. The value changes from 0 to -1 if the marked node is a Steiner node. The marked links consist only of links that are part of a layout and therefore equal 1 before being marked. Marking changes the link's value from 1 to -1. The algorithm must distinguish between links on the boundary of one polygon exclusively and links at the boundary between two polygons. A value of -1 indicates that the link is part of the boundary of a polygon. A link's value changes to -2 if the algorithm encounters a link already equal to -1 when marking the boundary of a polygon. A value of -2 indicates that the link is part of the boundary between two polygons.

For the loop marking algorithm to work properly the algorithm must place the cursor initially within the polygon at the boundary. It must point the cursor toward the boundary or parallel to the boundary in the clockwise direction. In this context "the

clockwise direction" means clockwise around the boundary of the polygon as in Figure 3-1a. The algorithm records the location and orientation of the cursor. Then the cursor moves about the edge of the polygon to mark the boundary using the procedures defined by the four rules. The algorithm has marked the boundary of the polygon completely when the cursor returns to the recorded position and orientation.

3.2.2. Example of the Loop Marking Algorithm

Figure 3-2a shows a portion of a grid in which two identifier values mark polygon areas. The hatched cells represent one value and the clear cells represent another value. A black triangle represents the cursor. Initially the cursor is in a clear cell. The cursor points parallel to the boundary between the two cell types in the clockwise direction. The term "clockwise direction" refers to clockwise from within the polygon created by the clear cells. The cursor moves one position in the forward direction, east. The algorithm does not mark a boundary because the cell identifier value does not change with this move. The cursor rotates 90° counter-clockwise whenever the identifier value does not change. Figure 3-2b illustrates that now the cursor points north.

Figure 3-2c shows that the cursor moves north one position. The cell identifier value changes indicating that the cursor has moved out of the polygon of clear cells. The cursor returns to its previous position whenever the identifier value changes. The algorithm marks the node and link that comprise the boundary between the two cells of different value and the cursor rotates 90° clockwise. Figure 3-2d illustrates the results of these actions.

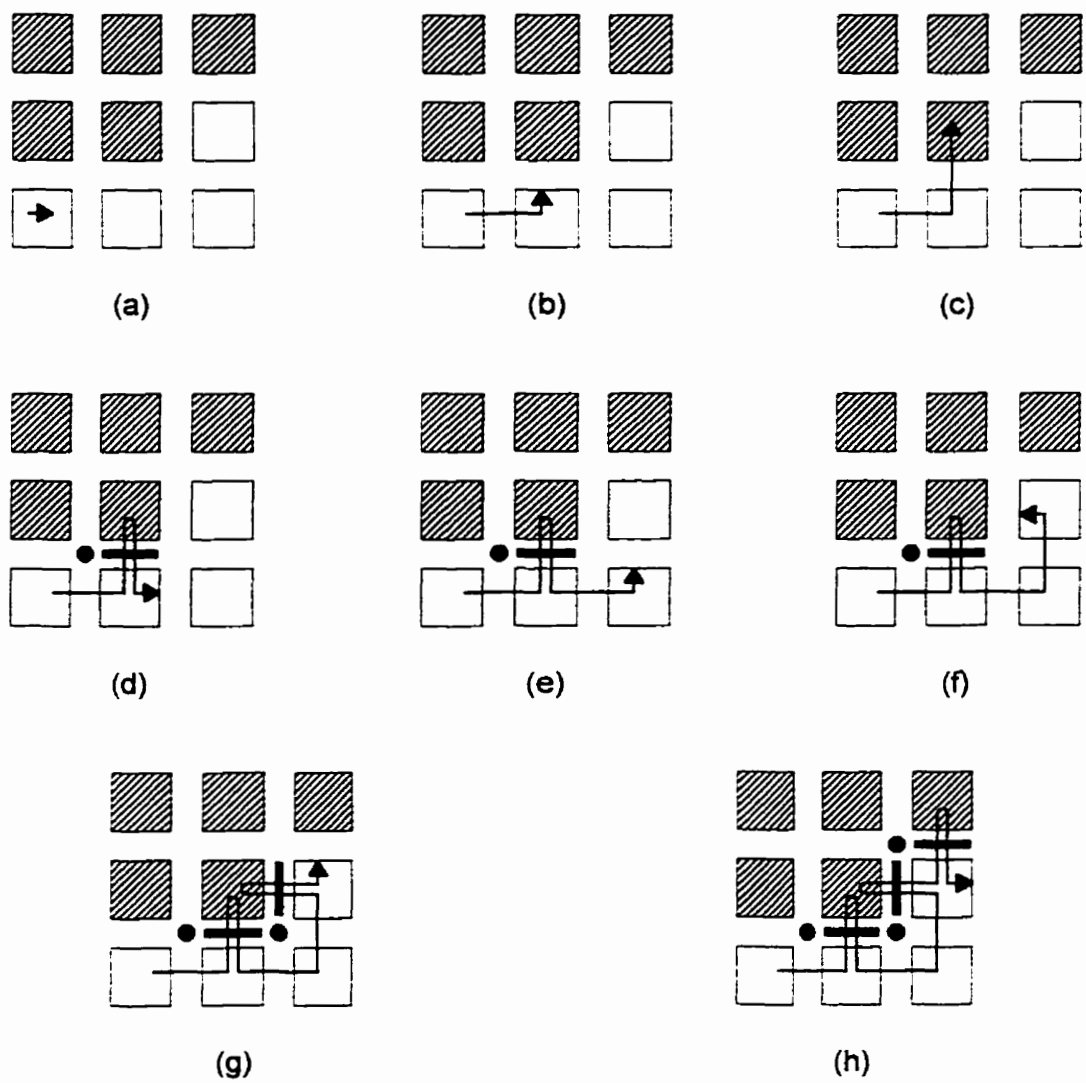


Figure 3-2: Example of the loop marking algorithm

The algorithm applies rules iteratively through Figure 3-2e to Figure 3-2h. Figure 3-2 depicts only a small subset of a sample Hanan grid. In an actual test the algorithm applies the rules as described using the entire grid. The cursor traverses the entire boundary of the polygon and returns to the starting position and orientation in Figure 3-2a. At this point the algorithm has marked all the nodes and links of the boundary of the polygon. The marked nodes and links are components of one natural loop.

3.2.3. Application of the Loop Marking Algorithm

Initially the loop marking algorithm requires that the cursor be at a polygon boundary in an orientation pointing either at the boundary or parallel to the boundary in the counter-clockwise direction. If the cursor is not at this starting position it can become trapped and cycle endlessly in the interior of the polygon. To find the starting position the algorithm scans the cells matrix in row-column order until it locates the first element equal to the identifier for the current polygon area. The cell is necessarily in the most northerly row of the polygon because of the scanning order. The algorithm places the cursor in this cell oriented in the east direction. The cursor is now in the proper location and orientation to mark the boundary of the polygon.

The loop marking algorithm marks polygons beginning with the identifier 2 and increments the identifier value with each iteration until it has marked all polygon boundaries. The iterations start at the identifier value 2 because the identifier value 1 identifies the area outside the polygons. Iterations terminate when a scan of the cells matrix reveals the identifier value has increased to a value for which there are no grid cells. Once iterations of the loop marking algorithm terminate the nodes and links have values according to Table 3-1.

Table 3-1: Values for nodes and links produced by the loop marking algorithm

Values for nodes:

-2	=	marked basic vertex (basic vertex is part of a natural loop)
-1	=	marked Steiner node (Steiner node is part of a natural loop)
0	=	unmarked Steiner node (Steiner node is not part of a natural loop)
1	=	unmarked basic vertex (Basic vertex is not part of a natural loop)

Values for links:

-2	=	link marked twice (link is common to two natural loops)
-1	=	link marked once only (link is part of one natural loop)
0	=	link was not selected as part of the network
1	=	link is selected but not part of a natural loop (the link is a branched component)

3.3. The Feasibility Tests

The algorithm performs two tests once the iterations of the flood-fill and loop marking algorithms terminate. The first test establishes if all the basic vertices are components of natural loops. The second test establishes if all the natural loops together produce a single contiguous network. The network layout is feasible only if all the basic vertices are part of a natural loop and the natural loops produce a single contiguous solution.

3.3.1. Connectivity of the Basic Vertices

The feasibility testing algorithm scans the nodes matrix for any elements equal to 1 to test the connectivity of the basic vertices. The feasibility testing algorithm undertakes the test on completion of all iterations of the flood-fill and loop marking algorithms. Any elements equal to 1 in the nodes matrix indicate that a basic vertex exists that connects to the network by branched components or is not part of the network at all. In either case the solution is infeasible.

3.3.2. Contiguity of the Natural Loops

If the connectivity test passes the solution is not necessarily feasible. The network must be a single contiguous solution to be feasible. The feasibility testing algorithm modifies the Hanan grid and applies a single iteration of the flood-fill algorithm to establish if the solution is contiguous.

The modification of the grid changes the values of some of the links and some of the cells. The modification changes the value of the links to 0 except those marked once only. The algorithm maintains the links marked only once at their current value of -1. The modification changes all the cells in the grid to 0 except those that are in the area outside the natural loops (i.e., cells set equal to 1). The Hanan grid retains only those links that form the boundary between the network and the area outside the natural loops after the modification. The only cells that remain marked are cells currently in the area outside the natural loops. If the network is not contiguous the cells now set to 0 produce polygons in physically separate regions.

The feasibility testing algorithm performs one iteration of the flood-fill algorithm with the identifier set to 2. If the network forms one contiguous region the single iteration

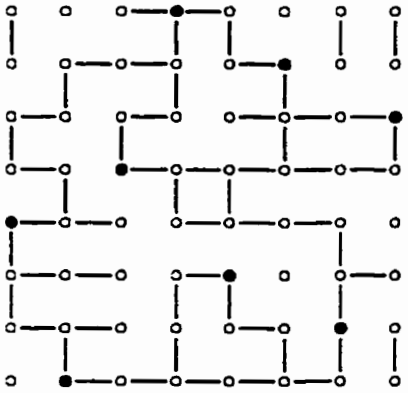
of the flood-fill algorithm changes all the cell values that currently equal 0 to the current identifier value 2. The algorithm scans the cells matrix and if any cells remain equal to 0 the network layout is infeasible.

3.3.3. Example of the Feasibility Testing Algorithm

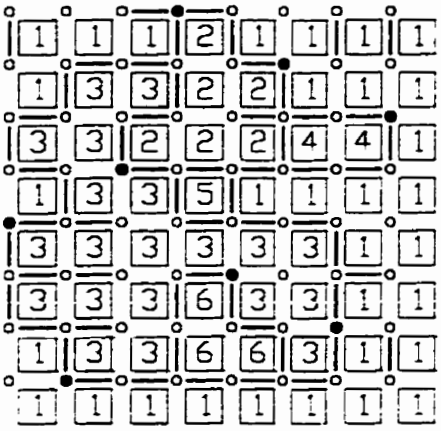
This section demonstrates the algorithm using a feasible example. Figure 3-3a shows the layout of the example network. The basic vertices are solid black circles. Potential Steiner nodes are white circles. Solid lines represent selected links. Figure 3-3a does not show the links of the grid that are not part of the layout solution.

The feasibility testing algorithm performs iterations of the flood-fill algorithm until it has marked all the cells in the grid. Six iterations of the flood-fill algorithm produce the grid in Figure 3-3b. Figure 3-3b shows the area identifier values of each of the cells. The feasibility testing algorithm performs an iteration of the loop marking algorithm for each area with an identifier value of 2 or greater. The iterations of the loop marking algorithm produce the grid in Figure 3-3c. Figure 3-3c shows each marked node with an additional circle drawn around it. Links that the algorithm has marked once are solid black bars. Links the algorithm has marked twice are white bars. Unmarked links appear as dotted lines.

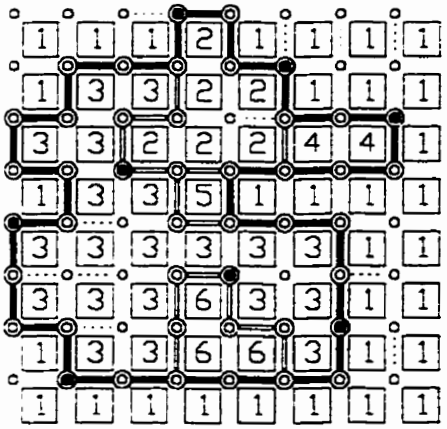
The algorithm scans the grid in Figure 3-3c for any unmarked basic vertices. Solid black circles that do not have an outer circle represent unmarked basic vertices. The layout is infeasible and the algorithm terminates if it finds any unmarked basic vertices. No unmarked basic vertices exist in the example in Figure 3-3c.



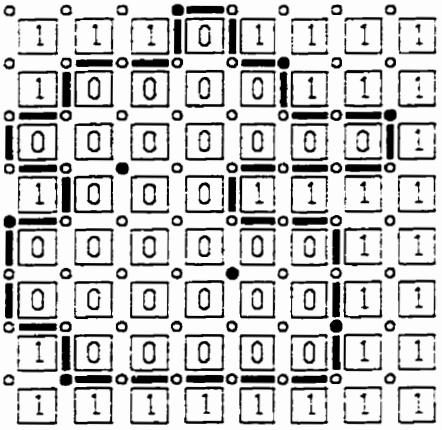
(a) Network layout



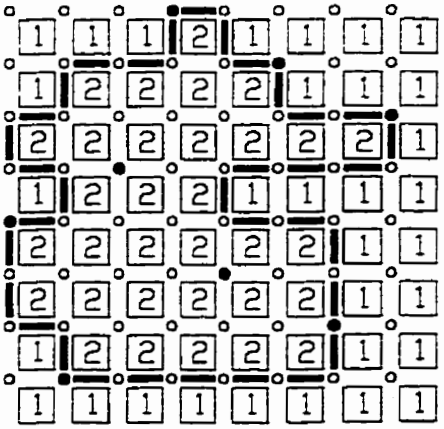
(b) Marked areas



(c) Marked loops



(d) Before contiguity test

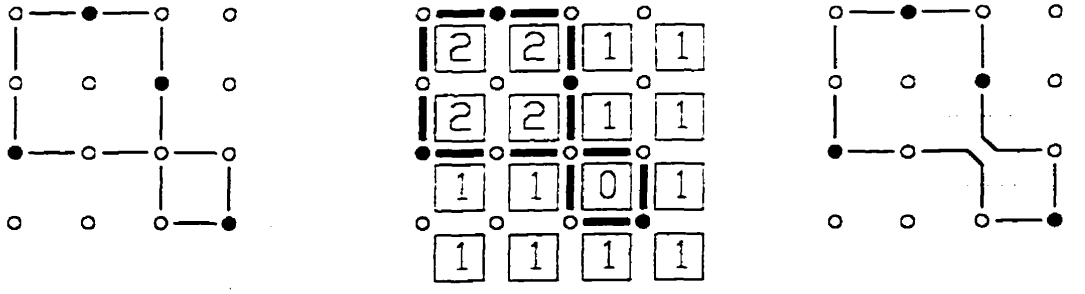


(e) After contiguity test

Figure 3-3: Example of the feasibility testing algorithm

The contiguity test retains the values of the links that the algorithm marked once only. The values for all other links change to 0. The change removes all the links shown as dotted lines or white bars. All cells except those set to 1 change to zero as well. Figure 3-3d shows the resulting grid. The flood-fill algorithm executes one final time filling only one polygon area. Figure 3-3e shows the grid that results from the single iteration of the flood-fill algorithm. The network is not contiguous and is therefore infeasible if any cells remain equal zero. The layout in Figure 3-3 is feasible because no cells that equal zero remain.

3.4. The Problem with Intersections



(a) Layout with intersection

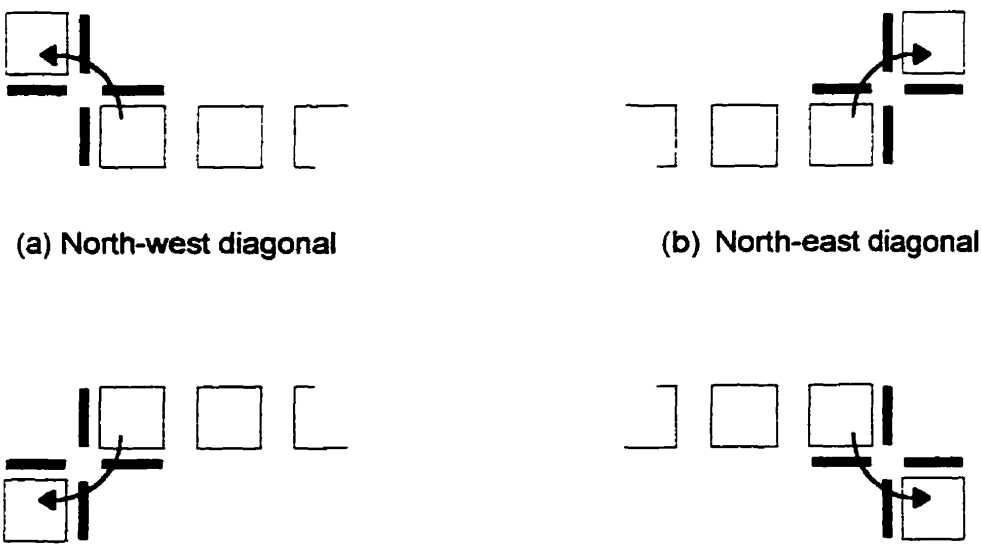
(b) Results of contiguity test

(c) "Back to back" elbows

Figure 3-4: Problem created by intersections

The network shown in Figure 3-4a is an example of a type of network that poses a problem for the feasibility testing algorithm. Figure 3-4b shows the example layout after the last application of the flood-fill routine similar to Figure 3-3e. The flood-fill routine leaves some cells set to 0 indicating that the layout is infeasible. The layout is indeed infeasible as the algorithm indicates because isolation of nodes occurs if a failure occurs

precisely at the intersection node (i.e., the node with four adjacent links). However, the layout is feasible if the intersection actually consists of two elbows "back-to-back" as indicated in Figure 3-4c. The thesis regards intersections of this type as two elbows. The flood-fill algorithm requires a slight modification to correctly identify layouts of this type as feasible.



(a) North-west diagonal

(b) North-east diagonal

(c) South-west diagonal

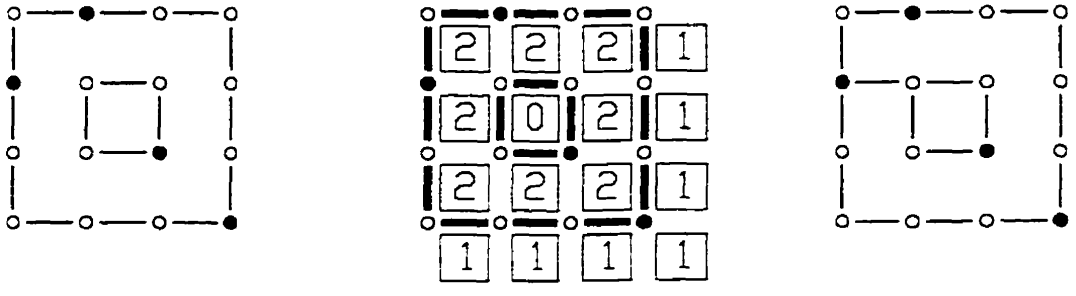
(d) South-east diagonal

Figure 3-5: Diagonal expansion of polygon areas

The flood-fill algorithm that the contiguity test uses is slightly different from the algorithm used to mark polygon areas. The modified algorithm has a different row marking procedure. (Refer to Appendix C, Section C.2, for details on the row marking algorithm.) The unmodified row marking procedure does not permit the polygon to expand in diagonal directions. Diagonal expansion is possible in the direction indicated by the arrows in Figure 3-5a or Figure 3-5c at the extreme west cell if four links exist in

the pattern shown in the figures. For the extreme east cell in a row the diagonal expansion is possible in the direction shown in Figure 3-5b and Figure 3-5d. Diagonal expansion consists of checking the cell indicated by the arrows in Figure 3-5a to Figure 3-5d to determine if the row closing procedure applies to that cell. (Refer to Appendix C, Section C.1, for details of the row closing algorithm.) The algorithm applies the row closing procedure if cell at the diagonal position is unmarked or not closed. The feasibility testing algorithm with this modification correctly identifies layouts of the type shown in Figure 3-4a as feasible layouts.

3.5. The Problem of Loops within Loops



(a) Loop in loop

(b) Feasibility test of loop in loop

(c) Loop in loop with branch

Figure 3-6: Loops within loops

Figure 3-6a shows a network in which a loop encloses another loop. The feasibility testing algorithm correctly identifies this type of network as infeasible without further modification. The loop marking algorithm proceeds in a counter-clockwise direction around the outer boundary only. It will not mark any inner boundary created by a loop contained within a polygon as in Figure 3-6a. The algorithm marks the nodes and links that comprise the inner loop once only even though the inner loop is the boundary

between two polygon areas. Figure 3-6b shows the layout in Figure 3-6a after the final application of the flood-fill algorithm. The area enclosed by the loop inside a polygon remains unmarked indicating correctly that the layout is infeasible. The algorithm correctly identifies the layout as infeasible even with the modifications that enable diagonal expansion. The results are exactly the same if a loop inside a polygon connects to the rest of the network by branched components as in Figure 3-6c. The feasibility testing algorithm correctly identifies the layout as infeasible.

3.6 Improving Layouts

The algorithms used to test feasibility of networks can improve layouts as well. None of the basic vertices of a feasible solution connect to the network through branched components. The improvement algorithm can remove all the branched components in a feasible solution because these components serve no function. The branched components are all those selected links that the loop marking algorithm does not mark. In the improvement procedure the flood-fill algorithm identifies all polygon areas. Then the loop marking algorithm marks the edges of all the polygons. The improvement algorithm removes any remaining links equal to 1, which does not change the feasibility of the solution.

4. Generating Feasible Solutions

The learning program requires a method that generates layout solutions randomly without favoring any region of the solution space or systematically eliminating any part of the solution space. The learning process acquires incomplete knowledge of the solution space if the random generation algorithm systematically eliminates any part of that space.

4.1. "Coin Toss" Random Generation

One simple method for "unbiased" random generation of solutions is randomly setting the values of the east and south links matrices to 0 or 1. (The method must set values in the east row of the east links matrix and the south row of the south links matrix to 0.) The difficulty with simple "coin toss" random selection of links is that infeasible solutions dominate the solution space to a large extent. The method very rarely produces feasible solutions. Davidson and Goulter (1994b) generated solutions using a small network consisting of six nodes. The method randomly assigned values of 0 or 1 to elements of the links matrices. Only 16 of 40,200 solutions were feasible.

4.2. Preference and Threshold

The approach to overcome the problem of infeasibility models any potential solution as a "preference scheme". The preference scheme neither strictly selects nor discards each link in the Hanan grid. All links obtain a "degree of preference" that takes a value between 0 and 1 exclusively. The method generates the preference values randomly.

Each solution requires a single threshold variable. The threshold variable takes a value between 0 and 1 inclusively. The method includes all links with preference values greater than the value of the threshold variable. All links with preference values lower than the threshold value are not part of the layout.

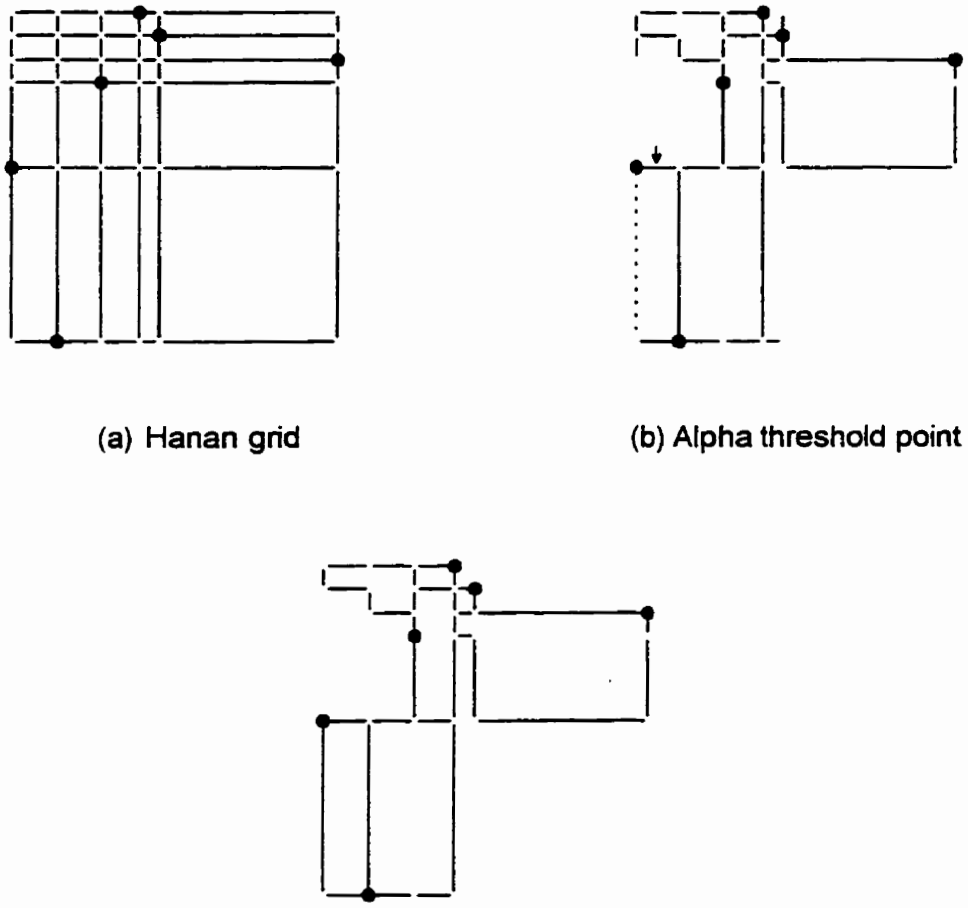
A range of threshold values that produces feasible solutions exists for any preference scheme. All the links in the Hanan grid are part of the layout when the threshold value equals 0 because all links must have preference values greater than 0. The layout consisting of all links in the Hanan grid is always feasible, but extremely over designed.

Raising the threshold value from zero excludes links from the network from least preferred to most preferred. Eventually the exclusion of a link produces an infeasible solution and any further exclusions continue to produce infeasible solutions. The term "alpha value" refers to the highest threshold value that produces a feasible solution. An alpha value exists for any preference scheme.

4.3. Example Solution Generated by Preference and Threshold

Figure 4-1b is a layout derived from the Hanan grid in Figure 4-1a using the preference and threshold process. The procedure assigns random generated preference values to the links in the Hanan grid. Raising the threshold value from 0 removes links with the lowest preference values first. The process removes 23 links from the Hanan grid in the order that results from the preference values. The feasibility testing algorithm tests the feasibility of the layout each time the increasing threshold value removes a link. The layout is not feasible if node isolation occurs when any single

link fails. Figure 4-1b shows the layout produced to this point. The layout does not violate the feasibility constraint.



(a) Hanan grid

(b) Alpha threshold point

(c) Branched components removed

Figure 4-1: Setting the threshold value and improving solutions

The next link to be removed according to the preference scheme is the link shown with the dotted line in Figure 4-1b. The removal of this link represents the first infeasible solution. Node isolation occurs if the link indicated by the arrow in Figure 4-1b

fails when the link shown by the dotted line is not part of the network. The layout with the dotted line removed fails the feasibility test. Therefore, the alpha value produces the layout shown in Figure 4-1b including the link shown as a dotted line.

If a network is feasible all branched components serve no function. An improvement algorithm developed in conjunction with the feasibility testing algorithm removes the branched components. Figure 4-1c shows the improved layout.

4.4. The Advantage of the Preference and Threshold Method

Chapter 7 (Section 7.1) compares the "coin toss" and preference and threshold methods. The "coin toss" method of generating solutions produces infeasible solutions at a high rate that grows exponentially with the problem size. The exponential growth in infeasible solutions renders the method ineffective except for very small problems. In contrast, every solution produced by the preference and threshold method is feasible. The graphs and tables in Section 7.1 indicate that a highly significant improvement in computational effort occurs as a result of producing only feasible solutions.

5. An Evolution Program

The representation scheme and algorithms described to this point provide the basis for an evolution program based on genetic algorithms. A simple genetic algorithm (Goldberg, 1989; Walters and Smith, 1995) operates on population of solutions using three operators, namely, selection, crossover and mutation. The greatest difficulty in applying the genetic algorithm to this solution coding scheme is the high probability of generating infeasible solutions. The preference and threshold method solves the problem of generating starting populations that consist only of feasible solutions. However, the populations quickly become completely dominated by infeasible solutions if the evolution program uses the conventional genetic algorithm operators of crossover and mutation.

5.1. A Replacement for the Crossover Operator

In the simple genetic algorithm the crossover operator takes two "parent" solutions chosen by the selection operator and separates each of the two solutions at the same point. The algorithm chooses the separation point randomly. Parts of the two solutions recombine to produce a new solution. The data from one parent solution that occur before the separation point combine with data from the other parent solution that occur after the separation point. The difficulty with crossover when used with solution codings that represent network layouts is that combining sections from different layouts does not generally produce contiguous or feasible networks.

Two approaches exist for tree networks as replacement strategies to crossover that improve the odds of generating feasible networks. Davidson and Goulter (1995)

separate each tree network into a set of paths between each sink node and the source. The method generates a child solution by selecting a path from the source to each of the sink nodes. It selects each path from one of the two parents chosen at random. However, at best only 50% of the layouts this type of recombination produces are feasible. Walters and Lohbeck (1993) use a recombination method that always produces feasible solutions. The method combines all the links from both parents and uses a spanning tree algorithm to select a subset of links at random that form a tree.

This section describes a method for combining looped network solutions. A specialized operator, referred to as "recombination", combines the data from both parents and produces only feasible solutions. The method for combining two parent solutions to form a single feasible child solution averages the preference values of each of the links of the two parent solutions. Then the method sets the threshold variable of the averaged solution to its alpha value.

The recombination operator preserves the order of preference for links in the child solutions when agreement occurs in the preference values of the two parent solutions. Consider, for example, the case in which both parent solutions have a high preference value for a given link. The average of two high values results in a high preference value for that link in the child solution. The high preference value results in a high probability that the link will be part of the resulting layout. Similarly, a link having a low preference value in both parents produces a link with a low preference value in the child solution. A low preference value results in a low probability of including the link in the child solution.

When disagreement between parent solutions occurs the outcome is not as predictable. Consider the case in which one parent assigns a high preference value to a link and the other assigns a low preference. The value assigned to the link in the child solution is in the middle ranges. A link with a preference value in this range could lie on either side of the alpha value. The selection of the link as part of the child layout depends on whether the preference scheme requires the link to produce a feasible solution.

The evolution program includes roulette wheel selection (Goldberg, 1989) to select parent solutions. The program can function without a mutation operator. However, experience with the two operators alone, selection and recombination, shows that populations tend to converge very quickly to a single solution. Rapid convergence results from repeated averaging, which eliminates diversity in the population. Once the population converges to a single solution continued improvement is impossible with only the selection and the recombination operators. The evolution program requires some mechanism analogous to mutation to produce diversity and prevent premature convergence on sub-optimal solutions.

5.2. The Mutation Operator

The procedure that performs mutation must make random changes to layouts that do not produce infeasible solutions. The method selects a small number of links at random from a preference scheme and generates new preference values for those links. The new preference values alter the positions of the links in the preference order. The method sets the threshold variable to the new alpha value after it has "mutated" the selected links. The new network is always feasible. The layout varies slightly from the

selection of links before mutation. Experience with the mutation operator shows that an average mutation rate of 1 link in 100 prevents the premature convergence problem associated with the recombination operator. Mutation maintains the search indefinitely without convergence. Higher rates of mutation are degenerative and lead to performance characteristic of random search.

5.3. The Problem of Diminishing Returns

Problems begin to occur with random processes as solutions improve. The layout solution in Figure 4-1b in the previous chapter is characteristic of randomly generated solutions found in initial populations. These solutions have a high degree of redundancy. Random processes eliminate the redundant components in these solutions easily without producing infeasible solutions. At the earlier stages of a search random changes are likely to produce improved solutions. Much of the redundancy disappears as solutions improve. Now improvements must consist of more complex changes to maintain feasibility. Random alterations of the preference order are less likely to produce complex changes.

Figure 5-1a shows a section of a layout. The improvement that produces layout in Figure 5-1b requires the elimination of one or more of four links and the simultaneous addition of two links. (The improvement does not require the removal of all four links because the removal of branched components eliminates the remaining links.) Improvements of this type must introduce specific links at the same time as they eliminate others.

Complex improvements are not impossible to produce with randomized search methods. However, they are less probable than changes that involve only the removal of

links. As a consequence the performance of the evolution program degrades as solutions improve. Larger numbers of iterations produce smaller improvements. The learning program, described in the next chapter, uses rules to overcome the problem of diminishing returns.

(a) Links before improvement (b) Links after improvement

Figure 5-1: Change of layout requiring addition and removal of links

6. The Learning Program

This chapter describes the learning program. The program creates a rule base for designing the layout geometry of looped water distribution networks. Conceptually, the learning program consists of three components: the agitator, the rule formulator and the production system. The agitator component experiments with solutions to a network problem by making small changes to the solutions at random. The rule formulator derives a rule when a random modification produces an improved solution. The rule performs the same changes as the random modification if the same geometric pattern occurs in a future layout. The production system applies the rules developed by the rule formulator as the program repeatedly experiments with sample problems. The learning program monitors the performance of each rule during the use of the production system. A process called "forgetting" discards rules that behave poorly.

6.1. The Agitator

The learning program requires procedures for the random generation and random manipulation of solutions. Chapter 4 explains the procedure for randomly generating populations of solutions for the evolution program. The process consists of assigning random preference values to all of the links, setting the threshold variable to the alpha value and removing all branched components from the resulting layout. As explained in Chapter 4 the random generation process is "unbiased". The process does not systematically favor or exclude any region of the solution space. An important property of the preference and threshold process is that it generates only feasible solutions from a solution space dominated by infeasible solutions.

Once random generation creates a solution the agitator alters the solution through random modification. The process used by the agitator for random modification is the same process used by the evolution program for mutation explained in Chapter 5. A small number of links chosen at random receive new preference values. The process sets the threshold variable to the new alpha value and removes any branched components from the resulting layout.

The agitator uses random manipulation to alter a solution. The newly altered solution replaces the original unaltered solution if the new solution represents an improvement over the old solution. The agitator repeats the random manipulation process if an improvement does not occur. The process repeats until an improvement occurs or until it reaches the maximum number of iterations of random manipulation. The maximum number limits repetition to prevent the agitator from cycling forever in the case of a global optimum, which it cannot improve. The agitator does not make solutions worse. It either produces improvement or fails to produce a change. Considered in isolation of the other components of the learning system, the agitator alone forms a simple type of evolution program operating with a population of one solution. This type of program is similar to the (1+1)-ES evolution strategy (Rechenberg, 1973, Schwefel, 1977)

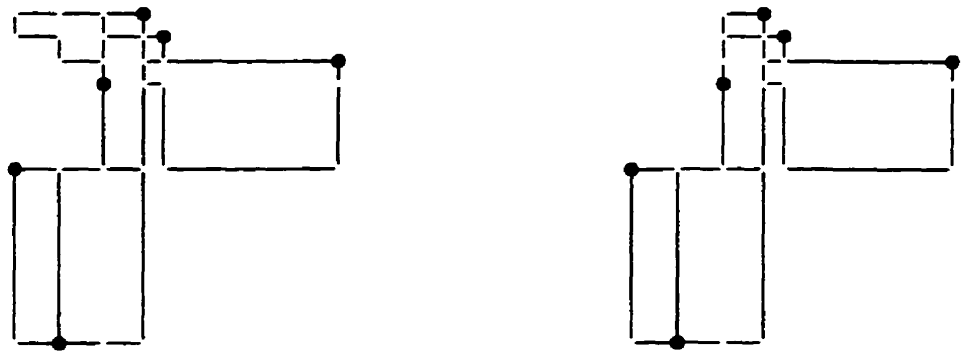
Experiments with the agitator show that it is subject to the problem of diminishing returns. The agitator generally requires more iterations as solutions improve. As the program generates solutions it displays the layouts graphically. The changes required to produce improvements in highly evolved solutions are often obvious to someone observing the progress of the program. These same obvious changes are usually complex and very difficult for random manipulation to produce.

Previous work with rule based systems (Davidson and Goulter, 1991a) demonstrated that rules can perform complex types of manipulations similar to those required during the later stages of the improvement process. The learning system formulates and uses rules to perform simple or complex manipulations.

6.2. The Rule Formulator

The agitator makes small changes to a single layout solution. When the changes to a solution produced by the agitator result in an improvement the learning program derives rules by examining the changes. The rule formulator compares the layout before the application of the agitator with the improved layout. The program records the pattern of links in the Hanan grid that change as well as the links immediately adjacent to the change.

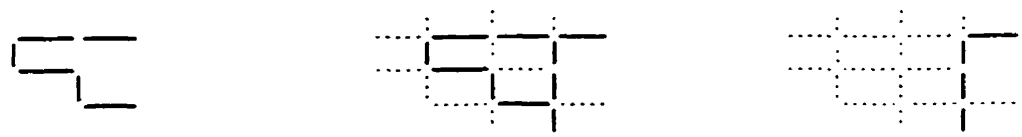
The layout in Figure 6-1b represents an improvement over the layout in Figure 6-1a. A comparison of the two layouts yields a subset of the Hanan grid consisting of only the altered links. Figure 6-2a shows the extracted subset. Each grid link immediately adjacent to the altered links augments the pattern in Figure 6-2a. Figure 6-2b shows the augmented pattern with the additional information. In Figure 6-2b a solid line indicates that the link in the pattern is part of the layout and a dotted line indicates that the link is not a selected link. The data in Figure 6-2b represent the pattern of links in the layout before improvement (Figure 6-1a) and creates the antecedent clause of the rule. The pattern in Figure 6-2c creates the consequent clause of the rule. The consequent clause produces the set of links in the improved solution (Figure 6-1b).



(a) Layout before the agitator (b) Layout after the agitator

Figure 6-1: Improvement produced by the agitator

The rule base records the patterns in Figure 6-2b and 6-2c. The production system uses the rule to alter a layout whenever the pattern in the antecedent clause occurs again. Use of the rule transforms the pattern of links matching the antecedent clause, Figure 6-2b, to the pattern in the consequent clause, Figure 6-2c. The rules do not include the length of the links. Rules require only the presence or absence of links to match for the rules to apply. In addition, the production system offsets the pattern in the antecedent clause to test every possible position in the Hanan grid.



(a) Changed links (b) Antecedent clause (c) Consequent clause

Figure 6-2: Antecedent and consequent clauses of a rule

6.3. The Production System

The third component of the learning system is the production system. The control strategy of the production system is an irrevocable best-first search described in more detail in Appendix A. The term "irrevocable" does not apply in the strictest sense. The strategy revokes a rule if the rule produces an infeasible solution. The control strategy does not revoke any rule that produces a feasible solution.

The program tests the resulting solution each time the production system applies a rule. The program records three parameters that measure the effectiveness of the rule:

1. The number of times the rule applies successfully;
2. The number of times the rule produces an increase in length; and
3. The number of times the rule produces an infeasible solution.

On the basis of the recorded parameters rules fall into four categories:

1. Rules that can produce infeasible solutions;
2. Rules that do not produce infeasible solutions but can produce an increase in total length;
3. Rules the production system uses infrequently and are not in category 1 or 2; and
4. Preferred rules.

According to their recorded history of use preferred rules are rules that apply frequently and never produce increased total length or infeasible solutions. Rule bases created by the procedure grow in size very rapidly. The ability of the production system to produce better solutions improves as the rule base increases in size. However, the

time required to produce improved solutions also increases. To improve the speed of the production system the algorithm halts periodically and the rule base removes all rules not performing well. The process, referred to as "forgetting", eliminates all categories of rules except preferred rules.

6.4. The Algorithm

The algorithm for generation of rules based on autonomous experimentation is as follows:

1. Generate a starting solution at random.
2. Apply all the rules in the rule base using the production system to improve the solution as much as current set of rules will permit.
3. Apply the agitator to the results of step 2 until an improvement occurs
4. Derive a new rule by comparing the results of step 2 with those of step 3.

Figure 6-3 illustrates the structure of the algorithm. The object of the algorithm is not simply to produce the optimal solution to a single instance of the problem as was the case with the evolution program of the previous chapter. This algorithm produces a rule based program and that rule based program performs the optimization of layout geometry.

6.5. Cheating

Rules overcome the problem of diminishing returns associated with random manipulation. Optimization no longer relies on the probability of successful random

manipulation, but rather, relies on the application of the appropriate rule. However, the process of acquiring rules itself requires random manipulation and is therefore subject to diminishing returns. An improved version of the algorithm reduces the problem of diminishing returns associated with learning. The improved algorithm repeatedly solves the same problem with slightly different starting solutions. The algorithm stores the best solution to the problem encountered to the present time. The agitator uses the stored "best" solution when it fails to produce an improved solution through random manipulation.

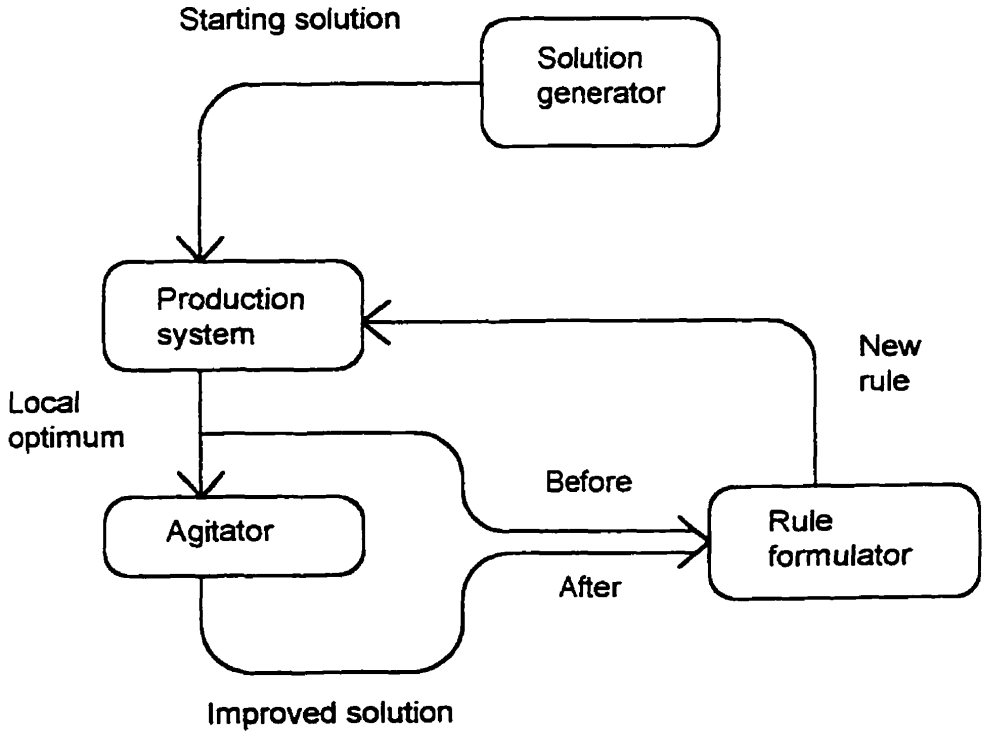


Figure 6-3: Structure of the learning program

The learning program uses the agitator after the production system reaches a local optimum. The agitator attempts to find a better solution than the local optimum through iterations of random manipulation. The agitator limits the random manipulations to a maximum number to prevent endless search in the case of a global optimum. Specifying a maximum number of iterations also prevents a very long search in the case of a solution requiring complex manipulations.

The algorithm generates a new rule by the method described previously if the agitator finds an improved solution before it reaches the maximum number of iterations. However, the agitator may reach the maximum number of iterations with no improvement. In this case the improved learning program compares the local optimum produced by the rule base with the "best" solution found to this time. The program creates a new rule from the comparison of the two solutions if the "best" solution represents an improvement over the solution generated by the rule base.

This process, referred to as "cheating", is capable of generating complex rules. The rule base often produces local optima for which improvement requires the simultaneous addition and removal of links. Random manipulation has a low probability of producing complex improvements of this type. Comparing the solution with a previously generated solution is a more efficient means of "improvement" than generating the required improvement through random manipulation. The probability of success of the cheating method is independent of the complexity of changes required to produce an improvement. Therefore, there is no problem of diminishing returns associated with the cheating method. The cheating method always executes in one iteration only.

The learning program does not necessarily produce new rules with each iteration if the algorithm is solving the same problem repeatedly from the same starting solution. The agitator cannot improve the solution produced by the rule base once the rule base has acquired sufficient rules to establish a path from the starting solution to the global optimum. The ideal approach is not to start each iteration from completely different starting solutions either. The local optimum produced by the rule base may not closely match the stored "best" solution if both solutions originate from very different starting solutions. A poor match is most probable in the case of a problem with many equivalent global optima. If the cheating process compares solutions that are very dissimilar, the resulting rules are usually large and very complex. Rules with large antecedent clauses have a low probability of matching patterns in the Hanan grid. To derive rules that the production system uses frequently the "best" solution should be similar to the solutions created by the rule base.

These arguments suggest that the algorithm should solve the same problem many times from slightly different starting solutions. To accomplish this the algorithm generates a starting solution and records it. At the start of each iteration the algorithm modifies the recorded starting solution slightly by using the agitator before the application of the rule base. Use of the agitator produces a starting solution that is only slightly different in each iteration. Figure 6-4 represents the modified algorithm in schematic form.

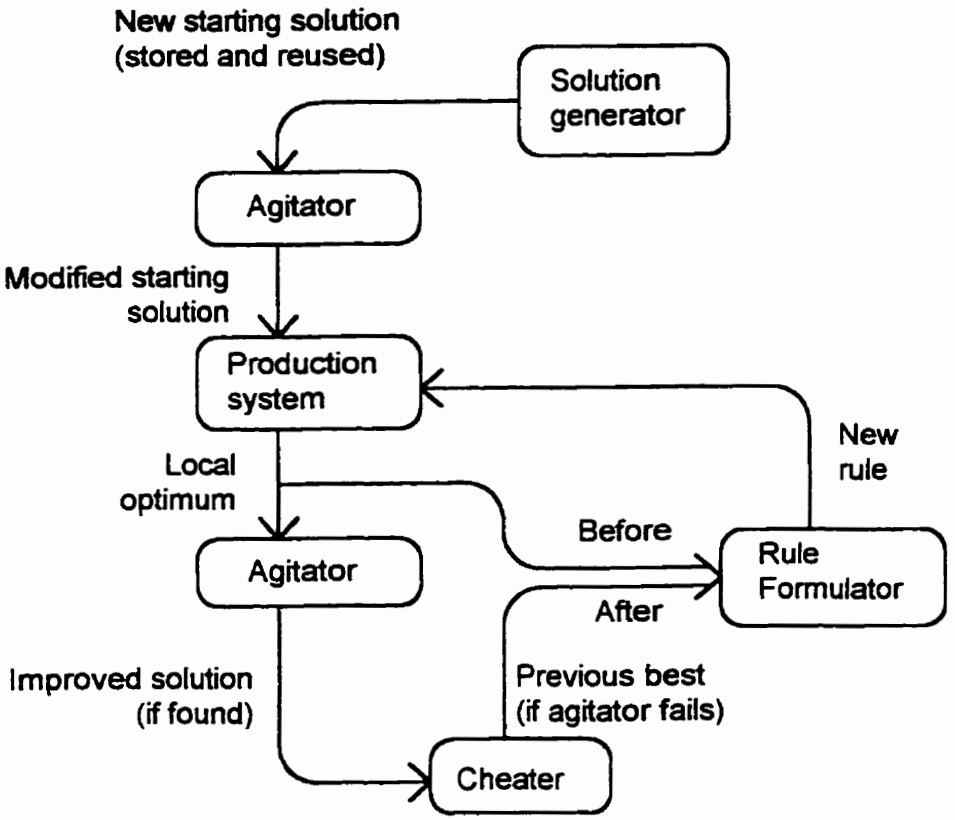


Figure 6-4: Structure of improved learning program

7. Experimental Trials

7.1. "Coin Toss" Method versus Preference and Threshold

7.1.1. Sample Problem Files

Both the "coin toss" and the preference and threshold methods can generate the random starting solutions required by the evolution program and the learning program. This section compares the two techniques on the basis of the computational effort required to generate feasible solutions. The comparison uses sample input files, which Appendix D lists along with the program source code. The sample input files represent problems of four different sizes each consisting of either four, six, eight or ten basic vertices. The entire set consists of a total of twelve files with three sample problem files included for each of the four selected problem sizes.

7.1.2. The Computational Effort of the "Coin Toss" Method

The method for generation of random numbers uses seed values to initiate the random generation process. A user can repeat any experiment and obtain the same results if he or she uses the same seed value. The experiments that compare the "coin toss" and preference and threshold method use nine seed values. The first part of the comparison involves the "coin toss" method only. This test generates 100,000 solutions for each of the nine seed values using the "coin toss" method. The test produces a total of 900,000 solutions for each of the twelve sample problems. A table summarizes the results for each of the twelve sample problems. Table 7-1 is an example of one of the twelve tables. Table 7-1 contains information on the time required to produce 100,000

solutions (column 4) and the number of solutions that were feasible out that sample of 100,000 solutions (column 5). The data listed in the rows below Table 7-1 summarize the results for the single sample file. These data include the ratio of all solutions (including infeasible) to feasible solutions and the average time required to produce a single feasible solution.

Table 7-1: "Coin toss" method used on a 10 node problem

(1) Number of basic vertices	(2) Problem file	(3) Seed value	(4) Time (sec)	(5) Feasible solutions
10	nodes01	0.1	440.17	2
10	nodes01	0.2	440.28	0
10	nodes01	0.3	439.46	0
10	nodes01	0.4	439.46	1
10	nodes01	0.5	439.90	2
10	nodes01	0.6	439.84	1
10	nodes01	0.7	439.68	0
10	nodes01	0.8	439.95	2
10	nodes01	0.9	439.79	0

Total number of solutions generated	900000
Total number of feasible solutions generated	8
Total time required to generate solutions (sec)	3958.53
Number of solutions per feasible solution	112500
Time required per feasible solution (sec)	494.8163

Table 7-2: Summary of "coin toss' data

(1) Problem file	(2) Number of basic vertices	(3) Solutions per feasible solution	(4) Time per feasible solution (sec)
nodes01	10	112500.00	494.816
nodes02	10	150000.00	659.755
nodes81	8	23076.92	57.438
nodes82	8	64285.71	159.931
nodes83	8	18750.00	46.675
nodes61	6	1952.28	2.542
nodes62	6	927.84	1.228
nodes63	6	2238.81	2.909
nodes41	4	169.84	0.129
nodes42	4	262.93	0.181
nodes43	4	233.40	0.166

Table 7-2 summarizes the results of all twelve tables similar to Table 7-1. However, Table 7-2 includes only eleven entries because one of the trials involving 10 basic vertices failed to produce even a single feasible solution in 900,000 random generations with the "coin toss" method. Figures 7-1 through Figure 7-4 represent the data in Table 7-2 graphically. Figure 7-1 shows the average time required to produce a feasible solution in seconds versus the size of the problem as measured by the number of basic vertices.

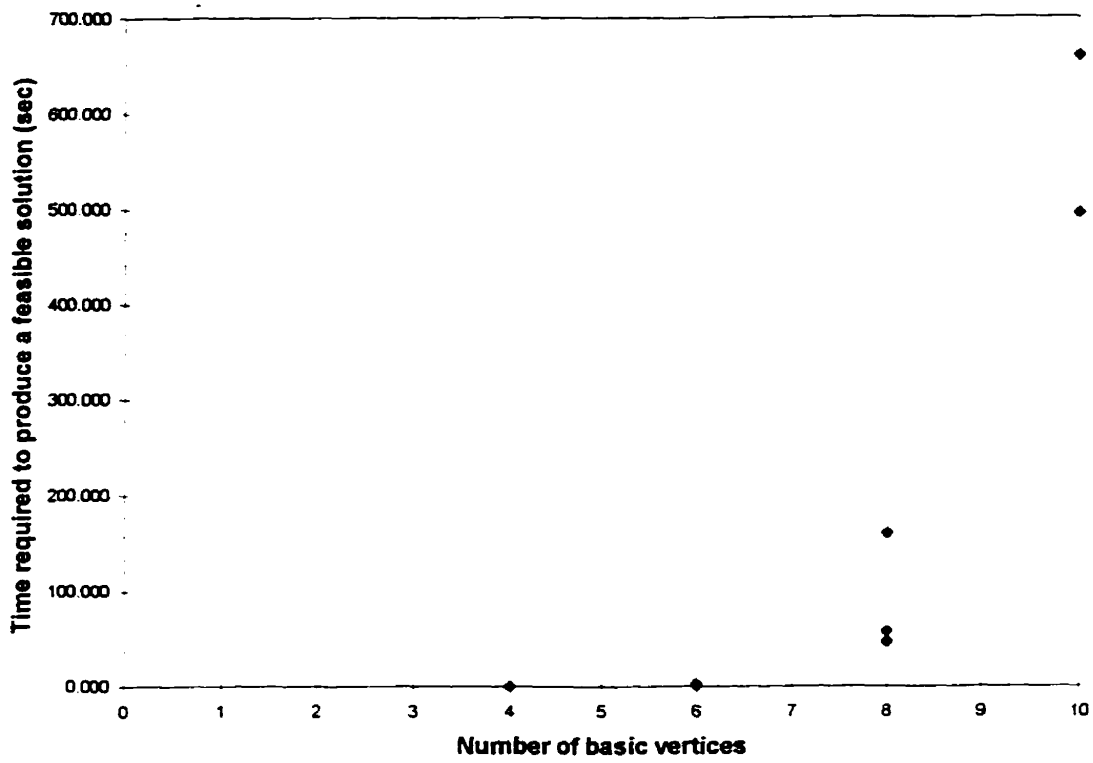


Figure 7-1: Computational effort of "coin toss" method

The plotted points in Figure 7-1 suggest exponential growth in computational effort with the size of the problem. Figure 7-2 shows the same points plotted on a graph with a logarithmic vertical scale. The line plotted through the points is the best fit exponential function obtained through least squares regression. The exponential function shows a good fit.

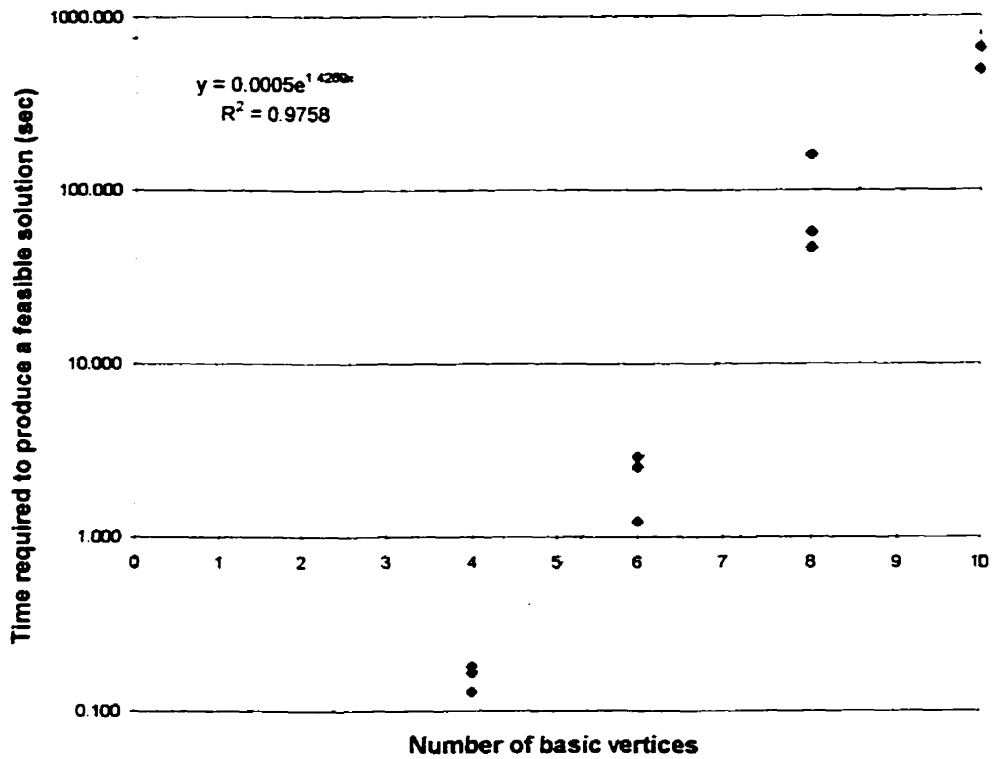


Figure 7-2: Computation effort of "coin toss" method (log)

The execution time of the feasibility testing algorithm is roughly proportional to the size of the Hanan grid. The Hanan grid in turn is proportional to the square of the number of basic vertices. Therefore, the computational effort of the feasibility testing algorithm is proportional to the square of the problem size. The steep rise of the plotted points in Figures 7-1 and 7-2 is not characteristic of computational effort that increases with the square of the problem size. The increase in computational effort must originate from a source other than the time required by the feasibility testing algorithm. The

generation of infeasible solutions is the only other possible source since the "coin toss" method consists only of random assignment of link values and feasibility testing.

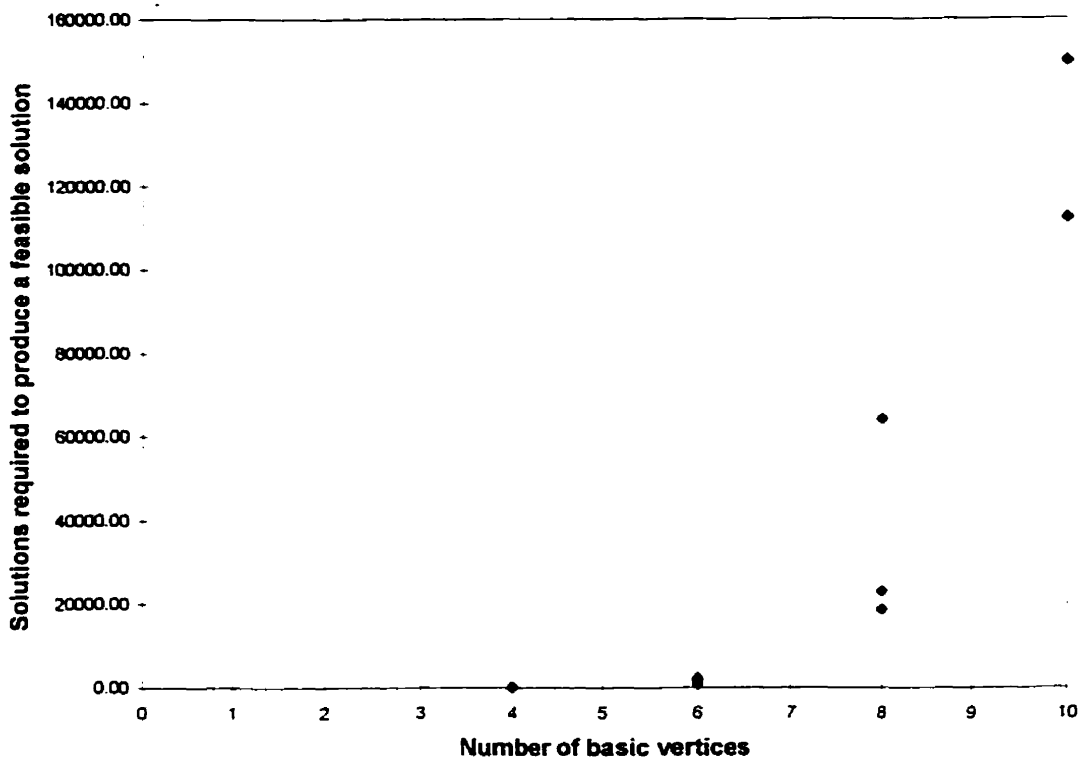


Figure 7-3: Solutions required to produce a feasible solution

Figure 7-3 shows the average number of solutions per feasible solution relative to the problem size (column 3 in Table 7-2). The number of infeasible solutions is independent of the time required to perform the feasibility test. Figure 7-3 shows a steep ascent of the plotted points similar to Figure 7-1. The time required to produce a feasible solution increases exponentially if the number of infeasible solutions required to produce a single feasible solution increases exponentially. The exponential increase in the probability of producing infeasible solutions is readily apparent from Table 7-2 and Figure 7-3. Figure 7-4 shows the same data as Figure 7-3 plotted on a graph with a logarithmic

vertical scale. The best fit line in Figure 7-4 is an exponential function, which appears linear due to the logarithmic scale.

The conclusion drawn from these results is that the "coin toss" method exhibits a non-polynomial computational effort and is therefore unusable, particularly for larger problems. The non-polynomial computational effort results from the increasing rate at which the algorithm generates infeasible solutions with increased problem size.

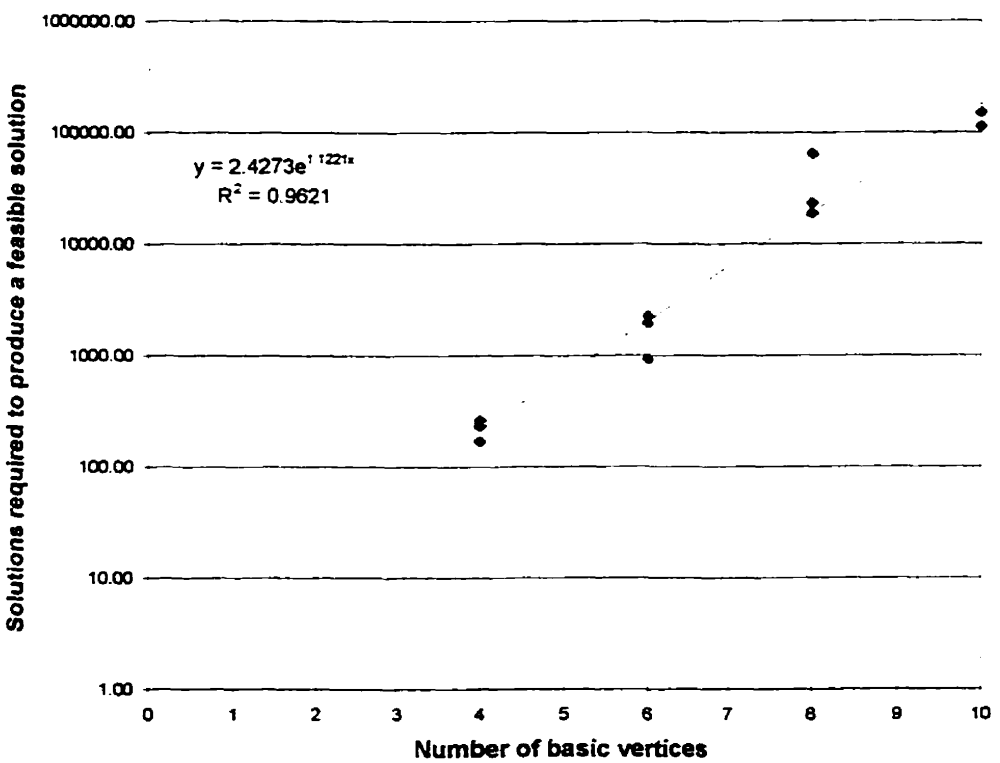


Figure 7-4: Solutions per feasible (log plot)

7.1.3. The Preference and Threshold Method

The preference and threshold method generates feasible solutions with every trial. The time required to generate each solution is longer than with the "coin toss"

method since the method must perform several feasibility tests to establish the alpha threshold value. The "coin toss" method performs only one feasibility test for each solution generated. However, every solution produced by the preference and threshold method is a feasible solution unlike the "coin toss" method.

The tests that assess the computational effort of the preference and threshold method use the same twelve sample problem files as the "coin toss" method. These tests generate fewer solutions because the preference and threshold method produces feasible solutions every time. However, a smaller number of trials does not bias the results in favor of the preference and threshold method. The statistic that forms the basis of comparison is the average length of time required to produce a single feasible solution.

Table 7-3: Preference and threshold method used on a 10 node problem

(1) Number of basic vertices	(2) Problem file	(3) Seed value	(4) Time (sec)	(5) Feasible solutions
10	nodes01	0.1	21.86	100
10	nodes01	0.2	22.08	100
10	nodes01	0.3	22.08	100
10	nodes01	0.4	22.30	100
10	nodes01	0.5	22.30	100
10	nodes01	0.6	22.30	100
10	nodes01	0.7	22.14	100
10	nodes01	0.8	22.02	100
10	nodes01	0.9	21.92	100

Total number of solutions generated	900
Total number of feasible solutions generated	900
Total time required to generate solutions (sec)	199
Number of solutions per feasible solution	1
Time required per feasible solution	0.221111

The test generates 100 solutions for each of the nine seed values for each of the twelve sample problems. The test produces tables similar to Table 7-1 for each of the sample problems. Table 7-3 is an example of one of these tables. The total number of solutions generated (900) and the number of feasible solutions (900) are the same in every case.

Table 7-4 summarizes the data for the preference and threshold method for all twelve sample problems. A graph similar to Figure 7-3 is meaningless for this method because the ratio of all solutions generated to feasible solutions is always 1.

Table 7-4: Summary of preference and threshold data

(1) Problem file	(2) Number of basic vertices	(4) Time per feasible
nodes01	10	0.2211
nodes02	10	0.2209
nodes03	10	0.2267
nodes81	8	0.1379
nodes82	8	0.1410
nodes83	8	0.1357
nodes61	6	0.0969
nodes62	6	0.0931
nodes63	6	0.0930
nodes41	4	0.0727
nodes42	4	0.0721
nodes43	4	0.0756

Figure 7-5 shows the time required to generate a single feasible solution relative the size of the problem in basic vertices. Three differences between Figure 7-1 and Figure 7-5 are evident.

1. Figure 7-5 shows significant improvement in the time required to generate a single feasible solution for every size of problem.
2. The slope of the plotted points suggests a well-behaved polynomial increase in computational effort in Figure 7-5. Figure 7-1 and Figure 7-2 indicate an exponential increase in computational effort. The best fit line in Figure 7-5, obtained through least squares regression, is a square function.
3. There is less scatter in the points in Figure 7-5, which suggests that the behavior of the preference and threshold algorithm is more predictable than the "coin toss" method.

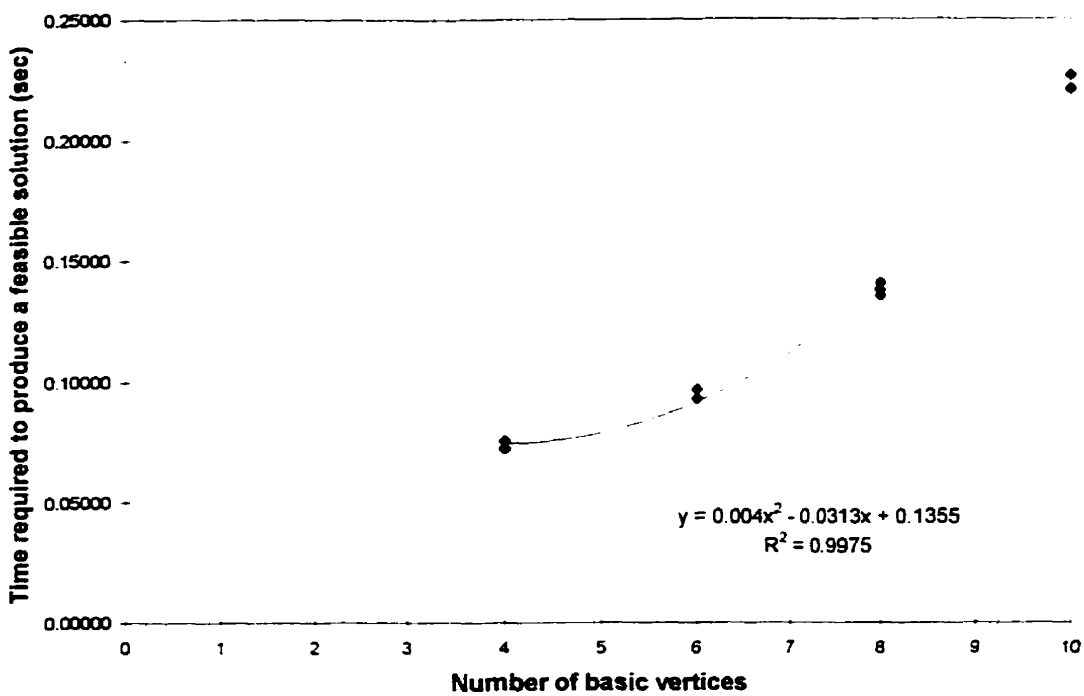


Figure 7-5: Computational effort of preference and threshold method

7.1.4. Batch Programs

Appendix D includes a source code listing of the programs used in this section and a listing of the sample problem data. The MS-DOS batch file "lk.bat" performs the

tests for the "coin toss" method and the file "pt.bat" performs the tests for the preference and threshold method.

7.2. The Evolution Program

7.2.1. Introduction

This section examines the performance of the evolution program described in Chapter 5. The algorithm is similar to the simple genetic algorithm (Goldberg, 1989) with modifications that eliminate the production of infeasible solutions. The program uses the preference and threshold method to generate starting populations. The operators of recombination and mutation incorporate the preference and threshold method as well.

7.2.2. The Method of Evaluation

A single parameter specifies the rate of mutation for the evolution program. The parameter takes a value between 0 and 1 inclusively. A higher value of the parameter results in a greater number of links that undergo mutation. If the value of the parameter is zero no mutation occurs at all. Without mutation the evolution program uses the selection and recombination operators only.

If the value of the mutation rate parameter is 1 the program mutates all links in the layout. Section 5.2 describes the mutation process. The process mutates a link by assigning a random preference value to it. None of the links retain preference values obtained through the previous step of recombination if all links undergo mutation. Mutating all links in a layout has the same effect as generating a completely new layout using random generation by the preference and threshold technique.

A parameter setting that mutates all links is useful as a basis for assessing the performance of the evolution program. The program operating in this mode produces the same quality of solutions as random generation by the preference and threshold method alone. The evolution program does not work properly if it fails to show any improvement over the baseline performance of random generation. The first test of the evolution program establishes the baseline performance of random generation.

A common approach to represent the performance of evolution programs is to plot the value of three statistics that describe how the population changes over time. The three statistics are the fitness of the best solution (minimum), the fitness of the worst solution (maximum) and the average fitness of the population at each iteration. In this context the fitness of a solution refers to the total length of a layout.

7.2.3. The Lower Bound on Feasible Solutions

A lower bound exists for feasible solutions based on the Hanan grid. Any feasible solution that coincides with the lower bound is necessarily a global optimum. The lower bound follows from the fact that a feasible solution must include no fewer than two links from every row and column in the Hanan grid. A solution that has only one link in a row or column of the Hanan grid is infeasible because failure of that link results in partitioning of the network. A feasible solution that consists of only two links in every row and column is necessarily an optimum. However, feasible solutions of this type do not necessarily exist for every problem.

The tests in this section use a single sample problem and experimental trials produced feasible solutions that coincide with the lower bound. Therefore, the global

optimum for the problem is known. This section compares the results of the experimental trials with the known global optimum.

7.2.4. Random Generation

In the tests conducted in this subsection the populations consist of 30 solutions. Figure 7-6 shows the results of the evolution program operating with the mutation parameter set to 1. The problem is a six node problem, which is one of the twelve sample problems used in the previous section. Appendix D refers to this sample problem as "nodes61". The data displayed in Figure 7-6 result from the random generation of 30 solutions (the population size) with each iteration. The trial produces a total of 6,030 solutions. The graph shows that the three statistics, minimum, average, and maximum length, fluctuate about a mean that remains relatively constant through all 200 iterations. There is no discernible trend toward improvement of the solutions over time.

The selection of a population size of 30 is arbitrary for random generation. The population size does not influence the performance of the algorithm and affects only the number of solutions plotted at each iteration. The test uses a population size of 30 to facilitate comparison with tests that follow. Two tests of the evolution program use this population size. Subsection 7.2.7 briefly considers the effect of population size on the performance of the evolution program.

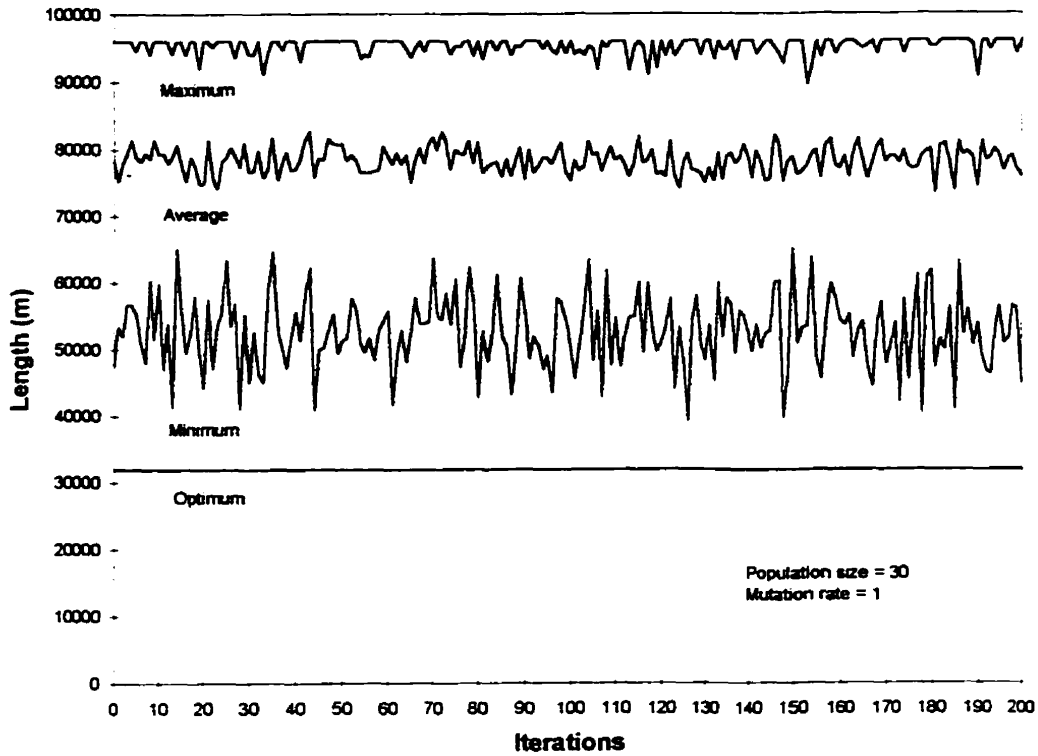


Figure 7-6: Random search with population size of 30

The straight horizontal line in Figure 7-6 labeled "Optimum" does not represent the value of any solution produced during the test. The line represents the value of the global optimum for the problem, a length of 31987.2 m. As mentioned previously, the value coincides with the lower bound on feasible solutions. None of the 6,030 solutions generated in the test achieved this value.

7.2.5. Performance Without Mutation

Figure 7-7 plots the performance of the evolution program for the same six node problem with the mutation parameter set to zero. Only the operators of selection and recombination function in this trial. The program obtains solutions in the first iteration (iteration 0) through random generation by the preference and threshold method. The

populations of all following iterations result from the operations of selection and recombination of the population of the previous iteration.

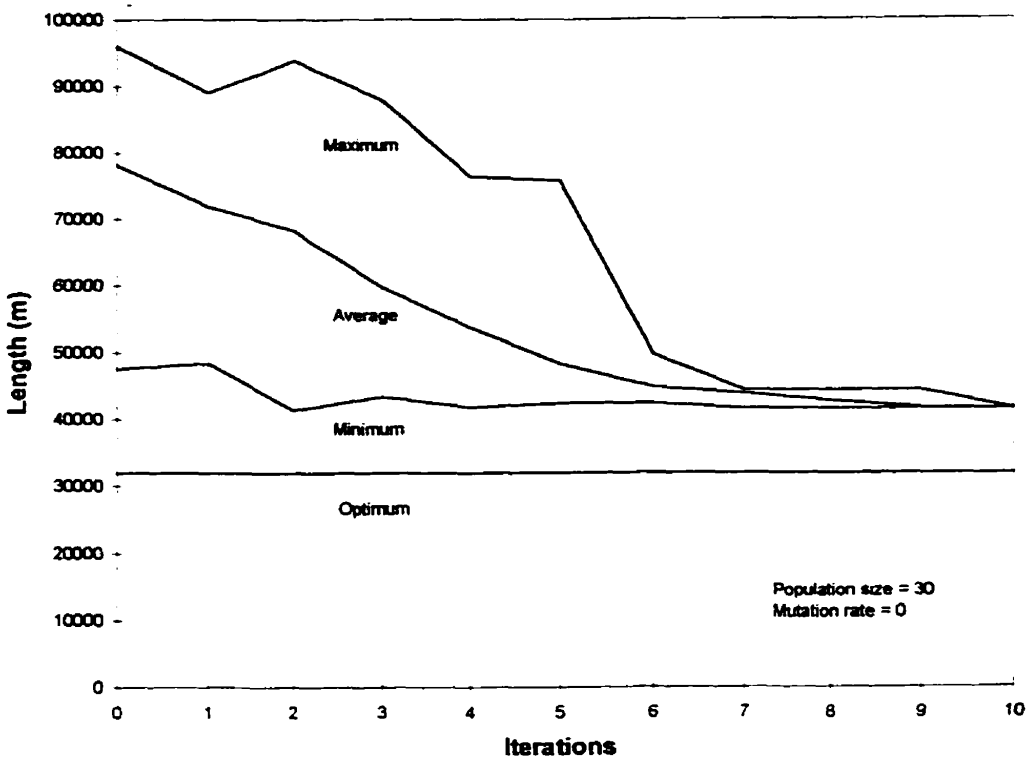


Figure 7-7: Selection and recombination operators only

Section 5.1 explains the recombination operator. The recombination operator cannot generate new solutions if the entire population converges to a single solution (i.e., a single set of preference values). Recombination alone cannot escape convergence because the process of averaging preference values used by the recombination operator cannot produce new preference values once convergence occurs. Averaging used alone or in combination with the selection operator guarantees that convergence will occur rapidly and will terminate the search process.

