Balanced Networks
and
Their Applications

by

Douglas M. F. Stone

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

MASTER OF SCIENCE

Department of Computer Science
University of Manitoba
Winnipeg, Manitoba.

August, 1991.

ISBN 0-315-77026-0

BALANCED NETWORKS AND THEIR APPLICATIONS

BY

DOUGLAS M.F. STONE

A thesis submitted to the Faculty of Graduate Studies of
the University of Manitoba in partial fulfillment of the requirements
of the degree of

MASTER OF SCIENCE

Abstract

This thesis is concerned mainly with the use of flows in finding f-factors, including maximum matchings, in a graph. A special kind of network called a *balanced network* is presented. Balanced networks have the useful property that every undirected graph G corresponds to a balanced network N, and there is a 1–1 correspondence between a subgraph H of G and a valid flow, called a *balanced flow*, in N.

The theory of balanced networks is developed in Chapter 2. This includes the definition of a balanced edge cut, and the development of the Max-Balanced-Flow–Min-Balanced-Cut theorem, which is analogous to the Max-Flow–Min-Cut theorem for general networks. Chapter 3 gives an algorithm for finding a maximum balanced flow in a balanced network. This algorithm generalises Edmonds' concept of a blossom.

Chapter 4 gives applications of balanced networks, and includes both theoretical and practical uses. On the practical side they can be used to find an f–factor H in a graph G in $O(m\varepsilon)$ time, where m and ε are the number of edges in H and G respectively. Specifically, this means that this algorithm can find a maximum matching in $O(n\varepsilon)$ time, where n is the number of vertices in G. On the theoretical side, once the necessary groundwork has been developed they can be used to give extremely short proofs of known results about f–factors. The most notable of these is the proof of Tutte's theorem, which gives a necessary and sufficient condition for the existence of a perfect matching.

Table of Contents

1. Flows and Matchings — an Overview

1.1. Graphs - Some Basic Definitions

A **graph** G consists of two finite disjoint sets: $V(G)=\{v_1,v_2,...v_n\}$, the **vertices** of G, and $E(G)=\{e_1,e_2,...e_m\}$, the **edges** of G. Every edge e has associated with it an unordered pair of vertices $\{v_i,v_j\}$, called the **endpoints** of e, which define v_j to be **adjacent** (or **joined**) to v_i, and v_i to be adjacent to v_j. We also say that e is **incident** on v_i and v_j, and that e goes between its endpoints. The **degree** of a vertex v is the number of edges incident on v. An edge which has an associated vertex pair $\{v_i,v_i\}$ is called a **loop**, and if there is more than one edge with the same associated vertex pair we call each such edge a **multiple edge**. A graph which has no loops and no multiple edges is called a **simple graph**.

A **directed graph** or **digraph** is very similar to a graph, and differs only in that there is a direction associated with the edges, which are therefore sometimes referred to as **directed edges** or **arcs**. More formally, a digraph D is the same as a graph except that every edge e in E(D) has associated with it some *ordered* pair of vertices (v_i,v_j), which defines v_j to be adjacent to v_i *but not the converse*. v_i is called the **tail** of e and v_j is called the **head**; e is said to be an **out-edge** from v_i and an **in-edge** to v_j, and is said to be **directed** from v_i to v_j. The **out-degree** of a vertex u is the number of directed edges in E(D) for which u is the tail (i.e. edges directed out from u), and the **in-degree** of u is the number of directed edges for which u is the head (i.e. edges directed in to u). A **multiple edge** in D is one of two or more edges in D with the same *ordered* vertex pair. For example, the edges e_1 and e_2 with respective associated ordered vertex pairs (v_i,v_j) and (v_j,v_i) would not be multiple edges in a digraph.

We can take any digraph D, and for every edge e in E(D) ignore the ordering on the vertex pair associated with e. We would then have a graph, which we call the **underlying graph** of D. Thus, provided that there is no superseding definition, any

definition for a graph can also be applied to a digraph D; we just apply the definition to the underlying graph of D.

A **walk** in a graph G is a sequence $W = v_1,e_1,v_2,e_2,...,e_{r-1},v_r$ such that the v_i are (not necessarily distinct) vertices of G and the e_i are (not necessarily distinct) edges, and e_i has associated with it the vertex pair $\{v_i,v_{i+1}\}$. If G is a digraph then e_i may have either (v_i,v_{i+1}) or (v_{i+1},v_i) as its associated vertex pair. If a directed edge e_i on W has the associated ordered vertex pair (v_i,v_{i+1}), then we say that we follow e_i in a forward direction and call it a **forward edge**. Conversely, if e_i has the associated ordered vertex pair (v_{i+1},v_i) then we say that we follow e_i in a backward direction and call it a **backward edge**. The concept of a **directed walk** is defined only for digraphs and is a walk in which every edge e_i is a forward edge. Thus, given a digraph D, a walk in D may contain both forward and backward edges, but a directed walk in D consists entirely of forward edges. For both walks and directed walks we define the **length of a walk** W to be the number of edges on W. The length of the walk $v_1,e_1,v_2,e_2,...,e_{r-1},v_r$ is therefore r-1, and the length of the walk defined by the sequence v_1 is 0.

A **path** P in a graph G is a walk in G on which there is no repeated vertex; that is, the vertices, and thus the edges, on the sequence that defines P are all distinct. All of the definitions for a walk also apply to a path, since a path *is* a walk; thus we are free to talk about directed paths and the length of a path. If we have a path P on which u and v are the first and last vertices respectively (so P begins at u and ends at v) then we say that P is a **uv-path**. Another important type of walk is a cycle. A **cycle** in a graph G is a walk $C = v_1,e_1,v_2,e_2,...,e_{r-1},v_1$ such that $v_i \neq v_j$ whenever $i \neq j$. If we slightly abuse the definition of a path, we may say that a cycle is a path that begins and ends at the same vertex.

One other important fundamental concept with respect to graphs is that of connectivity. We say that a graph G is **connected** if there is a path between every pair

of vertices in G. If G is not connected then choose any vertex $v \in V(G)$ and let C be the set of all vertices u, such that there is a path from v to u. We call C a **connected component** of G, and if we now choose some vertex $v \in V(G)-C$ and repeat this process, we get another connected component of G; thus it is clear that every graph G consists of one or more connected components. Each of these components is itself a connected graph, so that in most cases there is no loss in generality when we assume that a graph is connected; if it is not, then we just work on each of the connected components in turn.

When working with simple graphs it is convenient to use some notation which is shorthand for what has been given above. A simple graph does not have any multiple edges so an edge is uniquely defined by its endpoints and conversely. Consequently the notation $\{v_i,v_j\}$, or simply v_iv_j, is often used to denote a particular edge of a graph, and the notation $v_1v_2...v_r$ is used to denote a path. Where brackets are not given, the context determines whether or not the order of the vertices is important. A similar convention is also used when discussing networks, which will be introduced later.

1.2. The Breadth First Search

The most fundamental concept inherent in almost all graph algorithms is that of the search. Pseudo code which describes an algorithm to perform a **breadth first search** (often abbreviated to **BFS**) of a connected graph G is now given. The compound statement delimiters *begin* and *end* increase the length of a piece of pseudo code and thus make it harder to follow. Thus all pseudo code in this thesis uses the convention that compound statements are delimited using indentation, whilst *begin* and *end* are used only to delimit the statements in a sub-program.

```
Data Structures
    ScanQ:queue            { holds the vertices from which we still have to search }
    PrevPt[vertex]:vertex  { the parent of a vertex in the search tree }

Procedure BFS(G:graph)
1   begin
2   set the ScanQ to Ø
3   place some vertex r on the ScanQ
4   mark r searched
5   repeat
6       remove u from the ScanQ
7       for all vertices v adjacent to u do
8           if v has not been searched then
9               put v on the ScanQ
10              mark v searched
11              set PrevPt[v] = u
12  until the ScanQ is empty
13  end { Procedure BFS }
```

This algorithm uses a **queue** data structure, which is formally defined as a linearly ordered list Q with 2 operations:

- **enqueue**(Q,v) – place an element v into Q so that it is last in the ordering (i.e. at the back of the queue);
- x := **dequeue**(Q) – remove from Q the element x that is first in the ordering (i.e. at the front of the queue), and return x.

To enhance readability in the pseudo code, the operation enqueue(ScanQ,v) is described as "put v on the ScanQ", and the operation w := dequeue(ScanQ) is described as "remove w from the ScanQ".

The idea behind any search is that we begin by searching a (usually arbitrary) vertex $r \in V(G)$. We then (recursively) search an as yet unsearched vertex v that is adjacent to a searched vertex u through the edge uv, until every vertex in the graph has been searched. Searches proceed in a manner that guarantees that every vertex in the graph is eventually searched, provided that the graph is connected, and this is their most important property.

A **tree** is a connected graph T with n-1 edges, where n is the number of vertices in T. A tree T has no cycles, since any connected graph with a cycle has at least n edges. There is exactly one path between any pair of vertices in T, for if were two, T would contain a cycle. Any vertex with degree one in T is said to be a **leaf** of T, and all other vertices in T are called non leaves or internal vertices. Now suppose that one of the vertices of T, say r, is designated as the **root**; then T is called a **rooted tree**. If a vertex v is on the unique path P in T from r to another vertex u then we call v an **ancestor** of u, and u a **descendent** of v. If there are no vertices between v and u on P (i.e. u and v are the endpoints of an edge in E(T)) then we say that v is the **parent** of u and u is a **child** of v. Every vertex in T except the root has exactly one parent, but may have an arbitrary number of children. It is exactly those vertices with no children that are the leaves of T. The **depth** or **level** of of a vertex v in T is the length of the path from r to v.

Any search, including the BFS, conceptually builds what is called a **search tree**, and the search tree built by a BFS is usually called a **breadth first search tree**. As you can see from the above pseudo code, every vertex $v \in G$ (except the vertex r at which we began the search; this is the root of the search tree) is marked searched as a result of the algorithm finding that there is an edge, say uv, between v and an already searched vertex u. When this occurs we set PrevPt[v] = u. Thus we can make a graph T, where V(T)=V(G) and E(T) = all edges {v, PrevPt[v]}. If G has n vertices, then T is a connected graph with n vertices and n-1 edges; that is, T is a tree.

The defining property of a BFS is that we always visit all vertices at a given depth d before visiting any vertex with a depth greater that d. (Notice that in the above algorithm we consider a vertex to be searched at the time it is added to the ScanQ, and visited at the time it is removed. Thus vertices must be searched before they are visited, and their depth is defined at the time they are searched.) The reason for searching a graph G in this manner is that in the breadth first search tree, the length of the path from

the root r to any vertex v is equal to the length of the shortest path in G from r to v. It should now be easy to see that if we visit vertices in the same order in which they are searched, we perform a BFS; thus the need for the queue.

PrevPt is used to record of the parent of a vertex in T. Usually we are looking for a special kind of path P (which starts at the root), and when we discover P we have just searched some vertex v. Then, beginning at v, recursively following the edge from a vertex to its parent gives us the path from v to r (and therefore a path from r to v); thus PrevPt effectively records the required path.

We should also verify that T contains every vertex in G. To prove this is so, just suppose that some vertex v is not added into T and use the property that in a connected graph there is a path between every pair of vertices. There must therefore be a path in G between r and v, so let P be such a path and let w_i be the first vertex on P = $(r=w_0w_1w_2...w_{r-1}w_r=v)$ that was not searched (since $v=w_r$ is such a w_i we know that w_i does exist). Then w_{i-1} was searched, and since w_i is adjacent to w_{i-1} it must have been searched too, a contradiction.

To finish justifying that a breadth first search tree T is indeed a tree, we must prove that T is connected. Take any pair of vertices u and v in T; as described above, there is a path P from u to r, and a path Q from r to v. Let w be the first vertex on P that is also on Q (i.e. w is the first common ancestor of u and v in T) (maybe r=w), and let P^* be the portion of P from u to w, and Q^* be the portion of Q from w to v. Then P^*Q^* is a path from u to v.

We can also verify that T has no cycles, for suppose there is a cycle C in T, and consider the edge uv in C that was added to T last. C is a cycle, so there is already a path in T from u to v before we add the edge uv; thus u and v are both in T, and have both therefore been searched. But the algorithm never adds to T an edge for which both endpoints have already been searched, a contradiction.

One final point should be made here. The above discussion states that we may add a vertex v to the search tree T only if v is *adjacent* to a vertex u that is already in T, which implies that when searching directed graphs, we allow the search tree to contain only directed paths from the root r to a vertex v in T. This may be too restrictive though, so when dealing with directed graphs it is not unusual to also allow v to be added to T if $u \in T$ is adjacent to v. That is, rv-paths may contain forward and backward edges, and so are not necessarily directed.

1.3. Matchings in Bipartite Graphs

1.3.1. Introduction

A **matching** in a graph G is a set of edges $M \subseteq E(G)$ such that no vertex in V(G) is an endpoint of more than one edge in M. Any vertex that is the endpoint of an edge e in M is said to be **saturated**, and the two endpoints of e are said to be **matched** to one another. The vertex matched to a vertex u is denoted match[u], so if u is matched to v then match[u]=v and match[v]=u. A **maximum matching** in G is a matching M such that there is no matching in G with more edges than M, and a matching that saturates every vertex in G is called a **perfect matching**. An **alternating path** with respect to M is a path in G on which the edges are alternately in M and not in M.

Sometimes the vertices of a graph G can be partitioned into two sets, say $X=\{x_1,x_2,...,x_r\}$ and $Y=\{y_1,y_2,...,y_q\}$, such that every edge in G has one endpoint in X and one in Y. We say that such a graph is **bipartite** and call (X,Y) the **bipartition** of G. Bipartite graphs are a useful subclass of the set of all graphs. We often want to find out certain things about a graph (for example, find a maximum matching in G), and the algorithms that find these things in bipartite graphs are often much easier to understand and program than their counterparts, which operate on general graphs. The Hungarian Algorithm is such an algorithm.

1.3.2. The Hungarian Algorithm

The Hungarian Algorithm finds a maximum matching in a bipartite graph G by starting with an empty matching M, and repeatedly searching for an **augmenting path** P, which is an alternating path with respect to M that starts and ends at an unsaturated vertex. Thus P begins and ends with an edge that is not in M, so the number of edges on P that are in M exceeds the number of edges on P that are not in M by exactly one. It is therefore clear that if we consider M and P as sets of edges and set M equal to $M \oplus P$, where $\oplus$ represents the exclusive or operation performed on M and P, then not only will M still be a matching, but it will be a matching with exactly one more edge than before. The process of performing this "exclusive or" operation is called **augmenting on P** (see figure 1.1). In general, M is a maximum matching in G if and only if there is no augmenting path in G with respect to M (see [1] for a proof), and this is the fundamental principle on which both the Hungarian Algorithm and Edmonds' Algorithm, which finds a maximum matching in a general graph, are based.

The Hungarian Algorithm searches for an augmenting path in G by taking an unsaturated vertex $x_i \in X$, where G has bipartition (X,Y), and building a breadth first alternating search tree rooted at x_i.

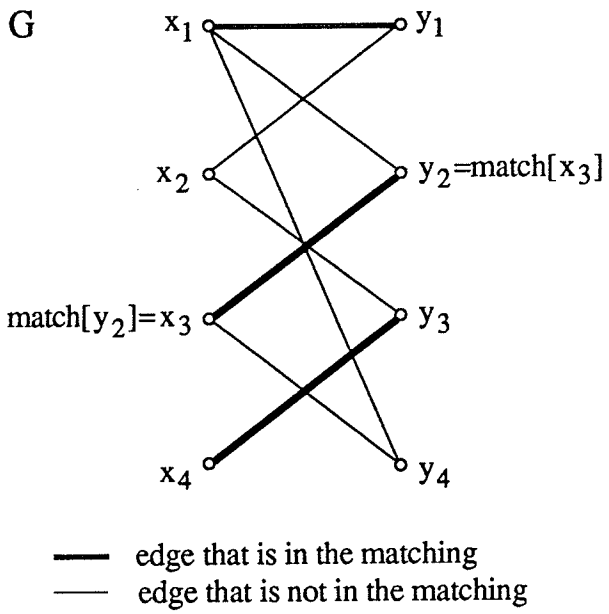


Figure 1.1
A bipartite graph with an augmenting path P ($x_2y_1x_1y_4$). The size of the matching can be increased by augmenting on P.

This search tree T is built using the following algorithm:

Data Structures

```
ScanQ:queue            { holds the vertices from which we still have to search }
Match[vertex]:vertex   { match[u] is the vertex matched to u }
```

Procedure Hungarian(G:graph)

```
1  begin
2  for i := 1 to n do   { G has n vertices }
3      if xi is unsaturated then
4          initialise ScanQ so that it is empty
5          put xi on ScanQ
6          repeat
7              remove x from the ScanQ
8              for each y adjacent to x do
9                  if y is not marked searched then
10                     if y is unsaturated then
11                         { we have found an augmenting path }
12                         use the augmenting path to augment the matching
13                     else
14                         mark y searched
15                         add edges xy and {y,match[y]} to the search tree
16                         add Match[y] to the ScanQ
17         until (an augmenting path is found) or (the ScanQ is empty)
18 end { Procedure Hungarian }
```

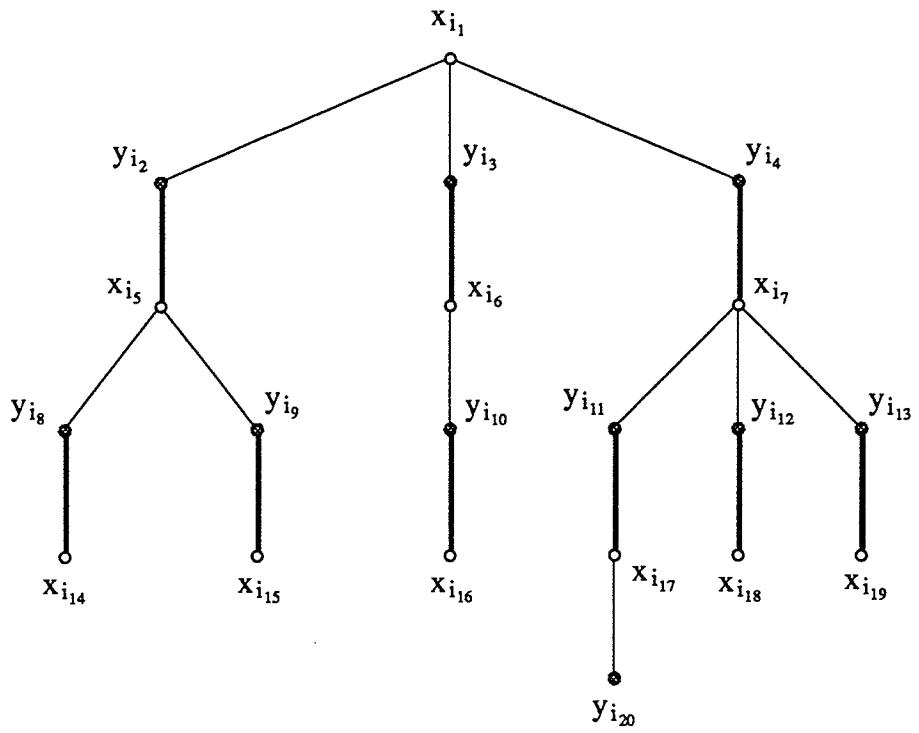
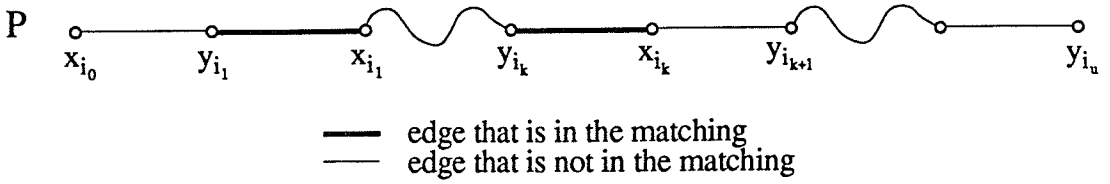


Figure 1.2
A breadth first alternating search tree. The inner vertices are shaded.