Two important differences between Figure 7-7 and Figure 7-6 are evident. Figure 7-6 shows the results from 200 iterations. The purpose of the trial was to obtain a representative sample of random generation. The program stops arbitrarily after 200 iterations. However, the search could have continued indefinitely. Figure 7-7 shows that the population converges after only 10 iterations with no mutation. The program cannot produce any new solutions after convergence and the algorithm must halt at this point.

The second difference between Figure 7-7 and Figure 7-6 is a discernible trend toward improvement in all three of the population statistics, maximum, average and minimum length. However, the trend toward improvement for minimum length is not as clearly evident in this example.

The data in Figure 7-7 show that the recombination and selection operators improve the quality of solutions over time. However, the rapid convergence of the population causes the search for new solutions to terminate prematurely. The use of a larger population of solutions will delay the convergence but will not eliminate it. The use of mutation prevents convergence, as in Figure 7-6, but full mutation produces the same poor results as random manipulation.

7.2.6. Performance with Mutation Included

Figure 7-8 shows the results of the evolution program with the same six node problem as Figure 7-6 and Figure 7-7. This example uses the same population size, 30 solutions. However, the mutation parameter is 0.001. This small introduction of mutation is sufficient to sustain the search indefinitely without convergence. The program terminates arbitrarily after 200 iterations but the program can maintain the search indefinitely with these parameters. A general trend towards improved solutions is evident

in the minimum and average values of the population but large fluctuations appear in the maximum value. The trend toward improvement occurs only in the first 25 iterations. Chapter 5 refers to the inability to sustain the trend toward improvement as the problem of diminishing returns. After the first 25 iterations the values of the minimum and average of the population appear to fluctuate about a constant mean. Through this section the values of the minimum and average are still above the optimum but well below the best values for these statistics obtained through random search in Figure 7-6.

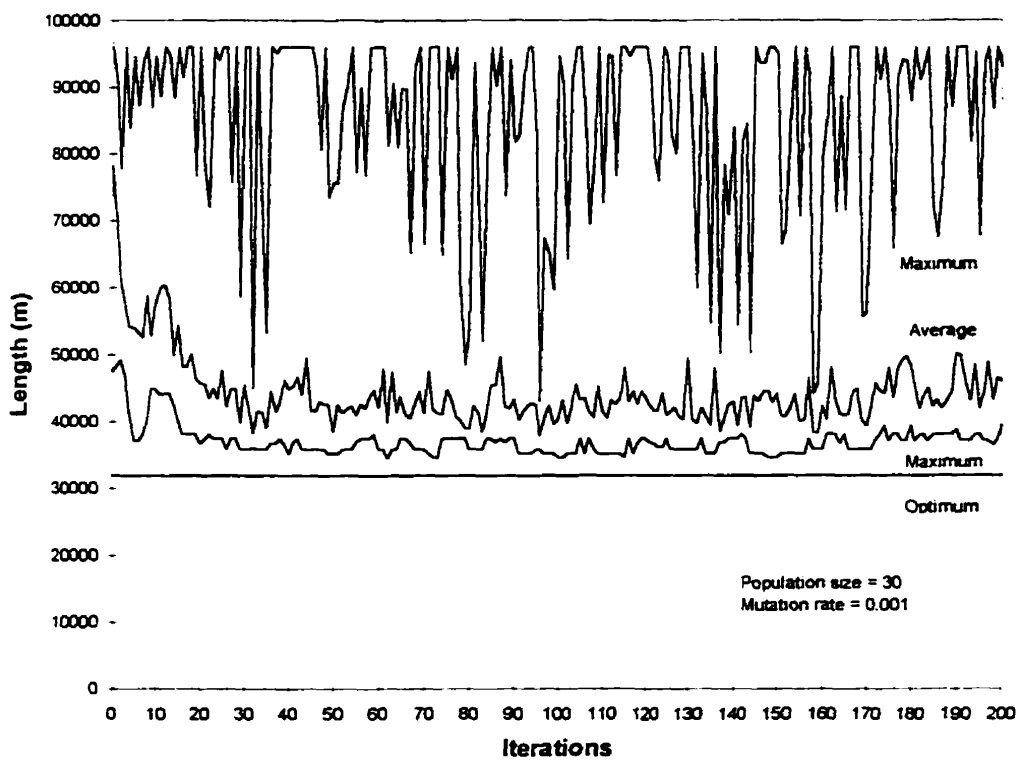


Figure 7-8: Population of 30 and low mutation rate

Figure 7-9 shows the results of a trial with the same problem. This trial uses a smaller population size of ten solutions and the mutation rate increases to 0.005. Experience shows that reducing the size of the population without increasing the

mutation rate introduces the possibility of premature convergence. The results of this test are similar to those obtained in the previous example, except that the average of the population fluctuates in a manner similar to the maximum solution in previous test. This test obtains the global optimum solution in iteration 172. However, the reader should not draw the conclusion that this set of configuration parameters (population size and mutation rate) guarantees the optimal solution. Due to the stochastic nature of the program, the production of the global optimum is unpredictable with any set of configuration parameters even when the minimal solution fluctuates near the optimum.

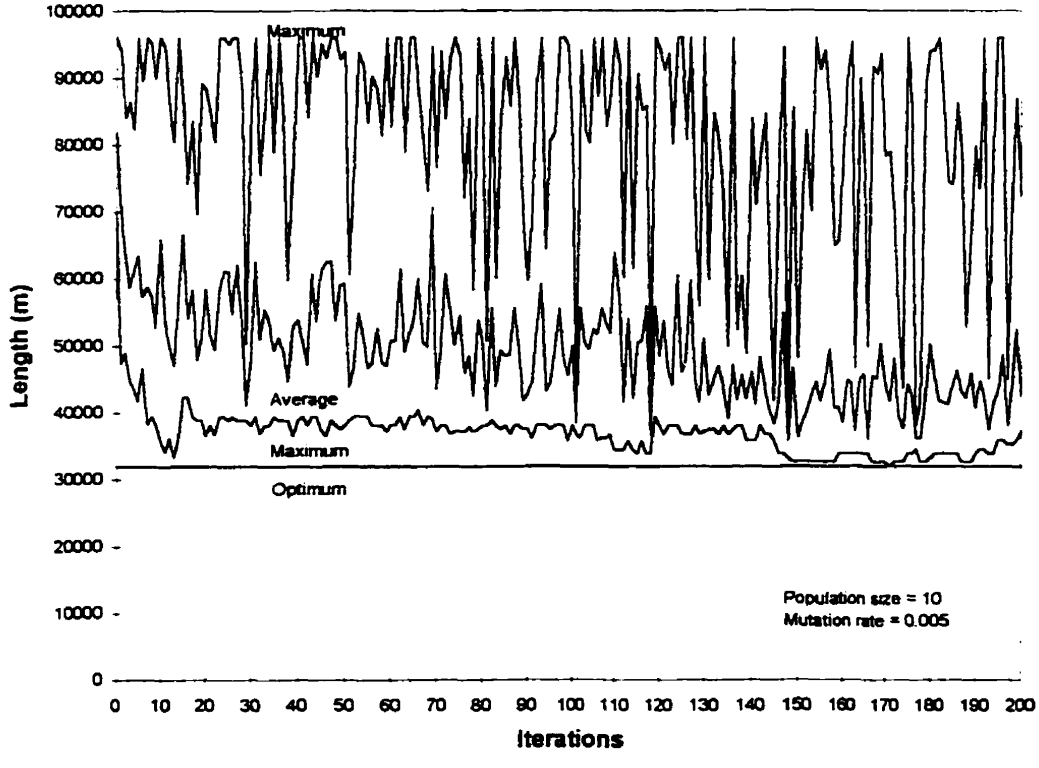


Figure 7-9: Population size of 10 and higher mutation rate

7.2.7. Conclusions from the Tests

The performance of the minimum solutions in the graphs of Figure 7-9 and Figure 7-8 is similar. However, the trial that uses a population of 10 solutions rather than 30 solutions obtains similar results in less time since each iteration generates fewer solutions. The results of these tests suggest that smaller sized populations with higher mutation rates achieve the same results as larger populations in less time. Both tests clearly demonstrate the problem of diminishing returns.

7.2.8. Batch Programs

Appendix D includes a source code listing of the program used in this section and a listing of the sample problem data. The MS-DOS batch file "epbat.bat" performs the tests.

7.3. The Agitator

7.3.1. Introduction

The agitator component of the learning program is similar to the (1+1)-ES evolution strategy developed by Rechenberg (Rechenberg, 1965). The term "(μ+λ)-ES" has the following meaning. The term "μ" refers to the base population size. The term "λ" refers the number of new solutions generated with each iteration. The "+" term indicates that during each iteration the algorithm selects the best μ solutions from both the solutions in the previous population (μ) and the new solutions generated (λ). The alternative approach "(μ, λ)-ES" selects the best μ solutions from only the λ new solutions generated.

The (1+1)-ES evolution strategy operates on a population consisting of one solution only. The program improves the single solution through a series of iterations. During each iteration the program makes a copy of the solution and mutates the copy by making minor changes to it at random. The program then compares the mutated copy with the original solution. It discards the mutated copy if the copy is not an improvement over the original solution. However, if the mutated copy represents an improvement the program replaces the original solution with the copy. Section 6.1 describes how the agitator component of the learning program uses this procedure.

Subsection 7.3.2 presents the results of a small number of tests on the agitator component operating in isolation from the rest of the learning program. The tests in this section use the same six node problem as the tests in Section 7.2. The tests compare the performance of the evolution program of the previous section with the agitator. Section 7.3.3 examines some of the results produced by the agitator in greater detail. This examination includes the actual changes that occur to network geometry and some speculation about the causes of poor performance.

7.3.2 Comparison of Evolution Program and Agitator

Table 7-5 summarizes the results of nine trials of the evolution program from Section 7.2. The sample problem is the six node problem referred to as "nodes61" in Appendix D. The configuration parameters used in each of the nine trials are the same as those used to produce Figure 7-9. The trial in row 5 of Table 7-5 is the test trial shown in Figure 7-9. The evolution program uses a population size of 10 and a mutation parameter of 0.005. Each trial executes for 200 iterations. Only the seed value changes

with each of the nine trials. If the program obtains the optimal solution during the trial, column 5 lists the iteration in which the solution occurred.

Table 7-6 summarizes the results of nine trials using the (1+1)-ES method of the agitator. Each trial in Table 7-6 executes for 200 iterations as in Table 7-5. According to Table 7-6, four out of nine trials produced the optimal solution. The results are substantially better than those of Table 7-5. In addition, the execution of all nine trials in Table 7-6 requires less time than a single trial in Table 7-5. The agitator produces only one solution per iteration while the evolution program produces 10 solutions per iteration. Therefore, the agitator requires one tenth of the time to perform an iteration.

Table 7-5: Nine trials with the evolution program

	(1) Problem file	(2) Number of basic vertices	(3) Seed value	(4) Best result	(5) Optimum obtained
(1)	nodes61	6	0.1	33203.8	
(2)	nodes61	6	0.2	34856.4	
(3)	nodes61	6	0.3	33203.8	
(4)	nodes61	6	0.4	33943.3	
(5)	nodes61	6	0.5	31987.2	171
(6)	nodes61	6	0.6	33943.3	
(7)	nodes61	6	0.7	33899.5	
(8)	nodes61	6	0.8	35291.1	
(9)	nodes61	6	0.9	33203.8	

Table 7-6: Nine trials with the agitator

	(1) Problem file	(2) Number of nodes	(3) Seed value	(4) Best result	(5) Optimum obtained
(1)	nodes61	6	0.1	36422.0	
(2)	nodes61	6	0.2	31987.2	86
(3)	nodes61	6	0.3	35810.8	
(4)	nodes61	6	0.4	39596.0	
(5)	nodes61	6	0.5	31987.2	200
(6)	nodes61	6	0.6	31987.2	21
(7)	nodes61	6	0.7	36942.1	
(8)	nodes61	6	0.8	31987.2	79
(9)	nodes61	6	0.9	36422.0	

7.3.3. Examination of Layout Geometry

Figure 7-10 plots the performance of the trial in row 6 of Table 7-6 over time as a solid line. This trial produced the optimal solution in the least time. Figure 7-10 also plots the performance of the trial in row 4 of Table 7-6 as a dotted line. This trial failed to produce the optimum and terminated at 200 iterations with the final solution with the greatest total length. The trials in row 6 and row 4 represent the extremes of performance in Table 7-6.

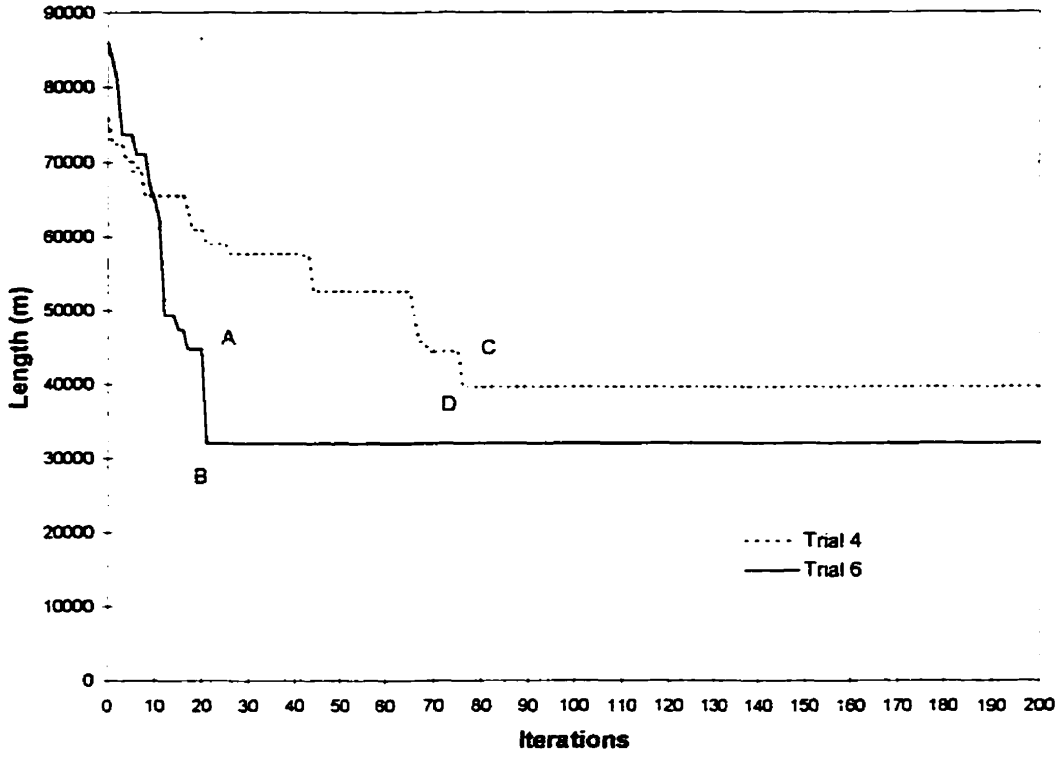


Figure 7-10: Best and worst performance of the agitator

There are two differences between performance graphs of the agitator (Figure 7-10) and the evolution program (Figures 7-6 through 7-9). (1) Figure 7-10 plots only one statistic per trial because the population consists of only one solution. (2) The statistics never increase in value in Figure 7-10 unlike the evolution program. The total length of the agitator's single solution either decreases or remains the same with each iteration.

The solid line in Figure 7-10 (the best trial) decreases initially with a steep slope. The graph maintains the steep slope until it reaches the optimal solution. In contrast the dotted line in Figure 7-10 decreases with a more gradual slope and becomes completely flat well above the optimal solution. The dotted line does not achieve the results

achieved by the solid line even though the starting solution of the dotted has less total length than the starting solution of the solid line.

The differing performance of the two lines in Figure 7-10 suggests that certain preference schemes optimize more readily than others due to a greater inherent probability for improvement. Section 5.3 considers this concept and the complexity of changes that improvement requires to maintain feasibility. Random search techniques are less likely to improve solutions that require a complex rearrangement of links. The remainder of this section explores this idea.

The discussions in this section offer an explanation for the behavior of the best and worst trials in Table 7-6. The explanation offered here is that certain starting solutions (preference schemes) are inherently better performers. Transitions that lead to the globally optimal solution have a higher likelihood of occurring with these solutions than transitions that would trap the program at a local optimum. In this context the term "local optimum" refers to a solution for which continued improvement requires a very lengthy and impractical search.

Figures 7-11a and 7-11b show two examples of the layouts produced by the sample trial in row 6 of Table 7-6. Figure 7-11a is the layout that occurs at the point labeled A in Figure 7-10. The solution has a total length of 44763.5 m. Figure 7-11b is the layout that occurs after the improvement in iteration 21 corresponding to the point labeled B in Figure 7-10. The layout in Figure 7-11b is a global optimum.

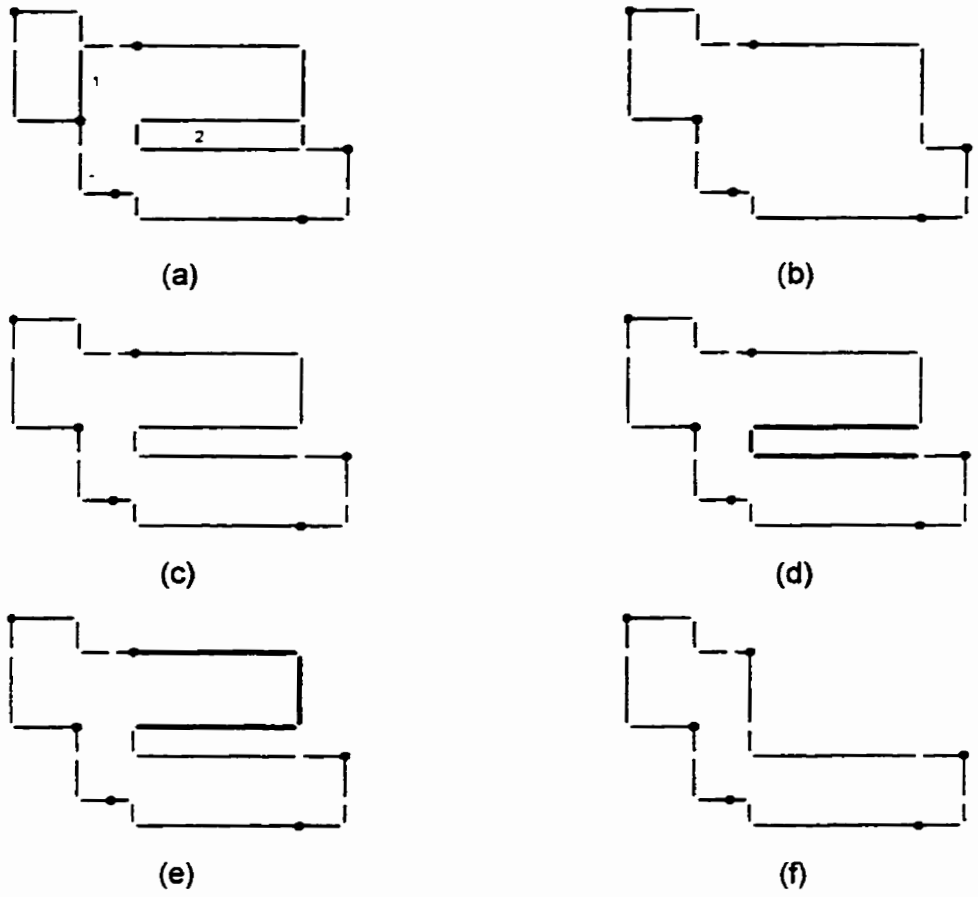


Figure 7-11: Improvement in trial 6 of Table 7-6

An examination of the transition between the layout in Figure 7-11a and Figure 7-11b shows that two improvements occur simultaneously. The optimal layout in Figure 7-11b consists of a single natural loop while the sub-optimal layout in Figure 7-11a consists of three natural loops. One of the improvements consists of the removal of the link labeled 1 in Figure 7-11a. The other improvement consists of the removal of one or more of the three internal links of the small natural loop labeled 2 in Figure 7-11a. (The improvement can consist of removing only one of the three internal links since the remaining links constitute branched components after the removal. The algorithm removes all branched components.) The improvement that occurs between the layouts

in Figures 7-11a and 7-11b consists only of the removal of links. It does not consist of a complex pattern that requires the simultaneous removal and addition of links to maintain feasibility. There are many combinations of removal patterns that can produce the transition between the layouts in Figures 7-11a and 7-11b because the algorithm removes any remaining branched components. Therefore, there is a relatively high probability of the agitator selecting this transition. However this is not the only possible transition.

Figure 7-11c offers an another possible transition originating from Figure 7-11a. The layout in Figure 7-11c does not include the link labeled 1 in Figure 7-11a. The layout also does not include the external link of the natural loop labeled 2 either. (The term "external" refers to a link that borders with the region outside the loops.) The layout in Figure 7-11c is a feasible improvement over the layout in Figure 7-11a. However, the transition to the optimal layout in Figure 7-11b now requires the simultaneous addition and removal of links to eliminate the circuitous routing. Figure 7-11d shows the required changes. The pattern represented in bold in Figure 7-11d represents an unnecessary "jog" in the layout. To remove the jog the program must remove one of the three links in bold while simultaneously restoring the link removed in the previous transition. Figure 7-11d represents the link the program must add as a dotted line.

Another possible improvement to the layout in Figure 7-11c results from identifying another set of links as a jog. Figure 7-11e is the same layout as Figure 7-11d except that the figure identifies a different jog in bold. The change removes of one of the bold links and simultaneously adds the link identified by the dotted line. The improvement produces the layout in Figure 7-11f as another global optimum.

Experience has shown that the agitator produces either of the types of improvement required by the layout in Figure 7-11c very rarely. Usually the agitator must sustain the search for a long time before the required improvement occurs. The program's inability to produce the required improvement becomes obvious to someone observing progress of the program.

Figures 7-11c through 7-11f represent possible alternatives that could have occurred in trial 6 of Table 7-6. However, the actual sequence of events included only the transition from Figure 7-11a to 7-11b. Had the transition proceeded from Figure 7-11a to that in Figure 7-11c the search would very likely have become trapped at this point. However, there are more combinations of link removal patterns that will produce the transition from Figure 7-11a to 7-11b than produce the transition from Figure 7-11a to 7-11c. If all patterns of removal of links have an equal probability of occurrence, then the layout in Figure 7-11a is more likely to reach the global optimum than it is to become trapped by the solution in Figure 7-11c.

Figures 7-12a and 7-12b represent the two solutions labeled C and D in Figure 7-11. Trial 4 of Table 7-6 produces the two solutions. The total length of the solution in Figure 7-12a is 44418.9m and the total length of the solution in Figure 7-12b is 39596.0 m. The improvement occurs as a result of the removal of a single link shown in bold in Figure 7-12a. The program fails to find a better solution than Figure 7-12b before the 200 iterations finish. However, the removal of the link shown in bold in Figure 7-12b accomplishes a relatively simple improvement. The removal of the link results in the layout in Figure 7-12c, which has a total length of 38248.2 m and therefore does not constitute a global optimum. The transition from Figure 7-12b to 7-12c is not a complex

one. The simple removal of one link should have a relatively high probability of occurrence. Yet the agitator failed to produce the change in well over 100 iterations.

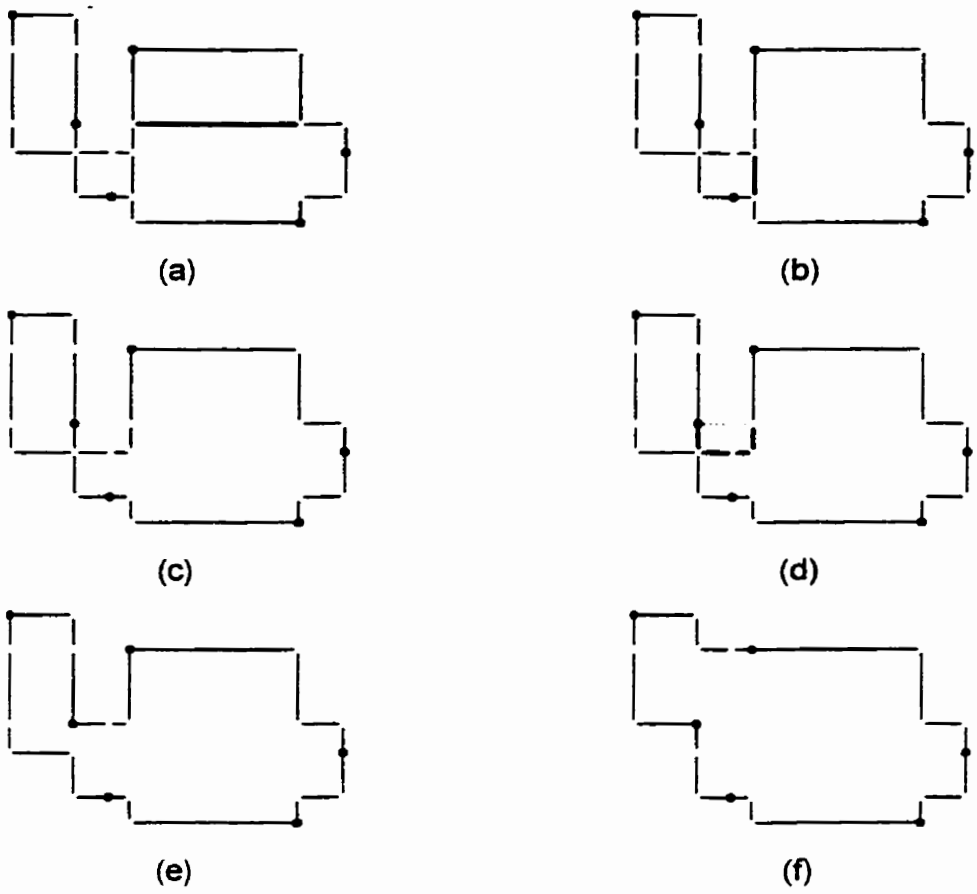


Figure 7-12: Improvement in trial 4 of Table 7-6

An extended version of trial 4 of Table 7-6 consisting of 1000 iterations did not produce the optimal solution as well. In the extended trial the transition from Figure 7-12b to 7-12c occurred at iteration 254. However, no more improvement occurs after the transition although much more improvement is possible. Figure 7-12d is a copy of the layout in Figure 7-12c with a jog identified in bold. The removal of the jog in Figure 7-12d produces the layout in Figure 7-12e. While Figure 7-12e represents an improvement

over the layout in Figure 7-12d, the layout in Figure 7-12e has a total length of 36509.0 m and still does not represent a global optimum. Improvement on the layout in Figure 7-12e is not possible without the simultaneous removal and addition of a large number of links. An example of this type of very complex change is the transition from Figure 7-12e to Figure 7-12f.

The argument that certain starting solutions are inherently easier to optimize suggests that the best search strategy is one that picks the best result from several trials with different starting solutions. In each trial the method terminates the search once improvement becomes difficult and then begins a new trial with a new starting solution.

Another approach to optimization of the layouts does not use a random search component at all. This approach identifies those patterns in the layout that are potential sources of improvement and replaces them. For example, the patterns identified in Figure 7-11 and 7-12 consist of two types

1. Single links that bridge two parts of the network (e.g., the bold link in Figure 7-12a)
2. The three link pattern referred to as a jog (e.g., the bold links in Figure 7-11c).

The rule based approach recognizes patterns in the layout geometry and makes modifications that are not random changes. The tests in the next sections compare the performance of the rule based approach with the evolution based strategies.

7.3.4. Batch Programs

Appendix D includes a source code listing of the programs used in this section and a listing of the sample problem data. The MS-DOS batch file "ep2.bat" performs the tests for the evolution program and the file "esbat.bat" performs the tests for the agitator.

7.4. Experiments with the Learning Program

7.4.1. Introduction

Section 7.4 examines the learning program by first presenting results produced by a simple version of the program. Each successive subsection presents results from increasingly complex versions of the program and demonstrates the effects of the improvements to the basic algorithm. This section does not explain the mechanisms or rationale for the improvements. The section only demonstrates the effects on performance. Chapter 6 explains the mechanisms and logic of each of the improvements.

The demonstrations use a small six node problem referred to as "nodes61" in Appendix D. These demonstrations use this problem for three reasons. (1) The problem provides consistency with the previous sections. (2) A small problem reduces the time required to produce the tests performed in these sections. (3) The simplicity of smaller problems affords greater detail and clarity when explaining the results of tests (e.g., knowing the optimal solution).

7.4.2. Generating Rules to Match the Agitator

Table 7-7 is an example of data obtained from the first of the tests on the learning program. This version of the learning program uses a single starting solution. The program uses the algorithm presented in Figure 6-3 in the previous chapter. The algorithm used in this section does not incorporate the mechanisms of cheating and forgetting. In addition the algorithm includes the agitator in only one location in the flowchart, unlike Figure 6-4. In this algorithm the production system operates directly on the starting solution. The program generates a single starting solution and uses that starting solution through all learning iterations. During each iteration the production system applies the rules to the starting solution using the best-first control strategy. The agitator attempts to improve the best result obtained by the production system. If the agitator obtains an improvement the program derives a new rule and adds the rule to the rule base. If the agitator cannot perform an improvement the program ends that iteration without learning a rule.

At the end of an iteration the program tests the production system. The program applies the rules that it has acquired to this point to test the production system's ability to improve the starting solution. The program writes the results of the test to a file. The results consist of the length of the solution that the production system produced and the number of rules in the rule base. The program performs the test after every learning iteration. The data collected document the increasing size of the rule base and the changes in the production system's ability to solve the problem.

This test measures the production system's ability to learn from the agitator, to store and reuse improvements that the agitator obtains through random search. Table 7-7 shows the results from 15 iterations of the test. Iteration 0 consists of the random

generation that produced the starting solution. The test uses the same six node problem as the previous sections. Global optima for this problem have a total length of 31987.2 m. By iteration 8 the rule base produces a solution with a total length of 34074.2 m. Although Table 7-7 shows only 15 iterations, the test continues to a total of 200 iterations without further improvement on this solution. Although the program has not produced the global optimum, no more learning is possible beyond the 34074.2 m solution because the agitator cannot generate further improvement within the 200 iterations.

Table 7-7: Results of a single trial with the learning program

(1) Iteration	(2) Solution length (m)	(3) Number of rules
0	85964.8	0
1	71496.1	2
2	61327.5	28
3	53504.7	44
4	51157.7	56
5	48288.5	64
6	41942.7	98
7	38812.2	102
8	34074.2	110
9	34074.2	110
10	34074.2	110
11	34074.2	110
12	34074.2	110
13	34074.2	110
14	34074.2	110
15	34074.2	110

Table 7-8 summarizes the results of 9 trials similar to the test in Table 7-7. Each of these tests operates on the same sample problem using a different seed value and therefore a different random number sequence and starting solution. At the beginning of each trial in Table 7-8 the rule base is empty. Each trial consists of 200 iterations. Column 5 lists the iteration in which the program first obtains the best solution. In all nine

cases the results of all subsequent iterations produced this best solution relatively early and the size of the rule base did not change beyond this point.

The results of this test are not useful from the perspective of solving real network problems. However, the test is important because it demonstrates the ability of the rule formulator and the production system to create a rule base that matches the ability of the agitator. Ideally the rule formulator and the production system should be capable of reproducing any improvement obtained by the agitator.

Table 7-8: Results of nine separate trials with learning program

	(1) Seed value	(2) Starting solution length (m)	(3) Final solution length (m)	(4) Number of rules	(5) Iteration of best solution
(1)	0.1	86096.4	31987.2	108	29
(2)	0.2	94049.8	36159.4	82	11
(3)	0.3	88528.8	34769.0	110	83
(4)	0.4	75796.1	34682.8	74	8
(5)	0.5	95961.6	38591.2	28	4
(6)	0.6	85964.8	34074.2	110	8
(7)	0.7	87920.5	31987.2	82	200
(8)	0.8	91355.4	31987.2	148	73
(9)	0.9	86096.4	31987.2	146	102

The final solutions in Table 7-8 are similar to those in Table 7-6 produced by the agitator alone. Four out of nine trials obtained the optimal solution. In the five trials that did not obtain optimal solutions the program reaches a solution that behaves like a local optimum at which improvement appears to stop. In the sub-optimal trials the program reaches the local optimum at an early iteration and all following iterations have no effect. Learning ceases because the agitator is not able to produce an improvement. The conclusion from the results of Table 7-8 is that the production system is fully capable of learning from the improvements made by the agitator. The agitator rather than the

production system appears to be the factor limiting the performance of this variation of the algorithm.

As stated previously, each trial in Table 7-8 begins with an empty rule base. The program does not retain knowledge derived from attempting to solve the problem from one starting solution when it begins a new trial with another starting solution. The next subsection explores the effects of sharing rules obtained from different starting solutions.

Table 7-9: Nine starting solutions with a common rule base

	(1) Starting solution length (m)	(2) Final solution length (m)	(3) Number of rules	(4) Iteration of best solution
(1)	95961.6	34072.6	300	17
(2)	70492.7	31987.2	268	4
(3)	69886.8	31987.2	276	5
(4)	72145.4	31987.2	300	17
(5)	80922.1	35289.2	320	62
(6)	90356.1	36248.0	320	62
(7)	76145.0	31987.2	268	4
(8)	79229.3	31987.2	268	4
(9)	88223.6	31987.2	320	62

7.4.3. Sharing Rules Obtained from Different Starting Solutions

Table 7-9 shows the results of a single trial of a modified version of the learning program. This trial uses nine starting solutions and a common rule base. The program can apply the rules derived from one starting solution in an iteration that begins with a different starting solution, unlike the trials in Table 7-8 which are independent of each other. The performance of the agitator with respect to any given starting solution does not limit the production system. Instead the production system can make use of improvements obtained from other starting solutions.

The algorithm used to obtain the results in Table 7-9 is the same as described in Subsection 7.4.2 and Figure 6-3. However, now a single iteration consists of performing the procedure in Figure 6-3 for each of the nine starting solutions. Upon completion of each iteration the program tests the performance of the production system for each of the nine starting solutions. The program records the length of the solutions and the size of the rule base with each iteration as in the previous test. Six of the nine starting solutions in Table 7-9 produced the optimal solution.

Table 7-10: Learning with 10 cycles per iteration

	(1) Starting solution length (m)	(2) Final solution length (m)	(3) Number of rules	(4) Iteration of best solution
(1)	95961.6	31987.2	268	2
(2)	70492.7	31987.2	268	2
(3)	69886.8	31987.2	268	2
(4)	72145.4	31987.2	276	3
(5)	80922.1	31987.2	244	1
(6)	90356.1	36159.9	284	4
(7)	76145.0	31987.2	244	1
(8)	79229.3	31987.2	244	1
(9)	88223.6	31987.2	276	3

Table 7-10 shows the results of an algorithm very similar to the algorithm that produced the data in Table 7-9. This algorithm performs ten iterations of the procedure in Figure 6-3 before changing the starting solution. From this point the term "learning cycle" refers to a single iteration of the procedure in Figure 6-3 or Figure 6-4. Each "iteration" of the learning program consists of one or more learning cycles with each of the starting solutions in the set. Therefore, this variation of the algorithm performs ten learning cycles with each training solution during each iteration. This section includes an example of the performance of this algorithm because it forms the basis of tests that

follow. The logic behind the next two modifications to the algorithm requires several learning cycles with each starting solution. Table 7-10 demonstrates that performing several learning cycles with each starting solution does not degrade the performance of the algorithm. The results in Table 7-10 appear to be superior to those of Table 7-9.

7.4.4. Cheating

Table 7-11 shows the results of the same algorithm used to produce Table 7-10. However, the algorithm now incorporates cheating. Once the program finds the global optimum from one of the starting solutions cheating virtually guarantees that the program produces an optimal solution from all of the starting solutions shortly thereafter. The cheating process stores the best solution that the program has found and uses that solution when the agitator fails to produce an improved solution. Table 7-11 indicates that by the second iteration the learning program has constructed a rule base consisting of 328 rules that is capable of producing the optimal solution from all nine starting solutions.

Table 7-11: Learning program using cheating

	(1) Starting solution length (m)	(2) Final solution length (m)	(3) Number of rules	(4) Iteration of best solution
(1)	95961.6	31987.2	328	2
(2)	70492.7	31987.2	328	2
(3)	69886.8	31987.2	328	2
(4)	72145.4	31987.2	272	1
(5)	80922.1	31987.2	272	1
(6)	90356.1	31987.2	328	2
(7)	76145.0	31987.2	272	1
(8)	79229.3	31987.2	272	1
(9)	88223.6	31987.2	272	1

7.4.5. Use of Sample Starting Solutions

The ability to obtain the optimal solution from all starting solutions represents an improvement over the algorithms presented to this point. However, to be useful the learning program must be able to demonstrate that it can use its knowledge to solve problems it has not encountered previously. The next tests are a step closer to this goal. In the tables and graphs that follow the tests use two sets of starting solutions. The program uses nine solutions for acquiring rules as in the previous tests. The term "training set" designates this set of starting solutions. Another larger set of starting solutions, referred to as "samples" or "additional samples" consists of 18 starting solutions that the program does not use in the learning process. The program tests the performance of the production system at the end of each iteration as before. It tests the production system on the starting solutions that it uses to acquire rules as with the previous algorithm. However, the tests now include the larger sample of starting solutions, the additional samples, that the program does not use in the learning process. The tests still involve solving the same problem repeatedly. However, the tests now measure performance on starting solutions other than those used to acquire rules. The performance of the production system with respect to the additional samples is an indication of the program's ability to generalize from learning based on the training set.

Table 7-12 shows the results after 20 iterations of the learning program. Each iteration consists of ten learning cycles with each of the nine starting solutions in the training set. Column 1 in Table 7-12 lists the total length of the starting solutions. Column 2 lists the results produced by the production system at the end of 20 iterations. An asterisk indicates an optimal solution. Table 7-12 shows that the production system is capable of producing the optimal solution from all of the starting solutions in the training

set. However, the performance with the additional samples is not nearly as good. The production system produces the optimal solution from only one of the eighteen starting solutions that the program did not use in the learning process.

Table 7-12: Results of 20 iterations of learning with cheating

		(1) Starting solution length (m)	(2) Final solution length (m)	
Training set	(1)	95961.6	31987.2	*
	(2)	70492.7	31987.2	*
	(3)	69886.8	31987.2	*
	(4)	72145.4	31987.2	*
	(5)	80922.1	31987.2	*
	(6)	90356.1	31987.2	*
	(7)	76145.0	31987.2	*
	(8)	79229.3	31987.2	*
	(9)	88223.6	31987.2	*
Samples	(1)	95961.6	31987.2	*
	(2)	85574.2	38724.0	
	(3)	60367.3	39679.2	
	(4)	85791.7	35289.2	
	(5)	90267.2	38766.2	
	(6)	88486.2	38248.2	
	(7)	62064.9	36942.6	
	(8)	79753.0	37767.4	
	(9)	61453.8	45025.0	
	(10)	76187.1	40592.3	
	(11)	76749.7	42849.6	
	(12)	83832.4	50759.1	
	(13)	85616.9	42981.2	
	(14)	84095.7	39027.2	
	(15)	47634.7	42243.3	
	(16)	94918.9	35855.6	
	(17)	55457.9	35289.2	
	(18)	55238.1	34507.8	

Table 7-13 shows the progress of the learning program with each iteration. Column 2 lists the mean length of solutions produced from the training set. Column 3 lists the mean length of solutions in the additional sample set. By iteration 3 the learning

program has acquired a sufficient rule base to optimize all nine training solutions fully. The rule base does not increase in size beyond iteration 3 because once the production system achieves optimality from all solutions in the training set no more learning is possible.

Table 7-13: Progress of mean length with cheating enabled

(1) Iteration	(2) Training set mean length (m)	(3) Sample mean length (m)	(4) Number of rules
0	80373.6	76080.6	0
1	35449.3	41029.1	276
2	33894.8	39505.4	356
3	31987.2	39251.9	396
4	31987.2	39251.9	396
5	31987.2	39251.9	396
6	31987.2	39251.9	396
7	31987.2	39251.9	396
8	31987.2	39251.9	396
9	31987.2	39251.9	396
10	31987.2	39251.9	396
11	31987.2	39251.9	396
12	31987.2	39251.9	396
13	31987.2	39251.9	396
14	31987.2	39251.9	396
15	31987.2	39251.9	396
16	31987.2	39251.9	396
17	31987.2	39251.9	396
18	31987.2	39251.9	396
19	31987.2	39251.9	396
20	31987.2	39251.9	396

Figure 7-13 plots the performance of the production system for each of the two sets of starting solutions. Figure 7-13 is a graph of the data in Table 7-13. The figure shows the mean length of solutions produced by the training set as a solid line. The mean for the training set reaches the optimal value by iteration 3. The figure shows the mean length of the additional samples as a dotted line. The performance of the mean of

the addition samples is relatively poor in relation to the training set. Figure 7-13 indicates clearly that the learning performed by the algorithm terminates early and the algorithm does a relatively poor job of generalization.

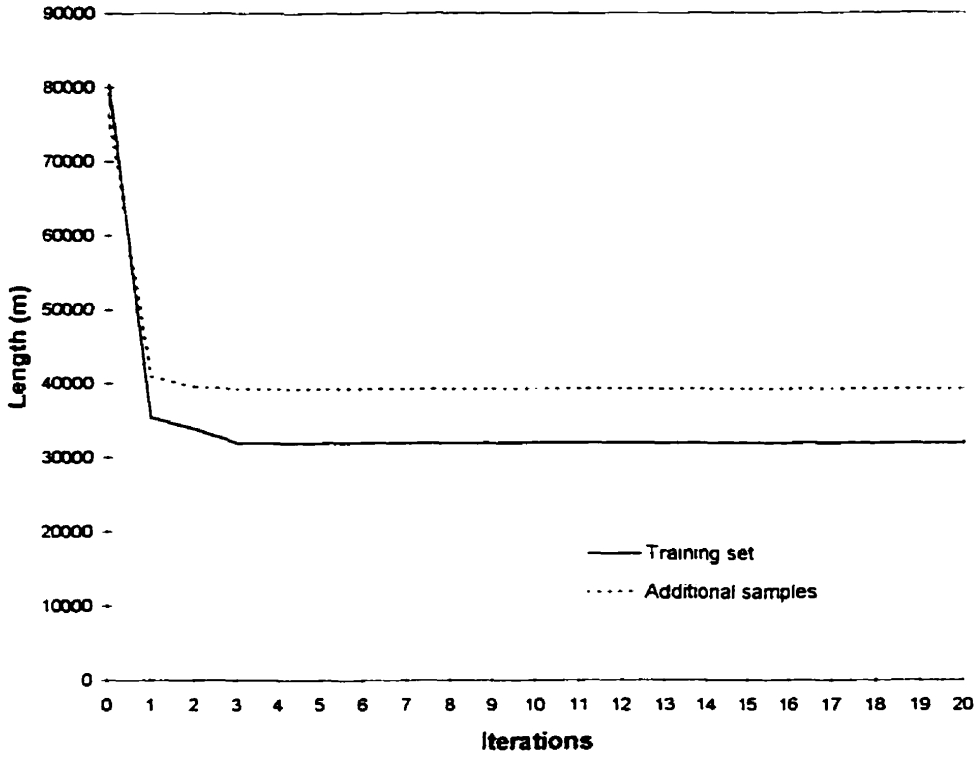


Figure 7-13: Plot of mean length with cheating enabled

7.4.6. Including a Second Agitator in the Learning Cycle

The algorithm introduced in this section uses an additional application of the agitator before the application of rules. The agitator modifies the training solutions slightly before the production system applies rules. Figure 6-4 is a flowchart of the learning cycle that this algorithm uses. The algorithms in Subsections 7.4.2 and 7.4.3 use the learning cycle in Figure 6-3. Subsections 7.4.4 and 7.4.5 introduced the "cheater" component to

the algorithm. This section introduces the additional application of the agitator, which completely transforms the learning cycle from that in Figure 6-3 to that in Figure 6-4.

Table 7-14: Results of 20 iterations of the learning cycle in Figure 6-4

		(1) Starting solution length (m)	(2) Final solution length (m)	
Training set	(1)	95961.6	31987.2	*
	(2)	70492.7	31987.2	*
	(3)	69886.8	31987.2	*
	(4)	72145.4	31987.2	*
	(5)	80922.1	31987.2	*
	(6)	90356.1	31987.2	*
	(7)	76145.0	31987.2	*
	(8)	79229.3	31987.2	*
	(9)	88223.6	31987.2	*
Samples	(1)	95961.6	31987.2	*
	(2)	85574.2	31987.2	*
	(3)	60367.3	31987.2	*
	(4)	85791.7	31987.2	*
	(5)	90267.2	31987.2	*
	(6)	88486.2	31987.2	*
	(7)	62064.9	31987.2	*
	(8)	79753.0	37767.4	
	(9)	61453.8	31987.2	*
	(10)	76187.1	31987.2	*
	(11)	76749.7	31987.2	*
	(12)	83832.4	46674.2	
	(13)	85616.9	31987.2	*
	(14)	84095.7	31987.2	*
	(15)	47634.7	37725.6	
	(16)	94918.9	31987.2	*
	(17)	55457.9	31987.2	*
	(18)	55238.1	31987.2	*

Table 7-14 shows the results of the algorithm at the end of 20 iterations. This trial uses the same set of starting solutions as the previous trial for both the training set and the additional samples. There is clearly a significant improvement in the program's ability to optimize the starting solutions in the additional sample set. Fifteen of eighteen

solutions produced by the additional samples are optimal compared to one in the previous example.

Table 7-15 shows the progress of the mean length obtained from the training set and the additional samples. Table 7-15 is similar to Table 7-13 from the previous test. Figure 7-14 is a graph of the data in column 2 and 3 of Table 7-15. The solid line in Figure 7-14 represents the mean of the solutions produced by the training set and the dashed line represents the results obtained from the additional samples.

As with most graphs, the poor quality of the solutions in iteration 0 explains the sharp decrease in both the solid and the dotted lines in Figures 7-13 and 7-14. Iteration 0 is not actually an iteration of the learning program but the results of the random generation of starting solutions through the preference and threshold method. These solutions are usually very poor solutions. A very small and relatively ineffective rule base can produce substantial improvements on these poor solutions.

In the sections of the graphs beyond iteration 1 there is an obvious difference between the graphs in Figure 7-13 and Figure 7-14. There is a large gap between the lines in Figure 7-13 while the lines in Figure 7-14 are very close. The rule base that produced Figure 7-13 is only effective on starting solutions used in the training set.

From a comparison of the results in Figures 7-13 and 7-14 the algorithm that produced the results in Figure 7-14 is more capable of optimizing starting solutions not used in the training process. The performance of the algorithm with respect to the additional samples continues to improve slightly through the later iterations. Clearly the algorithm that produced Figure 7-14 has a better ability to generalize, that is, to optimize

solutions that it did not use in training. This algorithm is also capable of sustaining the learning process over more iterations.

Table 7-15: Progress of mean length produced by learning cycle in Figure 6-4

(1) Iteration	(2) Training set mean length (m)	(3) Sample mean length (m)	(4) Number of rules
0	80373.6	76080.6	0
1	33570.7	36067.5	540
2	32778.9	33658.0	752
3	31987.2	33658.0	872
4	31987.2	33658.0	920
5	31987.2	33979.1	948
6	31987.2	33979.1	964
7	31987.2	33979.1	996
8	31987.2	33764.2	1044
9	31987.2	33764.2	1052
10	31987.2	33764.2	1068
11	31987.2	33764.2	1084
12	31987.2	33764.2	1140
13	31987.2	33764.2	1164
14	31987.2	33764.2	1172
15	31987.2	33764.2	1216
16	31987.2	33764.2	1224
17	31987.2	33764.2	1248
18	31987.2	33764.2	1256
19	31987.2	33443.1	1272
20	31987.2	33443.1	1296

The source of the improved performance of this variation of the algorithm is the number of rules the algorithm generates. According to Table 7-13 the earlier version of the algorithm generates a total of 396 rules and it generates all of these rules by iteration 3. In addition, the algorithm generates the vast majority of the rules, 276, in the first iteration. In contrast, Table 7-15 shows that the later version of the algorithm learns more rules within the first iteration (540) and the algorithm continues to learn a large

number of rules with each iteration. At the end of 20 iterations the new algorithm has learned roughly four times as many rules as the previous algorithm. The use of the agitator to modify the starting solutions slightly before each iteration offers a wider variety of solutions on which to apply the rule base, and therefore more opportunities for learning. The increased opportunities accelerate and sustain the learning process. Sustained learning accounts for the continued improvement in performance of the mean of the additional samples. The large number of rules accounts for the substantially better performance of the algorithm with respect to the additional samples.

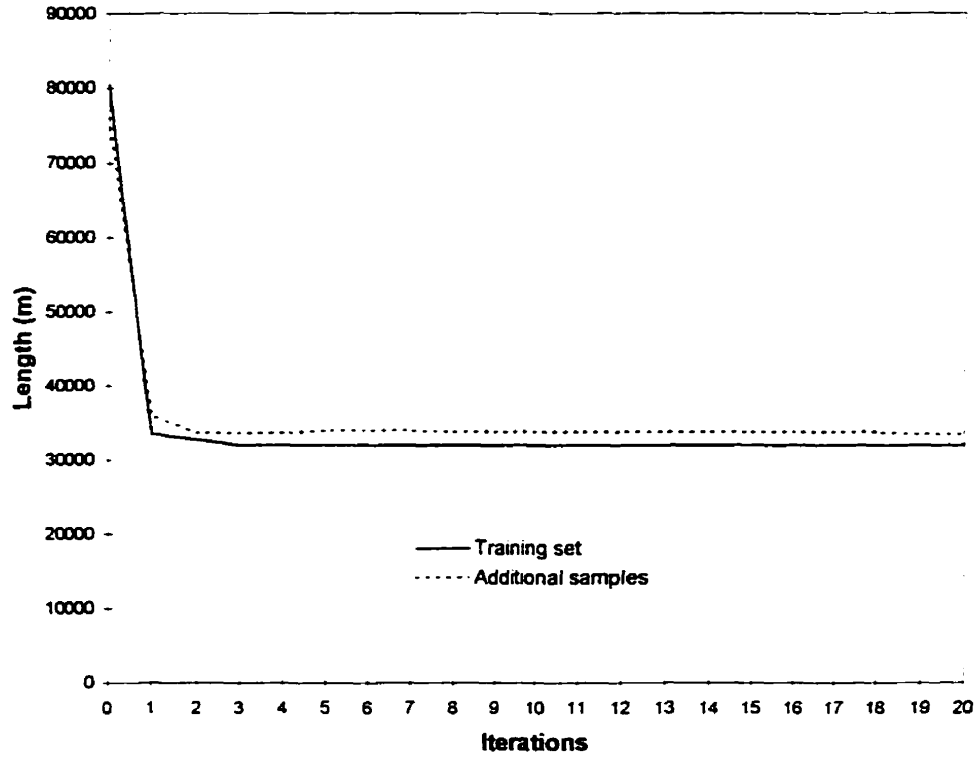


Figure 7-14: Plot of mean length produced by learning cycles in Figure 6-4

7.4.7. Experiments with Forgetting

The final component introduced in this chapter is "forgetting". The previous chapter proposes three reasons for forgetting rules.

1. The rule is no longer used.
2. The rule has produced an infeasible solution.
3. The rule has produced an increase in the total length of a solution.

The forgetting procedure introduced in this chapter consists of the elimination of rules on the basis of the first criterion only. The first criterion follows from the hypothesis that as the program acquires new rules many of the new rules may supersede old rules. A rule base that eliminates the older superseded rules will exhibit the same or better performance as a larger rule base if the hypothesis is correct.

The first test in this subsection is similar to the test in Subsection 7.4.4 in which the learning cycle uses a single agitator and the cheating process. This algorithm is capable of obtaining the optimal solution for all the starting solutions in the training set. The algorithm reaches the point at which it has fully optimized all starting solutions in the training set early. Once the algorithm reaches this point it does not acquire more rules because no more learning is possible.

The algorithm in this section includes an additional procedure that facilitates forgetting. At the start of an iteration the program writes to a file all the rules that it used at least once in the previous iteration. The program then empties the rule base

completely and recovers the rules from the file. The rule base now consists only of rules used in the previous iteration.

Table 7-16: Number of rules maintained through learning and forgetting

(1) Iteration	(2) Rules retained	(3) Total rules
0	0	0
1	0	276
2	236	320
3	236	288
4	240	240
5	240	240
6	240	240
7	240	240
8	240	240
9	240	240
10	240	240

Table 7-16 shows the results of this algorithm. Column 2 lists the number of rules retained from the previous iteration. Column 3 lists the number of rules at the end of the iteration. This number consists of the sum of the number of rules retained from the previous iteration and the number of rules learned during that iteration. According to Table 7-16 during iteration 1 the program learns 276 rules The program retains 236 of these rules in the next iteration. During each successive iteration the number of rules that the program learns (column 3 minus column 2) decreases while the number of rules the program retains increases gradually. By iteration 4 the program retains 240 rules. At this point the program does not learn any new rules. The program can solve the problem completely for each of the nine starting solutions using 240 rules and the program has eliminated all the other rules learned in the process of acquiring the 240 rules.

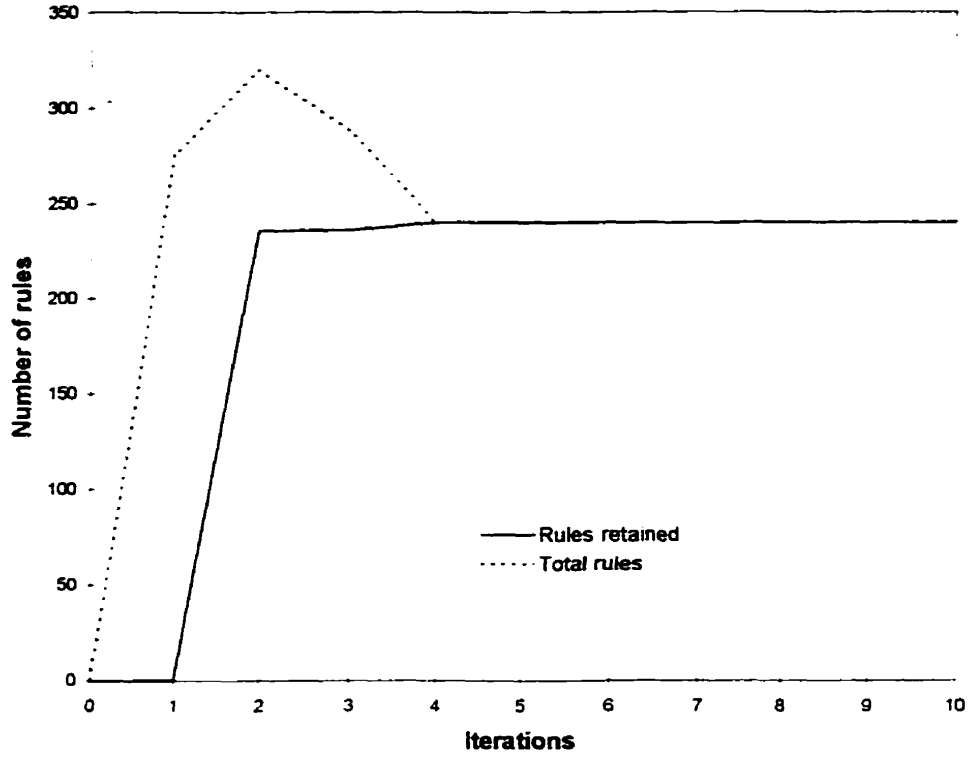


Figure 7-15: Rules maintained through learning and forgetting

Figure 7-15 is a graph of Table 7-16. The dotted line represents the total number of rules in the rule base at the end of an iteration. The solid line represents the number of rules the program has retained from the previous iteration. The gap between the two lines represents the number of rules learned during an iteration. The two lines converge by iteration 4 indicating that the program cannot improve on the solutions produced by the rules retained from the previous iteration either by the agitator or by cheating.

Table 7-17: Mean length and number of rules with forgetting

(1) Iteration	(2) Training set mean length (m)	(3) Sample mean length (m)	(4) Number of rules
0	80373.6	76080.6	0
1	33570.7	36067.5	540
2	31987.2	34411.2	582
3	31987.2	33520.3	610
4	31987.2	33121.9	594
5	31987.2	33121.9	546
6	31987.2	32803.1	554
7	31987.2	32803.1	526
8	31987.2	32803.1	510
9	31987.2	32803.1	526
10	31987.2	32803.1	542
11	31987.2	32803.1	558
12	31987.2	32803.1	510
13	31987.2	32803.1	514
14	31987.2	32803.1	510
15	31987.2	32803.1	514
16	31987.2	32803.1	558
17	31987.2	32803.1	518
18	31987.2	32803.1	522
19	31987.2	32803.1	506
20	31987.2	32803.1	510

The second test in this subsection is similar to the test in the previous subsection. The algorithm uses two agitators as in Subsection 7.4.6 and a forgetting procedure, which this subsection introduces. The forgetting procedure executes at the start of each iteration. Table 7-17 and Figure 7-16 shows that the performance of the algorithm is similar to that of the previous algorithm. However, the new algorithm does not exhibit the sustained rapid growth in the size of the rule base. Column 4 of Table 7-17 shows that the number of rules in the rule base reaches a maximum of 610 rules in iteration 3. In contrast, the values in column 4 of Table 7-15 continue to increase to a maximum of 1296 rules.

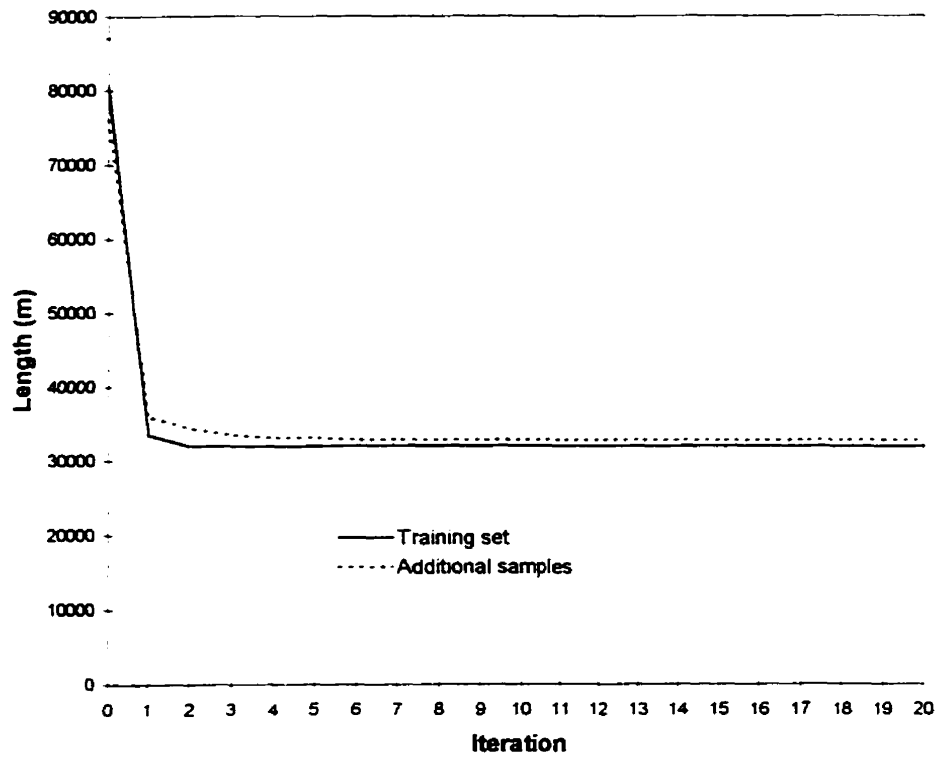


Figure 7-16: Progress of mean length with forgetting

A comparison of the relative power of the two algorithms uses the number of optimal solutions the rule base produces during each iteration. The number of optimal solutions includes those produced by both the training set and the additional samples. The algorithm used in the previous subsection produced the data in Table 7-18. The algorithm in that subsection does not use forgetting. Column 2 of Table 7-18 lists the number of optimal solutions in each iteration. Column 3 lists the number of rules. Column 4 lists a statistic that consists of the number of optimal solutions divided by the total number of rules. The statistic in column 4 represents the relative power of the rule

base per rule. Table 7-19 lists a similar set of results obtained from the algorithm that produced Table 7-17 and Figure 7-16. This algorithm includes forgetting.

Table 7-18: Number of optimal solutions per rule without forgetting

(1) Iteration	(2) Optimal solutions	(3) Number of rules	(4) Optimal solutions per rule
1	13	540	0.02407
2	22	752	0.02926
3	23	872	0.02638
4	23	920	0.02500
5	22	948	0.02321
6	22	964	0.02282
7	22	996	0.02209
8	23	1044	0.02203
9	23	1052	0.02186
10	23	1068	0.02154
11	23	1084	0.02122
12	23	1140	0.02018
13	23	1164	0.01976
14	23	1172	0.01962
15	23	1216	0.01891
16	23	1224	0.01879
17	23	1248	0.01843
18	23	1256	0.01831
19	24	1272	0.01887
20	24	1296	0.01852

The data in Table 7-19 indicate that the program produces no improvement past iteration 6. The results obtained at iteration 6 of Table 7-19 are better than those obtained in Table 7-18. However the program obtains the results in row 6 of Table 7-19 with less than half the number of rules as row 20 of Table 7-18. The example in Table 7-19 is unusual. It achieves a high level of performance earlier than expected. Normally, a learning program that uses forgetting requires more iterations to train to the equivalent

level of performance. However, even if this is the case, the algorithm that incorporates forgetting will obtain equivalent performance with fewer rules.

Table 7-19: Number of optimal solutions per rule with forgetting

(1) Iteration	(2) Optimal solutions	(3) Number of rules	(4) Optimal solutions per rule
1	13	540	0.02407
2	20	582	0.03436
3	23	610	0.03770
4	25	594	0.04209
5	25	546	0.04579
6	26	554	0.04693
7	26	526	0.04943
8	26	510	0.05098
9	26	526	0.04943
10	26	542	0.04797
11	26	558	0.04659
12	26	510	0.05098
13	26	514	0.05058
14	26	510	0.05098
15	26	514	0.05058
16	26	558	0.04659
17	26	518	0.05019
18	26	522	0.04981
19	26	506	0.05138
20	26	510	0.05098

Figure 7-17 is a graph of the data in column 4 of Table 7-18 and Table 7-19. The dotted line in Figure 7-17 represents the number of optimal solutions per rule obtained by the algorithm that uses forgetting. The solid line represents the same statistic for the algorithm that does not use forgetting. Figure 7-17 shows that initially the two algorithms are roughly equivalent. Through ensuing iterations the algorithm that uses forgetting produces roughly twice as many optimal solutions per iteration when corrected for the size of the rule base. Rule bases produced through forgetting can accomplish the same

results as those produced without forgetting using roughly half the number of rules. Fewer rules require less computer memory and shorter execution time for the production system.

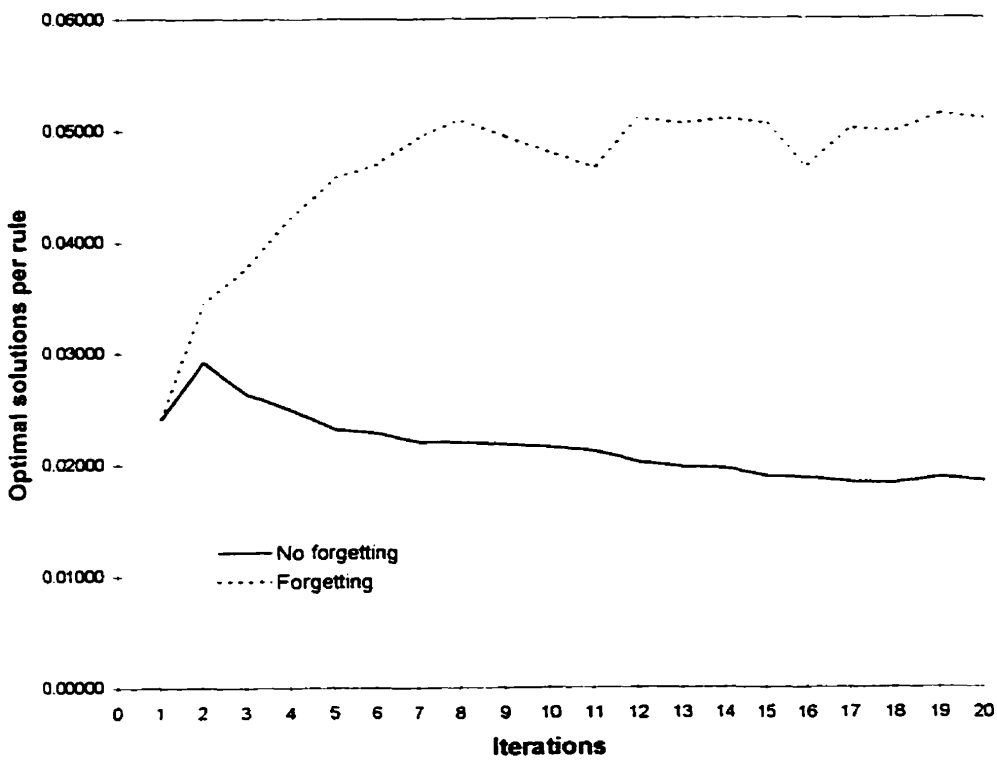


Figure 7-17: Progress of optimal solutions per rule

7.4.8. Batch Programs

Appendix D includes a source code listing of the programs used in this section and a listing of the sample problem data. The MS-DOS batch file "r1.bat" performs the tests used to create Table 7-8 and the file "r2.bat" performs all the remaining tests on the learning program.

7.5 Solution to a Sample Problem

The tests to this point have demonstrated the program on the problems used in the training process. This section demonstrates the ability of the program to solve a problem not encountered in a previous training session. Appendix D provides a 10 node sample problem referred to as "nodes04". The batch program "train.bat" creates a rule base consisting of 2810 rules. The learning program does not use the problem "nodes04" in the process of creating the rule base. The program "solve.bat" applies the rule base to 30 randomly generated starting solutions for this problem. The best solutions produced have a total length of 233193.6 m. The program produces two solutions out of 30 that have this length. Figure 7-18 is an example of one of these solutions.

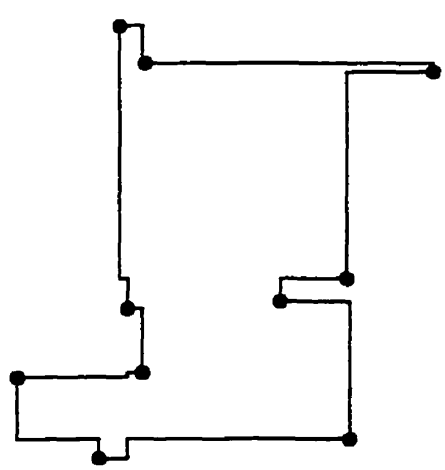
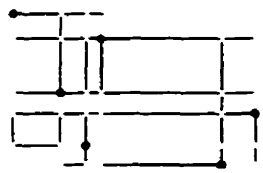
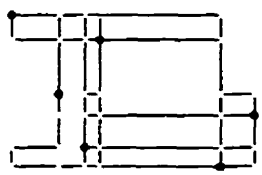
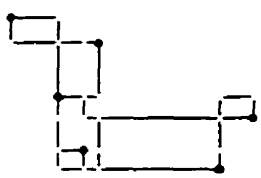
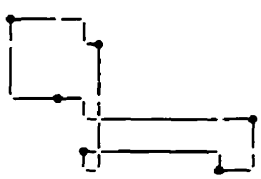
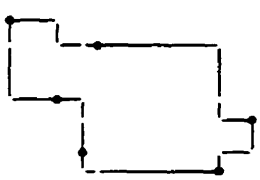


Figure 7-18: Solution to 10 node problem

7.6 Summary of results from Experiments

Table 7-20 summarizes the capabilities of the different methods based on the experiments in this chapter.

Table 7-20: Summary of methods

	Coin Toss • Generally produces infeasible solutions • Computational effort associated with generation of feasible solutions increases exponentially with problem size
	Preference and Threshold • Always produces feasible solutions • Computational effort increases with the square of the problem size • Solutions are over designed
	Evolution Program • Improvement over the preference and threshold method • Uses a population of solutions • Problem of diminishing returns
	Agitator (Evolution Strategy) • Optimizes a single solution rather than a population • Problem of diminishing returns • Sub-optimal solutions often possess jogs and unnecessary loops
	Learning Program • Program uses rules to produce complex changes • Solutions typically do not have jogs and unnecessary loops

8. Conclusions

The thesis describes and demonstrates a new method for the design of layout geometry of rectilinear looped water distribution networks. The method optimizes randomly generated solutions using a rule based production system. The rule base does not use rules derived from human expertise, but rather, the program acquires rules through a machine learning technique. The learning process autonomously derives rules from successful applications of a random search technique referred as the agitator component. The agitator component improves solutions through a method similar to the (1+1)-ES evolution strategy.

The method optimizes solutions on the basis of total length rather than cost. Optimization on the basis of cost requires a method to size the components of the network without eliminating the necessary redundancy. A computationally efficient method to perform the component sizing without eliminating redundancy is not available at the time of writing and the development of such a method represents the next logical step in this work.

The intent of the work presented in the thesis is not the optimization of actual water distribution networks. Instead, the objective of the thesis is the study of the proposed method's ability to optimize network geometry on the basis of a simpler and more readily attainable objective of minimizing length. Accomplishing the objective required the solution to several problems, which constitute the contributions made by this work. These contributions include the following:

1. A feasibility test for fully looped networks.
2. A method for generating feasible solutions (preference and threshold) for a solution space dominated by infeasible solutions.

3. Evolution program operators similar to the simple genetic algorithm operators of crossover and mutation that preserve the feasibility of resulting solutions.
4. The development of an architecture or conceptual framework for a rule based program that both acquires and evaluates its knowledge base autonomously.
5. The introduction of enhancements to the basic architecture of the program, namely cheating and forgetting, that accelerate the learning process and improve the efficiency of the rule base.

There are several directions for future research including the following:

1. Non-rectilinear systems
2. A method for sizing components that does not remove redundancy and facilitates evaluation based on cost rather than length - One possible direction appears to be a combination of Kessler et al. (1990) two tree algorithm with Dijkstra's algorithm.
3. Development of algorithms that can plan around obstacles.
4. Three dimensional variation of the problem using both looped and tree networks - Formulation of rules in three dimensions using human expertise is difficult. The program could be useful for mechanical systems in buildings such as electrical, plumbing and HVAC.
5. Further study of the process of formulation of rules. The application of the technique to a wider range of problems requires a more universal rule formulation process. In the application presented here the only additional information required in antecedent clauses was the position of the immediately adjacent links.

References

Awumah, K., Goulter, I., and Bhatt, S. K. (1991). "Entropy based redundancy measures in water distribution network design." *Journal of Hydraulic Engineering, ASCE*, 117(5), 595-614.

Alperovits, E., and Shamir, U. (1977). "Design of optimal water distribution systems." *Water Resources Research*, 13(6) 885-900.

Bern, M. W. and Graham, R. L. (1989) "The shortest network problem." *Scientific American*, 260(1), 84-89.

Bondy, J., and Murty, U. (1976). *Graph theory with applications*. North-Holland, New York.

Bouchart, F., and Goulter, I. (1991). "Reliability improvement in design of water distribution networks recognizing valve location." *Water Resources Research*, 27(12), 3029-3040.

Bratko, I. (1986). *Prolog programming for artificial intelligence*. Addison-Wesley Publishers Ltd., Wokingham, England.

Cembrowicz, R. G., and Krauter, G. E. (1977). "Optimization of urban and regional water supply systems." *Conference proceedings; Systems Approach for Development, IFAC*, Cairo, 449-454.

Cembrowicz, R. G. (1992). "Water supply systems optimization for developing countries." *Pipeline Systems*, B. Coulbeck and E. Evans, eds., Kluwer Academic Publishers, London England, 59-76.

Clocksin, W. F., and Mellish, C. S. (1984). *Programming in prolog*. Springer-Verlag, Berlin, Germany.

Cullinane, J. M., Mays, L. W., Lansey, K. E. (1992). "Optimization-availability design of water distribution networks." *Journal of Hydraulic Engineering, ASCE*, 118(3), 420-441.

Dandy, G. C., Simpson, A. R. and Murphy, L. J. (1996) "An improved genetic algorithm for pipe network optimization." *Water Resources Research*, 32(2), 449-458.

Davidson, J., and Goulter, I. (1995). "Evolution program for design of rectilinear branched networks." *Journal of Computing in Civil Engineering, ASCE*, 9(2), 1-10.

Davidson, J., and Goulter, I. (1994a). "Adaptive learning in the design of pipe layout geometry." *Complex Systems: Mechanism of Adaptation*. R. J. Stonier and X. H. Yu, eds., 2nd national conf. on complex systems, Central Queensland University, Rockhampton, Australia. 181-188.

Davidson, J., and Goulter, I. (1994b). "A new technique for design of layout of looped distribution systems." *Hydraulic engineering software V, Vol 1: Water Resources and Distribution*. W. R. Blain and K. L. Katsifarakis, eds., 5th Int. conf. on hydraulic engineering software, Wessex Institute of Tech. and Aristotle University of Thessaloniki, Porto Carras, Greece. 327-334.

Davidson, J., and Goulter, I. (1992). "Rule-based approach to layout design of rural water distribution networks." *Hydraulic engineering software IV: Fluid flow modeling*. W. R. Blain and E. Carera, eds., 4th Int. conf. on hydraulic engineering software, Wessex Univ. of Tech., Univ. of Portsmouth and Universidad Politecnica de Valencia. 267-280.

Davidson, J., and Goulter, I. (1991a). "Rule-based design of layout of rural natural gas networks." *Journal of Computing in Civil Engineering, ASCE*, 5(3), 300-314.

Davidson, J., and Goulter, I. (1991b). "Heuristic for layout design of rural gas systems." *Journal of Computing in Civil Engineering, ASCE*, 5(3), 315-332.

Davidson, J., and Goulter, I. (1989). "Micro-computer workstation for design of rural natural gas distribution systems." *Micro-Computers in Civil Engineering*, 4(1), 61-73.

Duan, N., and Mays, L. W. (1990). "Reliability analysis of pumping systems." *Journal of Hydraulic Engineering, ASCE*, 116(2), 230-248.

Duan, N,. Mays, L. W., and Lansey, K. E. (1990). "Optimal reliability-based design of water distribution networks." *Journal of Hydraulic Engineering, ASCE*, 116(2), 249-268.

Dubois, D. (1983). "A fuzzy, heuristic, interactive approach to the optimal network problem." *Advances in fuzzy sets, possibility theory and applications*. Paul P. Wang, ed., Plenum Press, New York, 253-276.

Eiger, G., Shamir, U., and Ben-Tal, A. (1994) "Optimal design of water distribution networks." *Water Resources Research*, 30(9), 2637-2646.

Fujiwara, O., DeSilva, A. U. (1990). "Algorithm for reliability-based optimal design of water networks." *Journal of Environmental Engineering, ASCE*, 116(3), 575-596.

Fujiwara, O., and Dey, D. (1988). "Method for optimal design of branched networks on flat terrain." *Journal of Environmental Engineering, ASCE*, 114(6), 1464-1475.

Fujiwara, O., and Dey, D. (1987). "Two adjacent pipe diameters at the optimal solution in the water distribution network models." *Water Resources Research*, 23(8), 1457-1460.

Fujiwara, O., and Khang, D. B. (1990). "A two-phase decomposition method for optimal design of looped water distribution networks." *Water Resources Research*, 26(4), 539-549.

Fujiwara, O., and Tung, H. D. (1991). "Reliability improvement for water distribution networks thorough increasing pipe size." *Water Resources Research*, 27(7), 1395-1405.

Garey, M. R., Graham, R. L. and Johnson, D. S. (1977). "The complexity of computing Steiner minimal trees." *SIAM Journal of Applied Mathematics*, 32(4), 835-859.

Garey, M. R., and Johnson, D. S. (1977). "The rectilinear Steiner tree problem is NP-complete." *SIAM Journal of Applied Mathematics*, 32(4), 826-834.

Gessler, J. (1982). "Optimization of pipe networks." *Proceedings of the Ninth International Symposium on Urban Hydrology, Hydraulics and Sediment Control*, University of Kentucky, Lexington.

Goldberg, D. E. (1989). *Genetic algorithms in search, optimization and machine learning*. Addison-Wesley Publishing Co., Reading, Massachusetts.

Goldberg, D. E., and Kuo, C. H. (1987). "Genetic algorithms in pipeline optimization." *Journal of Computing in Civil Engineering, ASCE*, 1(2), 128-141.

Goulter, I. C., and Bouchart, F. (1990). "Reliability-constrained pipe network model." *Journal of Hydraulic Engineering, ASCE*, 116(2), 211-229.

Goulter, I. C., and Bouchart, F. (1987). "Joint consideration of pipe breakage and pipe flow probabilities." *Proceedings of 1987 National Conference on Hydraulic Engineering*, Williamsburg, Virginia, 469-474.

Goulter, I. C., and Coals, A. V. (1986). "Quantitative approaches to reliability in pipe networks." *Journal of Transportation Engineering, ASCE*, 112(3), 287-301.

Goulter, I. C., and Morgan, D. R. (1984). Discussion of "Layout of water distribution systems" by W. F. Rowell and J. W. Barnes, *Journal of Hydraulics Division, ASCE*, 109(HY1), 67-68.

Hanan, M.(1966). "On Steiner's problem with rectilinear distance." *SIAM Journal of Applied Mathematics*, 14(2), 255-265.

Holland, J. H. (1975). *Adaptation in natural and artificial systems*. University of Michigan Press, Ann Arbor, Michigan.

Itai, A., and Rodeh, M. (1984). "The multi-tree approach to reliability in distributed networks." *Found. of Computer Science, Proceedings of the 25th Symposium*, 137-147.