No Y-vertex y is ever added to the ScanQ. This is because if y is unsaturated then an augmenting path has been found, and if y is saturated then the *alternating* rt-path in T, where r is the root of T, can be extended to exactly one vertex: x=match[y]. Thus we never need to visit y.

In order to prove that the Hungarian Algorithm works, we must prove that if an iteration of statement 2 first searches a vertex x, and there is an augmenting path in G that begins at x, then that iteration finds an augmenting path. This can be proved by induction on the length of an alternating path from the root of the search tree to an unsaturated vertex, and a sketch of such a proof is as follows:

Suppose there exists in G an augmenting path P between x_{i_0} and y_{i_u}, and that the current iteration of statement 2 first searches x_{i_0}.



If y_{i_1} is unsaturated then the current iteration will find P, and if y_{i_1} is saturated then y_{i_1} and match$[y_{i_1}]=x_{i_1}$ will be added to the alternating search tree.

Suppose some y_{i_k} is added to the alternating search tree. Then x_{i_k} will be added to the search tree as match$[y_{i_k}]$, and cannot already be in the search tree since it can only be added as match$[y_{i_k}]$.

Now suppose some x_{i_k} is added to the alternating search tree T. *Because G is bipartite*, the length of the path from x_{i_0} to x_{i_k} in T is even, so when x_{i_k} is searched by the algorithm, an *alternating* path to $y_{i_{k+1}}$ is discovered, so $y_{i_{k+1}}$ is added to T. However, if G is *not bipartite*, then the path in T from x_{i_0} to x_{i_k} may be odd; in this case the path discovered to $y_{i_{k+1}}$ is not alternating, so there is no guarantee that $y_{i_{k+1}}$ will ever be added to T. This is why the Hungarian Algorithm works only for bipartite graphs.

This shows that if there is an augmenting path P in the bipartite graph G, then all the vertices on P are eventually added to the search tree by the algorithm, and since this includes y_{i_u}, an augmenting path is eventually found.

A vertex with an even depth in an alternating search tree is called an **outer vertex** and a vertex with an odd depth is called an **inner vertex**. Thus, when we build an alternating search tree T in a bipartite graph (where T is rooted at an X-vertex), all outer vertices in T will be X-vertices and all inner vertices will be Y-vertices. Hence the Hungarian Algorithm never adds an inner vertex to the ScanQ. In figure 1.2, which depicts an alternating search tree, the inner vertices are indicated by the shading.

However, when dealing with graphs that are not bipartite, it is sometimes necessary to add an inner vertex to the ScanQ. Consider the example in figure 1.3. The graph

shown contains exactly one augmenting path P between u and y. Suppose we use the Hungarian Algorithm on this graph, beginning at u. Then x is added to the search tree T as an inner vertex. But y is adjacent to only x in G, so in order for P to be discovered, x must be searched.

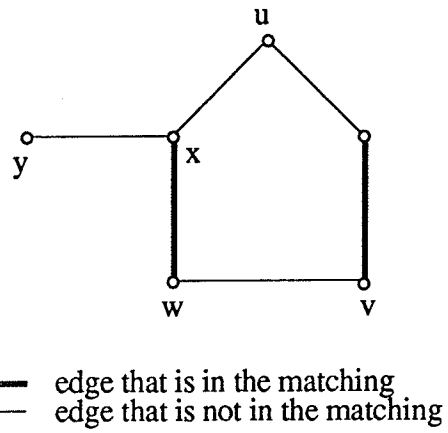


Figure 1.3
A graph with an alternating path that the Hungarian Algorithm may not find.

This problem arises because of the edge between the two outer vertices w and v, which allows us to travel back up the search tree to find an alternating path. Of course there is no such edge in a bipartite graph. It might appear at first glance that merely adding the inner vertices to the ScanQ might overcome this problem. In fact, while we do need to do this, on its own it is not enough. Even if we added the vertex x to the ScanQ we still could not extend the alternating search tree with edge xy because the path from u to y in T would not be alternating. The extra trick required to make things work is embodied in the concept of blossoms, which were discovered by Edmonds when he developed his algorithm [5] for finding matchings in general graphs. Edmonds' algorithm is discussed briefly in section 1.5.

1.4. Flows

1.4.1. Networks

A **network** N is a simple directed graph in which:

- one of the vertices is designated as a **source**, and another vertex is designated as the **target**[1];
- every edge e has associated with it a positive integer valued **capacity**, denoted cap(e).

A vertex v in N such that v is neither the source nor the target is called an **intermediate vertex**. Now let $S \subseteq V(N)$. Then $\overline{S}$ is the set of all vertices in $V(N)-S$, and $[S,\overline{S}]$, which is called an **edge cut** in N, is the set of all edges directed from a vertex in S to a vertex in $\overline{S}$. Whenever we consider an arbitrary edge cut $[S,\overline{S}]$, it will be assumed that $s \in S$ and $t \in \overline{S}$, unless otherwise stated.

If f is an integer valued function defined on E(N), and $K \subseteq E(N)$, then f(K) is used to denote $\sum_{e \in K} f(e)$. Also, $f^+(S)$ is used to denote $f([S,\overline{S}])$ and $f^-(S)$ is used to denote $f([\overline{S},S])$. The capacity of an edge is such a function, so we are free to talk about cap(K), the capacity of an edge cut K. We can therefore define a **minimum edge cut** or **minimum cut** in N to be an edge cut K^* such that every edge cut K in N satisfies $cap(K) \geq cap(K^*)$.

A **flow** in N is a non negative integer valued function defined on E(N) such that:

- the flow on an edge $e \in E(N)$, denoted flow(e), satisfies $0 \leq flow(e) \leq cap(e)$. This is called the **capacity constraint**;
- $flow^+(v) = flow^-(v)$ for all intermediate vertices v. This is called the **conservation condition**.

---

[1] In general multiple sources and targets can be allowed, but would unnecessarily complicate this discussion, as there is no loss in generality in assuming a single source and a single target.

Notice that a vertex $v \in N$ is a set of vertices that contains only one element. Thus $flow^+(v)$ is defined, and is the sum of the flows on all out-edges from v, and $flow^-(v)$ is the sum of the flows on all in-edges. For example, in figure 1.4 we have $flow^+(v_1)$ is 2+3=5, and $flow^-(v_1)$ is 2+3=5.

The **value** of a flow f in N, denoted **val(f)**, is defined to be $flow^+(source) - flow^-(source)$, and a **maximum flow** in N is a flow f^* such that every other flow in N has a value no larger than $val(f^*)$. Every network N has a valid flow, since the **zero-flow**, which is everywhere zero, satisfies the conditions for a flow.

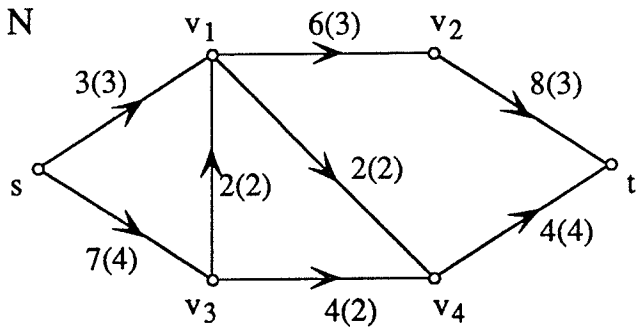


Figure 1.4
A network on which a non zero flow is defined. Standard notation, in which the capacity of each edge is indicated along with the flow in parentheses, is used. The value of the flow in the network is 7 units.

Now consider a path $P = v_1v_2...v_r$ in N. P is not necessarily a directed path, so the edges on P can be either forward or backward. The **residual capacity** of an edge e on P, denoted rescap(e), is defined as follows:

$$rescap(e) = \begin{cases} cap(e) - flow(e), \text{ if e is a forward edge;} \\ flow(e), \text{ if e is a backward edge.} \end{cases}$$

If an edge e on P has rescap(e)=0 then we say that both e and P are **saturated**, and if every edge e on P has rescap(e)>0 we say that P is an **unsaturated path**. For example, in the above diagram $sv_3v_4v_1v_2t$ is an unsaturated path Q in N. All the edges on Q are forward edges except v_1v_4 which is a backward edge. Since $e=v_1v_4$ is

followed in the direction v_4v_1 on P, it is usually convenient to call e the edge v_4v_1, even though this is technically incorrect. In general, to improve the clarity of the arguments used in this thesis, if $e=v_iv_j$ is a backward edge on a path P, then e will be referred to as the edge v_jv_i on P.

The residual capacity of an unsaturated path P is rescap(e), where e is an edge with the smallest residual capacity on P. In the above diagram the smallest residual capacity of an edge on Q is 2 (both the backward edge v_4v_1 and the forward edge v_3v_4 have a residual capacity of 2 on Q), so rescap(Q)=2. If P begins at the source s and ends at the target t, then we can increase the value of the flow in N by rescap(P)=δ units, where $\delta \geq 1$, by **augmenting on P**. This means that we increase the flow on all forward edges by δ units, and decrease the flow on all backward edges by the same amount.

To prove that this works we need to show that after augmenting on P we still have a valid flow in N, and that the value of this flow is larger, by δ units, than the flow that existed in N before we augmented. If the first edge on P is a forward edge then when we augment on P we increase $\text{flow}^+(s)$ by δ units, and if it is a backward edge we decrease $\text{flow}^-(s)$ by the same amount. Since augmenting on P causes only this change to the value of $\text{flow}^+(s)$ or $\text{flow}^-(s)$, it causes the value of the flow in N, $\text{flow}^+(s) - \text{flow}^-(s)$, to increase by δ units.

Now it is necessary to show that the conservation condition and the capacity constraint are both satisfied after we have augmented on P. Let us deal with the capacity constraint first. If an arbitrary forward edge e on P has cap(e)=c and flow(e)=f, then rescap(e)=c–f and after we augment on P flow(e)=f+δ, where $\delta \geq 1$. But δ was the smallest residual capacity of any edge on P, so $\delta \leq$ rescap(e)=c–f; thus $f+\delta \leq f+(c-f) = c$. Similarly for a backward edge e on P we reduce the flow on e by at most δ, where $\delta \leq f$. After we augment we therefore have flow(e)=$f-\delta \geq f-f=0$. This shows that if the capacity constraint is satisfied before we augment on P then it is satisfied afterward.

Now we deal with the conservation condition. Look at an arbitrary intermediate vertex v on P, and suppose the conservation condition holds before we augment on P. Then $\text{flow}^+(v) = \text{flow}^-(v)$. Since P is a path and v is an intermediate vertex (i.e. $v \neq s,t$), exactly two edges on P are incident on v; the one along which P enters v, and the one along which it leaves. Each of these edges may be either forward or backward, so we have four cases to consider. If both edges are forward then augmenting on P causes δ to be added to both $\text{flow}^+(v)$ and $\text{flow}^-(v)$, so after augmenting we have $(\text{flow}^+(v)+\delta) - (\text{flow}^-(v)+\delta) = 0$, since $\text{flow}^+(v) = \text{flow}^-(v)$ beforehand. If both edges are backward we get $(\text{flow}^+(v)-\delta) - (\text{flow}^-(v)-\delta) = 0$. If only the edge along which P enters v is forward we get $(\text{flow}^+(v)) - (\text{flow}^-(v)+\delta-\delta) = 0$, and if only the edge along which P leaves v is forward we get $(\text{flow}^+(v)-\delta+\delta) - (\text{flow}^-(v)) = 0$. Hence the conservation condition is also satisfied, so we still have a valid flow in N after augmenting on P.

We can therefore conclude that augmenting on P does indeed give us a larger flow in N, so we call any unsaturated path in N that begins at the source and ends at the target an **augmenting path**. In figure 1.4, $Q = sv_3v_4v_1v_2t$ is an augmenting path in N.

Now suppose that f is a flow in a network N, and let $[S,\bar{S}]$ be an edge cut in N such that $s \in S$ and $t \in \bar{S}$.

Lemma 1.1

$$\text{val}(f) = \text{flow}^+(S) - \text{flow}^-(S).$$

Proof:

$$\begin{aligned} \text{val}(f) &= \text{flow}^+(s) - \text{flow}^-(s) \quad \text{(by definition)} \\ &= \sum_{v \in S} (\text{flow}^+(v) - \text{flow}^-(v)) \end{aligned}$$

(since $\text{flow}^+(v) - \text{flow}^-(v) = 0$ for all $v \in S$, where $v \neq s$)

$$= \sum_{v \in S} \text{flow}^+(v) - \sum_{v \in S} \text{flow}^-(v) \quad (*)$$

$$= \text{flow}^+(S) - \text{flow}^-(S),$$

since every edge carrying flow between two vertices that are both in S contributes nothing to the total flow in the R.H.S. labelled (*) above. This is because the amount of the flow is added to the total in the first sum and subtracted from the total in the second sum. □

It is now clear that $\text{val}(f) \leq \text{cap}([S,\bar{S}])$, since the flow on an edge e cannot exceed the capacity of e. Thus the value of a maximum flow in N cannot exceed the value of a minimum edge cut. Furthermore, if one can find a flow f such that $\text{val}(f) = \text{cap}([S,\bar{S}])$, then val(f) must be maximum and $\text{cap}([S,\bar{S}])$ must be minimum. In fact one can always find such a flow. Use any well known technique, such as the Ford–Fulkerson algorithm described below, to augment the flow continually in N until no augmenting path remains; then $[S,\bar{S}]$, where S is the set of all vertices to which there is an unsaturated path from s, is such an edge cut. To verify this, observe that in order for us to be unable to continue any augmenting path from a vertex u in S to a vertex v in $\bar{S}$, either flow(uv) = capacity(uv) in the case that uv is a forward edge, or flow(uv) = 0 otherwise. Thus $\text{flow}^+(S) = \text{cap}([S,\bar{S}])$ and $\text{flow}^-(S) = 0$. But $s \in S$, and $t \in \bar{S}$ because there is no unsaturated st-path, so $\text{val}(f) = \text{flow}^+(S) - \text{flow}^-(S) = \text{cap}([S,\bar{S}])$. This proves:

Max-Flow–Min-Cut Theorem

The value of a maximum flow in a network N is equal to the capacity of a minimum edge cut in N.

If we augment on $Q = sv_3v_4v_1v_2t$ in the network N in figure 1.4, then the value of the flow in N will be maximum. Figure 1.5 shows the resulting maximum flow in N, along with the corresponding minimum edge cut $[S,\overline{S}]$.

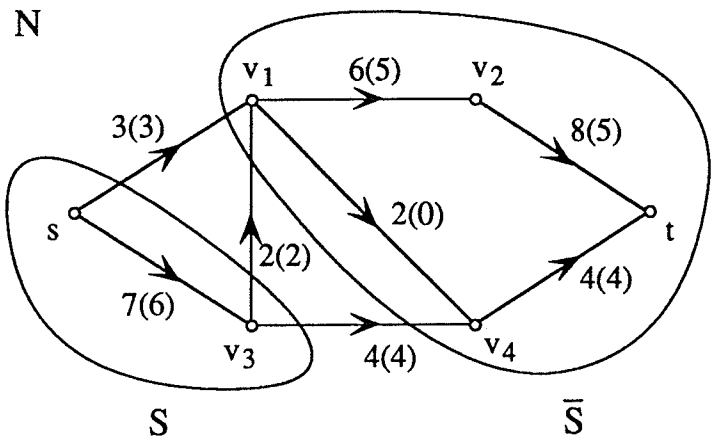


Figure 1.5
A network in which the flow is maximum (9 units). A minimum edge cut in N (which has capacity 9) consists of all edges directed from a vertex in S (i.e. s or v_3) to a vertex in $\overline{S}$.

1.4.2. The Ford–Fulkerson Algorithm

The Ford Fulkerson Algorithm finds a maximum flow in a network N. It works by repeatedly searching for an augmenting path P in N and increasing the flow in N by augmenting on P. Although more efficient maximum flow algorithms were subsequently discovered, the basic idea behind most of them is an extension of that used by the Ford–Fulkerson Algorithm. Since it is only this basic idea that is required for the work contained herein, a description of the Ford–Fulkerson Algorithm, rather than one of its more efficient but more complicated descendants, is now presented.

A breadth first search is used to look for an augmenting path, since always using a shortest augmenting path gives the algorithm the best complexity. This is called the Edmonds–Karp modification. If it is not possible to find an augmenting path then the flow in N is maximum, and the algorithm terminates. The pseudo code for this algorithm is as follows:

```
Mainline
1   init flow and maxflow to 0
2   repeat
3       flow:=IncreaseFlow()
4       maxflow:=maxflow+flow
5   until flow=0
{ maxflow is now the maximum flow in the network }

Function IncreaseFlow():integer
    { finds one augmenting path using a breadth first search }
    ScanQ:queue              { vertices from which we still have to search }
    Rescap[vertex]:integer   { min rescap up to a vertex on P }
1   begin
2   put the source on the ScanQ
3   mark the source searched
4   repeat
5       remove vertex u from the head of the ScanQ
6       for all vertices v adjacent to u do
7           if v has not been searched then
8               if uv is unsaturated then
9                   add v to the ScanQ
10                  mark v searched
11                  Rescap[v] := min rescap on path to v
12                  if v=target then
13                      augment the flow on P by Rescap[v]
14                      return(Rescap[v])
15  until the ScanQ is empty
16  return(0)
17  end { Function IncreaseFlow }
```

This algorithm creates a breadth first search tree T, which consists of unsaturated paths from the root; we naturally call this an unsaturated breadth first search tree. The vertices that are added to T in the final execution of function IncreaseFlow form the set S, consisting of all vertices in N to which there is an unsaturated path from s. Thus $[S,\bar{S}]$ is a minimum edge cut in N. The edges of T are only necessarily unsaturated in one direction – from the higher vertex (the one with the smaller depth) to the lower one; that is, down the tree. This idea leads us to the concept of a residual network.

The **residual network** of a network N with flow f, R(N,f), is created as follows:

1. The vertices of R(N,f) are those of N.
2. For each edge e=(u,v), where cap(e) = c and flow(e) = ϕ, create in R(N,f) an edge (u,v) with capacity $c - \phi$ if $c - \phi > 0$, and an edge (v,u) with capacity ϕ if $\phi > 0$.
3. Remove all multiple edges from R(N,f) as follows: if R(N,f) contains multiple edges between u and v, then replace them with a single edge uv, with capacity equal to the sum of the capacities of the multiple edges.

The initial flow in R(N,f) is the zero-flow.

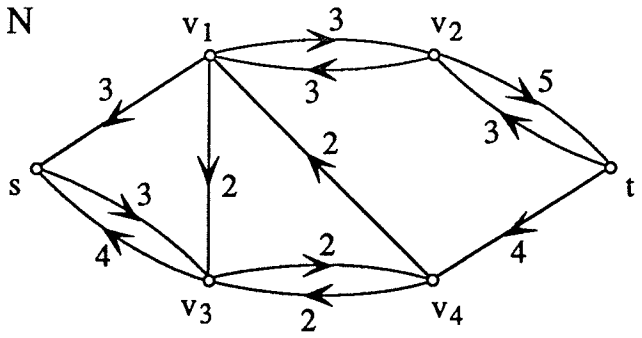


Figure 1.6
The residual network for the network in figure 1.4.

The capacities of the edges in R(N,f) are the residual capacities of the edges in N, and the edges of R(N,f) are unsaturated in a forward direction only; thus R(N,f) is the network containing all unsaturated paths in N. Any augmenting path in N corresponds to an augmenting path in R(N,f) that contains only forward edges (i.e. a directed augmenting path) and conversely. Also, a maximum flow in N has a value equal to the value of the current flow in N plus the value of a maximum flow in R(N,f); that is, (max flow in N) = (max flow in R(N,f)) + (current flow in N).

1.4.3. Using Flows to Find Matchings in Bipartite Graphs

Any bipartite graph G with bipartition (X,Y), where $X=\{x_1,x_2,...,x_r\}$ and $Y=\{y_1,y_2,...,y_s\}$, can be transformed into a network N. We direct all edges x_iy_j from X to Y, add a source vertex s and a target vertex t, add edges sx for all vertices $x \in X$ and edges sy for all vertices $y \in Y$, and give all edges in N capacity one. This process is illustrated in figure 1.7.

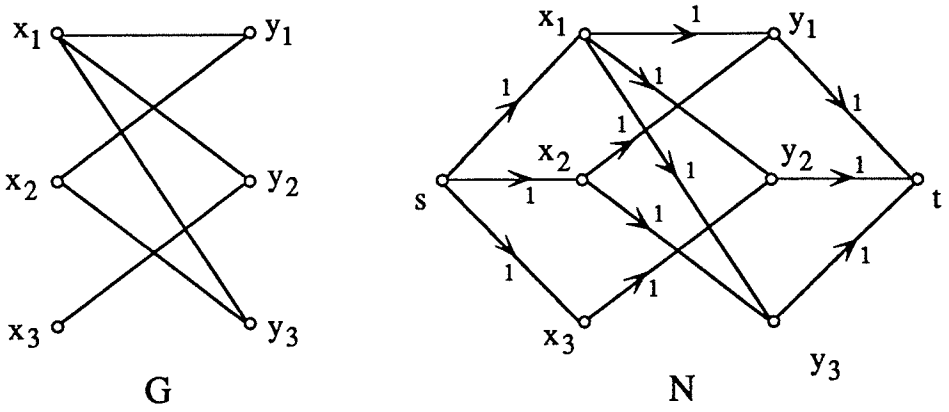


Figure 1.7
The transformation of a bipartite graph G into a network N so that a maximum flow in N will also define a maximum matching in G.

There is a 1–1 correspondence between a maximum flow f in N and a maximum matching M in G; the edges in M are the edges x_iy_j that carry flow in (N,f). In fact, if we examine the Ford-Fulkerson maximum flow algorithm and the Hungarian maximum matching algorithm, we see that they are actually the same algorithm when used to find a maximum matching in a bipartite graph. Augmenting paths in the Ford-Fulkerson algorithm consist of consecutive edges which have a positive residual capacity. This means that an augmenting path P in the above network must alternately follow forward edges, which carry no flow, and backward edges, which carry one unit of flow (excluding the first and last edges). This is exactly the same as following an alternating path in G, since edges with flow can be considered to be edges that are in the matching

and edges with no flow can be considered to be edges that are not in the matching. This proves lemma 1.2.

Lemma 1.2

Let G be a bipartite graph, and let N be the network constructed from G using the above technique. Then every maximum matching in G corresponds to a maximum flow in N, and conversely.

1.5. Edmonds' Algorithm for Matchings in General Graphs

As mentioned previously, Edmonds developed an algorithm for finding matchings in general graphs [5] which overcomes the problems encountered when trying to extend the Hungarian Algorithm. Edmonds' algorithm uses **blossoms**, which are recursive structures based on odd alternating cycles, to solve these problems. Consider the following example, which is the same example given previously but reproduced here with numbered vertices for convenience.

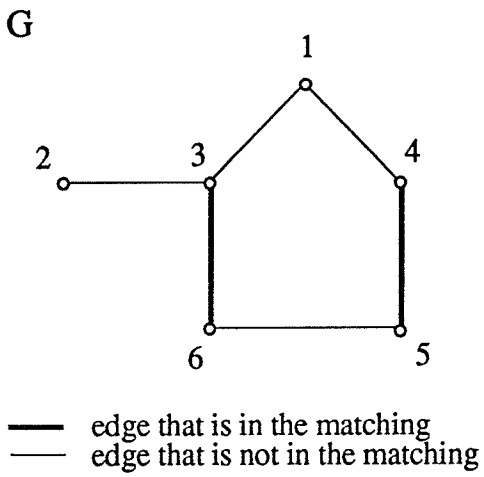


Figure 1.8
A graph with an augmenting path that a breadth first search cannot find.

As shown previously, the Hungarian Algorithm will never find the augmenting path P = 145632, because P ends at a vertex (2) which is joined only to inner vertices in the search tree, and these vertices are never visited. It was also shown previously that the

Hungarian Algorithm fails because there is an edge between two outer vertices (5 and 6) on the augmenting path. This edge creates an odd alternating cycle, C = 145631, in the search tree.

Edmonds called such a cycle a blossom. Every blossom B contains exactly one vertex that has only edges in G–M incident on it in B. This vertex is called the **base** of the blossom. In the current example, vertex 1 is the base of the blossom C. Edmonds recognised that if there is an edge from any vertex in a blossom to a vertex u outside the blossom, then there is always an alternating path from u to the base of the blossom. This means that if we shrink a blossom into a single vertex, the new graph with the shrunken blossom will contain an augmenting path if and only if G does. Edmonds' Algorithm therefore finds and shrinks blossoms as it searches for an augmenting path. This conceptual model is physically realised by adding to the ScanQ all vertices in a newly discovered blossom that have not yet been so added.

When Edmonds' Algorithm operates on the current example, the inner vertex 3, which was not added to the ScanQ when it was added to the search tree, is later added to the ScanQ when the algorithm finds the blossom C. This allows the algorithm to find the augmenting path in the graph.

The algorithm developed in this thesis uses flow techniques to find maximum matchings in general graphs. These techniques use a structure that is similar to Edmonds' blossom, but more general. This allows the algorithm to solve not only the maximum matching problem, also known as the 1-factor problem, but also the f-factor problem, which is a natural generalisation of the maximum matching problem.

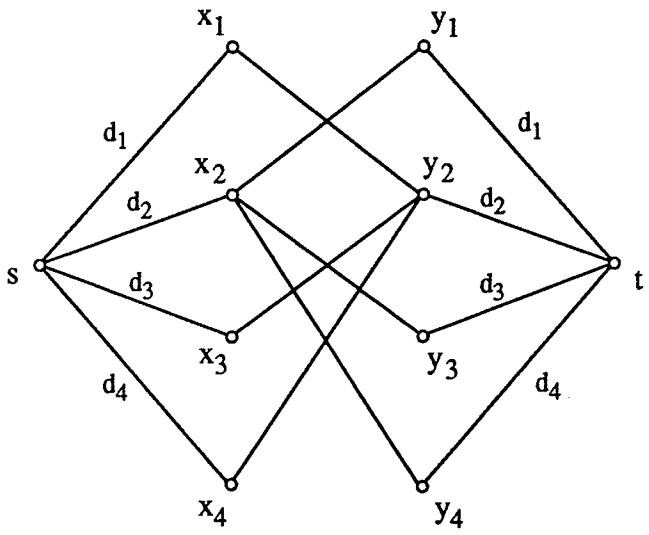
2. Balanced Networks

2.1. Introduction

A balanced network is a network N with the following properties:

- The vertices of N consist of a source s, a target t, and two sets of vertices $X=\{x_1,x_2,...,x_n\}$ and $Y=\{y_1,y_2,...,y_n\}$;
- N contains the pair of edges sx_i and y_it for $1\leq i\leq n$;
- The remaining edges of N are between X and Y, and occur in pairs; these pairs are either x_iy_j and x_jy_i or y_ix_j and y_jx_i, where $i\neq j$ for every such pair. Every such pair is called a **complementary pair**, as is every pair of edges of the form $\{sx_i,y_it\}$, and the two edges in any complementary pair are said to be **complementary edges**;
- The two edges in a complementary pair always have the same positive integer valued capacity;

The pairs of complementary edges partition E(N). If the two edges in a complementary pair carry the same flow, then we say that they satisfy the **rule of complementarity**. A **balanced flow** in N is a flow in which every complementary pair of edges in N satisfies the rule of complementarity. We work strictly with balanced flows in balanced networks. If f is a flow in a balanced network N, and f is not balanced, then f is not considered to be a valid flow in N. As can be seen from the following example, a balanced network is a symmetric bipartite network.



N

Figure 2.1
An example of a balanced network. Edges that are not labelled have a capacity of one and all edges are directed from left to right.

The vertices of N also occur in complementary pairs. We consider any pair of vertices $\{x_i,y_i\}$ to be such a pair, and call x_i and y_i **complementary vertices.** We also define y_0=s and x_0=t, so that {s,t} is also a complementary pair. An arbitrary edge or vertex in N occurs in exactly one complementary pair; thus it has a unique complementary edge or vertex, respectively.

If G and H are graphs such that $V(H)\subseteq V(G)$ and $E(H)\subseteq E(G)$, then we say that H is a **subgraph** of G. An extremely useful property of balanced networks, and indeed the motivation for their development, is the existence of a 1-1 correspondence between a flow in a special type of balanced network N and a subgraph H of a graph G. Every graph G corresponds to a unique balanced network N. If the vertices of G are 1,2,...,n then the vertices of N are $s,t,x_1,y_1,x_2,y_2,...,x_n,y_n$. The pairs of complementary edges in N that do not have any endpoint incident on s or t are all of the form $\{x_iy_j,x_jy_i\}$, and N contains such a pair if and only if ij is an edge of G. It is clear that every balanced network N with this structure also corresponds to a unique graph G. Every flow in N

corresponds to a subgraph H of G, and conversely. ij is an edge of H if and only if there is a non-zero flow defined on the pair of complementary edges x_iy_j and x_jy_i in G.

Now consider a path $Q = v_1,e_1,v_2,e_2,\ldots,e_{r-1},v_r$ in a balanced network N. The **complementary path** of Q is defined to be the path $Q' = v'_r,e'_{r-1},v'_{r-1},e'_{r-2},\ldots,e'_1,v'_1$, where v'_i and e'_i denote the complementary vertex and edge of v_i and e_i respectively. That is, Q′ contains all the complementary vertices and edges of the vertices and edges on Q, and these complementary vertices and edges appear on Q′ in exactly the opposite order from their complementary partners on Q. It can be seen directly from the relevant definitions that Q′ is indeed a path, and the foregoing observation regarding uniqueness of complementary edges and vertices makes it clear that Q′ is unique.

Since the idea of complementary pairs is a fundamental one in the study of balanced networks, the notational convention introduced above is now adopted permanently. Specifically, if z is one element in a complementary pair C, then z′ denotes its complementary partner, the other element in C. For example, if $v \in V(N)$ then the complementary vertex for v is denoted v′; thus if $v=x_i$ then $v'=y_i$, and if $v=y_i$ then $v'=x_i$. Similarly, if $e=x_iy_j \in E(N)$ then $e'=x_jy_i \in E(N)$, and if P is a path in N then P′ is its complementary path. For all complementary pairs $C=\{z,z'\}$ we have the identity $(z')'=z$.

A very important property of pairs of complementary paths is that *if P is a path that begins at u and ends at v (i.e. P is a uv-path), then P′ is a v′u′-path.* If u and v are a complementary pair then v=u′, so P′ begins at $(u')'=u$ and ends at u′=v; thus P′ is also a uv-path. As we shall see later, this is of particular use when we deal with an augmenting path P, since then P′ is also an augmenting path (see figure 2.4).

2.2. Maximum Flows and Minimum Edge Cuts

For the remainder of this thesis we assume we are working with a balanced network in which *all the edges that are not incident on s or t are directed from X to Y.* This is done because one of the main thrusts of the work in this thesis is the relationship between graphs and this special kind of balanced network. Thus, while many of the results obtained are true in general, generalising the discussion would unnecessarily complicate many of the arguments used.

First recall from Chapter 1 that whenever we consider an edge cut $K = [S,\overline{S}]$ in a network, we always assume that $s \in S$ and $t \in \overline{S}$, unless otherwise stated. Given an arbitrary edge cut $K = [S,\overline{S}]$ in a balanced network N, we may divide the pairs of complementary vertices of N, other that {s,t}, into 4 sets:

$A = \{x_i,y_i \mid x_i \in \overline{S},\ y_i \in S\}$

$B = \{x_i,y_i \mid x_i \in S,\ y_i \in \overline{S}\}$

$C = \{x_i,y_i \mid x_i \in S,\ y_i \in S\}$

$D = \{x_i,y_i \mid x_i \in \overline{S},\ y_i \in \overline{S}\}$

A may then be further subdivided into $A_x = \{x_i \mid x_i \in A\}$ and $A_y = \{y_i \mid y_i \in A\}$, and the other 3 sets may also be subdivided in the same manner (see figure 2.2). It is also convenient to use the notation $\overline{K}$ to denote the edge cut $[\overline{S},S]$. In particular, we then have $flow(\overline{K}) = flow^{-}(S)$; since it is clear that $flow(K) = flow^{+}(S)$, this shows us that $val(f) = flow(K) - flow(\overline{K})$.

By the capacity constraint, we know that in any network N, the value of a maximum flow cannot exceed the capacity of an edge cut $K = [S,\overline{S}]$, where $s \in S$. This is of course true for balanced networks too, but in this special case this statement is not as strong as it could be. With arbitrary networks one can always find a flow f such that val(f) = cap(K) for some edge cut K, by the Max-Flow–Min-Cut Theorem. But we work strictly with balanced flows in balanced networks, and we shall see later that it is

not always possible to find a balanced flow f such that val(f) = cap(K). It would therefore be useful to find a special kind of edge cut K in a balanced network N, such that K has properties in N that are analogous to those mentioned above for the usual kind of edge cut in an arbitrary network. This is what motivates the work in this section.

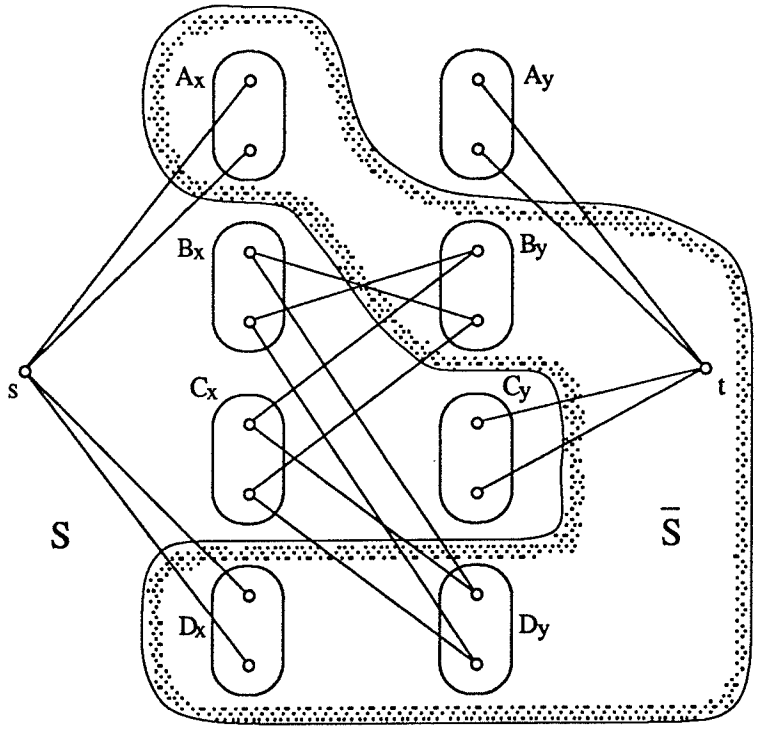


Figure 2.2
The general structure of an edge cut K in a balanced network. The edges drawn belong to K.

If G is a graph and $C \subseteq V(G)$, then we can make a graph H by taking V(H) = C and $E(H) = \{uv \mid uv \in E(G) \text{ and } u,v \in C\}$. H is a subgraph of G, and is called the **subgraph induced by C**. Now suppose we have an edge cut K in balanced network N. K has the general structure given earlier, so the 4 sets A,B,C and D partition the pairs of complementary vertices in $V(N)-\{s,t\}$. Let $C_1,C_2,...,C_k$ be the connected components of the subgraph induced by C, K_i be those edges of K that have an endpoint in C_i and K_r be the remaining edges in K; that is, the set of edges that are in K but not in any K_i.

Let us also define $\bar{K}_i$ to be those edges of $\bar{K}$ that have an endpoint in C, and $\bar{K}_r$ to be the remaining edges in $\bar{K}$.

Definition:

A **balanced edge cut** in a balanced network N is an edge cut in N that satisfies the following conditions:

- There are no edges between C and D;
- If $u \in C_i$ then $u' \in C_j$;
- $cap(K_i)$ is odd, $1 \leq i \leq k$.

This definition of a balanced edge cut appears to be totally arbitrary; however, we see later that it is just the kind of edge cut we are looking for. One important property of the sets C_i should be emphasized, since the following arguments rely on it entirely; this property is that just like the sets A,B,C and D, *each C_i consists entirely of pairs of complementary vertices.*

The following notation is now required. To avoid a subscripting nightmare, I will sometimes be used to denote C_i when an arbitrary C_i is being discussed. Thus I_x denotes all vertices in C_x that are also in C_i, and I_y is similarly defined (see figure 2.3). If [A,B] denotes the set of all edges directed from A to B in N (all such edges will in fact be directed from A_x to B_y), then X_{AB} is used to denote cap([A,B]) and F_{AB} to denote flow([A,B]). By the rule of complementarity we therefore have $X_{AB} = X_{BA}$ and $F_{AB} = F_{BA}$. If [s,A] denotes all edges directed from s to A and [A,t] denotes all edges directed from A to t, then X_A denotes cap([s,A]) = cap([A,t]) and F_A denotes flow([s,A]) = flow([A,t]). This notation is also extended to B,C,D and I in the obvious way.