Jacobs, P., and Goulter, I. (1991). "Estimation of maximum cut-set size for water network failure." *Journal of Water Resources Planning and Management, ASCE*, 117(5), 588-605.

Jeppson, R. W. (1976). *Analysis of flow in pipe networks*. Ann Arbor Science Publishers Inc., Ann Arbor, Michigan.

Karmeli, D., Gadish, Y., and Meyers, S. (1968). "Design of optimal water distribution networks." *Journal of the Pipeline Division, ASCE*, 94(PL1), 1-11.

Kessler, A., Ormsbee, L., and Shamir, U. (1990). A methodology for least-cost design of invulnerable water distribution networks." *Civil Engineering Systems*, 1, 20-28.

Kruskal, J. B. Jr. (1956). "On the shortest spanning subtree of a graph and the traveling salesman problem." *Proc. Amer Math. Soc.*, 7, 48-50.

Labye, Y., (1966). "Etudes des procédés de calcul ayant pour but de rendre minimal le coût d'un réseau de distribution d'eau sous pression." *La Houille Blanche*, 5.

Lai, D., and Schaake, J. (1969). "Linear programming and dynamic programming applications to water distribution network design." *Report 116*, Department of Civil Engineering, Massachusetts Institute of Technology, Cambridge, Massachusetts.

Landson, L. S., Warren, A. D., and Ratner, M. S. (1984) *GRG2 user's guide*. University of Texas, Austin, Texas.

Liebman, J. C. (1967). "A heuristic aid for the design of sewer networks." *Journal of the Sanitary Engineering Division, ASCE*, 93(SA4), 81-90.

Michalewicz, Z. (1992). *Genetic algorithms + data structures = evolution programs*. Springer-Verlag, Berlin, Germany.

Morgan, D. R., and Goulter, I. C. (1985). "Optimal urban water distribution design." *Water Resources Research*, 21(5), 642-652.

Ostfeld, A., and Shamir, U. (1993). "Incorporating reliability in optimal design of water distribution networks - review and new concepts." *Reliability Engineering and System Safety*, 42, 5-11.

Pearl, J. (1984). *Heuristics: intelligent search strategies for computer problem solving*. Addision-Wesley Publishing Co., Reading, Massachusetts.

Quindry, G., Brill, E. D., and Liebman, J. (1981). "Optimization of looped water distribution systems." *Journal of the Environmental Engineering Division, ASCE*, 107(EE4), 665-679.

Quindry, G., Brill, E. D., Liebman, J., and Robinson, A. (1979). Comments on "Design of optimal water distribution systems" by E. Alperovits and U. Shamir. *Water Resources Research*, 15(6) 1651-1654.

Rechenberg, I. (1973). *Evolutionsstratgie*. Friedrich Frommann Verlag, Stutgart, Germany.

Rechenberg, I. (1965). "Cybernetic solution path of an experimental problem." *Library translation No. 1122*, Royal Aircraft Establishment, Library Translation 1122, Farnborough, England.

Rich, E. (1983). *Artificial intelligence*. McGraw-Hill Book Co., New York.

Rowell, W. F., and Barnes, J. W. (1982) "Obtaining layout of water distribution systems." *Journal of the Hydraulics Division, ASCE*, 108(HY1), 137-148.

Savic, D. A., and Walters, G.A. (1977) "Genetic algorithms for least-cost design of water distribution networks." *Journal of Water Resources Planning and Management, ASCE*, 123(2), 67-77.

Schwefel, H. (1977). *Numerische optimierung von computer-modellen mittels der evolutionsstrategie*. Birkhauser Verlag, Basil, Switzerland.

Simpson, A. R., Dandy, G. C., and Murphy, L. J. (1994). "Genetic algorithms compared with other techniques for pipe optimization." *Journal of Water Resources Planning and Management, ASCE*, 120(4), 423-443.

Simpson, A. R., and Goldberg, D. E. (1994). "Pipeline optimization via genetic algorithms: from theory to practice." *Proceedings of the Second International Conference on Water Pipeline Systems*, Edinburgh, Scotland, 309-320

Su, Y. C., Mays, L. W., Duan, N., and Lansey, K. E. (1987). "Optimal reliability-based design of pumping and distribution systems." *Journal of Hydraulic Engineering, ASCE*, 113(12), 1539-1555.

Tanyimboh, T. T., and Templeman, A. B. (1994). Discussion of "Redundancy-constrained minimum-cost design of water distribution nets" by H. Park and J. C. Liebman. *Journal of Water Resources Planning and Management, ASCE*, 120(4), 568-569.

Tanyimboh, T. T., and Templeman, A. B. (1993). "Calculating maximum entropy flows in networks." *Journal of the Operational Research Society*, 44(4), 383-396.

Templeman, A. B. (1982). Discussion of "Optimization of looped water distribution systems" by G. E. Quindry, E. D. Brill, and J. C. Liebman. *Journal of the Environmental Engineering Division, ASCE*, 108(EE3), 599-602.

Thurston, W. P., and Weeks, J. R. (1984). "The mathematics of three-dimensional manifolds." *Scientific American*, 251(1), 108-120.

Tung, Y. (1986). "Evaluation of water distribution network reliability." *Water Forum '86: World Water Issues in Evolution, ASCE*, Vol. 2, 359-364.

Tung, Y., Lansey, K., and Mays, L. W. (1987) "Water distribution system design by chance constrained model." *Proceedings of 1987 National Conference on Hydraulic Engineering*, Williamsburg, Virginia, 588-593.

Wagner, J. M., Shamir, U., and Marks, D. H. (1988a). "Water distribution reliability: analytical methods." *Journal of Water Resources Planning and Management, ASCE*, 114(3), 253-275.

Wagner, J. M., Shamir, U., and Marks, D. H. (1988b). "Water distribution reliability: simulation methods." *Journal of Water Resources Planning and Management, ASCE*, 114(3), 276-294.

Walters, G. A., and Cembrowicz, R. G. (1993). "Optimal design of water distribution networks.", *Water Supply Systems: State of the Art and Future Trends*, E. Cabrera and F. Martinez, Computational Mechanics, Southampton, UK, 100-135.

Walters, G. A. (1985). "The design of the optimal layout for a sewer network." *Engineering Optimization*, 9(1), 37-50.

Walters, G. A., and Lohbeck, T. (1993). "Optimal layout of tree networks using genetic algorithms." *Engineering optimization*, 22(1), 27-48.

Walters, G. A., and Smith, D. K. (1995). "Evolutionary design algorithm for optimal layout of tree networks." *Engineering optimization*, 24, 261-281.

Wood, D. J. (1980). *User's manual computer analysis of flow in pipe networks inciuding extended period simulations*. University of Kentucky, Lexington, Kentucky.

Appendix A: The Production System

A.1. State Space Representation Scheme

This section explains the representation scheme and terminology used in Appendix A. The production system used by the learning program consists of three components:

1. Solution objects - An object exists in one of a very large number of potential states. Each state represents a possible solution to the problem. Transforming the data contained in the object changes the state of the object.
2. The rule base - The rule base is a set of production rules in "if ... then" format. Each production rule is a small program that alters the data in the object thereby changing the state of the object. A rule cannot execute unless data contained in the object match the pattern specified in the "if" or antecedent clause of the rule. If the data match the antecedent clause the procedures contained in the consequent clause can operate on the object.
3. The control strategy - The control strategy determines the order in which production rules apply and resolves conflicts between production rules.

State space graphs represent the state transformations that a solution object undergoes. The term "complete state space" refers to a representation of all states and state transformations that a rule base can produce including those transformations that the conflict resolution mechanisms of the control strategy eliminate. An actual graphical representation of a complete state space graph is usually infeasible due to the enormous number of states and state transformations. However, from a theoretical standpoint the concept of a complete state space graph is useful.

This appendix presents a slightly modified version of the conventional scheme for representing these graphs. Typically a circle represents an object in any given state and an arrow pointing from one state (circle) to another indicates a rule, representing the possible transformation of an object from one state to another. In the modified representation scheme the total length of the solution determines its position on the vertical axis of the figure. Solutions with greater total length appear at a higher location on the graph. The horizontal axis of the figures provides clarity only and has no special significance.

A.2. The Control Strategy

A production system uses rules to create solution states from an initial solution state. The states that exist in a complete state space graph depend on the rules in the rule base and the choice of the initial state. The learning program uses an approach that differs from the traditional approach to production systems used for optimization problems. In the traditional approach the rule base contains a sufficient number of rules to enable generation of all possible solutions to any problem. The use of an all-inclusive state space graph theoretically eliminates dependence on the choice of the initial state. In theory the production system can derive all possible states from any initial state. However, producing every solution is usually not possible in practice. A control strategy that performs a thorough search of an all-inclusive state space graph eventually produces the optimal solution, but usually complete searches are possible in only the most trivial of cases. The traditional approach overcomes the difficulties of searching enormous state space graphs by the use of elaborate control strategies. These control strategies are algorithms that focus on the most promising subsets of the state space graph rather than examining the entire space.

The learning program uses a different approach based on a simple control strategy, an irrevocable best-first search. The control strategy operates on a rule base that changes over time in response to discoveries made through autonomous learning.

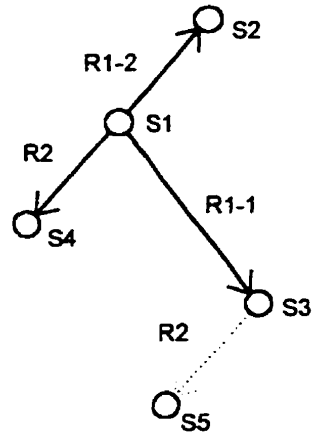


Figure A-1: Best-first control strategy

Figure A-1 illustrates how the control strategy applies rules and directs the search. In Figure A-1 State S1 is the first state created. As explained in Chapter 6 random generation creates State S1. State S1 satisfies the antecedent clauses of the two rules shown in Figure A-1 as Rule R1 and Rule R2. Rule R1 applies at two separate locations in the solution layout represented by State S1. R1-1 and R1-2 designate the two instances of Rule R1 in Figure A-1. Each of these rules when applied generates a unique state shown as States S2, S3 and S4. (The explanation presented here does not

consider State S5 at this point.) The best of the newly generated solutions is State S3, the state that is lowest on the graph. The control strategy chooses State S3 as the new current state in a decision that it cannot revoke at a later time.

There may be many more rules in the rule base than the two shown in Figure A-1. All rules do not necessarily apply to a state. The control strategy applies only those rules that have antecedent clauses that match the data represented in the current state. When a rule applies to the current state the rule has "instantiated". A rule instantiates more than once if the antecedent clause matches the data in more than one location. Rule R1 instantiates twice in Figure A-1.

The choice made in Figure A-1 illustrates that the control strategy bases the selection of the next current state purely on the immediate short term gains offered by the rules that have instantiated. To compare the rules that instantiate the control strategy assigns a "priority" value to each rule derived from the anticipated effect of the rule. However, this is possible only because length serves as a surrogate measure for cost. The control strategy can determine the effect of a rule on the total length of the solution without actually applying the rule. The control strategy calculates the priority of a rule by summing the length of the links to be removed and subtracting the lengths of the links added. A rule that instantiates with a positive priority improves the network similar to instances R1-1 and R2 in Figure A-1. A rule that instantiates with a negative priority makes the network worse as illustrated by instance R1-2. A rule instantiating with a priority of zero has no effect on the total length of the network.

As stated previously, the program can asses the effect of a rule on total length *a priori* without applying the rule. However, the program must apply a rule to assess the

effect of the rule on the feasibility of a solution. The control strategy will not necessarily apply every rule that instantiates. Therefore, it does not necessarily test the feasibility of each resulting state. The only states that undergo feasibility testing are those that result from rules the control strategy actually applies. The control may not apply a rule that has instantiated for one of two reasons. The control strategy does not apply a rule if the rule does not produce an improved solution as indicated by a negative or zero priority. Rules of higher priority apply first. A rule will not function as anticipated if a rule of higher priority changes the network such that the antecedent clause of the rule longer matches the solution data. The control strategy does not apply any rule if the rule conflicts with another rule of higher priority. Therefore the control strategy does not test the feasibility of the state produced by the lower priority rule. In this case the test is neither necessary nor possible.

The control strategy chooses the next rule by grouping all the rule instantiations and priorities in a sortable array called the "trigger set". The control strategy creates a record, referred to as an instantiation record, for each instantiation of a rule. Each instantiation record consists of a pointer to the rule in the rule base, the row and column in the Hanan grid where the rule instantiated and the priority based on the anticipated effect of the rule on the total length. The control strategy sorts the array on the basis of the priority values and disregards all the records for rule instantiations with priorities of zero or less.

Each rule in the rule base contains three counters that maintain totals describing the rule's history of performance. The forgetting process uses the rule histories. The following list provides the names and functions of each of the three counters.

1. Used - counts the number of times the rule applies successfully.
2. Dumped - counts the number of times the rule instantiates at a priority of zero or less causing its removal from the trigger set.
3. Failed - counts the number of times the rule produces an infeasible solution.

The control strategy fires the instantiation records in the trigger set with positive priority values in order of priority. The control strategy tests the antecedent clause again at the row and column position recorded in the instantiation record before it fires the each rule. If the antecedent clause does not match the solution data during the second test the network layout has changed since the rule instantiated. The rule no longer applies. In this case the rule does not fire and the counters do not change.

The rule fires if the second application of the antecedent clause of the rule succeeds. The rule alters the pattern of links at the location specified in the instantiation record to match the pattern dictated by its consequent clause. The control strategy tests the resulting layout to confirm that the solution is feasible. If the result is not feasible the control strategy retracts the rule by changing the pattern of links in the layout back to the pattern that existed before the execution of the consequent clause. The control strategy then increments the "failed" counter for the rule.

A.3. An Example Application of the Control Strategy

This section presents the example in Figure A-1 again to illustrate a single iteration of the control strategy. Figure A-2 depicts the rule base and trigger set. Figure A-2 shows only three of the rules in the rule base. Initially the trigger set is empty. The first step in the application of rules compares every rule in the rule base with solution State S1 to find every possible instantiation. The control strategy compares the pattern

in the antecedent clauses with the patterns in State S1. It conducts the comparison at every row and column position possible. An instantiation occurs when the pattern in the antecedent clause matches. Two of the three rules instantiate, Rule R1 and Rule R2. Rule R1 instantiates twice. The control strategy creates the instantiation records and enters them in the trigger set in the order in which the instantiations occur as shown in Figure A-2. Both the order of the rules in the rule base and the row column order of the solution matrices determine the order of occurrence. Each instantiation record records the row and column in the Hanan grid where the instantiation occurred, along with a pointer to the rule that instantiated and the calculated priority.

Next step sorts the instantiation records on the basis of priority. The order of the trigger set in Figure A-3 results from the sorting process.

The control strategy fires the instantiation of Rule R1-1 in preference of Rule R2 because of the greater reduction in total length. Both the priority values of the records and their order in the trigger set in Figure A-3 reflect the preference for Rule R1-1. The control strategy applies Rule R1-1 producing State S3 and then tests the feasibility of the new state. During this process Rule R1-1 must re-instantiate at the location, row 2 column 1, recorded in the instantiation record. The rule necessarily re-instantiates in this case because the solution State S1 (Figure A-1) has not changed yet. This is the first rule to fire.

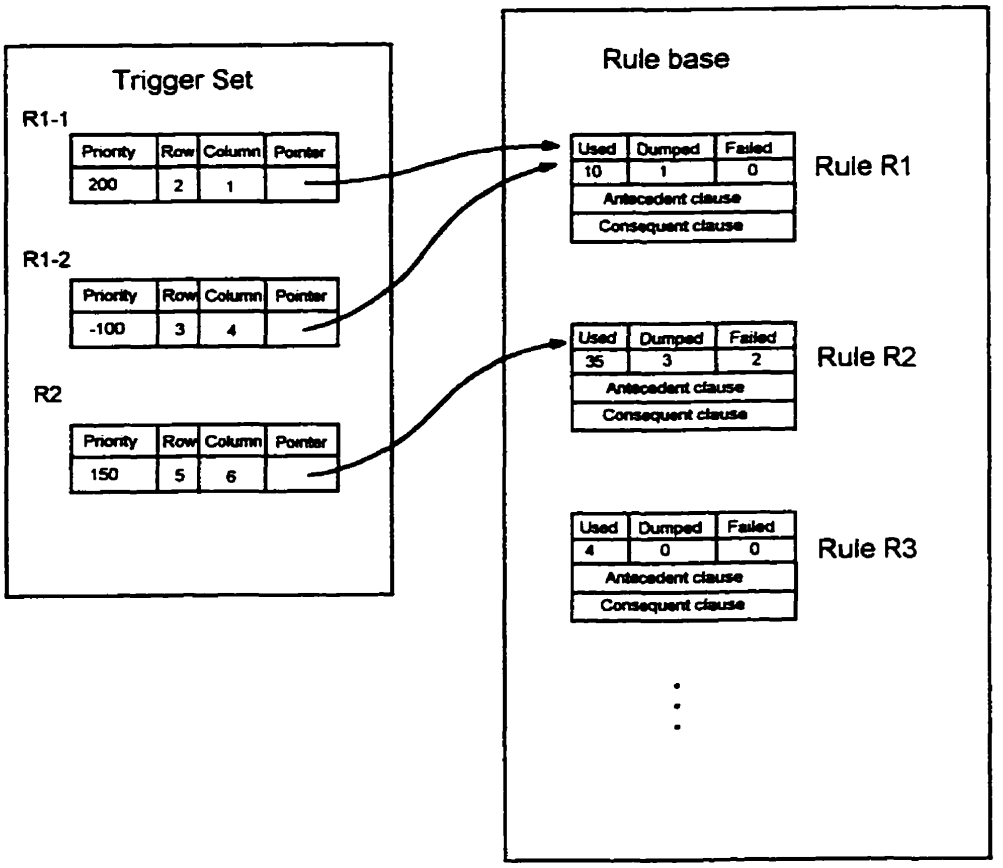


Figure A-2: Rule base and unsorted trigger set

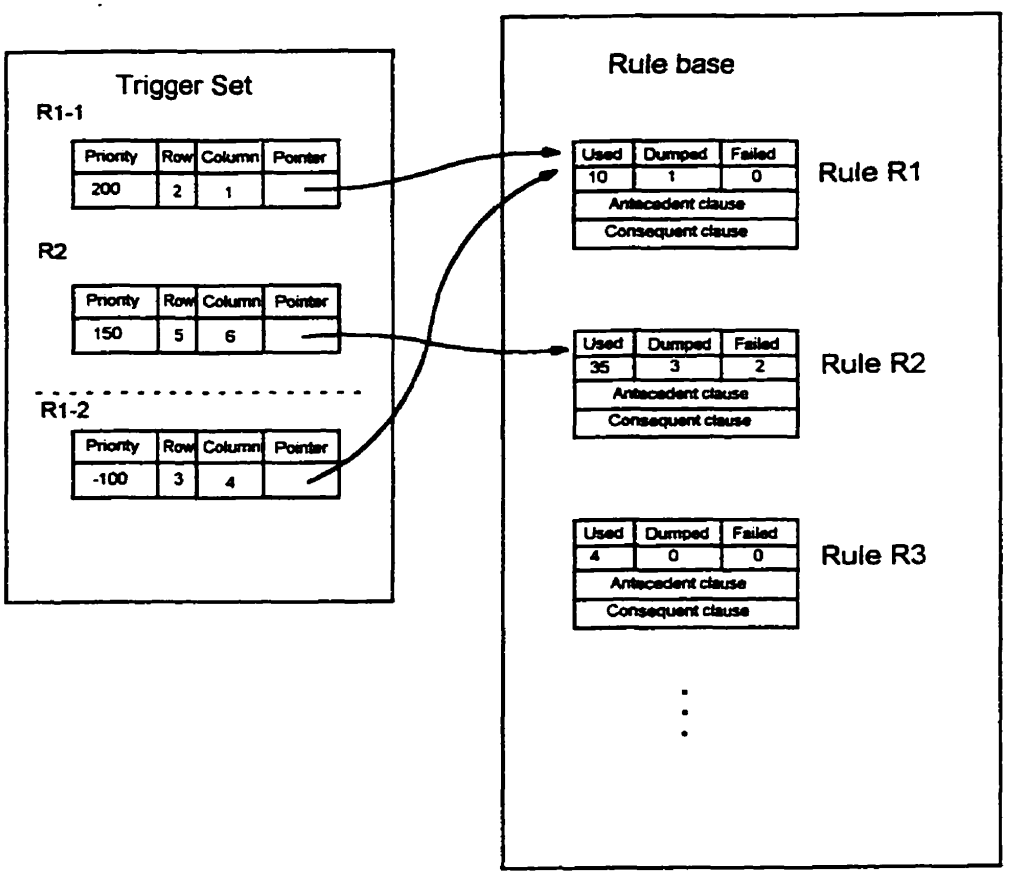


Figure A-3: Rule base and sorted trigger set

Rule R1-1 transforms state S1 to State S3. Regardless of whether State S3 is feasible or infeasible the control strategy increments the "used" counter for Rule R1 from 10 to 11. Then the control strategy tests the antecedent clause of Rule R2 at row 5 column 6 to determine if the rule still matches the layout. If the rule does not re-instantiate the control strategy does not apply the rule. This iteration of applying rules terminates because no more records with positive priority remain in the trigger set. If Rule R2 succeeds in re-instantiation, the rule applies transforming State S3 to State S5 shown in Figure A-1 at the end of the dotted arrow. The control strategy increments the "used" counter for Rule R2 from 35 to 36 and then tests the feasibility of State S3. State S5 becomes the new current state at the end of the iteration if the state is feasible. If State S5 is infeasible the control strategy increments the "failed" counter for Rule R2 from 2 to 3. Then it retracts the application of the rule leaving State S3 as the new current state at the end of the iteration.

If State S3 proves to be infeasible after the application of Rule R1-1 the control strategy increments the "failed" counter for Rule R1 from 0 to 1 and retracts Rule R1. The control strategy applies Rule R2 to State S1 rather than State S3 in this case. Rule R2 re-instantiates successfully because State S1 is the state where all the initial instantiations occurred. The control strategy tests the feasibility of State S4 and State S4 becomes the new current state at the end of the iteration if the state proves to be feasible. If State S4 is not feasible the control strategy retracts Rule R2 and the current state returns to State S1, the initial state.

Once the control strategy has attempted to apply all the instantiation records with positive priority values the remaining instantiation records consist of rules that do not improve the solution. The control strategy does not apply these rules but it must

increment the "dumped" counter for each rule. The instantiation record for Rule R1-2 has a non-positive priority and the control strategy increments the "dumped" counter for Rule R1 from 1 to 2. Once the control strategy updates all the dumped counters the iteration is complete. The control strategy empties the trigger set of all instantiation records at the end of the iteration.

At the end of an iteration the current state may be unchanged from the state that occurs at the beginning of the iteration. The lack of a change indicates that rule base was unable to produce a feasible improvement on the solution. Without a change in the contents of the rule base any continued iterations will produce the same result. The best-first strategy has reached a local optimum and the application of rules ceases.

A.4. Learning from a State Space Perspective

Best-first search techniques are notorious for terminating at local optima. Random manipulation is both the means of escaping local optima and the mechanism by which the learning program acquires new rules.

Figure A-4a represents the results of best-first application of three rules (Rules R1, R2 and R3) by the method described in the previous section. For simplicity the only rules shown in Figure A-4 are those that applied successfully. Unsuccessful attempts to apply rules include all rules that instantiate at a priority of zero or lower, rules that fail to re-instantiate or rules that the control strategy retracts because of infeasible results.

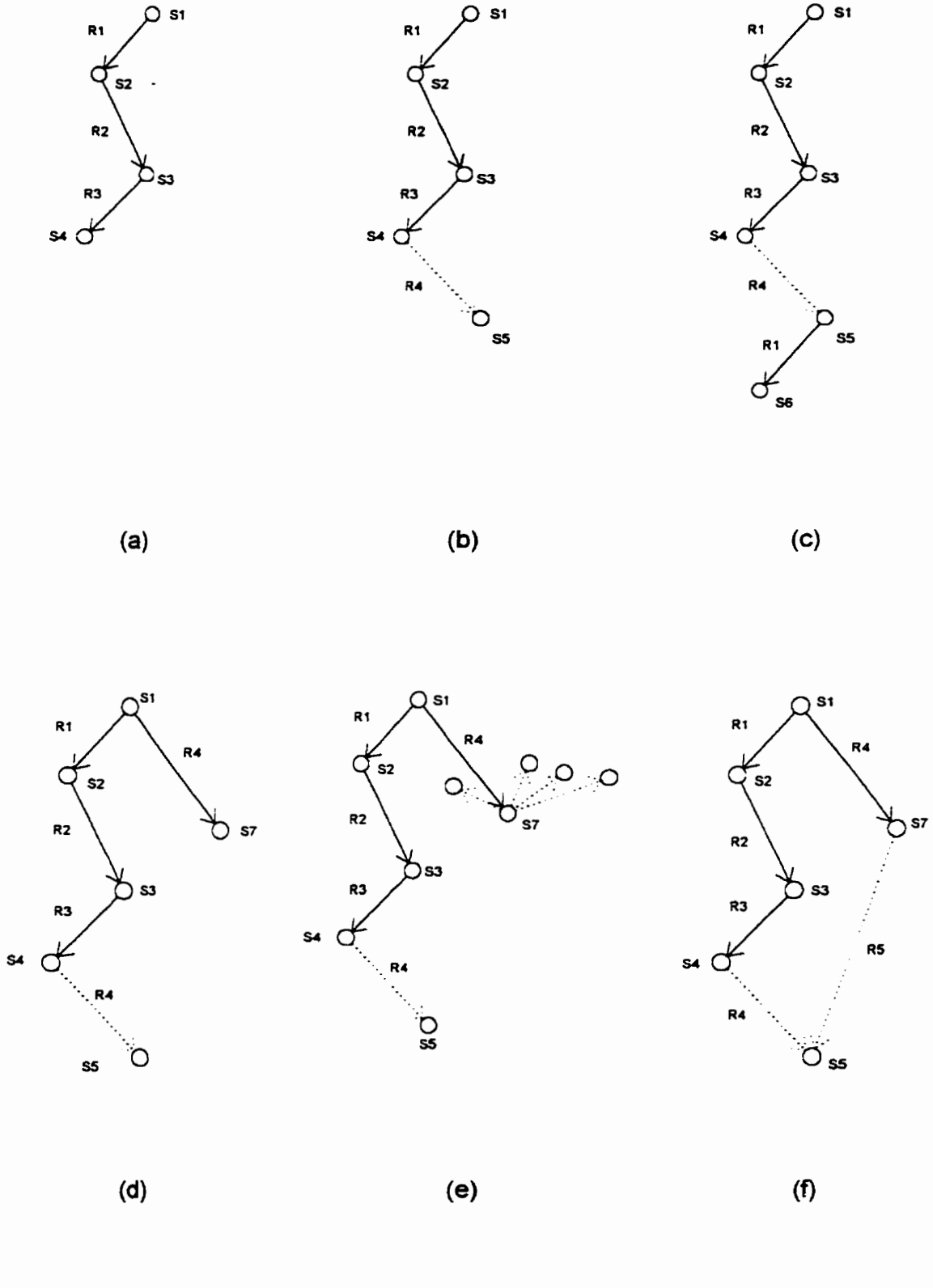


Figure A-4: Learning and cheating

At the last state in Figure A-4a, State S4, none of the rules contained in the rule base can produce a feasible improvement. Therefore, State S4 represents a local optimum. At this point the learning program invokes the agitator to iteratively alter the data in the object at State S4 to produce new states. The agitator evaluates each new solution state and the first solution that represents an improved state becomes the new current state. Figure A-4b shows this new solution state as State S5. The rule formulator compares the two solution states, States S4 and S5, to derive a new rule, Rule R4. The new rule is capable of transforming State S4 to State S5.

At the final state in Figure A-4b, State S5, rules that could not produce feasible improved solutions from State S4 may now be capable of producing improved solutions. These rules may now be able to instantiate because the layout has changed. Figure A-4c represents an example in which Rule R1, which did not instantiate at State S4, now produces an improved state, State S6, from State S5. However, the graph in Figure A-4c does not represent the actual method the control strategy uses. The control strategy does not continue to apply rules after it acquires a new rule as in Figure A-4c. The sequence actually proceeds from Figure A-4b directly to Figure A-4d. Figure A-4c represents a possible alternative approach to the control strategy. The thesis has not explored this approach.

Once the rule base acquires a new rule re-establishing a path between the initial state and the local optimum that generated the rule might not be possible. Rule R4 might intervene by instantiating at the highest priority at an intermediate state. If Rule R4 has its antecedent clause satisfied at any state along the path between S1 and S4 the rule could produce a greater immediate improvement than the rule previously applied. In this case the control strategy selects the newly added rule instead of the previous rule.

Figure A-4d is an example of a new rule that alters the behavior of the production system. In Figure A-4d the control strategy applies the rules to the same starting solution, State S1. The new rule, Rule R4 instantiates at State S1 with a greater immediately realizable improvement than Rule R1. State S7 now becomes the new current state.

If the situation illustrated in Figure A-4d occurs there are many possible outcomes. More rules may apply at State S7 possibly leading to a solution better than State S6. However, there is no guarantee that the application of the new rule, Rule R4 at State S1, results in a better solution than the previous iteration produced. For example the application of rules could terminate at State S7 leaving that state as a local optimum.

A.5. Cheating from a State Space Perspective

The addition of new rules to the rule base alters the state space graph thereby changing the location of local optima. New rules could create even more undesirable local optima than existed previously. Performance of the rule base generally improves with the addition of new rules but the improvement is not necessarily monotonic. The cheating mechanism enables the algorithm to avoid becoming trapped in newly created inferior optima. Cheating has the additional desirable effect of accelerating the process of learning complex rules discussed in Chapter 6.

Learning starts with an empty rule base. The first solution state starts from random generation and the first rule that the program learns results from random manipulation of the first state. In the earliest stages of learning when the rule base is smallest, random manipulation accomplishes the objective of producing an improved solution from a local optimum with relative ease. The local optima produced by the

production system during the early stages of learning are usually very poor solutions and the success rate for improvement through random manipulation is very high. The rule base increases in size and the production system becomes more competent as learning progresses. Producing improved solutions to local optima through random manipulation becomes more difficult. The preceding chapters refer to the problem as the problem of diminishing returns.

Figure A-4e illustrates a situation in which many iterations of random manipulation fail to produce an improved solution. In Figure A-4e the dotted arrows radiating from State S7 represent solutions produced through random manipulation. However, the algorithm maintains a copy of the best solution encountered since the search began. The algorithm can simply jump to this solution after a predetermined number of failed iterations. Figure A-4f illustrates the jump. The control strategy treats the jump exactly as if random manipulation produced the new state, State S5. The rule formulator derives a rule that can transform State S7 to State S5. Chapter 6 introduced the process of cheating and Figure 6-4 illustrates a different representation scheme for the process.

Cheating establishes pathways from local optima to better solutions that the program has already found. The concept of returning to previous solutions may appear to be a counter-productive feature of an optimization technique. The objective of the learning program is not simply to find an optimal solution to one problem. The approach regards each problem as an opportunity to explore the solution space of the problem domain. The goal is to arrive at a set of rules that works for the general class of problems in conjunction with the simple irrevocable best-first control strategy. Once the

program acquires a rule, the rule is available for application to other related problems. The objective of cheating is not to obtain new solutions, but rather, to obtain new rules.

With each iteration the initial state does not remain constant as might be concluded from Figure A-4d through A-4f. The initial state varies periodically to improve the ability of the production system to solve the problem from a variety of starting points. The learning program uses both random generation and random manipulation to produce new initial starting states. The program uses random generation periodically to produce entirely new starting states. More frequently the program applies random manipulation to an initial state to create small variations on one or more initial starting states as illustrated by Figure 6-4.

Appendix B: The Use of Objects in the Learning Program

B.1. Introduction to Learning Program Objects

In this appendix the term "object" is used in a different context than Appendix A. Appendix A uses the term "object" in the context of production systems. In that context an "object" designates a potential solution to the layout problem. Appendix B uses the term object in the context of object oriented programming. Objects in this context refer to a wider variety data structures that perform many functions within the program.

Figure 6-4 in Chapter 6 shows the structure of the learning program. The learning program uses objects to accomplish the tasks illustrated in Figure 6-4. The elements of Figure 6-4 are the solution generator, the agitator, the production system, the cheater and the rule formulator. Each of these elements represents a procedure and not an object as such. Figure B-1 shows the major objects the program uses. The procedures in Figure 6-4 result from the member functions of the objects in Figure B-1 and the interactions between those objects. This appendix describes the objects in Figure B-1 in detail. The appendix traces through the procedures illustrated in Figure 6-4 and explains the role of the objects in accomplishing these procedures. The first half of the appendix explains the processes from the solution generator through the agitator, production system, second agitator and cheater. The second half of the appendix explains the rule formulation process and specialized objects that facilitate rule formulation.

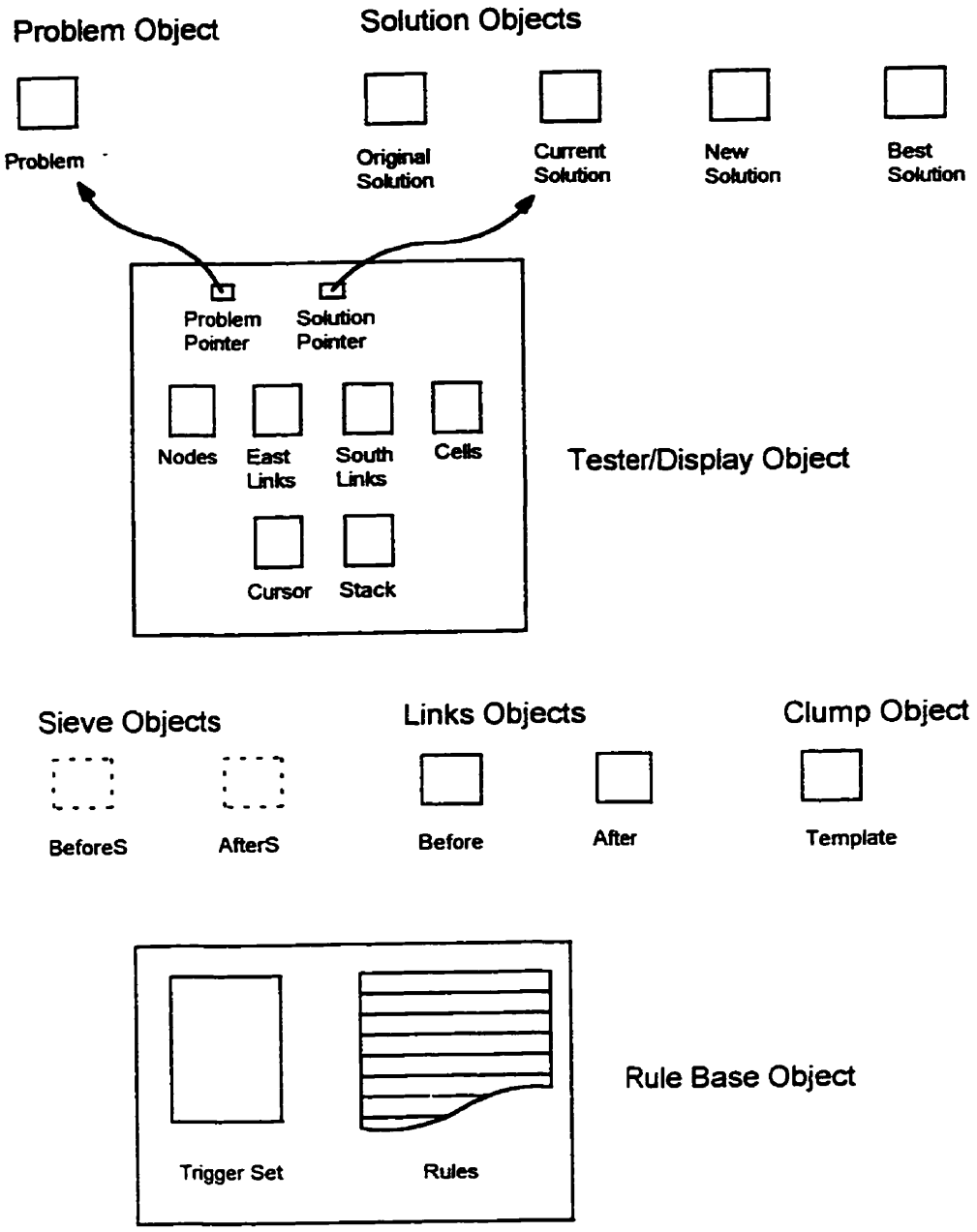


Figure B-1: Objects of the learning program

The tester/display object is the central object of the program. The tester/display object plays an important role in all operations performed by the learning program. There is only one instance of this class of object. As the name implies the tester/display object performs the feasibility testing and evaluation of solutions and controls the graphical display of solutions. The tester/display object is a container object that contains a copy of the problem data and a copy of one of several solution objects that the program maintains at any time. The copy of problem data consists of a nodes matrix and the solution data copy consists of the east and south links matrices as well as a cells matrix. The tester/display object uses copies of these data because the feasibility test is a destructive test. The tester/display object would destroy the only copy of critical data when the object performs a feasibility test if the tester/display object stored the original problem and solution data rather than copies. Instead the program maintains the problem and solution data in separate objects outside the tester/display object and the tester/display object accesses the original data through pointers to the problem data object and the solution data object.

The program currently operates on one problem at a time. In its current configuration the program requires only one instance of the problem (nodes) data. However the program maintains several solution objects. These solution objects contain matrices of links data assigned preference values between zero and one, rather than binary values. Each solution object also contains a threshold value and the total length of the solution. In contrast the links data contained in the tester/display object are in the binary format shown in Figure 2-5, which is the format the feasibility testing and evaluation routines require. The program does not copy the actual preference values of the links when it copies data from a solution object to the tester/display object. If a link in

a solution object has a preference value above the threshold value, the program assigns the link a binary value of 1 in the tester/display object. Conversely, if the link has a preference value below the threshold value, the program assigns a binary value of 0 in the tester/display object. Thus the copying procedure performs the necessary conversion of the links data from the non-binary preference scheme to the required binary representation. The program sets the solution pointer to point to a solution object when it copies the contents of that solution object to the tester/display object. In Figure B-1 the pointer points to the solution object "Current solution" indicating that it was the most recent object copied to the tester/display object.

B.2. Generating Starting Solutions

B.2.1. Assigning Random Preference Values

The procedures illustrated in Figure 6-4 require that the program maintains four solution objects. At the top of Figure 6-4 the "Solution generator" creates an object that the program stores and reuses during iterations that follow. This object is the solution object referred to as "Original solution" in Figure B-1. There is no "Solution generator" object as such. The program implements the solution generator as a member function or method of solution objects as a class. If a solution object receives a message "Init" the object assigns random values between zero and one to all the links contained in the object. The only solution object of the four to receive this message is "Original solution".

After receiving the message "Init", the solution object "Original solution" now contains the starting solution that the program uses repeatedly through the learning process. This starting solution serves as the root of the tree in graphs of the type shown in Figure A-4. Normally learning involves the process of returning to the same starting

solution or a slightly modified variation through many iterations. Through a cycle of these iterations the contents of "Original solution" remains unmodified. The tester/display object displays and operates on the contents of the solution "Current solution". The object "Original solution" generates the preference values for the links when the program starts. The program copies these data to "Current solution". The tester/display object displays "Current solution". As mentioned previously the cheating process requires that the program maintains a copy of the best solution encountered to the present time. The solution object "Best solution" maintains the copy of this best solution. Initially this solution is a copy of the starting solution contained in the object "Original solution".

B.2.2. Setting the Threshold Value

Once the object "Original solution" assigns random values to its links the program sets the object's threshold variable to the alpha value and evaluates the total length of the resulting solution. The program uses a method based on bisection to find the alpha value of the threshold variable. The method sorts links on the basis of preference values. The bisection method places an upper and lower bound on the alpha value. Through a series of iterations the upper and lower bounds converge to the alpha value. The upper bound is initially equal to 1, which excludes all links in the Hanan grid from the network. This solution is necessarily infeasible. The lower bound is initially equal to 0, which includes all links in the Hanan grid in the network. This solution is necessarily feasible. The method does not actually use threshold values but instead uses indices into the sorted array of links. This approach includes link elements as part of layout solution if the indices are lower than the threshold index.

The next step in the method sets the threshold index to include links from the sorted array up to an index exactly half way between the upper bound and the lower bound. The term "midpoint index" designates this position in the sorted array. The method then tests the resulting network. The program copies the contents of "Original solution" to the tester/display object converting the data to binary and the program passes the message "Test" to the tester/display object. The message causes the tester/display object to perform the feasibility test on the solution data. If the user of the program has set the display mode options to the highest settings the program displays the testing procedure graphically. The user can also set the program to step through the testing procedure one step at a time by setting the "Key_Pause" parameter.

The tester/display object passes the results of the feasibility test back to the solution object. If the solution is feasible, the midpoint index becomes the new lower bound. If the test results are infeasible the midpoint index becomes the new upper bound. The program finds a new mid point index halfway between the new upper and lower bound. The program performs the bisection procedure and test until the upper and lower bounds converge, which indicates the location of the alpha value. The bisection method is generally more efficient than the procedure of removing links one at a time and testing the results, particularly for large networks in the later stages of improvement.

B.2.3. Improvement

The program copies the contents of the solution object "Original solution" again to the tester/display object once the program has set the threshold variable to the alpha value. The program passes the message "Improve" to the tester/display object and the tester/display object identifies all the branched components in the solution. The program

then "copies" the contents of the tester/display object back to the solution object. The copying procedure actually involves setting the preference values of all links that are branched components to values below the alpha threshold value, thereby removing those links from the solution layout.

B.2.4. Evaluation

The final procedure in generating a new solution is to evaluate the total length. Once again the program copies the contents of the object "Original solution" to the tester/display object. The program passes the message "Evaluate" to the tester/display object. The tester/display object then sums the length of all the links with a value equal to 1 and passes the sum back to the solution object "Original solution". At this point the only links that have a value set equal to one are the looped components of a feasible network.

B.3. The Agitator

According to Figure 6-4 the program modifies a newly generated solution by applying the agitator. Like the "solution generator" the agitator is not an object but a collection of procedures. The program performs the agitation process using the two solution objects "Current solution" and "New solution" in combination with the tester/display object. To agitate the object "Original solution" the program first copies the contents of "Original solution" to the object "Current solution". Then the program again copies the contents of "Current solution" to the object "New solution". The program passes the message "Mutate" to "New solution" which causes "New solution" to randomly select a small number of links and reassign their preference values. The process of setting the threshold variable of "New solution" to the alpha value is the same

as was described for "Original solution". Once the object "New solution" finds the alpha value for the altered link preferences the program removes the object's branched components and evaluates the total length of the resulting solution.

However, if the method produces an improved solution the program copies the contents of "New solution" to "Current solution". The solution object "Current solution" now contains the solution referred to as "Modified starting solution" in Figure 6-4. If the solution in "New solution" is not an improvement over "Current solution" the program repeats the procedure of copying the contents of "Current solution" to "New solution". It modifies the copied data and evaluates the result until "New solution" evaluates with a total length that is less than the total length of "Current solution". Alternatively, the agitation process may repeat to the maximum number of iterations. If the maximum number of iterations occurs the agitator has failed to produce an improved solution.

If an improvement occurs the program compares the total length of the "New solution" object with the "Best solution" object. The program copies the contents of "New solution" to "Best solution" if "New solution" has less total length than "Best solution". "Best solution" maintains a copy of the best solution that the program has encountered to the present time.

B.4. The Production System

The production system applies rules using an approach similar to that used by the agitator. The approach uses two solution objects, "Current solution" and "New solution". The method also uses the tester/display object and a rule base object. The rule base object contains the rules and the trigger set shown in Figures A-2 and A-3.

The production system copies the contents of "Current solution" to "New solution" before applying any rules. The program passes the message "Apply_Rules" to the rule base object. On receiving this message the rule base object applies the rule associated with the instantiation record with the highest priority in the trigger set. It applies this rule to the object "New solution". If the trigger set is empty at this time the rule base object initiates an iteration of instantiation of rules. If no rules instantiate with priority greater than zero, "Rules" returns a value of zero indicating that the current solution is a local optimum and the application of rules should cease.

The program tests the modified solution in "New solution" for feasibility whenever the rule base object applies a rule instantiation from the trigger set. If "New solution" is feasible the program copies its contents back to "Current solution" and another iteration of application of rules occurs. If "New solution" is infeasible the reverse process occurs. The program copies the contents of "Current solution" to "New solution" and passes the message "Rule_Fails" to the rule base object informing it that the last rule to fire produced an infeasible solution. The rule base object increments the "Failed" counter. This is the method by which the program implements the retraction of rules that produce infeasible solutions.

The program applies rules to "New solution" as described until the rule base object returns a zero value indicating that the production system has reached a local optimum. At this point the production system is finished. The solution object "Current solution" contains the final result of all successful applications of rules that produced a feasible improvement. Figure 6-4 refers to the solution stored in "Current solution" at this point as "Local optimum".

At the end of the application of the rules the program checks the solution that results from the production system to see if it is the best solution found to this time. If the solution stored in the object "Current solution" has less total length than the solution in "Best solution" the program copies the contents of "Current solution" to "Best solution".

According to Figure 6-4 the program passes a copy of the "Local optimum" solution directly to the rule formulator. The program passes another copy to the agitator.

B.5. The Second Application of the Agitator

The process of applying the agitator to the "Local optimum" solution is exactly the same as described in the previous section in which the agitator modified "Starting solution". The agitator performs each iteration of random manipulation on a copy of "Current solution", "New solution", until an improvement occurs. The method bypasses the cheater process in Figure 6-4 if the agitator produces an improvement before the maximum number of iterations occurs. The method passes the results of the agitator directly to the rule formulator in Figure 6-4 in this case.

If the application of the agitator produced an improved solution the program compares the results of that improvement with the object "Best solution" as in the previous application of the agitator. If "New solution" has less total length than "Best solution" the program copies the contents of "New solution" to "Best solution".

In the first application of the agitator the program converts the "Starting solution" to the "Modified starting solution" in Figure 6-4. During this process the agitator is unlikely to reach the maximum number of iterations before it finds an improved solution. The program creates initial starting solutions using random generation and these solutions

are usually very poor. Improvement of these solutions by random manipulation is highly probable. The program sets the maximum number of iterations at a high value (100 iterations) to improve the probability that this early use of the agitator will succeed. However, if the production system is successful at producing very good local optima the probability that the second agitator will find an improved solution after 100 iterations may be very low. In this instance it may be necessary to use the best solution. If the agitator fails to produce an improved solution the cheater compares the total length of "Current solution" with "Best solution". The cheating process occurs if "Best solution" is better than "Current solution". The program then copies the contents of "Best solution" to "Current solution". The cheater, like all of the processes in Figure 6-4, is not actually an object within the program but is a process that results from the interaction of objects in Figure B-1.

B.6. Rule Formulation

B.6.1. Rule Formulation Objects

The program does not implement the rule formulator described in Chapter 6 as a single discrete object within the program. The member functions of several objects and the interfacing between those objects perform the rule formulation process. An understanding of the interactions between the objects used by the program is necessary for a complete understanding of the rule formulation process.

Three objects play an important role in the rule formulation process. These objects are of the class "links", the class "sieve" and the class "clump". The next sections explain the use of sieves, links and clumps in the rule formulation process in detail.

B.6.2. Links Objects

The term "links" is used in three contexts in describing the learning program. A "link" refers to the arc that connects two nodes. The term "links" is also used to refer to the links matrices shown in Figure 2-5. This chapter introduces a third context. The learning program makes use of an object of the class "links". There are two "links" objects, "Before" and "After", as shown in Figure B-1. "Links" objects contain solution data in the form of the east and south links matrices of the type shown in Figure 2-5.

B.6.3 Sieve Objects

Sieve objects remove changes that are of no consequence to the total length of the layout before the program formulates a rule. Sieves prevent the rule base from formulating rules that alter the position of links but produce functionally equivalent solutions. The process of forgetting eliminates rules of this type once the rule base has acquired them. However, sieve objects prevent these rules from being acquired initially. The sieve functions as a filter through which inconsequential changes cannot pass. The sieve removes the inconsequential change and returns the network layout to its form before the change.

The program uses two sieve objects to accomplish the elimination of inconsequential changes. The program copies the layout solution to one sieve object before making a change to the layout using the second agitator. After the program has made the change, it copies the altered solution to a second sieve object. The program passes the message "Match_All" to the second sieve along with a reference to the first sieve. The program finds the inconsequential changes in the second sieve object and

restores the changes to the form found in the first sieve object. The section that follows explains the operation of sieve objects in more detail.

The sieve uses the concept of equivalent atomic paths. An atomic path consists of a path that starts and ends on a terminal node but does not contain any other terminal nodes within the path. Terminal nodes consist of all nodes except Steiner nodes of degree 2. All basic vertices are terminal nodes regardless of their degree. Tees and intersections are terminal nodes. The only nodes that are not terminal are Steiner nodes that are part of the continuation of a straight section or Steiner nodes that form 90° elbows.

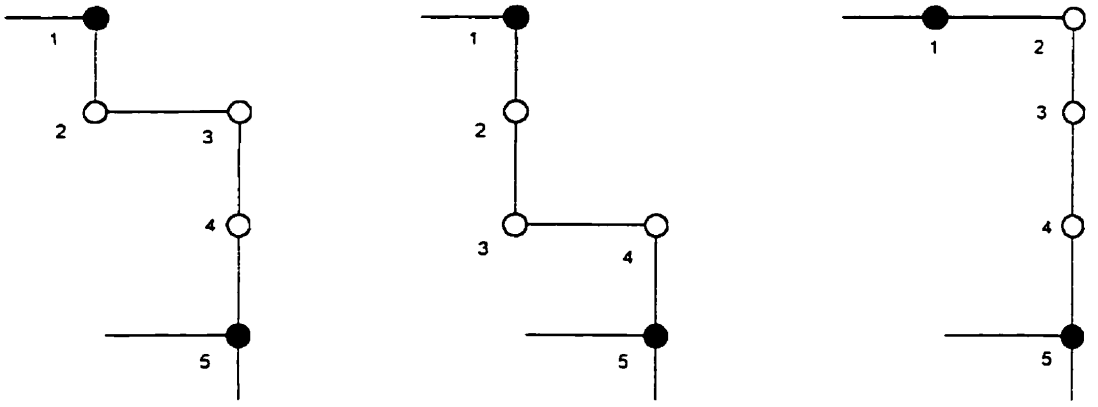


Figure B-2: Equivalent paths

Two paths must meet four criteria to be equivalent paths. (1) They must both be atomic paths. (2) They must start and end at the same terminal nodes regardless of

direction. (3) They must have the same length and (4) they must contain the same number of bends. In Figures B-1a, B-1b and B-1c each of the paths connecting node 1 to node 5 is an atomic path. However, all of these paths are not equivalent atomic paths. Nodes 1 and 5 are terminal nodes. Node 1 is terminal because it is a basic vertex (shown as a solid circle) and node 5 is terminal because it is a basic vertex and because it is of degree 3. Each of the nodes along the path, nodes 2, 3 and 4, are non-terminal.

By inspection each of the paths between nodes 1 and 5 are the same length. However, the number of bends in all of the paths is not the same. In Figure B-2a bends occur at nodes 2 and 3. Bends that may occur at the terminal nodes are part of the path as well. A bend occurs at node 1 in Figure B-2a but no such bend occurs at node 5. A bend in a path does not occur at a terminal node if a link is present at the terminal node opposite the path's incident link. The path's incident link at node 5 occurs on the north side of node 5. The path would have an additional bend if a link were not present to the south of node 5. The path in Figure B-2a has 3 bends.

The paths in Figures B-1a and B-1b meet the criteria for equivalent atomic paths. Both paths have the same terminal nodes. They have the same length and have the same number of bends. The paths are functionally equivalent and there is no advantage in selecting one over the other.

Two examples illustrate how sieve objects would operate on the equivalent paths in Figure B-2a and Figure B-2b. In the first example the agitator/cheater might change part of a network from the layout shown in Figure B-2a to Figure B-2b. Sieve objects restore the altered part of the layout back to the original path in Figure B-2a in this case. The second example is the reverse case. The change caused by the agitator/cheater

alters the layout from that shown in Figure B-2b to Figure B-2a. In this case the sieve objects would restore the layout back to the original layout in Figure B-2b.

The path in Figure B-2c, however, is not equivalent to the other two paths in Figure B-2. The path in Figure B-2c has the same terminal nodes and the same length. However, the path has two fewer bends. Sieve objects would not eliminate a change from either layout in Figure B-2a or Figure B-2b to the layout in Figure B-2c.

The sieve object performs a partially destructive test. The data are not in their original format at the end of the test but the object can recover the data. The sieve object contains data similar to the tester/display object. A sieve contains a copy of the nodes matrix and the east and south links matrices. However, there is no cells matrix as with the tester/display object. The sieve constructs a linked list with an entry for each atomic path in the layout. Each of the path records in the list contains the following:

1. A unique path identifier - integers starting at 2 and incremented with each path as it is marked and recorded.
2. Starting row and column - location in the Hanan grid of one of the two terminal nodes of the path.
3. End row and column - location in the Hanan grid of the other terminal node in the path.
4. The number of bends in the path.
5. The length of the path.
6. A flag - set to 0 or 1. The flag equals 1 if this path is equivalent to a path in another sieve object, otherwise the flag equals 0.
7. A pointer to the next record in the list.

The first step in constructing the list of atomic paths is to identify all terminal nodes. The nodes matrix already identifies all basic vertices with an element set equal to 1. All other nodes have corresponding elements equal zero. The sieve object must identify the remaining non-basic terminal nodes. The sieve object calculates the degree

of every element in the nodes matrix set equal to zero. The sieve sets the value of the corresponding element in the nodes matrix to 1 if the degree is equal to 1, 3 or 4. In this way the sieve identifies all terminal nodes by a value of 1 in the nodes matrix.

The next stage of the process is to traverse the links of every atomic path in the layout. Each of the selected links in the Hanan grid has its corresponding element in either the east links or south links matrices set equal to 1. The procedure examines every terminal node to find each incident link equal to 1 and traverse that path to the next terminal node marking the path with a unique identifier. The identifier values begin at 2 and increase as the sieve marks each path. As the sieve marks each path it constructs a path record and places the record in the list. The sieve records the path's identifier in the path record and the identifier serves as a cross reference between the path record and the path in the links matrices. As the sieve traverses the path the sieve maintains a running total of the length of the links and the number of bends. The sieve enters these totals in the path record once it reaches the terminal node at the end of the path. The process of traversing and marking paths ends when there are no terminal nodes with incident links set equal to 1.

Referring again to Figure 6-4 the program copies the solution referred to as "Local optimum" in the figure to the sieve object "BeforeS" in Figure B-1. Then the agitator/cheater combination modifies "Local optimum" to produce the solution referred to as either "Improved solution" or "Previous best" (Figure 6-4) depending on the success of the agitator component in finding an improved solution. The program copies this solution to another sieve object called "AfterS" (Figure B-1) before passing it to the rule formulator. At this time the program passes the message "Match_All" to "AfterS" along with a reference to "BeforeS".

Once "AfterS" receives the message "Match_All" it compares every path in its path list with the paths in "BeforeS". If "AfterS" finds an equivalent atomic path in the "BeforeS" list it checks if the flag of the path record is equal to zero. The program initially sets all flags to zero. A sieve object will set the flag of an atomic path to 1 if the sieve finds the path to be equivalent to a path in another sieve object. Sieve objects use the flag to prevent a path from matching more than one path in another object. A match with a path in the "AfterS" object occurs if a path in the list of "BeforeS" exists that meets the criteria for equivalence and has its flag equals zero. Once found "AfterS" sets the flag in the matching "BeforeS" record to 1 so that the object cannot pair with a path a second time. "AfterS" then removes every link in the matching equivalent path in its own links matrices. Once the object has removed all of the links in the path, "AfterS" copies the links of the equivalent atomic path in "BeforeS" to its links matrices.

The program copies the contents of sieve objects "BeforeS" and "AfterS" to links objects "Before" and "After" respectively (shown in Figure 6-4 and Figure B-1). The links objects contain east links and south links matrices in binary format. The program restores the contents of the east and south links matrices in the sieve objects to the original binary format when copying the matrices to the links objects. The program copies any link with a non-zero value in a sieve object as 1 in the links object. The program copies zero values as zero. The program destroys the sieve objects "BeforeS" and "AfterS" after it copies their contents to the links objects.

B.6.4. Clump Objects

The clump class is a derived class of the links class and each clump object contains an east links matrix and a south links matrix. The clump object creates a

template that identifies non-contiguous changes in the layout geometry enabling the rule formulator to derive separate rules for each of the disjoint changes in the layout geometry.

A clump object uses the concept of a "template". The "template" identifies the pattern of the links that differ when the rule formulator compares two links objects. The "logical exclusive OR" (XOR) function identifies the locations where links differ between the two links objects.

The logical XOR function differs from the logical OR function in ways that enable the XOR function to identify patterns of changed links. A logical OR function returns the value of a single bit based on the value of two input bits. The function is similar to the "or" conjunction used in logic. An "or" conjunction produces a true statement if either of the two statements it joins are true or if both statements are true. For the logical OR function a bit set to 1 is analogous to a true statement and a bit set to zero is analogous to a false statement. Two bits that serve as input to logical OR function produce a result of 1 if either or both of the two input bits equal 1. A zero output results only if both input bits equal zero.

The XOR function differs from the OR function in that it returns a value of 1 only if one of the two input bits is equal to 0 and the other is equal to 1. The function returns a value of zero if both input bits are equal to 1 or both are equal to 0. The learning program uses the function to construct a set of links matrices based on an exclusive OR of the corresponding elements of the two links matrices. The resulting matrices have elements equal to 1 at only those locations where the values differ between the two input

sets of matrices. The term "template" refers to the pattern of elements set to 1 that identify where changes occur.

The single clump object, referred to as "Template" in Figure B-1 accepts the links objects "Before" and "After" as input and creates a template based on the differences between these two objects. The next step, the object "Template" examines if the template pattern produced by the change consists of non-contiguous regions of the network layout. Non-contiguous regions consist of two or more contiguous changes that are physically separate. The object separates the non-contiguous template to enable the rule formulator to create separate rules for each contiguous pattern.

The process of identifying a contiguous region of altered links involves marking each of the links in the region with a unique identifier value. The identifier values begin with the number 2 and increase as the object identifies each contiguous region. The "Template" object scans its links matrices to find the first instance of a link set to 1 by the XOR function. Once the object finds a link equal to 1 it changes the link's value to -1. Then an iterative process begins in which the object scans the matrices for links set equal to -1. If the object finds a link equal to -1 it changes the link's value to the current clump identifier value, which is 2 initially. The next step in the procedure operates on the six immediately adjacent links to the link that changed in value.

Figure B-3a shows the pattern of the six immediately adjacent links of an east link and Figure B-3b shows the six immediately adjacent links of a south link. Figure B-3 shows the link marked with the current clump identifier as a rectangle and the adjacent links are solid lines. If any of the adjacent links are equal to 1 the object changes their values to -1. The object repeats the process of searching the links matrices for values

equal to -1. It changes them to the current clump identifier value and changes any adjacent links equal to -1. The object performs the procedure until it cannot find any links equal to -1 indicating that the object has marked the entire contiguous region.

(a) East link (b) South link

Figure B-3: Adjacent links

The object increments the clump identifier value. The object repeats the procedure from the beginning, scanning the east and south links matrices for any remaining elements equal to 1. If it finds any elements equal to 1 it repeats the marking process until no elements equal to 1 remain. Figure B-4 shows a flow chart of the algorithm. Once the algorithm terminates the "Template" object will have marked each of the disjoint regions with unique identifiers beginning with 2. In the final stage the "Template" object decrements the identifiers so that the numbering scheme begins at 1. The rule formulator uses the resulting patterns to derive rules in conjunction with the links object "Before".

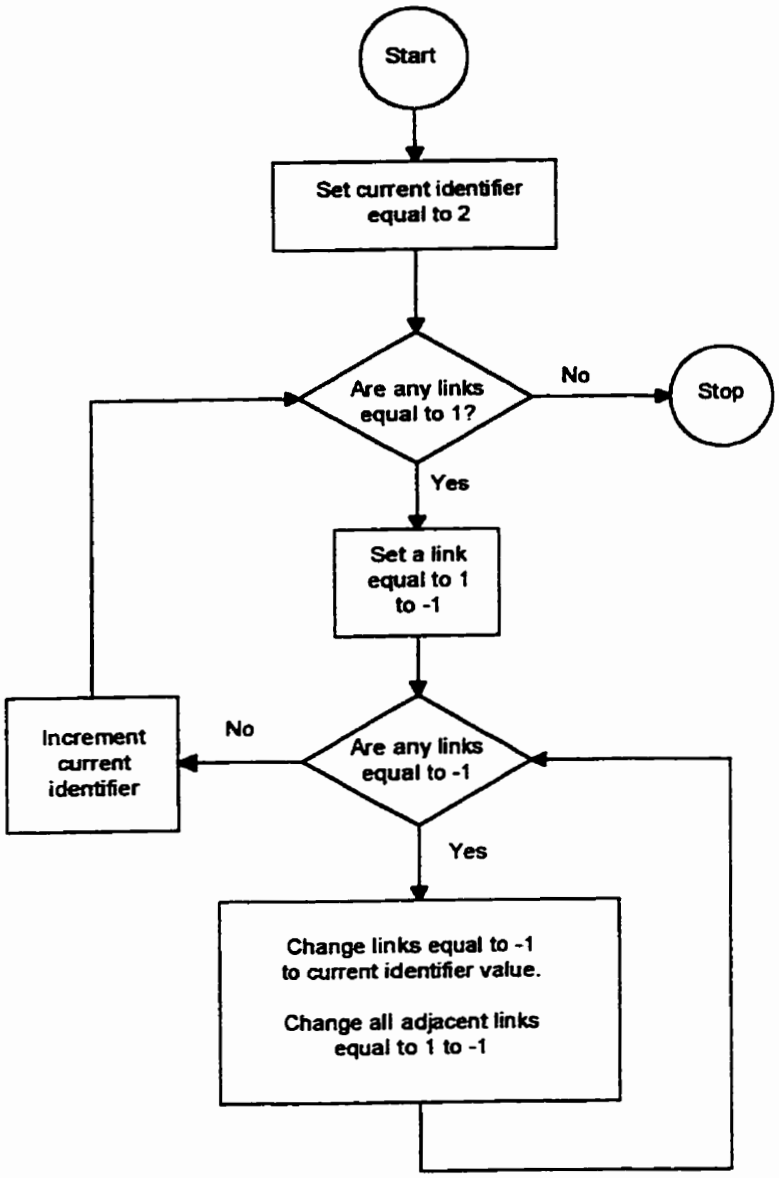


Figure B-4: Flowchart of clump algorithm

B.7. Formulating Rules

Member functions of the rule base object "Rules" perform the final steps in formulating rules. The rule base object creates new rules based on the patterns in the clump object "Template". The rule base object derives rules from each of the contiguous regions that "Template" identifies. In the first stage in the derivation of a rule, the rule base object determines the boundaries of the smallest rectangular array that encloses the links marked with an identifier in the "Template" object. The boundary must also include all the immediately adjacent links. The shape of the pattern of links marked by the identifier is not necessarily rectangular. However, rules use rectangular arrays to store the contents of the antecedent and consequent clauses. This fact implies that a rule does not require all the information contained in its rectangular arrays during instantiation or application. A rule requires all the data in the arrays only in the case in which a pattern of altered links is rectangular. The algorithm uses templates to define which elements of the rectangular arrays actually constitute parts of the rule. The rule stores the template as its consequent clause.

The pattern of altered links may exist only in the east links matrix of the "Template" object or entirely in the south links matrix. The pattern may exist in both matrices having different rectangular boundaries in each. The rule base object finds a single composite rectangular boundary for the rule template from the east and south links matrices for a single region.

The program derives a rule by passing the message "Add" to the rule base object along with several parameters. The parameters include the links object "Before", the clump object "Template", the rule's boundaries and the template identifier value. The

identifier value selects the contiguous region in the object "Template" from which the rule base object is to derive the new rule.

The rule base object allocates memory for the components of the new rule. The program implements the rule base as a doubly linked list. Each rule forms an element of the list. Each rule requires two pointers, one pointer pointing to the next element in the list and the other pointing to the previous element in the list. The rule includes the dimensions of the matrices and the counters described previously in addition to the matrices that define the antecedent and consequent clauses. The list that follows describes the components of a rule.

Next - pointer to the next rule

Last - pointer to the previous rule

Row - number of rows in each of the matrices

Cols - number of columns in each of the matrices

Sinks - a matrix that identifies all nodal positions that cannot be occupied by basic vertices

Ante_E - east links matrix of the antecedent clause

Ante_S - south links matrix of the antecedent clause

Conse_E - east links matrix of the consequent clause

Conse_S - south links matrix of the consequent clause

Used - counter that records the number of times the program has used the rule

Failed - counter that records the number of times the rule produced and infeasible solution

Dumped - counter that records the number of time the rule instantiated with a non-positive priority value

Type - integer identifier value explained in Appendix B (B.7.4)

Once the rule base object has allocated space for the matrices it copies data from the rectangular region specified in the rule's boundaries from the "Before" links object to the matrices that constitute the antecedent clause. The object copies the

corresponding data from the "Template" object to the consequent clause matrices. In each case in which an element in the "Template" object is equal to the identifier value, the object copies a value of 1 to the corresponding element in the consequent clause.

The program uses template data from the "Template" object to form the consequent clause rather than data from the "After" links object. The program uses a logical XOR function to apply the consequent clause in template format. Using data obtained from the "After" object might seem to be the simpler approach since the program could apply the rule by simply copying the contents of the consequent clause to the solution. However, the program uses template data and an XOR function because the data in template format preserves the shape of the pattern of links that the agitator/cheater changed, which is not necessarily rectangular. Simply substituting a rectangular array of data into a solution transfers information that is not properly part of the rule in the case of a non-rectangular pattern. The program also uses the consequent clause in template format to determine which links in the antecedent clause must match for the rule to instantiate. The links that must match are all those that the agitator or cheater modified, which equal 1 in the template, and all immediately adjacent links to those set to 1. The remaining links in the antecedent clause do not constitute part of the rule.

B.7.1. The "Sinks" Matrix

The "Sinks" matrix in a rule acts as an additional antecedent clause. The additional clause to prevents the application of a rule from disconnecting any basic vertices. The "Sinks" matrix contains node data and has the same dimensions as the matrices of the antecedent and consequent clauses. A member function of each rule

creates the data in the "Sinks" matrix. The function checks each node within the boundaries of the rule and calculates the degree of the node after the program has applied the rule. All basic vertices must be of degree 2 or higher in a fully looped network. The function marks any nodes with a value of 1 that have fewer than two adjacent links after the rule fires. Any nodes marked as 1 in the "Sinks" matrix cannot be basic vertices in the network layout for the rule to apply.

B.7.2. Checking for Previous Instances of Rules

The rule is fully formulated once the antecedent, consequent and "Sinks" matrices are complete. The "Rules" object checks the rule base to determine if it has acquired a similar rule previously. This test involves searching the entire rule base to find a rule that matches the rule just formulated. The comparison involves only the links that the agitator/cheater altered and immediately adjacent links. This is the same type of comparison as in the instantiation process. The comparison ignores any additional data included in the rectangular arrays. If the program cannot find a similar rule in the rule base the program adds the new rule to the rule base, otherwise the program destroys the new rule.

The program will not learn the same rule twice. At first glance the process of checking the rule base for an earlier instance of a newly formulated rule may appear unnecessary. The agitator/cheater combination should not be able to produce a change that would result in a rule that is already present in the rule base. If the rule exists already and can produce an improvement the production system should apply it before the agitator/cheater can generate the same improvement.

Checking the rule base for earlier instances of rules is necessary because non-contiguous changes can operate in combination to generate two or more rules. If one change creates a large improvement and another change makes the solution slightly worse the effect of the combination of changes is an improvement. The algorithm acquires rules based on both patterns in this case, the pattern that produced the large improvement and the pattern that increased the total length by a smaller amount. A rule corresponding to the pattern that creates the increase in length may already be present in the rule base. In this case the production system will not apply the rule since it will instantiate with a negative priority. The only way to guarantee that the rule base has not previously acquired a rule of this type is to search the rule base.

B.7.3. Rotation and Reflection

After the program has added a rule to the rule base the program rotates the pattern of the rule 90 degrees to produce another variation of the rule. The program tests the 90 degree variation to see if it has acquired the rule previously. The program adds the rule to the rule base if the variant is not already present. The program repeats the process to produce a 180 degree variant and a 270 degree variant and it checks both rules to determine if it has acquired the rules previously. Once the procedure has generated all four rotations the program mirror reflects the original variant of the rule. The procedure tests the mirror reflection to see if it has acquired this rule previously. If the mirror reflection is a new rule the program generates and tests 90°, 180° and 270° rotations of the rule. Rotation or reflection may produce the same rule because the rule may be symmetrical. Therefore program must test the rule base each time it generates a new variant of a rule through rotation or reflection. The program can generate up to eight

rules from a single contiguous region in the "Template" object depending on the symmetry of a pattern of changed links.

B.7.4. Numbering of Rules

The program usually acquires rules that have several variants obtained through rotation and reflection. Problems typically have asymmetrical geometry that does not offer the production system the opportunity to apply all the rule variants. Unequal testing opportunities produce an unrealistic assessment of the performance of individual variants. For example, the program may use a rule frequently to solve a problem and not use the 90 degree rotated variant. Consequently, the program will forget the 90 rotated variant due to infrequent use. In this instance forgetting will undo much of the benefit associated with the derivation of all rule variants. The resulting rule base will exhibit inconsistent and unpredictable performance. The program may be able to solve a given problem in one orientation but if the program attempts to solve the same problem rotated 90 degrees the results may be very poor.

The solution to the problem recognizes that the statistics gathered by the three counters in each rule apply equally to all variants of the same rule. Each rule contains a single integer variable in addition to the counters and clauses. This variable, referred to as the "type", takes the same value for all variants of a rule generated through rotation and reflection. While the type value is the same for all variants, each set of variants receives a unique type value. Before forgetting the program sums the counter values for all the rules with the same type value and assigns the sums to the counters of each of the individual variants. Therefore, immediately before forgetting each rule's counter values reflect the sum of the results it and all its related variants have obtained.

Appendix C: Algorithms

C.1. Closing Rows

The flood-fill algorithm divides the area it marks into units called "closed rows". A closed row is a contiguous row of cells undivided by links. The identifier value -1 marks the cells in a closed row indicating that the algorithm has pushed the location of the "starting point" of the row on the stack.

To close a row the algorithm sets the cursor position to the row and column of any cell in that row. The cursor moves to the west setting each cell to -2 until the cursor encounters either a cell set to -2 or a south link. A cell set to -2 indicates that the row runs the full width of the Hanan grid and wraps around on itself. All the cells in the row are now equal to -2. A south link is a barrier indicating the west end of the row. The cursor stops where it encounters a south link or cell set to -2. This location represents the starting point of the row and the algorithm pushes the location on the stack. In the case of a south link barrier the starting point is the west most cell in the row.

Next the cursor moves east marking all the cells with -1 until it encounters either a cell equal to -1 or a south link. A cell set to -1 occurs only in the case of a row that runs the full width of the Hanan grid. Encountering a cell equal to -1 indicates that the row is now a closed row, that is, all cells in the row equal -1. Similarly a south link indicates that the cursor has reached the east end of the row and all cells in the row equal -1. The algorithms included at the end of the appendix provide a more complete description of the row closing procedure.

C.2. Marking Cells in a Row

The flood-fill algorithm consists of identifying the closed rows that make up the area and "marking" each of them. Marking a closed row consists of replacing the -1 value of each of the cells in the row with a unique identifier value for that contiguous area. The process begins by popping the starting point of the row from the stack. The algorithm sets the starting cell to the polygon identifier value. Then the operation proceeds eastward setting every cell in the row currently equal to -1 to the polygon identifier value until the procedure encounters a cell that is not equal to -1. The flood-fill algorithm performs the two procedures, closing rows and marking rows, at the same time.

As the row marking procedure sets a cell to the current area identifier value it checks the row to the north of the cell and the row to the south. The check determines if the row closing procedure can close either row before the cursor moves to the next cell to the east. The algorithm closes the row to the north or to the south if an east link does not block access to the row and if previous operations have not closed or marked the row. If the rows to the north or the south (or both) are accessible and neither marked nor closed, the row closing algorithm closes the either or both rows as required.

Once the row closing procedure has checked the rows to the north and the south the row marking procedure moves the cursor one position to the east. The row marking procedure checks if the cursor passed across a south link or if the cell at the new position is already equal to the identifier value. If either condition occurs the algorithm has marked all the cells in the row and the row marking process terminates.

Once the row marking procedure has marked all the cells in a row the flood-fill algorithm pops the starting point of a new closed row from the stack and the algorithm invokes the row marking procedure to mark that row. When the stack is empty the marking process is complete for an entire contiguous region of cells The algorithms included at the end of the appendix provide a more complete description of the procedure for marking cells.

C.3. The Flood-Fill Algorithms

C.3.1. The row closing algorithm

Assume that all cells in the row have not been marked and a barrier has not been encountered.

Repeat the following until either all the cells are marked or a barrier is encountered:

set the cell at the current cursor position equal to -2;

if the south link at the current position is set equal to 1

recognize that a barrier has been encountered,

otherwise do the following:

move the cursor one position to the west;

if the cell at the current position is not set equal to 0 do the following:

recognize that all the cells in the row have been marked;

move the cursor one position to the east.

Push the current cursor position on the stack.

Assume again that a barrier has not been encountered.

While the cell at the current position is equal to either -2 or 0 and no barrier has been encountered repeat the following:

set the cell at the current position equal to -1;

move the cursor one position to the east;

if the south link at the current position is set equal to 1

recognize that a barrier has been encountered.

C.3.2. The cell marking algorithm

While the cell at the current cursor position is set equal to -1 repeat the following:

set the cell at the current cursor position equal to the current polygon identifier value;

if the east link at the current cursor position is set equal to 0 do the following:

record the position of the cursor and move the cursor one position to the north;

if the cell at the new position is set equal to 0

close the row at the current cursor position;

move the cursor back to the most recently recorded position;

record the position of the cursor and move the cursor one position to the south;

if the east link at the current cursor position is set equal to 0 and

if the cell at the new position is set equal to 0

close the row at the current cursor position.

move the cursor back to the most recently recorded position;

move the cursor one position to the east;

if the south link at the current position is set equal to 1 stop.

C.4. The Loop Marking Algorithm

This section provides a detailed description of the rules that control the cursor movement in the loop marking algorithm described in Chapter 3 (Section 3.2).

C.4.1. Four Rules that Define Cursor Movement

Rule 1

If the cursor is pointed north then do the following:

move the cursor one position to the north;

if the cell at the new position is equal to the current polygon identifier

change the direction of the cursor to point west,

otherwise do the following:

move the cursor one position to the south to return to the original position;

if the node at this position is equal to 1

set the node equal to -2,

otherwise

set the node equal to -1;

if value of the east link at this position is equal to -1

set the east link equal to -2,

otherwise

set the east link equal to -1;

change the direction of the cursor to point east.

Rule 2

If the cursor is pointed east do the following:

move the cursor one position to the east;

if the cell at the new position is equal to the current polygon identifier

change the direction of the cursor to point north,

otherwise do the following:

if the node at the new position is equal to 1

set the node equal to -2,

otherwise

set the node equal to -1;

if value of the south link at the new position is equal to -1

set the south link equal to -2,

otherwise

set the south link equal to -1;

move the cursor one position to the west to return to the original position;

change the direction of the cursor to point south.

Rule 3

If the cursor is pointed south do the following:

move the cursor one position to the south;

if the cell at the new position is equal to the current polygon identifier

change the direction of the cursor to point east,

otherwise do the following;

move the cursor one position to the east;

if the node at this position is equal to 1

set the node equal to -2,

otherwise

set the node equal to -1;

move the cursor one position to the west;

if the east link at this position is equal to -1

set the east link equal to -2,

otherwise

set the east link equal to -1;

move the cursor one position to the north to return to the original position;

change the direction of the cursor to point west.

Rule 4

If the cursor is pointed west do the following:

move the cursor one position to the west;

if the cell at the new position is equal to the current polygon identifier

change the direction of the cursor to point south,

otherwise do the following:

move the cursor one position to the east;

move the cursor one position to the south;

if the node at this position is equal to 1

set the node equal to -2,

otherwise

set the node equal to -1;

move the cursor one position to the north to return to the original position;

if the value of the south link at this position is equal to -1

set the south link equal to -2,

otherwise

set the south link equal to -1;

change the direction of the cursor to point north.

Appendix D: Listing of Source Code and Data Files

All of the C++ programs listed require the Watcom C++ Version 9.5 compiler. The batch file "make.bat" compiles all of the source code required to perform the tests in Chapter 7. The last subsection of each section in Chapter 7 provides the names of the batch files that perform the tests in that section.

D.1. Sample Problem Files

This appendix includes a listing of sample problem files. The problem files require data to be entered in specific fields. A user of the programs should not attempt to create these files using a text editor unless he or she takes special care to enter the data in the proper fields. The appendix provides the program "ndmain.exe" to create proper problem files. Anyone wishing to create their own problem files should use "ndmain.exe" rather than a text editor.