Since the C_i are the connected components of C and there are no edges between C and D, the only edges directed out of I_x are those directed from I to I, A and B. Hence $flow^+(I_x) = F_{II} + F_{IA} + F_{IB}$. The only edges directed into I_x are those from s,

so $\text{flow}^-(I_x) = F_I$. From the proof of lemma 1.1 it is clear that $\text{flow}^+(I_x) - \text{flow}^-(I_x) = 0$. But $\text{flow}^+(I_x) - \text{flow}^-(I_x) = (F_{II} + F_{IA} + F_{IB}) - (F_I)$, so we have the identity:

Identity 2.1 $\quad F_{II} + F_{IA} + F_{IB} - F_I = 0$

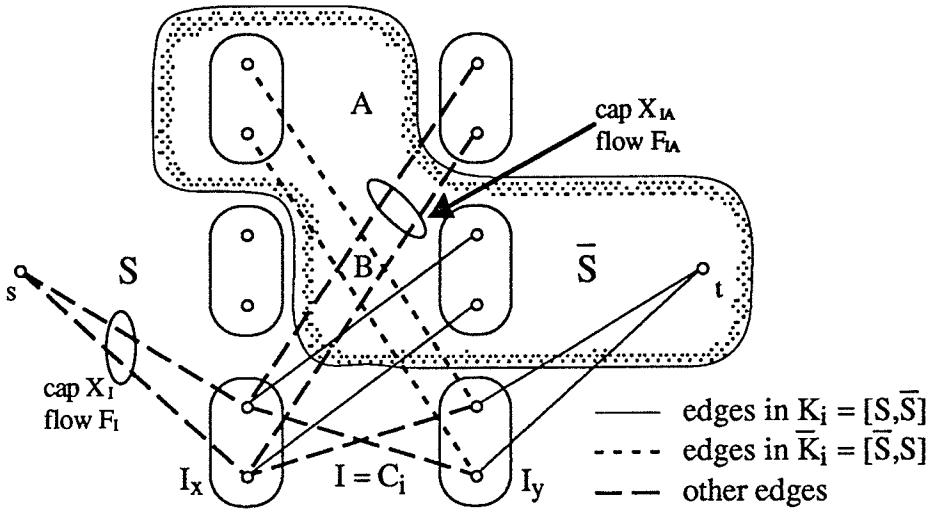


Figure 2.3

Part of an edge cut in a balanced network. The different kinds of edges in K_i and $\bar{K}_i$ are shown, along with some other edges relevant to the current discussion.

If e is an edge directed from A to A, then e′ is also such an edge. Thus F_{AA} is an even number, and so is any F_{UU}, where $U \in \{A,B,C,D,I\}$.

The only edges that are directed from I to a vertex in $\bar{S}$ are those directed from I_x to B_y, and those directed from I to t. Hence $\text{flow}(K_i) = F_I + F_{IB}$. The only edges directed from $\bar{S}$ to I are those directed from A_x to I_y, so $\text{flow}(\bar{K}_i) = F_{AI} = F_{IA}$.

Lemma 2.2

Let f be a balanced flow in a balanced network N, and K be a balanced edge cut in N. If $\text{flow}(\bar{K}_i) = 0$, then there is an edge $e \in K_i$ such that $\text{flow}(e) < \text{cap}(e)$.

Proof:

If $\text{flow}(\bar{K}_i) = 0$ then $F_{AI} = F_{IA} = 0$, so $F_{II} + F_{IA} + F_{IB} - F_I = F_{II} + F_{IB} - F_I = 0$ by identity 2.1. Hence $F_I = F_{II} + F_{IB}$, so $\text{flow}(K_i) = F_I + F_{IB} = 2F_{IB} + F_{II}$, which

is even because F_Π is. But $cap(K_i)$ is odd and $flow(K_i) \leq cap(K_i)$, so the required edge must exist. □

Since flow and capacity are integer valued functions, this shows that for this particular edge cut $K = [S,\bar{S}]$ we must have at least k edges $e \in E(N)$, one for each C_i, such that either $e \in K$ and $flow(e) \leq cap(e) - 1$, or $e \in \bar{K}$ and $flow(e) \geq 1$. Hence

$$(cap(K) - flow(K)) + flow(\bar{K}) \geq k$$

$$\therefore \quad flow(K) - flow(\bar{K}) \leq cap(K) - k$$

But $flow(K) - flow(\bar{K}) = val(f)$, so $val(f) \leq cap(K) - k$; that is, the value of a maximum flow in N cannot exceed $cap(K) - k$.

The capacity of a balanced edge cut K, denoted **balcap**(K) is therefore defined to be $cap(K) - k$. A **minimum balanced edge cut** is a balanced edge cut K^* such that every balanced edge cut K in N satisfies $balcap(K) \geq balcap(K^*)$. We have now proved lemma 2.3.

Lemma 2.3

$val(f) \leq balcap(K)$, where f is a balanced flow in a balanced network N and K is a balanced edge cut in N.

2.3. Complementary Augmenting Paths

The concept of the complementary path of an augmenting path in a balanced network is important enough that some of the concepts involved should be illuminated.

Suppose P is an augmenting path in a balanced network N in which there is no flow on any edge in E(N). Then it is easy to see that it's complementary path P' is also an augmenting path in N (see theorem 2.7). If we augment not only on P but on P' as well, we find that every edge in N carries the same flow as its complementary edge (see theorem 2.8), and it is again true that if Q is an augmenting path in N then its complementary path Q' is also an augmenting path in N. Since only balanced flows are

valid in N, we must be sure that as we try to find a maximum flow we keep the flow balanced; that is, we retain in N the property that every edge carries the same flow as its complementary edge. The technique just described ensures that this is indeed the case.

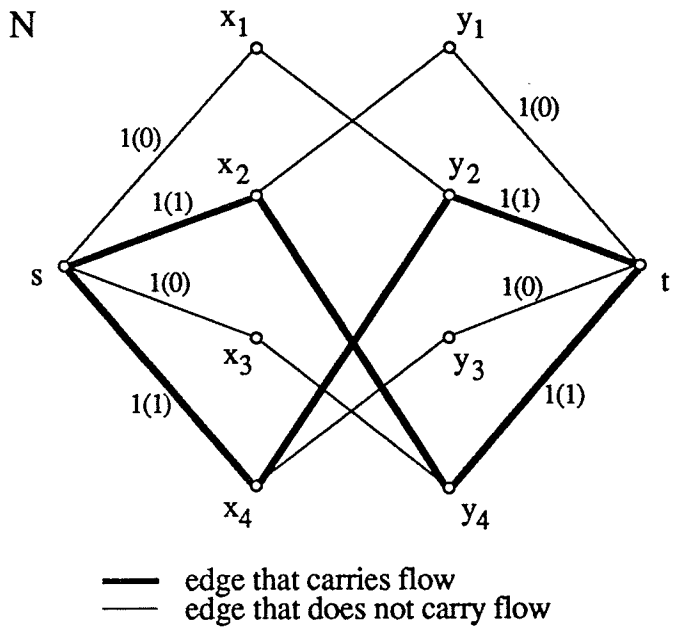


Figure 2.4
An illustration of the fact that if P is an augmenting path in a balanced network, then so is P′. All edges x_iy_j in N have capacity one. P = $sx_1y_2x_4y_3t$ is an augmenting path, and so therefore is P′ = $sx_3y_4x_2y_1t$.

Lemma 2.4

Suppose f is a balanced flow in a balanced network N. Then R(N,f) is balanced.

Proof:

Look at an arbitrary edge e=uv in N, where cap(e) = c and flow(e) = ϕ. Then cap(e′) = c by definition, and flow(e′) = ϕ by the rule of complementarity. uv therefore has a residual capacity of c – ϕ in the direction uv, and e′=v′u′ has the same residual capacity in the direction v′u′. Thus if c – ϕ > 0, then R(N,f) contains the complementary pair of edges {uv,v′u′}, and both edges in this complementary

pair have the same capacity in R(N,f). Similarly, if $\phi > 0$ then uv has a residual capacity of ϕ in the direction vu, so R(N,f) contains the complementary pair of edges {vu,u′v′}, and both edges in this complementary pair has the same capacity in R(N,f).

Since this comprises all edges of R(N,f), E(R(N,f)) is partitioned into pairs of complementary edges, where the two edges in a complementary pair always have the same capacity. Thus R(N,f) is indeed a balanced network. □

Lemma 2.5

If e is an arbitrary edge on an augmenting path P in a balanced network N then e′ has the same direction on P′ (i.e. forward or backward) that e has on P.

Proof:

Consider an arbitrary edge e=uv on P. Then e′=v′u′ is an edge on P′. By definition, e and e′ are either both directed from X to Y, or both from Y to X. If $u \in X$ then $v \in Y$ and $v' \in X$, so both e and e′ are forward edges on their respective paths. Similarly, if $u \in Y$ then e and e′ are both backward edges. □

Corollary 2.6

Given any augmenting path P in a balanced network N, the residual capacity of an edge e on P is the same as the residual capacity of its complementary edge e′ on P′.

Proof:

By definition e and e′ both carry the same flow and have the same capacity. Since e′ has the same direction on P′ as e has on P it must be true that rescap(e′) on P′ is the same as rescap(e) on P. □

Theorem 2.7

If P is an augmenting path with a residual capacity of r in a balanced network N, then P′ is also an augmenting path with a residual capacity of r in N.

Proof:

If P is an augmenting path then both P and P′ are st-paths, and by corollary 2.6 every edge e′ on P′ has the same residual capacity as e on P. Hence P′ is also an augmenting path, and rescap(P′) = rescap(P). □

Theorem 2.8

Suppose that every time we use an augmenting path P to augment the flow in a balanced network N, we also use P′ to further augment the flow in N by the same amount. Then every edge e in N always carries the same flow as e′ (i.e. the flow in N remains balanced).

Proof:

The proof is by induction on the number of times we have augmented the flow in N. When we have not yet augmented the flow in N every edge carries zero flow so the theorem is true.

Let us suppose we have augmented the flow in N k times in the manner stated in the theorem; then by the induction hypothesis every edge e in N carries the same flow as e′, so the flow in N is still balanced. Now suppose that there is an augmenting path P in N. Then by theorem 2.7 P′ is also an augmenting path in N. Now further suppose that we are able to augment the flow on both P and P′ by some amount δ. When we augment on P we change the flow on every edge e on P by δ, and when we augment on P′ we change the flow on every edge e′ on P′ by the same amount. Furthermore, e and e′ have the same direction on P and P′ respectively (by lemma 2.5), so augmenting on these two paths either increases the

flow on both e and e′ by δ, or decreases it by δ. In either case, after we have augmented on P and P′ these two edges carry the same flow, so it is clear that the flow in N remains balanced. □

At this stage we have to be a little bit careful. While it has been shown that if P is an augmenting path in a balanced network N then P′ is always an augmenting path in N, it has not been shown, and nor is it necessarily true, that we can always augment on both P and on P′. It is sometimes the case that augmenting on P destroys the augmenting path P′, and this is why in the above proof it was assumed that it was possible to augment on both P and P′ by δ. This leads nicely into a description of the restrictions that must be applied when searching for an augmenting path in a balanced network.

2.4. Finding a Maximum Flow

2.4.1. Valid and Invalid Paths

Theorem 2.9

Given an augmenting path P with residual capacity δ in a balanced network N, we can augment the flow in N on P and on P′ by at least one unit (but not necessarily by δ units), if and only if P does not contain a pair of complementary edges with a residual capacity of one.

Proof:

First suppose that P does contain such a pair of complementary edges. Then P has the form s...uv...v′u′...t, where rescap(uv)=1, and maybe u=s. When we augment on a path with this form, the residual capacity of uv and v′u′ on P′ becomes zero, so P′ is no longer an augmenting path in N.

Now suppose that P does not contain a pair of complementary edges with a residual capacity of one, and let e be an arbitrary edge on P. If e′ is not on P then augmenting on P does not change its flow, and so its residual capacity on P′ does

not change either, but remains $\geq \delta$. If e' is on P then it has a residual capacity of at least two before augmenting; thus if we augment on P by one unit, the residual capacity of e' on P' changes by one unit and is therefore still at least one. So, after augmenting on P by one unit, we can still augment on P' by at least one unit. □

Hence whenever P contains a pair of complementary edges that have a residual capacity of one, it is not possible to augment on both P and P' and retain in N the property that every edge carries the same flow as its complementary edge. We must therefore ensure that any algorithm we use to find a maximum flow in a balanced network N considers an unsaturated st-path to be an augmenting path only if it does not have this form.

Let N be a balanced network, and $v \in V(N)$. An unsaturated uv-path in N that contains a pair of complementary edges with a residual capacity of one is called an **invalid path**. Any other unsaturated uv-path is called a **valid path**, and P is an **augmenting path** in N only if it is a valid st-path. However, the term "valid augmenting path" will still be used when it is desirable to emphasize that an unsaturated st-path is valid. We say that a vertex v is **s-reachable** if there is a *valid* path in N from s to v, and **t-reachable** if there is a *valid* path from v to t.

2.4.2. Maximum Balanced Flows and Minimum Balanced Edge Cuts

There is an important detail that has yet to be taken care of. If N is a network and f is a flow defined on E(N), then there is a theorem that tells us that the value of f is maximum if and only if N does not contain an augmenting path. Without this theorem we would not be able to use the "augmenting path" technique to find a maximum flow in a network. Since every balanced network is also a network, this theorem still holds if N happens to be a balanced network. However, theorem 2.9 alerts us to the fact that we may only use *valid* augmenting paths in a balanced network N, so if we are to use the "augmenting path" technique to find a maximum flow in N, then we need a parallel

theorem for balanced networks. This section develops such a theorem, and another powerful analogy between balanced networks and their more general counterparts.

Let N be a balanced network and f be a balanced flow in N such that there is no valid augmenting path in (N,f). Consider the edge cut K = [S,$\bar{S}$] in N, where S is the set of all vertices to which there is a valid path from s. Of course K has the general form described in section 2.2; that is, the complementary pairs of vertices in N can be partitioned into the 4 sets A,B,C and D. Let us now once again adopt all the notation introduced in that section. Specifically, one should be familiar with the concepts represented by the terminology C_i, K_i, $\bar{K}_i$, K_r, $\bar{K}_r$, X_{AB}, F_{AB}, X_A and F_A before continuing.

Lemma 2.10

Every edge in $K_r \cup \bar{K}_r$ is saturated from S to $\bar{S}$.

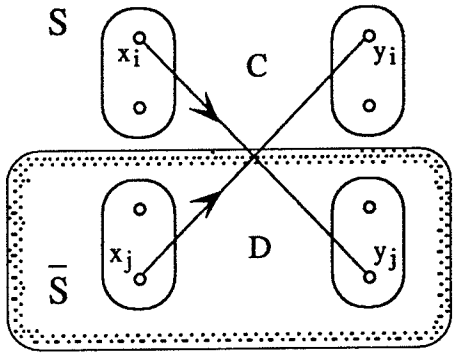
Proof:

Suppose that $uv \in K_r \cup \bar{K}_r$ is unsaturated from S to $\bar{S}$. Since uv is unsaturated, the edge $(uv)'$, and hence the vertex u', must be on every valid path to u, or there would be a valid path in N from s to $v \in \bar{S}$. Since $u \in S$ there is at least one valid path to u, so both u and u' are s-reachable; the complementary pair u,u' is therefore in C, so uv is in some $K_i \cup \bar{K}_i$. Thus $uv \notin K_r \cup \bar{K}_r$, a contradiction. □

Lemma 2.11

There are no edges between C and D.

Proof:

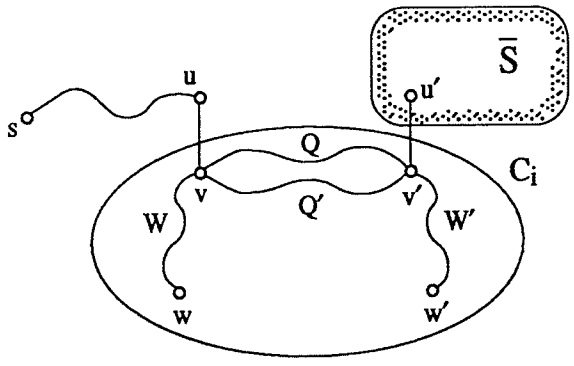


First notice that $D \subseteq \overline{S}$, so no vertex in D is s-reachable. Now suppose there is an edge $e=x_iy_j$ between C and D. Then $e'=x_jy_i$ is also such an edge, and without any loss in generality we may assume that $x_i, y_i \in C$ and $x_j, y_j \in D$. Either these two edges carry flow or they do not. If they do then x_jy_i is unsaturated, in a backward direction, from S to $\overline{S}$. But then x_j would be in S, not $\overline{S}$, since y_j is not on any valid path from s. If they do not carry flow, then x_iy_j is unsaturated from S to $\overline{S}$, so y_j would be in S; this is again a contradiction. □

Lemma 2.12

Each C_i consists entirely of pairs of complementary vertices.

Proof:



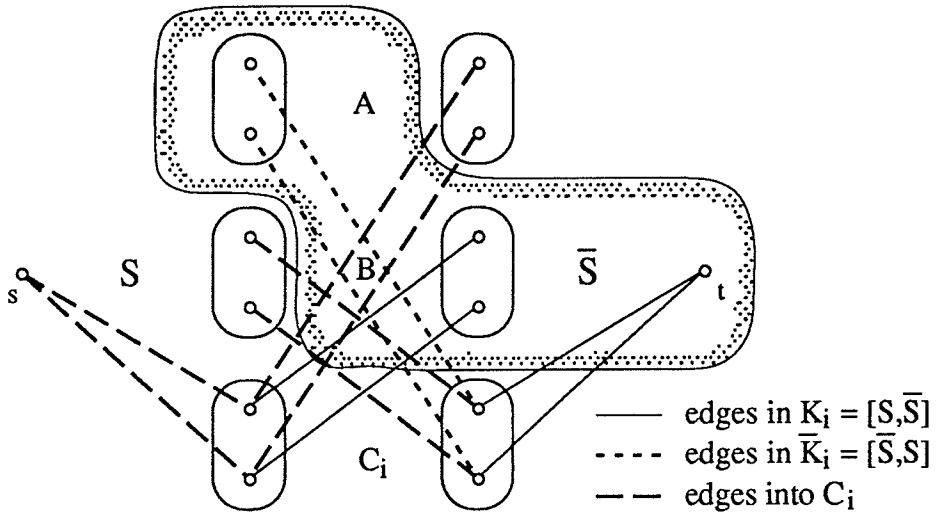
Let P be any valid path from s to $z \in C_i$, and suppose that P first enters C_i along the edge uv. Then either u=s, $u \in B_x$ or $u \in A_y$, and in all three cases $u' \in \overline{S}$. Also, $(uv)'=v'u'$ is unsaturated because uv is (by the rule of complementarity), so uv is on every valid sv′-path, or there would be a valid su′-path in N. Thus every valid sv′-path goes through v, and such a path P exists because $v' \in C$. But if Q is the sub-path of P that is a vv′-path, then Q′ is also a vv′-path. Thus every vertex on both Q and Q′ is in C, and therefore in C_i, so C_i contains at least one pair of complementary vertices.

Let $v,v' \in C_i$. If $w \in C_i$ then there is a path W connecting v to w, since C_i is connected. Thus W′ connects w′ to v′, so $w' \in C_i$. □

Lemma 2.13

At least one edge is unsaturated from S to $\overline{S}$ in each $K_i \cup \overline{K}_i$.

Proof:



Take an arbitrary component C_i, and any valid path P from s to a vertex in C_i. Let uv be the first edge on P where $v \in C_i$. $(uv)'=v'u'$ is unsaturated because uv is. $u \in S$, so if uv is a forward edge on P then either u=s or $u \in B_x$; in either case $v' \in S$,

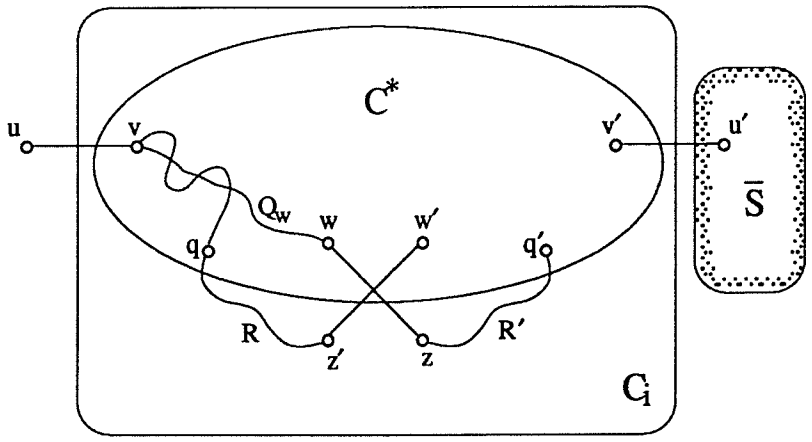
$u' \in \bar{S}$ and $(uv)' \in K_i$. If uv is a backward edge then $u \in A_y$, so $v' \in S$, $u' \in \bar{S}$ and $(uv)' \in \bar{K}_i$. In either case, $K_i \cup \bar{K}_i$ contains an edge that is unsaturated from S to $\bar{S}$. □

Theorem 2.14

Let $v \in C_i$ be a vertex such that there is a valid sv-path which first enters C_i at v. Then every $w \in C_i$ can be reached on a valid path Q_w from v, where $Q_w \subseteq C_i$.

Proof:

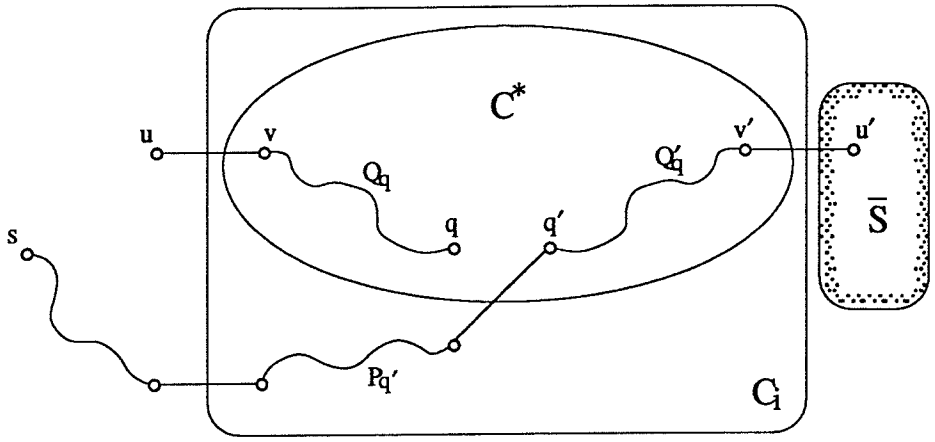
By contradiction. Let C^* be the largest non empty set of pairs of complementary vertices in C_i, with the property that every vertex $w \in C^*$ is reachable from v on a valid path. The paths Q and Q′ of lemma 2.12 are in C^*, so $C^* \neq \emptyset$.



Now suppose that $C^* \neq C_i$, so there is some edge wz such that $w \in C^*$ and $z \in C_i - C^*$; thus $w' \in C^*$ and $z' \in C_i - C^*$. If wz is saturated then zw, and hence w′z′, is unsaturated. Thus either wz or w′z′ is unsaturated from C^* to $C_i - C^*$, so suppose it is wz. $z' \in C_i$, so there is a valid sz′-path P.

Suppose $v \in P$. $v \in C^*$ and $z' \notin C^*$, so let P leave C^* for the last time from a vertex q, and let R be the portion of P from q to z′. Now let Q_w be a valid path from v to w that does not leave C^*. R′ is a valid path because R is. Hence if $Q_w$wzR′ is not a valid path, then R′ contains an edge e such that either e′=wz or e′ is on Q_w. It is clear that $e' \in Q_w$ is not possible, and if e′=wz then there is a valid zz′-path Z.

Hence Z' is also a valid zz'-path, so we can enlarge C^* with $V(Z \cup Z')$, a contradiction. Thus R' contains no such edge, so $Q_w wzR'$ is a valid path. But then we can enlarge C^* with $V(R \cup R')$, another a contradiction.

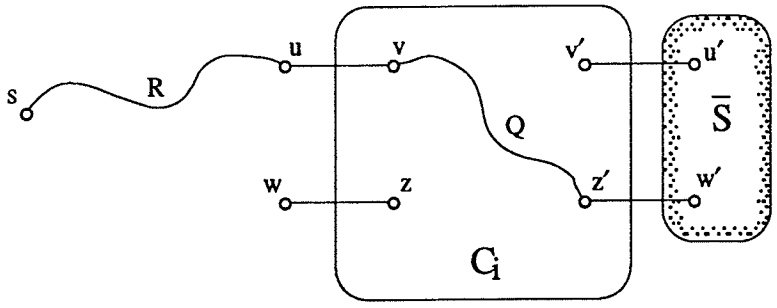


Now suppose $v \notin P$. If P does not enter C^*, then $Pz'w'$ is a valid path into C^*, so let q' be the first vertex at which $Pz'w'$ enters C^* (maybe $q'=w'$), and $P_{q'}$ be the portion of $Pz'w'$ up to q'. Then $P_{q'}Q'_q$ is a valid sv'-path that does not contain the edge uv, where Q_q is the valid vq-path that does not leave C^*. This is a contradiction, so $C^*=C_i$, as required. □

Lemma 2.15

At most one edge is unsaturated, by at most one unit, from S to $\bar{S}$ in each $K_i \cup \bar{K}_i$.

Proof:



If any edge of $K_i \cup \overline{K}_i$ is unsaturated from S to $\overline{S}$ by more than one unit then a vertex in $\overline{S}$ would be s-reachable, a contradiction. By theorem 2.14 there is always at least one such edge v′u′, where v is the only vertex in C_i on a valid sv-path R. Now suppose there is a second such edge, say z′w′. By theorem 2.14 there is a valid vz′-path Q that does not leave C_i. But then RQ is a valid sz′-path that does not use the edge wz, a contradiction. □

This proves that there is exactly one edge in each $K_i \cup \overline{K}_i$ that is unsaturated from S to $\overline{S}$. But by lemma 2.10 every edge in $K_r \cup \overline{K}_r$ is saturated from S to $\overline{S}$, so (cap(K) – flow(K)) + flow($\overline{K}$) = k. Since val(f) = flow(K) – flow($\overline{K}$), this proves corollary 2.16.

Corollary 2.16

val(f) = cap(K) – k.

Lemma 2.17

$cap(K_i)$ is odd, $1 \leq i \leq k$.

Proof:

By lemmas 2.13 and 2.15, either $flow(K_i) = cap(K_i)$ and $flow(\overline{K}_i) = 1$, or $flow(K_i) + 1 = cap(K_i)$ and $flow(\overline{K}_i) = 0$.

In the first case, $F_{AI} = F_{IA} = 1$. Thus $F_{II} + F_{IA} + F_{IB} - F_I = 1 + F_{II} + F_{IB} - F_I = 0$ by identity 2.1, and so $F_I = F_{II} + F_{IB} + 1$ (eqn 1). But then $flow(K_i) = F_{IB} + F_I = 2F_{IB} + F_{II} + 1$ (by eqn 1), which is odd because F_{II} is even. Since $cap(K_i) = flow(K_i)$, $cap(K_i)$ is therefore also odd.

In the second case, $F_{AI} = F_{IA} = 0$. Hence $F_{II} + F_{IA} + F_{IB} - F_I = F_{II} + F_{IB} - F_I = 0$ by identity 2.1, and so $F_I = F_{II} + F_{IB}$. Thus $flow(K_i) = F_{IB} + F_I = 2F_{IB} + F_{II}$ (eqn 2). But then $cap(K_i) = flow(K_i) + 1 = 2F_{IB} + F_{II} + 1$ (by eqn 2), which is odd because F_{II} is even. □

But by definition there are no edges between C_i and C_j when $i \neq j$, so by lemmas 2.11, 2.12 and 2.17 K is a balanced edge cut, and balcap(K) = cap(K) – k. Hence, by corollary 2.16, val(f) = balcap(K), so by lemma 2.3 f is a maximum balanced flow in N and K is a minimum balanced edge cut. Since the original assumption was that (N,f) does not have a valid augmenting path, and if such a path P does exist then f is not a maximum balanced flow, since augmenting on P and P′ will give a larger balanced flow, we have proved theorem 2.18. Some of these results are also used in the proofs of theorem 2.19 and the max-balanced-flow–min-balanced-cut theorem.

Theorem 2.18

Let N be a balanced network. f is a maximum balanced flow in N if and only if there is no valid augmenting path in N.

Theorem 2.19

If f is a *maximum* balanced flow in a balanced network N then $[S,\bar{S}]$ is a minimum balanced edge cut in N, where S is the set of all vertices in N to which there is a valid path from s.

Proof:

By theorem 2.18, if f is a maximum balanced flow in N then there is no valid augmenting path in (N,f). Thus, from the preceding discussion, $[S,\bar{S}]$ is a minimum balanced edge cut. □

It is possible that C=Ø for the minimum balanced edge cut $K = [S,\bar{S}]$ described in theorem 2.19. In this case k=0, so cap(K) = balcap(K). However, K is still both balanced, since it is trivial that any edge cut for which C=Ø is balanced, and minimum, by the max-flow–min-cut theorem.

Max-Balanced-Flow–Min-Balanced-Cut Theorem

Let N be a balanced network. The value of a maximum balanced flow in N is equal to the capacity of a minimum balanced edge cut.

Proof:

We can always find a balanced flow f in N such that there is no augmenting path in (N,f). We begin with the zero-flow in N, and continually augment on P and P′, where P is a valid augmenting path in (N,f), until no path such as P remains. The result now follows from the preceding discussion. □

2.4.3. Summary

To summarise, we may use the augmenting path technique as the basis for an algorithm to find a maximum flow in a balanced network, and this algorithm must take care of the following:

- we may only augment on an unsaturated st-path if it is valid; that is, it does not have the form s...uv...v′u′...t, where rescap(uv) = rescap(v′u′) = 1;
- when we augment the flow on a path P we also, immediately or simultaneously, augment the flow on P′.

3. A Maximum Balanced Flow Algorithm

3.1. The Algorithm

3.1.1. The Balanced Network Search and the Mirror Network

We must now try to develop an efficient way of finding a maximum flow in a balanced network. It is clear that the Ford-Fulkerson (or indeed any existing) max-flow algorithm does not work, since it has no mechanism for ensuring that we do not take an invalid, but none the less unsaturated, path.

Let us suppose for a moment that we have such a mechanism. The semantics of a breadth first search (BFS) dictate that it must not add to the search tree T any edge that introduces into T an invalid path. We could therefore use this mechanism to build a breadth first search tree that does not contain any invalid paths. This appears to form the basis for a maximum balanced flow algorithm in a balanced network. Unfortunately this is not so, and the reason is illustrated by the following example:

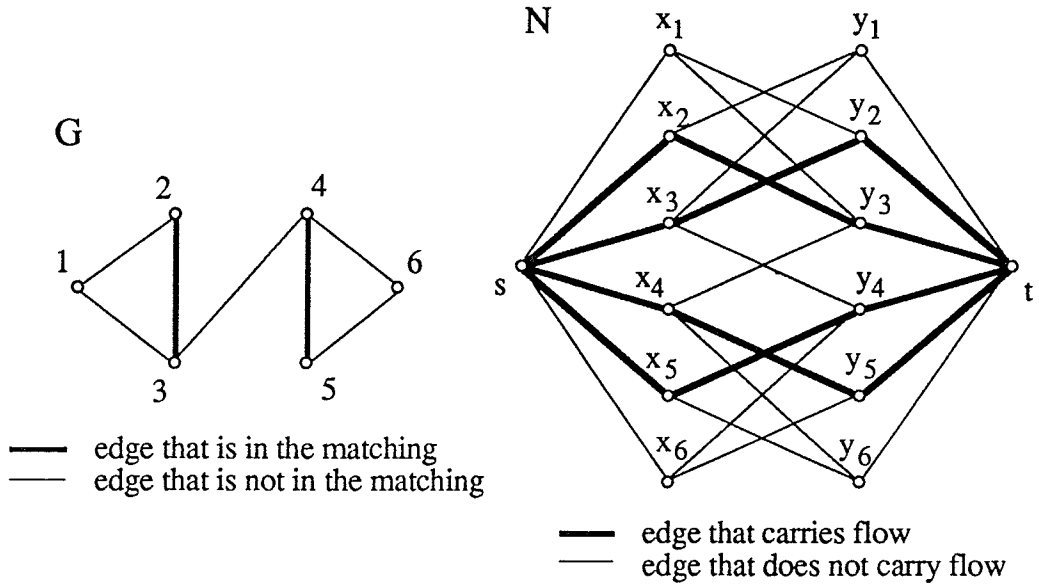


Figure 3.1
A balanced network N in which there is an augmenting path ($sx_1y_2x_3y_4x_5y_6t$). Every edge in N has capacity 1, and the thick edges carry flow.

Discussion of a balanced network N is often easier to follow when the graph G and subgraph H, corresponding to N and the flow f in N respectively, are available for reference. G and H are therefore shown for N and f in figure 3.1. H is a matching in G, and N is a balanced network which contains an augmenting path. There is a 1–1 correspondence between augmenting paths in G with respect to H, and pairs of complementary augmenting paths in N; the augmenting path $v_1v_2...v_r$ in G corresponds to $P = sx_{v_1}y_{v_2}...y_{v_r}t$ and P' in G. It is this relationship that makes illustration of G and H along with N useful. Now perform a BFS of N with the restriction mentioned above regarding invalid paths. We get the following search tree:

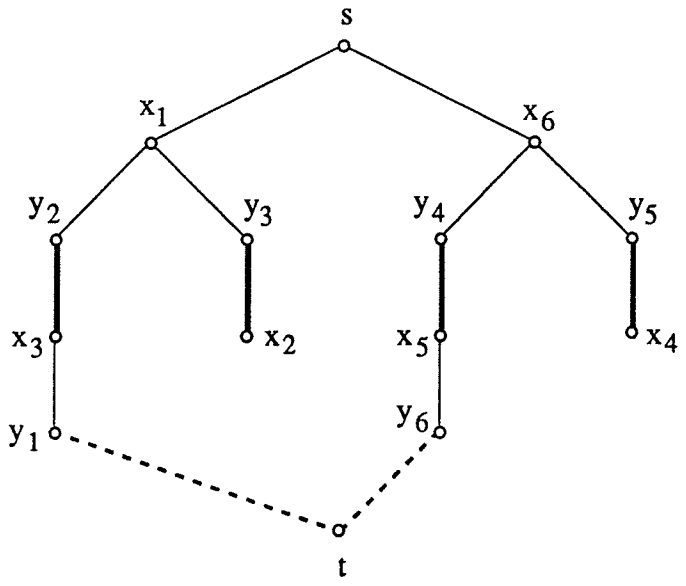


Figure 3.2
The breadth first search tree for the balanced network N in figure 3.1. Dotted lines indicate edges that could not be added to the tree.

y_1 has an unsaturated edge e to t, but is visited on a path that contains e′; similarly for y_6. Consequently, despite the fact that N contains an augmenting path, the search concludes that no such path exists.

The above problem can be solved by using a **Balanced Network Search** or **BNS** to search N, rather than a BFS. A BNS builds a search *network* called a

mirror network, rather than just a search tree. This network is a balanced sub-network of the residual network R(N,f), where f is the flow in N. The following pseudo code describes the BNS algorithm:

```
Data Structures
    ScanQ:queue – holds the vertices from which we still have to search
    SwitchEdge[vertex]:edge – the switch edge for a vertex
    sReach[vertex]:boolean – has u been searched (i.e. is u s-reachable?)
    { A:mirror network – the search network }
    { Note that E(A) is implicitly recorded in the SwitchEdge array, }
    { and V(A) is implicitly recorded in the sReach array }

Procedure BNS(N:balanced network)
1   begin
2   set all sReach[u] to false
3   set the ScanQ to Ø
4   place s on the ScanQ
5   sReach[s] := true
6   repeat
7       remove u from the ScanQ
8       for all v←u by an unsaturated edge e do
            { let rescap(e)=r }
9           if v has not been searched then
10              put v on the ScanQ
11              mark v searched        { which implicitly... }
                    { adds v to T and v′ to T′ (and thus adds v and v′ to A) }
12              SwitchEdge[v] := uv   { which implicitly... }
                    { adds uv with capacity r to T (and thus to A) }
                    { adds v′u′ with capacity r to T′ (and thus to A) }
13          else if uv is a switch edge then
                { another vertex w in A becomes s-reachable if uv is added to A, }
                { and maybe another vertex z in A becomes s-reachable if }
                { e′=v′u′ is added to A }
14              for all such vertices w do
15                  SwitchEdge[w] := uv       { which implicitly... }
                        { adds uv with capacity r to A  (if not yet added) }
                        { adds v′u′ with capacity r to A (if not yet added) }
16              for all such vertices z do
17                  SwitchEdge[z] := v′u′      { which implicitly... }
                        { adds uv with capacity r to A (if not yet added) }
                        { adds v′u′ with capacity r to A (if not yet added) }
18  until the ScanQ is empty
19  end { Procedure BNS }
```

A BNS makes use of the fact that in a balanced network there is a valid su-path if and only if there is a valid u't-path (proof: these two paths are complementary). As is the case with a BFS, a BNS builds a search tree T rooted at s and consisting of those vertices which are s-reachable, but a BNS also simultaneously builds a search tree T′ rooted at t and consisting of those vertices which are t-reachable. This second tree is a mirror image of the first, and the complementary partner of any vertex or edge in T is its image in T′. This means that the direction of the edges is reversed in T′; thus we can only follow paths down T and up T′. These two trees, along with some extra edges, form the mirror network A. A is a sub-network of R(N,f), the residual network for N, and is itself balanced, as is R(N,f).