NODES01

Total number of nodes 10

Num	X	Y
1	31246.7	56234.5
2	32975.1	23964.1
3	38549.3	73749.3
4	35436.4	78474.2
5	36437.4	42833.3
6	54833.4	43734.2
7	63348.2	46632.0
8	45695.3	57934.3
9	47928.2	37834.2
10	47945.4	47584.5

NODES02

Total number of nodes 10

Num	X	Y
1	73874.2	72649.3
2	38549.3	73749.3
3	35436.4	78474.2

4	36437.4	42833.3
5	63745.2	26396.2
6	47928.2	37834.2
7	47835.3	87445.3
8	56784.4	7964.2
9	47945.4	47584.5
10	80459.4	49846.5

NODES03

Total number of nodes 10

Num	X	Y
1	36437.4	42833.3
2	63745.2	26396.2
3	38224.4	34734.4
4	63348.2	46632.0
5	45695.3	57934.3
6	47928.2	37834.2
7	47945.4	47584.5
8	80459.4	49846.5
9	83875.3	49850.4
10	48739.4	37835.0

NODES04

Total number of nodes 10

Num	X	Y
1	32975.1	23964.1
2	73874.2	72649.3
3	38549.3	73749.3
4	35436.4	78474.2
5	36437.4	42833.3
6	63745.2	26396.2
7	38224.4	34734.4
8	54833.4	43734.2
9	63348.2	46632.0
10	23487.3	34233.2

NODES41

Total number of nodes 4

Num	X	Y
1	34234.5	66345.6
2	32975.1	23964.1
3	73874.2	72649.3
4	38549.3	73749.3

NODES42

Total number of nodes 4

Num	X	Y
1	35436.4	78474.2
2	36437.4	42833.3
3	63745.2	26396.2
4	38224.4	34734.4

NODES43

Total number of nodes 4

Num	X	Y
1	38224.4	34734.4
2	54833.4	43734.2
3	63348.2	46632.0
4	23487.3	34233.2

NODES61

Total number of nodes 6

Num	X	Y
1	7169.1	7565.2
2	6126.4	9782.6
3	7777.4	12043.5
4	4214.6	13087.0
5	12600.3	6782.6
6	13903.8	8913.0

NODES62

Total number of nodes 6

Num	X	Y
1	31246.7	56234.5
2	34234.5	66345.6
3	32975.1	23964.1
4	73874.2	72649.3
5	38549.3	73749.3
6	35436.4	78474.2

NODES63

Total number of nodes 6

Num	X	Y
1	34234.5	66345.6
2	73874.2	72649.3
3	38549.3	73749.3
4	36437.4	42833.3
5	63745.2	26396.2
6	54833.4	43734.2

NODES81

Total number of nodes 8

Num	X	Y
1	31246.7	56234.5
2	34234.5	66345.6
3	32975.1	23964.1
4	73874.2	72649.3
5	38549.3	73749.3
6	35436.4	78474.2
7	36437.4	42833.3
8	63745.2	26396.2

NODES82

```
Total number of nodes    8

Num          X            Y
---------------------------------
  1      36437.4      42833.3
  2      63745.2      26396.2
  3      38224.4      34734.4
  4      54833.4      43734.2
  5      63348.2      46632.0
  6      23487.3      34233.2
  7      45695.3      57934.3
  8      47928.2      37834.2
```

NODES83

```
Total number of nodes    8

Num          X            Y
---------------------------------
  1      45695.3      57934.3
  2      47928.2      37834.2
  3      47835.3      87445.3
  4      56784.4       7964.2
  5      47945.4      47584.5
  6      80459.4      49846.5
  7      83875.3      49850.4
  8      48739.4      37835.0
```

D.2. MS-DOS Batch Files

For each of the experiments documented in Chapter 7 a single MS-DOS batch program controls copying seed files, maintains information on execution times, and provides any necessary command line parameters. A user can perform any of the experiments in Chapter 7 by typing just the name of the batch file. The section that follows includes all of the batch files for the experiments.

LK.BAT

```
call lkbat nodes41 stats41 time41
call lkbat nodes42 stats42 time42
call lkbat nodes43 stats43 time43
call lkbat nodes61 stats61 time61
call lkbat nodes62 stats62 time62
call lkbat nodes63 stats63 time63
call lkbat nodes81 stats81 time81
call lkbat nodes82 stats82 time82
call lkbat nodes83 stats83 time83
call lkbat nodes01 stats01 time01
call lkbat nodes02 stats02 time02
call lkbat nodes03 stats03 time03
cls
rem    Statistics files produced
rem
rem    problem file     statistics     execution times
```

```
rem     ************     **********     **************
rem     nodes61          stats41        time41
rem     nodes42          stats42        time42
rem     nodes43          stats43        time43
rem     nodes61          stats61        time61
rem     nodes62          stats62        time62
rem     nodes63          stats63        time63
rem     nodes81          stats81        time81
rem     nodes82          stats82        time82
rem     nodes83          stats83        time83
rem     nodes01          stats01        time01
rem     nodes02          stats02        time02
rem     nodes03          stats03        time03
```

LKBAT.BAT

```
del time1
copy seed1 seed.new
time << enter >> time1
lkmain %1 seed stats1
copy seed2 seed.new
time << enter >> time1
lkmain %1 seed stats2
copy seed3 seed.new
time << enter >> time1
lkmain %1 seed stats3
copy seed4 seed.new
time << enter >> time1
lkmain %1 seed stats4
copy seed5 seed.new
time << enter >> time1
lkmain %1 seed stats5
copy seed6 seed.new
time << enter >> time1
lkmain %1 seed stats6
copy seed7 seed.new
time << enter >> time1
lkmain %1 seed stats7
copy seed8 seed.new
time << enter >> time1
lkmain %1 seed stats8
copy seed9 seed.new
time << enter >> time1
lkmain %1 seed stats9
time << enter >> time1
copy stats1+stats2+stats3+stats4+stats5+stats6+stats7+stats8+stats9 %2
copy time1 %3
```

PT.BAT

```
call ptbat nodes41 ptime41
call ptbat nodes42 ptime42
call ptbat nodes43 ptime43
call ptbat nodes61 ptime61
call ptbat nodes62 ptime62
call ptbat nodes63 ptime63
call ptbat nodes81 ptime81
call ptbat nodes82 ptime82
call ptbat nodes83 ptime83
call ptbat nodes01 ptime01
call ptbat nodes02 ptime02
call ptbat nodes03 ptime03
cls
rem    Statistics files produced
rem
rem    Problem file    Execution time
rem    ************    **************
rem    nodes41         ptime41
rem    nodes42         ptime42
rem    nodes43         ptime43
rem    nodes61         ptime61
rem    nodes62         ptime62
rem    nodes63         ptime63
```

```
rem     nodes81          ptime81
rem     nodes82          ptime82
rem     nodes83          ptime83
rem     nodes01          ptime01
rem     nodes02          ptime02
rem     nodes03          ptime03
```

PTBAT.BAT

```
del time1
copy seed1 seed new
time << enter >> time1
ptmain %1 seed stats1 100
copy seed2 seed new
time << enter >> time1
ptmain %1 seed stats1 100
copy seed3 seed new
time << enter >> time1
ptmain %1 seed stats1 100
copy seed4 seed new
time << enter >> time1
ptmain %1 seed stats1 100
copy seed5 seed new
time << enter >> time1
ptmain %1 seed stats1 100
copy seed6 seed new
time << enter >> time1
ptmain %1 seed stats1 100
copy seed7 seed new
time << enter >> time1
ptmain %1 seed stats1 100
copy seed8 seed new
time << enter >> time1
ptmain %1 seed stats1 100
copy seed9 seed new
time << enter >> time1
ptmain %1 seed stats1 100
time << enter >> time1
del stats1
copy time1 %2
```

EPBAT.BAT

```
copy seed5 seed new
epmain 30 1 200 nodes01 seed estat01 0
copy seed5 seed new
epmain 30 0 200 nodes01 seed estat02 0
copy seed5 seed new
epmain 30 0.001 200 nodes01 seed estat03 1
copy seed5 seed new
epmain 10 0 005 200 nodes01 seed estat04 1
copy seed5 seed new
epmain 30 1 200 nodes61 seed estat61 0
copy seed5 seed new
epmain 30 0 200 nodes61 seed estat62 0
copy seed5 seed new
epmain 30 0 001 200 nodes61 seed estat63 1
copy seed5 seed new
epmain 10 0.005 200 nodes61 seed estat64 1
cls
rem     Statistics files produced
rem
rem     Output file   basic vertices   population size   mutation rate
rem     ***********   **************   ***************   *************
rem     estat01       10               30                1
rem     estat02       10               30                0
rem     estat03       10               30                0.001
rem     estat04       10               10                0.005
rem     estat61       6                30                1
rem     estat62       6                30                0
rem     estat63       6                30                0.001
rem     estat64       6                10                0.005
```

EP2.BAT

```
copy seed1 seed.new
epmain 10 0.005 200 nodes61 seed ep2611 1
copy seed2 seed.new
epmain 10 0.005 200 nodes61 seed ep2612 1
copy seed3 seed.new
epmain 10 0.005 200 nodes61 seed ep2613 1
copy seed4 seed.new
epmain 10 0.005 200 nodes61 seed ep2614 1
copy seed5 seed.new
epmain 10 0.005 200 nodes61 seed ep2615 1
copy seed6 seed.new
epmain 10 0.005 200 nodes61 seed ep2616 1
copy seed7 seed.new
epmain 10 0.005 200 nodes61 seed ep2617 1
copy seed8 seed.new
epmain 10 0.005 200 nodes61 seed ep2618 1
copy seed9 seed.new
epmain 10 0.005 200 nodes61 seed ep2619 1
cls
rem     Statistics files produced
rem
rem     Output file   seed value   vertices   population   mutation rate
rem     ***********   **********   ********   **********   *************
rem     ep2611        0.1          6          10           0.005
rem     ep2612        0.2          6          10           0.005
rem     ep2613        0.3          6          10           0.005
rem     ep2614        0.4          6          10           0.005
rem     ep2615        0.5          6          10           0.005
rem     ep2616        0.6          6          10           0.005
rem     ep2617        0.7          6          10           0.005
rem     ep2618        0.8          6          10           0.005
rem     ep2619        0.9          6          10           0.005
```

ESBAT.BAT

```
copy seed1 seed.new
esmain 0.01 200 nodes61 seed es01
copy seed2 seed.new
esmain 0.01 200 nodes61 seed es02
copy seed3 seed.new
esmain 0.01 200 nodes61 seed es03
copy seed4 seed.new
esmain 0.01 200 nodes61 seed es04
copy seed5 seed.new
esmain 0.01 200 nodes61 seed es05
copy seed6 seed.new
esmain 0.01 200 nodes61 seed es06
copy seed7 seed.new
esmain 0.01 200 nodes61 seed es07
copy seed8 seed.new
esmain 0.01 200 nodes61 seed es08
copy seed9 seed.new
esmain 0.01 200 nodes61 seed es09
cls
rem     Statistics files produced
rem
rem     Output file   seed value   vertices   population   mutation rate
rem     ***********   **********   ********   **********   *************
rem     es01          0.1          6          1            0.01
rem     es02          0.2          6          1            0.01
rem     es03          0.3          6          1            0.01
rem     es04          0.4          6          1            0.01
rem     es05          0.5          6          1            0.01
rem     es06          0.6          6          1            0.01
rem     es07          0.7          6          1            0.01
rem     es08          0.8          6          1            0.01
rem     es09          0.9          6          1            0.01
```

R1.BAT

```
call rule1 nodes61 seed1 rstat1
call rule1 nodes61 seed2 rstat2
call rule1 nodes61 seed3 rstat3
call rule1 nodes61 seed4 rstat4
call rule1 nodes61 seed5 rstat5
call rule1 nodes61 seed6 rstat6
call rule1 nodes61 seed7 rstat7
call rule1 nodes61 seed8 rstat8
call rule1 nodes61 seed9 rstat9
cls
rem    Statistics files produced
rem
rem    Output file  seed value  vertices
rem    ***********  **********  ********
rem    rstat1       0 1         6
rem    rstat2       0 2         6
rem    rstat3       0 3         6
rem    rstat4       0 4         6
rem    rstat5       0 5         6
rem    rstat6       0 6         6
rem    rstat7       0 7         6
rem    rstat8       0 8         6
rem    rstat9       0 9         6
```

RULE1.BAT

```
copy %2 seed.new
stmain %1 1 seed
copy start2 start
copy %2 seed.new
nnmain %1 1 0 1 1 seed %3 1 200 0 0 1 0
```

R2.BAT

```
call rule2 nodes61 r2stat5
call rule3 nodes61 r3stat5
call rule4 nodes61 r4stat5
call rule5 nodes61 r5stat5
call rule6 nodes61 r6stat5
call rule7 nodes61 r7stat5
call rule8 nodes61 r8stat5
cls
rem    Statistics files produced
rem
rem    Output file  cycles  iterations  cheat  samples  agitate  forget
rem    ***********  ******  **********  *****  *******  *******  ******
rem    r2stat5      1       200         no     no       no       no
rem    r3stat5      10      20          no     no       no       no
rem    r4stat5      10      20          yes    no       no       no
rem    r5stat5      10      20          yes    yes      no       no
rem    r6stat5      10      20          yes    yes      yes      no
rem    r7stat5      10      20          yes    yes      no       yes
rem    r8stat5      10      20          yes    yes      yes      yes
```

RULE2.BAT

```
copy seed5 seed.new
stmain %1 9 seed
copy start2 start
copy seed5 seed.new
nnmain %1 9 0 1 1 seed %2 1 200 0 0 1 0
```

RULE3.BAT

```
copy seed5 seed.new
stmain %1 9 seed
copy start2 start
copy seed5 seed new
nnmain %1 9 0 1 10 seed %2 1 20 0 0 1 0
```

RULE4.BAT

```
copy seed5 seed new
stmain %1 9 seed
copy start2 start
copy seed5 seed new
nnmain %1 9 0 1 10 seed %2 1 20 0 1 1 0
```

RULE5.BAT

```
copy seed5 seed.new
stmain %1 27 seed
copy start2 start
copy seed5 seed.new
nnmain %1 9 18 1 10 seed %2 1 20 0 1 1 0
```

RULE6.BAT

```
copy seed5 seed.new
stmain %1 27 seed
copy start2 start
copy seed5 seed.new
nnmain %1 9 18 1 10 seed %2 1 20 0 1 1
```

RULE7.BAT

```
copy seed5 seed.new
stmain %1 27 seed
copy start2 start
copy seed5 seed new
nnmain %1 9 18 1 10 seed %2 1 10 1 1 1 0
```

RULE8.BAT

```
copy seed5 seed new
stmain %1 27 seed
copy start2 start
copy seed5 seed.new
nnmain %1 9 18 1 10 seed %2 1 20 1 1 1
```

TRAIN.BAT

```
call train1
call train4
call train7
call train2
call train5
call train8
cls
rem Rule base is in file "ruleout".
```

TRAIN1.BAT

```
del rulein
del ruleout
copy seed5 seed new
stmain nodes61 27 seed
copy start2 start
nnmain nodes61 9 18 2 10 seed T1stat1 1 10 2 1 1
del rulein
ren ruleout rulein
stmain nodes62 27 seed
copy start2 start
nnmain nodes62 9 18 2 10 seed T2stat1 1 10 2 1 1
del rulein
ren ruleout rulein
stmain nodes63 27 seed
copy start2 start
nnmain nodes63 9 18 2 10 seed T3stat1 1 10 2 1 1
```

TRAIN2.BAT

```
del rulein
ren ruleout rulein
stmain nodes61 27 seed
copy start2 start
nnmain nodes61 9 18 2 10 seed T1stat2 1 6 1 1 1
del rulein
ren ruleout rulein
stmain nodes62 27 seed
copy start2 start
nnmain nodes62 9 18 2 10 seed T2stat2 1 6 1 1 1
del rulein
ren ruleout rulein
stmain nodes63 27 seed
copy start2 start
nnmain nodes63 9 18 2 10 seed T3stat2 1 6 1 1 1
```

TRAIN4.BAT

```
del rulein
ren ruleout rulein
stmain nodes81 27 seed
copy start2 start
nnmain nodes81 9 18 2 10 seed T1stat4 1 10 2 1 1
del rulein
ren ruleout rulein
stmain nodes82 27 seed
copy start2 start
nnmain nodes82 9 18 2 10 seed T2stat4 1 10 2 1 1
del rulein
ren ruleout rulein
stmain nodes83 27 seed
copy start2 start
nnmain nodes83 9 18 2 10 seed T3stat4 1 10 2 1 1
```

TRAIN5.BAT

```
del rulein
ren ruleout rulein
stmain nodes81 27 seed
copy start2 start
nnmain nodes81 9 18 2 10 seed T1stat5 1 10 1 1 1
del rulein
ren ruleout rulein
stmain nodes82 27 seed
copy start2 start
```

```
nnmain nodes82 9 18 2 10 seed T2stat5 1 10 1 1 1
del rulein
ren ruleout rulein
stmain nodes83 27 seed
copy start2 start
nnmain nodes83 9 18 2 10 seed T3stat5 1 10 1 1 1
```

TRAIN7.BAT

```
del rulein
ren ruleout rulein
stmain nodes01 27 seed
copy start2 start
nnmain nodes01 9 18 2 10 seed T1stat7 1 10 2 1 1
del rulein
ren ruleout rulein
stmain nodes02 27 seed
copy start2 start
nnmain nodes02 9 18 2 10 seed T2stat7 1 10 2 1 1
del rulein
ren ruleout rulein
stmain nodes03 27 seed
copy start2 start
nnmain nodes03 9 18 2 10 seed T3stat7 1 10 2 1 1
```

TRAIN8.BAT

```
del rulein
ren ruleout rulein
stmain nodes01 27 seed
copy start2 start
nnmain nodes01 9 18 2 10 seed T1stat8 1 6 1 1 1
del rulein
ren ruleout rulein
stmain nodes02 27 seed
copy start2 start
nnmain nodes02 9 18 2 10 seed T2stat8 1 6 1 1 1
del rulein
ren ruleout rulein
stmain nodes03 27 seed
copy start2 start
nnmain nodes03 9 18 2 10 seed T3stat8 1 6 1 1 1
```

SOLVE.BAT

```
del rulein
ren ruleout rulein
copy seed_5 seed.new
stmain nodes04 30 seed
copy start2 start
nnmain nodes04 0 30 0 0 seed solvstat 0 0 0 1 1
cls
rem Solution statistics are in file "solvstat"
```

D.3. Seed files

Seed files control the random number generation process. The seed files provide a single number between 0 and 1 that starts the pseudo-random number sequence. A program will generate the same random number sequence if the user restores the value

in the seed file to its previous value. Programs overwrite the value in the seed file during each execution. The tests included in the thesis make use of nine seed files which store values of 0.1, 0.2, 0.3, ..., 0.9 in files called seed1, seed2, seed3, ... , seed9 respectively. The programs do not use the nine seed files directly. A batch program copies the contents of the seed file to a temporary file and the program uses the temporary file. This approach protects the contents of the nine files from programs that overwrite their contents.

Appendix D does not include a listing of the seed files since each of these files contains one number only.

D.4. Make Files

All of the programs demonstrated in Chapter 7 use the library of C++ classes included in this appendix. The Watcom C++ Version 9.5 compiler produced the executable versions of the programs used in each of the experiments in Chapter 7. The "make" utility provided with the compiler constructed each program. This appendix includes the "make" files, which are the files that control the make utility and produce the executable files. A single MS-DOS batch program called "make.bat" constructs all the executable program files used in Chapter 7. The batch program uses the make utility and make files. This section includes the file "make.bat" file rather than the Section D.2 covering batch files.

MAKE.BAT

```
wmake /f ndmain.mak
del *.obj
wmake /f lkmain.mak
del *.obj
wmake /f ptmain.mak
del *.obj
```

```
wmake /f epmain.mak
del *.obj
wmake /f esmain.mak
del *.obj
wmake /f ilmain.mak
del *.obj
wpp386 stmain.cpp /4r /7 /ot
wlink f stmain.obj, ilmain.lib
wpp386 nnmain.cpp /4r /7 /ot
wlink f nnmain.obj, ilmain.lib
del *.obj
del *.lib
del *.bak
```

NDMAIN.MAK

```
wpp_options = /4r /7 /ot
link_options =

custio_h = custio.h
nodes_h = nodes.h

ndmain.exe : ndmain.obj ndmain.lib
   wlink $(link_options) f $^&, ndmain.lib
ndmain.obj : ndmain.cpp $(nodes_h)
   wpp386 $^& cpp $(wpp_options)
ndmain.lib    nodes.obj
   wlib $^& -+nodes
nodes.obj    nodes.cpp $(custio_h) $(nodes_h)
   wpp386 $^&.cpp $(wpp_options)
ndmain.lib : custio.obj
   wlib $^& -+custio
custio.obj    custio.cpp $(custio_h)
   wpp386 $^&.cpp $(wpp_options)
```

LKMAIN.MAK

```
wpp_options = /4r /7 /ot
link_options =

custio_h = custio.h
chaos_h = chaos.h
nodes_h = nodes.h
loops2_h = $+ $(nodes_h) $- loops2.h
loops3_h = $+ $(loops2_h) $- loops3.h
links_h = $+ $(chaos_h) $(loops2) $- links.h

lkmain.exe : lkmain.obj lkmain.lib
   wlink $(link_options) f $^&, lkmain.lib
lkmain.obj : lkmain.cpp $(loops3_h)  $(links_h)
   wpp386 $^&.cpp $(wpp_options)
lkmain.lib    links.obj
   wlib $^& -+links
links.obj : links.cpp $(custio_h) $(links_h)
   wpp386 $^&.cpp $(wpp_options)
lkmain.lib : loops3.obj
   wlib $^& -+loops3
loops3.obj    loops3.cpp $(custio_h) $(loops3_h)
   wpp386 $^&.cpp $(wpp_options)
lkmain.lib : loops2.obj
   wlib $^& -+loops2
loops2.obj : loops2.cpp $(custio_h) $(loops2_h)
   wpp386 $^&.cpp $(wpp_options)
lkmain.lib : nodes.obj
   wlib $^& -+nodes
nodes.obj : nodes.cpp $(custio_h) $(nodes_h)
   wpp386 $^&.cpp $(wpp_options)
lkmain.lib : chaos.obj
   wlib $^& -+chaos
chaos.obj    chaos.cpp $(custio_h) $(chaos_h)
   wpp386 $^&.cpp $(wpp_options)
lkmain.lib : custio.obj
   wlib $^& -+custio
```

```
custio.obj : custio.cpp $(custio_h)
   wpp386 $^&.cpp $(wpp_options)
```

PTMAIN.MAK

```
wpp_options = /4r /7 /ot
link_options =

custio_h = custio.h
chaos_h = chaos.h
nodes_h = nodes.h
loops2_h = $+ $(nodes_h) $- loops2.h
loops3_h = $+ $(loops2_h) $- loops3.h
fuzzlink_h = $+ $(chaos_h) $(loops2_h) $- fuzzlink.h

ptmain.exe : ptmain.obj ptmain.lib
   wlink $(link_options) f $^&, ptmain.lib
ptmain.obj : ptmain.cpp $(loops3_h) $(fuzzlink_h)
   wpp386 $^& cpp $(wpp_options)
ptmain.lib : fuzzlink.obj
   wlib $^& -+fuzzlink
fuzzlink.obj : fuzzlink.cpp $(custio_h) $(fuzzlink_h)
   wpp386 $^&.cpp $(wpp_options)
ptmain.lib : loops3.obj
   wlib $^& -+loops3
loops3.obj : loops3.cpp $(custio_h) $(loops3_h)
   wpp386 $^& cpp $(wpp_options)
ptmain.lib : loops2.obj
   wlib $^& -+loops2
loops2.obj : loops2.cpp $(custio_h) $(loops2_h)
   wpp386 $^& cpp $(wpp_options)
ptmain.lib : nodes.obj
   wlib $^& -+nodes
nodes.obj : nodes.cpp $(custio_h) $(nodes_h)
   wpp386 $^&.cpp $(wpp_options)
ptmain.lib : chaos.obj
   wlib $^& -+chaos
chaos.obj : chaos.cpp $(custio_h) $(chaos_h)
   wpp386 $^& cpp $(wpp_options)
ptmain.lib : custio.obj
   wlib $^& -+custio
custio.obj : custio.cpp $(custio_h)
   wpp386 $^& cpp $(wpp_options)
```

EPMAIN.MAK

```
wpp_options = /4r /7 /ot
link_options =

custio_h = custio.h
chaos_h = chaos.h
nodes_h = nodes.h
loops2_h = $+ $(nodes_h) $- loops2.h
loops3_h = $+ $(loops2_h) $- loops3.h
fuzzlink_h = $+ $(chaos_h) $(loops2_h) $- fuzzlink.h
epclass_h = $+ $(loops3_h) $- epclass.h

epmain.exe : epmain.obj epmain.lib
   wlink $(link_options) f $^&, epmain.lib
epmain.obj : epmain.cpp $(epclass_h)
   wpp386 $^&.cpp $(wpp_options)
epmain.lib : epclass.obj
   wlib $^& -+epclass
epclass.obj : epclass.cpp $(epclass_h)
   wpp386 $^& cpp $(wpp_options)
epmain.lib : fuzzlink.obj
   wlib $^& -+fuzzlink
fuzzlink.obj : fuzzlink.cpp $(custio_h) $(fuzzlink_h)
   wpp386 $^&.cpp $(wpp_options)
epmain.lib : loops3.obj
   wlib $^& -+loops3
loops3.obj : loops3.cpp $(custio_h) $(loops3_h)
   wpp386 $^&.cpp $(wpp_options)
```

```
epmain.lib : loops2.obj
   wlib $^& -+loops2
loops2.obj : loops2.cpp $(custio_h) $(loops2_h)
   wpp386 $^&.cpp $(wpp_options)
epmain.lib : nodes.obj
   wlib $^& -+nodes
nodes.obj : nodes.cpp $(custio_h) $(nodes_h)
   wpp386 $^& cpp $(wpp_options)
epmain.lib : chaos.obj
   wlib $^& -+chaos
chaos.obj : chaos.cpp $(custio_h) $(chaos_h)
   wpp386 $^&.cpp $(wpp_options)
epmain.lib : custio.obj
   wlib $^& -+custio
custio.obj : custio.cpp $(custio_h)
   wpp386 $^& cpp $(wpp_options)
```

ESMAIN.MAK

```
wpp_options = /4r /7 /ot
link_options =

custio_h = custio.h
chaos_h = chaos.h
nodes_h = nodes.h
loops2_h = $+ $(nodes_h) $- loops2.h
loops3_h = $+ $(loops2_h) $- loops3.h
fuzzlink_h = $+ $(chaos_h) $(loops2_h) $- fuzzlink.h
esclass_h = $+ $(loops3_h) $- esclass.h

esmain.exe : esmain.obj esmain.lib
   wlink $(link_options) f $^&. esmain lib
esmain.obj : esmain.cpp $(esclass_h)
   wpp386 $^&.cpp $(wpp_options)
esmain.lib : esclass.obj
   wlib $^& -+esclass
esclass.obj : esclass.cpp $(esclass_h)
   wpp386 $^& cpp $(wpp_options)
esmain.lib : fuzzlink.obj
   wlib $^& -+fuzzlink
fuzzlink.obj : fuzzlink.cpp $(custio_h) $(fuzzlink_h)
   wpp386 $^&.cpp $(wpp_options)
esmain.lib : loops3.obj
   wlib $^& -+loops3
loops3.obj : loops3.cpp $(custio_h) $(loops3_h)
   wpp386 $^&.cpp $(wpp_options)
esmain.lib : loops2.obj
   wlib $^& -+loops2
loops2.obj : loops2.cpp $(custio_h) $(loops2_h)
   wpp386 $^& cpp $(wpp_options)
esmain.lib : nodes.obj
   wlib $^& -+nodes
nodes.obj : nodes.cpp $(custio_h) $(nodes_h)
   wpp386 $^&.cpp $(wpp_options)
esmain.lib : chaos.obj
   wlib $^& -+chaos
chaos.obj : chaos.cpp $(custio_h) $(chaos_h)
   wpp386 $^&.cpp $(wpp_options)
esmain.lib : custio.obj
   wlib $^& -+custio
custio.obj : custio.cpp $(custio_h)
   wpp386 $^& cpp $(wpp_options)
```

ILMAIN.MAK

```
wpp_options = /4r /7 /ot
link_options =

custio_h = custio.h
chaos_h = chaos.h
nodes_h = nodes.h
loops2_h = $+ $(nodes_h) $- loops2.h
loops3_h = $+ $(loops2_h) $- loops3.h
```

```
links_h = $+ $(chaos_h) $(loops2) $- links.h
fuzzlink_h = $+ $(chaos_h) $(loops2_h) $- fuzzlink.h
sieve_h = $+ $(links_h) $(loops3_h) $- sieve.h
clump_h = $+ $(links_h) $(loops3_h) $- clump.h
rule_h = $+ $(clump_h) $(fuzzlink_h) $- rule.h
comm_h = $+ $(rule_h) $+ comm.h

ilmain.exe : ilmain.obj ilmain.lib
   wlink $(link_options) f $^&, ilmain.lib
ilmain.obj : ilmain.cpp $(comm_h)
   wpp386 $^&.cpp $(wpp_options)
ilmain.lib : comm.obj
   wlib $^& -+comm
comm.obj : comm.cpp $(custio_h) $(sieve_h) $(comm_h)
   wpp386 $^& cpp $(wpp_options)
ilmain.lib : rule.obj
   wlib $^& -+rule
rule.obj : rule.cpp $(custio_h) $(rule_h)
   wpp386 $^& cpp $(wpp_options)
ilmain.lib : clump.obj
   wlib $^& -+clump
clump.obj : clump.cpp $(clump_h)
   wpp386 $^& cpp $(wpp_options)
ilmain.lib : sieve.obj
   wlib $^& -+sieve
sieve.obj : sieve.cpp $(sieve_h)
   wpp386 $^& cpp $(wpp_options)
ilmain.lib : fuzzlink.obj
   wlib $^& -+fuzzlink
fuzzlink.obj : fuzzlink.cpp $(custio_h) $(fuzzlink_h)
   wpp386 $^& cpp $(wpp_options)
ilmain.lib : links.obj
   wlib $^& -+links
links.obj : links.cpp $(custio_h) $(links_h)
   wpp386 $^& cpp $(wpp_options)
ilmain.lib : loops3.obj
   wlib $^& -+loops3
loops3.obj : loops3.cpp $(custio_h) $(loops3_h)
   wpp386 $^& cpp $(wpp_options)
ilmain.lib : loops2.obj
   wlib $^& -+loops2
loops2.obj : loops2.cpp $(custio_h) $(loops2_h)
   wpp386 $^& cpp $(wpp_options)
ilmain.lib : nodes.obj
   wlib $^& -+nodes
nodes.obj : nodes.cpp $(custio_h) $(nodes_h)
   wpp386 $^& cpp $(wpp_options)
ilmain.lib : chaos.obj
   wlib $^& -+chaos
chaos.obj : chaos.cpp $(custio_h) $(chaos_h)
   wpp386 $^& cpp $(wpp_options)
ilmain.lib : custio.obj
   wlib $^& -+custio
custio.obj : custio.cpp $(custio_h)
   wpp386 $^& cpp $(wpp_options)
```

D.5. C++ Main Files

The programs demonstrated in Chapter 7 use the library of C++ classes included in this appendix. The Watcom C++ Version 9.5 compiler produced the executable versions of the programs used in each of the experiments in Chapter 7. This section list all the C++ source code main files. Each file produces an executable program in conjuction with the C++ class library included in this appendix.

NDMAIN.CPP

```
//-------------------------------------------------------------------
//
//   This is the file ndmain.cpp
//
//-------------------------------------------------------------------

// Implementation dependencies ---------------------------------------

#ifndef _FSTREAM_H_INCLUDED
#include <fstream.h>
#endif

#ifndef __NODES_H
#include "nodes.h"
#endif

// End implementation dependencies -----------------------------------


int main
(
   int argc
   char *argv[]
)
{
   if (argc < 2) {
      cout << "Improper argument specified\n".
      cout <<
         "   Required argument is the name of the file to store node data\n"
   } else {
      Nodes Node_Obj(cin, cout)
      ofstream Out(argv[1]).
      Out << Node_Obj.
      Out.close().
   }
   return 0
}
```

PTMAIN.CPP

```
//-------------------------------------------------------------------

//   This is the file ptmain.cpp

//-------------------------------------------------------------------

// Implementation dependencies ---------------------------------------

#ifndef _FSTREAM_H_INCLUDED
#include <fstream.h>
#endif

#ifndef _STDLIB_H_INCLUDED
#include <stdlib.h>
#endif

#ifndef __LOOPS3_H
#include "loops3.h"
#endif

#ifndef __FUZZLINK_H
#include "fuzzlink.h"
#endif

// End implementaion dependencies ------------------------------------

int main
(
   int argc.
   char *argv[]
)
{
   char *NodeFile:          // Source of nodes data
```

```
    char *SeedFile;             // Seed file for random numbers
    char *StatFile;             // File for output of statistics
    long Iterations;            // Number of iterations
    int Disp;                   // Display mode
    int Block;                  // Flag (1 = improve step not displayed)
    int Pause;                  // Flag (1 = display pauses for key)
    int DMode;                  // Temporary storage for display mode
    long i;                     // Counter of number of trials

    if (argc < 4) {
       cout << "Improper arguments specified\n";
       cout << "    Required arguments are\n";
       cout << "        problem source file\n";
       cout << "        random number seed file\n";
       cout << "        statistics file\n";
       cout << "    Optional arguments\n";
       cout << "        number of iterations\n";
       cout << "        display mode\n";
       cout << "        block display of improvement\n";
       cout << "        pause for key\n";
    } else {
       NodeFile = argv[1];
       SeedFile = argv[2];
       StatFile = argv[3];
       Iterations = 100;
       Disp = 1;
       Block = 1;
       Pause = 0;
       if (argc >= 5) {
          Iterations = atoi(argv[4]);
          if (argc >= 6) {
             Disp = atoi(argv[5]);
             if (argc >= 7) {
                Block = atoi(argv[6]);
                if (argc >= 8)
                   Pause = atoi(argv[7]);
             }
          }
       }

       // Create data structures
       Nodes2 Problem(NodeFile);
       Loop_Mod Tester(Problem, Disp, Block, Pause);
       FuzzLink FTest(&Tester);
       Chaos01 Bits(SeedFile);
       ofstream Out(StatFile);

       // Perform tests
       DMode = Tester.Get_Mode();
       Tester.Set_Mode(0);
       for (i = 0; i < Iterations; ++i) {
          FTest.Set_Count();
          FTest.Init(Bits);
          FTest.Best_Alpha();
          FTest.Remove_Redun(Bits);
          Tester.Init();
          FTest >> Tester;
          Tester.Evaluate();
          Tester.Set_Mode(DMode);
          Tester.Display();
          Tester.Set_Mode(0);
          Out << FTest.Get_Cost() << "\n";
       }
       Out << "Number of trials  " << i << "\n\n";
       Out.close();
    }
    return 0;
}
```

LKMAIN.CPP

```
//-------------------------------------------------------------------
//
//   This is the file lkmain.cpp
//
//-------------------------------------------------------------------
```

```
// Implementation dependencies ----------------------------------------

#ifndef _FSTREAM_H_INCLUDED
#include <fstream.h>
#endif

#ifndef _STDLIB_H_INCLUDED
#include <stdlib.h>
#endif

#ifndef __LOOPS3_H
#include "loops3.h"
#endif

#ifndef __LINKS_H
#include "links.h"
#endif

// End implementaion dependencies -------------------------------------

int main
(
   int argc,
   char *argv[]
)
{
   char *NodeFile;              // Source of nodes data
   char *SeedFile;              // Seed file for random numbers
   char *StatFile;              // File for output of statistics
   long Iterations;             // Number of iterations
   int Disp;                    // Display mode
   int Block;                   // Flag (1 = improve step not displayed)
   int Pause;                   // Flag (1 = display pauses for key)
   int DMode;                   // Temporary storage for display mode
   int Feasible;                // Number of feasible solutions
   long i;                      // Counter of number of trials

   if (argc < 4) {
      cout << "Improper arguments specified\n";
      cout << "   Required arguments are\n";
      cout << "      problem source file\n";
      cout << "      random number seed file\n";
      cout << "      statistics file\n";
      cout << "   Optional arguments\n";
      cout << "      number of iterations\n";
      cout << "      display mode\n";
      cout << "      block display of improvement\n";
      cout << "      pause for key\n";
   } else {
      NodeFile = argv[1];
      SeedFile = argv[2];
      StatFile = argv[3];
      Iterations = 100000;
      Disp = 1;
      Block = 1;
      Pause = 0;
      if (argc >= 5) {
         Iterations = atol(argv[4]);
         if (argc >= 6) {
            Disp = atoi(argv[5]);
            if (argc >= 7) {
               Block = atoi(argv[6]);
               if (argc >= 8)
                  Pause = atoi(argv[7]);
            }
         }
      }

      // Create data structures
      Nodes2 Problem(NodeFile);
      Loop_Mod Tester(Problem, Disp, Block, Pause);
      Links LTest(Problem.Get_Num());
      Chaos_Str Bits(SeedFile);
      ofstream Out(StatFile);

      // Perform tests
      DMode = Tester.Get_Mode();
      Tester.Set_Mode(0);
      Feasible = 0;
      for (i = 0; i < Iterations; ++i) {
```

```
            LTest = Bits;
            Tester.Init();
            LTest >> Tester;
            if (Tester.Test()) {
               Tester.Init();
               LTest >> Tester;
               Tester.Put_T_State(Loop_Base, Feasible);
               Tester.Improve();
               LTest = Tester;
               Tester.Init();
               LTest >> Tester;
               Tester.Put_T_State(Loop_Base, Improved);
               Tester.Evaluate();
               Tester.Set_Mode(DMode);
               Tester.Display();
               Tester.Set_Mode(0);
               // Out << LTest.Get_Cost() << "\n";
               ++Feasible;
            }
         }
         Out << Feasible << " feasible solutions in " << i << " trials\n\n";
         Out.close();
      }
      return 0;
   }
```

EPMAIN.CPP

```
//---------------------------------------------------------------------
//
//   This is the file epmain.cpp
//
//---------------------------------------------------------------------

// Implementation dependencies -----------------------------------------

#ifndef _IOSTREAM_H_INCLUDED
#include <iostream.h>
#endif

#ifndef _STDLIB_H_INCLUDED
#include <stdlib.h>
#endif

#ifndef __EPCLASS_H
#include "epclass.h"
#endif

// End implementation dependencies -------------------------------------

int main
(
   int argc,
   char *argv[]
)
{
   int Pop_Size;              // Population size
   double Rate;               // Mutation rate
   int Iterations;            // Maximum number of iterations
   char *Prob_File;           // Nodes data file
   char *Seed_File;           // Seed file for random numbers
   char *Stat_File;           // File to collect statistics
   int Best_Only;             // Flag (1 = best solution is displayed only)
   int Disp;                  // Display mode
   int Block;                 // Flag (1 = improve step not displayed)
   int Key_Pause;             // Flag (1 = display pauses for key)

   Best_Only = 1;
   Disp = 1;
   Block = 1;
   Key_Pause = 0;
   if (argc < 7) {
      cout << "Improper arguments specified\n";
      cout << "   Required arguments are\n";
      cout << "      population size\n";
      cout << "      mutation rate\n";
      cout << "      number of iterations\n";
```

```
      cout << "       problem source file\n";
      cout << "       random number seed file\n";
      cout << "       statistics file\n";
      cout << "   Optional arguments\n";
      cout << "       best solution only flag\n";
      cout << "       display mode\n";
      cout << "       block display of improvement\n";
      cout << "       pause for key\n";
   } else {
      Pop_Size = atoi(argv[1]);
      Rate = atof(argv[2]);
      Iterations = atoi(argv[3]);
      Prob_File = argv[4];
      Seed_File = argv[5];
      Stat_File = argv[6];
      if (argc >= 8) {
         Best_Only = atoi(argv[7]);
         if (argc >= 9) {
            Disp = atoi(argv[8]);
            if (argc >= 10) {
               Block = atoi(argv[9]);
               if (argc >= 11)
                  Key_Pause = atoi(argv[10]);
            }
         }
      }
      E_P_Class EP(Pop_Size, Rate, Iterations, Prob_File,Seed_File,
                   Stat_File, Best_Only, Disp, Block, Key_Pause);
      EP Run_GA();
   }
   return 0;
}
```

ESMAIN.CPP

```
//---------------------------------------------------------------------------
//
//   This is the file esmain cpp
//
//---------------------------------------------------------------------------

// Implementation dependencies ---------------------------------------

#ifndef _IOSTREAM_H_INCLUDED
#include <iostream.h>
#endif

#ifndef _STDLIB_H_INCLUDED
#include <stdlib.h>
#endif

#ifndef __ESCLASS_H
#include "esclass.h"
#endif

// End implementation dependencies -----------------------------------

int main
(
   int argc,
   char *argv[]
)
{
   double Rate;                 // Mutation rate
   int Iterations;              // Maximum number of iterations
   char *Prob_File;             // Nodes data file
   char *Seed_File;             // Seed file for random numbers
   char *Stat_File;             // File to collect statistics
   int Disp;                    // Display mode
   int Block;                   // Flag (1 = improve step not displayed)
   int Key_Pause;               // Flag (1 = display pauses for key)

   Disp = 1;
   Block = 1;
   Key_Pause = 0;
   if (argc < 6) {
      cout << "Improper arguments specified\n";
```

```
      cout << "   Required arguments are\n";
      cout << "      mutation rate\n";
      cout << "      number of iterations\n";
      cout << "      problem source file\n";
      cout << "      random number seed file\n";
      cout << "      statistics file\n";
      cout << "   Optional arguments\n";
      cout << "      display mode\n";
      cout << "      block display of improvement\n";
      cout << "      pause for key\n";
   } else {
      Rate = atof(argv[1]);
      Iterations = atoi(argv[2]);
      Prob_File = argv[3];
      Seed_File = argv[4];
      Stat_File = argv[5];
      if (argc >= 7) {
         Disp = atoi(argv[6]);
         if (argc >= 8) {
            Block = atoi(argv[7]);
            if (argc >= 9)
               Key_Pause = atoi(argv[8]);
         }
      }
      E_S_Class ES(Rate, Iterations, Prob_File,Seed_File,
                   Stat_File, Disp, Block, Key_Pause);
      ES.Run_ES();
   }
   return 0;
}
```

STMAIN.CPP

```
//---------------------------------------------------------------------
//
//   This is the file stmain.cpp
//
//---------------------------------------------------------------------

// Implementation dependencies ----------------------------------------

#ifndef _FSTREAM_H_INCLUDED
#include <fstream.h>
#endif

#ifndef _STDLIB_H_INCLUDED
#include <stdlib.h>
#endif

#ifndef __SIEVE_H
#include "sieve.h"
#endif

#ifndef __RULE_H
#include "rule.h"
#endif

#ifndef __COMM_H
#include "comm.h"
#endif

// End implementation dependencies ------------------------------------


main
(
   int argc,
   char *argv[]
)
{
   char *NodeFile;            // Source of nodes data
   int Num_Sols;              // Number of starting solutions
   char *SeedFile;            // Seed file for random numbers

   if (argc < 4) {
      cout << "Improper arguments specified\n";
```

```
      cout << "   Required arguments are\n";
      cout << "      problem source file\n";
      cout << "      number of starting solutions\n";
      cout << "      random number seed file\n";
   } else {
      NodeFile = argv[1];
      Num_Sols = atoi(argv[2]);
      SeedFile = argv[3];

      Chaos01 Bits(SeedFile);
      Command LearnObj(NodeFile, Num_Sols, 0, 0, 0, 1, 1, 0);
      LearnObj.Setup_Orig(Bits);
      LearnObj.Write_Orig("start2");

      cout << " " << Num_Sols << " starting solutions written\n\n";
   }
   return 0;
}
```

NNMAIN.CPP

```
//---------------------------------------------------------------------------
//
//   This is the file nnmain.cpp
//
//---------------------------------------------------------------------------

// Implementation dependencies ---------------------------------------

#ifndef _FSTREAM_H_INCLUDED
#include <fstream.h>
#endif

#ifndef _STDLIB_H_INCLUDED
#include <stdlib.h>
#endif

#ifndef __SIEVE_H
#include "sieve.h"
#endif

#ifndef __RULE_H
#include "rule.h"
#endif

#ifndef __COMM_H
#include "comm.h"
#endif

// End implementation dependencies -----------------------------------


main
(
   int argc,
   char *argv[]
)
{
   char *NodeFile,            // Source of nodes data
   int Num_Sols;              // Number of starting solutions
   int Test_Sols;             // Number of starting solutions
   int Num_Cycles;            // Number of learning cycles
   int Cycle_Length;          // Length of learning cycles
   char *SeedFile;            // Seed file for random numbers
   char *StatFile;            // Output statistics file
   int Sam_Length;            // Sample cycle length
   int Num_Sams;              // Number of sample cycles
   int Forget_Flag;           // Flag (1 = use forgetting)
   int Cheating_Flag;         // Flag (1 = use cheating)
   int Sieve_Flag;            // Flag (1 = use sieves and clump)
   int Ag_Flag;               // Flag (1 = use agitator before rules)
   int Disp;                  // Display mode
   int Block;                 // Flag (1 = improve step not displayed)
   int Pause;                 // Flag (1 = display pauses for key)
   Command::Forget_Spec Forget_Choice;
                              // Choice for use of forgetting
```

```
Command::Cheat_Spec Cheat_Choice;
                        // Choice for use of cheating
Command::SC_Spec SC_Choice;
                        // Choice for use of sieves and clumps
Command::Ag_Spec Ag_Choice;
                        // Choice for use of agitator before rules
int Temp_Length;        // Temporary (Sam_Length minus one)
int i;                  // Counter
int j;                  // Counter

if (argc < 13) {
   cout << "Improper arguments specified\n";
   cout << "   Required arguments are\n";
   cout << "      problem source file\n";
   cout << "      number of starting solutions\n";
   cout << "      number of additional testing solutions\n";
   cout << "      number of learning cycles\n";
   cout << "      length of learning cycles\n";
   cout << "      random number seed file\n";
   cout << "      output statistics file\n";
   cout << "      sample cycle length\n";
   cout << "      number of sample cycles\n";
   cout << "      forgetting flag\n";
   cout << "      cheating flag\n";
   cout << "      sieves and clumps flag\n";
   cout << "   Optional arguments\n";
   cout << "      agitator flag\n";
   cout << "      display mode\n";
   cout << "      block display of improvement\n";
   cout << "      pause for key\n";
} else {
   NodeFile = argv[1];
   Num_Sols = atoi(argv[2]);
   Test_Sols = atoi(argv[3]);
   Num_Cycles = atoi(argv[4]);
   Cycle_Length = atoi(argv[5]);
   SeedFile = argv[6];
   StatFile = argv[7];
   Sam_Length = atoi(argv[8]);
   Num_Sams = atoi(argv[9]);
   Forget_Flag = atoi(argv[10]);
   Cheating_Flag = atoi(argv[11]);
   Sieve_Flag = atoi(argv[12]);
   Ag_Flag = 1;
   Disp = 1;
   Block = 1;
   Pause = 0;
   if (argc >= 14) {
      Ag_Flag = atoi(argv[13]);
      if (argc >= 15) {
         Disp = atoi(argv[14]);
         if (argc >= 16) {
            Block = atoi(argv[15]);
            if (argc >= 17)
               Pause = atoi(argv[16]);
         }
      }
   }

   switch (Forget_Flag) {
      case 2:
         Forget_Choice = Command::Use_Full_Forget;
         break;
      case 1:
         Forget_Choice = Command::Partial_Forget;
         break;
      default:
         Forget_Choice = Command::No_Forget;
   }
   if (Cheating_Flag)
      Cheat_Choice = Command::Use_Cheat;
   else
      Cheat_Choice = Command::No_Cheat;
   if (Sieve_Flag)
      SC_Choice = Command::Use_SC;
   else
      SC_Choice = Command::No_SC;
   if (Ag_Flag)
      Ag_Choice = Command::Use_Ag;
   else
```

```
        Ag_Choice = Command::No_Ag;

    Chaos01 Bits(SeedFile);
    Command LearnObj(NodeFile, Num_Sols, Test_Sols, Num_Cycles,
                     Cycle_Length, Disp, Block, Pause);
    ofstream Out(StatFile);
        LearnObj.Setup_Orig(Bits);
        LearnObj.Read_Orig("start", Bits);
        LearnObj.Read_Rule_Base("rulein");
        LearnObj.Test(Out);
        Temp_Length = Sam_Length - 1;
        for (i = 0; i < Num_Sams; ++i) {
           for (j = 0; j < Temp_Length; ++j) {
              LearnObj.Learn(Bits, Forget_Choice, Cheat_Choice,
                             SC_Choice, Ag_Choice);
              LearnObj.Test_No_Write();
           }
           LearnObj.Learn(Bits, Forget_Choice, Cheat_Choice,
                          SC_Choice, Ag_Choice);
           LearnObj.Test(Out);
        }
    Out.close();
    LearnObj.Write_Rules("ruleout", No_Dispose);
    cout << "  Statistics file written\n";
  }
  return 0;
}
```

ILMAIN.CPP

```
//-------------------------------------------------------------------------
//
//   This is the file ilmain.cpp
//
//-------------------------------------------------------------------------

// Implementation dependencies ---------------------------------------

#ifndef _FSTREAM_H_INCLUDED
#include <fstream.h>
#endif

#ifndef _STDLIB_H_INCLUDED
#include <stdlib.h>
#endif

#ifndef __SIEVE_H
#include "sieve.h"
#endif

#ifndef __RULE_H
#include "rule.h"
#endif

#ifndef __COMM_H
#include "comm.h"
#endif

// End implementation dependencies -----------------------------------


main
(
   int argc,
   char *argv[]
)
{
   char *NodeFile;          // Source of nodes data
   int Num_Sols;            // Number of starting solutions
   int Test_Sols;           // Number of starting solutions
   int Num_Cycles;          // Number of learning cycles
   int Cycle_Length;        // Length of learning cycles
   char *SeedFile;          // Seed file for random numbers
   int Disp;                // Display mode
   int Block;               // Flag (1 = improve step not displayed)
   int Pause;               // Flag (1 = display pauses for key)
```

```
    long Num_Rules;            // Number of rules in rule base
    long Num_Read;             // Number of rules read in last read
    long Num_Written;          // Number of rules written in last write

    if (argc < 7) {
       cout << "Improper arguments specified\n";
       cout << "    Required arguments are\n";
       cout << "       problem source file\n";
       cout << "       number of starting solutions\n";
       cout << "       number of additional testing solutions\n";
       cout << "       number of learning cycles\n";
       cout << "       length of learning cycles\n";
       cout << "       random number seed file\n";
       cout << "    Optional arguments\n";
       cout << "       display mode\n";
       cout << "       block display of improvement\n";
       cout << "       pause for key\n";
    } else {
       NodeFile = argv[1];
       Num_Sols = atoi(argv[2]);
       Test_Sols = atoi(argv[3]);
       Num_Cycles = atoi(argv[4]);
       Cycle_Length = atoi(argv[5]);
       SeedFile = argv[6];
       Disp = 1;
       Block = 1;
       Pause = 0;
       if (argc >= 8) {
          Disp = atoi(argv[7]);
          if (argc >= 9) {
             Block = atoi(argv[8]);
             if (argc >= 10)
                Pause = atoi(argv[9]);
          }
       }

       Chaos01 Bits(SeedFile);
       Command LearnObj(NodeFile, Num_Sols, Test_Sols, Num_Cycles,
                        Cycle_Length, Disp, Block, Pause);
       LearnObj.Run(Bits);
       Num_Rules = LearnObj.Get_Num_Rules();
       Num_Read = LearnObj.Get_Num_Read();
       Num_Written = LearnObj.Get_Num_Written();

       cout << "  Number of rules in rule base          "
            << Num_Rules << "\n";
       cout << "  Number of rules read in last read     "
            << Num_Read << "\n";
       cout << "  Number of rules written in last write "
            << Num_Written << "\n\n\n";
    }
    return 0;
}
```

D.6. C++ Class Library

The programs demonstrated in Chapter 7 use the library of C++ classes included in this section. The Watcom C++ Version 9.5 compiler produced the executable versions of the programs used in each of the experiments in Chapter 7.

CUSTIO.H

```
//---------------------------------------------------------------------
//
//   This is the file custio.h
//
```

```
//-----------------------------------------------------------------------

// Interface dependencies ------------------------------------------

#ifndef _IOSTREAM_H_INCLUDED
#include <iostream.h>
#endif

// End interface dependencies --------------------------------------

#ifndef __CUSTIO_H
#define __CUSTIO_H

#ifndef NULL
#define NULL 0L
#endif

//-----------------------------------------------------------------------
//
//   The routines below provide for formatted output of integers
//   and doubles.
//
//-----------------------------------------------------------------------
void Set_Int(int);                          // Sets width for integers
void Set_Double(int, int);                  // Sets width and precision for
                                            // doubles
ostream &Cust_I(ostream&);                  // Manipulator for integer
ostream &operator <<(ostream&, int);        // Insertor for integer
ostream &Cust_D(ostream&);                  // Manipulator for double
ostream &operator <<(ostream&, double);     // Insertor for double


//-----------------------------------------------------------------------
//
//   The routines below manipulate and test character strings
//   intended to be used as file names.
//
//-----------------------------------------------------------------------
char *Filter_File_Name(char*  char*);       // Ensures that file name is
                                            // valid
int Alpha_Num(char);                        // Tests char to see if it is
                                            // alphanumeric
unsigned short Pause_Key();                 // Pauses until a key is pressed
                                            // handles special function keys

#endif
```

CUSTIO.CPP

```
//-----------------------------------------------------------------------
//
//   This is the file custio.cpp
//
//-----------------------------------------------------------------------

// Implementation dependencies -------------------------------------

#ifndef _MATH_H_INCLUDED
#include <math.h>
#endif

#ifndef _CONIO_H_INCLUDED
#include <conio.h>
#endif

#ifndef __CUSTIO_H
#include "custio.h"
#endif

// End implemenation dependencies ----------------------------------

static int Int_Width;          // Width of integer output
static int Double_Width;       // Width of double precision output
static int Double_Precision;   // Number of decimal places for double
                               // precision

void Set_Int
//-----------------------------------------------------------------------
```

```
//
//   Sets the width of integer output using < as insertor.
//
//-------------------------------------------------------------------
{
// Input Parameter
      int Width               // New width of integer output
}
{
   Int_Width = Width;
   return;
}

void Set_Double
//-------------------------------------------------------------------
//
//   Sets the width of integer output using < as insertor.
//
//-------------------------------------------------------------------
{
// Input Parameters
      int Width;              // New width of double precision output
      int Precision           // Number of decimal places
}
{
   Double_Width = Width;
   Double_Precision = Precision;
   return;
}

ostream &Cust_I
//-------------------------------------------------------------------
//
//   Output manipulator
//
//   Sets the width of integer to be printed.  Width specified
//   by Int_Width
//
//   ostream << Cust_I << int
//
//-------------------------------------------------------------------
{
// Input Parameter
      ostream &Out            // Output stream
}
{
   Out.setf(ios::showpoint);
   Out.width(Int_Width);
   return Out;
}

ostream &operator <
//-------------------------------------------------------------------
//
//   Prints an integer with width specified by Int_Width.
//   Use brackets if used with <<
//
//   (ostream < int)
//
//-------------------------------------------------------------------
{
// Input Parameters
      ostream &Out;           // Output stream
      int X                   // Integer to be formatted
}
{
   Out.setf(ios::showpoint);
   Out.width(Int_Width);
   Out << X;
   return Out;
}

ostream &Cust_D
//-------------------------------------------------------------------
//
//   Output manipulator
//
//   Sets the width and precision of double to be printed.  Width
//   and precision specified by Double_Width and Double_Precision.
//
```

```
//   ostream << Cust_D << double
//
  ------------------------------------------------------------------------
//
//  Input Parameter
      ostream &Out           // Output stream
)
{
   Out.unsetf(ios::scientific);
   Out.setf(ios::showpoint | ios::fixed);
   Out.width(Double_Width);
   Out.precision(Double_Precision);
   return Out;
}

  ------------------------------------------------------------------------

//   Prints a double with width specified by Double_Width and
//   precision specified by Double_Precision  Use brackets if
//   used with <<
//
//      (ostream < int)
//
//------------------------------------------------------------------------
ostream &operator <
(
// Input Parameters
      ostream &Out,          // Output stream
      double X               // Double to be formatted
)
{
   Out.unsetf(ios::scientific);
   Out.setf(ios::showpoint | ios::fixed);
   Out.width(Double_Width);
   Out.precision(Double_Precision);
   Out << X;
   return Out;
}

char *Filter_File_Name
//------------------------------------------------------------------------

//   This routine ensures that the file name consists of legal
//   characters and is of the proper length  If the extension is
//   not specified one of " new" is added
//
  ------------------------------------------------------------------------
(
//  Output Parameter
      char *Dest             // Destination file name
// Input Parameter
      char *Source           // Source file name
)
{
// Local variables
      char * Out_Ptr;        // Output pointer (null if name fails)
      int i, j;              // Counters


   Out_Ptr = Dest;
   for (i = 0; Alpha_Num(Source[i]) && (i < 8); ++i)
      Dest[i] = Source[i];
   if (i == 0) {
      Out_Ptr = NULL;
   } else {
      if (Source[i] == '.') {
         Dest[i] = Source[i];
         for (++i, j = 0; Alpha_Num(Source[i]) && (j < 3); ++i, ++j)
            Dest[i] = Source[i];
      } else {
         Dest[i++] = '.';
         Dest[i++] = 'n';
         Dest[i++] = 'e';
         Dest[i] = 'w';
      }
   }
   Dest[++i] = 0;
   return Out_Ptr;
}
```

```
int Alpha_Num
//---------------------------------------------------------------------
//
//   This rountine determines if a character is alphanumeric
//   1 is returned if it is and 0 if it is not.
//
//---------------------------------------------------------------------
{
// Input parameters
      char C                  // Character to be tested
)
{
// Local variables
      int AN_Flag;            // Flag (0 = charater not aphanumeric)

   AN_Flag = 0;
   if (((C >= '0') && (C <= '9')) ||
       ((C >= 'A') && (C <= 'Z')) ||
       ((C >= 'a') && (C <= 'z')))
      AN_Flag = 1;
   return AN_Flag;
}

unsigned short Pause_Key()
//---------------------------------------------------------------------
//
//   This rountine pauses until a key is pressed. The ASCII value of
//   the key is returned  If the key is a special function key that
//   returns two bytes (0 and a second byte) the value of the second
//   byte + 128 is returned.
//
//---------------------------------------------------------------------
{
// Local variables
      unsigned short Key;     // Temporary storage for the key value

   cout.flush();
   while (!kbhit())
   Key = (unsigned short) getch();
   if (Key == 0)
      Key = ((unsigned short) getch()) + 128;
   return Key;
}
```

CHAOS.H

```
//---------------------------------------------------------------------
//
//   This is the file chaos.h
//
//---------------------------------------------------------------------

// Interface dependencies ---------------------------------------------

#ifndef _IOSTREAM_H_INCLUDED
#include <iostream.h>
#endif

// End interface dependencies -----------------------------------------

#ifndef __CHAOS_H
#define __CHAOS_H

#define DEFAULT_A 3.9999

class Chaos_Base {
//---------------------------------------------------------------------
//
//   This is the base class for all chaotic objects. Seed file
//   operations are maintained in this class.
//
//---------------------------------------------------------------------
protected:
   char *Seed_File;           // Name of the file containing the seed value
   double A;                  // Coefficient of parabolic function
   double X;                  // Chaotic variable
```

```
// Implementation functions
   double Next();              // Calculates and returns next value

public:
   Chaos_Base(){}              // Vanilla constructor
   Chaos_Base(char *A, double B = DEFAULT_A); // { Create(A, B); }
                               // Actual constructor
   virtual ~Chaos_Base();      // Saves the last value of the variable
};

class Chaos01 : public Chaos_Base {
//---------------------------------------------------------------------
//
//   Objects of this class are used to generate a chaotic sequence
//   of numbers with values between 0 and 1.
//
//   operators
//      Chaos01(char*, double)
//     ++Chaos01
//
//---------------------------------------------------------------------
public:
   Chaos01(){}                 // Vanilla constructor
   Chaos01(char *A, double B = DEFAULT_A) : Chaos_Base(A, B) {}
                               // Creates and initializes the object
   double operator ++(){ return Chaos_Base::Next(); }
                               // Calculates and returns next value
};

class Chaos_Str : public Chaos_Base {
//---------------------------------------------------------------------
//
//   Objects of this class generate a chaotic sequence of bits
//   based on converting the significand of a double precision
//   number to a string of bits.
//
//   operators
//      Chaos_Str(char*)
//      ++Chaos_Str
//      ostream << Chaos_Str
//
//---------------------------------------------------------------------
protected:
   int Bit_Pointer;            // Points to the last outputted bit (0 - 7)
   int Byte_Pointer;           // Points to the byte in which the last
                               // outputted bit occurs
   char *Chaos_Bits;           // Seven bytes of data corresponding to the
                               // significand of a double precision number
                               // 52 bits (6.5 bytes) are used

// Implementation functions
   void Next_Bytes();          // Computes the next bytes in the series

public
   Chaos_Str(){}               // Vanilla constructor
   Chaos_Str(char*, double = DEFAULT_A);
                               // Allocates space and initalizes object
   virtual ~Chaos_Str();       // Frees space
   int operator ++();          // Get the next bit in the series
   friend ostream &operator <<(ostream&, Chaos_Str&);
                               // Print operator
};

class Chaos_Range : public Chaos_Base {
//---------------------------------------------------------------------
//
//   Objects of this class are used to generate a chaotic sequence
//   of integer numbers between zero and specified number (inclusive
//   of zero, exclusive of the specified number).  The upper bound
//   is specified when the object is instantiated.
//
//   operators
//      Chaos_Range(char*, int, double)
//      ++Chaos_Range
//
//---------------------------------------------------------------------
protected:
   double Range;               // Specifies the range of the output values

public:
```

```
   Chaos_Range(){}            // Vanilla constructor
   Chaos_Range(char*, int, double = DEFAULT_A);
                              // Initializes the variable and specifies
                              // the range
   virtual ~Chaos_Range(){}   // Destructor
   int operator ++();         // Calculates and returns the next value
};

class Spec_Range_Base : public Chaos_Base {
//---------------------------------------------------------------------
//
//   This class returns a chaotic integer ranging from 0 to an
//   an upper bound specifed as an input parameter to "Next".
//
//---------------------------------------------------------------------
protected:

// Implementation functions
   int Next(int);             // Calculates and returns the next value

public:
   Spec_Range_Base(){}        // Vanilla constructor
   Spec_Range_Base(char *A, double B = DEFAULT_A) : Chaos_Base(A, B) {}
                              // Actual constructor
   virtual ~Spec_Range_Base(){}
                              // Destructor
};

class Spec_Range : public Spec_Range_Base {
//---------------------------------------------------------------------
//
//   Objects of this class are used to generate a chaotic sequence
//   of integer numbers between zero and specified number (inclusive
//   of zero, exclusive of the specified number).  The upper bound
//   is specified using the * operator.
//
//   operators
//      Spec_Range(char*, double)
//      Spec_Range * int
//
//---------------------------------------------------------------------
public:
   Spec_Range(){}             // Vanilla constructor
   Spec_Range(char *A, double B = DEFAULT_A) : Spec_Range_Base(A, B) {}
                              // Creates and initializes the object
   virtual ~Spec_Range(){}    // Destructor
   int operator *(int A){ return Next(A); }
                              // Calculates and returns the next value
};

class Mult_Range : public Spec_Range_Base {
//---------------------------------------------------------------------
//
//   Objects of this class return integer numbers generated
//   chaotically.  The same number is never returned twice.  When
//   no numbers remain the function returns the upper bound of
//   the range.  The numbers are selected then removed from an array.
//   therefore the object should only be used with a small range of
//   numbers.
//
//   operators
//      Mult_Range(char*, int, double)
//      --Mult_Range
//      ++Mult_Range
//
//---------------------------------------------------------------------
protected:
   int Start_Range;           // Initial upper bound of values to be chosen
   int Current_Range;         // Current upper bound
   int *Pick;                 // Pick array
                              // range of values

public:
   Mult_Range(){}             // Vanilla constructor
   Mult_Range(char*, int, double = DEFAULT_A);
                              // Allocates space and initializes the object
   virtual ~Mult_Range();     // Frees space and updates seed file
   Mult_Range &operator --();
                              // Initializes the object
   int operator ++();         // Choses the next number not repeating the last
```

```
};

#endif
```

CHAOS.CPP

```
//---------------------------------------------------------------------------
//
//   This is the file chaos.cpp
//
//---------------------------------------------------------------------------

// Implementation dependencies ----------------------------------------

#ifndef _STDLIB_H_INCLUDED
#include <stdlib.h>
#endif

#ifndef _FSTREAM_H_INCLUDED
#include <fstream.h>
#endif

#ifndef __CUSTIO_H
#include "custio.h"
#endif

#ifndef __CHAOS_H
#include "chaos.h"
#endif

// End implementation dependencies ------------------------------------

double Chaos_Base::Next()
//---------------------------------------------------------------------------
//
//   This routine calculates the next value in the chaotic series.
//
//---------------------------------------------------------------------------
{
   if ((X <= 0.0) || (X >= 1.0))
      X = 0.5;
   else
      X = A * X * (1.0 - X);
   return X;
}

Chaos_Base::Chaos_Base
//---------------------------------------------------------------------------
//
//   This constructor looks for a file with a name specified by
//   "File_Name".  If this file is not found the seed value is
//   initialized at 0.5.  Otherwise the value is read from the
//   file.
//
//---------------------------------------------------------------------------
(
// Input parameters
      char *File_Name         // The name of the file containing the seed
      double A_Input          // The coefficient in the parabolic function
)
{
   Seed_File = new char[13];
   A = A_Input;
   if (Filter_File_Name(Seed_File, File_Name)) {
      ifstream Seed(Seed_File);
         if (!Seed)
            X = 0.5;
         else {
            Seed >> X;
            Seed.close();
         }
   } else {
      cerr << "\n<Chaos_Base 01> Error: illegal file name " << File_Name << "\n";
      Pause_Key();
      exit(1);
   }
   if ((A < 3.5) || (A >= 4.0)) {
```

```
      cerr << "\n<Chaos_Base 02> Error: illegal coefficient associated with "
           << File_Name << "\n";
      exit(1);
   }
}

Chaos_Base::~Chaos_Base()
//---------------------------------------------------------------------
//
//   This routine updates the file containing the seed value
//   and closes the file.
//
//---------------------------------------------------------------------
{
   ofstream Seed(Seed_File);
      Seed << X << "\n";
   Seed.close();
   delete Seed_File;
}

void Chaos_Str::Next_Bytes()
//---------------------------------------------------------------------
//
//   This routine calculates the next number in the chaotic
//   sequence and converts it to a bit string.  This is done by
//   converting the significand of a double precision number between
//   0 and 1 to a string of bits.  1 is added to the number to
//   correctly position the significand before converting
//
//---------------------------------------------------------------------
{
// Local variables
      double Bits;            // Double precision number to be converted
      char Mask_High;         // Masks out upper 4 bits of a byte
      char *Byte;             // Pointer to a byte
      int i;                  // Counter

   Bits = Next();
   Mask_High = 0x0F;
   // Position significand of Bits
   ++Bits;
   // Convert Bits to string of bytes
   Byte = (char*) &Bits;
   // Copy low order bytes of significand
   for (i = 0; i < 6; ++ i)
      Chaos_Bits[i] = Byte[i];
   // Copy remaining 4 bits of significand
   Chaos_Bits[6] = Byte[i] & Mask_High;
   return;
}

Chaos_Str::Chaos_Str
//---------------------------------------------------------------------
//
//   This routine initializes the X variable using the file
//   specified and records the value provided for the coefficient
//   of the parabolic equation.
//
//---------------------------------------------------------------------
{
// Input parameters
      char *InFile;           // The name of the file containing the seed
      double New_A            // The coefficient in the parabolic function
)
  Chaos_Base(InFile, New_A)
{
   Chaos_Bits = new char[7];
   Bit_Pointer = Byte_Pointer = 0;
   Next_Bytes();
}

Chaos_Str::~Chaos_Str()
//---------------------------------------------------------------------
//
//   This destructor frees space occupied by "Chaos_Bits".
//
//---------------------------------------------------------------------
{
   delete Chaos_Bits;
}
```

```
int Chaos_Str::operator ++()
//---------------------------------------------------------------------
//
//   Gets the next bit in the sequence.  If the last bit was
//   outputted in the previous call a new string of bits is
//   calculated and the pointer reset.
//
//---------------------------------------------------------------------
{
// Local variables
      char Mask.              // Masks all but the last bit in a byte

   Mask = 1;
   if (Byte_Pointer == 6)
      if (Bit_Pointer == 3) {
         Next_Bytes();
         Byte_Pointer = Bit_Pointer = 0;
      } else
         ++Bit_Pointer;
   else
      if (Bit_Pointer == 7) {
         ++Byte_Pointer;
         Bit_Pointer = 0;
      } else
         ++Bit_Pointer;
   return (int) (Mask & (Chaos_Bits[Byte_Pointer] >> Bit_Pointer));
}

ostream &operator <<
//---------------------------------------------------------------------
//
//   This routine prints out the chaos string as a 52 bit string
//
//---------------------------------------------------------------------
(
// Input parameters
      ostream &Out.           // Output stream
      Chaos_Str &Str          //  Output object
)
{
// Local variables
      char Mask.              // Masks all but last bit
      int i, j.               // Counters

   Set_Int(1);
   Mask = 1;
   for (j = 3; j >= 0; --j)
      Out << (Mask & (Str.Chaos_Bits[6] >> j));
   for (i = 5; i >= 0; --i)
      for (j = 7; j >= 0; --j)
         Out << (Mask & (Str.Chaos_Bits[i] >> j));
   Out << "\n";
   return Out;
}

Chaos_Range::Chaos_Range
//---------------------------------------------------------------------
//
//   This routine initializes the variable X using the file
//   specified and initializes the range and coefficient variables.
//
//---------------------------------------------------------------------
(
// Input parameters
      char *File_Name.        // File containing seed value
      int New_Range.          // Range of integer values
      double New_A            // Coefficient of equation
)
  Chaos_Base(File_Name, New_A)
{
   Range = (double) New_Range;
}

int Chaos_Range::operator ++()
//---------------------------------------------------------------------
//
//   This routine calculates the next X value in the chaotic
//   series and returns the value of of an integer within the
//   previously specified range.
```

```
//
//-------------------------------------------------------------------------
{
   return (int) (Range * Next()).
}

int Spec_Range_Base::Next
//-------------------------------------------------------------------------
//
//   This routine calculates the next X value in the chaotic
//   series and returns the value of an integer within the
//   range specified as the input parameter.
//
//-------------------------------------------------------------------------
(
// Input parameter
   int Range                  // Upper bound on output
)
{
   return (int) (((double) Range) * Chaos_Base::Next()).
}

Mult_Range::Mult_Range
//-------------------------------------------------------------------------
//
//   This routine allocates space for the chooser, a Spec_Range
//   object, and the pick array. The pick array is then initialized
//   using the operator --.
//
//-------------------------------------------------------------------------
(
// Input Parameters
      char *Seed_File,        // Name of seed file
      int New_Start,          // Upper bound of integer values
      double New_A            // Coefficient of equation
)
  Spec_Range_Base(Seed_File, New_A)
{
   Start_Range = New_Start;
   Pick = new int[Start_Range];

   // Initialize the array of pick values
   operator --();
}

Mult_Range::~Mult_Range()
//-------------------------------------------------------------------------
//
//   This is a destructor that free space taken by the pick array.
//   The chooser is eliminated automatically by its destructor.
//
//-------------------------------------------------------------------------
{
   delete Pick;
}

Mult_Range &Mult_Range::operator --()
//-------------------------------------------------------------------------
//
//   This routine initializes the pick array. The array contains
//   integer numbers from 0 to 1 less than the upper bound. As each
//   number is chosen it is eliminated from the array.
//
//-------------------------------------------------------------------------
{
// Local variables
      int i;                  // Counter

   Current_Range = Start_Range;
   for (i = 0; i < Current_Range; ++i)
      Pick[i] = i;
   return *this;
}

int Mult_Range::operator ++()
//-------------------------------------------------------------------------
//
//   This routine selects, removes and returns a number from
//   the pick array. No number will be returned twice. If all
//   numbers have been used the value of the initial range is
```

```
//   returned.
//
//-------------------------------------------------------------------
{
// Local variables
      int Removed;            // Element chosen
      int Choice;             // Index of the chosen element
      int i, j;               // Counters

   if (Current_Range == 0) {
      Removed = Start_Range;
   } else {
      Choice = Next[Current_Range];
      Removed = Pick[Choice];
      for (i = 0, j = 0; i < Current_Range; ++i)
         if (i != Choice) {
            Pick[j] = Pick[i];
            ++j;
         }
      Current_Range = j;
   }
   return Removed;
}
```

NODES.H

```
//-------------------------------------------------------------------
//
//   This is the file nodes.h
//
//-------------------------------------------------------------------

// Interface dependencies ---------------------------------------------

#ifndef _IOSTREAM_H_INCLUDED
#include <iostream.h>
#endif

// End interface dependencies -----------------------------------------

#ifndef __NODES_H
#define __NODES_H

struct Node_Type {
   double X;                  // X-coordinate of a node
   double Y;                  // Y-coordinate of a node
};

class Nodes {
//-------------------------------------------------------------------
//
//   Objects of this class are initialized by reading node
//   data, i.e., x and y coordinates, from an ASCII file.
//
//   operators
//      Nodes(char*)
//      Nodes(istream&, ostream&)
//      ostream << Nodes
//
//   interface functions
//      int Get_Num()
//
//-------------------------------------------------------------------
protected
   int Num_Nodes;             // The number of basic vertices
   Node_Type *Base_Node;      // Array of original nodes

// Implementation functions
   virtual void Print(ostream&);
                              // Prints out object

public:
   Nodes(){}                  // Vanilla constructor
   Nodes(char*);              // Allocates space for array and reads
                              // coordinates from file
   Nodes(istream&, ostream&);
                              // Gets node data from an input stream
```

```
   virtual ~Nodes();          // Destructor - frees allocated space
   friend ostream &operator <<(ostream&, Nodes&);
                              // Operator that prints object to output
                              // stream

// Interface functions
   int Get_Num() const { return Num_Nodes; }
                              // Returns the number of nodes
};

class Nodes2   public Nodes {
//---------------------------------------------------------------------
//
//   Objects of this class read coordinates of nodes from an
//   ASCII file, sort the coordinates, assemble the nodes matrix
//   and record the location of the source node. The source is
//   assumed to be the first node in the file.
//
//   operators
//      Nodes2(char*)
//      ostream << Nodes2
//
//   interface functions
//      int Get_Num()
//      int Get_Node(int, int)
//      double Get_X(int)
//      double Get_Y(int)
//      double Get_DX(int)
//      double Get_DY(int)
//      int Get_Source_Row()
//      int Get_Source_Col()
//
//---------------------------------------------------------------------
protected:
   double *X;                 // Array of x-coordinates
   double *Y;                 // Array of y-coordinates
   double *DX;                // Lengths of east links
   double *DY;                // Lengths of south links
   int **Nodes0;              // Nodes matrix
   int Source_Row;            // The row in which the source is found
   int Source_Col;            // The column in which the source is found

// Implementation functions
   void Set_Up();             // Sets up remaining data structures
   void Sort_X_Y()            // Sorts arrays X and Y in ascending order
   void D_Calc();             // Creates the vectors of east and south
                              // lengths
   void Create_Nodes();       // Creates nodes matrix
   virtual void Print(ostream&);
                              // Prints out object

public
   Nodes2(){}                 // Vanilla constructor
   Nodes2(istream&, ostream&);
                              // Gets node data from an input stream
   Nodes2(char*);             // Allocates space for arrays
                              // and calculates the contents
   virtual ~Nodes2();         // Frees space allocated
   friend ostream &operator <<(ostream&, Nodes2&);

// Interface functions
   int Get_Node(int i, int j) const { return Nodes0[i][j]; }
   double Get_X(int i) const { return X[i]; }
   double Get_Y(int i) const { return Y[i]; }
   double Get_DX(int i) const { return DX[i]; }
   double Get_DY(int i) const { return DY[i]; }
   int Get_Source_Row() const { return Source_Row; }
   int Get_Source_Col() const { return Source_Col; }
};

#endif
```