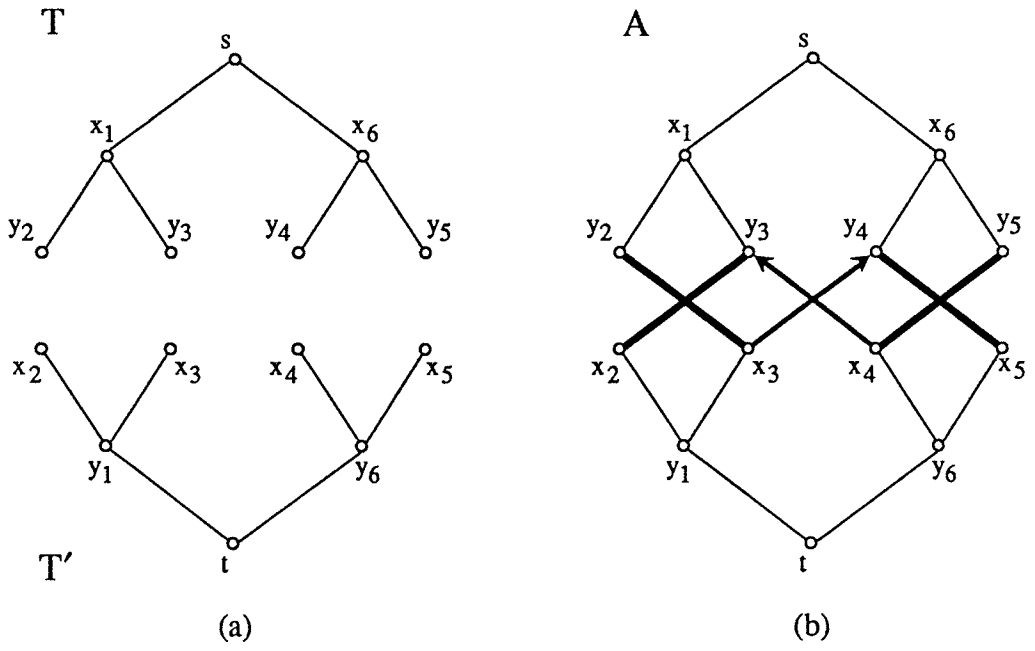


Figure 3.3
Example of a mirror network. (a) shows the search trees T and T′ in the mirror network A, which is shown in (b). A was created by a BNS of the balanced network N in figure 3.1.

Figure 3.3 illustrates what happens when we perform a BNS of the balanced network N in figure 3.1, thus creating a mirror network A. Part (a) shows the two search trees that extend from s and t. Each vertex in V(A) appears in exactly one of

these trees, and vertices are always added to A in complementary pairs; one vertex goes into T, the other into T′. Edges are also added to A in complementary pairs, and if an edge e goes into T then e′ goes into T′. However, A usually contains edges that are not in either T or T′. It is clear then that neither T nor T′ contains an invalid path, so all the vertices in T are s-reachable and all the vertices in T′ are t-reachable. This illustrates the two different types of path in A: those paths that extend from s and make vertices s-reachable, and those that extend to t and make vertices t-reachable. The complementary path of any path from s to u in T is always from u′ to t in T′. Part (b) shows the complete mirror network A, which includes some edges that are not in T or T′. A contains the augmenting paths $P = sx_1y_2x_3y_4x_5y_6t$ and $P' = sx_6y_5x_4y_3x_2y_1t$ in N.

3.1.2. SwitchEdges

In order to fully understand how a mirror network A is created, one must understand how the paths in A are recorded. For this purpose it is useful to draw an analogy between a BFS and a BNS. A BFS of a graph G builds a search tree T rooted at r, and uses T to find and record an rv-path, if one exists, for every v∈ V(G). A BNS searches a balanced network N with a balanced flow f; it builds a mirror network A with a source s, and uses A to find and record a valid sv-path, if one exists, for every v∈ V(G).

Consider how a BFS works. Every time it finds an edge e = uv that creates a path from r to v, where there is currently no such path in T, it adds e to T. It records these paths using PrevPt[], which effectively records the *edge* e=uv for vertex v, since we cannot follow a path unless we know which *edges* to traverse. Thus a BFS, via PrevPt[], conceptually records an *edge* e=uv for each vertex v∈ T. The only reason that PrevPt[v], which is a vertex, is sufficient to record e, which is an edge, is that v is always an endpoint of the edge.

Since we are searching for a *valid* st-path in N, whenever we discover a valid sv-path, where $v \in V(N)$, we should add this path into the mirror network A so that we may then visit v and possibly find a valid path from s to yet another vertex. In figure 3.3 (b) the last pair of complementary edges added to A were x_3y_4 and x_4y_3. Addition of x_3y_4 to A created the valid st-path $sx_1y_2x_3y_4x_5y_6t$ and such a path did not exist beforehand. Before x_3y_4 is added to A, x_3 is *s-reachable* on a valid path R and y_4 is *t-reachable* on a valid path Q. x_3y_4 switches us from R to Q; that is, from a valid path extending from s to a valid path extending to t; thus we say that x_3y_4 is the *switch edge* of t, since t becomes s-reachable when x_3y_4 is added to A. We therefore set SwitchEdge[t] equal to x_3y_4. (Observe that addition of x_4y_3 also creates a valid st-path in A, so we could equally well set SwitchEdge[t] equal to x_4y_3 instead.) This is in many ways analogous to PrevPt[] in a BFS. PrevPt[v] in a BFS records the first edge e encountered, such that adding e to the search tree T creates in T an rv-path. SwitchEdge[v] in a BNS records one of the edges e encountered, such that adding e to the mirror network A creates in A a valid sv-path. However, in the first case it is guaranteed that v is always an endpoint of the edge, but in the second case, as is illustrated by SwitchEdge[t] above, this is not so.

PrevPt[] in a BFS gives us a recursive method for following the path from r to v; if PrevPt[v]=u, then the path from r to v is the path from r to u concatenated with the edge uv. SwitchEdge[] in a BNS gives us a slightly more complex recursive method for following the path from s to v; if SwitchEdge[v]=wz then the valid path from s to v is the valid path from s to w concatenated with the edge wz concatenated with the valid path from z to v.

Definition:

In a BNS of a balanced network N, a **switch edge** is any edge e encountered, such that if e is added to A then A will contain another s-reachable vertex, where A is the mirror network created by the BNS.

Such an edge e is a switch edge for every such vertex v, but the algorithm may not make the assignment SwitchEdge[v] := e for every such v. This is because only one of the potentially many switch edges for each vertex v in A is recorded by the BNS, and although this is usually the first switch edge encountered, situations do arise where this is not the case. It really does not matter which switch edge is recorded for v, but if v has a switch edge then one must be recorded. (It would be nice if we could efficiently record the one that gives us the shortest valid path to v, but that is beyond the scope of this algorithm, and a nice future research topic.)

The SwitchEdge array is used to record the paths in a mirror network A, just as PrevPt[] is used to record the paths in a search tree. From the definition of a switch edge, it is clear that *every* edge in A is a switch edge, not just those in E(A) – E(T)∪E(T′). However, we shall see later that switch edges are divided into two types, and these two types are closely related to this partitioning of the edges in A.

It is easy to see what SwitchEdge[v] is for all vertices v that are in T. Statement 12 in the body of the compound statement 9 in the BNS sets SwitchEdge[v] = uv whenever an unsearched vertex v is found to be the other endpoint of an edge incident on a searched vertex u. Since every such vertex v is added to T (and v′ is added to T′), T is just a breadth first search tree, and SwitchEdge[v] is the edge {PrevPt[v],v} that would be recorded by a breadth first search. That is, SwitchEdge[v] is just uv, where u is the parent of v in T. However, an example will serve to solidify the idea of a SwitchEdge for the other vertices in A; that is, the vertices in T′.

Refer to figure 3.3 and suppose the BNS has reached the point where it has created T and T′ exactly as depicted in part (a). At this point, if v is in T′ then there is no valid sv-path in A=T∪T′, so SwitchEdge[v] has not yet been recorded for any vertex v in T′. This appears to illustrate a difference between PrevPt[] in a BFS and SwitchEdge[] in a BNS. In a BFS of a graph G, PrevPt[v] is defined for a vertex v only if v is in the search tree T, but in a BNS of a balanced network N the search network A usually contains vertices v for which SwitchEdge[v] is not defined. However, SwitchEdge[v] is defined for a vertex v in A if v is s-reachable, which is completely analogous to the BFS, in which PrevPt[v] is defined for a vertex v if there is a path from r to v.

Now suppose the BNS visits vertex y_2 next. Then the edge $e=y_2x_3$ is examined and, since there will be a valid sx_3-path in A if y_2x_3 is added, the BNS makes the assignment SwitchEdge[x_3] := (y_2,x_3). Also, there is a valid sx_2-path in A if $e'=y_3x_2$ is added to A, so the assignment SwitchEdge[x_2] := (y_3,x_2) is made. The BNS now makes the assignment SwitchEdge[y_1] := either (y_2,x_3) or (y_3,x_2). This is because there is a valid sy_1-path P in A if $e=(y_2,x_3)$ is added, and there is also a valid sy_1-path P^* in A if $e=(y_3,x_2)$ is added. If we let P=RQ, where R is the portion of P from s to $(y_1)'=x_1$, and similarly let $P^*=R^*Q^*$, then $Q^*=Q'$. This not a coincidence. In general, if SwitchEdge[u] = (x_i,y_j), and the su-path in the mirror network goes through u′, then (x_j,y_i) is also a valid SwitchEdge for u. Furthermore, if u=t and we put P, the su-path created by (x_i,y_j), into the form P=RQ as above, then R=Ø and the su-path created by (x_j,y_i) is P′. This gives us the two complementary augmenting paths in N.

Continuing with the example depicted in figure 3.3, we can see that the BNS marks as s-reachable the vertices x_2,x_3 and y_1, and so these vertices are subsequently searched. It should now be easy to verify that as the BNS proceeds it eventually makes the following assignments: SwitchEdge[x_4] := (y_5,x_4), SwitchEdge[x_5] := (y_4,x_5), SwitchEdge[y_6] := either (y_4,x_5) or (y_5,x_4) and SwitchEdge[t] := either (x_3,y_4) or

(x_4,y_3). This last assignment indicates that t is s-reachable, so we have found an augmenting path.

3.1.3. The Paths in a Mirror Network

A word about the paths in a mirror network A is now in order. If the BNS discovers that wz is a switch edge for a vertex v, then there must be a valid sw-path and a valid zv-path in A. However, it is apparent that the SwitchEdge array records only valid sv-paths, and every vertex on every such path is s-reachable. How then is it possible to determine that there is a valid zv-path in A, when v is not s-reachable? It would appear that there is no such path in A, since no such path is recorded in the SwitchEdge array. (Of course there is such a path if v=z, but this is often not the case.)

The answer lies with the *complementary paths* of the paths recorded by the SwitchEdge array. In the above examples two trees, namely T and T′, are built as part of the process that creates the mirror network A, and it is assumed that the paths in T′ are part of A. But when v and v′ are added to A, where $v \in T$ and $v' \in T'$, only SwitchEdge[v] is recorded, so it appears that this assumption is incorrect. However, every path in T′ is the complementary path of a path in T. Thus the well defined relationship between pairs of complementary paths allows us to traverse the paths in T′, so these paths are indeed paths in A.

This is a special case of the general situation with respect to paths in A. The switch edge array records only valid sv-paths, and such a path is recorded if and only if SwitchEdge[v] is defined. However, if P is a valid sv-path then P′ is a valid v′t-path. Thus there is a valid sv-path in A if and only if SwitchEdge[v] is defined, and a valid vt-path in A if and only if SwitchEdge[v′] is defined.

As mentioned earlier, there are two different types of path in A. The first is a valid sv-path for some vertex v, which is recorded directly or explicitly in the SwitchEdge array; this is referred to as an **explicit path** in A. The second is a valid vt-path, which

is the complementary path of the explicit sv′-path in A, which is therefore recorded indirectly or implicitly in the SwitchEdge array; this is referred to as an **implicit path** in A. It is not possible to use the SwitchEdge array to follow an implicit path P′ directly; we must follow the explicit path P and deduce P′.

A subtle point regarding the paths in A should now be described. If T is a search tree, then the only paths in T that we are interested in are those that go from root to leaf; furthermore, we are only interested in sub-paths of these paths. T is really a collection of vertices and *paths*, rather than vertices and edges, and the edges in T are only considered in the context of these paths. A mirror network A is analogous to a search tree in this regard. A is also a collection of vertices and *paths*, and not vertices and edges. Thus we consider P to be a path in A only if P is recorded in the SwitchEdge array, just as we consider Q to be a path in T only if it is recorded in the PrevPt array. This point is made because while it is obvious that one does not want to go up and down a search tree, the paths in a mirror network are far less easily identified.

Since the paths in A are clearly directed, this tells us that if v∈ V(A) then there is at most one path that ends at v in A, namely the explicit sv-path, and at most one path that begins at v, namely the implicit vt-path. A path of length zero in A is clearly both implicit and explicit. These observations help us to prove lemma 3.1.

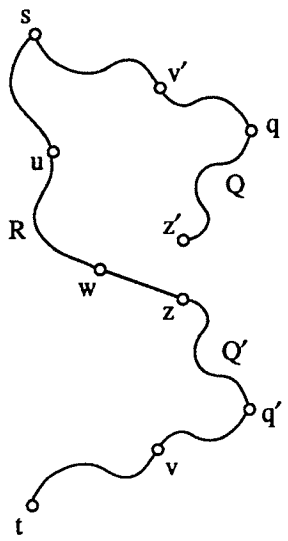
Lemma 3.1

If P is a non-empty explicit uv-path in A, then P has the form RwzQ′, where SwitchEdge[v] = wz, R is an explicit uw-path and Q is an explicit v′z′-path.

Proof:

The proof is by induction on the number of switch edges in A. If there is one switch edge in A then it is SwitchEdge[v] = sv for some v (recall that every edge in A is a switch edge). The only non empty explicit path in A is therefore the sv-path

of length one, and this has the form RsvQ′, where R and Q′ are both paths of length zero.



Now suppose the theorem is true whenever A has at most k switch edges. Consider what happens when a k+1st switch edge is added, say SwitchEdge[v] = wz, which creates a valid sv-path in A. If adding wz to A creates a valid uq-path P in A, for any pair of vertices u and q, then P must use wz; P therefore has the form P = RwzQ′, where R is a valid path that ends at w and Q′ is a valid path that begins at z. But from the foregoing discussion R must be a (possibly improper) sub-path of the explicit sw-path in A, and is therefore itself explicit. Similarly, Q′ is implicit, so P has the required form. □

This illustrates the recursive structure of the explicit paths in A, which gives us the method by which we can traverse valid augmenting paths in a mirror network, and thus augment a balanced flow.

Lemma 3.2

Let v be a vertex that is not t-reachable, P=RQ be an explicit sw-path that enters v, and R be portion of P from s to v. Then there is an explicit path from v to every vertex on Q.

Proof:

Every vertex in T′ is t-reachable, so v is in T. Hence SwitchEdge[v] = uv for some vertex u. Since v is not t-reachable it is not on any implicit path, so the explicit sv-path V must be part the sq-path to every vertex q on Q. But in order to follow V we must examine SwitchEdge[v]; hence we must follow a vq-path, and this path is clearly explicit. □

3.1.4. Augmenting a Balanced Flow

It is assumed that a BNS has been performed on a balanced network N, so there is a mirror network A and the value SwitchEdge[v] is recorded for every s-reachable vertex v in A. It is also assumed that t is s-reachable, so SwitchEdge[t] is defined. Then the following recursive procedure gives a pseudo code description of an algorithm to augment a balanced flow in a balanced network.

```
Procedure PullFlow(u,v:vertex)  { pull flow from u down to v }

1  begin
2  if u = v then    { augment flow on path of length 0 }
3      return
4  else
       let SwitchEdge[v] be wz
5      simultaneously augment the flow on e=wz and e′=z′w′
6      PullFlow(u,w)
7      PullFlow(v′,z′)  { note – realises PullFlow(z,v) }
8  end { Procedure PullFlow }
```

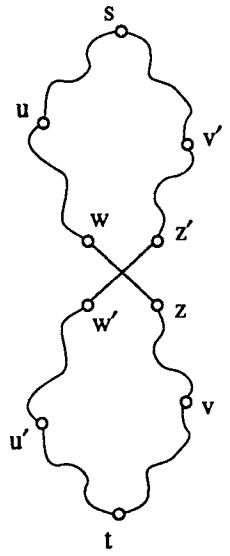


Figure 3.4
The paths on which the flow is augmented by procedure PullFlow, where SwitchEdge[v] = wz.

When it is discovered that t is s-reachable, an augmenting path has been found; SwitchEdge[t] is assigned a value and the balanced network maximum balanced flow algorithm executes the call PullFlow(s,t). This call simultaneously augments the flow on P and P′, where P is the explicit st-path in A, by recursively constructing P and P′ from P. In general, whenever the call PullFlow(u,v) is made there is an explicit uv-path V, so by lemma 3.1 V=RwzQ′, where R and Q′ are explicit paths. The call PullFlow(u,w) augments the flow on R and R′, the call PullFlow(v′,z′) augments the flow on Q′ and Q, and the current call augments the flow on wz and z′w′. Thus the call PullFlow(u,v) augments the flow on exactly V and V′ (see figure 3.4).

An example should clarify this. Consider again the mirror network in figure 3.3 (b). The switch edge for every vertex v∈A was given earlier, and figure 3.5 shows A with these switch edges given in square brackets. SwitchEdge[t] is defined, so there is an explicit st-path in A, and so we can use PullFlow to augment the balanced flow in A. Figures 3.6 and 3.7 illustrate the recursive calls that are executed during this process.

The pair of complementary augmenting paths recorded by the SwitchEdge array are $P = sx_1y_2x_3y_4x_5y_6t$, the explicit st-path in A, and $P' = sx_6y_5x_4y_3x_2y_1t$, the implicit st-path, and it is exactly the edges on these two paths that have their flow augmented.

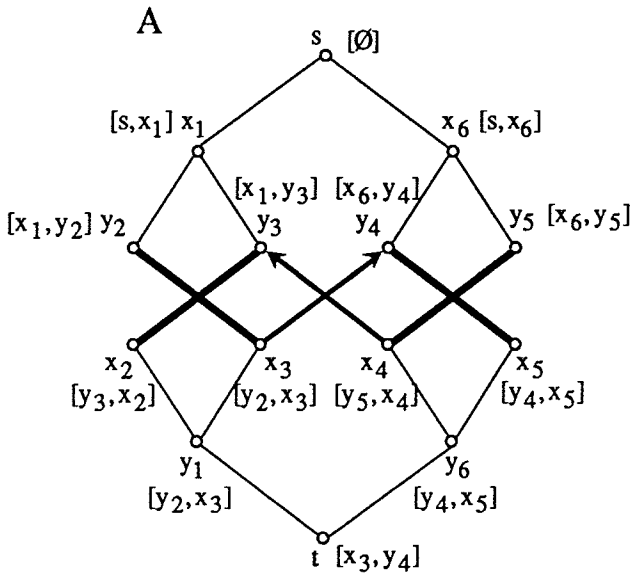


Figure 3.5
A mirror network A with SwitchEdge[v] shown in square brackets for every vertex $v \in V(A)$.