NODES.CPP

```
//---------------------------------------------------------------------
//
//   This is the file nodes.cpp
```

```
//
//-------------------------------------------------------------------------

// Implementation dependencies ---------------------------------------

#ifndef _FSTREAM_H_INCLUDED
#include <fstream.h>
#endif

#ifndef __CUSTIO_H
#include "custio.h"
#endif

#ifndef __NODES_H
#include "nodes.h"
#endif

// End implementation dependencies -----------------------------------

void Nodes::Print
//-------------------------------------------------------------------------
//
//   This routine prints out the node data in table form.
//
//   Number of records at byte 22
//   First record at byte 87
//   Record Length is 29
//
//-------------------------------------------------------------------------
(
// Input parameter
    ostream& Out            // Output stream
)
{
// Local variables
    int i;                  // Counter

  Set_Int(3);
  Set_Double(10, 1);
  (Out << "Total number of nodes " << Num_Nodes) << "\n\n";
  Out << "Num          X           Y \n";
  Out << "_____________________________\n";
  for (i = 0; i < Num_Nodes; ++i) {
    (Out << i + 1) <<"   ";
    (Out << Base_Node[i].X) << "   ";
    (Out << Base_Node[i].Y) << "\n";
  }
  Out << "\n\n";
  return;
}

Nodes::Nodes
//-------------------------------------------------------------------------
//
//   This constructor reads the file "Infile".  The first entry is the
//   number of basic vertices (sources and sinks) in the problem.
//   The function will read this number, allocate space for an array
//   of coordinates, and read the coordinates into the array.
//
//   Number of records at byte 22
//   First record at byte 87
//   Record Length is 29
//
//-------------------------------------------------------------------------
(
// Input parameter
    char *Infile            // Name of file containing node data
)
{
// Local variables
    int i;                  // Array index
    int j;                  // Pointer to start of record

  ifstream In(Infile);

    // Read in number of nodes
    In.seekg(22, ios::beg);
    In >> Num_Nodes;

    // Allocate space for array
```

```
      Base_Node = new Node_Type[Num_Nodes];

      // Read in coordinates of nodes
      j = 87;
      for (i = 0; i < Num_Nodes; ++i) {
         In.seekg(j + 5, ios::beg);
         In >> Base_Node[i].X;
         In.seekg(j + 17, ios::beg);
         In >> Base_Node[i].Y;
         j += 29;
      }
   In.close();
}

Nodes::Nodes()
//---------------------------------------------------------------------
//
//   This constructor prompts the user for node input from the keyboard
//   The first number inputted is the number of original nodes  Space
//   is allocated on this basis  Then the routine asks for the x and
//   y coordinates of each node
//
//---------------------------------------------------------------------
{
// Input parameters
      istream &In,             // Input stream
      ostream &Out             // Output stream
)
{
// Local variables
      int Input;               // Flag (1 = input required)
      int i;                   // Counter

   // Read in number of nodes
   Input = 1;
   while (Input) {
      Out << "Input the number of nodes: ";
      In >> Num_Nodes;
      if (Num_Nodes > 0) Input = 0;
   }
   Out << "\n";

   // Allocate space for array
   Base_Node = new Node_Type[Num_Nodes];

   // Read in coordinates of nodes
   for (i = 0; i < Num_Nodes; ++i) {
      Out << "Input the x coordinate for node " << i + 1 << ": ";
      In >> Base_Node[i].X;
      Out << "Input the y coordinate for node " << i + 1 << ": ";
      In >> Base_Node[i].Y;
      Out << "\n";
   }
}

Nodes::~Nodes()
//---------------------------------------------------------------------
//
//   This destructor frees space that was dynamically allocated
//   by the Create routines
//
//---------------------------------------------------------------------
{
   delete Base_Node;
}

ostream &operator <<(ostream &A, Nodes &B) { B.Print(A); return A; }

void Nodes2::Set_Up()
//---------------------------------------------------------------------
//
//   The Nodes0 array is initialized to zero then routines
//   are called to create the contents of these structures.
//
//---------------------------------------------------------------------
{
// Local variables
      int i, j;                // Counters

   // Allocate space for arrays
```

```
   X = new double[Num_Nodes];
   Y = new double[Num_Nodes];
   DX = new double[Num_Nodes];
   DY = new double[Num_Nodes];
   Nodes0 = new int*[Num_Nodes];
   for (i = 0; i < Num_Nodes; ++i)
      Nodes0[i] = new int[Num_Nodes];

   // Initialize Nodes0 array
   for (i = 0; i < Num_Nodes; ++i)
      for (j = 0; j < Num_Nodes; ++j)
         Nodes0[i][j] = 0;
   Sort_X_Y();
   D_Calc();
   Create_Nodes();
   return;
}

void Nodes2::Sort_X_Y()
//---------------------------------------------------------------------------
//
//   This subroutine assigns values to the elements in this array
//   (from the array Base_Nodes), and sorts the elements of the
//   x array in ascending order and y in descending order using a
//   simple bubble sort.
//
//---------------------------------------------------------------------------
{
// Local variables
      int N;                    // The number of original nodes - 1
      double Temp;              // Temporary variable
      int i, j;                 // Counters

   // Calculate size of arrays
   N = Num_Nodes - 1;

   // Assign initial values
   for (i = 0; i < Num_Nodes; ++i) {
      X[i] = Base_Node[i].X;
      Y[i] = Base_Node[i].Y;
   }

   // Sort elements
   for (i = 0; i < N; ++i)
      for (j = N; j > i; --j) {
         if (X[j - 1] > X[j]) {
            Temp = X[j - 1];
            X[j - 1] = X[j];
            X[j] = Temp;
         }
         if (Y[j - 1] < Y[j]) {
            Temp = Y[j - 1];
            Y[j - 1] = Y[j];
            Y[j] = Temp;
         }
      }
   return;
}

void Nodes2::D_Calc()
//---------------------------------------------------------------------------
//
//   This function calculates the east and south lengths based on
//   the sorted arrays X and Y.
//
//---------------------------------------------------------------------------
{
// Local variables
      int N;                    // The number of nodes - 1
      int i;                    // Counter

   // Calculate size of array
   N = Num_Nodes - 1;

   // Assign values
   for (i = 0; i < N; ++i) {
      DX[i] = X[i + 1] - X[i];
      DY[i] = Y[i] - Y[i + 1];
   }
   DX[N] = DY[N] = 0.0;
```

```
   return;
}

void Nodes2::Create_Nodes()
//---------------------------------------------------------------------
//
//   This function creates the nodes matrix for the problem. Space
//   is allocated for the matrix. Next the matrix is initialized with
//   zeros. Each of the basic vertices is then tested against the x
//   and y coordinate vectors to determine the node's position in the
//   nodes matrix.
//
//---------------------------------------------------------------------
{
// Local variables
      int i, j, k;            // Counters

   // Assign a value of 1 for each original node
   for (i = 0; i < Num_Nodes; ++i)
      if (Base_Node[0].Y == Y[i]) break;
   Source_Row = i;
   for (j = 0; j < Num_Nodes; ++j)
      if (Base_Node[0].X == X[j]) break;
   Source_Col = j;
   Nodes0[i][j] = 1;
   for (k = 1; k < Num_Nodes; ++k) {
      for (i = 0; i < Num_Nodes; ++i)
         if (Base_Node[k].Y == Y[i]) break;
      for (j = 0; j < Num_Nodes; ++j)
         if (Base_Node[k].X == X[j]) break;
      Nodes0[i][j] = 1;
   }
   return;
}

void Nodes2::Print
//---------------------------------------------------------------------
//
//   This routine prints out the object in table form.
//
//---------------------------------------------------------------------
(
// Input parameter
      ostream &Out            // Output stream
)
{
// Local variables
      int i, j;               // Counters

   Nodes::Print(Out);
   Out << "Num          X            Y            DX           DY \n";
   Out << "_____________________________________________________\n";
   for (i = 0; i < Num_Nodes; ++ i) {
      (Out < i + 1) << "   ";
      (Out < X[i]) << "   ";
      (Out < Y[i]) << "   ";
      (Out < DX[i]) << "   ";
      (Out < DY[i]) << "\n";
   }
   Out << "\n\n";

   Out << "Nodes Matrix\n";
   Out << "____________\n";
   for (i = 0; i < Num_Nodes; ++i) {
      for (j = 0; j < Num_Nodes; ++ j) {
         Out << Nodes0[i][j];
      }
      Out << "\n";
   }
   Out << "\n\n";
   return;
}

Nodes2::Nodes2
//---------------------------------------------------------------------
//
//   This constructor creates the base class structures (Nodes)
//   from the keyboard, then allocate space for the remaining
//   structure. This routine sets up the additional structures.
//
```

```
//---------------------------------------------------------------------
{
// Input parameters
      istream &In.            // Input stream
      ostream &Out            // Output stream
}
  Nodes(In. Out)
{
   Set_Up().
}

Nodes2::Nodes2
//---------------------------------------------------------------------
//
//   This routine creates the base class structures (Nodes)
//   from a file. then allocate space for the remaining structure
//   This routine sets up the additional structures.
//
//---------------------------------------------------------------------
{
// Input parameter
      char *Infile            // Input file
}
  Nodes(Infile)
{
   Set_Up().
}

Nodes2::~Nodes2()
//---------------------------------------------------------------------
//
//   This destructor frees space that was dynamically allocated
//   by the Create routines.
//
//---------------------------------------------------------------------
{
// Local variables
      int i.                  // Counter

   for (i = Num_Nodes - 1. i >= 0. --i)
      delete Nodes0[i].
   delete Nodes0.
   delete DY.
   delete DX.
   delete Y.
   delete X.
}

ostream &operator <<(ostream &A. Nodes2 &B){ B.Print(A). return A. }
```

LOOPS2.H

```
//---------------------------------------------------------------------
//
//   This is the file loops2 h
//
//---------------------------------------------------------------------

// Interface dependencies ------------------------------------------

#ifndef __NODES_H
#include "nodes.h"
#endif

// End interface dependencies --------------------------------------

#ifndef __LOOPS2_H
#define __LOOPS2_H

class Solution {
//---------------------------------------------------------------------
//
//   This class forms a base class for derived classes of solutions
//   to the layout problem.
//
//   operators
//      Solution()
```

```
//
//   interface routines
//      double Get_Cost()
//      Put_Cost(double)
//      Cost_State Get_E_State()
//      Put_E_State(Cost_State)
//
//-------------------------------------------------------------------
public:

  enum Cost_State {           // State of cost variable

     Evaluated,               // Cost is up to date
     Not_Evaluated            // Cost is not up to date
  };

protected:
   double Cost;               // Cost (length) of solution
   Cost_State E_State;        // Is cost variable up to date?

public:
   Solution(){}               // Vanilla constructor
   virtual ~Solution(){}      // Destructor


// Interface functions
   double Get_Cost() const { return Cost; }
                              // Returns value of cost variable
   void Put_Cost(double C){ Cost = C; return; }
                              // Sets value of cost variable
   Cost_State Get_E_State() const { return E_State; }
                              // Returns state of cost variable
   void Put_E_State(Cost_State E){ E_State = E; return; }
                              // Sets state of cost variable
};

class Cell_Stack {
//-------------------------------------------------------------------

//   Objects of this class are used to store pairs of integers
//   representing row and column locations in the Hanan grid.  The
//   stack is a LIFO (last in first out) structure.
//
//      operators
//         Cell_Stack(int)
//
//      functions
//         Empty()
//         int EmptyP()
//         Push(int, int)
//         Pop(int, int)
//
//-------------------------------------------------------------------
public:

   struct Position {          // Position in the Hanan grid

      int Row;                // Row position
      int Col                 // Column position
   };

protected:
   int Stack_Size;            // Maximum size of the stack
   Position *Stack;           // The stack array
   int Top_Stack;             // Index to the top of the stack

public:
   Cell_Stack(){}             // Vanilla constructor
   Cell_Stack(int);           // Constructor
   virtual ~Cell_Stack();     // Destructor
   void Empty();              // Empties the stack
   int EmptyP();              // Determines if the stack is empty
   void Push(int, int);       // Pushes the location on the stack
   void Pop(int&, int&);      // Pops the top location from the stack
};

class Loop_Base {
//-------------------------------------------------------------------
//
//   This is an abstract base class that forms the basis for the
```

```
//   display tester class of objects  The data members and interface
//   functions are declared.
//
//   operators
//      Loop_Base(Nodes2. int. int. int)
//      Loop_Base = Nodes2
//
//   functions
//      Init()
//
//   interface functions
//      Get_Prob()
//      Put_Prob(Nodes2*)
//      Solution* Get_Sol()
//      Put_Sol(Solution*)
//      int Get_Num()
//      int Get_Node(int. int)
//      Put_Node(int. int. int)
//      int Get_East(int. int)
//      int Get_South(int. int)
//      Clear_East(int. int)
//      Clear_South(int. int)
//      Set_East(int. int)
//      Set_South(int. int)
//      Set_Position(int. int)
//      int Get_Mode()
//      Set_Mode(int)
//      double Get_Cost()
//      Put_Cost(double)
//      Test_State Get_T_State()
//      Put_T_State(Test_State)
//      Binary_State Get_B_State()
//      Put_B_State(Binary_State)
//      Evaluation Get_E_State()
//      Put_E_State(Evaluation)
//
//   pure virtual functions
//      int Test()
//      Improve()
//      Evaluate()
//      Display()
//      Draw_1_Node()
//      Draw_1_East()
//      Draw_1_South()
//      Draw_1_Cell()
//      Toggle_Draw_East()
//      Toggle_Draw_South()
//      Mod_Show()
//      Mod_Hide()
//      Mod_Up()
//      Mod_Left()
//      Mod_Down()
//      Mod_Right()
//      int Toggle_Link()
//      int Toggle_Link(int. int. Net_Elem)
//      int Get_Mod_Row()
//      int Get_Mod_Col()
//      int Get_Mod_Elem()
//
//-------------------------------------------------------------------------
public

   enum Binary_State {          // Representation states

      Binary.                   // The solution is in binary format
      Not_Binary                // The solution is not in binary format
   };

   enum Test_State {            // Feasibility states

      Clear.                    // Solution structures have been cleared
      Unknown.                  // Feasibility is unknown
      Not_Feasible.             // The solution is not feasible
      Feasible.                 // The solution is feasible
      Improved                  // The solution is feasible and has been
                                // improved
   };

   enum Evaluation {            // Evaluation states
```

```
      Not_Evaluated,          // The solution has not been evaluated
      Evaluated               // The solution has been evaluated
   };

   struct Cursor_Rec {        // Data required to draw element at
                              // the current cursor position

      int X;                  // X position in pixels
      int Y;                  // Y position in pixels
      int XP;                 // Left of cell in pixels
      int YP;                 // Top of cell in pixels
      int XM;                 // Right of cell in pixels
      int YM;                 // Bottom of cell in pixels
   };

   enum Net_Elem {            // Element on which modification cursor
                              // rests
      Node_El,                // Node
      East_El,                // East link
      South_El,               // South link
      Cell_El                 // Cell
   };

protected:
   const Nodes2 *Prob_Source,
                              // Pointer to problem data (nodes)
   Solution *Sol_Source;      // Pointer to solution data (links)
   int Num_Nodes;             // Number of basic vertices
   int **Node_Copy;           // Nodes matrix
   int **East_Copy;           // East links matrix
   int **South_Copy;          // South links matrix
   int **Cell;                // Cells matrix
   double Cost;               // Cost of solution (length)
   int Row;                   // Row position of cursor
   int Col;                   // Column position of cursor
   int Key_Pause;             // Flag (1 = display will pause until key
                              // is pressed)
   int Display_Mode;          // Display mode (0 - 5)
   int Block_Imp;             // Flag (1 = improvement step will not be
                              // displayed)
   Binary_State B_State;      // Representation state
   Evaluation E_State;        // Evaluation state
   Test_State T_State;        // Feasibility state

public:
   Loop_Base(){}              // Vanilla constructor
   Loop_Base(Nodes2&, int = 0, int = 1, int = 0);
                              // Constructor
   virtual ~Loop_Base();      // Destructor
//   Loop_Base &operator =(const Nodes2&);
                              // Copies nodes data to this object
   void Init();               // Initializes values of matrices

// Interface functions
   const Nodes2 *Get_Prob() const { return Prob_Source; }
                              // Returns the pointer to the problem
   void Put_Prob(Nodes2 *P){ Prob_Source = P; return; }
                              // Sets the pointer to the problem
   Solution *Get_Sol() const { return Sol_Source; }
                              // Returns the pointer to the solution
   void Put_Sol(Solution *S){ Sol_Source = S; return; }
                              // Sets the pointer to the solution
   int Get_Num() const { return Num_Nodes; }
                              // Returns the number of nodes
   int Get_Node(int, int) const;
                              // Returns the value of the node at the
                              // specified location
   void Put_Node(int, int, int);
                              // Sets the value of the node at the
                              // specified location
   int Get_East(int, int) const;
                              // Returns the value of the east links
                              // at the specified location
   int Get_South(int, int) const;
                              // Returns the value of the south links
                              // at the specified location
   void Clear_East(int, int);
                              // Sets the value of the east links
                              // to 0 at the specified location
   void Clear_South(int, int);
```

```
                              // Sets the value of the south links
                              // to 0 at the specified location
   void Set_East(int, int);
                              // Sets the value of the east links
                              // to 1 at the specified location
   void Set_South(int, int);
                              // Sets the value of the south links
                              // to 1 at the specified location
   void Set_Position(int i, int j){ Row = i; Col = j; return; }
                              // Sets the location value of the cursor
                              // to the specified values
   int Get_Mode() const { return Display_Mode; }
                              // Returns the display mode
   void Set_Mode(int i){ Display_Mode = i; return; }
                              // Sets the display mode
   double Get_Cost() const { return Cost; }
                              // Returns the cost of the solutions
   void Put_Cost(double C){ Cost = C; return; }
                              // Sets the cost of the solutions
   Test_State Get_T_State() const { return T_State; }
                              // Returns feasibility of solution
   void Put_T_State(Test_State T){ T_State = T; return; }
                              // Sets feasibility of solution
   Binary_State Get_B_State() const { return B_State; }
                              // Returns representation state
   void Put_B_State(Binary_State B){ B_State = B; return; }
                              // Sets representation state
   Evaluation Get_E_State() const { return E_State; }
                              // Returns evaluation state
   void Put_E_State(Evaluation E){ E_State = E; return; }
                              // Sets evaluation state

// Pure virtual functions that must be overridden in derived classes
   virtual int Test() = 0;                  // Tests feasibility of solution
   virtual void Improve() = 0;              // Removes branched components
   virtual double Evaluate() = 0;           // Evaluates total length of
                                            // solution
   virtual void Display() = 0;              // Draws entire Hanan grid
   virtual void Draw_1_Node() = 0;          // Draws one node
   virtual void Draw_1_East() = 0;          // Draws one east link
   virtual void Draw_1_South() = 0;         // Draws one south link
   virtual void Draw_1_Cell() = 0;          // Draws one cell
   virtual void Toggle_Draw_East() = 0;     // Draws a toggled east link
   virtual void Toggle_Draw_South() = 0;    // Draws a toggled south link
   virtual void Mod_Show() = 0;     // Displays the modification cursor
   virtual void Mod_Hide() = 0;     // Hides the modification cursor
   virtual void Mod_Up() = 0;       // Moves the modification cursor up
   virtual void Mod_Left() = 0;     // Moves the modification cursor up
   virtual void Mod_Down() = 0;     // Moves the modification cursor up
   virtual void Mod_Right() = 0;    // Moves the modification cursor up
   virtual int Toggle_Link() = 0;   // Toggles the link at the modification
                                    // cursor position
   virtual int Toggle_Link(int, int, Net_Elem) = 0;
                              // Toggles the link at the specified
                              // position
   virtual int Get_Mod_Row() const = 0;
                              // Returns modification cursor row
   virtual int Get_Mod_Col() const = 0;
                              // Returns modification cursor column
   virtual int Get_Mod_Elem() const = 0;
                              // Returns 1 if modification cursor
                              // element is a south link
};

class Loop_Cell : public Loop_Base {
//------------------------------------------------------------------------
//
//   This is an abstract base class that performs the polygon fill
//   routine which is the first step in testing the feasibility of
//   layout geometry.
//
//   operators
//      Loop_Cell(Nodes2, int, int, int)
//
//   functions
//      same as Loop_Base
//
//   interface functions
//      same as Loop_Base
//
```

```
//   pure virtual functions
//      same as Loop_Base
//
//-------------------------------------------------------------------------
public:

   enum Jump_Type {          // Controls diagonal jumps in fill routine

      Jump,                  // Allows diagonal jumps in fill routine
      No_Jump                // No diagonal jumps in fill routine
   };

protected:

// Implementation functions
   void Find_Loops();        // Marks all cells in the Hanan grid with
                             // a polygon identifier
   void Loop_Finder(int, Jump_Type = No_Jump);
                             // Mark cells of an unmarked polygon
   int Find_Zero();          // Finds the starting row and column of
                             // an unmarked polygon
   void Row_Set(Cell_Stack&, int, Jump_Type);
                             // Marks all the cells in a row
   void Left_Up(Cell_Stack&);
                             // One of four diagonal jump routines
   void Left_Down(Cell_Stack&);
                             // One of four diagonal jump routines
   void Right_Up(Cell_Stack&);
                             // One of four diagonal jump routines
   void Right_Down(Cell_Stack&);
                             // One of four diagonal jump routines
   void Look_Up(Cell_Stack&);
                             // Closes the row above the cursor position
   void Look_Down(Cell_Stack&);
                             // Closes the row below the cursor position
   void Adjust_Left();       // Moves cursor to west end of row
   void Close_Row();         // Closes a row by marking with -1
   void Move_Left();         // Moves cursor left
   void Move_Right();        // Moves cursor right
   void Move_Up();           // Moves cursor up
   void Move_Down();         // Moves cursor down

public:
   Loop_Cell(){}             // Vanilla constructor
   Loop_Cell(Nodes2 &A, int B = 0, int C = 1, int D = 0)
        Loop_Base(A, B, C, D) {}
                             // Actual constructor
   virtual ~Loop_Cell(){}    // Destructor

// Pure virtual functions that must be overridden in derived classes
   virtual int Test() = 0;               // Tests feasibility of solution
   virtual void Improve() = 0;           // Removes branched components
   virtual double Evaluate() = 0;        // Evaluates total length of
                                         // solution
   virtual void Display() = 0;           // Draws entire Hanan grid
   virtual void Draw_1_Node() = 0;       // Draws one node
   virtual void Draw_1_East() = 0;       // Draws one east link
   virtual void Draw_1_South() = 0;      // Draws one south link
   virtual void Draw_1_Cell() = 0;       // Draws one cell
   virtual void Toggle_Draw_East() = 0;  // Draws a toggled east link
   virtual void Toggle_Draw_South() = 0; // Draws a toggled south link
   virtual void Mod_Show() = 0;    // Displays the modification cursor
   virtual void Mod_Hide() = 0;    // Hides the modification cursor
   virtual void Mod_Up() = 0;      // Moves the modification cursor up
   virtual void Mod_Left() = 0;    // Moves the modification cursor up
   virtual void Mod_Down() = 0;    // Moves the modification cursor up
   virtual void Mod_Right() = 0;   // Moves the modification cursor up
   virtual int Toggle_Link() = 0; // Toggles the link at the modification
                                  // cursor position
   virtual int Toggle_Link(int, int, Net_Elem) = 0;
                             // Toggles the link at the specified
                             // position
   virtual int Get_Mod_Row() const = 0;
                             // Returns modification cursor row
   virtual int Get_Mod_Col() const = 0;
                             // Returns modification cursor column
   virtual int Get_Mod_Elem() const = 0;
                             // Returns 1 if modification cursor
                             // element is a south link
};
```

```
class Loop_Edge : public Loop_Cell {
//-------------------------------------------------------------------------
//
//   This is an abstract base class that performs the edge polygon
//   marking routine which is the second step in testing the
//   feasibility of layout geometry.
//
//   operators
//      Loop_Edge(Nodes2, int, int, int)
//
//   functions
//      same as Loop_Cell
//
//   interface functions
//      same as Loop_Cell
//
//   pure virtual functions
//      same as Loop_Cell
//
//-------------------------------------------------------------------------
public:

   enum Bump {                  // Direction of cursor

      Up,                       // Cursor points north
      Down,                     // Cursor points south
      Right,                    // Cursor points west
      Left                      // Cursor points east
   };

protected:

// Implementation functions
   void Find_Edge();            // Marks the edges of the polygons
   int Find_Start(int);         // Finds starting location of polygon
                                // to be marked
   Bump Bump_Up(int);           // Moves cursor north to find edge
   Bump Bump_Right(int);        // Moves cursor west to find edge
   Bump Bump_Down(int);         // Moves cursor south to find edge
   Bump Bump_Left(int);         // Moves cursor east to find edge
   void Mark_Node();            // Marks a node in a natural loop
   void Mark_East();            // Marks an east link in a natural loop
   void Mark_South();           // Marks a sount link in a natural loop

public:
   Loop_Edge(){}                // Vanilla constructor
   Loop_Edge(Nodes2 &A, int B = 0, int C = 1, int D = 0)
        Loop_Cell(A, B, C, D) {}
                                // Actual constructor
   virtual ~Loop_Edge(){}       // Destructor

// Pure virtual functions that must be overridden in derived classes
   virtual int Test() = 0;              // Tests feasibility of solution
   virtual void Improve() = 0;          // Removes branched components
   virtual double Evaluate() = 0;       // Evaluates total length of
                                        // solution
   virtual void Display() = 0;          // Draws entire Hanan grid
   virtual void Draw_1_Node() = 0;      // Draws one node
   virtual void Draw_1_East() = 0;      // Draws one east link
   virtual void Draw_1_South() = 0;     // Draws one south link
   virtual void Draw_1_Cell() = 0;      // Draws one cell
   virtual void Toggle_Draw_East() = 0; // Draws a toggled east link
   virtual void Toggle_Draw_South() = 0; // Draws a toggled south link
   virtual void Mod_Show() = 0;    // Displays the modification cursor
   virtual void Mod_Hide() = 0;    // Hides the modification cursor
   virtual void Mod_Up() = 0;      // Moves the modification cursor up
   virtual void Mod_Left() = 0;    // Moves the modification cursor up
   virtual void Mod_Down() = 0;    // Moves the modification cursor up
   virtual void Mod_Right() = 0;   // Moves the modification cursor up
   virtual int Toggle_Link() = 0; // Toggles the link at the modification
                                  // cursor position
   virtual int Toggle_Link(int, int, Net_Elem) = 0;
                                // Toggles the link at the specified
                                // position
   virtual int Get_Mod_Row() const = 0;
                                // Returns modification cursor row
   virtual int Get_Mod_Col() const = 0;
                                // Returns modification cursor column
   virtual int Get_Mod_Elem() const = 0;
```

```
                              // Returns 1 if modification cursor
                              // element is a south link
};

class Loop_Test : public Loop_Edge {
//-------------------------------------------------------------------------
//
//   This is an abstract base class that performs the high level
//   feasibility testing routine as well as improvement and
//   evaluation routines
//
//   operators
//      Loop_Test(Nodes2, int, int, int)
//
//   functions
//      int Test()
//      void Improve()
//      double Evaluate()
//
//   interface functions
//      same as Loop_Edge
//
//   pure virtual functions
//      same as Loop_Edge
//
//-------------------------------------------------------------------------
public:
   Loop_Test(){}              // Vanilla constructor
   Loop_Test(Nodes2 &A, int B = 0, int C = 1, int D = 0)
        Loop_Edge(A, B, C, D) {}
                              // Actual constructor
   virtual ~Loop_Test(){}     // Destructor
   virtual int Test();        // Tests feasibility of solution
   virtual void Improve();    // Removes branched components
   virtual double Evaluate();
                              // Evaluates total length of solution

// Pure virtual functions that must be overridden in derived classes
   virtual void Display() = 0;           // Draws entire Hanan grid
   virtual void Draw_1_Node() = 0;       // Draws one node
   virtual void Draw_1_East() = 0;       // Draws one east link
   virtual void Draw_1_South() = 0;      // Draws one south link
   virtual void Draw_1_Cell() = 0;       // Draws one cell
   virtual void Toggle_Draw_East() = 0;  // Draws a toggled east link
   virtual void Toggle_Draw_South() = 0; // Draws a toggled south link
   virtual void Mod_Show() = 0;    // Displays the modification cursor
   virtual void Mod_Hide() = 0;    // Hides the modification cursor
   virtual void Mod_Up() = 0;      // Moves the modification cursor up
   virtual void Mod_Left() = 0;    // Moves the modification cursor up
   virtual void Mod_Down() = 0;    // Moves the modification cursor up
   virtual void Mod_Right() = 0;   // Moves the modification cursor up
   virtual int Toggle_Link() = 0; // Toggles the link at the modification
                                  // cursor position
   virtual int Toggle_Link(int, int, Net_Elem) = 0;
                              // Toggles the link at the specified
                              // position
   virtual int Get_Mod_Row() const = 0;
                              // Returns modification cursor row
   virtual int Get_Mod_Col() const = 0;
                              // Returns modification cursor column
   virtual int Get_Mod_Elem() const = 0;
                              // Returns 1 if modification cursor
                              // element is a south link
};

#endif
```

LOOPS2.CPP

```
//-------------------------------------------------------------------------
//
//   This is the file loops2.cpp
//
//-------------------------------------------------------------------------

// Implementation dependencies ---------------------------------------
```

```
#ifndef _STDLIB_H_INCLUDED
#include <stdlib.h>
#endif

#ifndef __CUSTIO_H
#include "custio.h"
#endif

#ifndef __LOOPS2_H
#include "loops2.h"
#endif

// End implementation dependencies ------------------------------------

Cell_Stack::Cell_Stack
//--------------------------------------------------------------------
//
//   This constructor creates a stack capable of storing every node
//   in the Hanan grid.  The size allocated for the stack is the
//   square of the number of basic vertices.
//
//--------------------------------------------------------------------
(
// Input parameters
     int Num_Nodes          // Number of basic vertices
)
{
   Stack_Size = Num_Nodes * Num_Nodes;
   Stack = new Position[Stack_Size];
}

Cell_Stack::~Cell_Stack()
//--------------------------------------------------------------------
//
//   This destructor frees the space allocated for the stack.
//
//--------------------------------------------------------------------
{
   delete Stack;
}

void Cell_Stack::Empty()
//--------------------------------------------------------------------
//
//   This function empties the stack.
//
//--------------------------------------------------------------------
{
   Top_Stack = 0;
   return;
}

int Cell_Stack::EmptyP()
//--------------------------------------------------------------------
//
//   This is a predicate that returns a value of 1 if the stack is
//   empty and a value of 0 if there are elements on the stack.
//
//--------------------------------------------------------------------
{
   return (Top_Stack > 0) ? 0   1.
}

void Cell_Stack::Push
//--------------------------------------------------------------------
//
//   This function places the values of a row and column on the top
//   of the stack.
//
//--------------------------------------------------------------------
(
// Input parameters
     int i.                 // Row of position
     int j                  // Column of position
)
{
   Stack[Top_Stack].Row = i;
   Stack[Top_Stack].Col = j;
   ++Top_Stack;
   if (Top_Stack == Stack_Size) {
```

```
      cerr << "\n<Cell_Stack 01> Error: Cell stack overflow\n";
      exit(1);
   }
   return;
}

void Cell_Stack::Pop
//---------------------------------------------------------------------
//
//   This function move the values of a row and column from the top
//   of the stack to the variables referenced by "&i" and "&j"
//   respectively.
//
//---------------------------------------------------------------------
(
// Output parameters
      int &i                  // Row of position
      int &j                  // Column of position

)
{
   --Top_Stack;
   i = Stack[Top_Stack].Row;
   j = Stack[Top_Stack].Col;
   return;
}

Loop_Base::Loop_Base
//---------------------------------------------------------------------
//
//   This constructor allocates space for the nodes, east links, south
//   links and cells matrices.
//
//---------------------------------------------------------------------
(
// Input parameters
      Nodes2 &Problem,        // Source of problem data (nodes)
      int New_D_Mode,         // Display mode
      int New_Block,          // Flag (1 = do not display improvement
                              // step)
      int New_Key             // Flag (1 = pause after display until
                              // key is pressed)
)
{
// Local variables
      int i;                  // Counter

   Prob_Source = &Problem;
   Num_Nodes = Problem.Get_Num();

   // Allocate space
   Node_Copy = new int*[Num_Nodes];
   East_Copy = new int*[Num_Nodes];
   South_Copy = new int*[Num_Nodes];
   Cell = new int*[Num_Nodes];
   for (i = 0; i < Num_Nodes; ++i) {
      Node_Copy[i] = new int[Num_Nodes];
      East_Copy[i] = new int[Num_Nodes];
      South_Copy[i] = new int[Num_Nodes];
      Cell[i] = new int[Num_Nodes];
   }

   // Set display modes
   Display_Mode = New_D_Mode;
   Block_Imp = New_Block;
   Key_Pause = New_Key;

   // Set state variables
   T_State = Unknown;
   E_State = Not_Evaluated;
   B_State = Not_Binary;
}

Loop_Base::~Loop_Base()
//---------------------------------------------------------------------
//
//   This destructor frees space allocated for the tester/display
//   object.
//
//---------------------------------------------------------------------
{
```

```
// Local variables
      int i.                        // Counter

   for (i = Num_Nodes - 1; i >= 0; --i) {
      delete Cell[i].
      delete South_Copy[i].
      delete East_Copy[i].
      delete Node_Copy[i].
   }
   delete Cell.
   delete South_Copy.
   delete East_Copy.
   delete Node_Copy.
}

/*
Loop_Base &Loop_Base  operator =
//---------------------------------------------------------------------
//
//   This operator copies data from a Nodes2 object to the nodes
//   matrix.
//
//---------------------------------------------------------------------
(
// Input parameters
      const Nodes2 &Problem  // Nodes data source
)
{
// Local variables
      int i.                        // Row index
      int j.                        // Column index

   for  (i = 0; i < Num_Nodes; ++i)
      for (j = 0; j < Num_Nodes; ++j)
         Node_Copy[i][j] = Problem.Get_Node(i, j);
   Prob_Source = &Problem;
   return *this;
}
*/

void Loop_Base::Init()
//---------------------------------------------------------------------
//
//   This function resets the nodes matrix to data from the problem
//   source and sets the east and south cell matrices to zero.
//
//---------------------------------------------------------------------
{
// Local variables
      int i.                        // Counter
      int j.                        // Counter

   for  (i = 0; i < Num_Nodes; ++i)
      for (j = 0; j < Num_Nodes; ++j)
         Node_Copy[i][j] = Prob_Source->Get_Node(i, j);
   for  (i = 0; i < Num_Nodes; ++i)
      for (j = 0; j < Num_Nodes; ++j) {
         East_Copy[i][j] = 0;
         South_Copy[i][j] = 0;
         Cell[i][j] = 0;
      }

   // Set state variables
   B_State = Binary;
   T_State = Clear;
   Cost = 0;
   E_State = Evaluated;
   return;
}

/*
int Loop_Base::Toggle_Link
//---------------------------------------------------------------------
//
//   This function toggles the link at the specified position. If
//   the function removes a link the function returns a value of
//   -1. If the function adds a link it returns a value of 1. If
//   the specified value is not a link a value of 0 is returned.
//
//---------------------------------------------------------------------
```

```
{
// Input parameters
      const Nodes2 &Problem. // Nodes data source
      int i.                 // Row position
      int j.                 // Column position
      Loop_Base::Net_Elem El // Element type
}
{
// Local variables
      int LinkP.             // Return value

   // Set cursor to specified values
   Row = i;
   Col = j;

   // Initialize return value
   LinkP = 0;

   // If element is either east or south link toggle the link
   switch (El) {

      // If element is a east link toggle the east link
      case East_El:

         // If link is present remove it
         if (East_Copy[Row][Col]) {
            East_Copy[Row][Col] = 0.
            if (E_State == Evaluated)
               Cost -= Problem.Get_DX(Col);
            LinkP = -1;

         // Otherwise add the link
         } else {
            East_Copy[Row][Col] = 1;
            if (E_State == Evaluated)
               Cost += Problem.Get_DX(Col).
            LinkP = 1;
         }
         if (Display_Mode > 0)
            Toggle_Draw_East();
         break;

      // If element is a south link toggle the south link
      case South_El:

         // If the link is present remove it
         if (South_Copy[Row][Col]) {
            South_Copy[Row][Col] = 0;
            if (E_State == Evaluated)
               Cost -= Problem.Get_DY(Row).
            LinkP = -1;

         // Otherwise add the link
         } else {
            South_Copy[Row][Col] = 1;
            if (E_State == Evaluated)
               Cost += Problem.Get_DY(Row).
            LinkP = 1;
         }
         if (Display_Mode > 0)
            Toggle_Draw_South();
   }
   T_State = Unknown;
   return LinkP;
}

int Loop_Base::Toggle_Link
//---------------------------------------------------------------------
//
//   This function toggles the link at the current position of
//   the modification cursor. If the function removes a link the
//   function returns a value of -1. If the function adds a link
//   it returns a value of 1. If the specified value is not a link
//   a value of 0 is returned.
//
//---------------------------------------------------------------------
{
// Input parameters
      const Nodes2 &Problem  // Nodes data source
}
```

```
{
   return Toggle_Link(Problem, Mod_Row, Mod_Col, Mod_Elem);
}
*/

int Loop_Base::Get_Node
//---------------------------------------------------------------------
//
//   This function returns the value of the nodes matrix at the row
//   and column position if the location is in range.
//
//---------------------------------------------------------------------
(
// Input parameters
      int i,                    // Row position
      int j                     // Column position
)
const
{
   return
      (i >= 0 && i < Num_Nodes && j >= 0 && j < Num_Nodes) ?
         Node_Copy[i][j] : 0;
}

void Loop_Base::Put_Node
//---------------------------------------------------------------------
//
//   This function puts the new node value in the nodes matrix at the
//   specified location if it is in range.
//
//---------------------------------------------------------------------
(
// Input parameters
      int New_Node,             // New node value
      int i,                    // Row position
      int j                     // Column position
)
{
   if (i >= 0 && i < Num_Nodes && j >= 0 && j < Num_Nodes)
      Node_Copy[i][j] = New_Node;
   return;
}

int Loop_Base::Get_East
//---------------------------------------------------------------------
//
//   This function returns the value of the east links matrix at the
//   row and column position if the location is in range
//
//---------------------------------------------------------------------
(
// Input parameters
      int i,                    // Row position
      int j                     // Column position
)
const
{
   return
      (i >= 0 && i < Num_Nodes && j >= 0 && j < Num_Nodes) ?
         East_Copy[i][j] : 0;
}

int Loop_Base::Get_South
//---------------------------------------------------------------------
//
//   This function returns the value of the east links matrix at the
//   row and column position if the location is in range.
//
//---------------------------------------------------------------------
(
// Input parameters
      int i,                    // Row position
      int j                     // Column position
)
const
{
   return
      (i >= 0 && i < Num_Nodes && j >= 0 && j < Num_Nodes) ?
         South_Copy[i][j] : 0;
}
```

```
void Loop_Base::Clear_East
//---------------------------------------------------------------------
//
//   This function sets the value in the east links matrix at the
//   specified location to 0 if the location is in range.
//
//---------------------------------------------------------------------
(
// Input parameters
      int i.                    // Row position
      int j                     // Column position
)
{
   if (i >= 0 && i < Num_Nodes && j >= 0 && j < Num_Nodes)
      East_Copy[i][j] = 0.
   return;
}

void Loop_Base::Clear_South
//---------------------------------------------------------------------
//
//   This function sets the value in the south links matrix at the
//   specified location to 0 if the location is in range.
//
//---------------------------------------------------------------------
(
// Input parameters
      int i.                    // Row position
      int j                     // Column position
)
{
   if (i >= 0 && i < Num_Nodes && j >= 0 && j < Num_Nodes)
      South_Copy[i][j] = 0;
   return;
}

void Loop_Base::Set_East
//---------------------------------------------------------------------
//
//   This function sets the value in the east links matrix at the
//   specified location to 1 if the location is in range.
//
//---------------------------------------------------------------------
(
// Input parameters
      int i.                    // Row position
      int j                     // Column position
)
{
   if (i >= 0 && i < Num_Nodes && j >= 0 && j < Num_Nodes)
      East_Copy[i][j] = 1.
   return;
}

void Loop_Base::Set_South
//---------------------------------------------------------------------
//
//   This function sets the value in the south links matrix at the
//   specified location to 1 if the location is in range.
//
//---------------------------------------------------------------------
(
// Input parameters
      int i.                    // Row position
      int j                     // Column position
)
{
   if (i >= 0 && i < Num_Nodes && j >= 0 && j < Num_Nodes)
      South_Copy[i][j] = 1;
   return;
}

void Loop_Cell::Find_Loops()
//---------------------------------------------------------------------
//
//   This function is a top level routine that marks all the cells
//   in the Hanan grid with a polygon identifier value.
//
//---------------------------------------------------------------------
```

```
{
// Local variables
      int Loop_Num;              // Polygon identifier variable

   // Set state variables
   B_State = Not_Binary;
   T_State = Unknown;

   // Display
   if (Display_Mode >= 4)
      Display();

   // Find area outside polygons
   Loop_Num = 1;
   Row = 0;
   Col = Num_Nodes - 1;
   Loop_Finder(Loop_Num);

   // Find all remaining polygons
   while (Find_Zero())
      Loop_Finder(++Loop_Num);

   // Display
   if (Display_Mode == 4)
      Display();
   return;
}

void Loop_Cell::Loop_Finder
//-------------------------------------------------------------------------
//
//   This function marks the cells of an unmarked polygon with the
//   value specified by "Loop_Num".  The data members "Row" and "Col"
//   must be placed inside the polygon.
//
//-------------------------------------------------------------------------
(
// Inputparameters
      int Loop_Num,                        // Polygon identifier
      Jump_Type Diag                       // Flag (determines if diagonal
                                           // jumps are permitted)
)
{
// Local variables
      Cell_Stack Stack(Num_Nodes);         // Recursion stack

   // Initialize the stack
   Stack.Empty();

   // Close the first row
   Adjust_Left();
   Stack.Push(Row, Col);
   Close_Row();

   // Mark the first row
   Stack.Pop(Row, Col);
   Row_Set(Stack, Loop_Num, Diag);

   // Mark all remaining rows
   while (!Stack.EmptyP()) {
      Stack.Pop(Row, Col);
      Row_Set(Stack, Loop_Num, Diag);
   }
   return;
}

int Loop_Cell::Find_Zero()
//-------------------------------------------------------------------------
//
//   This function sets the "Row" and "Col" data members to the
//   starting row and column of an unmarked polygon.  If no more
//   unmarked cells are found the function returns 0 otherwise it
//   returns 1.
//
//-------------------------------------------------------------------------
{
// Local variables
      int Found;                 // Flag (0 = no more unmarked cells)
      int i;                     // Counter
      int j;                     // Counter
```

```
   Found = 0;
   for (i = 0; i < Num_Nodes && !Found; ++i)
      for (j = 0; j < Num_Nodes && !Found; ++j)
         if (!Cell[i][j]) {
            Found = 1;
            Row = i;
            Col = j;
         }
   return Found;
}

void Loop_Cell::Row_Set
//---------------------------------------------------------------------
//
//   This function marks all the cell in a row while checking the
//   cell above and cell below to see if the row above and the row
//   below should be closed (included in the polygon). At the extreme
//   left and right a diagonal check will occur if permitted
//
//---------------------------------------------------------------------
(
// Input parameters
      Cell_Stack &Stack;      // Reference to the stack
      int Loop_Num;           // Polygon identifier
      Jump_Type Diag          // Diagonal jump mode
)
{
// Local variables
      int Row_Copy;           // Records row position
      int Col_Copy;           // Records column position
      int Pipe;               // Flag (1 = pipe has been encountered)
      int End_Row;            // Flag (1 = end of row encountered)

   // Perform left diagonal jumps if permitted
   if (Diag == Jump) {

      // Record position and check upper left
      Row_Copy = Row;
      Col_Copy = Col;
      Left_Up(Stack);

      // Return to recorded position and check lower left
      Row = Row_Copy;
      Col = Col_Copy;
      Left_Down(Stack);

      // return to recorded position
      Row = Row_Copy;
      Col = Col_Copy;
   }

   // Assume end of row has not been encountered
   Pipe = End_Row = 0;

   // Mark all cells in the row
   while (!Pipe && !End_Row) {

      // Change the value of the cell and update the display
      Cell[Row][Col] = Loop_Num;
      if (Display_Mode >= 5)
         Draw_1_Cell();

      // Record position and check row above
      Row_Copy = Row;
      Col_Copy = Col;
      Look_Up(Stack);

      // Return to recorded position and check row below
      Row = Row_Copy;
      Col = Col_Copy;
      Look_Down(Stack);

      // Return to recorded position and move right
      Row = Row_Copy;
      Col = Col_Copy;
      Move_Right();

      // Stop when end of row is encountered
      if (South_Copy[Row][Col]) {
```

```
            Pipe = 1;
            Move_Left();
         } else
            if (Cell[Row][Col] != -1) {
               End_Row = 1;
               Move_Left();
            }
      }

      // Perform right diagonal jumps if permitted
      if (Diag == Jump) {

         // Record position and check upper right
         Row_Copy = Row;
         Col_Copy = Col;
         Right_Up(Stack);

         // Return to recorded position and check lower right
         Row = Row_Copy;
         Col = Col_Copy;
         Right_Down(Stack);

         // Return to recorded position
         Row = Row_Copy;
         Col = Col_Copy;
      }
      return;
}

void Loop_Cell::Left_Up
//---------------------------------------------------------------------------
//
//   This function performs a diagonal jump from the extreme west
//   (left) cell in the northwest (upper left) direction.  The
//   diagonal jump can only occur if a pattern of intersecting links
//   occurs and the upper left cell is unmarked.
//
//   The cursor is returned to the original position.
//
//---------------------------------------------------------------------------
(
// Input parameters
      Cell_Stack &Stack       // Reference to the stack
)
{
// Local variables
      int Row_Copy;           // Records row position
      int Col_Copy;           // Records column position

   // Record original position before moving
   Row_Copy = Row;
   Col_Copy = Col;

   // Check to see if required links are in place while moving to
   // upper left cell
   if (South_Copy[Row][Col])
      if (East_Copy[Row][Col]) {
         Move_Left();
         if (East_Copy[Row][Col]) {
            Move_Up();
            Move_Right();
            if (South_Copy[Row][Col]) {
               Move_Left();

               // If upper left cell is unmarked close the row
               if (!Cell[Row][Col]) {
                  Adjust_Left();
                  Stack.Push(Row, Col);
                  Close_Row();
               }
            }
         }

      }

   // Return to recorded position
   Row = Row_Copy;
   Col = Col_Copy;
   return;
}
```

```
void Loop_Cell::Left_Down
//-------------------------------------------------------------------
//
//   This function performs a diagonal jump from the extreme west
//   (left) cell in the southhwest (lower left) direction.  The
//   diagonal jump can only occur if a pattern of intersecting links
//   occurs and the lower left cell is unmarked.
//
//   The cursor is returned to the original position.
//
//-------------------------------------------------------------------

// Input parameters
     Cell_Stack &Stack      // Reference to the stack
)
{
// Local variables
     int Row_Copy;          // Records row position
     int Col_Copy;          // Records column position

   // Record original position before moving
   Row_Copy = Row;
   Col_Copy = Col;

   // Check to see if required links are in place while moving to
   // lower left cell
   if (South_Copy[Row][Col]) {
      Move_Down();
      if(East_Copy[Row][Col]) {
         if (South_Copy[Row][Col]) {
            Move_Left();
            if (East_Copy[Row][Col]) {

               // If lower left cell is unmarked close the row
               if (!Cell[Row][Col]) {
                  Adjust_Left();
                  Stack.Push(Row, Col);
                  Close_Row();
               }
            }
         }
      }
   }

   // Return to recorded position
   Row = Row_Copy;
   Col = Col_Copy;
   return;
}

void Loop_Cell::Right_Up
//-------------------------------------------------------------------
//
//   This function performs a diagonal jump from the extreme east
//   (right) cell in the northeast (upper right) direction.  The
//   diagonal jump can only occur if a pattern of intersecting links
//   occurs and the upper right cell is unmarked.
//
//   The cursor is returned to the original position.
//
//-------------------------------------------------------------------
(
// Input parameters
     Cell_Stack &Stack      // Reference to the stack
)
{
// Local variables
     int Row_Copy;          // Records row position
     int Col_Copy;          // Records column position

   // Record original position before moving
   Row_Copy = Row;
   Col_Copy = Col;

   // Check to see if required links are in place while moving to
   // upper right cell
   if (East_Copy[Row][Col]) {
      Move_Right();
      if (South_Copy[Row][Col]) {
```

```
         if (East_Copy[Row][Col]) {
            Move_Up();
            if (South_Copy[Row][Col]) {

               // If upper right cell is unmarked close the row
               if (!Cell[Row][Col]) {
                  Adjust_Left();
                  Stack.Push(Row, Col);
                  Close_Row();
               }
            }
         }
      }
   }

   // Return to recorded position
   Row = Row_Copy;
   Col = Col_Copy;
   return;
}

void Loop_Cell::Right_Down
//---------------------------------------------------------------------
//
//   This function performs a diagonal jump from the extreme east
//   (right) cell in the southeast (lower right) direction.  The
//   diagonal jump can only occur if a pattern of intersecting links
//   occurs and the lower right cell is unmarked.
//
//   The cursor is returned to the original position.
//
//---------------------------------------------------------------------
(
// Input parameters
      Cell_Stack &Stack      // Reference to the stack
)
{
// Local variables
      int Row_Copy;          // Records row position
      int Col_Copy;          // Records column position

   // Record original position before moving
   Row_Copy = Row;
   Col_Copy = Col;

   // Check to see if required links are in place while moving to
   // lower right cell
   Move_Down();
   if (East_Copy[Row][Col]) {
      Move_Up();
      Move_Right();
      if(South_Copy[Row][Col]) {
         Move_Down();
         if (East_Copy[Row][Col]) {
            if (South_Copy[Row][Col]) {

               // If lower right cell is unmarked close the row
               if (!Cell[Row][Col]) {
                  Adjust_Left();
                  Stack.Push(Row, Col);
                  Close_Row();
               }
            }
         }
      }
   }

   // Return to recorded position
   Row = Row_Copy;
   Col = Col_Copy;
   return;
}

void Loop_Cell::Look_Up
//---------------------------------------------------------------------
//
//   This function will close the row above the position specified
//   by the data memebers "Row" and "Col" if required.  The row will
//   be closed if the cell above is neither closed nor marked and
//   no link forms a barrier.
```

```
//
//   The cursor is returned to the original position.
//
//-------------------------------------------------------------------
{
// Input parameters
      Cell_Stack &Stack      // Reference to the stack
}
{
// Local variables
      int Row_Copy;          // Records row position
      int Col_Copy.          // Records column position

   // Record original position before moving
   Row_Copy = Row.
   Col_Copy = Col.

   // If no link is present and the cell is umarked close the row
   if (!East_Copy[Row][Col]) {
      Move_Up();
      if (!Cell[Row][Col]) {
         Adjust_Left();
         Stack Push(Row. Col)
         Close_Row();
      }
   }

   // Return to recorded position
   Row = Row_Copy.
   Col = Col_Copy
   return.
}

void Loop_Cell .Look_Down
//-------------------------------------------------------------------
//
//   This function will close the row below the position specified
//   by the data memebers "Row" and "Col" if required. The row will
//   be closed if the cell below is neither closed nor marked and
//   no link forms a barrier.
//
//   The cursor is returned to the original position
//
//-------------------------------------------------------------------
{
// Input parameters
      Cell_Stack &Stack      // Reference to the stack
}
{
// Local variables
      int Row_Copy.          // Records row position
      int Col_Copy.          // Records column position

   // Record original position before moving
   Row_Copy = Row.
   Col_Copy = Col.

   // If no link is present and the cell is umarked close the row
   Move_Down();
   if (!East_Copy[Row][Col])
      if (!Cell[Row][Col]) {
         Adjust_Left().
         Stack.Push(Row. Col).
         Close_Row().
      }

   // Return to recorded position
   Row = Row_Copy;
   Col = Col_Copy.
   return;
}

void Loop_Cell::Adjust_Left()
//-------------------------------------------------------------------
//
//   This function moves the cursor to the extreme west (left) of a
//   row marking cells with -2.
//
//   The cursor is not returned to the original position.
//
```

```
//---------------------------------------------------------------------------
{
// Local variables
      int Pipe;                  // Flag (1 = pipe encountered)
      int End_Row;               // Flag (1 = end of row encountered)

   // Until end of row is reached move left
   Pipe = End_Row = 0;
   while (!End_Row && !Pipe) {

      // Change cell value and update display
      Cell[Row][Col] = -2;
      if (Display_Mode >= 5)
         Draw_1_Cell();

      // Move left if possible, if not stop
      if (South_Copy[Row][Col])
         Pipe = 1;
      else {
         Move_Left();
         if (Cell[Row][Col]) {
            End_Row = 1;
            Move_Right();
         }
      }
   }
   return;
}

void Loop_Cell::Close_Row()
//---------------------------------------------------------------------------
//
//   This function marks all the cells in a row with -1.
//
//   The cursor is not returned to the original position.
//
//---------------------------------------------------------------------------
{
// Local variables
      int Pipe;                  // Flag (1 = pipe encountered)

   // Until end of row is reached move right
   Pipe = 0;
   while ((Cell[Row][Col] == -2 || !Cell[Row][Col]) && !Pipe) {

      // Change cell value and update display
      Cell[Row][Col] = -1;
      if (Display_Mode >= 5)
         Draw_1_Cell();

      // Move right and determine if end is reached
      Move_Right();
      if (South_Copy[Row][Col])
         Pipe = 1;
   }
   return;
}

void Loop_Cell::Move_Left()
//---------------------------------------------------------------------------
//
//   This function moves the cursor left (east) one position. The
//   function accommodates the 2-torus "wrap-around" properties of
//   the Hanan grid.
//
//---------------------------------------------------------------------------
{
   if (Col == 0)
      Col = Num_Nodes - 1;
   else
      --Col;
   return;
}

void Loop_Cell::Move_Right()
//---------------------------------------------------------------------------
//
//   This function moves the cursor right (west) one position. The
//   function accommodates the 2-torus "wrap-around" properties of
//   the Hanan grid.
```

```
//
//-------------------------------------------------------------------------
{
   ++Col;
   if (Col == Num_Nodes)
      Col = 0;
   return;
}

void Loop_Cell::Move_Up()
//-------------------------------------------------------------------------
//
//   This function moves the cursor up (north) one position  The
//   function accommodates the 2-torus "wrap-around" properties of
//   the Hanan grid.
//
//-------------------------------------------------------------------------
{
   if (Row == 0)
      Row = Num_Nodes - 1;
   else
      --Row;
   return;
}

void Loop_Cell::Move_Down()
//-------------------------------------------------------------------------
//
//   This function moves the cursor down (south) one position  The
//   function accommodates the 2-torus "wrap-around" properties of
//   the Hanan grid.
//
//-------------------------------------------------------------------------
{
   ++Row;
   if (Row == Num_Nodes)
      Row = 0;
   return;
}

void Loop_Edge::Find_Edge()
//-------------------------------------------------------------------------
//
//   This function is the high level function that marks the edges of
//   the polygons. The edge of each polygon with an identifier value
//   greater than two is marked.
//
//-------------------------------------------------------------------------
{
// Local variables
      int Loop_Num;           // Polygon identifer
      Bump Direct;            // Cursor direction
      int Start_Row;          // Starting position - row
      int Start_Column;       // Starting position - column
      int Edge_Stop;          // Flag (1 = edge marking is complete)

   // Set state variables - this test is destructive
   B_State = Not_Binary;
   T_State = Unknown;

   // For every polygon identfied with a value greater than 1
   // find the edge
   Loop_Num = 2;
   while (Find_Start(Loop_Num)) {

      // Record the starting position
      // Starting direction is always right
      Start_Row = Row;
      Start_Column = Col;
      Direct = Right;

      // Mark edges until the cursor returns to the starting position
      // and direction
      Edge_Stop = 0;
      while (!Edge_Stop) {
         switch (Direct) {
            case Up:
               Direct = Bump_Up(Loop_Num);
               break;
            case Right:
```

```
            Direct = Bump_Right(Loop_Num);
            break;
         case Down:
            Direct = Bump_Down(Loop_Num);
            break;
         case Left:
            Direct = Bump_Left(Loop_Num);
      }

      // Determine if the starting position and direction have been
      // reached
      if (Row == Start_Row && Col == Start_Column && Direct == Right)
         Edge_Stop = 1;
   }

   // Move to the next polygon
   ++Loop_Num;
}

// If display mode requires display the results
if ((Display_Mode >= 2) && (Display_Mode <= 4))
   Display();
return;
}

int Loop_Edge::Find_Start
//-------------------------------------------------------------------------
//
//   This function finds the starting location for the of a polygon
//   marked with the identifier specified by "Loop_Num".  The grid
//   is searched in row-column order for the first cell equal to
//   "Loop_Num".  The data members "Row" and "Col" are set equal to
//   the row and column of this cell.  A value of 1 is returned if a
//   starting location is found
//
//-------------------------------------------------------------------------
{
// Input parameters
   int Loop_Num            // Polygon identifer value
)
{
// Local variables
   int Found;              // Flag (1 = starting point found)

Found = Row = 0;
while (Row < Num_Nodes && !Found) {
   Col = 0;
   while (Col < Num_Nodes && !Found)
      if (Cell[Row][Col] == Loop_Num)
         Found = 1;
      else
         ++Col;
   if (!Found)
     ++Row;
}
return Found;
}

Loop_Edge::Bump Loop_Edge::Bump_Up
//-------------------------------------------------------------------------
//
//   This function moves the cursor up one position to determine if
//   the polygon identifier value has changed from that specified
//   by "Loop_Num".  If a change does not occur the cursor points
//   left. If a change does occur the node and link that define the
//   edge are marked. Then the cursor backs up and turns right.  In
//   either case the value of the new direction is returned.
//
//-------------------------------------------------------------------------
(
// Input parameters
   int Loop_Num            // Polygon identifier value
)
{
// Local variables
   Bump Direct;            // New direction of cursor

   // Move forward and determine if the edge of a polygon has
   // been crossed
   Move_Up();
```

```
   // If has not been crossed turn left
   if (Cell[Row][Col] == Loop_Num)
      Direct = Left;

   // Otherwise mark the edge, back up and turn right
   else {
      Move_Down();
      Mark_Node();
      Mark_East();
      Direct = Right;
   }
   return Direct;
}

Loop_Edge::Bump Loop_Edge::Bump_Right
//---------------------------------------------------------------------
//
//   This function moves the cursor right one position to determine if
//   the polygon identifier value has changed from that specified
//   by "Loop_Num".  If a change does not occur the cursor points
//   left.  If a change does occur the node and link that define the
//   edge are marked.  Then the cursor backs up and turns right.  In
//   either case the value of the new direction is returned.
//
//---------------------------------------------------------------------
(
// Input parameters
      int Loop_Num           // Polygon identifier value
)
{
// Local variables
      Bump Direct;           // New direction of cursor

   // Move forward and determine if the edge of a polygon has
   // been crossed
   Move_Right();

   // If edge has not been crossed turn left
   if (Cell[Row][Col] == Loop_Num)
      Direct = Up;

   // Otherwise mark the edge, back up and turn right
   else {
      Mark_Node();
      Mark_South();
      Move_Left();
      Direct = Down;
   }
   return Direct;
}

Loop_Edge::Bump Loop_Edge::Bump_Down
//---------------------------------------------------------------------
//
//   This function moves the cursor down one position to determine if
//   the polygon identifier value has changed from that specified
//   by "Loop_Num".  If a change does not occur the cursor points
//   left.  If a change does occur the node and link that define the
//   edge are marked.  Then the cursor backs up and turns right.  In
//   either case the value of the new direction is returned.
//
//---------------------------------------------------------------------
(
// Input parameters
      int Loop_Num           // Polygon identifier value
)
{
// Local variables
      Bump Direct;           // New direction of cursor

   // Move forward and determine if the edge of a polygon has
   // been crossed
   Move_Down();

   // If edge has not been crossed turn left
   if (Cell[Row][Col] == Loop_Num)
      Direct = Right;
   // Otherwise mark the edge, back up and turn right
   else {
```

```
      Move_Right();
      Mark_Node();
      Move_Left();
      Mark_East();
      Move_Up();
      Direct = Left;
   }
   return Direct;
}

Loop_Edge::Bump Loop_Edge::Bump_Left
//---------------------------------------------------------------------
//
//   This function moves the cursor left one position to determine if
//   the polygon identifier value has changed from that specified
//   by "Loop_Num".  If a change does not occur the cursor points
//   left.  If a change does occur the node and link that define the
//   edge are marked.  Then the cursor backs up and turns right.  In
//   either case the value of the new direction is returned.
//
//---------------------------------------------------------------------
(
// Input parameters
      int Loop_Num            // Polygon identifier value
)
{
// Local variables
      Bump Direct;            // New direction of cursor

   // Move forward and determine if the edge of a polygon has
   // been crossed
   Move_Left();

   // If edge has not been crossed turn left
   if (Cell[Row][Col] == Loop_Num)
      Direct = Down;

   // Otherwise mark the edge, back up and turn right
   else {
      Move_Right();
      Move_Down();
      Mark_Node();
      Move_Up();
      Mark_South();
      Direct = Up;
   }
   return Direct;
}

void Loop_Edge::Mark_Node()
//---------------------------------------------------------------------
//
//   This function marks a node at the location specified by the
//   data members "Row" and "Col".  If the node is a basic vertex
//   with a value of 1 it is changes to -2 otherwise it is changed
//   to -1.
//
//---------------------------------------------------------------------
{
   if (Node_Copy[Row][Col] != -2)
      if (Node_Copy[Row][Col] == 1)
         Node_Copy[Row][Col] = -2;
      else
         Node_Copy[Row][Col] = -1;

   // Display the change if the display mode requires
   if (Display_Mode >= 5)
      Draw_1_Node();
   return;
}

void Loop_Edge::Mark_East()
//---------------------------------------------------------------------
//
//   This function marks the east link at the location specified by
//   the data members "Row" and "Col".  The first time the link is
//   marked it is set to -1.  The second time it is set to -2.
//
//---------------------------------------------------------------------
{
```

```
   if (East_Copy[Row][Col] == -1 || East_Copy[Row][Col] == -2)
      East_Copy[Row][Col] = -2;
   else
      East_Copy[Row][Col] = -1;

   // Display the change if the display mode requires
   if (Display_Mode >= 5)
      Draw_1_East();
   return;
}

void Loop_Edge::Mark_South()
//---------------------------------------------------------------------
//
//   This function marks the south link at the location specified by
//   the data members "Row" and "Col".  The first time the link is
//   marked it is set to -1.  The second time it is set to -2.
//
//---------------------------------------------------------------------
{
   if (South_Copy[Row][Col] == -1 || South_Copy[Row][Col] == -2)
      South_Copy[Row][Col] = -2;
   else
      South_Copy[Row][Col] = -1;

   // Display the change if the display mode requires
   if (Display_Mode >= 5)
      Draw_1_South();
   return;
}

int Loop_Test::Test()
//---------------------------------------------------------------------
//
//   This function is a destructive test that determines if the
//   solution layout is feasible.  The function returns 1 if the
//   solution is feasible and 0 is not.
//
//---------------------------------------------------------------------
{
   // Local variables
      int FeasibleP;          // Flag (1 = solution is feasible)
      int i;                  // Counter
      int j;                  // Counter

   if (T_State != Feasible &&
       T_State != Not_Feasible &&
       T_State != Clear) {
      if (B_State == Binary) {
         FeasibleP = 1;
         Find_Loops();
         Find_Edge();
         // Search all entries in Node_Copy
         for (i = 0; i < Num_Nodes && FeasibleP; ++i)
            for (j = 0; j < Num_Nodes && FeasibleP; ++j)
               if (Node_Copy[i][j] == 1)
                  FeasibleP = 0;
         if (FeasibleP) {
            // Remove pipes except those marked once
            // and set all loops to zero
            for (i = 0; i < Num_Nodes; ++i)
               for (j = 0; j < Num_Nodes; ++j) {
                  if (East_Copy[i][j] != -1)
                     East_Copy[i][j] = 0;
                  if (South_Copy[i][j] != -1)
                     South_Copy[i][j] = 0;
                  if (Cell[i][j] > 1)
                     Cell[i][j] = 0;
               }
            if (Display_Mode >= 5)
               Display();
            // Mark loop as 2
            Find_Zero();
            Loop_Finder(2, Jump);
            if (Display_Mode >= 4)
               Display();
            // Check to see if any zeros remain
            for (i = 0; i < Num_Nodes && FeasibleP; ++i)
               for (j = 0; j < Num_Nodes && FeasibleP; ++j)
                  if (Cell[i][j] == 0)
```

```
                    FeasibleP = 0;
            }
            if (FeasibleP)
               T_State = Feasible;
            else
               T_State = Not_Feasible;
         } else {
            cerr << "\n<Loop_Test 01> Error: Attempt to test non-binary solution\n";
            exit(1);
         }
      } else {
         if (T_State == Feasible)
            FeasibleP = 1;
         else
            FeasibleP = 0;
      }
      return FeasibleP;
}

void Loop_Test::Improve()
//---------------------------------------------------------------------------
//
//   This function improves a feasible binary coded solution by
//   marking the natural loops in the layout.
//
//---------------------------------------------------------------------------
{
// Local variables
      int Temp_Mode;          // Temporary storage of display mode

   // Make sure conditions are right for the improvement step
   if (T_State != Improved) {
      if (T_State == Feasible) {
         if (B_State == Binary) {

            // If improvement step is not to be displayed
            // block display
            Temp_Mode = Display_Mode;
            if (Block_Imp)
               Display_Mode = 0;

            // Find the polygons and edges
            Find_Loops();
            Find_Edge();

            // Display results if required
            Display_Mode = Temp_Mode;
            if (Display_Mode == 1)
               Display();

            T_State = Improved;
         } else {
            cerr << "\n<Loop_Test 02> Error: Attempt to improve non-binary
solution\n";
            exit(1);
         }
      } else {
         cerr << "\n<Loop_Test 03> Error: Attempt to improve infeasible
solution\n";
         exit(1);
      }
   }
   return;
}

double Loop_Test::Evaluate()
//---------------------------------------------------------------------------
//
//   This function evaluates the cost (length) of a feasible
//   solution and transfers the result to the solution source
//   object.
//
//---------------------------------------------------------------------------
{
// Local variables
      double New_Cost;        // Temporary running total of cost
      int N1;                 // Temporary (1 minus number of nodes)
      int i;                  // Counter
      int j;                  // Counter
```

```
   // Make sure conditions are right for evaluation
   if (E_State != Evaluated)
      if (B_State == Binary) {

         // Initialize variables
         New_Cost = 0.0;
         N1 = Num_Nodes - 1;

         // Total up cost (length) of east and south links
         for (i = 0; i < Num_Nodes; ++i)
            for (j = 0; j < N1; ++j)
               if (East_Copy[i][j])
                  New_Cost += Prob_Source->Get_DX(j);
         for (i = 0; i < N1; ++i)
            for (j = 0; j < Num_Nodes; ++j)
               if (South_Copy[i][j])
                  New_Cost += Prob_Source->Get_DY(i);

         // Update cost and state variable
         Cost = New_Cost;
         E_State = Evaluated;

         // Copy the results to the source object
         // and change its state variable
         Sol_Source->Put_Cost(Cost);
         Sol_Source->Put_E_State(Solution::Evaluated);
      } else {
         cerr << "\n<Loop_Test 04> Error: Attempt to evaluate non-binary link
data\n";
         exit(1);
      }
   return New_Cost;
}
```

LOOPS3.H

```
//---------------------------------------------------------------------
//
//   This is the file loops3.h
//
//---------------------------------------------------------------------

// Interface dependencies ---------------------------------------------

#ifndef __LOOPS2_H
#include "loops2.h"
#endif

// End interface dependencies -----------------------------------------

#ifndef __LOOPS3_H
#define __LOOPS3_H

//---------------------------------------------------------------------
//
//   This enumerated type is required by the graphics functions
//
//---------------------------------------------------------------------
enum {
   BLACK = 0,
   BLUE = 1,
   GREEN = 2,
   CYAN = 3,
   RED = 4,
   LIGHTGRAY = 7,
   LIGHTGREEN = 10,
   LIGHTRED = 12,
   YELLOW = 14,
   WHITE = 15
};

class Graph_Measures {
//---------------------------------------------------------------------
//
//   Objects of this class are used to a modified set of dimensions
//   for the Hanan grid for display purposes only to display rows
//   and columns that would otherwise be to narrow to be seen
```

```
//   properly.
//
//   operators
//      Graph_Measures(Nodes2)
//
//   interface routines
//      double Get_TotalX()
//      double Get_TotalY()
//      double Get_DX(int)
//      double Get_DY(int)
//
//-------------------------------------------------------------------
protected:
   int Num_Nodes.              // Number of nodes
   double *DX;                 // Horizontal (east-west) spacing
   double *DY;                 // Vertical (north-south) spacing
   double TotalX;              // Total width (east-west) of grid
   double TotalY;              // Total length (north-south) of grid

// Implementation functions
   void Measure_Graph(const Nodes2&);
                               // Copies spacing data from source
   void Adjust_Size();         // Ensures no spacing is below the
                               // minimum

public:
   Graph_Measures(){}          // Vanilla constructor
   Graph_Measures(const Nodes2&);
                               // Constructor
                               // Allocates and initializes object
   virtual ~Graph_Measures();
                               // Destructor

// Interface functions
   double Get_TotalX(){ return TotalX; }
                               // Returns the width of the grid
   double Get_TotalY(){ return TotalY; }
                               // Returns the length of the grid
   double Get_DX(int i){ return DX[i]; }
                               // Returns the horizontal spacing of
                               // column i
   double Get_DY(int i){ return DY[i]; }
                               // Returns the vertical spacing of
                               // row i
};


class Loop_EGA   public Loop_Test {
//-------------------------------------------------------------------
//
//   Objects of this class are able to display the results and
//   operations of the base "Loop_Test" class.
//
//   operators
//      Loop_EGA(Nodes2&, int, int, int);
//
//   functions
//      Display()
//      Draw_1_Node()
//      Draw_1_East()
//      Draw_1_South()
//      Draw_1_Cell()
//      Toggle_Draw_East()
//      Toggle_Draw_South()
//
//-------------------------------------------------------------------
protected:
   int Page;                   // Current video page
   Cursor_Rec Cursor;          // Storage key coordinate locations at
                               // current cursor position
   int *StepX;                 // Horizontal spacing of Hanan grid in
                               // pixels
   int *StepY;                 // Vertical spacing of Hanan grid in
                               // pixels
   int Cost_X1;                // X coordinate of cost box (left)
   int Cost_X2;                // X coordinate of cost box (right)
   int Cost_Y1;                // Y coordinate of cost box (top)
   int Cost_Y2;                // Y coordinate of cost box (bottom)
   int VMode;                  // Stores previous video mode
```

```
// Implementation functions
   void Calc_Spacing(const Nodes2&).
                              // Calculates the spacing of the grid
                              // in pixels
   void Move_Cur_X().         // Updates the cursor record to current
                              // x position
   void Move_Cur_Y().         // Updates the cursor record to current
                              // y position
   void Draw_Node() const:    // Draws node at the cursor record
   void Draw_East() const:    // Draws east link at the cursor record
   void Draw_South() const:   // Draws south link at the cursor record
   void Draw_Cell() const:    // Draws cell at the cursor record
   void Print_Cost() const:   // Prints out the cost in the display box
   void Clear_Cost() const.   // Clears the cost display box

public:
   Loop_EGA(){}                    // Vanilla constructor
   Loop_EGA(Nodes2&. int = 0. int = 1. int = 0).
                                   // Constructor
   virtual ~Loop_EGA().            // Destructor
   virtual void Display().         // Draws entire Hanan grid
   virtual void Draw_1_Node().     // Draws one node
   virtual void Draw_1_East().     // Draws one east link
   virtual void Draw_1_South().    // Draws one south link
   virtual void Draw_1_Cell().     // Draws one cell
   void Toggle_Draw_East().        // Draws a toggled east link
   void Toggle_Draw_South().       // Draws a toggled south link

   virtual void Mod_Show() = 0;    // Displays the modification cursor
   virtual void Mod_Hide() = 0;    // Hides the modification cursor
   virtual void Mod_Up() = 0.      // Moves the modification cursor up
   virtual void Mod_Left() = 0;    // Moves the modification cursor up
   virtual void Mod_Down() = 0;    // Moves the modification cursor up
   virtual void Mod_Right() = 0;   // Moves the modification cursor up
   virtual int Toggle_Link() = 0; // Toggles the link at the modification
                                   // cursor position
   virtual int Toggle_Link(int. int. Net_Elem) = 0;
                                   // Toggles the link at the specified
                                   // position
   virtual int Get_Mod_Row() const = 0;
                              // Returns modification cursor row
   virtual int Get_Mod_Col() const = 0;
                              // Returns modification cursor column
   virtual int Get_Mod_Elem() const = 0;
                              // Returns 1 if modification cursor
                              // element is a south link
};

class Loop_Mod : public Loop_EGA {
//-------------------------------------------------------------------------
//
//   Objects of this class operate interactively in addition to
//   being able to display the results and operations of the base
//   "Loop_Test" class.  A separate cursor, the modification cursor,
//   is created.  When this cursor is used it is synchonized with
//   the cursor created by the base classes.  The cursor is
//   initialized located at the center node.  From that point on
//   it is located where it was last used.
//
//   operators
//      Loop_Mod(Nodes2&. int. int. int);
//
//   functions
//      int Modify()
//      Mod_Show()
//      Mod_Hide()
//      Mod_Up()
//      Mod_Left()
//      Mod_Down()
//      Mod_Right()
//      int Toggle_Link()
//      int Toggle_Link(int. int. Net_Elem);
//
//   interface functions
//      int Get_Mod_Row()
//      int Get_Mod_Col()
//      int Get_Mod_Elem()
//      Set_Mod_Row(int)
//      Set_Mod_Col(int)
//      Set_Mod_Elem(Net_Elem)
```

```
//
//-------------------------------------------------------------------------
protected:
   int Mod_Row;                  // Modification cursor - row position
   int Mod_Col;                  // Modification cursor - column position
   Net_Elem Mod_Elem;            // Modification cursor - grid element

public:
   Loop_Mod(){}                  // Vanilla constructor
   Loop_Mod(Nodes2&, int = 0, int = 1, int = 0);
                                 // Constructor
   virtual ~Loop_Mod(){}         // Destructor
   int Modify();                 // Interactive routine that toggles links
   virtual void Mod_Show();      // Displays the modification cursor
   virtual void Mod_Hide();      // Hides the modification cursor
   virtual void Mod_Up();        // Moves the modification cursor up
   virtual void Mod_Left();      // Moves the modification cursor up
   virtual void Mod_Down();      // Moves the modification cursor up
   virtual void Mod_Right();     // Moves the modification cursor up
   virtual int Toggle_Link();
                                 // Toggles the link at the modification
                                 // cursor position
   virtual int Toggle_Link(int, int, Net_Elem);
                                 // Toggles the link at the specified
                                 // position

// Interface functions
   virtual int Get_Mod_Row() const { return Mod_Row; }
                                 // Returns modification cursor row
   virtual int Get_Mod_Col() const { return Mod_Col; }
                                 // Returns modification cursor column
   virtual int Get_Mod_Elem() const
               { return (Mod_Elem == South_El) ? 1 : 0; }
                                 // Returns 1 if modification cursor
                                 // element is a south link
   void Set_Mod_Row(int i){ Mod_Row = i; return; }
                                 // Sets modification cursor row
   void Set_Mod_Col(int i){ Mod_Col = i; return; }
                                 // Sets modification cursor column
   void Set_Mod_Elem(Net_Elem i){ Mod_Elem = i; return; }
                                 // Sets modification cursor element
};

#endif
```