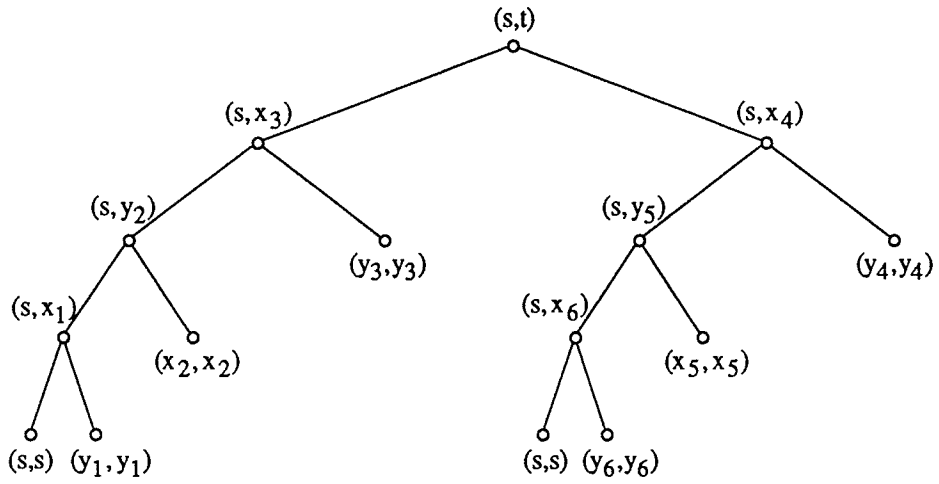


Figure 3.6
The tree of recursive calls that will result from the call PullFlow(s,t) for the mirror network A in figure 3.5. (u,v) denotes the recursive call PullFlow(u,v).

PullFlow(s,t) { x_3,y_4 and x_4,y_3 }
PullFlow(s,x_3) { y_2,x_3 and y_3,x_2 }
PullFlow(s,y_2) { x_1,y_2 and x_2,y_1 }
PullFlow(s,x_1) { s,x_1 and y_1,t }
PullFlow(s,s)
PullFlow(y_1,y_1)
PullFlow(x_2,x_2)
PullFlow(y_3,y_3)
PullFlow(s,x_4) { y_4,x_5 and y_5,x_4 }
PullFlow(s,y_5) { x_5,y_6 and x_6,y_5 }
PullFlow(s,x_6) { y_6,t and s,x_6 }
PullFlow(s,s)
PullFlow(y_6,y_6)
PullFlow(x_5,x_5)
PullFlow(y_4,y_4)

Figure 3.7
This shows how the flow on the edges on the two augmenting paths in A is augmented by the above recursive calls. The edges on which the flow is directly augmented by a particular call are shown in curly brackets and the levels of recursion are indicated by the indentation.

Before we can verify that PullFlow works properly, we must define precisely what it does. It augments the flow on all edges on a path P from u to v, where P is an *explicit* path in A, and in so doing simultaneously augments the flow on the edges on P′. The order of P and P′ in this definition is important, since it is not possible to use the SwitchEdge array to follow an implicit path directly, and P′ is such a path.

Lemma 3.3

If P is an explicit uv-path in A, then PullFlow(u,v) augments the flow on exactly P and P′.

Proof:

The proof is by induction on the length of P. If the length of P is zero then PullFlow executes the easy case, which clearly works by doing nothing.

Now for the recursive case. Let SwitchEdge[v] = wz. By lemma 3.1 P has the form P = RwzQ′, where R is an explicit valid uw-path and Q is an explicit valid v′z′-path. If P contains k edges then, since wz is an edge on P, the paths R and Q′ each contain less than k edges. Thus, by the induction hypothesis, the call PullFlow(u,w) at statement 6 augments the flow on R and R′, and the call PullFlow(v′,z′) at statement 7 augments the flow on Q′ and Q (observe that this would not be the case if the call PullFlow(z,v) was executed, since the zv-path is *implicit*). The flow on wz and z′w′ is augmented in statement 5. □

Thus, provided that there is an explicit valid st-path P when the call PullFlow(s,t) is made, this call will augment the flow on exactly P and P′, thus leaving the flow in N balanced. This gives us corollary 3.4.

Corollary 3.4

PullFlow works.

3.1.5. Blossoms and Recognition of SwitchEdges

The following arguments assume that every edge in the balanced networks we subsequently consider has capacity 1. This supposition is made merely for the purposes of simplicity, to enhance clarity in the following discussion; all the following results are independent of it, and consequently also apply when some edges have capacities that exceed one.

We now have the basis for a maximum balanced flow algorithm; the only part that is missing is a framework in which we can recognise the switch edges as we encounter them. That is, when we examine an edge e, we need to be able to tell immediately whether adding e to A would make another vertex in A s-reachable. Let us therefore now develop such a framework.

If we discover an edge e to a vertex that is not in A, then it is obvious that e is a switch edge and should be added to T, and thus to A; let us therefore concentrate on recognising whether or not an edge between vertices that are both already in A is a switch edge. The basic idea is to partition the vertices of A in such a way that an edge between two vertices in the same partition can never be a switch edge. Then as we add edges to A we amalgamate partitions, thus reducing their number and hence the number of edges in N that are potentially switch edges. It would seem that putting each vertex into a different partition would be a good initial partitioning of A; however, it turns out that we want these partitions, which we will call blossoms, to consist entirely of *complementary pairs* of vertices; thus we initially place every *complementary pair of vertices* into a different partition. This is quite feasible, since vertices are always added to A in complementary pairs.

For the purpose of giving some insight into the following work, it is worth pointing out that when the max-balanced-flow algorithm, which is presented shortly, terminates, the set of k blossoms in the final mirror network that the algorithm builds are actually the k C_i described in Chapter two; that is, the k connected components of N[C]. This is the reason that the blossoms must consist entirely of complementary pairs of vertices.

Now let us develop the method by which these initial partitions are merged to form a larger partition. Let uv be an edge between vertices that are both in A. Every vertex in T is already s-reachable and every vertex in A is in either T or T′; thus if some vertex w′ becomes s-reachable when we add uv to A, then $w' \in T'$. The explicit sw′-path in A begins with the explicit su-path U, follows the edge uv, then ends with a sub-path of the implicit vt-path V′, which is the *complementary path* of the explicit sv′-path V.

Since we want to identify all vertices that become s-reachable upon addition of uv to A, we would like to know just how far we can follow V′. We can certainly follow it as far as w′, and we can continue to follow it so long as the complementary edge of

e=b′a′, the next edge we would like to follow on V′, is not on U; that is, e′=ab is not on U (see figure 3.8).

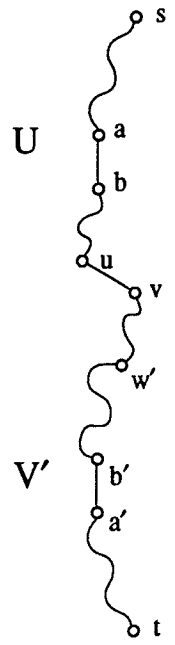


Figure 3.8
We can continue following V′ until we reach b′.

When we add uv to A, this gives us a valid sb′-path of the form RQ, where R is an sb-path and Q is a bb′-path. It turns out that we should amalgamate all the partitions on Q and Q′ into one big partition, and the pseudo code for function ExtendLongBase, which does just that, is presented below. ExtendLongBase uses the concept of the **depth** of a vertex to accomplish this.

$$\text{depth}[v] = \begin{cases} 0 \text{ if } v=s; \\ \text{Depth}[u]+1 \text{ if } v\neq s \text{ and } v\in T, \text{ where SwitchEdge}[v]=uv; \\ \text{Depth}[v'] \text{ if } v\in T'. \end{cases}$$

Depth[v] is defined for every vertex v∈A and is never altered once assigned, and the depth of s-reachable vertices in T′ is defined before they are found to be s-reachable. Thus, unlike a similar concept in a BFS, it can not help one measure the lengths of the paths from s to an s-reachable vertex.

We also need a mainline for the algorithm. This is easy though; it is just a BNS inside a loop, where each iteration of the loop finds an augmenting path P in N and uses P and P′ to augment the balanced flow. Thus the following pseudo code completes the algorithm.

Data Structures

ScanQ:queue – holds the vertices from which we still have to search
LongBase[vertex]:vertex – the LongBase of u
PrevPt[vertex]:vertex – parent of u in T or T′
sReach[vertex]:boolean – is u s-reachable?
SwitchEdge[vertex]:edge – the switch edge for u
Depth[vertex]:integer – the depth of a vertex in A

Mainline

```
1  begin
2  Repeat
3      set the ScanQ to Ø
4      set all LongBase[u] to -1   { or any invalid value }
5      set all sReach[u] to false   { tReach[u] = sReach[u'] }
6      put s on ScanQ
7      sReach[s] := true
8      LongBase[s] := t
9      LongBase[t] := t
10     Depth[s] := 0
11     Depth[t] := 0
12     augmented := false
13     Repeat
14         remove u from ScanQ       { u = x_i or y_i }
15         uBase := FindBase(u)
16         for all v←u by an unsaturated edge do
17             vBase := FindBase(v)
18             if vBase = -1 then    { v not in either tree }
19                 PrevPt[v'] := u'  { needed by ExtendLongBase }
20                 Depth[v] := Depth[u] + 1   { ditto }
21                 Depth[v'] := Depth[v]
22                 put v on ScanQ
23                 sReach[v] := true
24                 LongBase[v] := v'  { LongBase entries are in bottom tree }
25                 LongBase[v'] := v'
26                 SwitchEdge[v] := (u,v)
27             else if (uBase ≠ vBase) and (sReach[v']) then
                       { uv may be a switch edge }
28                 if (v is not parent of u in lower tree) or (Rescap[uv]>1) then
                           { we will not add an invalid path into A }
29                     uBase := ExtendLongBase(uv)
30                     if uBase = t then   { augmenting path found }
31                         PullFlow(s,t)
32                         augmented := true
33     Until (ScanQ is empty) or (augmented)
34 Until not augmented
35 end  { Mainline }
   { the balanced flow is now maximum }
```

```
Function ExtendLongBase(uv:edge):vertex
    { assigns switch edges and merges a set of blossoms into a single blossom }
1   begin
2   p := FindBase(u)
3   q := FindBase(v)
4   while (p≠q) do
5       if Depth[p] > Depth[q] then
6           p := FindBase(PrevPt[p])
7       else
8           q := FindBase(PrevPt[q])
9   w := p   { common base }
10  p := FindBase(u)
11  while (p ≠ w) do
12      LongBase[p] := w
        { Note: it is possible to hit some non s-reachable vertices and then run }
        { into some that are s-reachable (on this path back to the common base) }
13      if not sReach[p] then
            { switch edge always set at time vertex is marked s-reachable }
14          SwitchEdge[p] := (v',u')
15          sReach[p] := true
16          put p on ScanQ
17      p := FindBase(PrevPt[p])
18  q := FindBase(v)
19  while (q≠w) do
20      LongBase[q] := w
21      if not sReach[q] then
22          SwitchEdge[q] := (u,v)
23          sReach[q] := true
24          put q on ScanQ
25      q := FindBase(PrevPt[q])
    { w may already be s-reachable }
26  if not sReach[w] then
27      SwitchEdge[w] := (u,v)   { could equally well be complementary edge }
28      sReach[w] := true
29      put w on ScanQ
30  return(w)
31  end { Function ExtendLongBase }
```

The blossoms in A change dynamically as the algorithm progresses. Blossoms are therefore algorithm dependent structures, so they are defined in terms of the algorithm.

Definition:

Let N be a balanced network and A be the mirror network that is being created by a BNS of N. The algorithm changes the **blossoms** in A as follows:

a) The blossom $\{v,v'\}$ is added to A by statements 24 and 25 in the mainline;

b) Let $B_0,B_1,...,B_k$ be the set of blossoms visited in statements 6 or 8 by a single execution of the function ExtendLongBase (ELB). Then an execution of ELB adds to A the blossom $B = B_0 \cup B_1 \cup ... \cup B_k$, and removes from A the blossoms $B_0,B_1,...,B_k$.

It follows from the definition that blossoms partition the vertices in A. Any blossom that contains two vertices is called a trivial blossom. Thus it is clear that trivial blossoms are added to A only under part a) of the definition, and non-trivial blossoms are added to A only under part b), since $k \geq 2$.

Whenever the algorithm discovers an edge uv that could be a switch edge, and both u and v are already in A, ELB is called (statement 29 in the mainline). As mentioned earlier, the key property of blossoms is that an edge between two vertices in the same blossom is never a switch edge (see property 5 in section 3.1.7); thus statement 27 in the mainline checks that u and v are in different blossoms with the conditional uBase≠vBase. But if uv is a switch edge then adding uv to A must create in A a valid path from s to a vertex q, where no such path already exists. From the discussion on the paths in A, there must therefore be an explicit su-path and an implicit vt-path in A. The explicit su-path exists only if u is s-reachable; this is clearly so, because the algorithm only visits s-reachable vertices. The implicit vt-path exists only if v is t-reachable; thus the need for the second conditional in statement 27.

Function ExtendLongBase serves two purposes. The first is to set SwitchEdge[q] = uv for all vertices q for which uv is a *possibly unique* switch edge. In fact ELB usually sets SwitchEdge[q] = uv for *all* vertices q for which uv is a switch edge, but as noted in the definition of a switch edge, situations do arise where ELB will not assign uv as the switch edge of a vertex such as q. However, whenever this happens it is always *guaranteed* that the algorithm will identify another switch edge wz for q, *and make the assignment SwitchEdge[q] := wz*. The second purpose of ELB is to dynamically alter the blossoms in A. An execution of ELB reduces the total number of blossoms in A by at least one, since it merges, and thus deletes, at least two blossoms in A to form a new blossom. This is important to the complexity of the algorithm. It is possible for the call ExtendLongBase(u,v) to be executed when uv is not a switch edge, but it is now clear that such a call is still useful.

Definition:

The **base** of a blossom B is:

$$\begin{cases} v' \text{ if } B = \{v,v'\}, \text{ where } v' \in T'; \text{ that is, if B is trivial;} \\ \text{the vertex w from statement 9 of ExtendLongBase otherwise.} \end{cases}$$

Blossoms are implemented in the algorithm with a merge find data structure (see [2] for a description of this data structure). Each blossom is represented by its base, and **LongBase**[v] is the base of the blossom containing the vertex v. Since ELB updates only LongBase[b] when a blossom B is being merged into a larger blossom, where b is the base of B, the LongBase array becomes out of date. We therefore need the function FindBase(u), which takes as input a vertex u, returns the base of the blossom containing u, and updates all the LongBase[] entries it traverses in the process.

```
Function FindBase(v:vertex):vertex

1  begin
2  if (LongBase[v] = -1) or (LongBase[v] = v) then
   { (v is unsearched) or (v is the base) }
3      return(LongBase[v])
4  else
5      u := FindBase(LongBase[v])
6      LongBase[v] := u        { path compression }
7      return(u)
8  end { Procedure PullFlow }
```

Cursory examination of the body of the 'while' loop which is statement 4 in ELB tells us that p and q are always the base of a blossom. Since every initial blossom has its base in T', a simple inductive argument shows us that the base of a non trivial blossom is therefore also in T'; thus the base of a blossom must always be in T'.

3.1.6. Some Examples

It is now time to look at some examples: the first illustrates some blossoms built by ExtendLongBase, the second further illustrates the structure of T and T' and the third portrays the structure of blossoms. In order to follow these examples it is necessary to know that every vertex in a non-trivial blossom is both s-reachable and t-reachable. (See corollary 3.6, which is presented later, and note that if v is in a non-trivial blossom B then so is v'; thus if there is a valid sv'-path P in A, then P' is a valid vt-path.)

Consider the example given in figure 3.9. The first non trivial blossom formed by the algorithm is $D=\{x_7,y_{11},x_{12},y_7,y_{12},x_{11}\}$ with base y_7, formed when the edge $y_{11}x_{12}$ is discovered. Then discovery of the edge $y_{13}x_{14}$ will cause the algorithm to create the blossom $C=\{x_8,y_{13},x_{14},y_8,y_{14},x_{13}\}$ based at y_8. Since x_{12} is now s-reachable it is searched and the edge $e=x_{12}y_{13}$ examined. e goes between two different blossoms, D and C, and y_{13} is t-reachable, so the call

ExtendLongBase(x_{12},y_{13}) is made. A new non-trivial blossom B is formed by merging blossoms C,D and $B_i = \{x_i, y_i\}$, for i = 3,4,1.

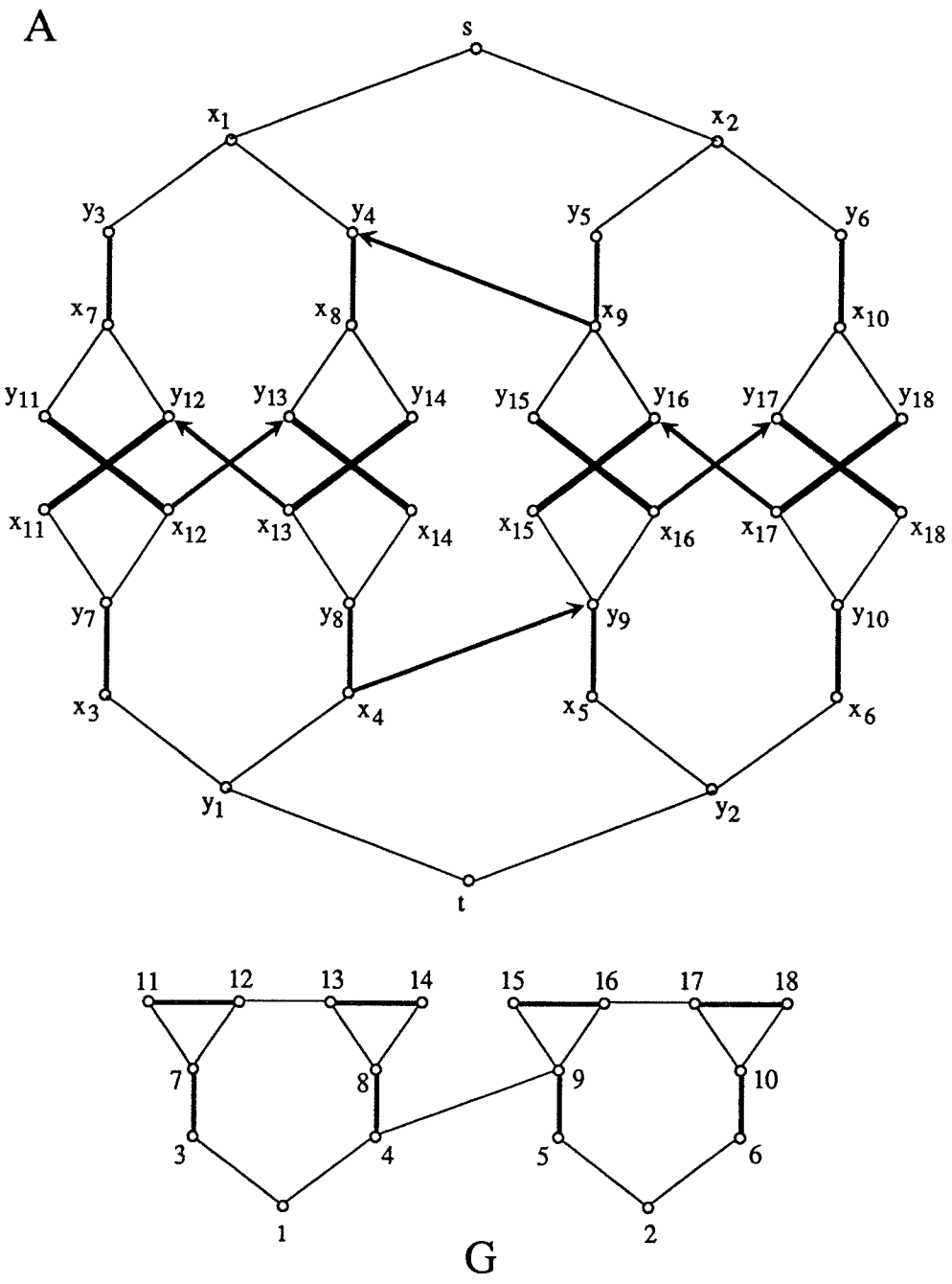


Figure 3.9
The mirror network A that is constructed from the balanced network for the graph G and the subgraph of G indicated by the thicker lines.