LOOPS3.CPP

```
//-------------------------------------------------------------------------
//
//   This is the file loops3.cpp
//
//-------------------------------------------------------------------------

// Implementation dependencies ---------------------------------------

#ifndef _STDIO_H_INCLUDED
#include <stdio.h>
#endif

#ifndef _CONIO_H_INCLUDED
#include <conio.h>
#endif

#ifndef _GRAPH_H_INCLUDED
#include <graph.h>
#endif

#ifndef __CUSTIO_H
#include "custio.h"
#endif

#ifndef __LOOPS3_H
#include "loops3.h"
#endif

// End implementation dependencies -----------------------------------
```

```
void Graph_Measures::Measure_Graph
//---------------------------------------------------------------------
//
//   This function copies spacing data from the nodes source object
//   and totals the dimensions of the grid.
//
//---------------------------------------------------------------------
(
// Input parameters
      const Nodes2 &N         // Reference to nodes source
)
{
// Local variables
      int i;                  // Counter

   TotalX = TotalY = 0.0;
   for (i = 0; i < Num_Nodes; ++i) {
      TotalX += DX[i] = N.Get_DX(i);
      TotalY += DY[i] = N.Get_DY(i);
   }
   return;
}

void Graph_Measures::Adjust_Size()
//---------------------------------------------------------------------
//
//   This function adjusts the spacing of the grid so that no spacing
//   is smaller than the specified minimum - 8% of the length or
//   width of the grid, which ever is less.
//
//---------------------------------------------------------------------
{
// Local variables
      double MinX;            // Calculated horizontal minimum
      double MinY;            // Calculated vertical minimum
      double Max_Min;         // Minimum spacing
      int i;                  // Counter

   // Find minimum spacing
   MinX = TotalX * 0.08;
   MinY = TotalY * 0.08;
   Max_Min = (MinX > MinY) ? MinX : MinY;

   // Adjust spacing
   for (i = 0; i < Num_Nodes; ++i) {
      if (DX[i] < Max_Min) {
         TotalX -= DX[i];
         DX[i] = Max_Min;
         TotalX += Max_Min;
      }
      if (DY[i] < Max_Min) {
         TotalY -= DY[i];
         DY[i] = Max_Min;
         TotalY += Max_Min;
      }
   }
   return;
}

Graph_Measures::Graph_Measures
//---------------------------------------------------------------------
//
//   This function allocates space for the object, gets the data
//   and adjusts to ensure no spacing is below the minimum.
//
//---------------------------------------------------------------------
(
// Input parameters
      const Nodes2 &N         // Reference to nodes source
)
{
   Num_Nodes = N.Get_Num();
   DX = new double[Num_Nodes];
   DY = new double[Num_Nodes];
   Measure_Graph(N);
   Adjust_Size();
}

Graph_Measures::~Graph_Measures()
//---------------------------------------------------------------------
```

```
//
//   This destructor frees space allocated to the object
//
//-------------------------------------------------------------------------
{
   delete DY;
   delete DX;
}

void Loop_EGA::Calc_Spacing
//-------------------------------------------------------------------------
//
//   This function calculates the spacing of the Hanan grid in
//   in pixels.
//
//-------------------------------------------------------------------------
(
   Input parameters
      const Nodes2 &Problem  // Reference to nodes data
)
{
// Local variables
      Graph_Measures Mod_Graph(Problem);
                             // Temporary storage of adjusted grid
                             // dimensions
      videoconfig Vid;       // Stores video configuration
      int StartX;            // X coordinate of origin of grid
      int StartY;            // Y coordinate of origin of grid
      int SpaceX;            // Maximum width of graph in pixels
      int SpaceY;            // Maximum length of graph in pixels
      double ScaleX;         // Scale calculated based on x coordinates
      double ScaleY;         // Scale calculated based on y coordinates
      double Scale;          // Scale of grid
      int i;                 // Counter

   // Get video configuration information
   _getvideoconfig(&Vid);

   // Calculate available pixels for grid
   StartX = Vid.numxpixels / 20;
   StartY = Vid.numypixels / 20;
   SpaceX = (Vid.numxpixels - (2 * StartX));
   SpaceY = (Vid.numypixels - (2 * StartY));

   // Ensure same scale
   ScaleX = Mod_Graph.Get_TotalX() / ((double) SpaceX);
   ScaleY = Mod_Graph.Get_TotalY() / ((double) SpaceY);
   Scale = (ScaleX > ScaleY) ? ScaleX : ScaleY;

   // Readjust spacing
   StepX[0] = StepY[0] = SpaceX = SpaceY = 0;
   for (i = 0; i < Num_Nodes; ++i) {
      StepX[i + 1] = SpaceX += (int) (Mod_Graph.Get_DX(i) / Scale);
      StepY[i + 1] = SpaceY += (int) (Mod_Graph.Get_DY(i) / Scale);
   }
   StartX = (Vid.numxpixels - SpaceX) / 2;
   StartY = (Vid.numypixels - SpaceY) / 2;
   for (i = 0; i <= Num_Nodes; ++i) {
      StepX[i] += StartX;
      StepY[i] += StartY;
   }

   // Set dimensions of cost display rectangle
   Cost_X1 = 444;
   Cost_X2 = 516;
   Cost_Y1 = 332;
   Cost_Y2 = 342;
   return;
}

void Loop_EGA::Move_Cur_X()
//-------------------------------------------------------------------------
//
//   This function assumes the column in which the cursor is located
//   has changed and updates the "Cursor" data member based on the
//   value of the data member "Col".
//
//-------------------------------------------------------------------------
{
   if (Display_Mode > 0) {
```

```
      Cursor.X = StepX[Col];
      Cursor.XP = StepX[Col] + 5;
      Cursor.XM = StepX[Col + 1] - 5;
   }
   return;
}

void Loop_EGA::Move_Cur_Y()
//---------------------------------------------------------------------
//
//   This function assumes the row in which the cursor is located
//   has changed and updates the "Cursor" data member based on the
//   value of the data member "Row".
//
//---------------------------------------------------------------------
{
   if (Display_Mode > 0) {
      Cursor.Y = StepY[Row];
      Cursor.YP = StepY[Row] + 4;
      Cursor.YM = StepY[Row + 1] - 4;
   }
   return;
}

void Loop_EGA::Draw_Node() const
//---------------------------------------------------------------------
//
//   This function displays a single node with the appropriate
//   color at the location specified in the data member "Cursor"
//
//---------------------------------------------------------------------
{
// Local variables
      int Color;              // Numerical value of selected color

   if (Display_Mode > 0) {

      // Select color and draw link if display mode is greater
      // than 2
      if (Display_Mode > 2) {
         switch (Node_Copy[Row][Col]) {
            case -2:
               Color = RED;
               break;
            case -1:
               Color = YELLOW;
               break;
            case 0:
               Color = BLUE;
               break;
            case 1:
               Color = LIGHTGREEN;
         }
         _setcolor(WHITE);
         _ellipse(_GBORDER, Cursor.X - 3, Cursor.Y - 3,
                  Cursor.X + 3, Cursor.Y + 3);
         _setcolor(Color);
         _ellipse(_GBORDER, Cursor.X - 2, Cursor.Y - 2,
                  Cursor.X + 2, Cursor.Y + 2);
         _setpixel(Cursor.X, Cursor.Y);

      // Select color and draw node if display mode
      // is less than or equal to 2
      } else if (Node_Copy[Row][Col] != 0) {
         if ((Node_Copy[Row][Col] == -2) || (Node_Copy[Row][Col] == 1)) {
            _setcolor(LIGHTRED);
            _ellipse(_GBORDER, Cursor.X - 2, Cursor.Y - 2,
                     Cursor.X + 2, Cursor.Y + 2);
            _setpixel(Cursor.X, Cursor.Y);
         }
      }
   }
   return;
}

void Loop_EGA::Draw_East() const
//---------------------------------------------------------------------
//
//   This function displays a single south link with the appropriate
//   color at the location specified in the data member "Cursor".
```

```
//
//---------------------------------------------------------------------
{
// Local variables
      int Color.              // Numerical value of selected color

   if (Display_Mode > 0) {

      // Select color and draw link if display mode is greater
      // than 2
      if (Display_Mode > 2) {
         switch (East_Copy[Row][Col]) {
            case -2
               Color = LIGHTRED.
               break.
            case -1.
               Color = YELLOW.
               break;
            case 0
               Color = BLACK.
               break.
            case 1.
               Color = LIGHTGRAY.
         }
         _setcolor(Color).
         _moveto(Cursor.XP. Cursor.Y - 1).
         _lineto(Cursor.XM. Cursor.Y - 1).
         _moveto(Cursor.XP. Cursor.Y).
         _lineto(Cursor.XM. Cursor.Y).
         _moveto(Cursor.XP. Cursor.Y + 1).
         _lineto(Cursor.XM. Cursor.Y + 1).

      // Select color and draw link if display mode
      // is less than or equal to 2
      } else {
         switch (East_Copy[Row][Col]) {
            case 1
               Color = GREEN.
               break.
            case 0.
               Color = BLACK.
               break;
            default
               Color = YELLOW.
         }
         _setcolor(Color).
         _moveto(Cursor.XP. Cursor.Y).
         _lineto(Cursor.XM. Cursor.Y).
      }
   }
   return.
}

void Loop_EGA::Draw_South() const
//---------------------------------------------------------------------
//
//   This function displays a single south link with the appropriate
//   color at the location specified in the data member "Cursor"
//
//---------------------------------------------------------------------
{
// Local variables
      int Color.              // Numerical value of selected color

   if (Display_Mode > 0) {

      // Select color and draw link if display mode is greater
      // than 2
      if (Display_Mode > 2) {
         switch (South_Copy[Row][Col]) {
            case -2
               Color = LIGHTRED.
               break.
            case -1.
               Color = YELLOW.
               break;
            case 0
               Color = BLACK.
               break;
            case 1:
```

```
                Color = LIGHTGRAY;
            }
            _setcolor(Color);
            _moveto(Cursor.X - 1, Cursor.YP + 1);
            _lineto(Cursor.X - 1, Cursor.YM - 1);
            _moveto(Cursor.X, Cursor.YP + 1);
            _lineto(Cursor.X, Cursor.YM - 1);
            _moveto(Cursor.X + 1, Cursor.YP + 1);
            _lineto(Cursor.X + 1, Cursor.YM - 1);

         // Select color and draw link if display mode
         // is less than or equal to 2
         } else {
            switch (South_Copy[Row][Col]) {
               case 1:
                  Color = GREEN;
                  break;
               case 0:
                  Color = BLACK;
                  break;
               default:
                  Color = YELLOW;
            }
            _setcolor(Color);
            _moveto(Cursor.X, Cursor.YP);
            _lineto(Cursor.X, Cursor.YM);
         }
      }
      return;
}

void Loop_EGA::Draw_Cell() const
//---------------------------------------------------------------------
//
//   This function displays a single cell with the appropriate color
//   at the location specified in the data member "Cursor"
//
//---------------------------------------------------------------------
{
   Local variables
      int i;                  // Counter
      int j;                  // Counter
      int Color;              // Numerical value of selected color

   if (Display_Mode > 0) {

      // Determine the appropriate color based on the contents
      // of the cell
      switch (Cell[Row][Col]) {
         case -2:
            Color = RED;
            break;
         case -1:
            Color = LIGHTRED;
            break;
         case 0:
            Color = BLUE;
            break;
         case 1:
            Color = CYAN;
            break;
         default:
            Color = ((Cell[Row][Col] - 2) % 5) + 5;
      }
      _setcolor(Color);
      _rectangle(_GBORDER, Cursor.XP, Cursor.YP, Cursor.XM, Cursor.YM);

      // Fill the cell
      for (i = Cursor.XP + 2; i <= Cursor.XM; i += 2)
         for (j = Cursor.YP + 2; j <= Cursor.YM; j += 3) {
            _setpixel(i - 1, j);
            _setpixel(i, j - 1);
         }
   }
   return;
}

void Loop_EGA::Print_Cost() const
//---------------------------------------------------------------------
//
```

```
//   This function displays the cost (length) of the network in its
//   current state
//
//-------------------------------------------------------------------------
{
// Local variables
      char OutBuff[40];       // Output buffer

   if (Display_Mode > 0) {
      Clear_Cost();
      if (E_State == Evaluated) {
         _setcolor(WHITE);
         sprintf(OutBuff, "%8.1f", Cost);
         _grtext(Cost_X1, Cost_Y1 + 2, OutBuff);
      }
   }
   return;
}

void Loop_EGA::Clear_Cost() const
//-------------------------------------------------------------------------
//
//   This function clears the area of the screen where the running
//   total of cost (length) is displayed.
//
//-------------------------------------------------------------------------
{
// Local variables
      int x;                  // X coordinate
      int y;                  // Y coordinate

   if (Display_Mode > 0) {
      _setcolor(BLACK);
      for (y = Cost_Y1; y <= Cost_Y2; ++y)
         for (x = Cost_X1; x < Cost_X2; ++x)
            _setpixel(x, y);
   }
   return;
}

Loop_EGA::Loop_EGA
//-------------------------------------------------------------------------
//
//   This constructor allocates space for the object, sets up
//   spacing of the grid and initializes video page.
//
//-------------------------------------------------------------------------
{
// Input parameters
      Nodes2 &Problem,        // Reference to nodes data
      int New_D_Mode,         // Display mode (default is 0)
      int New_Block,          // Display of improvement step (default
                              // is 0)
      int New_Key             // Pause for key (default is 0)
)
  Loop_Test(Problem, New_D_Mode, New_Block, New_Key)
{
// Local variables
      videoconfig Vid;        // Stores video configuration

   _getvideoconfig(&Vid);
   VMode = Vid.mode;
   if (New_D_Mode > 0) {
      _setvideomode(_ERESCOLOR);
      _setcharsize(10, 7);
      _setcharspacing(2);
      _setactivepage(0);
      _setvisualpage(0);
      StepX = new int[Num_Nodes + 1];
      StepY = new int[Num_Nodes + 1];
      Calc_Spacing(Problem);
      Page = 0;
   } else
      StepX = StepY = NULL;
}


Loop_EGA::~Loop_EGA()
//-------------------------------------------------------------------------
//
```

```
//   This destructor frees space allocated to the object
//
//-----------------------------------------------------------------------
{
// Local variables
   int i;                    // Counter

   if (StepX) {
      delete StepY;
      delete StepX;
   }
   _setvideomode(VMode);
}

void Loop_EGA::Display()
//-----------------------------------------------------------------------
//
//   This function draws the entire Hanan grid and pauses until a
//   key is pressed if required.  For a smooth display the grid is
//   drawn on a separate page of the video card and then that page
//   is displayed
//
//-----------------------------------------------------------------------
{
// Local variables
   char OutBuff[40];         // Output Buffer

   if (Display_Mode > 0) {

      // Change the active page to a new page and clear it
      Page = (Page) ? 0 : 1;
      _setactivepage(Page);
      _clearscreen(_GCLEARSCREEN);

      // Draw all the elements in the Hanan grid
      for (Row = 0; Row < Num_Nodes; ++Row) {
         Move_Cur_Y();
         for (Col = 0; Col < Num_Nodes; ++Col) {
            Move_Cur_X();
            Draw_Node();
            Draw_East();
            Draw_South();
            if (Display_Mode > 2)
               Draw_Cell();
         }
      }

      // Display the cost (length)
      _setcolor(WHITE);
      sprintf(OutBuff, "Total length: ");
      _grtext(320, 332, OutBuff);
      Print_Cost();

      // Display the new page
      _setvisualpage(Page);

      // Pause until key is pressed if required
      if (Key_Pause) {
         while(!kbhit());
         getch();
      }

   }
   return;
}

void Loop_EGA::Draw_1_Node()
//-----------------------------------------------------------------------
//
//   This function draws a node at the location specified by the
//   data memebers "Row" and "Col" and pauses until a key is
//   pressed if required.
//
//-----------------------------------------------------------------------
{
   if (Display_Mode > 0) {
      Move_Cur_Y();
      Move_Cur_X();
      Draw_Node();
      if (Key_Pause) {
```

```
            while(!kbhit());
            getch();
         }
      }
      return;
}

void Loop_EGA::Draw_1_East()
//---------------------------------------------------------------------
//
//    This function draws a single east link at the location
//    specified by the data memebers "Row" and "Col" and pauses
//    until a key is pressed if required.
//
//---------------------------------------------------------------------
{
      if (Display_Mode > 0) {
         Move_Cur_Y();
         Move_Cur_X();
         Draw_East();
         if (Key_Pause) {
            while(!kbhit());
            getch();
         }
      }
      return;
}

void Loop_EGA::Draw_1_South()
//---------------------------------------------------------------------
//
//    This function draws a single south link at the location
//    specified by the data memebers "Row" and "Col" and pauses
//    until a key is pressed if required.
//
//---------------------------------------------------------------------
{
      if (Display_Mode > 0) {
         Move_Cur_Y();
         Move_Cur_X();
         Draw_South();
         if (Key_Pause) {
            while(!kbhit());
            getch();
         }
      }
      return;
}

void Loop_EGA::Draw_1_Cell()
//---------------------------------------------------------------------
//
//    This function draws a single cell at the location specified
//    by the data memebers "Row" and "Col" and pauses until a key
//    is pressed if required.
//
//---------------------------------------------------------------------
{
      if (Display_Mode > 0) {
         Move_Cur_Y();
         Move_Cur_X();
         Draw_Cell();
         if (Key_Pause) {
            while(!kbhit());
            getch();
         }
      }
      return;
}

void Loop_EGA::Toggle_Draw_East()
//---------------------------------------------------------------------
//
//    This function assumes the east link at the cursor position
//    specified by the data memebers "Row" and "Col" has been altered
//    and the value of the cost (length) has been updated. The link
//    is redrawn and the updated cost displayed.
//
//---------------------------------------------------------------------
{
```

```
   if (Display_Mode > 0) {
      Move_Cur_Y();
      Move_Cur_X();
      Draw_East();
      Print_Cost();
   }
   return;
}

void Loop_EGA::Toggle_Draw_South()
//---------------------------------------------------------------------
//
//   This function assumes the south link at the cursor position
//   specified by the data memebers "Row" and "Col" has been altered
//   and the value of the cost (length) has been updated.  The link
//   is redrawn and the updated cost displayed.
//
//---------------------------------------------------------------------
{
   if (Display_Mode > 0) {
      Move_Cur_Y();
      Move_Cur_X();
      Draw_South();
      Print_Cost();
   }
   return;
}

void Loop_Mod::Mod_Show()
//---------------------------------------------------------------------
//
//   This function is a drawing routine that shows the modification
//   cursor.
//
//---------------------------------------------------------------------
{
   if (Display_Mode > 0) {

      // Modification cursor is synchronized with cursor
      Row = Mod_Row;
      Col = Mod_Col;

      // Cursor record is updated
      Move_Cur_Y();
      Move_Cur_X();

      // Appropriate element is drawn
      if (Display_Mode > 2)
         _setcolor(LIGHTRED);
      else
         _setcolor(WHITE);
      switch (Mod_Elem) {
         case Node_El:
            _ellipse(_GBORDER, Cursor.X - 3, Cursor.Y - 3,
                     Cursor.X + 3, Cursor.Y + 3);
            break;
         case East_El:
            _rectangle(_GBORDER, Cursor.XP, Cursor.Y - 2,
                       Cursor.XM, Cursor.Y + 2);
            break;
         case South_El:
            _rectangle(_GBORDER, Cursor.X - 2, Cursor.YP,
                       Cursor.X + 2, Cursor.YM);
            break;
         case Cell_El:
            _rectangle(_GBORDER, Cursor.XP - 1, Cursor.YP - 1,
                       Cursor.XM + 1, Cursor.YM + 1);
      }
   }
   return;
}

void Loop_Mod::Mod_Hide()
//---------------------------------------------------------------------
//
//   This function is a drawing routine that hides the modification
//   cursor.
//
//---------------------------------------------------------------------
{
```

```
   if (Display_Mode > 0) {

      // Modification cursor is synchronized with cursor
      Row = Mod_Row;
      Col = Mod_Col;

      // Cursor record is updated
      Move_Cur_Y();
      Move_Cur_X();

      // Appropriate element is "un-drawn"
      switch (Mod_Elem) {
         case Node_El:
            if (Display_Mode > 2)
               _setcolor(WHITE);
            else
               _setcolor(BLACK);
            _ellipse(_GBORDER, Cursor.X - 3, Cursor.Y - 3,
                     Cursor.X + 3, Cursor.Y + 3);
            break;
         case East_El:
            _setcolor(BLACK);
            _rectangle(_GBORDER, Cursor.XP, Cursor.Y - 2,
                       Cursor.XM, Cursor.Y + 2);
            break;
         case South_El:
            _setcolor(BLACK);
            _rectangle(_GBORDER, Cursor.X - 2, Cursor.YP,
                       Cursor.X + 2, Cursor.YM);
            break;
         case Cell_El:
            _setcolor(BLACK);
            _rectangle(_GBORDER, Cursor.XP - 1, Cursor.YP - 1,
                       Cursor.XM + 1, Cursor.YM + 1);
      }
   }
   return;
}

void Loop_Mod::Mod_Up()
//-----------------------------------------------------------------------
//
//   This function moves the modification cursor one element up
//   The 2-torus "wrap-around" properties of the Hanan grid are
//   accommodated
//
//-----------------------------------------------------------------------
{
   switch (Mod_Elem) {
      case Node_El:
         if (Mod_Row == 0)
            Mod_Row = Num_Nodes - 1;
         else {
            --Mod_Row;
            Mod_Elem = South_El;
         }
         break;
      case East_El:
         if (Mod_Row == 0)
            Mod_Row = Num_Nodes - 1;
         else {
            --Mod_Row;
            Mod_Elem = Cell_El;
         }
         break;
      case South_El:
         Mod_Elem = Node_El;
         break;
      case Cell_El:
         Mod_Elem = East_El;
   }
   return;
}

void Loop_Mod::Mod_Left()
//-----------------------------------------------------------------------
//
//   This function moves the modification cursor one element to the
//   left. The 2-torus "wrap-around" properties of the Hanan grid
//   are accommodated
```

```
//
//-------------------------------------------------------------------------
{
   switch (Mod_Elem) {
      case Node_El:
         if (Mod_Col == 0)
            Mod_Col = Num_Nodes - 1;
         else {
            --Mod_Col;
            Mod_Elem = East_El;
         }
         break;
      case East_El:
         Mod_Elem = Node_El;
         break;
      case South_El:
         if (Mod_Col == 0)
            Mod_Col = Num_Nodes - 1;
         else {
            --Mod_Col;
            Mod_Elem = Cell_El;
         }
         break;
      case Cell_El:
         Mod_Elem = South_El;
   }
   return;
}

void Loop_Mod::Mod_Down()
//-------------------------------------------------------------------------
//
//   This function moves the modification cursor one element down
//   The 2-torus "wrap-around" properties of the Hanan grid are
//   accommodated.
//
//-------------------------------------------------------------------------
{
   switch (Mod_Elem) {
      case Node_El:
         if (Mod_Row == Num_Nodes - 1)
            Mod_Row = 0;
         else
            Mod_Elem = South_El;
         break;
      case East_El:
         if (Mod_Row == Num_Nodes - 1) {
            Mod_Row = 0;
         } else {
            Mod_Elem = Cell_El;
         }
         break;
      case South_El:
         ++Mod_Row;
         Mod_Elem = Node_El;
         break;
      case Cell_El:
         ++Mod_Row;
         Mod_Elem = East_El;
   }
   return;
}

void Loop_Mod::Mod_Right()
//-------------------------------------------------------------------------
//
//   This function moves the modification cursor one element to the
//   right. The 2-torus "wrap-around" properties of the Hanan grid
//   are accommodated.
//
//-------------------------------------------------------------------------
{
   switch (Mod_Elem) {
      case Node_El:
         if (Mod_Col == Num_Nodes - 1)
            Mod_Col = 0;
         else
            Mod_Elem = East_El;
         break;
      case East_El:
```

```
            ++Mod_Col;
            Mod_Elem = Node_El;
            break;
         case South_El:
            if (Mod_Col == Num_Nodes - 1)
               Mod_Col = 0;
            else
               Mod_Elem = Cell_El;
            break;
         case Cell_El:
            ++Mod_Col;
            Mod_Elem = South_El;
      }
      return;
}

int Loop_Mod_Toggle_Link()
//---------------------------------------------------------------------
//
//    This function toggles the value of an east or a south link and
//    updates the cost.  The position and element type are determined
//    by the data members "Mod_Row", "Mod_Col" and "Mod_Elem".  Returns
//    1 if a link is added.  Returns -1 if a link is removed and
//    returns 0 if neither.
//
//---------------------------------------------------------------------
{
   return Toggle_Link(Mod_Row, Mod_Col, Mod_Elem);
}

int Loop_Mod::Toggle_Link
//---------------------------------------------------------------------
//
//    This function toggles the value of an east or a south link and
//    updates the cost.  Returns 1 if a link is added.  Returns -1 if
//    a link is removed and returns 0 if neither.
//
//---------------------------------------------------------------------
(
// Input parameters
      int i,                  // Row position
      int j                   // Column position
      Net_Elem El             // Element type

{
      int LinkP               // Registers if link has been added or
                              // removed

   // Set the cursor to the position specified
   Row = i;
   Col = j;

   // Initialize LinkP
   LinkP = 0

   // Perform operation based on element type
   switch (El) {

      // If element is an east link
      case East_El:

         // If the east link is selected remove it and
         // update cost
         if (East_Copy[Row][Col]) {
            East_Copy[Row][Col] = 0;
            if (E_State == Evaluated)
               Cost -= Prob_Source->Get_DX(Col);
            LinkP = -1;

         // If east link is not selected add it and
         // update cost
         } else {
            East_Copy[Row][Col] = 1;
            if (E_State == Evaluated)
               Cost += Prob_Source->Get_DX(Col);
            LinkP = 1;
         }

         // Display if required
         if (Display_Mode > 0)
```

```
            Toggle_Draw_East().
         break.

      // If element is a south link
      case South_El.

         // If the south link is selected remove it and
         // update cost
         if (South_Copy[Row][Col]) {
            South_Copy[Row][Col] = 0.
            if (E_State == Evaluated)
               Cost -= Prob_Source->Get_DY(Row).
            LinkP = -1.

         // If south link is not selected add it and
         // update cost
         } else {
            South_Copy[Row][Col] = 1.
            if (E_State == Evaluated)
               Cost += Prob_Source->Get_DY(Row).
            LinkP = 1.
         }
         // Display if required
         if (Display_Mode > 0)
            Toggle_Draw_South()
   }

   // Toggle of links may change feasibility
   T_State = Unknown.
   return LinkP.
}

Loop_Mod::Loop_Mod
//---------------------------------------------------------------------
//
//   This constructor call the constructor of the base class and
//   initializes the position of the modification cursor at the
//   center node
//
//---------------------------------------------------------------------

   Input parameters
      Nodes2 &Problem.        // Reference to nodes data
      int New_D_Mode.         // Display mode (default is 0)
      int New_Block.          // Display of improvement step (default
                              // is 0)
      int New_Key             // Pause for key (default is 0)
)
  Loop_EGA(Problem, New_D_Mode, New_Block, New_Key)
{
   Mod_Row = Mod_Col = Num_Nodes / 2.
   Mod_Elem = Node_El.
}

int Loop_Mod::Modify()
//---------------------------------------------------------------------
//
//   This function is the top level interactive function that
//   displays the cursor, moves it around the grid, toggles links
//   and hides the cursor if the end or insert keys are pressed. A
//   value of 1 is returned if the insert key was pressed and 0 if
//   the end key was pressed
//
//---------------------------------------------------------------------
{
// Local variables
      unsigned short Choice; // Value of key that was pressed
      int i.                 // Counter
      int j.                 // Counter


   // Display cursor
   Mod_Show().

   // Wait for key to be pressed
   Choice = Pause_Key().

   // If key is not end or insert key continue
   while ((Choice != 207) && (Choice != 210)) {    // End and insert keys
```

```
        // If key is not return, end or insert continue
        while ((Choice != 13) && (Choice != 207) && (Choice != 210)) {
                        // Return, end and insert keys

           // Hide the cursor
           Mod_Hide();

           // Move the cursor
           switch (Choice) {
              // Cursor up key
              case 200:
                 Mod_Up();
                 break;
              // Cursor left key
              case 203:
                 Mod_Left();
                 break;
              // Cursor down key
              case 208:
                 Mod_Down();
                 break;
              // Cursor right key
              case 205:
                 Mod_Right();
                 break;
           }

           // Show the cursor
           Mod_Show();

           // Wait for a new key to be pressed
           Choice = Pause_Key();
        }

        // If the key was a return key toggle the value
        // of the element
        if (Choice == 13) {     // Return key
           Toggle_Link(Mod_Row, Mod_Col, Mod_Elem);
           Choice = Pause_Key();
        }
     }

     // Hide the cursor
     Mod_Hide();

     // Return 1 if the insert key was pressed
     // or 0 if the end key was pressed
     return (Choice == 210) ? 1 : 0;
}
```

LINKS.H

```
//---------------------------------------------------------------------
//
//   This is the file links.h
//
//---------------------------------------------------------------------

// Interface dependencies ---------------------------------------------

#ifndef _FSTREAM_H_INCLUDED
#include <fstream.h>
#endif

#ifndef __CHAOS_H
#include "chaos.h"
#endif

#ifndef __LOOPS2_H
#include "loops2.h"
#endif

// End interface dependencies -----------------------------------------

#ifndef __LINKS_H
#define __LINKS_H
```

```
class Links : public Solution {
//---------------------------------------------------------------------
//
//   This class contains the south and east links matrices the
//   comprise a solution to the layout problem
//
//   operators
//      Links(int)
//      ostream << Links
//      Links = Chaos_Str
//      Links = Links
//      Links >> Loop_Base
//      Links = Loop_Base
//      --Links
//      +=Links
//      -=Links
//
//   functions
//      Links.XOR(Links, Links)
//
//   interface functions
//      int Get_Num()
//      int Get_East(int, int)
//      int Get_South(int, int)
//      Clear_East(int, int)
//      Clear_South(int, int)
//      Set_East(int, int)
//      Set_South(int, int)
//
//---------------------------------------------------------------------
protected:
   int Num_Nodes;              // Number of nodes in the solution
   int **East;                 // Matrix of east links
   int **South;                // Matrix of south links

   void Allocate();            // Allocates space for matrices
   virtual void Print(ostream&)
                               // Prints out object

public:
   Links(){}                    // Vanilla constructor
   Links(istream&, ostream&, int);
                               // Creates from input stream
   Links(char*);               // Creates from a file
   Links(int);                 // Creates from number of nodes
   virtual ~Links();           // Frees space
   Links &operator =(Chaos_Str &);
                               // Copys layout from chaotic bits
   Links &operator =(Links&);
                               // Copys layout from Links object
   void operator >>(Loop_Base&) const;
                               // Copies the contents of this object to
                               // tester/display object
   Links &operator =(const Loop_Base&);
                               // Copies the contents of tester/display
                               // object to this object
   Links &operator --();       // Sets all links to zero
   Links &operator +=(Links&);
                               // Performs logical or with Links object
   Links &operator -=(Links&);
                               // Removes all duplicate links
   friend ostream &operator <<(ostream &, Links &);
                               // Print operator
   void XOR(const Links&, const Links&);
                               // Performs bitwise exclusive OR

// Interface functions
   int Get_Num() const { return Num_Nodes; }
                               // Returns the number of nodes
   int Get_East(int i, int j) const { return East[i][j]; }
                               // Returns the value of the east links
                               // matrix at row i and column j
   int Get_South(int i, int j) const { return South[i][j]; }
                               // Returns the value of the south links
                               // matrix at row i and column j
   void Clear_East(int i, int j){ East[i][j] = 0; return; }
                               // Sets the value of the east links
                               // matrix at row i and column j to 0
   void Clear_South(int i, int j){ South[i][j] = 0; return; }
                               // Sets the value of the south links
```

```
                                // matrix at row i and column j to 0
   void Set_East(int i, int j){ East[i][j] = 1; return; }
                                // Sets the value of the east links
                                // matrix at row i and column j to 1
   void Set_South(int i, int j){ South[i][j] = 1; return; }
                                // Sets the value of the south links
                                // matrix at row i and column j to 1
};

#endif
```

LINKS.CPP

```
//---------------------------------------------------------------------
//
//   This is the file links.cpp
//
//---------------------------------------------------------------------

// Implementation dependencies ----------------------------------------

#ifndef _STDLIB_H_INCLUDED
#include <stdlib.h>
#endif

#ifndef __CUSTIO_H
#include "custio.h"
#endif

#ifndef __LINKS_H
#include "links.h"
#endif

// End implementation dependencies ------------------------------------

void Links::Allocate()
//---------------------------------------------------------------------
//
//   This function allocates space for the two matrices South and
//   East.
//
//---------------------------------------------------------------------
{
// Local variable
      int i;                    // Counter

   // Allocate space for matrices
   South = new int*[Num_Nodes];
   East = new int*[Num_Nodes];
   for (i = 0; i < Num_Nodes; ++i) {
      South[i] = new int[Num_Nodes];
      East[i] = new int[Num_Nodes];
   }
   return;
}

void Links::Print
//---------------------------------------------------------------------
//
//   This function prints out the object in table form. Output
//   files created by this routine can be read by
//   Links::Create(char*).
//
//---------------------------------------------------------------------
(
// Input parameters
      ostream &Out          // Output stream
)
{
// Local variables
      int i, j;             // Counters

   // Print out the data
   Set_Int(3);
   (Out << "Total number of nodes " << Num_Nodes) << "\n\n";
   Out << "South Links\n";
   Out << "___________\n";
```

```
   Set_Int(i);
   for (i = 0; i < Num_Nodes - 1; ++i) {
      for (j = 0; j < Num_Nodes; ++j) {
         Out < South[i][j];
      }
      Out << "\n";
   }
   Out << "\n\n";
   Out << "East Links\n";
   Out << "----------\n";
   for (i = 0; i < Num_Nodes; ++i) {
      for (j = 0; j < Num_Nodes - 1; ++j) {
         Out < East[i][j];
      }
      Out << "\n"
   }
   Out << "\n\n";
   return;
}

Links::Links
//---------------------------------------------------------------------

//
//   This constructor creates the object from input from the keyboard
//
//---------------------------------------------------------------------

// Input parameters
      istream &In,            // Input stream
      ostream &Out,           // Output stream
      int New_Num             // Number of nodes
)
  Solution()
{
// Local variables
   int Input;                 // Flag (1 = input required)
   int i, j;                  // Counters

   // Get the number of nodes
   Num_Nodes = New_Num;
   Out << "The number of nodes is " << Num_Nodes << "\n";
   Allocate();

   // Input data
   Out << "Input south link data\n";
   for (i = 0; i < Num_Nodes - 1; ++i) {
      for (j = 0; j < Num_Nodes; ++j) {
         Input = 1;
         while (Input) {
            Out << "   Link(" << i + 1 << "," << j + 1 << ")  ";
            In >> South[i][j];
            if ((South[i][j] == 0) || (South[i][j] == 1)) Input = 0;
         }
      }
      Out << "\n";
   }
   Out << "\n\n";
   Out << "Input east link data\n";
   for (i = 0; i < Num_Nodes; ++i) {
      for (j = 0; j < Num_Nodes - 1; ++j) {
         Input = 1;
         while (Input) {
            Out << "   Link(" << i + 1 << "," << j + 1 << "): ";
            In >> East[i][j];
            if ((East[i][j] == 0) || (East[i][j] == 1)) Input = 0;
         }
      }
      Out << "\n";
   }
   Out << "\n\n";

   // Set the last row of South and last column East to zero
   for (i = 0; i < Num_Nodes; ++i) {
      South[Num_Nodes - 1][i] = 0;
      East[i][Num_Nodes - 1] = 0;
   }
}

Links::Links
//---------------------------------------------------------------------
```

```
//
//   This constructor creates the object from input from a file
//   Input file is in the same format as output from
//   Links::Print(ostream&)
//
//-------------------------------------------------------------------------
(
// Input parameters
      char *Infile            // Input file
)
  Solution()
{
// Local variables
      char C;                 // Input buffer for single character
      int N;                  // Num_Nodes - 1 (frequently used)
      int i, j;               // Indices to array
      int k;                  // Pointer to byte in file

   ifstream In(Infile);
      In.seekg(22, ios::beg);
      In >> Num_Nodes;
      N = Num_Nodes - 1;
      Allocate();
      k = 55;
      for (i = 0; i < N; ++i) {
         for (j = 0; j < Num_Nodes; ++j) {
            In.seekg(k, ios::beg);
            In >> C;
            if (C == '0')
               South[i][j] = 0;
            else if (C == '1')
               South[i][j] = 1;
            else {
               cerr << "\n<Links 01> Error: Reading input for south links\n";
               exit(1);
            }
            ++k;
         }
         k += 2;
      }
      k += 28;
      for (i = 0; i < Num_Nodes; ++i) {
         for (j = 0; j < N; ++j) {
            In.seekg(k, ios::beg);
            In >> C;
            if (C == '0')
               East[i][j] = 0;
            else if (C == '1')
               East[i][j] = 1;
            else {
               cerr << "\n<Links 02> Error: Reading input for east links\n";
               exit(1);
            }
            ++k;
         }
         k += 2;
      }
   In.close();

   // Set the last row of South and last column East to zero
   for (i = 0; i < Num_Nodes; ++i) {
      South[N][i] = 0;
      East[i][N] = 0;
   }
}

Links::Links
//-------------------------------------------------------------------------
//
//   This constructor allocates space for the links based on the
//   number of nodes specified by the input parameter.  All links
//   are initialized to zero.
//
//-------------------------------------------------------------------------
(
// Input parameters
      int New_Num             // Number of nodes
)
: Solution()
{
```

```
   Num_Nodes = New_Num;
   Allocate();
   operator --();
}

Links::~Links()
//---------------------------------------------------------------------------
//
//   This destructor frees space allocated by Links::Allocate().
//
//---------------------------------------------------------------------------
{
// Local variable
      int i;                    // Counter

   // Free space
   for (i = Num_Nodes - 1; i >= 0; --i) {
      delete East[i];
      delete South[i];
   }
   delete East;
   delete South;
}

Links &Links::operator =
//---------------------------------------------------------------------------
//
//   This operator copies the bit data from the Chaos_Str object
//   to south and east links matrices.
//
//---------------------------------------------------------------------------
(
// Input parameter
      Chaos_Str &Chaos          // Pointer to chaos bit
)
{
// Local variables
      int N;                    // Number of nodes minus 1
      int i, j;                 // Array indices


   // Copy bits over from bit string
   N = Num_Nodes - 1;
   for (i = 0; i < N; ++i)
      for (j = 0; j < Num_Nodes; ++j)
         South[i][j] = ++Chaos;
   for (i = 0; i < Num_Nodes; ++i)
      for (j = 0; j < N; ++j)
         East[i][j] = ++Chaos;
   // Set the last row of South and last column East to zero
   for (i = 0; i < Num_Nodes; ++i) {
      South[Num_Nodes - 1][i] = 0;
      East[i][Num_Nodes - 1] = 0;
   }
   return *this;
}

Links &Links::operator =
//---------------------------------------------------------------------------
//
//   This operator copies the links data from a Links type object.
//
//---------------------------------------------------------------------------
(
// Input parameter
   Links &New_Links          // Pointer to links data
)
{
// Local variables
      int i, j;                 // Array indices

   for (i = 0; i < Num_Nodes; ++i)
      for (j = 0; j < Num_Nodes; ++j) {
         South[i][j] = New_Links.South[i][j];
         East[i][j] = New_Links.East[i][j];
      }
   return *this;
}

void Links::operator >>
```

```
//---------------------------------------------------------------------
//
//   This operator copies the links data to a tester/display object.
//
//---------------------------------------------------------------------
(
// Input parameters
      Loop_Base &Tester       // Reference to tester/display object
)
const
{
// Local variables
      int i.                  // Row index
      int j.                  // Column index

   // Copy the contents of the links matrices to the
   // tester/display object
   for (i = 0; i < Num_Nodes; ++i)
      for (j = 0; j < Num_Nodes; ++j) {
         if (East[i][j])
            Tester.Set_East(i, j);
         else
            Tester.Clear_East(i, j);
         if (South[i][j])
            Tester.Set_South(i, j);
         else
            Tester.Clear_South(i, j);
      }

   // Set state variables
   Tester.Put_B_State(Loop_Base::Binary);
   Tester.Put_T_State(Loop_Base::Unknown);
   Tester.Put_E_State(Loop_Base::Not_Evaluated);
   Tester.Put_Sol((Solution*) this);
   return;
}

Links &Links::operator =
//---------------------------------------------------------------------
//
//   This operator copies the links data from a tester/display object.
//
//---------------------------------------------------------------------
(
// Input parameter
   const Loop_Base &Tester   // Reference to tester/display object
)
{
// Local variables
      int i, j.               // Array indices

   if (Tester.Get_B_State() == Loop_Base::Binary) {
      for (i = 0; i < Num_Nodes; ++i)
         for (j = 0; j < Num_Nodes; ++j) {
            if (Tester.Get_South(i, j))
               South[i][j] = 1;
            else
               South[i][j] = 0;
            if (Tester.Get_East(i, j))
               East[i][j] = 1;
            else
               East[i][j] = 0;
         }
   } else {
      if (Tester.Get_T_State() == Loop_Base::Improved) {
         for (i = 0; i < Num_Nodes; ++i)
            for (j = 0; j < Num_Nodes; ++j) {
               if (Tester.Get_South(i, j) == -1 ||
                   Tester.Get_South(i, j) == -2)
                  South[i][j] = 1;
               else
                  South[i][j] = 0;
               if (Tester.Get_East(i, j) == -1 ||
                   Tester.Get_East(i, j) == -2)
                  East[i][j] = 1;
               else
                  East[i][j] = 0;
            }
      } else {
         cerr <<
```

```
   "\n<Links 03> Error. Attempt to copy from improper test data\n".
         exit(1).
      }
   }
   return *this.
}

Links &Links::operator --()
//---------------------------------------------------------------------
//
//   This operator initializes all links to zero.
//
//---------------------------------------------------------------------
{
// Local variables
      int i, j;                  // Counters

   for (i = 0; i < Num_Nodes; ++i)
      for (j = 0; j < Num_Nodes; ++j) {
         South[i][j] = 0;
         East[i][j] = 0;
      }
   return *this;
}

Links &Links::operator +=
//---------------------------------------------------------------------
//
//   This operator performs logical or with a Links type object.
//
//---------------------------------------------------------------------
(
// Input parameter
   Links &New_Links           // Pointer to links data
)
{
// Local variables
      int N;                  // Number of nodes minus 1
      int i, j;               // Array indices

   for (i = 0; i < Num_Nodes; ++i)
      for (j = 0; j < Num_Nodes; ++j) {
         if (New_Links.South[i][j])
            South[i][j] = 1;
         if (New_Links.East[i][j])
            East[i][j] = 1;
      }
   return *this;
}

Links &Links::operator -=
//---------------------------------------------------------------------
//
//   This operator clears a link if it is set in the input object.
//
//---------------------------------------------------------------------
(
// Input parameters
      Links &New_Links        // Pointer to links data
)
{
// Local variables
      int i, j;               // Array indices

   for (i = 0; i < Num_Nodes; ++i)
      for (j = 0; j < Num_Nodes; ++j) {
         if (New_Links.South[i][j])
            South[i][j] = 0;
         if (New_Links.East[i][j])
            East[i][j] = 0;
      }
   return *this;
}

ostream &operator <<
//---------------------------------------------------------------------
//
//   This operator prints the links object
//
//---------------------------------------------------------------------
```

```
{
// Input parameters
      ostream &A.              // Output stream
      Links &B                 // Object to be printed
}
{
   B.Print(A);
   return A;
}

void Links::XOR
//---------------------------------------------------------------------

//   This function performs a bitwise exclusive OR operation.

//---------------------------------------------------------------------

//   Input parameters
      const Links &A.          // Pointer to links data
      const Links &B           // Pointer to links data
}
{
// Local variables
      int i, j.                // Array indices

   for (i = 0; i < Num_Nodes; ++i)
      for (j = 0; j < Num_Nodes; ++j) {
         East[i][j] =  A.East[i][j] ^ B.East[i][j];
         South[i][j] =  A.South[i][j] ^ B.South[i][j];
      }
   return.
}
```

FUZZLINK.H

```
//---------------------------------------------------------------------
//
//   This is the file fuzzlink.h
//
//---------------------------------------------------------------------

// Interface dependencies ---------------------------------------------

#ifndef __CHAOS_H
#include "chaos.h"
#endif

#ifndef __LOOPS2_H
#include "loops2.h"
#endif

// End interface dependencies -----------------------------------------

#ifndef __FUZZLINK_H
#define __FUZZLINK_H

class FuzzLink   public Solution {
//---------------------------------------------------------------------
//
//   This class contains the layout solution as a preference order
//   scheme.
//
//   operators
//      FuzzLink(LoopBase*)
//      ostream << FuzzLink
//      istream >> FuzzLink
//      FuzzLink = FuzzLink
//      FuzzLink >> Loop_Base
//
//   functions
//      Set_Count()
//      Init(Chaos01)
//      Sort_Mem()
//      Sort_Ind()
//      Best_Alpha()
//      Remove_Redun(Chaos01)
//      Combine(FuzzLink, FuzzLink, Chaos01, double)
```

```
//      Mutate(FuzzLink, Chaos01, double)
//      Add_Link()
//      Sub_Link()
//      int Test()
//      int Modify()
//      New_Link_On(int, int, int)
//      New_Link_Off(int, int, int)
//
//   interface functions
//      int Get_South(int)
//      int Get_Row(int)
//      int Get_Col(int)
//      Alpha_State Get_A_State()
//      Put_A_State(Alpha_State)
//      Sort_State Get_S_State()
//      Put_S_State(Sort_State)
//
//-------------------------------------------------------------------
public:

  enum Alpha_State {          // State of threshold variable

     Feasible,                // Solution is feasible
     Not_Feasible,            // Solution is not feasible
     Alpha_Found,             // Threshold variable is set to alpha value
     Alpha_Set,               // Threshold variable has been set
     Not_Set                  // Threshold variable has not been set
  };

  enum Sort_State {           // Sort state of links

     Mem_Sort,                // Links have been sorted by preference
     Ind_Sort,                // Links have been sorted by index
     No_Sort                  // Links have not been sorted
  };

  struct FuzzLink_Rec {       // Record containing link data

     int South;               // Flag (1 = links is a south link)
     int Row;                 // Row position of link in Hanan grid
     int Col;                 // Column position of link in Hanan grid
     double Membership;       // Preference value between 0 and 1
     int Index;               // Row-column-east/south order scheme
  };

protected
   Loop_Base *Test_Obj;       // Pointer to tester/display object
   int Num_Nodes;             // The number of basic vertices
   int Num_Recs;              // The number of links in the Hanan grid
   FuzzLink_Rec *Link_Rec;    // Array of link records
   int Alpha;                 // The threshold variable
   Alpha_State A_State;       // State variable describing threshold variable
   Sort_State S_State;        // State variable describeing state of the
                              // array of link records

// Implementation functions
   void Print(ostream&) const;
                              // Prints out the object
   void Read(istream&);
                              // Reads data into object

public:
   FuzzLink(){}               // Vanilla constructor
   FuzzLink(Loop_Base*);      // Constructor
   ~FuzzLink();               // Destructor
   friend ostream &operator <<(ostream&, FuzzLink&);
                              // Extractor for output stream
   friend istream &operator >>(istream&, FuzzLink&);
                              // Insertor for input stream
   FuzzLink &operator =(const FuzzLink&);
                              // Copy operator
   void operator >>(Loop_Base&) const;
                              // Insertor for tester/display object

   void Set_Count();          // Sets the south, row, column and index values
   void Init(Chaos01&);       // Assigns random preference values
   void Sort_Mem();           // Sorts links on the basis of preference
   void Sort_Ind();           // Sorts links on the basis of index values
   void Best_Alpha();         // Set the threshold to the alpha value
   void Remove_Redun(Chaos01&);
```

```
                                // Removes all redundant links
    void Combine(FuzzLink&, FuzzLink&, Chaos01&, double);
                                // Recombines two solutions to create a new
                                // one
    void Mutate(FuzzLink&, Chaos01&, double);
                                // Random mutation function
    void Add_Link();            // Increases the threshold to add a link
    void Sub_Link();            // Decreases the threshold to remove a link
    int Test();                 // Tests feasibility of solution
    int Modify();               // Provides interactive modification of the
                                // object
    void New_Link_On(int, int, int);
                                // Removes the specified link from the layout
    void New_Link_Off(int, int, int);
                                // Adds the specified link to the layout

// Interface functions
    int Get_South(int i) const { return Link_Rec[i].South; }
                                // Gets south flag of specified link
    int Get_Row(int i) const { return Link_Rec[i].Row; }
                                // Gets row of specified link
    int Get_Col(int i) const { return Link_Rec[i].Col; }
                                // Gets column of specified link
    Alpha_State Get_A_State() const { return A_State; }
                                // Returns the state of the threshold variable
    void Put_A_State(Alpha_State A){ A_State = A; return; }
                                // Changes the state of the threshold variable
    Sort_State Get_S_State() const { return S_State; }
                                // Returns the state of the array of records
    void Put_S_State(Sort_State S){ S_State = S; return; }
                                // Changes the state of the array of records
};

#endif
```

FUZZLINK.CPP

```
//---------------------------------------------------------------------
//
//   This is the file fuzzlink.cpp
//
//---------------------------------------------------------------------

// Implementation dependencies ----------------------------------------

#ifndef _STDLIB_H_INCLUDED
#include <stdlib.h>
#endif

#ifndef __CUSTIO_H
#include "custio.h"
#endif

#ifndef __FUZZLINK_H
#include "fuzzlink.h"
#endif

// End implementation dependencies ------------------------------------

void FuzzLink::Print
//---------------------------------------------------------------------
//
//   This function prints out the array of link records.
//
//---------------------------------------------------------------------
(
// Input parameter
      ostream &Out              // Output stream
)
const
{
// Local variables
      int i;                    // Row counter
      int j;                    // Column counter
      int k;                    // Link type counter
      int l;                    // Record index
```

```
   Out << Num_Nodes << "\n";
   for (l = i = 0; i < Num_Nodes; ++i)
      for (j = 0 ; j < Num_Nodes; ++j)
         for (k = 0; k < 2; ++k, ++l) {
            Out << Link_Rec[l].South << "\n";
            Out << Link_Rec[l].Row << "\n";
            Out << Link_Rec[l].Col << "\n";
            Out << Link_Rec[l].Membership << "\n";
            Out << Link_Rec[l].Index << "\n";
         }
   return;
}

void FuzzLink::Read
(
   istream &In
)
{
// Local variables
      int New_Num;              // Temporary for new number of nodes
      int i;                    // Row counter
      int j;                    // Column counter
      int k;                    // Link type counter
      int l;                    // Record index

   In >> New_Num;
   if (New_Num == Num_Nodes) {
      for (l = i = 0; i < Num_Nodes; ++i)
         for (j = 0 ; j < Num_Nodes; ++j)
            for (k = 0; k < 2; ++k, ++l) {
               In >> Link_Rec[l].South;
               In >> Link_Rec[l].Row;
               In >> Link_Rec[l].Col;
               In >> Link_Rec[l].Membership;
               In >> Link_Rec[l].Index;
            }
      S_State = No_Sort;
      A_State = Not_Set;
      E_State = Not_Evaluated;
   } else {
      cerr <<
"\n<FuzzLink 01> Error  Size of file to be read does not match\n";
      Pause_Key();
      exit(1);
   }
   return;
}

FuzzLink::FuzzLink
//-------------------------------------------------------------------------
//
//   This constructor allocates space for the array of link records
//   and sets state variables
//
//-------------------------------------------------------------------------
(
// Input parameters
      Loop_Base *New_Tester  // Pointer to tester/display object
)
  Solution()
{
   Test_Obj = New_Tester;
   Num_Nodes = Test_Obj->Get_Num();
   Num_Recs = 2 * Num_Nodes * Num_Nodes;
   Link_Rec = new FuzzLink_Rec[Num_Recs];

   // Set state variables
   S_State = No_Sort;
   A_State = Not_Set;
   E_State = Not_Evaluated;
   Cost = 0.0;
}

FuzzLink::~FuzzLink()
//-------------------------------------------------------------------------
//
//   This frees space allocated to the array of link records.
//
//-------------------------------------------------------------------------
{
```

```
   delete Link_Rec;
}

ostream &operator <<
//---------------------------------------------------------------------
//
//   This extractor prints out the FuzzLink object to the output
//   stream
//
//---------------------------------------------------------------------
(
// Input paramters
      ostream &Out.          // Output stream
      FuzzLink &A            // Output object
)
{
   A.Print(Out);
   return Out;
}

istream &operator >>
//---------------------------------------------------------------------
//
//   This insertor reads in the FuzzLink object from the input
//   stream.
//
//---------------------------------------------------------------------
(
// Input paramters
      istream &In.           // Output stream
      FuzzLink &A            // Output object
)
{
   A.Read(In);
   return In;
}

FuzzLink &FuzzLink::operator =
//---------------------------------------------------------------------
//
//   This operator copies data from one FuzzLink object to another.
//
//---------------------------------------------------------------------
(
// Input parameters
   const FuzzLink &Source  // Source of data
)
{
   // Loacal variables
      int i.                  // Record index

   // Copy all records
   for (i = 0; i < Num_Recs; ++i)
      Link_Rec[i] = Source.Link_Rec[i];

   // Set state variables of source object
   Alpha = Source.Alpha;
   A_State = Source.A_State;
   S_State = Source.S_State;
   E_State = Source.E_State;
   Cost = Source.Cost;
   return *this;
}

void FuzzLink::operator >>
//---------------------------------------------------------------------
//
//   This operator copies data from this object to the
//   tester/display object.
//
//---------------------------------------------------------------------
(
// Output parameters
      Loop_Base &Tester       // Destination for FuzzLink data
)
const
{
// Local variables
      int N;                  // Temporary (Number of basic vertices - 1)
      int i;                  // Record index
```

```
   if (A_State != Not_Set) {

      // Initialize the tester/display object if required
      if (Tester.Get_T_State() != Loop_Base::Clear)
         Tester.Init();

      // Set all links in tester/display object for records above
      // the threshold value
      for (i = 0; i < Alpha; ++i)
         if (Link_Rec[i].South)
            Tester.Set_South(Link_Rec[i].Row, Link_Rec[i].Col);
         else
            Tester.Set_East(Link_Rec[i].Row, Link_Rec[i].Col);

      // Clear the south row and east column
      N = Num_Nodes - 1;
      for (i = 0; i < Num_Nodes; ++i) {
         Tester.Clear_South(N, i);
         Tester.Clear_East(i, N);
      }

      // Set feasibility state of tester/display object
      if (A_State == Alpha_Found || A_State == Feasible)
         Tester.Put_T_State(Loop_Base::Feasible);
      else if (A_State == Not_Feasible)
         Tester.Put_T_State(Loop_Base::Not_Feasible);
      else
         Tester.Put_T_State(Loop_Base::Unknown);

      // Set tester/display object to binary state
      Tester.Put_B_State(Loop_Base::Binary);
      if (E_State == Evaluated) {
         Tester.Put_Cost(Cost);
         Tester.Put_E_State(Loop_Base::Evaluated);
      } else
         Tester.Put_E_State(Loop_Base::Not_Evaluated);
      Tester.Put_Sol((Solution*) this);
   } else {
      cerr <<
"\n<FuzzLink.02> Error: Uninitialized object copied to Loop_Mod\n";
      Pause_Key();
      exit(1);
   }
   return;
}

void FuzzLink::Set_Count()
//---------------------------------------------------------------------
//
//   This function sets the south, row, column and index values of
//   each link record.  Preference (Membership) values are initialized
//   at 0.
//
//---------------------------------------------------------------------
{
// Local variables
      int i;                  // Row counter
      int j;                  // Column counter
      int k;                  // Link type counter
      int l;                  // Record index

   // Set initial values for each record in the array
   for (l = i = 0; i < Num_Nodes; ++i)
      for (j = 0 ; j < Num_Nodes; ++j)
         for (k = 0; k < 2; ++k, ++l) {
            Link_Rec[l].South = k;
            Link_Rec[l].Row = i;
            Link_Rec[l].Col = j;
            Link_Rec[l].Membership = 0.0;
            Link_Rec[l].Index = l;
         }

   // Set state variables
   S_State = Ind_Sort;
   A_State = Not_Set;
   return;
}

void FuzzLink::Init
```

```
//---------------------------------------------------------------------
//
//   This function assigns random preference (Membership) values to
//   each link record. The link records are sorted on the basis of
//   these preference values.
//
//---------------------------------------------------------------------
{
// Input parameters
      Chaos01 &Member         // Source of random numbers
)
{
// Local variables
      int i;                  // Record index

   // Set threshold state variable
   A_State = Not_Set;

   // Assign a random preference value to each link
   for (i = 0; i < Num_Recs; ++i) {
      Link_Rec[i].Membership = ++Member;
   }

   // Sort all links on the basis of preference
   Sort_Mem();
   return;
}

void FuzzLink  Sort_Mem()
//---------------------------------------------------------------------
//
//   This function sorts the link records in descending order of
//   records in descending order of preference (Membership) value.
//
//---------------------------------------------------------------------
{
// Local variables
      FuzzLink_Rec Temp;      // Temporary link record
      int N;                  // Temporary (Number of records - 1)
      int Change;             // Flag (1 = change of position took place)
      int i;                  // Index
      int j;                  // Index

   if (S_State != Mem_Sort) {
      N = Num_Recs - 1;
      Change = 1;
      for (i = 0; i < N && Change; ++i) {
         Change = 0;
         for (j = N; j > i; --j)
            if (Link_Rec[j - 1].Membership < Link_Rec[j].Membership) {
               Change = 1;
               Temp = Link_Rec[j];
               Link_Rec[j] = Link_Rec[j - 1];
               Link_Rec[j - 1] = Temp;
            }
      }

      // Set sort state
      S_State = Mem_Sort;
   }

   // Set threshold state variable
   A_State = Not_Set;
   return;
}

void FuzzLink  Sort_Ind()
//---------------------------------------------------------------------
//
//   This function sorts the link records in ascending order of
//   records in descending order of index value.
//
//---------------------------------------------------------------------
{
// Local variables
      FuzzLink_Rec Temp;      // Temporary link record
      int N;                  // Temporary (Number of records - 1)
      int Change;             // Flag (1 = change of position took place)
      int i;                  // Index
      int j;                  // Index
```

```
   if (S_State != Ind_Sort) {
      N = Num_Recs - 1;
      Change = 1;
      for (i = 0; i < N && Change; ++i) {
         Change = 0;
         for (j = N; j > i; --j)
            if (Link_Rec[j - 1].Index > Link_Rec[j].Index) {
               Change = 1;
               Temp = Link_Rec[j];
               Link_Rec[j] = Link_Rec[j - 1];
               Link_Rec[j - 1] = Temp;
            }
      }

      // Set sort state
      S_State = Ind_Sort;
   }

   // Set threshold state variable
   A_State = Not_Set;
   return;
}

void FuzzLink::Best_Alpha()
//---------------------------------------------------------------------
//
//   This function sets the threshold variable to the alpha value
//   the threshold value that results in the least length feasible
//   solution for the preference order
//
//---------------------------------------------------------------------
{
// Local variables
      int High_Alpha;         // Upper bound on alpha value
      int Low_Alpha;          // Lower bound on alpha value

   // Check to see if threshold has been set already
   if (A_State != Alpha_Found) {

      // Check to see if object has been sorted by preference
      if (S_State != Mem_Sort)
         Sort_Mem();

      // Set upper and lower bounds and set threshold to midpoint
      High_Alpha = Num_Recs;
      Alpha = Num_Recs / 2;
      Low_Alpha = 0;

      // Test for convergence
      while (Alpha != Low_Alpha && Alpha != High_Alpha) {

         // Set state variables prior to test
         A_State = Alpha_Set;
         E_State = Not_Evaluated;

         Test_Obj->Init();
         operator >>(*Test_Obj);

         // If result is feasible current threshold becomes
         // higher bound
         if (Test_Obj->Test())
            High_Alpha = Alpha;

         // Otherwise current threshold becomes lower bound
         else
            Low_Alpha = Alpha;

         // Set new threshold value
         Alpha = (Low_Alpha + High_Alpha) / 2;

      }

      // Set threshold to alpha value and set state variable
      Alpha = High_Alpha;
      A_State = Alpha_Found;

      // Restore tester/display object to alpha solution
      Test_Obj->Init();
      operator >>(*Test_Obj);
```

```
   }
   return.
}

void FuzzLink. Remove_Redun
//-------------------------------------------------------------------------
//
//   This function identifies and removes all branched components
//   from a solution. The branched components are identified in row
//   column order and placed at the end of the array  They are then
//   assigned random preference values that will keep them at the end
//   of the array  The array is then sorted according to preference
//
//-------------------------------------------------------------------------
(
// Input parameters
     Chaos01 &Bit_Source     // Source of random numbers
)
{
// Local variables
     int Mode:               // Temporary storage of display mode
     int Found:              // Flag (1 = branched components)
     int N:                  // Temporary (Number of basic vertices -1
                             //            or number of records -1)
     int i:                  // Row index
     int j.                  // Column index
     int First_Row:          // Row of first branched component found
     int First_Col:          // Column of first branched component found
     int First_South:        // South flag of first branched component found
     double Max_Membership.  // Highest preference value of removed branched
                             // components
     int Sort_Index:         // Position in array of first branched
                             // component removed
     int Change.             // Flag (1 = sort has changed position of
                             // record
     FuzzLink_Rec Temp.      // Temporary record used in sort

   N = Num_Nodes - 1:

   // If the current solution has not been set to a feasible
   // threshold value set it to the alpha value
   if (A_State != Feasible &&
       A_State != Alpha_Found) {
      Mode = Test_Obj->Get_Mode().
      Test_Obj->Set_Mode(0):
      Best_Alpha().
      Test_Obj->Evaluate():
      Test_Obj->Set_Mode(Mode).
   }

   // If the tester/display object is not displaying this object
   // copy this object to the tester/display
   if (Test_Obj->Get_Sol() != this ||
       Test_Obj->Get_B_State() != Loop_Base::Binary) {
      Test_Obj->Init().
      operator >>(*Test_Obj).
   }

   // Set display mode to 0 and improve the solution
   Mode = Test_Obj->Get_Mode():
   Test_Obj->Set_Mode(0).
   Test_Obj->Improve():
   Test_Obj->Set_Mode(Mode).

   // Identify and remove all branched east links
   Found = 0.
   for (i = 0: i < Num_Nodes: ++ i)
      for (j = 0: j < N: ++j)
         if (Test_Obj->Get_East(i. j] == 1) {
            Test_Obj->Clear_East(i. j]:
            if (Test_Obj->Get_E_State() == Loop_Base::Evaluated)
               Test_Obj->Put_Cost(Test_Obj->Get_Cost()
                                  - Test_Obj->Get_Prob()->Get_DX(j)):
            New_Link_Off(i. j. 0).

            // Keep a record if the first branched component found
            if (!Found) {
               First_Row = i:
               First_Col = j:
               First_South = 0:
```

```
                Found = 1;
             }
          }

    // Identify and remove all branched south links
    for (i = 0; i < N; ++i)
       for (j = 0; j < Num_Nodes; ++j)
          if (Test_Obj->Get_South(i, j) == 1) {
             Test_Obj->Clear_South(i, j);
             if (Test_Obj->Get_E_State() == Loop_Base::Evaluated)
                Test_Obj->Put_Cost(Test_Obj->Get_Cost()
                                   - Test_Obj->Get_Prob()->Get_DY(i));
             New_Link_Off(i, j, 1);

             // Keep a record if the first branched component found
             if (!Found) {
                First_Row = i;
                First_Col = j;
                First_South = 1;
                Found = 1;
             }
          }
    A_State = Alpha_Found;
    if (Found) {

       // Find the first branched component to be placed at the
       // bottom of the array
       for (Found = 0, i = Num_Recs - 1; i >= 0 && !Found; --i) {
          if (First_Row == Link_Rec[i].Row &&
              First_Col == Link_Rec[i].Col &&
              First_South == Link_Rec[i].South) {
             Max_Membership = Link_Rec[i].Membership;
             Sort_Index = i;
             Found = 1;
          }
       }
       if (!Found) {
          cerr <<
"\n<FuzzLink 03> Error  Sort point not found\n";
          Pause_Key();
          exit(1);
       }

       // Assign random preference values less than the maximum
       // branched preference value
       for (i = Sort_Index; i < Num_Recs; ++i)
          Link_Rec[i].Membership = Max_Membership * (++Bit_Source);
       N = Num_Recs - 1;
       Change = 1;

       // Sort the branched components according to their
       // newly assigned preference values
       for (i = Sort_Index; i < N && Change; ++i) {
          Change = 0;
          for (j = N; j > i; --j)
             if (Link_Rec[j - 1].Membership < Link_Rec[j].Membership) {
                Change = 1;
                Temp = Link_Rec[j];
                Link_Rec[j] = Link_Rec[j - 1];
                Link_Rec[j - 1] = Temp;
             }
       }
    }

    // Update the tester/display object
    Test_Obj->Init();
    operator >>(*Test_Obj);
    return;
}

void FuzzLink::Combine
//---------------------------------------------------------------------
//
//   This function performs the recombination operation used by the
//   evolution program.
//
//---------------------------------------------------------------------
(
// Input parameters
      FuzzLink &Source1,     // Parent solution
```

```
      FuzzLink &Source2.      // Parent solution
      Chaos01 &Member.        // Source of random numbers
      double Rate             // Mutation rate
)
{
// Local variables
      int i.                  // Record index

   // Make sure that the two source objects and this object
   // are sorted according to index value
   if (Source1.S_State != Ind_Sort)
      Source1.Sort_Ind().
   if (Source2.S_State != Ind_Sort)
      Source2.Sort_Ind().
   if (S_State != Ind_Sort)
      Sort_Ind().

   // For every link combine or mutate the preference values
   for (i = 0. i < Num_Recs. ++i)

      // If the random value is below the rate value
      // mutate the link
      if (++Member < Rate)
         Link_Rec[i].Membership = ++Member.

      // Otherwise combine the link values
      else
         Link_Rec[i].Membership = (Source1.Link_Rec[i].Membership +
                                   Source2.Link_Rec[i].Membership) / 2.0.

   // Set state variables and cost
   S_State = No_Sort.
   A_State = Not_Set.
   E_State = Not_Evaluated.
   Cost = 0.0.
   return.
}

void FuzzLink::Mutate
//---------------------------------------------------------------------
//
//   This function mutates the solution by changing the preference
//   value for a small number of links.
//
//---------------------------------------------------------------------
(
// Input parameters
      FuzzLink &Source.       // Source links data
      Chaos01 &Member.        // Source of random numbers
      double Rate             // Mutation rate
)
{
// Local variables
      int i.                  // Record index

   for (i = 0. i < Num_Recs. ++i)

      // If the random value is below the rate value
      // mutate the link
      if (++Member < Rate)
         Link_Rec[i].Membership = ++Member.

      // Otherwise combine the link values
      else
         Link_Rec[i] = Source.Link_Rec[i].

   // Set state variables and cost
   S_State = No_Sort.
   A_State = Not_Set.
   E_State = Not_Evaluated.
   Cost = 0.0.
   return.
}

void FuzzLink::Add_Link()
//---------------------------------------------------------------------
//
//   This function increases the threshold variable to include one
//   more link. The display in the tester/display object is updated
//   to include this change.
```

```
//
//-------------------------------------------------------------------------
{
// Local variables
      double D_Cost.          // Incremental increase in cost
      int Found:              // Flag (1= next link has been found)
      int South:              // Flag (1 = new link is a south link)
      int i:                  // Row position of new link
      int j;                  // Column position of new link

   // Check to see that threshold is set
   if (A_State != Not_Set) {

      // Check to see records are sorted according to preference
      if (S_State == Mem_Sort) {

         // Check to see that tester/display object points to
         // this object
         if (Test_Obj->Get_Sol() != this ||
             Test_Obj->Get_B_State() != Loop_Base::Binary)
            operator >>(*Test_Obj);
         if (Alpha < Num_Recs) {
            Found = 0;

            // Keep increasing alpha until a link that is not
            // in the south row or east column is added
            while (!Found && Alpha < Num_Recs) {
               i = Link_Rec[Alpha].Row;
               j = Link_Rec[Alpha].Col;
               South = Link_Rec[Alpha].South;
               if (South) {
                  if (i != Num_Nodes - 1) {
                     Found = 1;

                     // Increase the cost of the object
                     D_Cost = Test_Obj->Get_Prob()->Get_DY(i);
                     Cost += D_Cost;

                     // Update the tester/display object
                     Test_Obj->Put_T_State(Loop_Base::Unknown);
                     if (Test_Obj->Get_E_State() == Loop_Base::Evaluated)
                        Test_Obj->Put_Cost(Test_Obj->Get_Cost() + D_Cost);
                     Test_Obj->Set_Position(i, j);
                     Test_Obj->Set_South(i, j);
                     Test_Obj->Toggle_Draw_South();
                  }
               } else {
                  if (j != Num_Nodes - 1) {
                     Found = 1;

                     // Increase the cost of the object
                     D_Cost = Test_Obj->Get_Prob()->Get_DX(j);
                     Cost += D_Cost;

                     // Update the tester/display object
                     Test_Obj->Put_T_State(Loop_Base::Unknown);
                     if (Test_Obj->Get_E_State() == Loop_Base::Evaluated)
                        Test_Obj->Put_Cost(Test_Obj->Get_Cost() + D_Cost);
                     Test_Obj->Set_Position(i, j);
                     Test_Obj->Set_East(i,j);
                     Test_Obj->Toggle_Draw_East();
                  }
               }
               Alpha++ ;
            }
         }
         A_State = Alpha_Set;
      } else {
         cerr <<
"\n<FuzzLink 04> Error: Increased alpha of unsorted object\n";
         Pause_Key();
         exit(1);
      }
   } else {
      cerr <<
"\n<FuzzLink 05> Error: Uninitialized alpha value increased\n";
      Pause_Key();
      exit(1);
   }
   return;
```

```
}

void FuzzLink  Sub_Link()
//---------------------------------------------------------------------
//
//   This function decreases the threshold variable to remove one
//   link. The display in the tester/display object is updated
//   to remove this link
//
//---------------------------------------------------------------------
{
// Local variables
      double D_Cost;          // Decrease in cost
      int Found;              // Flag (1= next link has been found)
      int South;              // Flag (1 = new link is a south link)
      int i;                  // Row position of new link
      int j;                  // Column position of new link

   // Check to see that threshold is set
   if (A_State != Not_Set) {

      // Check to see records are sorted according to preference
      if (S_State == Mem_Sort) {

         // Check to see that tester/display object points to
         // this object
         if (Test_Obj->Get_B_State() != Loop_Base::Binary ||
             Test_Obj->Get_Sol() != this)
            operator >>(*Test_Obj);
         if (Alpha > 0) {
            Found = 0;

            // Keep decreasing alpha until a link that is not
            // in the south row or east column is removed
            while ((!Found) && (Alpha > 0)) {
               --Alpha;
               i = Link_Rec[Alpha].Row;
               j = Link_Rec[Alpha].Col;
               South = Link_Rec[Alpha].South;
               if (South) {
                  if (i != Num_Nodes - 1) {
                     Found = 1;

                     // Decrease the cost of the object
                     D_Cost = Test_Obj->Get_Prob()->Get_DY(i);
                     Cost -= D_Cost;

                     // Update the tester/display object
                     Test_Obj->Put_T_State(Loop_Base::Unknown);
                     if (Test_Obj->Get_E_State() == Loop_Base::Evaluated)
                        Test_Obj->Put_Cost(Test_Obj->Get_Cost() - D_Cost);
                     Test_Obj->Set_Position(i, j);
                     Test_Obj->Clear_South(i,j);
                     Test_Obj->Toggle_Draw_South();
                  }
               } else {
                  if (j != Num_Nodes - 1) {
                     Found = 1;

                     // Decrease the cost of the object
                     D_Cost = Test_Obj->Get_Prob()->Get_DX(j);
                     Cost -= D_Cost;

                     // Update the tester/display object
                     Test_Obj->Put_T_State(Loop_Base::Unknown);
                     if (Test_Obj->Get_E_State() == Loop_Base::Evaluated)
                        Test_Obj->Put_Cost(Test_Obj->Get_Cost() - D_Cost);
                     Test_Obj->Set_Position(i, j);
                     Test_Obj->Clear_East(i,j);
                     Test_Obj->Toggle_Draw_East();
                  }
               }
            }
         }
         A_State = Alpha_Set;
      } else {
         cerr <<
"\n<FuzzLink 06> Error: Decreased alpha of unsorted object\n";
         Pause_Key();
         exit(1);
```

```
      }
   } else {
      cerr <<
"\n<FuzzLink 07> Error  Uninitialized alpha value Decreased\n".
      Pause_Key();
      exit(1);
   }
   return;
}

int FuzzLink::Test()
//---------------------------------------------------------------------
//
//   This function returns a value of 1 if the object represents
//   a feasible solution or 0 if the solution is infeasible
//
//---------------------------------------------------------------------
{
// Local variables
      int FeasibleP;          // Flag (1 = solution is feasible)

   FeasibleP = 0;

   //If feasibility is unknown test it
   if (A_State != Feasible &&
       A_State != Not_Feasible &&
       A_State != Alpha_Found) {

      // Check to see if threshold has been set previously
      if (A_State != Not_Set) {

         // Set up for destructive testing
         Test_Obj->Init();
         operator >>(*Test_Obj);
         if (Test_Obj->Test()) {
            A_State = Feasible;
            FeasibleP = 1;
         } else
            A_State = Not_Feasible;

         // Correct for destructive testing
         Test_Obj->Init();
         operator >>(*Test_Obj);
      } else {
         cerr <<
"\n<FuzzLink 08> Error  Test of uninitialized object\n";
         Pause_Key();
         exit(1);
      }
   } else {
      if (A_State == Feasible || A_State == Alpha_Found)
         FeasibleP = 1;
   }
   return FeasibleP;
}

int FuzzLink::Modify()
//---------------------------------------------------------------------
//
//   This function provides interactive modification of the layout.
//   A cursor is displayed and moved about the layout using the
//   keys. If the enter key is pressed the link at the cursor
//   location is toggled and the contents of the tester/display
//   object and this object are modified to reflect the change. The
//   interactive loop is terminanted when the <insert> or <end>
//   key is pressed. The function returns 1 if the <insert> key is
//   pressed and 0 if the <end> key is pressed.
//
//---------------------------------------------------------------------
{
// Local variables
      unsigned short Choice; // Key pressed by user
      int Mod_Row;           // Row position of modification cursor
      int Mod_Col;           // Column position of modification cursor
      int South;             // Link type of cursor position
      int OnOff;             // Flag (1 = link is to be included
                             //       -1 = link is to be excluded)

   // Check to see that threshold has been set
   if (A_State != Not_Set) {
```

```
      // Check to see that object has been sorted on the basis
      // of preference
      if (S_State == Mem_Sort) {

         // Make sure tester/display is a copy of this object
         Test_Obj->Mod_Show();
         Choice = Pause_Key();

         // End loop if <end> or <insert> keys are pressed
         while ((Choice != 207) && (Choice != 210)) {

            // End loop if <return> or <end> or <insert> keys are pressed
            while ((Choice != 13) && (Choice != 207) && (Choice != 210)) {

               // Hide the cursor and move it if necessary
               Test_Obj->Mod_Hide();
               switch (Choice) {
                  // Cursor up key
                  case 200:
                     Test_Obj->Mod_Up();
                     break;
                  // Cursor left key
                  case 203:
                     Test_Obj->Mod_Left();
                     break;
                  // Cursor down key
                  case 208:
                     Test_Obj->Mod_Down();
                     break;
                  // Cursor right key
                  case 205:
                     Test_Obj->Mod_Right();
                     break;
               }

               // Show the cursor again
               Test_Obj->Mod_Show();
               Choice = Pause_Key();
            }

            // If the return key was pressed
            if (Choice == 13) {

               // Determine if the link is to be turned on or off
               OnOff = Test_Obj->Toggle_Link();

               // Get the location of the cursor
               Mod_Row = Test_Obj->Get_Mod_Row();
               Mod_Col = Test_Obj->Get_Mod_Col();
               South = Test_Obj->Get_Mod_Elem();
               switch (OnOff) {
                  case 1:
                     New_Link_On(Mod_Row, Mod_Col, South);
                     break;
                  case -1:
                     New_Link_Off(Mod_Row, Mod_Col, South);
               }
               Choice = Pause_Key();
            }
         }

         // Hide the cursor
         Test_Obj->Mod_Hide();

         // Set the threshold state variable
         A_State = Alpha_Set;
      } else {
         cerr <<
"\n<FuzzLink 09> Error: Modification of unsorted object\n";
         Pause_Key();
         exit(1);
      }
   } else {
      cerr <<
"\n<FuzzLink 10> Error: Modification before alpha set\n";
      Pause_Key();
      exit(1);
   }
   return (Choice == 210) ? 1 : 0;
```

```
}

void FuzzLink::New_Link_Off
//---------------------------------------------------------------------------
//
//   This function places the link record that matches the specified
//   row, column and link type at the end of the array.
//
//---------------------------------------------------------------------------
(
// Input parameters
      int Row,                  // Row position
      int Col,                  // Column position
      int South                 // Flag (1 = south link)
)
{
// Local variables
      FuzzLink_Rec Temp         // Temporary link record
      int Top,                  // Temporary (Number of records minus 1)
      int Found,                // Flag (1 = matching record is found)
      int i;                    // Record index

   // Check to see that the threshold is set
   if (A_State != Not_Set) {

      // Check to see that records are sorted by preference
      if (S_State == Mem_Sort) {

         // Find the matching record and store a copy of it
         for (Found = i = 0; i < Alpha && !Found; ++i)
            if (Link_Rec[i].Row == Row &&
                Link_Rec[i].Col == Col &&
                Link_Rec[i].South == South) {
               Temp.Row = Link_Rec[i].Row;
               Temp.Col = Link_Rec[i].Col;
               Temp.South = Link_Rec[i].South;
               Temp.Index = Link_Rec[i].Index;
               Found = 1;
            }

         // Move all records below the matching record up one position
         for (--i, Top = Num_Recs - 1; i < Top; ++i) {
            Link_Rec[i].Row = Link_Rec[i + 1].Row;
            Link_Rec[i].Col = Link_Rec[i + 1].Col;
            Link_Rec[i].South = Link_Rec[i + 1].South;
            Link_Rec[i].Index = Link_Rec[i + 1].Index;
         }

         // Place the matching record at the end of the array
         Link_Rec[i].Row = Temp.Row;
         Link_Rec[i].Col = Temp.Col;
         Link_Rec[i].South = Temp.South;
         Link_Rec[i].Index = Temp.Index;

         // Decrease the threshold
         --Alpha;

         // Set theshold state variable
         A_State = Alpha_Set;

         // Update the cost
         if (South)
            Cost -= Test_Obj->Get_Prob()->Get_DY(Row);
         else
            Cost -= Test_Obj->Get_Prob()->Get_DX(Col);
      } else {
         cerr <<
"\n<FuzzLink 11> Error: Link removed from unsorted object\n";
         Pause_Key();
         exit(1);
      }
   } else {
      cerr <<
"\n<FuzzLink 12> Error: Link removed before alpha set\n";
      Pause_Key();
      exit(1);
   }
   return;
}
```

```
void FuzzLink::New_Link_On
//---------------------------------------------------------------------------
//
//   This function places the link record that matches the specified
//   row, column and link type at the start of the array
//
//---------------------------------------------------------------------------
(
// Input parameters
      int Row,                  // Row position
      int Col,                  // Column position
      int South                 // Flag (1 = south link)
)
{
// Local variables
      FuzzLink_Rec Temp;        // Temporary link record
      int Found;                // Flag (1 = matching record is found)
      int i;                    // Record index

   // Check to see that the threshold is set
   if (A_State != Not_Set) {

      // Check to see that records are sorted by preference
      if (S_State == Mem_Sort) {

         // Find the matching record and store a copy of it
         for (Found = 0, i = Alpha; i < Num_Recs && !Found; ++i)
            if (Link_Rec[i].Row == Row &&
                Link_Rec[i].Col == Col &&
                Link_Rec[i].South == South) {
               Temp.Row = Link_Rec[i].Row;
               Temp.Col = Link_Rec[i].Col;
               Temp.South = Link_Rec[i].South;
               Temp.Index = Link_Rec[i].Index;
               Found = 1;
            }

         // Move all records above the matching record down one position
         for (--i; i > 0; --i) {
            Link_Rec[i].Row = Link_Rec[i - 1].Row;
            Link_Rec[i].Col = Link_Rec[i - 1].Col;
            Link_Rec[i].South = Link_Rec[i - 1].South;
            Link_Rec[i].Index = Link_Rec[i - 1].Index;
         }

         // Place the matching record at the end of the array
         Link_Rec[i].Row = Temp.Row;
         Link_Rec[i].Col = Temp.Col;
         Link_Rec[i].South = Temp.South;
         Link_Rec[i].Index = Temp.Index;

         // Increase the threshold
         ++Alpha;

         // Set theshold state variable
         A_State = Alpha_Set;

         // Update the cost
         if (South)
            Cost += Test_Obj->Get_Prob()->Get_DY(Row);
         else
            Cost += Test_Obj->Get_Prob()->Get_DX(Col);
      } else {
         cerr <<
"\n<FuzzLink 13> Error: Link added to unsorted object\n";
         Pause_Key();
         exit(1);
      }
   } else {
      cerr <<
"\n<FuzzLink 12> Error: Link added before alpha set\n";
      Pause_Key();
      exit(1);
   }
   return;
}
```

EPCLASS.H

```
//-------------------------------------------------------------------
//
//   This is the file epclass.h
//
//-------------------------------------------------------------------

// Interface dependencies ---------------------------------------

#ifndef _FSTREAM_H_INCLUDED
#include <fstream.h>
#endif

#ifndef __LOOPS3_H
#include "loops3.h"
#endif

#ifndef __FUZZLINK_H
#include "fuzzlink.h"
#endif

// End interface dependencies -----------------------------------

#ifndef __EPCLASS_H
#define __EPCLASS_H

class E_P_Class {
//-------------------------------------------------------------------
//
//   This class performs an evolution program using a recombination
//   operator based on averaging preference values.
//
//   operators
//      E_P_Class(int, double, int, char*, char*, char*, int  int, int)
//
//   functions
//      Run_GA()
//
//-------------------------------------------------------------------
protected
   int Pop_Size;              // Population size
   double Rate;               // Mutation rate
   int Iterations;            // Maximum number of iterations
   char *Prob_File;           // Nodes data file
   char *Seed_File;           // Seed file for random numbers
   char *Stat_File;           // File to collect statistics
   int Best_Only;             // Flag (1 = best solution is displayed only)
   int Disp;                  // Display mode
   int Block;                 // Flag (1 = improve step not displayed)
   int Key_Pause;             // Flag (1 = display pauses for key)
   Nodes2 *Problem;           // Pointer to nodes data
   Loop_Mod *Tester;          // Pointer to tester/display object
   Chaos01 *Bits;             // Pointer random number source
   FuzzLink **Old_Pop;        // Old population - array of solutions
   FuzzLink **New_Pop;        // New population - array of solutions
   double *Wheel;             // Roulette wheel
   double Best_Cost;          // Cost of best solution in a generation
   double Worst_Cost;         // Cost of wost solution in a generation
   double Best_Sol;           // Cost of best solution found
   ofstream *Out;             // Output stream for statistics file

// Implementation function
   void Start_Pop();          // Creates starting population
   void Set_Wheel();          // Sets up roulette wheel
   int Select_Parent();       // Selects a parent solution
   int New_Generation();      // Creates a new generation of solutions

public
   E_P_Class(){}              // Vanilla constructor
   E_P_Class(int, double, int, char*, char*, char*,
             int = 1, int = 1, int = 1, int = 0);
                              // Actual constructor
   virtual ~E_P_Class();      // Destructor
   void Run_GA();             // Executes evolution program
};

#endif
```

EPCLASS.CPP

```
//---------------------------------------------------------------------
//
//   This is the file epclass.cpp
//
//---------------------------------------------------------------------

// Implementation dependencies ----------------------------------------

#ifndef __EPCLASS_H
#include "epclass.h"
#endif

// End implementation dependencies ------------------------------------

void E_P_Class::Start_Pop()
//---------------------------------------------------------------------
//
//   This function creates the starting population.
//
//---------------------------------------------------------------------
{
// Local variables
      double Average.          // Average cost of solution
      int Mode.                // Temporary storage for display mode
      int i.                   // Index for old population

   // Generate starting population
   Best_Cost = Worst_Cost = Best_Sol = -1.0;
   Average = 0.0;
   for (i = 0; i < Pop_Size; ++i) {

      // Set threshold of new solution
      Mode = Tester->Get_Mode();
      Tester->Set_Mode(0);
      Old_Pop[i]->Init(*Bits);
      Old_Pop[i]->Best_Alpha();
      Old_Pop[i]->Remove_Redun(*Bits);
      Tester->Set_Mode(Mode);

      // Evaluate new solution
      *Old_Pop[i] >> *Tester
      Tester->Evaluate();

      // Update statistics
      if (Worst_Cost == -1.0 || Worst_Cost < Old_Pop[i]->Get_Cost())
         Worst_Cost = Old_Pop[i]->Get_Cost();
      if (Best_Cost == -1.0 || Best_Cost > Old_Pop[i]->Get_Cost())
         Best_Cost = Old_Pop[i]->Get_Cost();
      if (Best_Sol == -1.0 || Best_Sol > Old_Pop[i]->Get_Cost()) {
         Best_Sol = Old_Pop[i]->Get_Cost();
         if (Best_Only)
            Tester->Display();
      }
      Average += Old_Pop[i]->Get_Cost();
      if (!Best_Only)
         Tester->Display();
   }

   // Write statistics
   Average = Average / Pop_Size;
   *Out << Best_Cost << " " << Worst_Cost << " " << Average << "\n";
   return;
}

void E_P_Class::Set_Wheel()
//---------------------------------------------------------------------
//
//   This function sets up the roulette wheel for the selection of
//   parent solutions. Wheel is based on the solutions in the array
//   "Old_Pop".
//
//---------------------------------------------------------------------
{
   // Local variables
```

```
      double Top;              // Datum from which fitness is calculated
      int i;                   // Index for old population

   // Set up roulette wheel
   Top = Worst_Cost + 0.2 * (Worst_Cost - Best_Cost);
   Wheel[0] = Top - Old_Pop[0]->Get_Cost();
   for (i = 1; i < Pop_Size; ++i)
      Wheel[i] = Wheel[i - 1] + Top - Old_Pop[i]->Get_Cost();
   return;
}

int E_P_Class::Select_Parent()
//---------------------------------------------------------------------
//
//   This function returns the index of the selected parent
//   solution based on the roulette wheel created by the function
//   "Set_Wheel()"
//
//---------------------------------------------------------------------
{
// Local variables
      double Select;           // Selected position in roulette wheel
      int Parent;              // Selected parent solution

   // Select parent 1 if population has not converged
   if (Wheel[Pop_Size-1] > 0.0) {
      Select = Wheel[Pop_Size - 1] * ++*Bits;
      for (Parent = Pop_Size - 1; Parent >= 0 && Select < Wheel[Parent];
           --Parent);
      ++Parent;
   }

   // Ensure selected parent is in range
   if (Parent < 0 || Parent >= Pop_Size)
      Parent = 0;
   return Parent;
}

int E_P_Class::New_Generation()
//---------------------------------------------------------------------
//
//   This function creates the next population of solutions  The
//   function returns 1 if the population has converged to solutions
//   of the same cost, otherwise a zero is returned
//
//---------------------------------------------------------------------
{
// Local variables
      double Average;          // Average cost of solution
      int Parent1;             // Index to parent solution
      int Parent2;             // Index to parent solution
      int Mode;                // Temporary storage of display mode
      int i;                   // Index for new population


   // Set up the roulette wheel and initialize statistics
   Set_Wheel();
   Best_Cost = Worst_Cost = -1.0;
   Average = 0.0;
   for (i = 0; i < Pop_Size; ++i) {

      // Select parents
      Parent1 = Select_Parent();
      Parent2 = Select_Parent();

      // Combine parents for each member of new population
      Old_Pop[Parent1]->Sort_Ind();
      Old_Pop[Parent2]->Sort_Ind();
      New_Pop[i]->Sort_Ind();
      New_Pop[i]->Combine(*Old_Pop[Parent1], *Old_Pop[Parent2], *Bits, Rate);

      // Set threshold of new solution
      Mode = Tester->Get_Mode();
      Tester->Set_Mode(0);
      New_Pop[i]->Best_Alpha();
      New_Pop[i]->Remove_Redun(*Bits);
      Tester->Set_Mode(Mode);

      // Evaluate new solution
      *New_Pop[i] >> *Tester;
```

```
      Tester->Evaluate();

      // Update statistics
      if (Worst_Cost == -1.0 || Worst_Cost < New_Pop[i]->Get_Cost())
         Worst_Cost = New_Pop[i]->Get_Cost();
      if (Best_Cost == -1.0 || Best_Cost > New_Pop[i]->Get_Cost())
         Best_Cost = New_Pop[i]->Get_Cost();
      if (Best_Sol == -1.0 || Best_Sol > New_Pop[i]->Get_Cost()) {
         Best_Sol = New_Pop[i]->Get_Cost();
         if (Best_Only)
            Tester->Display();
      }
      Average += New_Pop[i]->Get_Cost();
      if (!Best_Only)
         Tester->Display();
   }

   // Write statistics for generation
   Average = Average / Pop_Size;
   *Out << Best_Cost << " " << Worst_Cost << " " << Average << "\n"

   // Return 1 if population has converged
   return (Best_Cost == Worst_Cost) ? 1 : 0;
}

E_P_Class::E_P_Class
//---------------------------------------------------------------------------
//
//   This constructor allocates space for the object.
//
//---------------------------------------------------------------------------
(
   // Input parameters
      int New_Pop_Size,       // Size of population
      double New_Rate,        // Mutation rate
      int New_Iterations,     // Maximum number of iterations
      char *New_Prob_File,    // Problem data file
      char *New_Seed_File,    // Seed file for random numbers
      char *New_Stat_File,    // File to collect statistics
      int New_Best_Only,      // Flag (1 = best solution is displayed only)
      int New_Disp,           // Display mode
      int New_Block,          // Flag (1 = improvement step not displayed)
      int New_Key_Pause       // Flag (1 = displays pause for key)
)
{
   // Local variables
      int i;                  // Index for populations

   // Set parameters
   Pop_Size = New_Pop_Size;
   Rate = New_Rate;
   Iterations = New_Iterations;
   Prob_File = New_Prob_File;
   Seed_File = New_Seed_File;
   Stat_File = New_Stat_File;
   Best_Only = New_Best_Only;
   Disp = New_Disp;
   Block = New_Block;
   Key_Pause = New_Key_Pause;

   // Allocate and initialize object
   Problem = new Nodes2(Prob_File);
   Tester = new Loop_Mod(*Problem, Disp, Block, Key_Pause);
   Bits = new Chaos01(Seed_File);
   Old_Pop = new FuzzLink*[Pop_Size];
   New_Pop = new FuzzLink*[Pop_Size];
   for (i = 0; i < Pop_Size; ++i) {
      Old_Pop[i] = new FuzzLink(Tester);
      Old_Pop[i]->Set_Count();
      New_Pop[i] = new FuzzLink(Tester);
      New_Pop[i]->Set_Count();
   }
   Wheel = new double[Pop_Size];
   Out = new ofstream(Stat_File);
}

E_P_Class::~E_P_Class()
//---------------------------------------------------------------------------
//
//   This destructor frees allocated space for the object.
```

```
//
//-------------------------------------------------------------------------
{
// Local variables
      int i.                  // Index for populations

   Out->close();
   delete Out;
   delete Wheel;
   for (i = Pop_Size - 1; i >= 0; --i) {
      delete New_Pop[i];
      delete Old_Pop[i];
   }
   delete New_Pop;
   delete Old_Pop;
   delete Bits;
   delete Tester;
   delete Problem;
}

void E_P_Class::Run_GA()
//-------------------------------------------------------------------------
//
//   This function is the top level function that runs the evolution
//   program.
//
//-------------------------------------------------------------------------
{
      int Converged;          // Flag (1 = population has converged)
      int i;                  // Counter that counts number of iterations
      int j;                  // Index for populations

   // Set up starting population
   Start_Pop();

   // Generate new populations of solutions
   for (i = Converged = 0; i < Iterations && !Converged; ++i) {

      // Create next generation
      Converged = New_Generation();

      // Copy new population to old
      for (j = 0; j < Pop_Size; ++j)
         *Old_Pop[j] = *New_Pop[j];
   }
   return;
}
```

ESCLASS.H

```
//-------------------------------------------------------------------------
//
//   This is the file esclass.h
//
//-------------------------------------------------------------------------

// Interface dependencies ---------------------------------------------

#ifndef _FSTREAM_H_INCLUDED
#include <fstream.h>
#endif

#ifndef __LOOPS3_H
#include "loops3.h"
#endif

#ifndef __FUZZLINK_H
#include "fuzzlink.h"
#endif

// End interface dependencies -----------------------------------------

#ifndef __ESCLASS_H
#define __ESCLASS_H

class E_S_Class {
//-------------------------------------------------------------------------
```

```
//
//   This class performs an evolution program based on evolution
//   strategy.
//
//   operators
//      E_S_Class(double, int, char*, char*, char*, int, int, int)
//
//   functions
//      Run_ES()
//
//-------------------------------------------------------------------
protected:
   double Rate;                  // Mutation rate
   int Iterations;               // Maximum number of iterations
   char *Prob_File;              // Nodes data file
   char *Seed_File;              // Seed file for random numbers
   char *Stat_File;              // File to collect statistics
   int Disp;                     // Display mode
   int Block;                    // Flag (1 = improve step not displayed)
   int Key_Pause;                // Flag (1 = display pauses for key)
   Nodes2 *Problem;              // Pointer to nodes data
   Loop_Mod *Tester;             // Pointer to tester/display object
   Chaos01 *Bits;                // Pointer random number source
   FuzzLink *Old_Sol             // Old solution
   FuzzLink *New_Sol;            // New solution
   ofstream *Out;                // Output stream for statistics file

// Implementation function
   void Start_Sol();             // Creates starting population
   void Generate();              // Creates mutated copy solution

public:
   E_S_Class(){}                 // Vanilla constructor
   E_S_Class(double, int, char*, char*, char*,
             int = 1, int = 1, int = 0);
                                 // Actual constructor
   virtual ~E_S_Class();         // Destructor
   void Run_ES();                // Executes evolution strategy
};

#endif
```

ESCLASS.CPP

```
//-------------------------------------------------------------------
//
//   This is the file epclass cpp
//
//-------------------------------------------------------------------

// Implementation dependencies ---------------------------------------

#ifndef __ESCLASS_H
#include "esclass.h"
#endif

// End implementation dependencies -----------------------------------

void E_S_Class::Start_Sol()
//-------------------------------------------------------------------
//
//   This function creates the starting population.
//
//-------------------------------------------------------------------
{
// Local variables
      int Mode;                  // Temporary storage for display mode

   // Set threshold of new solution
   Mode = Tester->Get_Mode();
   Tester->Set_Mode(0);
   Old_Sol->Init(*Bits);
   Old_Sol->Best_Alpha();
   Old_Sol->Remove_Redun(*Bits);
   Tester->Set_Mode(Mode);

   // Evaluate new solution
```

```
   *Old_Sol >> *Tester;
   Tester->Evaluate();

   // Write statistics
   *Out << Old_Sol->Get_Cost() << "\n";
   return;
}

void E_S_Class::Generate()
//---------------------------------------------------------------------------
//
//   This function creates a mutated copy of the solution and the
//   mutated copy replaces the original solution if the mutated
//   is an improvement
//
//---------------------------------------------------------------------------
{
// Local variables
      int Mode;                // Temporary storage of display mode

   // Mutate solution
   *New_Sol = *Old_Sol;
   New_Sol->Mutate(*Old_Sol, *Bits, Rate);

   // Set threshold of new solution
   Mode = Tester->Get_Mode();
   Tester->Set_Mode(0);
   New_Sol->Best_Alpha();
   New_Sol->Remove_Redun(*Bits);
   Tester->Set_Mode(Mode);

   // Evaluate new solution
   *New_Sol >> *Tester;
   Tester->Evaluate();


   if (New_Sol->Get_Cost() < Old_Sol->Get_Cost()) {
      *Old_Sol = *New_Sol;
      Tester->Display();
   }

   // Write statistics
   *Out << Old_Sol->Get_Cost() << "\n";

   return;
}

E_S_Class::E_S_Class
//---------------------------------------------------------------------------
//
//   This constructor allocates space for the object
//
//---------------------------------------------------------------------------
(
// Input parameters
      double New_Rate,         // Mutation rate
      int New_Iterations,      // Maximum number of iterations
      char *New_Prob_File,     // Problem data file
      char *New_Seed_File,     // Seed file for random numbers
      char *New_Stat_File,     // File to collect statistics
      int New_Disp,            // Display mode
      int New_Block,           // Flag (1 = improvement step not displayed)
      int New_Key_Pause        // Flag (1 = displays pause for key)
)
{
   // Set parameters
   Rate = New_Rate;
   Iterations = New_Iterations;
   Prob_File = New_Prob_File;
   Seed_File = New_Seed_File;
   Stat_File = New_Stat_File;
   Disp = New_Disp;
   Block = New_Block;
   Key_Pause = New_Key_Pause;

   // Allocate and initialize object
   Problem = new Nodes2(Prob_File);
   Tester = new Loop_Mod(*Problem, Disp, Block, Key_Pause);
   Bits = new Chaos01(Seed_File);
   Old_Sol = new FuzzLink(Tester);
```

```
   Old_Sol->Set_Count();
   New_Sol = new FuzzLink(Tester);
   New_Sol->Set_Count();
   Out = new ofstream(Stat_File);
}

E_S_Class::~E_S_Class()
//---------------------------------------------------------------------
//
//   This destructor frees allocated space for the object.
//
//---------------------------------------------------------------------
{
   Out->close();
   delete Out;
   delete New_Sol;
   delete Old_Sol;
   delete Bits;
   delete Tester;
   delete Problem;
}

void E_S_Class::Run_ES()
//---------------------------------------------------------------------
//
//   This function is the top level function that runs the evolution
//   stategy.
//
//---------------------------------------------------------------------
{
// Local variables
      int i;                    // Counter that counts number of iterations

   // Set up starting population
   Start_Sol();
   for (i = 0; i < Iterations; ++i)
       Generate();
   return;
}
```

SIEVE.H

```
//---------------------------------------------------------------------
//
//   This is the file sieve.h
//
//---------------------------------------------------------------------

// Interface dependencies ---------------------------------------------

#ifndef __LINKS_H
#include "links.h"
#endif

#ifndef __LOOPS3_H
#include "loops3.h"
#endif

// End interface dependencies -----------------------------------------

#ifndef __SIEVE_H
#define __SIEVE_H

class Sieve : public Solution {
//---------------------------------------------------------------------
//
//   This class of objects that creates a list of "atomic" paths in
//   the layout.  Two objects can compare the list of paths for
//   equivalent paths.  Equivalent paths have the same end nodes, the
//   same length and the same number of bends.  During the match
//   procedure equivalent paths in this object are change to those
//   in the input object.
//
//   operators
//      Sieve(Nodes2&, Loop_Base&);
//      Sieve >> Links
//      Sieve >> Loop_Base&
```

```
//
//   functions
//      Match_All(Sieve&)
//
//   interface routines
//      Copy(Nodes2&)
//      Copy(Links&)
//      Copy(Loop_Base&).
//      int Get_East(int, int)
//      int Get_South(int, int)
//
//-------------------------------------------------------------------------
public:
   struct Path_Rec {          // Record of an "atomic" path in the
                              // layout

      int Label;              // Identifier of path
      int Start_Row;          // Row position of start of path
      int Start_Col;          // Column position of start of path
      int End_Row;            // Row position of end of path
      int End_Col;            // Column position of end of path
      int Bends;              // Number of bends in path
      double Length;          // Length of path
      int Match;              // Flag (1 = path has been matched to
                              // another)
      Path_Rec *Last;         // Link field - points to next record
   };

protected:
   int Num;                   // Number of nodes
   int **Node_Copy;           // Nodes matrix
   int **East_Copy;           // East links matrix
   int **South_Copy;          // South links matrix
   int Start_Row;             // Recorded cursor row position
   int Start_Col;             // Recorded cursor column position
   int Row;                   // Cursor row position
   int Col;                   // Cursor column position
   int Direct;                // Direction of tranverse
   int Path;                  // Label of current path
   int Bends;                 // Running total of number of bends
                              // in path
   double Length;             // Running total of length of path
   int *SNAM;                 // Single nodes adjacency matrix
   Path_Rec *Path_Set;        // Pointer to linked list of paths

// Implementation functions
   void Separate_Paths(const Nodes2&);
                              // Creates the list of "atomic" paths
   void Flag_Nodes();         // Set up terminal nodes
   void Get_SNAM(int, int);   // Creates SNAM based on unmarked paths
   void Old_SNAM(int, int);   // Creates SNAM based on all paths
   int Count_SNAM() const;    // Finds degree of node based on SNAM
   int Link_Walk(const Nodes2&);
                              // Traverses a path one link at a time
   int Find_Start();          // Begins the traverse of a path
   void End_Path();           // Ends the traverse of a path
   Path_Rec *Find_Match(const Path_Rec*);
                              // Finds a matching path from the list
                              // of paths

public:
   Sieve(){}                  // Vanilla constructor
   Sieve(const Nodes2&, const Loop_Base&);
                              // Constructor
   ~Sieve();                  // Destructor
   void operator >>(Links&) const;
                              // Puts data to a links object
   void operator >>(Loop_Base&) const;
                              // Puts data to a tester/display object
   void Match_All(const Sieve&);
   void Copy(const Nodes2&); // Gets data from a nodes object
   void Copy(const Links&);  // Gets data from a links object
   void Copy(const Loop_Base&);
                              // Gets data from a tester/display object

// Interface functions
   int Get_East(int i, int j){ return (East_Copy[i][j] >= 1) ? 1 : 0; }
                              // Returns a 1 if the east link is set
   int Get_South(int i, int j){ return (South_Copy[i][j] >= 1) ? 1 : 0; }
                              // Returns a 1 if the south link is set
```

```
};

#endif
```

SIEVE.CPP

```
//---------------------------------------------------------------------
//
//   This is the file sieve cpp
//
//---------------------------------------------------------------------

// Implementation dependencies ----------------------------------------

#ifndef  _MATH_H_INCLUDED
#include <math h>
#endif

#ifndef __SIEVE_H
#include "sieve h"
#endif

// End implementation dependencies ------------------------------------

void Sieve::Separate_Paths
//---------------------------------------------------------------------
//
//   This function creates a list of all the "atomic" paths between
//   terminal nodes in a network.
//
//---------------------------------------------------------------------
(
// Input parameters
      const Nodes2 &N         // Reference to nodes source
)
{
// Local variables
      int EndP;               // Flag (1 = end of the path has been
                              // reached)
      int MoreP.              // Flag (1 = there are more paths)

   // Set up the nodes matrix for terminal nodes
   Flag_Nodes();

   // Traverse all the paths
   MoreP = Find_Start();
   while (MoreP) {
      EndP = 0;
      while (!EndP) {
         EndP = Link_Walk(N);
      }
      End_Path();
      MoreP = Find_Start();
   }
   return;
}

void Sieve::Flag_Nodes()
//---------------------------------------------------------------------
//
//   This function marks all non-basic vertices of degree other
//   than 2 or 0 with 1 similar to basic vertices. Terminal nodes
//   are basic vertices or nodes of degree 1 or greater than 2.
//
//---------------------------------------------------------------------
{
      int Deg;                // Degree of the node
      int i;                  // Row index
      int j;                  // Column index

   for (i = 0; i < Num; ++i)
      for (j = 0; j < Num; ++j)
         if (!Node_Copy[i][j]) {
            Get_SNAM(i, j);
            Deg = Count_SNAM();
            Node_Copy[i][j] = (Deg == 1 || Deg > 2) ? 1 : 0;
         }
```

```
   return.
}

void Sieve. Get_SNAM
//-----------------------------------------------------------------------
//
//   This function creates an SNAM using only unmarked paths
//
//-----------------------------------------------------------------------
(
// Input parameters
      int i.                  // Row position
      int j                   // Column position
)
{
   SNAM[0] = (i <= 0) ? 0    (South_Copy[i - 1][j] == 1) ? 1    0.
   SNAM[1] = (j >= Num - 1) ? 0  : (East_Copy[i][j] == 1) ? 1    0.
   SNAM[2] = (i >= Num - 1) ? 0    (South_Copy[i][j] == 1) ? 1    0.
   SNAM[3] = (j <= 0) ? 0  : (East_Copy[i][j - 1] == 1) ? 1    0.
   return.
}

void Sieve. Old_SNAM
  ----------------------------------------------------------------------
.
//   This function creates an SNAM includes marked paths.
//
//-----------------------------------------------------------------------
(
      int i.                  // Row position
      int j                   // Column position
)
{
   SNAM[0] = (i <= 0) ? 0  : (South_Copy[i - 1][j] >= 1) ? 1    0.
   SNAM[1] = (j >= Num - 1) ? 0    (East_Copy[i][j] >= 1) ? 1    0.
   SNAM[2] = (i >= Num - 1) ? 0  : (South_Copy[i][j] >= 1) ? 1    0.
   SNAM[3] = (j <= 0) ? 0    (East_Copy[i][j - 1] >= 1) ? 1    0
   return.
}

int Sieve  Count_SNAM() const
//-----------------------------------------------------------------------
//
//   This function returns the degree of a node based on the current
//   contents of the SNAM.
//
//-----------------------------------------------------------------------
:
   return SNAM[0] + SNAM[1] + SNAM[2] + SNAM[3].
;

int Sieve  Link_Walk
//-----------------------------------------------------------------------
//
//   This function traverses a path between two terminal nodes one
//   link at a time while maintaining running totals and marking
//   links with a path label. It returns the value of the current
//   node. 1 if terminal otherwise 0
//
//-----------------------------------------------------------------------
(
      const Nodes2 &N         // Reference to nodes source
)
{
      int Old_Direct.         // Previous direction of traverse

   // Mark link and update length of path
   switch (Direct) {
      case 0:
         --Row:
         South_Copy[Row][Col] = Path:
         Length += N.Get_DY(Row):
         break:
      case 1:
         East_Copy[Row][Col] = Path:
         Length += N.Get_DX(Col):
         ++Col:
         break:
      case 2:
         South_Copy[Row][Col] = Path:
```

```
            Length += N.Get_DY(Row);
            ++Row;
            break;
         case 3:
            --Col;
            East_Copy[Row][Col] = Path;
            Length += N.Get_DX(Col);
            break;
      }
      // Determine new direction and bends
      if (!Node_Copy[Row][Col]) {
         Old_Direct = Direct;
         Get_SNAM(Row, Col);
         if (SNAM[0]) {
            Direct = 0;
         } else if (SNAM[1]) {
            Direct = 1;
         } else if (SNAM[2]) {
            Direct = 2;
         } else if (SNAM[3]) {
            Direct = 3;
         }
         if (Direct != Old_Direct)
            ++Bends;
      }
      return Node_Copy[Row][Col];
   }

   int Sieve::Find_Start()
   //-------------------------------------------------------------------------
   //
   //   This function finds a terminal node with at least one unmarked
   //   path and begins the traverse of that path.  It returns a value
   //   of 1 if a new path is found.
   //
   //-------------------------------------------------------------------------
   {
      int i;                    // Row index
      int j;                    // Column index
      int Found;                // Flag (1 = the start of a path was
                                //       found)

      // Search entire grid for starting location of new path
      Found = 0;
      for (i = 0; i < Num && !Found; ++i)
         for (j = 0; j < Num && !Found; ++j)
            if (Node_Copy[i][j]) {

               // Find an unmarked path and its direction
               Get_SNAM(i, j);
               if (Count_SNAM()) {
                  if (SNAM[0]) {
                     Direct = 0;
                  } else if (SNAM[1]) {
                     Direct = 1;
                  } else if (SNAM[2]) {
                     Direct = 2;
                  } else {
                     Direct = 3;
                  }
                  Row = i;
                  Col = j;
                  Start_Row = Row;
                  Start_Col = Col;

                  // Determine the number of bends at the start
                  // of the path
                  Bends = 0;
                  Old_SNAM(Row, Col);
                  if (Count_SNAM() > 1)
                     switch (Direct) {
                        case 0:
                           if (!SNAM[2])
                              ++Bends;
                           break;
                        case 1:
                           if (!SNAM[3])
                              ++Bends;
                           break;
                        case 2:
```

```
                        if (!SNAM[0])
                           ++Bends;
                        break;
                     case 3:
                        if (!SNAM[1])
                           ++Bends;
                        break;
                  }

               // Initialize running total of length
               Length = 0.0;

               // Create a new path label
               ++Path;
               Found = 1;
            }
         }
   return Found;
}

void Sieve::End_Path()
//---------------------------------------------------------------------
//
//   This function completes the traverse of a path. All running
//   totals are completed and a new record is added to the list
//   of paths.
//
//---------------------------------------------------------------------
{
// Local variables
      Path_Rec *Next;         // Pointer to a newly created record

   // Determine bends at the end of a path
   Old_SNAM(Row, Col);
   if (Count_SNAM() > 1)
      switch (Direct) {
         case 0:
            if (!SNAM[0])
               ++Bends;
            break;
         case 1:
            if (!SNAM[1])
               ++Bends;
            break;
         case 2:
            if (!SNAM[2])
               ++Bends;
            break;
         case 3:
            if (!SNAM[3])
               ++Bends;
            break;
      }

   // Create a new record and add to the list
   Next = new Path_Rec;
   Next->Label = Path;
   Next->Start_Row = Start_Row;
   Next->Start_Col = Start_Col;
   Next->End_Row = Row;
   Next->End_Col = Col;
   Next->Bends = Bends;
   Next->Length = Length;
   Next->Match = 0;
   Next->Last = Path_Set;
   Path_Set = Next;
   return;
}

Sieve::Path_Rec *Sieve::Find_Match
//---------------------------------------------------------------------
//
//   This function searches all elements in the list of paths to
//   see if one of the paths is equivalent to the pattern
//   specified. It returns a pointer to the matching path if a
//   match is found otherwise it returns NULL.
//
//---------------------------------------------------------------------
{
// Input parameters
```

```
      const Path_Rec *Pattern     // Pointer to element to be matched
)
{
// Local variables
      Path_Rec *Cursor;           // Cursor used to search list
      Path_Rec *Match;            // Pointer to a matching element

   // Assume there is no match
   Match = NULL;

   // Search the entire list
   Cursor = Path_Set;
   while (Cursor != NULL && Match == NULL) {

      // If element in list has not been matched already
      if (!Cursor->Match)

         // Check to see if terminal nodes match
         if ((Pattern->Start_Row == Cursor->Start_Row &&
              Pattern->Start_Col == Cursor->Start_Col &&
              Pattern->End_Row == Cursor->End_Row &&
              Pattern->End_Col == Cursor->End_Col) ||
             (Pattern->Start_Row == Cursor->End_Row &&
              Pattern->Start_Col == Cursor->End_Col &&
              Pattern->End_Row == Cursor->Start_Row &&
              Pattern->End_Col == Cursor->Start_Col))

            // Check to see if lengths match within a tolerance
            if ((fabs(Pattern->Length - Cursor->Length) < 0.5) &&
                (Pattern->Bends == Cursor->Bends)) {
               Cursor->Match = 1;
               Match = Cursor;
            }
      Cursor = Cursor->Last;
   }
   return Match;
}

Sieve::Sieve
//---------------------------------------------------------------------
//
//   This constructor allocates space for the object, copies the
//   specified input data and creates the list of paths
//
//---------------------------------------------------------------------
(
      const Nodes2 &N             // Reference to nodes source
      const Loop_Base &Tester     // Reference to tester/display object

  Solution()
{
// Local variables
      int i;                      // Row index

   // Initialize variables
   Num = N.Get_Num();
   Path = 1

   // Allocate space for SNAM
   SNAM = new int[4];

   // Initialize the pointer to the list
   Path_Set = NULL;

   // Allocate space for matrices
   Node_Copy = new int*[Num];
   East_Copy = new int*[Num];
   South_Copy = new int*[Num];
   for (i = 0; i < Num; ++i) {
      Node_Copy[i] = new int[Num];
      East_Copy[i] = new int[Num];
      South_Copy[i] = new int[Num];
   }

   // Get the input data
   Copy(N);
   Copy(Tester);

   // Create the list of paths
   Separate_Paths(N);
```

```
}

Sieve::~Sieve()
//---------------------------------------------------------------------
//
//   This destructor frees all space allocated to this object.
//
//---------------------------------------------------------------------
{
// Local variables
      Path_Rec *Next;          // Cursor for linked list
      int i;                   // Row index

   // Free all space allocated to the linked list
   while (Path_Set != NULL) {
      Next = Path_Set->Last;
      delete Path_Set;
      Path_Set = Next;
   }

   // Free spaces allocated to arrays
   for (i = Num - 1; i >= 0; --i) {
      delete South_Copy[i];
      delete East_Copy[i];
      delete Node_Copy[i];
   }
   delete South_Copy;
   delete East_Copy;
   delete Node_Copy;
   delete SNAM;
}

void Sieve::operator >>
//---------------------------------------------------------------------
//
//   This operator copies the contents of the sieve object to the
//   specified links object.  The operator returns this object.
//
//---------------------------------------------------------------------
(
// Input parameters
      Links &Link_Obj          // Reference to links object
)
const
{
// Local variables
      int i;                   // Row index
      int j;                   // Column index

   // Copy the contents of the links matrices to the links object
   for (i = 0; i < Num; ++i)
      for (j = 0; j < Num; ++j) {
         // If the elements of east links matrix are non-zero
         // set the corresponding elements in the links
         // object otherwise clear the elements
         if (East_Copy[i][j] >= 1)
            Link_Obj.Set_East(i, j);
         else
            Link_Obj.Clear_East(i, j);
         // If the elements of south links matrix are non-zero
         // set the corresponding elements in the links
         // object otherwise clear the elements
         if (South_Copy[i][j] >= 1)
            Link_Obj.Set_South(i, j);
         else
            Link_Obj.Clear_South(i, j);
      }
   return;
}

void Sieve::operator >>
//---------------------------------------------------------------------
//
//   This operator copies the contents of the sieve object to the
//   specified tester/display object.  The operator returns this
//   object.
//
//---------------------------------------------------------------------
(
// Input parameters
```

```
      Loop_Base &Tester       // Reference to tester/display object
)
const
{
// Local variables
      int N1;                 // Temporary (Number of nodes minus 1)
      int i;                  // Row index
      int j;                  // Column index

   // Calculate value of temporary variable
   N1 = Num - 1;

   // Copy the contents of the nodes matrix to the
   // tester/display object
   for (i = 0; i < Num; ++i)
      for (j = 0; j < Num; ++j)
         Tester.Put_Node(Node_Copy[i][j], i, j);

   // If the elements of east links matrix are non-zero
   // set the corresponding elements in the tester/display
   // object otherwise clear the elements
   for (i = 0; i < Num; ++i)
      for (j = 0; j < N1; ++j)
         if (East_Copy[i][j])
            Tester.Set_East(i, j);
         else
            Tester.Clear_East(i, j);

   // If the elements of south links matrix are non-zero
   // set the corresponding elements in the tester/display
   // object otherwise clear the elements
   for (i = 0; i < N1; ++i)
      for (j = 0; j < Num; ++j)
         if (South_Copy[i][j])
            Tester.Set_South(i, j);
         else
            Tester.Clear_South(i, j);

   // Set state variables
   Tester.Put_B_State(Loop_Base::Binary);
   Tester.Put_T_State(Loop_Base::Unknown);
   Tester.Put_E_State(Loop_Base::Not_Evaluated);
   Tester.Put_Sol((Solution*) this);
   return;
}

void Sieve::Match_All
//---------------------------------------------------------------------
//
//   This function iterates through all elements in the list of
//   paths of another structure to determin which are equivalent
//   in this structure.  Any equivalent paths if found are changed
//   in the links matrices to those found in the other structure
//
//---------------------------------------------------------------------
(
      const Sieve &Source     // Reference to another sieve
)
{
      Path_Rec *Cursor;       // Pointer to record in other structure
                              // currently being compared
      Path_Rec *Match;        // Pointer to matching record in this
                              // structure
      int Remove;             // Label of path to be removed
      int Replace;            // Label of path that replaces
                              // removed path
      int Clear;              // Flag (1 = path is clear)
      int i;                  // Row index
      int j;                  // Column index

   // Compare every path from the other structure until finished
   Cursor = Source.Path_Set;
   while (Cursor != NULL) {
      Match = Find_Match(Cursor);

      // If a match occurs
      if (Match) {

         Remove = Match->Label;
         Replace = Cursor->Label;
```

```
         // Determine if matched path can be moved back
         Clear = 1;
         for (i = 0; i < Num && Clear; ++i)
            for (j = 0; j < Num && Clear; ++j) {
               if (Source.East_Copy[i][j] == Replace &&
                   East_Copy[i][j] != Remove &&
                   East_Copy[i][j] != 0)
                  Clear = 0;
               if (Source.South_Copy[i][j] == Replace &&
                  South_Copy[i][j] != Remove &&
                  South_Copy[i][j] != 0)
                  Clear = 0;
            }

         // If path is clear move matched path back to original
         // position
         if (Clear) {

            // All links removed in the matching path are removed
            for (i = 0; i < Num; ++i)
               for (j = 0; j < Num; ++j) {
                  if (East_Copy[i][j] == Remove)
                     East_Copy[i][j] = 0;
                  if (South_Copy[i][j] == Remove)
                     South_Copy[i][j] = 0;
               }

            // Matched path from ther structure takes the place
            // of the removed path
            for (i = 0; i < Num; ++i)
               for (j = 0; j < Num; ++j) {
                  if (Source.East_Copy[i][j] == Replace)
                     East_Copy[i][j] = Remove;
                  if (Source.South_Copy[i][j] == Replace)
                     South_Copy[i][j] = Remove;
               }
         }
      }
      Cursor = Cursor->Last;
   }
   return;
}

void Sieve::Copy
//---------------------------------------------------------------------
//
//   This function copies the contents of the nodes object to this
//   object
//
//---------------------------------------------------------------------
(
   Input parameters
      const Nodes2 &Snapshot // Reference to nodes source
)
{
// Local variables
      int i;                 // Row index
      int j;                 // Column index

   for (i = 0; i < Num; ++i)
      for (j = 0; j < Num; ++j)
         Node_Copy[i][j] = Snapshot.Get_Node(i, j);
   return;
}

void Sieve::Copy
//---------------------------------------------------------------------
//
//   This function copies the contents of the links object to this
//   object.
//
//---------------------------------------------------------------------
(
// Input parameters
      const Links &Snapshot  // Reference to links object
)
{
// Local variables
      int N1;                // Temporary (number of nodes minus 1)
```

```
      int i;                    // Row index
      int j;                    // Column index

   // Calculate value of temporary variable
   N1 = Num - 1;

   // Copy contents of east links matrix to this object
   for (i = 0; i < Num; ++i)
      for (j = 0; j < N1; ++j)
         East_Copy[i][j] = Snapshot.Get_East(i, j);

   // Set east column to 0
   for (i = 0; i < Num; ++i)
      East_Copy[i][N1] = 0;

   // Copy contents of south links matrix to this object
   for (i = 0; i < N1; ++i)
      for (j = 0; j < Num; ++j)
         South_Copy[i][j] = Snapshot.Get_South(i, j);

   // Set south row to 0
   for (j = 0; j < Num; ++j)
      South_Copy[N1][j] = 0;
   return;
}

void Sieve::Copy
//-------------------------------------------------------------------------
//
//   This function copies the contents of the tester/display object
//   to this object
//
//-------------------------------------------------------------------------
(
// Input parameters
      const Loop_Base &Snapshot    // Reference to tester/display
                                   // object

)
{
// Local variables
      int N1;                      // Temporary (number of nodes minus 1)
      int i;                       // Row index
      int j;                       // Column index

   // Calculate value of temporary variable
   N1 = Num - 1;

   // Copy contents of east links matrix to this object
   for (i = 0; i < Num; ++i)
      for (j = 0; j < N1; ++j)
         East_Copy[i][j] = Snapshot.Get_East(i, j);

   // Set east column to 0
   for (i = 0; i < Num; ++i)
      East_Copy[i][N1] = 0;

   // Copy contents of south links matrix to this object
   for (i = 0; i < N1; ++i)
      for (j = 0; j < Num; ++j)
         South_Copy[i][j] = Snapshot Get_South(i, j);

   // Set south row to 0
   for (j = 0; j < Num; ++j)
      South_Copy[N1][j] = 0;
   return;
}
```

CLUMP.H

```
//-------------------------------------------------------------------------
//
//   This is the file clump.h
//
//-------------------------------------------------------------------------

// Interface dependencies ---------------------------------------------
```

```
#ifndef __LINKS_H
#include "links.h"
#endif

#ifndef __LOOPS3_H
#include "loops3.h"
#endif

// End interface dependencies ------------------------------------

#ifndef __CLUMP_H
#define __CLUMP_H

class Clump : public Links {
//------------------------------------------------------------------------
//
//   Objects of this class label patterns of contiguous link with
//   unique identifiers with the top level function "Sift"
//
//   functions
//      Sift()
//
//   interface functions
//      Copy_To(Loop_Base&, int)
//      int Get_East(int, int, int)
//      int Get_South(int, int, int)
//
//------------------------------------------------------------------------
protected

// Implementation functions
   int Find_Start();          // Finds the starting location of a clump
   void Mark_East(int, int, int);
                              // Marks an east link in a clump
   void Mark_South(int, int, int);
                              // Marks a south link in a clump
   int Scan(int);             // Finds all the links in a clump
   void Dec_Clumps();         // Decrements the values of all the
                              // labels of the clumps

public:
   Clump(){}                  // Vanilla constructor
   Clump(int N) : Links(N) {}
                              // Creates from number of nodes
   virtual ~Clump(){}         // Destructor
   void Sift();               // Find all the clumps
   void Copy_To(Loop_Base&, int);
                              // Copy a single clump to the
                              // tester/display object
   int Get_East(int i, int j, int Num) const
      { return (East[i][j] == Num) ? 1 : 0; }
                              // Returns 1 if the east link is set
   int Get_South(int i, int j, int Num) const
      { return (South[i][j] == Num) ? 1 : 0; }
                              // Returns 1 if the south link is set
};

#endif
```

CLUMP.CPP

```
//------------------------------------------------------------------------
//
//   This is the file clump.cpp
//
//------------------------------------------------------------------------

// Implementation dependencies -----------------------------------

#ifndef __CLUMP_H
#include "clump.h"
#endif

// End implementation dependencies -------------------------------

int Clump::Find_Start()
//------------------------------------------------------------------------
```

```
//
//   This function searches by row and column to find the first east
//   or south link equal to 1 and sets it equal to -1.  If a link is
//   found 1 is returned otherwise 0 is returned.
//
//---------------------------------------------------------------------
{
// Local variables
      int Found;                  // Flag (1 = location of clump found)
      int N1;                     // Temporary (number of nodes - 1)
      int i;                      // Row index
      int j;                      // Column index

   // Search for an east or south link equal to 1
   N1 = Num_Nodes - 1;
   Found = 0;
   for (i = 0; i < Num_Nodes && !Found; ++i)
      for (j = 0; j < Num_Nodes && !Found; ++j) {
         // Do not search last column
         if (j < N1)
            if (East[i][j] == 1) {
               East[i][j] = -1;
               Found = 1;
            }
         // Do not search last row
         if (!Found && i < N1)
            if (South[i][j] == 1) {
               South[i][j] = -1;
               Found = 1;
            }
      }
   return Found;
}

void Clump::Mark_East
//---------------------------------------------------------------------
//
//   This function changes the value of an east link specified at
//   row "i" and column "j" to the number "Clump-Num".  Any of the
//   six possible adjacent links that are set equal to 1 are changed
//   to -1.
//
//---------------------------------------------------------------------
{
// Input parameters
      int i;                      // Row position
      int j;                      // Column position
      int Clump_Num               // Label of the clump

)
{
   // Set the link at the current position to the clump number
   East[i][j] = Clump_Num;

   // Search all six adjacent links and change to -1 if
   // the link is set to 1
   if (j > 0)
      if (East[i][j - 1] == 1)
         East[i][j - 1] = -1;
   if (j < Num_Nodes - 2)
      if (East[i][j + 1] == 1)
         East[i][j + 1] = -1;

   if (i > 0)
      if (South[i - 1][j] == 1)
         South[i - 1][j] = -1;
   if (i < Num_Nodes - 1)
      if (South[i][j] == 1)
         South[i][j] = -1;

   if (i > 0)
      if (South[i - 1][j + 1] == 1)
         South[i - 1][j + 1] = -1;
   if (i < Num_Nodes - 1)
      if (South[i][j + 1] == 1)
         South[i][j + 1] = -1;
   return;
}

void Clump::Mark_South
//---------------------------------------------------------------------
```

```
//
//   This function changes the value of an south link specified at
//   row "i" and column "j" to the number "Clump-Num".  Any of the
//   six possible adjacent links that are set equal to 1 are changed
//   to -1.
//
//-------------------------------------------------------------------
(
// Input parameters
      int i.                    // Row position
      int j.                    // Column position
      int Clump_Num             // Label of the clump
)
{
   // Set the link at the current position to the clump number
   South[i][j] = Clump_Num.

   // Search all six adjacent links and change to -1 if
   // the link is set to 1
   if (i > 0)
      if (South[i - 1][j] == 1)
         South[i - 1][j] = -1.
   if (i < Num_Nodes - 2)
      if (South[i + 1][j] == 1)
         South[i + 1][j] = -1.

   if (j > 0)
      if (East[i][j - 1] == 1)
         East[i][j - 1] = -1.
   if (j < Num_Nodes - 1)
      if (East[i][j] == 1)
         East[i][j] = -1.

   if (j > 0)
      if (East[i + 1][j - 1] == 1)
         East[i + 1][j - 1] = -1.
   if (j < Num_Nodes - 1)
      if (East[i + 1][j] == 1)
         East[i + 1][j] = -1.
   return.
}

int Clump::Scan
//-------------------------------------------------------------------
//
//   This function marks all links set to -1 with clump number and
//   changes all links set to 1 to -1.
//
//-------------------------------------------------------------------
(
      int Clump_Num             // Current clump number
)
{
      int Found.                // Flag (1 = link or links set to -1
                                // found)
      int N1.                   // Temporary (number of nodes minus 1)
      int i.                    // Row index
      int j.                    // Column index

   // Search all east and south links and mark every one
   // that is equal to -1 with the clump number
   N1 = Num_Nodes - 1.
   Found = 0.
   for (i = 0. i < Num_Nodes. ++i)
      for (j = 0. j < N1. ++j)
         if (East[i][j] == -1) {
            Found = 1.
            Mark_East(i. j. Clump_Num).
         }
   for (i = 0. i < N1. ++i)
      for (j = 0. j < Num_Nodes. ++j)
         if (South[i][j] == -1) {
            Found = 1.
            Mark_South(i. j. Clump_Num).
         }
   return Found.
}

void Clump::Dec_Clumps()
//-------------------------------------------------------------------
```

```
//
//   This function searchs the entire Hanan grid and decrements the
//   value of any link not set to 0.
//
//-------------------------------------------------------------------------
{
// Local variables
      int N1.                    // Temporary (number of nodes minus 1)
      int i;                     // Row index
      int j.                     // Column index

   N1 = Num_Nodes - 1;
   for (i = 0. i < Num_Nodes. ++i)
      for (j = 0; j < N1. ++j)
         if (East[i][j] > 1)
            --East[i][j].
   for (i = 0. i < N1. ++i)
      for (j = 0; j < Num_Nodes; ++j)
         if (South[i][j] > 1)
            --South[i][j].
   return.
}

void Clump..Sift()
//-------------------------------------------------------------------------
//
//   This function is a top level routine that assigns a unique
//   identifier to each clump or contiguous arrangement of links.
//
//-------------------------------------------------------------------------
{
// Local variables
      int Clump_Num.             // Label for each clump

   // Initialize clump number
   Clump_Num = 2.

   // Scan all clumps
   while (Find_Start()) {
      while (Scan(Clump_Num)).
      ++Clump_Num.
   }

   // Decrement clumps
   Dec_Clumps().
   return.
}

void Clump. Copy_To
//-------------------------------------------------------------------------
//
//   This function copies the pattern in the clump specified by "T"
//   into the tester/display object "Tester"
//
//-------------------------------------------------------------------------
(
// Input parameters
      Loop_Base &Tester.         // Reference to tester/display object
      int T                      // Label of clump to be copied
)
{
// Local variables
      int N1;                    // Temporary (number of nodes minus 1)
      int i;                     // Row index
      int j.                     // Column index

   // Calculate N1
   N1 = Num_Nodes - 1;

   // Set each east link equal to T
   for (i = 0. i < Num_Nodes. ++i)
      for (j = 0; j < N1; ++j)
        if (East[i][j] == T)
           Tester.Set_East(i. j);
        else
           Tester.Clear_East(i. j);

   // Set each south link equal to T
   for (i = 0; i < N1; ++i)
      for (j = 0; j < Num_Nodes; ++j)
```

```
        if (South[i][j] == T)
           Tester Set_South(i, j);
        else
           Tester Clear_South(i, j);
     return;
  }
```

RULE.H

```
//------------------------------------------------------------------------
//
//   This is the file rule.h
//
//------------------------------------------------------------------------

// Interface dependencies ------------------------------------------------

#ifndef __LOOPS3_H
#include "loops3.h"
#endif

#ifndef __CLUMP_H
#include "clump.h"
#endif

#ifndef __FUZZLINK_H
#include "fuzzlink.h"
#endif

// End interface dependencies --------------------------------------------

#ifndef __RULE_H
#define __RULE_H

#define MIN_DEGREE 2
#define ROUND_OFF 0.005

enum Dispose_Spec {          // Request to dispose rules with poor history

   Dispose_Bad,              // Dispose all bad rules
   Dispose_Unused,           // Dispose unused rules
   No_Dispose                // Keep all rules
};

class Rule {
//------------------------------------------------------------------------
//
//   This class contains the antecedent and consequent clauses of a
//   rule as well as the counter that monitor the rule's performance.
//
//   operators
//      Rule(int, int)
//
//   general functions
//      int Read(istream)
//      Print(ostream)
//
//   rule formulation functions
//      Derive(Links, Links, int, int)
//      Derive(Links, Clump, int, int, int)
//      Copy90(Rule*)
//      Copy180(Rule*)
//      Copy270(Rule*)
//      Reflect(Rule*)
//      Set_Sinks()
//      int Rule_Cmp(Rule*)
//
//   rule application functions
//      int Instantiate(Loop_Base, int, int)
//      double Calc_Priority(Loop_Base, int, int)
//      Fire(Loop_Base, FuzzLink, int, int);
//
//   interface functions
//      Rule* Get_Next()
//      Put_Next(Rule*)
//      Rule* Get_Last()
//      Put_Last(Rule*)
```

```
//      int Get_Rows()
//      int Get_Cols()
//      Use()
//      int Get_Used()
//      Put_Used(int)
//      Failure()
//      int Get_Failed()
//      Put_Failed(int)
//      Dump()
//      int Get_Dumped()
//      Put_Dumped(int)
//      int Get_Category
//      Put_Category(int)
//
//-------------------------------------------------------------------------
protected:
   Rule *Next;                  // Pointer to next rule in rule base
   Rule *Last;                  // Pointer to last rule in rule base
   int Rows;                    // Number of rows in matrices
   int Cols;                    // Number of columns in matrices
   int **Sinks;                 // Sinks matrix
   int **Ante_E;                // East antecedent matrix
   int **Ante_S;                // South antecedent matrix
   int **Conse_E;               // East consequent matrix
   int **Conse_S;               // South consequent matrix
   int Used;                    // Used counter
   int Failed;                  // Failed counter
   int Dumped;                  // Dumped conter
   int Category;                // Number common to all variants of the
                                // rule

// Implementation functions
   int Verify_East(const Loop_Base&, int, int, int, int) const;
                                // Checks that east link matches
                                // in tester/display
   int Verify_South(const Loop_Base&, int, int, int, int) const;
                                // Checks that south link matches
                                // in tester/display
   int Cmp_East(const Rule*) const;
                                // Checks that east link matches
                                // in other rule
   int Cmp_South(const Rule*) const;
                                // Checks that east link matches
                                // in other rule

public:
   Rule(){}                     // Vanilla constructor
   Rule(int, int);              // Actual constructor
   virtual ~Rule();             // Destructor
   int Read(istream&, Dispose_Spec);
                                // Reads a rule from the input stream
   int Print(ostream&, Dispose_Spec) const;
                                // Prints a rule to the output stream
   void Derive(const Links&, const Links&, int, int);
                                // Derives contents of a rule
   void Derive(const Links&, const Clump&, int, int, int);
                                // Derives contents of a rule
   void Copy90(const Rule*);    // Creates 90 degree rotated copy
   void Copy180(const Rule*);   // Creates 180 degree rotated copy
   void Copy270(const Rule*);   // Creates 270 degrees rotated copy
   void Reflect(const Rule*);   // Creates a reflected copy
   void Set_Sinks();            // Derives the contents of the sinks
                                // matrix
   int Rule_Cmp(const Rule*) const;
                                // Compares this rule to another

   int Instantiate(const Loop_Base&, int, int) const;
                                // Tests to see if rule will instantiate
   double Calc_Priority(const Loop_Base&, int, int);
                                // Calculates priority of instantiation
   void Fire(Loop_Base&, FuzzLink&, int, int);
                                // Fires the rule

// Interface functions
   Rule *Get_Next() const { return Next; }
                           // Gets the next rule in the linked list
   void Put_Next(Rule *New_Next){ Next = New_Next; return; }
                           // Connects the rule to the next rule
   Rule *Get_Last() const { return Last; }
                           // Gets the previous rule in the linked list
```

```
   void Put_Last(Rule *New_Last){ Last = New_Last; return; }
                              // Connects the rule to the previous rule
   int Get_Rows() const { return Rows; }
                              // Returns the number of rows in the matrices
   int Get_Cols() const { return Cols; }
                              // Returns the number of columns in the
                              // matrices
   void Use() { ++Used; return; }
                              // Increments the "Used" counter
   int Get_Used() const { return Used; }
                              // Returns the value of the "Used" counter
   void Put_Used(int New_Used ) { Used = New_Used; return; }
                              //  Changes the value of the "Used" counter
   void Failure() { ++Failed; return; }
                              // Increments the "Failed" counter
   int Get_Failed() const { return Failed; }
                              // Returns the value of the "Failed" counter
   void Put_Failed(int New_Failed ) { Failed = New_Failed; return; }
                              //  Changes the value of the "Failed" counter
   void Dump() { ++Dumped; return; }
                              // Increments the "Dumped" counter
   int Get_Dumped() const { return Dumped; }
                              // Returns the value of the "Dumped" counter
   void Put_Dumped(int New_Dumped ) { Dumped = New_Dumped; return; }
                              //  Changes the value of the "Dumped" counter
   int Get_Category() const { return Category; }
                              // Returns the category value of the rule
   void Put_Category(int New_Cat) { Category = New_Cat; return; }
                              // Assigns a new category value to the rule
};

struct Boundary {             // Defines the boundaries of a change

   int Min_Row;               // Minmum row position
   int Min_Col;               // Minmum column position
   int Max_Row;               // Maximum row position
   int Max_Col;               // Maximum column position
};

struct Trig_Rule {            // Instantiation record

   Rule *Triggered;           // Pointer to rule
   double Priority;           // Priority of rule
   int Row;                   // Row offset
   int Col;                   // Column offset
   Trig_Rule *Last;           // Pointer to previous record
};

class Rule_Base {
//-------------------------------------------------------------------------
//
//   This class contains the antecedent and consequent clauses of a
//   rule as well as the counter that monitor the rule's performance
//
//   operators
//
//   general functions
//      Collect_Counters()
//      Print(ostream, Dispose_Spec)
//      Fprint(char*, Dispose_Spec)
//      Add(char*, Dispose_Spec)
//
//   rule formulation functions
//      Add(Links, Clump, Boundary, int);
//
//   rule application functions
//      int Apply(FuzzLink)
//      Trigger()
//      Fire2(FuzzLink&)
//      Empty()
//      Clean_Up()
//      Rule_Fails()
//
//   interface functions
//      int Get_Rows(Rule*)
//      int Get_Cols(Rule*)
//      long Get_Num_Rules()
//
//-------------------------------------------------------------------------
protected:
```

```
    Loop_Base *Tester;          // Pointer to tester/display object
    Rule *Head;                 // Pointer to head of rule list
    Rule *Tail;                 // Pointer to tail of rule list
    int Type_Counter;           // Counter for each type of rule
    int Read_Type;              // Type of last rule read
    Trig_Rule *Trig_Set;        // Pointer to trigger set list
    int Trig_Count;             // Number of records in trigger set
    Trig_Rule **T_Array;        // Trigger set array
    int Next_Rule;              // Pointer to next instantiation to fire
    long Num_Rules;             // Number of rules in rule base
    long Rules_Written;         // Number of rules written in last write
    long Rules_Read;            // Number of rules read in last read
    long Max_Rules;             // Number of rules in rule base before last
                                // forget

    void Attach_Rule(Rule*);    // Attaches a rule to the rule base
    int Is_KnownP(const Rule*) const;
                                // Checks to see if rule was acquired
                                // previously

public
    Rule_Base(){}               // Vanilla constructor
    Rule_Base(Loop_Base*);      // Actual constructor
    virtual ~Rule_Base();       // Destructor
    void Collect_Counters();    // Collects and distributes the counters
                                // among each category of rules
    void Set_All_Counters();    // Sets counters so all rules will be saved
                                // even when forgetting is enabled
    void Print(ostream&, Dispose_Spec);
                                // Prints rule base to output stream
    void Fprint(char*, Dispose_Spec);
                                // Writes rules to a file
    void Add(char*, Dispose_Spec);
                                // Reads in one rule from a file

    void Add(const Links&, const Clump&, Boundary, int);
                                // Derives and adds a new rule to the rule base

    int Apply(FuzzLink&);       // Applies rules the next instantiation
    void Trigger();             // Creates a new trigger set
    int Fire2(FuzzLink&);       // Fires the next rule in the trigger set
    void Empty();               // Empties rule base
    void Clean_Up();            // Destroys a trigger set
    void Rule_Fails();          // Updates the "Failed" counter of the last
                                // rule to fire

// Interface functions
    int Get_Rows(Rule *Current) const { return Current->Get_Rows(); }
                                // Returns the number of rows in the matrices
    int Get_Cols(Rule *Current) const { return Current->Get_Cols(); }
                                // Returns the number of columns in the
                                // matrices
    long Get_Num_Rules() const { return Num_Rules; }
                                // Returns the number of rules in the rule
                                // base
    long Get_Num_Written() const { return Rules_Written; }
                                // Returns the number of rules written in
                                // last write
    long Get_Num_Read() const { return Rules_Read; }
                                // Returns the number of rules read in
                                // last read
    void Update_Max() { Max_Rules = Num_Rules; return; }
                                // Records number of rules in the rule base
                                // prior to forgetting
    long Get_Max() const { return Max_Rules; }
                                // Returns the number of rules prior to the
                                // last forget
};

#endif
```

RULE.CPP

```
//---------------------------------------------------------------------
//
//   This is the file rule.cpp
//
```

```
//-----------------------------------------------------------------------

// Implementation dependencies ------------------------------------------

#ifndef __STDLIB_H
#include <stdlib.h>
#endif

#ifndef __CUSTIO_H
#include "custio.h"
#endif

#ifndef __RULE_H
#include "rule.h"
#endif

// End implementation dependencies --------------------------------------

int Rule::Verify_East
//-----------------------------------------------------------------------
//
//   This function returns a 1 if an east link and six adjacent
//   links in the antecedent clause at location i, j match the
//   tester/display object at location k, l.
//
//   Applies rules
//
//-----------------------------------------------------------------------
(
// Input parameters
      const Loop_Base &Tester,
                              // Reference to tester/display object
      int i,                  // Row index
      int j,                  // Column index
      int k,                  // Row index with offset
      int l                   // Column index with offset
)
const
{
// Local variables
      int Num;                // Dimensions of the Hanan grid

   // Get the dimensions of the grid
   Num = Tester.Get_Num();
   return

   // Check east links
   (l == 0 &&
      Ante_E[i][j - 1] != 0) ? 0 :
   (l != 0 &&
      Ante_E[i][j - 1] != Tester.Get_East(k, l - 1)) ? 0 :

   (Ante_E[i][j] != Tester.Get_East(k, l)) ? 0 :

   (l == Num - 2 &&
      Ante_E[i][j + 1] != 0) ? 0 :
   (l != Num - 2 &&
      Ante_E[i][j + 1] != Tester.Get_East(k, l + 1)) ? 0 :

   // Check west south links
   (k == 0 &&
      Ante_S[i - 1][j] != 0) ? 0 :
   (k != 0 &&
      Ante_S[i - 1][j] != Tester.Get_South(k - 1, l)) ? 0 :

   (k == Num - 1 &&
      Ante_S[i][j] != 0) ? 0 :
   (k != Num - 1 &&
      Ante_S[i][j] != Tester.Get_South(k, l)) ? 0 :

   // Check east south links
   (k == 0 &&
      Ante_S[i - 1][j + 1] != 0) ? 0 :
   (k != 0 &&
      Ante_S[i - 1][j + 1] != Tester.Get_South(k - 1, l + 1)) ? 0 :

   (k == Num - 1 &&
      Ante_S[i][j + 1] != 0) ? 0 :
   (k != Num - 1 &&
      Ante_S[i][j + 1] != Tester.Get_South(k, l + 1)) ? 0 :
```

```
   // Check nodes
   (Sinks[i][j] == 0 && Tester.Get_Node(k, l) != 0) ? 0
   (Sinks[i][j + 1] == 0 && Tester.Get_Node(k, l + 1) != 0) ? 0 : 1;
}

int Rule::Verify_South
//---------------------------------------------------------------------
//
//   This function returns a 1 if an south link and six adjacent
//   links in the antecedent clause at location i, j match the
//   tester/display object at location k, l.
//
//   Applies rules
//
//---------------------------------------------------------------------
(
// Input parameters
      const Loop_Base &Tester,
                              // Reference to tester/display object
      int i,                  // Row index
      int j,                  // Column index
      int k,                  // Row index with offset
      int l                   // Column index with offset
)
const
{
// Local variables
      int Num;                // Dimensions of the Hanan grid

   // Get the dimensions of the grid
   Num = Tester.Get_Num();
   return

   // Check south links
   (k == 0 &&
      Ante_S[i - 1][j] != 0) ? 0 :
   (k != 0 &&
      Ante_S[i - 1][j] != Tester.Get_South(k - 1, l)) ? 0 :

   (Ante_S[i][j] != Tester.Get_South(k, l)) ? 0 :

   (k == Num - 2 &&
      Ante_S[i + 1][j] != 0) ? 0 :
   (k != Num - 2 &&
      Ante_S[i + 1][j] != Tester.Get_South(k + 1, l)) ? 0 :

   // Check north east links
   (l == 0 &&
      Ante_E[i][j - 1] != 0) ? 0 :
   (l != 0 &&
      Ante_E[i][j - 1] != Tester.Get_East(k, l - 1)) ? 0 :

   (l == Num - 1 &&
      Ante_E[i][j] != 0) ? 0 :
   (l != Num - 1 &&
      Ante_E[i][j] != Tester.Get_East(k, l)) ? 0 :

   // Check south east links
   (l == 0 &&
      Ante_E[i + 1][j - 1] != 0) ? 0 :
   (l != 0 &&
      Ante_E[i + 1][j - 1] != Tester.Get_East(k + 1, l - 1)) ? 0 :

   (l == Num - 1 &&
      Ante_E[i + 1][j] != 0) ? 0 :
   (l != Num - 1 &&
      Ante_E[i + 1][j] != Tester.Get_East(k + 1, l)) ? 0 :

   // Check nodes
   (Sinks[i][j] == 0 && Tester.Get_Node(k, l) != 0) ? 0 :
   (Sinks[i + 1][j] == 0 && Tester.Get_Node(k + 1, l) != 0) ? 0 : 1;
}

int Rule::Cmp_East
//---------------------------------------------------------------------
//
//   This function returns 1 if this rule's east antecedent matrix
//   matches the specified rule's matrix. Only the links in the
//   pattern specified by the consequent clause is checked.
```

```
//
//   Formulate
//
//-------------------------------------------------------------------------
(
// Input parameters
      const Rule *New_Rule   // Pointer to another rule
)
const
{
// Local variables
      int Same.              // Flag (1 = matrices are the same)
      int C1.                // Temporary (Number of columns minus 1)
      int i.                 // Row index
      int j.                 // Column index

   // Calculate temporary variable
   C1 = Cols - 1.

   // Assume matrices are the same
   Same = 1;

   // Where the link is set in the consequent matrix compare the
   // links and adjacent links
   for (i = 1. i < Rows && Same. ++i)
      for (j = 1; j < C1 && Same. ++j)
         if (Conse_E[i][j])
            Same =

   // Check east links
   (Ante_E[i][j - 1] != New_Rule->Ante_E[i][j - 1]) ? 0 :
   (Ante_E[i][j] != New_Rule->Ante_E[i][j]) ? 0
   (Ante_E[i][j + 1] != New_Rule->Ante_E[i][j + 1]) ? 0 :

   // Check east south links
   (Ante_S[i - 1][j] != New_Rule->Ante_S[i - 1][j]) ? 0 :
   (Ante_S[i][j] != New_Rule->Ante_S[i][j]) ? 0

   // Check west south links
   (Ante_S[i - 1][j + 1] != New_Rule->Ante_S[i - 1][j + 1]) ? 0
   (Ante_S[i][j + 1] != New_Rule->Ante_S[i][j + 1]) ? 0 : 1;

   return Same.
}

int Rule::Cmp_South
//-------------------------------------------------------------------------
//
//   This function returns 1 if this rule's south antecedent matrix
//   matches the specified rule's matrix  Only the links in the
//   pattern specified by the consequent clause is checked.
//
//   Formulate
//
//-------------------------------------------------------------------------
(
// Input parameters
      const Rule *New_Rule   // Pointer to another rule
)
const
{
// Local variables
      int Same:              // Flag (1 = matrices are the same)
      int R1:                // Temporary (Number of rows minus 1)
      int i.                 // Row index
      int j:                 // Column index

   // Calculate temporary variable
   R1 = Rows - 1.

   // Assume matrices are the same
   Same = 1;

   // Where the link is set in the consequent matrix compare the
   // links and adjacent links
   for (i = 1; i < R1 && Same; ++i)
      for (j = 1; j < Cols && Same; ++j)
         if (Conse_S[i][j])
            Same =
```

```
   // Check south links
   (Ante_S[i - 1][j] != New_Rule->Ante_S[i - 1][j]) ? 0
   (Ante_S[i][j] != New_Rule->Ante_S[i][j]) ? 0
   (Ante_S[i + 1][j] != New_Rule->Ante_S[i + 1][j]) ? 0

   // Check north east links
   (Ante_E[i][j - 1] != New_Rule->Ante_E[i][j - 1]) ? 0
   (Ante_E[i][j] != New_Rule->Ante_E[i][j]) ? 0

   // Check south east links
   (Ante_E[i + 1][j - 1] != New_Rule->Ante_E[i + 1][j - 1]) ? 0
   (Ante_E[i + 1][j] != New_Rule->Ante_E[i + 1][j]) ? 0 : 1;

   return Same;
}

Rule::Rule
//---------------------------------------------------------------------
//
//   This constructor allocates space and initializes data in a rule.
//
//   Formuation and use
//
//---------------------------------------------------------------------
(
// Input parameters
      int New_Rows;           // Number of rows in rule matrices
      int New_Cols            // Number of columns in rule matrices
)
{
// Local variables
      int i;                  // Row index
      int j;                  // Column index

   Rows = New_Rows;
   Cols = New_Cols;

   // Allocate space for matrices
   Sinks = new int*[Rows];
   Ante_E = new int*[Rows];
   Ante_S = new int*[Rows];
   Conse_E = new int*[Rows];
   Conse_S = new int*[Rows];
   for (i = 0; i < Rows; ++i) {
      Sinks[i] = new int[Cols];
      Ante_E[i] = new int[Cols];
      Ante_S[i] = new int[Cols];
      Conse_E[i] = new int[Cols];
      Conse_S[i] = new int[Cols];
   }

   // Initialize matrices
   for (i = 0; i < Rows; ++i)
      for (j = 0; j < Cols; ++j) {
         Sinks[i][j] = 0;
         Ante_E[i][j] = 0;
         Ante_S[i][j] = 0;
         Conse_E[i][j] = 0;
         Conse_S[i][j] = 0;
      }

   // Initialize counters
   Used = Failed = Dumped = Category = 0;
}

Rule::~Rule()
//---------------------------------------------------------------------
//
//   This destructor frees space used by a rule.
//
//   Formulation and use
//
//---------------------------------------------------------------------
{
// Local variables
      int i;                  // Row index

   if (Rows && Cols) {
      for (i = Rows - 1; i >= 0; --i) {
         delete Conse_S[i];
```

```
        delete Conse_E[i];
        delete Ante_S[i];
        delete Ante_E[i];
        delete Sinks[i];
      }
      delete Conse_S;
      delete Conse_E;
      delete Ante_S;
      delete Ante_E;
      delete Sinks;
   }
}

int Rule::Read
//---------------------------------------------------------------------
//
//   This function reads in all rule data except the size of the
//   rules (row and column).  The function requires that the size
//   data have already been read and the appropriate space has
//   been allocated to the rule
//
//   Applies rules
//
//---------------------------------------------------------------------
(
   Input parameters
      istream &In,                  // Input stream
      Dispose_Spec Forget           // Specifies to dispose poor rules
)
{
// Local variables
      char InChar[2];               // Buffer for reading binary matrices
      int Keep;                     // Flag (1 = the rule will be kept)
      int i;                        // Row index
      int j;                        // Column index

   // Assume rule should be kept
   Keep = 1;

   In.seekg(11L, ios::cur);
   In >> Used;
   In.seekg(9L, ios::cur);
   In >> Failed;
   In.seekg(9L, ios::cur);
   In >> Dumped;
   In.seekg(9L, ios::cur);
   In >> Category;
   InChar[1] = 0;
   In.seekg(9L, ios::cur);
   for (i = 0; i < Rows; ++i) {
      In.seekg(2L, ios::cur);
      for (j = 0; j < Cols; ++j) {
         In >> InChar[0];
         Sinks[i][j] = atoi(InChar);
      }
   }
   In.seekg(12L, ios::cur);
   for (i = 0; i < Rows; ++i) {
      In.seekg(2L, ios::cur);
      for (j = 0; j < Cols; ++j) {
         In >> InChar[0];
         Ante_E[i][j] = atoi(InChar);
      }
   }
   In.seekg(12L, ios::cur);
   for (i = 0; i < Rows; ++i) {
      In.seekg(2L, ios::cur);
      for (j = 0; j < Cols; ++j) {
         In >> InChar[0];
         Ante_S[i][j] = atoi(InChar);
      }
   }
   In.seekg(13L, ios::cur);
   for (i = 0; i < Rows; ++i) {
      In.seekg(2L, ios::cur);
      for (j = 0; j < Cols; ++j) {
         In >> InChar[0];
         Conse_E[i][j] = atoi(InChar);
      }
   }
```

```
   In.seekg(13L, ios::cur);
   for (i = 0; i < Rows; ++i) {
      In.seekg(2L, ios::cur);
      for (j = 0; j < Cols; ++j) {
         In >> InChar[0];
         Conse_S[i][j] = atoi(InChar);
      }
   }

   // Criterion for disposing rules
   if (Forget == Dispose_Unused)
      if (Used == 0)
         Keep = 0;
   if (Forget == Dispose_Bad)
      if (Used == 0 || Failed > 0 || Dumped != 0)
         Keep = 0;

   // Reset all counters
   Used = 0;
   Dumped = 0;
   Failed = 0;

   return Keep;
}

int Rule_Print
//---------------------------------------------------------------------

//   This function prints the rule on the output stream. If the
//   rule meets the criterion for keeping the rule the rule is
//   written and a value of 1 is returned. If the input parameter
//   "Forget" is equal to 0 the rule is always written
//
//   Formulates rules
//
//---------------------------------------------------------------------
(
// Input parameters
      ostream &Out,           // Output stream
      Dispose_Spec Forget     // Specifies to dispose poor rules
)
const
{
// Local variables
      int Written;            // Flag (1 = rule has been written)
      int i;                  // Row index
      int j;                  // Column index

   Written = 0;

   // Criterion for keeping a rule
   if ((Forget == No_Dispose) ||
       (Forget == Dispose_Unused && Used != 0) ||
       (Forget == Dispose_Bad && Dumped == 0 && Failed == 0 && Used != 0)) {

      // Print size
      Set_Int(2);
      (Out << "Rows " << Rows) << "\n";
      (Out << "Cols " << Cols) << "\n\n";

      // Print counters
      Set_Int(5);
      (Out << "Used   " << Used) << "\n";
      (Out << "Failed " << Failed) << "\n";
      (Out << "Dumped " << Dumped) << "\n";
      (Out << "Type   " << Category) << "\n\n";

      // Print sinks
      Set_Int(1);
      Out << "Nodes" << "\n";
      for (i = 0; i < Rows; ++i) {
         for (j = 0; j < Cols; ++j)
            Out << Sinks[i][j];
         Out << "\n";
      }
      Out << "\n\n";

      // Print antecedent clause
      Out << "Ante_E" << "\n";
      for (i = 0; i < Rows; ++i) {
```

```
            for (j = 0; j < Cols; ++j)
               Out << Ante_E[i][j];
            Out << "\n";
         }
         Out << "\n\n";
         Out << "Ante_S" << "\n";
         for (i = 0; i < Rows; ++i) {
            for (j = 0; j < Cols; ++j)
               Out << Ante_S[i][j];
            Out << "\n";
         }
         Out << "\n\n";

         // Print consequent clause
         Out << "Conse_E" << "\n";
         for (i = 0; i < Rows; ++i) {
            for (j = 0; j < Cols; ++j)
               Out << Conse_E[i][j];
            Out << "\n"
         }
         Out << "\n\n";
         Out << "Conse_S" << "\n";
         for (i = 0; i < Rows; ++i) {
            for (j = 0; j < Cols; ++j)
               Out << Conse_S[i][j];
            Out << "\n";
         }
         Out << "\n\n";
         Written = 1;
      }
      return Written;
}

void Rule  Derive
//---------------------------------------------------------------------
//
//   This function sets the valuse of the clause matrices of the
//   rule  Data are obtained from two links objects. The dimension
//   of the matrices ("Row" and "Col") must be set prior to using
//   the function.
//
//   Formulation
//
//---------------------------------------------------------------------
{
// Input parameters
      const Links &Before;    // Source of antecedent clause data
      const Links &Toggles;   // Source of consequent clause data
      int Min_Row;            // Row offset
      int Min_Col             // Column offset
}
{
// Local variables
      int Num;                // Number of rows and columns in Hanan grid
      int i;                  // Row index
      int j;                  // Column index
      int k;                  // Row index (with offset)
      int l;                  // Column index (with offset)

   // Get number of rows and columns in matrix
   Num = Before.Get_Num();

   for (i = 0, k = Min_Row; i < Rows; ++i, ++k)
      for (j = 0, l = Min_Col; j < Cols; ++j, ++l) {

         // Copy data from east matrices
         // If out of bounds set to zero
         if ((k < 0) || (l < 0) || (k >= Num) || (l >= Num - 1)) {
            Ante_E[i][j] = 0;
            Conse_E[i][j] = 0;

         // Otherwise copy from source
         } else {
            Ante_E[i][j] = Before.Get_East(k, l);
            Conse_E[i][j] = Toggles.Get_East(k, l);
         }

         // Copy data from south matrices
         // If out of bounds set to zero
         if ((k < 0) || (l < 0) || (k >= Num - 1) || (l >= Num)) {
```

```
            Ante_S[i][j] = 0;
            Conse_S[i][j] = 0;

         // Otherwise copy from source
         } else {
            Ante_S[i][j] = Before.Get_South(k, l);
            Conse_S[i][j] = Toggles.Get_South(k, l);
         }
      }
   return;
}

void Rule::Derive
//-------------------------------------------------------------------
//
//   This function sets the valuse of the clause matrices of the
//   rule.  Data are obtained from a links object and a clump object.
//   The clump object may contain the consequent clause for many
//   rules.  "Clump_Num" is used to specify a single rule.  The
//   dimension of the matrices ("Row" and "Col") must be set prior
//   to using the function.
//
//   Formulation
//
//-------------------------------------------------------------------
(
// Input parameters
      const Links &Before,    // Source of antecedent clause data
      const Clump &Toggles,   // Source of consequent clause data
      int Min_Row,            // Row offset
      int Min_Col,            // Column offset
      int Clump_Num           // Clump number of rule
)
{
// Local variables
      int Num;                // Number of rows and columns in Hanan grid
      int i;                  // Row index
      int j;                  // Column index
      int k;                  // Row index (with offset)
      int l;                  // Column index (with offset)

   // Get number of rows and columns in matrix
   Num = Before.Get_Num();

   for (i = 0, k = Min_Row; i < Rows; ++i, ++k)
      for (j = 0, l = Min_Col; j < Cols; ++j, ++l) {

         // Copy data from east matrices
         // If out of bounds set to zero
         if ((k < 0) || (l < 0) || (k >= Num) || (l >= Num - 1)) {
            Ante_E[i][j] = 0;
            Conse_E[i][j] = 0;

         // Otherwise copy from source
         } else {
            Ante_E[i][j] = Before.Get_East(k, l);
            Conse_E[i][j] = Toggles.Get_East(k, l, Clump_Num);
         }

         // Copy data from south matrices
         // If out of bounds set to zero
         if ((k < 0) || (l < 0) || (k >= Num - 1) || (l >= Num)) {
            Ante_S[i][j] = 0;
            Conse_S[i][j] = 0;

         // Otherwise copy from source
         } else {
            Ante_S[i][j] = Before.Get_South(k, l);
            Conse_S[i][j] = Toggles.Get_South(k, l, Clump_Num);
         }
      }
   return;
}

void Rule::Copy90
//-------------------------------------------------------------------
//
//   This function transposes the data in the rule specified as the
//   input parameter through a 90 degree clockwise rotation.
//
```

```
//   Formulates rules
//
//-----------------------------------------------------------------------
(
// Input parameters
      const Rule *New_Rule    // Pointer to rule to be rotated
)
{
// Local variables
      int i;                  // Index for rows and columns
      int j;                  // Index for rows and columns
      int l;                  // Index for rows and columns

//    Ante_E[0][0] = New_Rule->Ante_S[Cols - 1][0];
//    Ante_E[0][1] = New_Rule->Ante_S[Cols - 2][0];
//    ...
//    Ante_E[0][Cols - 1] = New_Rule->Ante_S[0][0];

   for (i = 0, l = Rows; i < Rows; ++i)
      for (j = 0, l = Cols - 1; j < Cols; ++j, --l) {
         Ante_E[i][j] = New_Rule->Ante_S[l][i];
         Conse_E[i][j] = New_Rule->Conse_S[l][i];
      }

//    Ante_S[0][0] = 0;
//    Ante_S[0][1] = New_Rule->Ante_E[Cols - 1][0];
//
//    Ante_S[0][Cols - 1] = New_Rule->Ante_E[1][0];

   for (i = 0; i < Rows; ++i) {
      Sinks[i][0] = 0;
      Ante_S[i][0] = 0;
      Conse_S[i][0] = 0;
      for (j = 1, l = Cols - 1; j < Cols; ++j, --l) {
         Sinks[i][j] = New_Rule->Sinks[l][i];
         Ante_S[i][j] = New_Rule->Ante_E[l][i];
         Conse_S[i][j] = New_Rule->Conse_E[l][i];
      }
   }
   return;
}

void Rule::Copy180
//-----------------------------------------------------------------------
//
//    This function transposes the data in the rule specified as the
//    input parameter through a 180 degree clockwise rotation.
//
//    Formulates rules
//
//-----------------------------------------------------------------------
(
// Input parameters
      const Rule *New_Rule    // Pointer to rule to be rotated
)
{
// Local variables
      int i;                  // Index for rows and columns
      int j;                  // Index for rows and columns
      int k;                  // Index for rows and columns
      int l;                  // Index for rows and columns

//    Ante_E[0][0] = 0;
//    Ante_E[0][1] = 0;
//    ...
//    Ante_E[0][Cols - 1] = 0;

//    Ante_E[1][0] = New_Rule->Ante_E[Row - 1][Cols - 1];
//    Ante_E[1][1] = New_Rule->Ante_E[Row - 1][Cols - 2];
//    ...
//    Ante_E[1][Cols - 1] = New_Rule->Ante_E[Row - 1][0];

   for (j = 0; j < Cols; ++j) {
      Ante_E[0][j] = 0;
      Conse_E[0][j] = 0;
   }
   for (i = 1, k = Rows - 1; i < Rows; ++i, --k)
      for (j = 0, l = Cols - 1; j < Cols; ++j, --l) {
         Ante_E[i][j] = New_Rule->Ante_E[k][l];
         Conse_E[i][j] = New_Rule->Conse_E[k][l];
```

```
      }

//   Ante_S[0][0] = 0;
//   Ante_S[0][1] = New_Rule->Ante_S[Row - 1][Cols - 1];
//
//   Ante_S[0][Cols - 1] = New_Rule->Ante_S[Row - 1][1];

   for (i = 0, k = Rows - 1; i < Rows; ++i, --k) {
      Ante_S[i][0] = 0;
      Conse_S[i][0] = 0;
      for (j = 1, l = Cols - 1; j < Cols; ++j, --l) {
         Ante_S[i][j] = New_Rule->Ante_S[k][l];
         Conse_S[i][j] = New_Rule->Conse_S[k][l];
      }
   }
   for (j = 0; j < Cols; ++j)
      Sinks[0][j] = 0;
   for (i = 1, k = Rows - 1; i < Rows; ++i, --k) {
      Sinks[i][0] = 0;
      for (j = 1, l = Cols - 1; j < Cols; ++j, --l)
         Sinks[i][j] = New_Rule->Sinks[k][l];
   }
   return;
}

void Rule::Copy270
//---------------------------------------------------------------------
//
//   This function transposes the data in the rule specified as the
//   input parameter through a 270 degree clockwise rotation
//
//   Formulates rules
//
//---------------------------------------------------------------------
(
// Input parameters
      const Rule *New_Rule   // Pointer to rule to be rotated
)
{
// Local variables
      int i;                 // Index for rows and columns
      int j;                 // Index for rows and columns
      int k;                 // Index for rows and columns

//   Ante_E[0][0] = 0;
//   Ante_E[0][1] = 0;
//
//   Ante_E[0][Cols - 1] = 0;

//   Ante_E[1][0] = New_Rule->Ante_S[0][Rows - 1];
//   Ante_E[1][1] = New_Rule->Ante_S[1][Rows - 1];
//
//   Ante_E[1][Cols - 1] = New_Rule->Ante_S[Cols - 1][Rows - 1];

   for (j = 0; j < Cols; ++j) {
      Ante_E[0][j] = 0;
      Conse_E[0][j] = 0;
   }
   for (i = 1, k = Rows - 1; i < Rows; ++i, --k)
      for (j = 0; j < Cols; ++j) {
         Ante_E[i][j] = New_Rule->Ante_S[j][k];
         Conse_E[i][j] = New_Rule->Conse_S[j][k];
      }

//   Ante_S[0][0] = New_Rule->Ante_E[0][Row - 1];
//   Ante_S[0][1] = New_Rule->Ante_E[1][Rows - 1];
//
//   Ante_S[0][Cols - 1] = New_Rule->Ante_E[Cols - 1][Row - 1];

   for (i = 0, k = Rows - 1; i < Rows; ++i, --k)
      for (j = 0; j < Cols; ++j) {
         Ante_S[i][j] = New_Rule->Ante_E[j][k];
         Conse_S[i][j] = New_Rule->Conse_E[j][k];
      }
   for (j = 0; j < Cols; ++j)
      Sinks[0][j] = 0;
   for (i = 1, k = Rows - 1; i < Rows; ++i, --k)
      for (j = 0; j < Cols; ++j)
         Sinks[i][j] = New_Rule->Sinks[j][k];
   return;
```

```
}

void Rule  Reflect
//---------------------------------------------------------------------
//
//   This function reflects the data in the rule specified as the
//   input parameter about a north south line to the east of the
//   matrices.
//
//   Formulates rules
//
//---------------------------------------------------------------------
(
// Input parameters
      const Rule *New_Rule    // Pointer to rule to be rotated
)
{
// Local variables
      int i;                  // Index for rows and columns
      int j;                  // Index for rows and columns
      int l;                  // Index for rows and columns

//    Ante_E[0][0] = New_Rule->Ante_E[0][Cols - 1];
//    Ante_E[0][1] = New_Rule->Ante_E[0][Cols - 2];
//     ...
//    Ante_E[0][Cols - 1] = New_Rule->Ante_E[0][0];

   for (i = 0; i < Rows; ++i)
      for (j = 0, l = Cols - 1; j < Cols; ++j, --l) {
         Ante_E[i][j] = New_Rule->Ante_E[i][l];
         Conse_E[i][j] = New_Rule->Conse_E[i][l];
      }

//    Ante_S[0][0] = 0;
//    Ante_S[0][1] = New_Rule->Ante_S[0][Cols - 1];
//
//    Ante_S[0][Cols - 1] = New_Rule->Ante_S[0][1];

   for (i = 0; i < Rows; ++i) {
      Sinks[i][0] = 0;
      Ante_S[i][0] = 0;
      Conse_S[i][0] = 0;
      for (j = 1, l = Cols - 1; j < Cols; ++j, --l) {
         Sinks[i][j] = New_Rule->Sinks[i][l];
         Ante_S[i][j] = New_Rule->Ante_S[i][l];
         Conse_S[i][j] = New_Rule->Conse_S[i][l];
      }
   }
   return;
}

void Rule  Set_Sinks()
//---------------------------------------------------------------------
//
//   This function applies the rule at each node and calculates the
//   degree of the node.  If the degree falls below "MIN_DERGREE" (2)
//   the node is flagged.
//
//   Formulation
//
//---------------------------------------------------------------------
{
// local variables
      int Deg;                // Degree of a node
      int i;                  // Row index
      int j;                  // Column index

   for (i = 1; i < Rows; ++i)
      for (j = 1; j < Cols; ++j) {
         Deg =
            (Ante_S[i - 1][j] * Conse_S[i - 1][j]) +
            (Ante_S[i][j] * Conse_S[i][j]) +
            (Ante_E[i][j - 1] * Conse_E[i][j - 1]) +
            (Ante_E[i][j] * Conse_E[i][j]);
         if (Deg >= MIN_DEGREE) {
            Sinks[i][j] = 1;
         }
      }
   return;
}
```

```
int Rule::Rule_Cmp
//---------------------------------------------------------------------
//
//   This function compares this rule with another to determine
//   if the rules are the same.
//
//   Formulates rules
//
//---------------------------------------------------------------------
(
// Input parameters
      const Rule *New_Rule   // Pointer to another rule
)
const
{
// Local variables
      int Same;              // Flag (1 = rules are the same)
      int R1;                // Temporary (number of rows minus 1)
      int C1;                // Temporary (number of rows minus 1)
      int i;                 // Row index
      int j;                 // Column index

   // Assume both rules are the same
   Same = 1;

   // Compare size of matrices
   // Continue comparison if the same size
   if (Rows == New_Rule->Rows && Cols == New_Rule->Cols) {

      // Calculate values of temporary variables
      R1 = Rows - 1;
      C1 = Cols - 1;

      // Check patterns in consequent east links matrices
      for (i = 1; i < Rows && Same; ++i)
         for (j = 1; j < C1 && Same; ++j)
            if (Conse_E[i][j] != New_Rule->Conse_E[i][j])
               Same = 0;

      // Check patterns in consequent south links matrices
      for (i = 1; i < R1 && Same; ++i)
         for (j = 1; j < Cols && Same; ++j)
            if (Conse_S[i][j] != New_Rule->Conse_S[i][j])
               Same = 0;

      // If size and consequent clauses match check antecedent
      // clauses
      if (Same)
         Same = Cmp_East(New_Rule);
      if (Same)
         Same = Cmp_South(New_Rule);

   // If matrices are not the same size rules are different
   } else
     Same = 0;

   return Same;
}

int Rule  Instantiate
//---------------------------------------------------------------------
//
//   This function returns a value of 1 if links of the antecedent
//   clause match the pattern in the tester/display object at the
//   offset location. Only the links and the adjacent links in the
//   pattern specified by the consequent clause are tested.
//
//   Applies rules
//
//---------------------------------------------------------------------
(
// Input parameters
      const Loop_Base &Tester,
                             // Reference to tester/display object
      int m,                 // Row offset
      int n                  // Column offset
)
const
{
```

```
// Local variables
      int Same;                  // Flag (1 = patterns match)
      int i;                     // Row index
      int j;                     // Column index
      int k;                     // Row index with offset
      int l;                     // Column index with offset

   Same = 1;
   for (i = 1, k = m + 1; i < Rows && Same; ++i, ++k)
      for (j = 1, l = n + 1; j < Cols && Same; ++j, ++l) {
         if (Conse_E[i][j])
            Same = Verify_East(Tester, i, j, k, l);
         if (Conse_S[i][j])
            Same = Same && Verify_South(Tester, i, j, k, l);
      }
   return Same;
}

double Rule::Calc_Priority
//---------------------------------------------------------------------
//
//   This function returns the priority of a rule. If a link is
//   set in the consequent clause the rule changes the links value
//   If the link is also set in the antecedent clause the rule
//   removes the link and therefore the priority is positive. If
//   the link is set in the consequent clause but not set in the
//   antecedent clause the rule adds the link and the priority is
//   negative.
//
//   Formulation and use
//
//---------------------------------------------------------------------
(
// Input parameters
      const Loop_Base &Tester,
                                 // Pointer to tester/display object
      int m,                     // Row offset
      int n                      // Column offset
)
{
// Local variables
      double Priority;           // Calculated priority
      int i;                     // Row index
      int j;                     // Column index

   Priority = 0.0;
   for (i = 0; i < Rows; ++i)
      for (j = 0; j < Cols; ++j) {
         if (Conse_E[i][j]) {
            if (Ante_E[i][j]) {
               Priority += Tester.Get_Prob()->Get_DX(n + j);
            } else {
               Priority -= Tester.Get_Prob()->Get_DX(n + j);
            }
         }
         if (Conse_S[i][j]) {
            if (Ante_S[i][j]) {
               Priority += Tester.Get_Prob()->Get_DY(m + i);
            } else {
               Priority -= Tester.Get_Prob()->Get_DY(m + i);
            }
         }
      }
   return Priority;
}

void Rule::Fire
//---------------------------------------------------------------------
//
//   This function fires the rule. The contents of the tester/display
//   object and the solution source object are changed according
//   to the consequent clause of rule.
//
//   Applies rules
//
//---------------------------------------------------------------------
(
// Input parameters
      Loop_Base &Tester,         // Reference to tester/display object
      FuzzLink &Sol,             // Reference to solution object
```

```
      int m,                  // Row offset
      int n                   // Column offset
)
{
// Local variables
      int OnOff;              // Flag (1 = link is to be turned on
                              //      -1 = link is to be turned off)
      int i;                  // Row index
      int j;                  // Column index
      int k;                  // Row index with offset
      int l;                  // Column index with offset

   // Check all elements in the consequent clause matrices
   // and update the solution where necessary
   for (i = 1, k = m + 1; i < Rows; ++i, ++k)
      for (j = 1, l = n + 1; j < Cols, ++j, ++l) {

         // Check to see if east link should change
         if (Conse_E[i][j]) {

            // Toggle link and determine if the link was turned
            // on or off
            OnOff = Tester.Toggle_Link(k, l, Loop_Mod, East_El);

            // Update the solution
            switch (OnOff) {
               case 1:
                  Sol.New_Link_On(k, l, 0);
                  break;
               case -1:
                  Sol.New_Link_Off(k, l, 0);
               }
         }

         // Check to see if south link should change
         if (Conse_S[i][j]) {

            // Toggle link and determine if the link was turned
            // on or off
            OnOff = Tester.Toggle_Link(k, l, Loop_Mod, South_El);

            // Update the solution
            switch (OnOff) {
               case 1:
                  Sol.New_Link_On(k, l, 1);
                  break;
               case -1:
                  Sol.New_Link_Off(k, l, 1);
               }
         }
      }

   // Update the "used" counter
   Use();
   return;
}

void Rule_Base::Attach_Rule
//---------------------------------------------------------------------
//
//   This function connects the supplied rule to the rule base
//   The rule base is a doubly linked list
//
//   Formulates and applies rules
//
//---------------------------------------------------------------------
{
// Input parameters
      Rule *New_Rule          // Pointer to a rule
)
{
   if (Tail == NULL) {
      Head = New_Rule;
      New_Rule->Put_Last(NULL);
   } else {
      Tail->Put_Next(New_Rule);
      New_Rule->Put_Last(Tail);
   }
   New_Rule->Put_Next(NULL);
   Tail = New_Rule;
```

```
   ++Num_Rules;
   return;
}

int Rule_Base::Is_KnownP
//---------------------------------------------------------------------
//
//   This function returns 1 if the rule supplied as an input
//   parameter is already in the rule base.
//
//   Formulates and applies rules
//
//---------------------------------------------------------------------
(
// Input paramters
      const Rule *New_Rule    // Pointer to new rule
)
const
{
   // Local variables
      int Known;              // Flag (1 - the rule has been learned
                              //       previously)
      Rule *Known_Rule;       // Pointer to a rule in the rule base

   Known_Rule = Tail;
   Known = 0;
   while (Known_Rule != NULL && !Known) {
      Known = Known_Rule->Rule_Cmp(New_Rule);
      Known_Rule = Known_Rule->Get_Last();
   }
   return Known;
}

Rule_Base::Rule_Base
//---------------------------------------------------------------------
//
//   This constructor initializes the pointers to the linked lists.
//
//   Formulates and applies rules
//
//---------------------------------------------------------------------
(
// Input parameters
      Loop_Base *New_Tester   // Pointer to tester/display object
)
{
   Tester = New_Tester;
   Head = Tail = NULL;
   Type_Counter = 0;
   Read_Type = 0;
   T_Array = NULL;
   Trig_Set = NULL;
   Next_Rule = -1;
   Num_Rules = 0L;
   Rules_Written = 0L;
   Rules_Read = 0L;
   Max_Rules = 0L;
}

Rule_Base::~Rule_Base()
//---------------------------------------------------------------------
//
//   This destructor frees space allocated to the rule base.
//
//   Formulates and applies rules
//
//---------------------------------------------------------------------
{
   Empty();
}

void Rule_Base::Collect_Counters()
//---------------------------------------------------------------------
//
//   This function collects all the statistics on a category of
//   rules. The collection process consists of summing the counters
//   and writes the sums to all rules of the category.
//
//   Formulates and applies rules
//
```

```
//-------------------------------------------------------------------
{
// Local variables
      Rule *Current_Rule.     // Pointer to a rule in the rule base
      Rule *Next_Rule:        // Pointer to a rule in the rule base
      Rule *Start:            // First rule of a new type
      Rule *End:              // First rule of next type
      int Rule_Type:          // Type number of rule
      int Used:               // Running total of counter value
      int Dumped.             // Running total of counter value
      int Failed:             // Running total of counter value

   Current_Rule = Head.
   Rule_Type = 0:
   while (Current_Rule != NULL) {

      // If a new rule type is encountered
      if (Rule_Type != Current_Rule->Get_Category()) {

         // Update the value of "Rule_Type"
         Rule_Type = Current_Rule->Get_Category().

         // Record position of the first rule
         Start = Next_Rule = Current_Rule.

         // Find the start of the next rule type
         End = NULL.
         while (Next_Rule != NULL) {
            if (Rule_Type != Next_Rule->Get_Category()) {
               End = Next_Rule:
               Next_Rule = NULL:
            } else
               Next_Rule = Next_Rule->Get_Next().
         }

         // Total counter values
         Used = Dumped = Failed = 0:
         Next_Rule = Start:
         while (Next_Rule != End) {
            Used += Next_Rule->Get_Used().
            Dumped += Next_Rule->Get_Dumped():
            Failed += Next_Rule->Get_Failed():
            Next_Rule = Next_Rule->Get_Next():
         }

         // Write totals back to rules
         Next_Rule = Start.
         while (Next_Rule != End) {
            Next_Rule->Put_Used(Used):
            Next_Rule->Put_Dumped(Dumped):
            Next_Rule->Put_Failed(Failed):
            Next_Rule = Next_Rule->Get_Next().
         }

      }
      Current_Rule = Current_Rule->Get_Next().
   }
   return.
}

void Rule_Base::Set_All_Counters()
//-------------------------------------------------------------------
//
//   This function sets the counters in all rules so that the rules
//   will be saved even when forgetting is fully enabled. The "Used"
//   counters are all set to 1. The "Dumped" and "Failed" counters
//   are all set to 0. This function overrides forgetting.
//
//   Formulates and applies rules
//
//-------------------------------------------------------------------
{
// Local variables
      Rule *Current_Rule.     // Pointer to a rule in the rule base

   Current_Rule = Head:
   while (Current_Rule != NULL) {
      Current_Rule->Put_Used(1):
      Current_Rule->Put_Dumped(0):
      Current_Rule->Put_Failed(0):
```

```
      Current_Rule = Current_Rule->Get_Next();
   }
   return;
}

void Rule_Base::Print
//---------------------------------------------------------------------
//
//   This function prints the rule base to the output stream.
//
//   Formulates rules
//
//---------------------------------------------------------------------
(
// Input parameters
      ostream &Out,            // Output stream
      Dispose_Spec Forget      // Specifies disposal of poor rules
)
{
// Local variables
      Rule *Next;              // Pointer to rule

   Rules_Written = 0L;
   Next = Head;
   while (Next != NULL) {
      if (Next->Print(Out, Forget))
         ++Rules_Written;
      Next = Next->Get_Next();
   }
   return;
}

void Rule_Base::Fprint
//---------------------------------------------------------------------
//
//   This function prints the rule base to a file.
//
//   Formulates rules
//
//---------------------------------------------------------------------
(
// Input parameters
      char *OutFile,           // Output file
      Dispose_Spec Forget      // Specifies disposal of poor rules
)
{
   ofstream Out(OutFile);
      Print(Out, Forget);
   Out.close();
   return;
}

void Rule_Base::Add
//---------------------------------------------------------------------
//
//   This function reads rules from a file.
//
//   Applies rules
//
//---------------------------------------------------------------------
(
// Input parameters
      char *InFile,            // File name
      Dispose_Spec Forget      // Specifies disposal of poor rules
)
{
// Local variables
      Rule *New_Rule;          // Pointer to newly read rule
      int Rows;                // Number of rows in the rule matrices
      int Cols;                // Number of columns in the rule matrices

   Rules_Read = 0L;
   Read_Type = 0;
   ifstream In(InFile);
   if (In) {
      In.seekg(5L, ios::beg);
      In >> Rows;
      while (!In.eof()) {
         In.seekg(7L, ios::cur);
         In >> Cols;
```

```
            New_Rule = new Rule(Rows, Cols);
            if (New_Rule->Read(In, Forget)) {

               // Check to see if rule exists already
               if (Is_KnownP(New_Rule)) {
                  delete New_Rule;
               } else {

                  // If rule type has changed update "Read_Type"
                  // and "Type_Counter"
                  if (Read_Type != New_Rule->Get_Category()) {
                     Read_Type = New_Rule->Get_Category();
                     ++Type_Counter;
                  }

                  New_Rule->Put_Category(Type_Counter);
                  Attach_Rule(New_Rule);
                  ++Rules_Read;
               }
            } else
               delete New_Rule;
            In.seekg(1lL, ios::cur);
            In >> Rows;
         }
         In.close();
      }
      return;
}

void Rule_Base::Add
//---------------------------------------------------------------------
//
//    This function adds a new rule to the rule base. The source of
//    the antecedent clause is a Links object and the source of the
//    consequent clause is a clump object. The boundary of the region
//    of the Hanan grid to be used to formulate the rule matrices is
//    specified by a boundary object. New rules are rotated 90
//    degrees to produce four variants. Then they are mirror reflected
//    and rotated to produce a total of 8 variants. All variants are
//    checked to see if they have been learned previously.
//
//    Formulates rules
//
//---------------------------------------------------------------------
(
// Input parameters
      const Links &Before,    // Antecedent clause source data
      const Clump &Toggles,   // Consequent clause source data
      Boundary Bounds,        // Boundaries of rectangle surrounding data
      int Clump_Num           // Clump number of rule
)
{
// Local variables
      Rule *New_Rule;         // Pointer to new rule
      Rule *Rot;              // Pointer to rotated variant
      Rule *Ref0;             // Pointer to reflected variant
      int Rows;               // Number of rows in rule matrices
      int Cols;               // Number of columns in rule matrices

   // Calculate size of rule matrices
   Rows = Bounds.Max_Row - Bounds.Min_Row + 1;
   Cols = Bounds.Max_Col - Bounds.Min_Col + 1;

   // Allocate space for rule
   New_Rule = new Rule(Rows, Cols);

   // Place contents in rule
   New_Rule->Derive(Before, Toggles,
                    Bounds.Min_Row, Bounds.Min_Col, Clump_Num);

   // Set up sinks matrix
   New_Rule->Set_Sinks();

   // Check to see if rule exists already
   if (Is_KnownP(New_Rule))
      delete New_Rule;
   else {
      ++Type_Counter;
      New_Rule->Put_Category(Type_Counter);
      Attach_Rule(New_Rule);
```

```
      // Rotate reflection 90 degrees three times
      Rot = new Rule(Cols, Rows);
      Rot->Copy90(New_Rule);
      if (Is_KnownP(Rot))
         delete Rot;
      else {
         Rot->Put_Category(Type_Counter);
         Attach_Rule(Rot);
      }
      Rot = new Rule(Rows, Cols);
      Rot->Copy180(New_Rule);
      if (Is_KnownP(Rot))
         delete Rot;
      else {
         Rot->Put_Category(Type_Counter);
         Attach_Rule(Rot);
      }
      Rot = new Rule(Cols, Rows);
      Rot->Copy270(New_Rule);
      if (Is_KnownP(Rot))
         delete Rot;
      else {
         Rot->Put_Category(Type_Counter);
         Attach_Rule(Rot);
      }
      Ref0 = new Rule(Rows, Cols);

      // Mirror reflect the rule
      Ref0->Reflect(New_Rule);
      if (Is_KnownP(Ref0))
         delete Ref0;
      else {
         Ref0->Put_Category(Type_Counter);
         Attach_Rule(Ref0);

         // Rotate reflection 90 degrees three times
         Rot = new Rule(Cols, Rows);
         Rot->Copy90(Ref0);
         if (Is_KnownP(Rot))
            delete Rot;
         else {
            Rot->Put_Category(Type_Counter);
            Attach_Rule(Rot);
         }
         Rot = new Rule(Rows, Cols);
         Rot->Copy180(Ref0);
         if (Is_KnownP(Rot))
            delete Rot;
         else {
            Rot->Put_Category(Type_Counter);
            Attach_Rule(Rot);
         }
         Rot = new Rule(Cols, Rows);
         Rot->Copy270(Ref0);
         if (Is_KnownP(Rot))
           delete Rot;
         else {
            Rot->Put_Category(Type_Counter);
            Attach_Rule(Rot);
         }
      }
   }
   return;
}

int Rule_Base::Apply
//-------------------------------------------------------------------------
//
//   This function performs one iteration of application of a rule
//   if a rule that can be applied is present in the rule base.
//
//   Applies rules
//
//-------------------------------------------------------------------------
(
// Input parameters
     FuzzLink &Sol           // Reference to solution data
)
{
```

```
// Local variables
      int More_Rules;          // Flag (1 = trigger set is empty)
      int Fired;               // Flag (1 = iteration is complete)

   More_Rules = 1;

   // Initiate another iteration of instaniation of rules
   // if the trigger set is empty
   if (Next_Rule == -1)
      Trigger();

   if (Trig_Count > 0) {

      // Assume no rules have been fired
      Fired = 0;
      while (!Fired) {
         if (Next_Rule < Trig_Count) {
            if (T_Array[Next_Rule]->Priority >= ROUND_OFF) {

               // Apply rule if rule re-instaniates
               if (T_Array[Next_Rule]->Triggered->Instantiate("Tester",
                     T_Array[Next_Rule]->Row, T_Array[Next_Rule]->Col)) {
                  T_Array[Next_Rule]->Triggered->Fire("Tester", Sol,
                        T_Array[Next_Rule]->Row, T_Array[Next_Rule]->Col);
                  Fired = 1;
               }
               ++Next_Rule;
            } else {

               // Dump all rules with non-positive priority
               while (Next_Rule < Trig_Count) {
                  T_Array[Next_Rule]->Triggered->Dump();
                  ++Next_Rule;
               }
            }
         } else {

            // The trigger set is empty so end iteration
            Fired = 1;
            Clean_Up();
            More_Rules = 0;
         }
      }
   } else
      More_Rules = 0;
   return More_Rules;
}

void Rule_Base::Trigger()
//---------------------------------------------------------------------
//
//   This function performs one iteration of instantiation of rules
//   It creates a trigger set that is a sorted array of instantiation
//   records
//
//   Applies rules
//
//---------------------------------------------------------------------
{
// Local variables
      Rule *Current;           // Pointer to rule in rule base
      Trig_Rule *Next;         // Pointer to record in the trigger set
      int Num;                 // Number of basic vertices
      int Top_Row;             // Number of row offsets required
      int Top_Col;             // Number of column offsets required
      int i;                   // Row index
      int j;                   // Column index
      Trig_Rule *Cursor;       // Pointer to instantiation record
      int T;                   // Temporary (Trig_Count minus 1)
      Trig_Rule *Temp;         // Temporary storage for record
      int Change;              // Flag (1 = order in array has changed)

   Num = Tester->Get_Num();
   Clean_Up();
   Trig_Count = 0;
   Trig_Set = NULL;
   Current = Head;

   // Search entire rule base
   while(Current != NULL) {
```

```
      Top_Row = Num - Current->Get_Rows();
      Top_Col = Num - Current->Get_Cols();
      for (i = -1; i <= Top_Row; ++i) {
         for (j = -1; j <= Top_Col; ++j) {
            if (Current->Instantiate("Tester", i, j)) {
               Next = new Trig_Rule;
               Next->Triggered = Current;
               Next->Priority = Current->Calc_Priority("Tester", i, j);
               Next->Row = i;
               Next->Col = j;
               Next->Last = Trig_Set;
               Trig_Set = Next;
               ++Trig_Count;
            }
         }
      }
      Current = Current->Get_Next();
   }

   // Create array of records
   T_Array = new Trig_Rule*[Trig_Count];

   // Copy records in linked list to array
   Cursor = Trig_Set;
   for (i = 0; i < Trig_Count && Cursor != NULL; ++i) {
      T_Array[i] = Cursor;
      Cursor = Cursor->Last;
   }

   // Sort records
   T = Trig_Count - 1;
   Change = 1;
   for (i = 0; i < T && Change; ++i) {
      Change = 0;
      for (j = T; j > i; --j)
         if (T_Array[j]->Priority > T_Array[j - 1]->Priority) {
            Temp = T_Array[j - 1]
            T_Array[j - 1] = T_Array[j];
            T_Array[j] = Temp;
            Change = 1;
         }
   }
   if (Trig_Count > 0)
      Next_Rule = 0;
   else
      Next_Rule = -1;
   return;
}

int Rule_Base::Fire2
// ---------------------------------------------------------------------
//
//   This function fires the next rule in the trigger set
//   regardless of the consequences
//
//---------------------------------------------------------------------
(
// Input parameters
      FuzzLink &Sol            // Reference to solution data
)
{
// Local variables
      int Continue;

   if (Next_Rule < Trig_Count) {
      T_Array[Next_Rule]->Triggered->
                  Fire("Tester", Sol,
                        T_Array[Next_Rule]->Row,
                           T_Array[Next_Rule]->Col);
      ++Next_Rule;
      Continue = 1;
   } else {
      Next_Rule = -1;
      Continue = 0;
   }

   return Continue;
}

void Rule_Base::Empty()
```

```
//-------------------------------------------------------------------------
//
//   This function empties the rule base
//
//   Formulates and applies rules
//
//-------------------------------------------------------------------------
{
// Local variables
      Rule *Cursor;          // Pointer to a rule in the rule base
      Rule *Last;            // Additional pointer to rule

   // Empty the trigger set
   Clean_Up();

   // If the rule base is not empty delete all rules
   if (Tail != NULL) {
      Cursor = Tail;
      Last = Cursor->Get_Last();
      delete Cursor;
      while (Last != NULL) {
         Cursor = Last;
         Last = Cursor->Get_Last();
         delete Cursor;
      }
   }

   Tail = NULL;
   Head = NULL;
   Num_Rules = 0L;
   return;
}

void Rule_Base::Clean_Up()
//-------------------------------------------------------------------------
//
//   This function empties the trigger set
//
//   Applies rules
//
//-------------------------------------------------------------------------
{
// Local variables
      Trig_Rule *Last;       // Pointer to record in the trigger set

   if (T_Array != NULL) {
      delete T_Array;
      T_Array = NULL;
   }
   while (Trig_Set != NULL) {
      Last = Trig_Set->Last;
      delete Trig_Set;
      Trig_Set = Last;
   }
   Next_Rule = -1;
   return;
}

void Rule_Base::Rule_Fails()
//-------------------------------------------------------------------------
//
//   This function increments the fail counter of the last rule
//   applied.
//
//   Applies rules
//
//-------------------------------------------------------------------------
{
   T_Array[Next_Rule - 1]->Triggered->Failure();
   return;
}
```

COMM.H

```
//-------------------------------------------------------------------------
//
//   This is the file comm.h
```

```
//
//------------------------------------------------------------------------

// Interface dependencies ------------------------------------------------

#ifndef __RULE_H
#include "rule.h"
#endif

// End interface dependencies --------------------------------------------

#ifndef __COMM_H
#define __COMM_H

class Command {
//------------------------------------------------------------------------
//
//   An object of this class contains the three components of
//   the learning program.  The agitator, production system and
//   the rule formulator are implemented in objects of this class.
//
//   operators
//      Command(char*, int, int, int, int, int)
//
//   functions
//      Setup_Orig(Chaos01)
//      Read_Orig(char*, Chaos01)
//      Write_Orig(char*)
//      Learn(Chaos01, Forget_Spec, Cheat_Spec, SC_Spec, Ag_Spec)
//      Apply_Rules()
//      Cycle()
//      Run(Chaos01)
//      Read_Rules(char*, Dispose_Spec)
//      Write_Rules(char*, Dispose_Spec)
//      Test(ostream&)
//      Test_No_Write()
//
//   interface functions
//      double Get_Current_Cost()
//      long Get_Num_Rules()
//      long Get_Num_Read()
//      long Get_Num_Written()
//
//------------------------------------------------------------------------
public:
   enum Cheat_Spec {          // Request for cheating during agitation

      Use_Cheat,              // Cheating required
      No_Cheat                // Cheating not required
   };

   enum Remove_Spec {         // Request for removal of branched components

      Remove_Branched,        // Branched components should be removed
      No_Remove               // Branched components not removed
   };

   enum Forget_Spec {         // Request for use of forgetting

      Use_Full_Forget,        // Forgetting required
      Partial_Forget,         // Forget only those not used
      No_Forget               // Forgetting not required
   };

   enum SC_Spec {             // Request for use of sieves and clumps

      Use_SC,                 // Sieves and clumps required
      No_SC                   // Sieves and clumps not required
   };

   enum Ag_Spec {             // Request for use of agitator before rules
                              // during learning

      Use_Ag,                 // Agitator is applied before rules
      No_Ag                   // Agitator is not applied before rules
   };

protected:
   Nodes2 *Problem;           // Problem data
   Loop_Mod *Tester;          // Tester display object
```

```
    FuzzLink **Original_Sol;  // Matrix of original starting solutions
    int Start;                // Index to starting solution
    int Num_Sols;             // Number of starting solutions
    int Num_Test;             // Number of additional test solutions
    int Total_Sols;           // Total number of starting solutions
    FuzzLink *Current_Sol;    // Current solution
    FuzzLink *New_Sol;        // Solution used for tests
    FuzzLink *Best_Sol;       // Best solution found
    Links *Before;            // Object used for before snapshot
    Links *After;             // Object used for after snapshot
    Clump *Template;          // Templates used to generate rules
    Links *Temp_Link;         // Alternate source of templates
    Rule_Base *Rules;         // Rule base
    int Num_Cycles;           // Number forgetting cycles
    int Cycle_Length;         // Number of iterations per starting solution

// Implementation functions
    int Agitate(Chaos01&, Remove_Spec, Cheat_Spec);
                              // Agitates solution
    void Modify();            // Interactive modification of solution
    void Check_Paths();       // Modification
    void Disp_Best();         // Displays the best solution
    void Set_Alpha(Chaos01&, Remove_Spec);
                              // Sets threshold of solution to alpha
    void Form_Rules(Chaos01&, Cheat_Spec, SC_Spec);
                              // Creates a rule through random manipulation
    void Apply_1_Set();       // Applies one trigger set of rules
    void Apply_1_Rule();      // Applies one rule in rule base
    void View_Trigger();      // Views rules in the current trigger set
    void Inspect(Clump &Toggles);
                              // Derives rules from a clump template
    int Find_Position(Boundary&, Clump&, int);
                              // Finds boundary of one rule in a clump
                              // template
    int Check_East(Boundary&, Clump&, int);
                              // Finds boundary of rule is east links
                              // matrix of clump template
    int Check_South(Boundary&, Clump&, int);
                              // Finds boundary of rule is south links
                              // matrix of clump template
    void Reset();             // Sets current solution to starting solution
    void Learn_Int(Chaos01&, Forget_Spec, Cheat_Spec, SC_Spec, Ag_Spec);
                              // Interactive autonomous learning

public:
    Command(){}               // Vanilla constructor
    Command(char*, int, int, int, int = 1, int = 1, int = 0);
                              // Actual constructor
    virtual ~Command();       // Destructor
    void Setup_Orig(Chaos01&);
                              // Generates starting solutions
    void Read_Orig(char*, Chaos01&);
                              // Reads starting solutions from a file
    void Write_Orig(char*) const;
                              // Writes starting solutions to a file
    void Learn(Chaos01&, Forget_Spec, Cheat_Spec, SC_Spec, Ag_Spec);
                              // Performs autonomous learning
    void Apply_Rules();       // Applies rules in rule base
    void Cycle();             // Cycles through starting solutions
    void Run(Chaos01&);       // Runs interactive main loop
    void Read_Rule_Base(char*);
                              // Reads in a previous rule base
    void Read_Rules(char*, Dispose_Spec);
                              // Reads rules from a file
    void Write_Rules(char*, Dispose_Spec);
                              // Writes rules to a file
    void Test(ostream&);      // Tests current rule base on test solutions
    void Test_No_Write();     // Dummy test routine

// Interface functions
    double Get_Current_Cost() const { return Current_Sol->Get_Cost(); }
                              // Returns the cost of the current solution
    long Get_Num_Rules() const { return Rules->Get_Num_Rules(); }
                              // Returns the number of rules in the rule
                              // base
    long Get_Num_Read() const { return Rules->Get_Num_Read(); }
                              // Returns the number of rules read in
                              // last read
    long Get_Num_Written() const { return Rules->Get_Num_Written(); }
                              // Returns the number of rules written in
```

```
                                    // last write
};

#endif
```

COMM.CPP

```
//---------------------------------------------------------------------------
//
//   This is the file comm.cpp
//
//---------------------------------------------------------------------------

// Interface dependencies --------------------------------------------------

#ifndef _FSTREAM_H_INCLUDED
#include <fstream.h>
#endif

#ifndef _STDLIB_H_INCLUDED
#include <stdlib.h>
#endif

#ifndef _CONIO_H_INCLUDED
#include <conio.h>
#endif

#ifndef __CUSTIO_H
#include "custio.h"
#endif

#ifndef __SIEVE_H
#include "sieve.h"
#endif

#ifndef __COMM_H
#include "comm.h"
#endif

// End interface dependencies ----------------------------------------------

int Command::Agitate
//---------------------------------------------------------------------------
//
//   This function performs random modifications on the current
//   solution until an improvement occurs or the maximum number of
//   iterations is reached.  If the maximum is reached the best
//   solution, "Best_Sol", is used to produce an improvement.  If
//   "Best_Sol" is an improvement.  If the function succeeds in
//   improving "Current_Sol" a value of 1 is returned, otherwise a
//   value of 0 is returned
//
//---------------------------------------------------------------------------
(
// Input parameters
   Chaos01 &Bit_Source,        // Source of random numbers
   Remove_Spec Remove,         // Specifies removal of branched components
   Cheat_Spec Cheat            // Specifies use of cheating
)
{
// Local variables
      int Mode;                // Temporary storage of display mode
      int Worse;               // Flag (1 = mutated solution is worse than
                               //       current solution)
      int Improve;             // Flag (1 = agitation or cheating successful)
      int i;                   // Counter

   // Assume function is not successful
   Improve = 0;

   // Turn off display
   Mode = Tester->Get_Mode();
   Tester->Set_Mode(0);

   // Mutate the current until an improvement occurs or
   // 100 mutations are performed
   for (Worse = 1, i = 0; i < 100 && Worse; ++i) {
```

```
      // The routine needs this copy procedure for some reason
      *New_Sol = *Current_Sol;
      New_Sol->Mutate(*Current_Sol, Bit_Source, 0.01);
      New_Sol->Best_Alpha();
      if (Remove == Remove_Branched)
         New_Sol->Remove_Redun(Bit_Source);
      Tester->Evaluate();
      // Include new solution if better but allow for
      // round-off error
      if (New_Sol->Get_Cost() + ROUND_OFF < Current_Sol->Get_Cost())
         Worse = 0;
   }

   // If mutation fails to produce improvement cheat
   if (Worse) {
      if (Cheat == Use_Cheat) {
         if (Best_Sol->Get_Cost() + ROUND_OFF  < Current_Sol->Get_Cost()) {
            *Current_Sol = *Best_Sol;
            Improve = 1;
         }
      }

   // Otherwise use mutated solution
   } else {
      *Current_Sol = *New_Sol;
      Improve = 1;
   }

   // Check to see if this solution is better than
   // "Best_Sol"
   if (Current_Sol->Get_Cost() <= Best_Sol->Get_Cost())
      *Best_Sol = *Current_Sol;

   // Display current solution
   Tester->Init();
   *Current_Sol >> *Tester;
   Tester->Set_Mode(Mode);
   Tester->Display();

   // Return 1 if successful
   return Improve;
}

void Command::Modify()
//---------------------------------------------------------------------
//
//   This function allows the user to modify the solution in the
//   tester/display object.  A rule is derived if the user requests
//   using the insert key.
//
//---------------------------------------------------------------------
{
   // Check to make sure current solution is displayed
   if (Tester->Get_Sol() != Current_Sol) {
      Tester->Init();
      *Current_Sol >> *Tester;
      Tester->Display();
   }

   // Check to see that display is in binary state
   if (Tester->Get_B_State() == Loop_Base::Binary) {

      // Take snapshot of solution before modification
      *Before = *Tester;

      // Derive rule if user requests with <insert> key
      if (Current_Sol->Modify()) {
         *After = *Tester;
         Template->XOR(*Before, *After);
         Inspect(*Template);
      }

   } else {
      cerr <<
"\n<Command 01> Error: Attempt to modify non_binary object\n";
      Pause_Key();
      exit(1);
   }
   return;
```

```
}

void Command::Check_Paths()
//---------------------------------------------------------------------
//
//   This function demonstrates the sieve objects' ability to
//   restore equivalent atomic paths. Interactive modification
//   alters the solution. All equivalent atomic paths are restored
//   and the results displayed.
//
//---------------------------------------------------------------------
{
// Local variables
      Sieve *BeforeS;          // Pointer to "before" snapshot
      Sieve *AfterS;           // Pointer to "after" snapshot

   // Take snapshot of current solution
   BeforeS = new Sieve(*Problem, *Tester);

   // Perform interactive modification of the solution
   if (Tester->Modify()) {

      // Take snapshot after modification
      AfterS = new Sieve(*Problem, *Tester);

      // Restore all equivalent atomic paths
      AfterS->Match_All(*BeforeS);

      // Separate out templates
      *BeforeS >> *Before;
      *AfterS >> *After;
      Template->XOR(*Before, *After);
      Template->Sift();

      // Derive rules
      Inspect(*Template);

      // Display the result with restored equivalent paths
      Tester->Init();
      *After >> *Tester;
      Tester->Evaluate();
      Tester->Display();
      Pause_Key();

      // Clean up
      delete AfterS;
   }

   // Display solution prior to modification
   Tester->Init();
   *Current_Sol >> *Tester;
   Tester->Display();

   // Clean up
   delete BeforeS;
   return;
}

void Command::Disp_Best()
//---------------------------------------------------------------------
//
//   This function copies the best solution found to this time,
//   "Best_Sol", to the current solution, "Current_Sol" and
//   displays it.
//
//---------------------------------------------------------------------
{
   *Current_Sol = *Best_Sol;
   Tester->Init();
   *Current_Sol >> *Tester;
   Tester->Display();
   return;
}

void Command::Set_Alpha
//---------------------------------------------------------------------
//
//   This function sets the threshold of "Current_Sol" to the alpha
//   value and removes the branched components.
//
```

```
//---------------------------------------------------------------------
(
// Input parameters
      Chaos01 &Bit_Source.    // Source of random numbers
      Remove_Spec Remove      // Specifies removal of branched components
)
{
// Local variables
      int Mode.               // Temporary storage of display mode

   // Turn off the display mode
   Mode = Tester->Get_Mode().
   Tester->Set_Mode(0).

   // Set threshold to alpha value
   Current_Sol->Best_Alpha().

   // Remove all branched components if required
   if (Remove == Remove_Branched)
      Current_Sol->Remove_Redun(Bit_Source).

   // Evaluate the solution
   Tester->Evaluate().

   // Display the results
   Tester->Set_Mode(Mode).
   Tester->Display().
   return.
}

void Command::Form_Rules
//---------------------------------------------------------------------
//
//   This function derives a new rule using the agitator. Sieve
//   objects are used to restore any equivalent atomic paths and
//   a clump object, "Template", is used to separate changes into
//   individual rules. However, if the parameter "SC" equals 0
//   no sieves and clumps are used.
//
//---------------------------------------------------------------------
(
// Input parameters
      Chaos01 &Bit_Source.    // Source of random numbers
      Cheat_Spec Cheat.       // Specifies use of cheating
      SC_Spec SC              // Specifies use of sieves and clumps
)
{
// Local variables
      Sieve *BeforeS.         // Pointer to "before" snapshot
      Sieve *AfterS.          // Pointer to "after" snapshot

   // Take snapshot of solution before agitation
   BeforeS = new Sieve(*Problem, *Tester).

   // Apply agitator
   Agitate(Bit_Source, Remove_Branched, Cheat).

   // Take snapshot after agitation
   AfterS = new Sieve(*Problem, *Tester).

   // Restore all equivalent atomic paths
   if (SC == Use_SC)
        AfterS->Match_All(*BeforeS).

   // Derive templates
   *BeforeS >> *Before.
   *AfterS >> *After.
   Template->XOR(*Before, *After).
   if (SC == Use_SC)
      Template->Sift().

   // Derive rules
   Inspect(*Template).

   // Clean up
   delete AfterS.
   delete BeforeS.

   return.
}
```

```
void Command::Apply_1_Set()
//---------------------------------------------------------------------
//
//   This function is similar to "Apply_Rules()" except that only
//   one trigger set is applied.
//
//---------------------------------------------------------------------
{
// Local variables
      int Mode.                  // Temporaray storage of display mode

   // Set up variables for application of rules
   Mode = Tester->Get_Mode();
   Tester->Set_Mode(0);
   *New_Sol = *Current_Sol.

   // Apply rules until a trigger set cannot produce
   // a feasible improvement

   // Empty the trigger set
   Rules->Clean_Up().

   // Apply all rules in the trigger set
   while (Rules->Apply(*New_Sol)) {

      // If "New_Sol" is feasible copy it
      // to "Current_Sol"
      if (New_Sol->Test()) {
         *Current_Sol = *New_Sol;

      // Otherwise restore the "New-Sol" to
      // "Current_Sol" and update "Fail" counter
      // of rule
      } else {
         *New_Sol = *Current_Sol;
         *Current_Sol >> *Tester;
         Rules->Rule_Fails();
      }
   }

   // Restore display mode and display solution
   Tester->Set_Mode(Mode);
   *Current_Sol >> *Tester;
   Tester->Display();
   return;
}

void Command: Apply_1_Rule()
//---------------------------------------------------------------------
//
//   This function applies 1 rule in the trigger set
//
//---------------------------------------------------------------------
{
// Local variables
      int Mode;                  // Temporaray storage of display mode

   // Set up variables for application of rules
   Mode = Tester->Get_Mode();
   Tester->Set_Mode(0);
   *New_Sol = *Current_Sol.

   // Apply a rule
   if (Rules->Apply(*New_Sol)) {

      // If "New_Sol" is feasible copy it
      // to "Current_Sol"
      if (New_Sol->Test()) {
         *Current_Sol = *New_Sol.

      // Otherwise restore the "New-Sol" to
      // "Current_Sol" and update "Fail" counter
      // of rule
      } else {
         *New_Sol = *Current_Sol;
         *Current_Sol >> *Tester;
         Rules->Rule_Fails();
      }
   }
```

```
   // Restore display mode and display solution
   Tester->Set_Mode(Mode);
   *Current_Sol >> *Tester;
   Tester->Display();
   return;
}

void Command::View_Trigger()
//---------------------------------------------------------------------
//
//   This function constructs a trigger set and allows the user
//   to view the effect of each rule individually.
//
//---------------------------------------------------------------------
{
   Local variables
      int Mode;                  // Temporaray storage of display mode
      int Continue;              // Flag (1 = continue to apply rules)

   // Set up variables for application of rules
   Mode = Tester->Get_Mode();
   Tester->Set_Mode(0);

   *New_Sol = *Current_Sol;

   // Empty the trigger set
   Rules->Clean_Up();
   Rules->Trigger();

   Continue = 1;
   while(Continue) {
      Continue = Rules->Fire2(*New_Sol);
      if (Continue) {
         Tester->Init();
         *New_Sol >> *Tester;
         Tester->Set_Mode(Mode);
         Tester->Display();
         Tester->Set_Mode(0);
         Pause_Key();
         Tester->Init();
         *Current_Sol >> *Tester;
         *New_Sol = *Current_Sol;
      }
   }

   Rules->Clean_Up();

   Tester->Init();
   *Current_Sol >> *Tester;
   Tester->Set_Mode(Mode);
   Tester->Display();
   return;
}

void Command::Inspect
//---------------------------------------------------------------------
//
//   This function derives a rule for every template in the clumps
//   object "Toggles"
//
//---------------------------------------------------------------------
(
// Input parameters
      Clump &Toggles             // Container of rule templates
)
{
// Local variables
      int Clump_Num;             // Clump identifier
      Boundary Bounds;           // Boundaries of rule template

   Clump_Num = 1;
   while (Find_Position(Bounds, Toggles, Clump_Num)) {
      Rules->Add(*Before, Toggles, Bounds, Clump_Num);
      ++Clump_Num;
   }
   return;
}

int Command::Find_Position
```

```
//-----------------------------------------------------------------------
//
//   This function returns 1 if no links in the clump object
//   "Toggles" are set to the value of "Clump_Num".  If links
//   are found set to this value a value of 0 is returned and the
//   boundaries of the enclosing rectangle are placed in "Bounds".
//
//-----------------------------------------------------------------------
(
// Output parameters
     Boundary &Bounds.        // Boundaries of links set to "Clump_Num"
// Input parameters
     Clump &Toggles.          // Reference to clump object
     int Clump_Num            // Indentifier of clump
)
{
// Local variables
     int Good_Rule;           // Flag (1 = links are set to "Clump_Num")
     int Empty_E;             // Flag (1 = no links set in east matrix)
     int Empty_S;             // Flag (1 = no links set in east matrix)
     Boundary E_Bounds;       // Boundaries of set links in east matrix
     Boundary S_Bounds;       // Boundaries of set links in south matrix

  // Check the east links matrix for links set to "Clump_Num"
  Empty_E = Check_East(E_Bounds, Toggles, Clump_Num);

  // Check the south links matrix for links set to "Clump_Num"
  Empty_S = Check_South(S_Bounds, Toggles, Clump_Num);

  // If links are found set to "Clump_Num" set return value to 1
  // otherwise set return value to 0
  Good_Rule = (!Empty_E || !Empty_S) ? 1 : 0;

  // If links are found set to "Clump_Num" set boundaries
  if (Good_Rule) {

     // If no links were set in the east matrix use the
     // boundaries of the south matrix
     if (Empty_E)
        Bounds = S_Bounds;

     // If no links were set in the south matrix use the
     // boundaries of the east matrix
     else if (Empty_S)
        Bounds = E_Bounds;

     // Determine boundaries based on extremes of both matrices
     else {
        Bounds.Min_Row = (E_Bounds.Min_Row < S_Bounds.Min_Row) ?
                          E_Bounds.Min_Row : S_Bounds.Min_Row;
        Bounds.Min_Col = (E_Bounds.Min_Col < S_Bounds.Min_Col) ?
                          E_Bounds.Min_Col : S_Bounds.Min_Col;
        Bounds.Max_Row = (E_Bounds.Max_Row > S_Bounds.Max_Row) ?
                          E_Bounds.Max_Row : S_Bounds.Max_Row;
        Bounds.Max_Col = (E_Bounds.Max_Col > S_Bounds.Max_Col) ?
                          E_Bounds.Max_Col : S_Bounds.Max_Col;
     }
  }
  return Good_Rule;
}

int Command::Check_East
//-----------------------------------------------------------------------
//
//   This function returns 1 if no links in the east matrix of the
//   clump object "Toggles" are set to the value of "Clump_Num".
//   If links are found set to this value a value of 0 is returned
//   and the boundaries of the enclosing rectangle are placed in
//   "Bounds".
//
//-----------------------------------------------------------------------
(
// Output parameters
     Boundary &Bounds.        // Boundaries of links set to "Clump_Num"
// Input parameters
     Clump &Toggles.          // Reference to clump object
     int Clump_Num            // Indentifier of clump
)
{
// Local variables
```

```
    int Empty;                  // Flag (1 = no toggled links in the
                                //       south matrix)
    int Num_Nodes;              // Number of basic vertices
    int Num_Spaces;             // Temporary (basic vertices minus 1)
    int Found;                  // Flag (1 = toggled link is found)
    int i;                      // Row index
    int j;                      // Column index

// Get number of basic vertices and initialize variables
Num_Nodes = Toggles.Get_Num();
Num_Spaces = Num_Nodes - 1;
Empty = 0;

// Search east matrix for toggled links from north to south
Found = 0;
for (i = 0; i < Num_Nodes && !Found; ++i)
   for (j = 0; j < Num_Spaces && !Found; ++j)
      if (Toggles.Get_East(i, j, Clump_Num)) {
         Found = 1;
         Bounds.Min_Row = i - 1;
      }

// If no toggled links are found set "Empty" flag to 1 and exit
if (!Found)
   Empty = 1;

// Otherwise find other boundaries
else {

   // Search east matrix for toggled links from west to east
   Found = 0;
   for (j = 0; j < Num_Spaces && !Found; ++j)
      for (i = 0; i < Num_Nodes && !Found; ++i)
         if (Toggles.Get_East(i, j, Clump_Num)) {
            Found = 1;
            Bounds.Min_Col = j - 1;
         }

   // Search east matrix for toggled links from south to north
   Found = 0;
   for (i = Num_Nodes - 1; i >= 0 && !Found; --i)
      for (j = Num_Spaces - 1; j >= 0 && !Found; --j)
         if (Toggles.Get_East(i, j, Clump_Num)) {
            Found = 1;
            Bounds.Max_Row = i;
         }

   // Search east matrix for toggled links from east to west
   Found = 0;
   for (j = Num_Spaces - 1; j >= 0 && !Found; --j)
      for (i = Num_Nodes - 1; i >= 0 && !Found; --i)
         if (Toggles.Get_East(i, j, Clump_Num)) {
            Found = 1;
            Bounds.Max_Col = j + 1;
         }
}
return Empty;
}

int Command::Check_South
//-------------------------------------------------------------------
//
//   This function returns 1 if no links in the south matrix of the
//   clump object "Toggles" are set to the value of "Clump_Num"
//   If links are found set to this value a value of 0 is returned
//   and the boundaries of the enclosing rectangle are placed in
//   "Bounds".
//
//-------------------------------------------------------------------
(
// Output parameters
     Boundary &Bounds;      // Boundaries of links set to "Clump_Num"
// Input parameters
     Clump &Toggles;        // Reference to clump object
     int Clump_Num          // Indentifier of clump
)
{
// Local variables
     int Empty;             // Flag (1 = no toggled links in the
                            //       east matrix)
```

```
    int Num_Nodes;          // Number of basic vertices
    int Num_Spaces;         // Temporary (basic vertices minus 1)
    int Found;              // Flag (1 = toggled link is found)
    int i;                  // Row index
    int j;                  // Column index

// Get number of basic vertices and initialize variables
Num_Nodes = Toggles.Get_Num();
Num_Spaces = Num_Nodes - 1;
Empty = 0;

// Search south matrix for toggled links from north to south
Found = 0;
for (i = 0; i < Num_Spaces && !Found; ++i)
   for (j = 0; j < Num_Nodes && !Found; ++j)
      if (Toggles.Get_South(i, j).Clump_Num()) {
         Found = 1;
         Bounds.Min_Row = i - 1;
      }

// If no toggled links are found set "Empty" flag to 1 and exit
if (!Found)
   Empty = 1;

// Otherwise find other boundaries
else {

   // Search south matrix for toggled links from west to east
   Found = 0;
   for (j = 0; j < Num_Nodes && !Found; ++j)
      for (i = 0; i < Num_Spaces && !Found; ++i)
         if (Toggles.Get_South(i, j).Clump_Num()) {
            Found = 1;
            Bounds.Min_Col = j - 1;
      }

   // Search south matrix for toggled links from south to north
   Found = 0;
   for (i = Num_Spaces - 1; i >= 0 && !Found; --i)
      for (j = Num_Nodes - 1; j >= 0 && !Found; --j)
         if (Toggles.Get_South(i, j).Clump_Num()) {
            Found = 1;
            Bounds.Max_Row = i + 1;
      }

   // Search south matrix for toggled links from east to west
   Found = 0;
   for (j = Num_Nodes - 1; j >= 0 && !Found; --j)
      for (i = Num_Spaces - 1; i >= 0 && !Found; --i)
         if (Toggles.Get_South(i, j).Clump_Num()) {
            Found = 1;
            Bounds.Max_Col = j;
         }
   }
   return Empty;
}

void Command::Reset()
//---------------------------------------------------------------------
//
//   This function copies the contents of "Original_Sol" to
//   "Current_Sol".  The tester/display object is updated to display
//   "Current_Sol".
//
//---------------------------------------------------------------------
{
   *Current_Sol = *Original_Sol[Start];
   Tester->Init();
   *Current_Sol >> *Tester;
   Tester->Display();
   return;
}

void Command::Learn_Int
//---------------------------------------------------------------------
//
//   This function is the interactive version of "Learn".
//
//---------------------------------------------------------------------
{
```

```
// Input parameters
      Chaos01 &Bit_Source.    // Source of random numbers
      Forget_Spec Forget.     // Specifies use of forgetting
      Cheat_Spec Cheat.       // Specifies use of cheating
      SC_Spec SC.             // Species the use of sieves and clumps
      Ag_Spec Ag              // Specifies the use of the agitator
)
{
// Local
      int i.                  // Counter
      int j.                  // Counter
      int k.                  // Counter

   // Remove all undesirable rules
   if (!kbhit() && Forget == Use_Full_Forget) {
      Rules->Collect_Counters().
      Rules->Fprint("memory". Dispose_Bad).
      Rules->Update_Max().
      Rules->Empty().
      Rules->Add("memory". No_Dispose).
   }

   // Remove all undesirable rules
   if (!kbhit() && Forget == Partial_Forget) {
      Rules->Collect_Counters().
      Rules->Fprint("memory". Dispose_Unused).
      Rules->Update_Max().
      Rules->Empty().
      Rules->Add("memory". No_Dispose).
   }

   // Perform learning cycles for each starting solution with forgetting
   for (i = 0. i < Num_Cycles && !kbhit(). ++ i) {

      // Initialize learning system
      Start = Num_Sols.
      Cycle().

      // Perform learning cycles for each starting solution
      for (j = 0. j < Num_Sols && !kbhit(). ++ j) {

         // Peform learning cycles with single starting solution
         for (k = 0  k < Cycle_Length && !kbhit(). ++k) {
            Reset()
            if (Ag == Use_Ag)
               Agitate(Bit_Source. Remove_Branched. No_Cheat).
            Apply_Rules().
            Form_Rules(Bit_Source. Cheat. SC).
         }

         // Change starting solution
         Cycle().
      }

   }

   if (kbhit())
      getch().
   cout << '\a'.
   Modify().
   return.
}

Command: Command
//-------------------------------------------------------------------------
//
//   This constructor allocates space for the object.
//
//-------------------------------------------------------------------------
(
      char *NodeFile.         // Problem file name
      int New_Num_Sols.       // Number of starting solutions
      int New_Num_Test.       // Number of additional test solutions
      int New_Num_Cycles.     // Number forgetting cycles
      int New_Cycle_Length.   // Number of iterations per starting solution
      int Disp.               // Display mode
      int Block.              // Flag (1 = do not display improve step)
      int Key_Pause           // Flag (1 = pause display for key)
)
{
```

```
// Local
      int i.                    // Index to starting solutions

   Num_Sols = New_Num_Sols;
   Num_Test = New_Num_Test;
   Total_Sols = Num_Sols + Num_Test;
   Num_Cycles = New_Num_Cycles;
   Cycle_Length = New_Cycle_Length;
   Problem = new Nodes2(NodeFile);
   Tester = new Loop_Mod(*Problem, Disp, Block, Key_Pause);
   Original_Sol = new FuzzLink*[Total_Sols];
   for (i = 0; i < Total_Sols; ++i)
      Original_Sol[i] = new FuzzLink(Tester);
   Current_Sol = new FuzzLink(Tester);
   New_Sol = new FuzzLink(Tester);
   Best_Sol = new FuzzLink(Tester);
   Before = new Links(Problem->Get_Num());
   After = new Links(Problem->Get_Num());
   Template = new Clump(Problem->Get_Num());
   Temp_Link = new Links(Problem->Get_Num());
   Rules = new Rule_Base(Tester);
}

Command::~Command()
//-------------------------------------------------------------------
//
//   This destructor frees space occupied by the object
//
//-------------------------------------------------------------------
{
// Local variables
      int i.                    // Index to original solutions

   delete Rules;
   delete Temp_Link;
   delete Template;
   delete After;
   delete Before;
   delete Best_Sol;
   delete New_Sol;
   delete Current_Sol;
   for (i = Total_Sols - 1; i >= 0; --i)
      delete Original_Sol[i];
   delete Original_Sol;
   delete Tester;
   delete Problem;
}

void Command::Setup_Orig
//-------------------------------------------------------------------
//
//   This function generates the starting solution "Original_Sol"
//   and copies it to the current solution object "Current_Sol"
//
//-------------------------------------------------------------------
(
// Input parameters
      Chaos01 &Bit_Source     // Source of random numbers
)
{
// Local variables
      int Mode;               // Temporaray storage of display mode
      int i.                  // Index to original solutions

   // Turn off display
   Mode = Tester->Get_Mode();
   Tester->Set_Mode(0);

   // Create starting solutions
   for (i = 0; i < Total_Sols; ++i) {
      Original_Sol[i]->Set_Count();
      Original_Sol[i]->Init(Bit_Source);
      Original_Sol[i]->Best_Alpha();
      Original_Sol[i]->Remove_Redun(Bit_Source);
      Tester->Evaluate();
   }

   // Display first original solution
   Start = 0;
   Tester->Init();
```

```
   *Original_Sol[Start] >> *Tester;
   Tester->Set_Mode(Mode);
   Tester->Display();

   // Copy original solution to current solution
   *Current_Sol = *Original_Sol[Start];
   *Best_Sol = *Original_Sol[Start];
   return;
}

void Command::Read_Orig
//---------------------------------------------------------------------------
//
//   This function reads the set of starting solutions from a file
//
//---------------------------------------------------------------------------
(
// Input parameters
      char *Start_File,       // Input file name
      Chaos01 &Bit_Source     // Source of random numbers
)
{
// Local variables
      int New_Num_Sols;       // New number of starting solutions
      int Mode;               // Temporaray storage of display mode
      int i;                  // Index to original solutions

   // Turn off display
   Mode = Tester->Get_Mode();
   Tester->Set_Mode(0);

   // Open file and read in solutions
   ifstream In(Start_File);
   if (Start_File) {

      // Read in number of starting solutions
      In >> New_Num_Sols;
      if (New_Num_Sols == Total_Sols) {

         // Read starting solutions
         for (i = 0; i < Total_Sols; ++i)
            In >> *Original_Sol[i];

         In.close();

         // Set up starting solutions
         for (i = 0; i < Total_Sols; ++i) {
            Original_Sol[i]->Best_Alpha();
            Original_Sol[i]->Remove_Redun(Bit_Source);
            Tester->Evaluate();
         }

         // Display first original solution
         Start = 0;
         Tester->Init();
         *Original_Sol[Start] >> *Tester;
         Tester->Set_Mode(Mode);
         Tester->Display();

         // Copy original solution to current solution
         *Current_Sol = *Original_Sol[Start];
         *Best_Sol = *Original_Sol[Start];
      } else {
         In.close();
         cerr <<
"\n<Command 02> Error: Number of solutions in input file does not match\n";
         Pause_Key();
         exit(1);
      }
   }
   return;
}

void Command::Write_Orig
//---------------------------------------------------------------------------
//
//   This function writes the set of starting solutions to a file
//
//---------------------------------------------------------------------------
{
```

```
// Input parameters
      char *Start_File        // Output file name
)
const
{
// Local variables
      int i;                  // Index to original solutions

   ofstream Out(Start_File);
      Out << Total_Sols << "\n";

      // Write starting solutions
      for (i = 0; i < Total_Sols; ++i)
         Out << *Original_Sol[i];

   Out.close();
   return;
}

void Command::Learn
//---------------------------------------------------------------------
//
//   This function derives rules through iterations of autonomous
//   exploration of the solution space.
//
//---------------------------------------------------------------------
(
// Input parameters
      Chaos01 &Bit_Source,    // Source of random numbers
      Forget_Spec Forget,     // Specifies use of forgetting
      Cheat_Spec Cheat,       // Specifies use of cheating
      SC_Spec SC,             // Specifies use of sieves and clumps
      Ag_Spec Ag              // Specifies use of agitator
)
{
// Local variables
      int i;                  // Counter
      int j;                  // Counter
      int k;                  // Counter

   // Remove all undesirable rules
   if (Forget == Use_Full_Forget) {
      Rules->Collect_Counters();
      Rules->Fprint("memory", Dispose_Bad);
      Rules->Update_Max();
      Rules->Empty();
      Rules->Add("memory", No_Dispose);
   }

   // Remove all undesirable rules
   if (Forget == Partial_Forget) {
      Rules->Collect_Counters();
      Rules->Fprint("memory", Dispose_Unused);
      Rules->Update_Max();
      Rules->Empty();
      Rules->Add("memory", No_Dispose);
   }

   // Perform learning cycles for each starting solution
   for (i = 0; i < Num_Cycles; ++ i) {

      // Initialize learning system
      Start = Num_Sols;
      Cycle();

      // Perform learning cycles for each starting solution with forgetting
      for (j = 0; j < Num_Sols; ++ j) {

         // Peform learning cycles with single starting solution
         for (k = 0; k < Cycle_Length; ++k) {
            Reset();
            if (Ag == Use_Ag)
               Agitate(Bit_Source, Remove_Branched, No_Cheat);
            Apply_Rules();
            Form_Rules(Bit_Source, Cheat, SC);
         }
         // Change starting solution
         Cycle();
      }
   }
```

```
   return.
}

void Command::Apply_Rules()
//---------------------------------------------------------------------
//
//   This function applies rules in the rule base to the solution
//   "New_Sol" which is a copy of "Current_Sol". Each time a rule
//   is applied the feasibility of the resulting solution "New_Sol"
//   is tested. If the solution is feasible the results are copied
//   to "Current_Sol" otherwise the results are discarded. The
//   application of rules terminates when a trigger set is produced
//   that cannot obtain a feasible improvement on the solution
//
//---------------------------------------------------------------------
{
// Local variables
      int Mode.                  // Temporaray storage of display mode
      int Change:                // Flag (1 = The solution changed during the
                                 //       iteration)

   // Set up variables for application of rules
   Mode = Tester->Get_Mode().
   Tester->Set_Mode(0).
   *New_Sol = *Current_Sol.
   Change = 1.

   // Apply rules until a trigger set cannot produce
   // a feasible improvement
   while (Change) {
      Change = 0.

      // Empty the trigger set
      Rules->Clean_Up().

      // Apply all rules in the trigger set
      while (Rules->Apply(*New_Sol)) {

         // If "New_Sol" is feasible copy it
         // to "Current_Sol"
         if (New_Sol->Test()) {
            *Current_Sol = *New_Sol:
            Change = 1.

         // Otherwise restore the "New-Sol" to
         // "Current_Sol" and update "Fail" counter
         // of rule
         } else {
            *New_Sol = *Current_Sol:
            *Current_Sol >> *Tester.
            Rules->Rule_Fails().
         }
      }
   }

   // Check to see if this solution is better than
   // "Best_Sol"
   if (Current_Sol->Get_Cost() <= Best_Sol->Get_Cost())
      *Best_Sol = *Current_Sol.


   // Restore display mode and display solution
   Tester->Set_Mode(Mode).
   *Current_Sol >> *Tester:
   Tester->Display().
   return:
}

void Command::Cycle()
//---------------------------------------------------------------------
//
//   This function increments the index to the set of starting
//   solutions.
//
//---------------------------------------------------------------------
{
   if (++Start >= Num_Sols) Start = 0:
   Reset().
   // *Best_Sol = *Original_Sol[Start]:
```

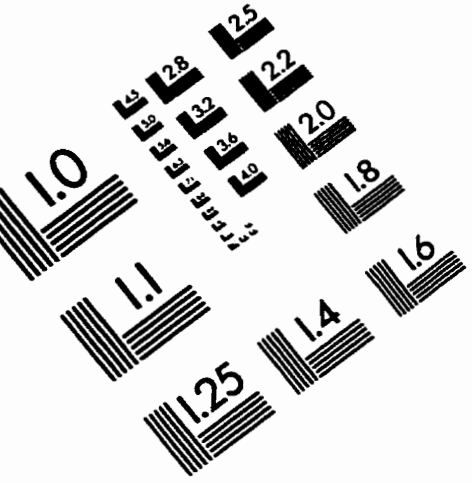
```
   return;
}

void Command: Run
//---------------------------------------------------------------------
//
//   This function is the main loop that processes keyboard input
//   and executes commands
//
//---------------------------------------------------------------------
(
// Input parameters
     Chaos01 &Bits          // Source of random numbers
)
{
// Local variables
     unsigned short Choice; // Stores key pressed by user

   ofstream Out("costs");
      Setup_Orig(Bits);
      *Current_Sol >> *Tester;
      Tester->Display();
      Choice = Pause_Key();

      // Main loop
      while (Choice != ' ') {
         switch (Choice) {
            case '+':
               Current_Sol->Add_Link();
               break;
            case '-':
               Current_Sol->Sub_Link();
               break;
            // Enter
            case 13:
               Current_Sol->Test();
               Pause_Key();
               Tester->Init();
               *Current_Sol >> *Tester;
               Tester->Display();
               break;
            // Home key
            case 199:
               Modify();
               break;
            // F1 key
            case 187:
               Apply_Rules();
               break;
            // Shift F1 key
            case 212:
               Apply_1_Set();
               break;
            // Control F1 key
            case 222:
               Apply_1_Rule();
               break;
            // F2 key
            case 188:
               Read_Rules("rule", Dispose_Bad);
               break;
            // Shift F2 key
            case 213:
               Read_Rules("rule", No_Dispose);
               break;
            // Control F2 key
            case 223:
               Read_Orig("start", Bits);
               break;
            // F3 key
            case 189:
               Write_Rules("rule2", Dispose_Bad);
               break;
            // Shift F3 key
            case 214:
               Write_Rules("rule2", No_Dispose);
               break;
            // Control F3 key
            case 224:
               Write_Orig("start2");
```

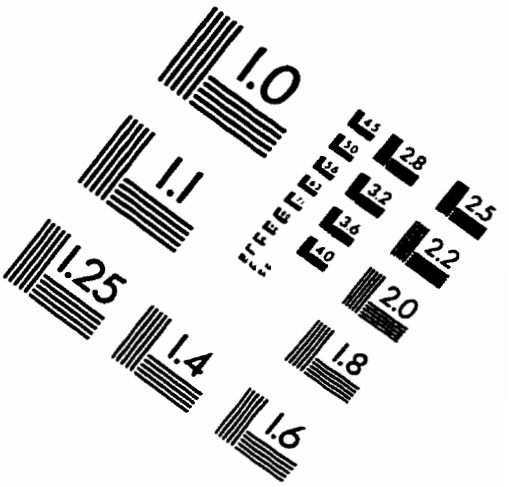
```
      break;
   // F4 key
   case 190:
      Check_Paths();
      break;
   // F5 key
   case 191:
      Agitate(Bits, Remove_Branched, Use_Cheat);
      break;
   // Shift F5 key
   case 216:
      Agitate(Bits, No_Remove, Use_Cheat);
      break;
   // Control F5 key
   case 226:
      Agitate(Bits, Remove_Branched, No_Cheat);
      break;
   // F6 key
   case 192:
      Reset();
      break;
   // Shift F6 key
   case 217:
      Disp_Best();
      break;
   // Control F6 key
   case 227:
      Cycle();
      break;
   // F7 key
   case 193:
      Set_Alpha(Bits, Remove_Branched);
      break;
   // Shift F7 key
   case 218:
      Set_Alpha(Bits, No_Remove);
      break;
   // F8 key
   case 194:
      Current_Sol->Remove_Redun(Bits);
      Tester->Display();
      break;
   // F9 key
   case 195:
      Form_Rules(Bits, Use_Cheat, Use_SC);
      break;
   // Shift F9 key
   case 220:
      Form_Rules(Bits, No_Cheat, Use_SC);
      break;
   // Control F9 key
   case 230:
      Form_Rules(Bits, Use_Cheat, No_SC);
      break;
   // F10 key
   case 196:
      Learn_Int(Bits, Use_Full_Forget, Use_Cheat, Use_SC, Use_Ag);
      break;
   // Shift F10 key
   case 221:
      Learn_Int(Bits, Use_Full_Forget, No_Cheat, Use_SC, Use_Ag);
      break;
   // Control F10 key
   case 231:
      Learn_Int(Bits, Use_Full_Forget, Use_Cheat, No_SC, Use_Ag);
      break;
   // F11 key
   case 261:
      Learn_Int(Bits, No_Forget, Use_Cheat, Use_SC, Use_Ag);
      break;
   // Shift F11 key
   case 263:
      Learn_Int(Bits, No_Forget, Use_Cheat, Use_SC, No_Ag);
      break;
   // F12 key
   case 262:
      View_Trigger();
      break;
   // Shift F12 key
   case 264:
```

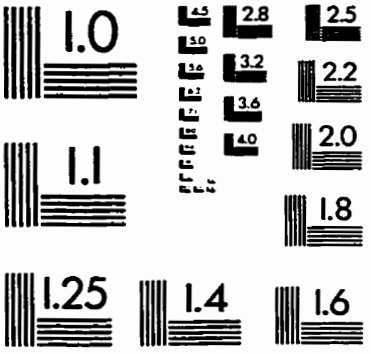
```
            Test(Out);
         }
         Choice = Pause_Key();
      }
   Out.close();
   return;
}

void Command::Read_Rule_Base
//---------------------------------------------------------------------
//
//   This function reads rules from a previous rule base.  The
//   rules are protected from forgetting in the first iteration
//   by the override command "Set_All_Counters()"
//
//---------------------------------------------------------------------
(
// Input parameters
      char *Rule_File       // Rule base file
)
{
   Rules->Add(Rule_File, No_Dispose);
   Rules->Set_All_Counters();
   return;
}

void Command::Read_Rules
//---------------------------------------------------------------------

//   This function reads rules from a file to the rule base.
//
//---------------------------------------------------------------------
(
// Input parameters
      char *Rule_File,      // Name of file containing rules
      Dispose_Spec D_Spec   // Specifies level of forgetting
)
{
   Rules->Add(Rule_File, D_Spec);
   return;
}

void Command::Write_Rules
//---------------------------------------------------------------------
//
//   This function writes rules in the rule base to a file
//
//---------------------------------------------------------------------
(
// Input parameters
      char *Rule_File,
      Dispose_Spec D_Spec
)
{
   Rules->Fprint(Rule_File, D_Spec);
   return;
}

void Command::Test
//---------------------------------------------------------------------
//
//   This function tests the rule base against all the testing
//   soltions and writes the results to a file.
//
//---------------------------------------------------------------------
(
// Input parameters
      ostream &Out          // Output stream
)
{
// Local variables
      int i;                // Index to test solutions

   for (i = 0; i < Total_Sols; ++i) {
      *Current_Sol = *Original_Sol[i];
      Tester->Init();
      *Current_Sol >> *Tester;
      Apply_Rules();
      Out << Current_Sol->Get_Cost() << " ";
   }
```

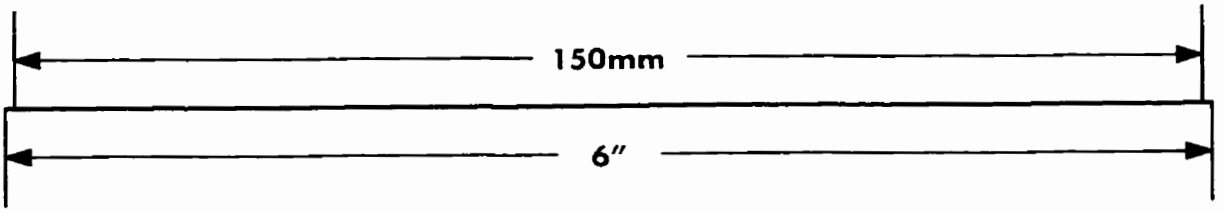
```
   Out << Rules->Get_Num_Read() << " ";
   Out << Rules->Get_Num_Rules() << "\n";
   Reset();
   return;
}

void Command::Test_No_Write()
//-----------------------------------------------------------------------
//
//   This function performs the actions as "Test(ostream&)" but
//   does not write results to a file.
//
//-----------------------------------------------------------------------
{
   Local variables
      int i;                    // Indes to test solutions

   for (i = 0; i < Total_Sols; ++i) {
      *Current_Sol = *Original_Sol[i];
      Tester->Init();
      *Current_Sol >> *Tester;
      Apply_Rules()
   }
   Reset();
   return;
}
```

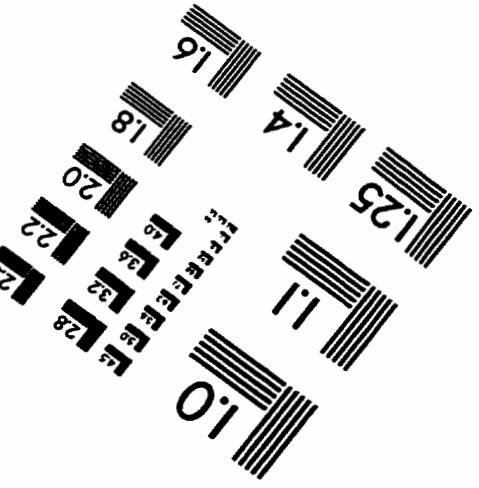
IMAGE EVALUATION
TEST TARGET (QA-3)

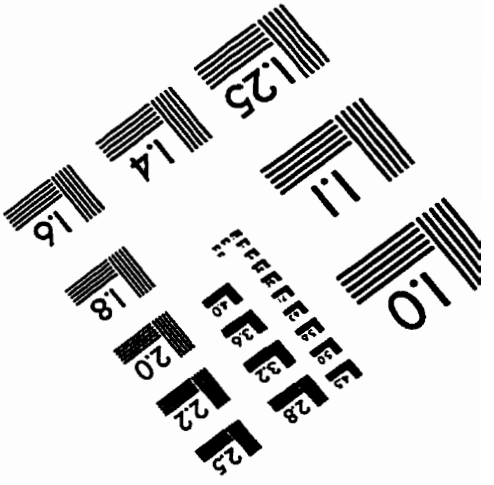












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989