This illustrates some of the blossoms that the algorithm forms, but this example has another purpose. When the edge x_9y_4 is examined y_4 is not yet t-reachable, so the algorithm will determine that x_9y_4 can not be a switch edge. However, without this edge we do not find the two complementary augmenting paths in N. This problem is solved when we examine $(x_9y_4)' = x_4y_9$; y_9 is t-reachable, because x_9 was searched and is therefore s-reachable, so x_4y_9 is identified as a switch edge and is thus added to A *along with its complementary edge* x_9y_4. In general, if x_iy_j is a switch edge then y_j eventually becomes t-reachable on a path P; but then P′ is a valid sx_j-path, so x_j is eventually searched. Thus the edge $(x_iy_j)' = x_jy_i$ is examined and found to be a switch edge. Hence the above solution always occurs.

Figure 3.10 gives an example which further illustrates the structure of T and T′. In all previous examples T, and thus T′, has been a tree. However, recall that when the BNS first finds an edge to a vertex $v \notin A$, it adds v into T and v′ into T′. Consider the vertex y_9 in A in figure 3.10. y_9 is in T because it is added to A when the edge x_4y_9 is examined. But there is no path in T to y_9, so y_9 is not in the tree rooted at s; rather it is the root of a new tree in T. Hence T is actually a forest, rather than a tree. This does not in any way injure the correctness of the preceding discussion, since no part of the algorithm so far relied on the fact that T was a tree. T′ is still the mirror image of T, and the definition of a mirror network is also still valid. In light of this discovery the following convention is now used: T represents the forest, T_0 represents the tree in T rooted at s, and T_i represents an arbitrary tree in T.

The non trivial blossoms that are formed by the algorithm, in order, are: $B_1 = \{x_i,y_i \mid i = 1,3,4\}$, or simply $\{1,3,4\}$, based at base=y_1 and created upon discovery of edge=y_3x_4; $B_2 = \{6,7,8\}$, base=y_6, edge=y_7x_8; $B_3 = \{10,11,12\}$, base=y_{10}, edge=$y_{11}x_{12}$. Discovery of the switch edge $x_{12}y_7$ causes B_4, a non trivial blossom

based at t and containing every vertex in the graph, to be formed. t is therefore s-reachable, so an augmenting path has been found.

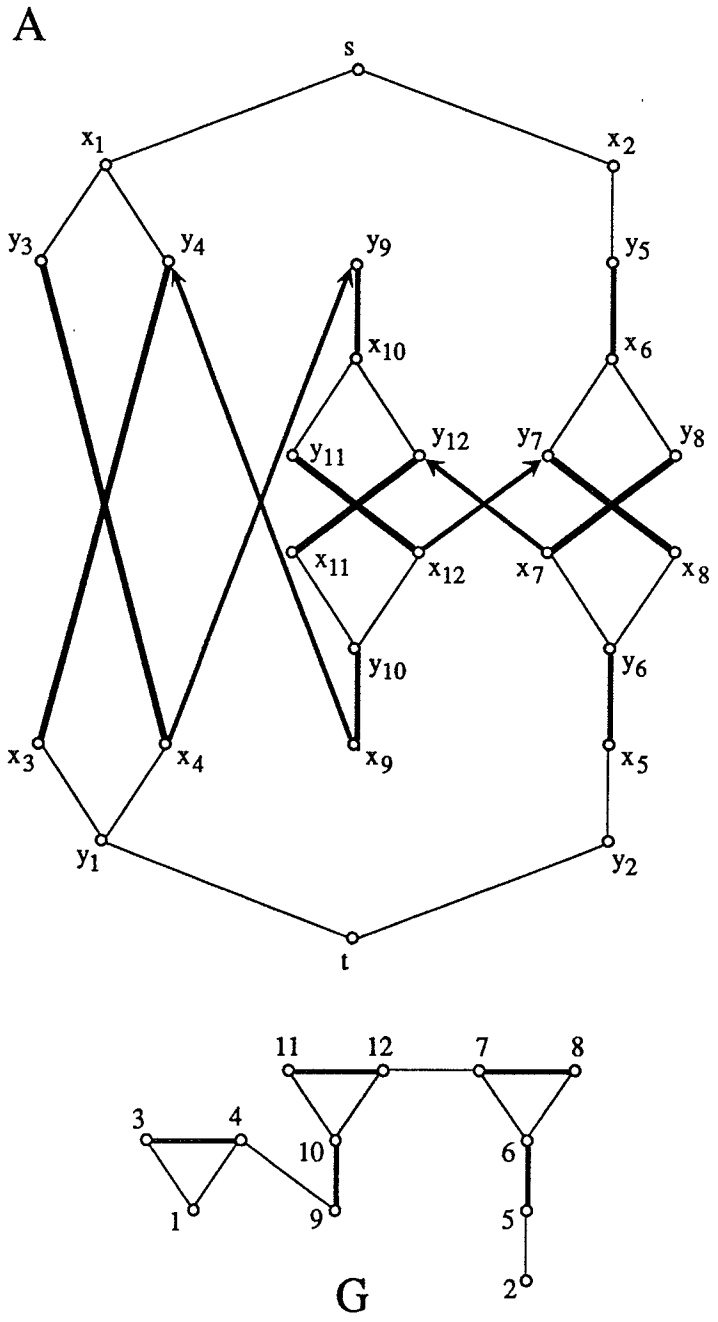


Figure 3.10
The mirror search tree A that is constructed from the balanced network for the graph G and the subgraph of G indicated by the thicker lines.

It may be helpful to follow the process by which ELB creates B_4, so let us do that now. ELB determines which blossoms should be in B_4 by following two paths of blossom bases (in the body of statement 4), one from each endpoint of the current switch edge, which in this case is $x_{12}y_7$. These two paths end when they reach a blossom C that is common to both, and then every blossom on one of these two paths is included in the new blossom, which is based at the base of C. In the example the path of blossom bases from x_{12} is y_{10},x_9,y_1,t, the path from y_7 is y_6,x_5,y_2,t, and $\{s,t\}$ is the first blossom that is common to both paths.

Notice how Depth is used by ELB to accomplish this. Let v be a vertex in T. Then PrevPt[v′] = u′ (statement 19 in the mainline) and SwitchEdge[v] = uv (statement 26 in the mainline) for some vertex u in A. By definition, Depth[u] < Depth[v] and Depth[u′] < Depth[v′]. But the base of a blossom is always in T′, and ELB goes from v′ to the base of the blossom containing u′ = PrevPt[v′] in finding the next base on the path of blossom bases. That is, ELB goes to the base of a blossom containing a vertex with a depth smaller than that of v′. It will be proved later that the depth of any vertex in a blossom is at least the depth of the base of the blossom. Hence if b′ is the base of the blossom containing u′, so that b′ is the next blossom on the path of blossom bases followed by ELB, then the depth of b′ is at most the depth of u′. The depth of the blossom bases on both paths of blossom bases followed by ELB is therefore strictly decreasing.

Since ELB is looking for the first blossom common to both paths, the next path we should advance along is the one on which the current blossom base has the highest depth. This is exactly what ELB does. In the current example, the depths of the bases on the path of blossoms followed from x_{12} are Depth[y_{10}]=4, Depth[x_9]=3, Depth[y_1]=1 and Depth[t]=0, a strictly decreasing sequence.

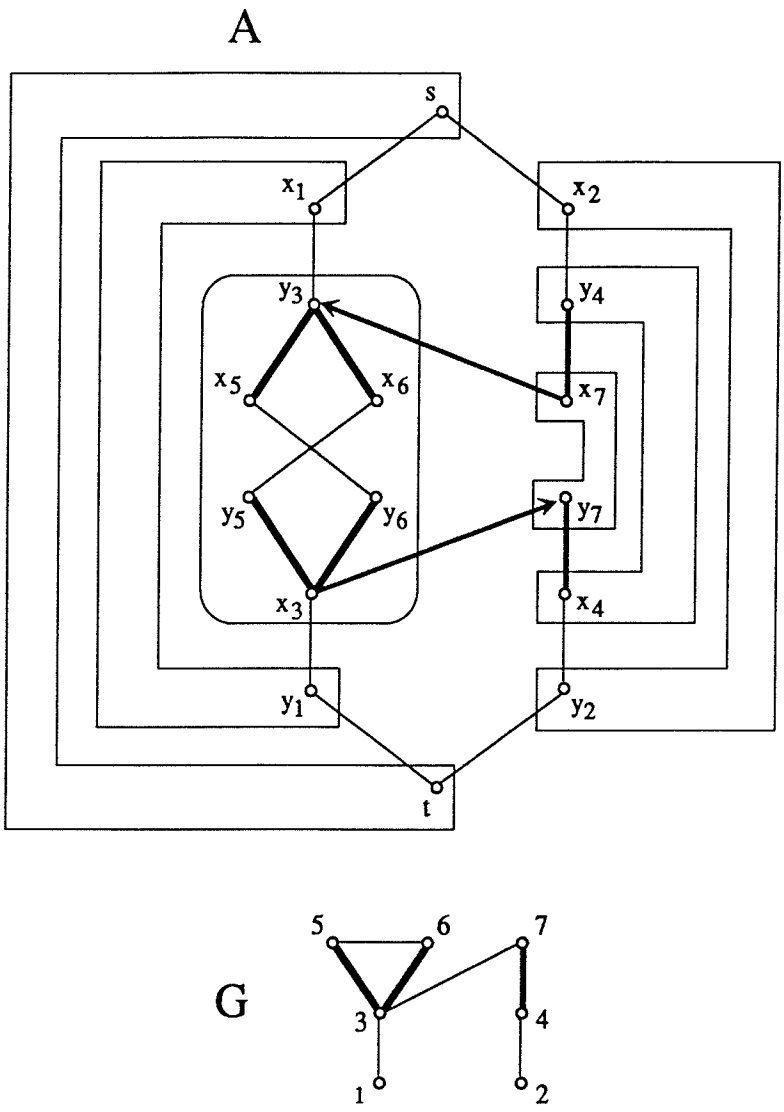


Figure 3.11
A mirror network A. Indicated in A are the blossoms that exist immediately before addition to A of the pair of complementary edges x_3y_7 and x_7y_3.

Some of the following proofs are easier to understand if one first becomes familiar with a pictorial representation of the blossoms in a mirror network. *At this point the assumption that all edges in the balanced network have capacity one is weakened; we now assume that this is so only for edges that are not incident on s or t.* Figure 3.11 portrays a graph G and the blossoms in A just before the edge x_3y_7 is discovered. All edges in the balanced network N for G have capacity one, except for the edges sx_3 and y_3t, which have capacity two. The important thing to notice in this diagram is the 'C'

shape of the blossoms. In the diagram, all the 'C' shaped blossoms are trivial, but non-trivial blossoms may also have this form. As an example, consider the pair of complementary edges x_3y_7 and x_7y_3. When they are added to A, a big blossom containing all the vertices in A is created, and this new blossom is C shaped and non trivial (the gap in the 'C' is between x_7 and y_7).

3.1.7. Some Properties of Mirror Networks

An important concept used in these properties is the differentiation between the two kinds of edges in a mirror network A; those for which one endpoint is added to A along with the edge, and those that go between two vertices that are already both in A. One can verify this by examining the mainline; statement 26 effectively adds the first kind, while statement 26 effectively adds the second, and these are the only places that edges are added to A. This first kind of edge is referred to hereafter as a **search edge**, because it corresponds to an edge that would be added in a conventional search. If v is the endpoint of a search edge uv, where v was added to A when uv was, then we say that uv is the *search edge for v*. The second kind of edge is called a **link edge**, because it links together two paths in the search network A to form a path to another vertex in A. These two sets of edges partition the edges in A.

The search edges are themselves partitioned into two types: the first is those that have both endpoints in either T or T′; the second is those that go from a vertex in T′ to one in T. To verify this, recall that search edges have some as yet unsearched vertex as an endpoint. If the algorithm is currently searching a vertex u and finds an edge to an unsearched vertex v, it places v into T and v′ into T′ (implicitly at statements 22–26 in the mainline), and this is the only place that vertices are added to A. If $u \in T$ these two edges are clearly of the first type, otherwise $u \in T'$ and they are of the second type. It is exactly the second type of edge that causes us to start building a new tree in T, and an example of this is the edge x_4y_9 in figure 3.10.

The following discussion will often refer to PrevPt[v], for a vertex v. If $v \in T$ then PrevPt[v] is u, where SwitchEdge[v] = uv. If $v \in T'$ then PrevPt[v] = u, where SwitchEdge[v'] = u'v'. It is easy to see that PrevPt[v] is simply the other endpoint of the search edge uv that was added to A at the same time as v. Thus the statements "uv is a search edge" and "u = PrevPt[v]" are equivalent. Also, if v is in T then these two statements are equivalent to the statement "SwitchEdge[v] = uv".

Let f be a balanced flow in a balanced network N, and A be the mirror network constructed by the max-balanced-flow algorithm in searching for an augmenting path in (N,f).

Property 0

If uv is a search edge and $u \in T$, then $v \in T$ also.

Proof:

As noted above, if the mainline is visiting a vertex w and it discovers an edge to an unsearched vertex z, then it always puts z into T and z' into T', and adds to A the search edges wz and z'w'. This is the only way in which search edges are added to A. Since u is the tail of such an edge, if $u \in T$ then u=w for some w (because $u \neq z'$ for any z', since $z' \in T'$), and the only possibility is that $v=z \in T$ for some vertex z. □

However, the converse is not true. If $u \in T'$ then it is certainly possible that u=z'; but w, and hence w', may be in T or T', since vertices in T' do get searched. Hence v=w' may be in T or T'.

Property 1

If e is an edge between blossoms in A, then e is a search edge.

Proof:

Whenever a link edge uv is added to the mirror network A, ExtendLongBase(uv) is executed; this puts u and v into the same blossom (lines 12 and 20 in ELB). □

Property 2

Let u′ be the base of a blossom B. Then every $z \in B$ ($z \neq u,u'$) has depth[z] > depth[u′].

Proof:

By induction on the number of calls to ExtendLongBase (or ELB).

This property is certainly true after ELB has not been called, since every blossom contains exactly two vertices, so there is no vertex such as z. To verify this examine lines 24 and 25 in the mainline, which is where v and v′, a pair of newly added complementary vertices, are first put into a blossom in A; they are both put into the same blossom, and this blossom is certainly new.

Suppose the property is true after $\leq k$ calls to ELB and look at the $k+1^{st}$ call, ExtendLongBase(uv). This call merges some set of blossoms, say $C_0,C_1,...,C_r$ with respective bases $w'_0,w'_1,...,w'_r$, into a single blossom B. This can be verified by examining the code for ELB and confirming that it only changes the LongBase of vertices that are already the base of a blossom (statements 12 and 20). ELB begins in the blossom containing u, say C_{i_1} based at w'_{i_1}, and moves into another blossom, say C_{i_2} based at w'_{i_2}, along the edge $w'_{i_1}z'_{i_2}$, where z'_{i_2}=PrevPt[w'_{i_1}] (statement 6 or 8) (see figure 3.12). Hence SwitchEdge[w_{i_1}] = $z_{i_2}w_{i_1}$ (see statements 19 and 26 in the mainline), so Depth[z'_{i_2}] < Depth[w'_{i_1}] by definition;

furthermore, C_{i_2} was created after at most k calls to ELB, so by the induction hypothesis $Depth[w'_{i_2}] \leq Depth[z'_{i_2}]$ (and we have equality only if $w'_{i_2}=z'_{i_2}$). ELB then leaves C_{i_2} in the same way, and continues along this path of blossoms until it gets to C_{i_j}, the blossom that has its base w'_{i_j} become the base of B. Every blossom on this path becomes part of the new blossom B, and it is clear that every vertex $q \neq w_{i_j}, w'_{i_j}$ in these blossoms has $Depth[q] > Depth[w'_{i_j}]$.

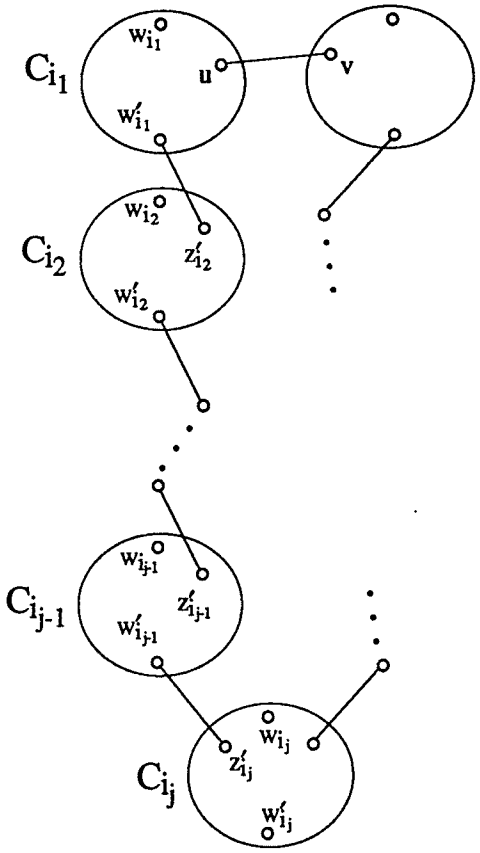


Figure 3.12
The path of blossoms followed by function ExtendLongBase.

This process happens concurrently with an identical process for a path of blossoms from the blossom containing v up to C_{i_j} (in the body of statement 4), so all the vertices q in these blossoms also have $Depth[q] > Depth[w'_{i_j}]$. But the blossoms on these two paths are exactly the blossoms $C_0, C_1, \ldots, C_r$, so this property is true after k+1 calls to ELB. □

Property 3

Let B be a blossom, v be a vertex in $B \cap T$ and u = PrevPt[v] (u may be in T or T′). v′ is the base of B if and only if $u' \notin B$ (so $u \notin B$ also).

Proof:

From statements 19 and 26 in the mainline it is clear that SwitchEdge[v] = uv; hence Depth[u] < Depth[v], so if v′ is the base of B then by property 2 $u' \notin B$. If v′ is not the base of B then B is not trivial, so there must have been a call to ELB that merged {v,v′} into a larger blossom. This call must have followed the edge v′u′ (statement 6 or 8) and put v′ and u′ (and hence v and u) into the same blossom; thus $u' \in B$, since blossoms are never split apart. □

The following notation is used to describe property 4. Given a pair of complementary vertices $w \in T$ and $w' \in T'$, w^T represents w, and $w^{T'}$ represents w′. This notation is necessary because sometimes we label a pair of complementary vertices w and w′, but do not know which of the two vertices is in T, and which is in T′.

Property 4

Let uv be a search edge, and let u and v be in different blossoms, B and C respectively. Then either $u^{T'}$ is the base of B, or $v^{T'}$ is the base of C.

Proof:

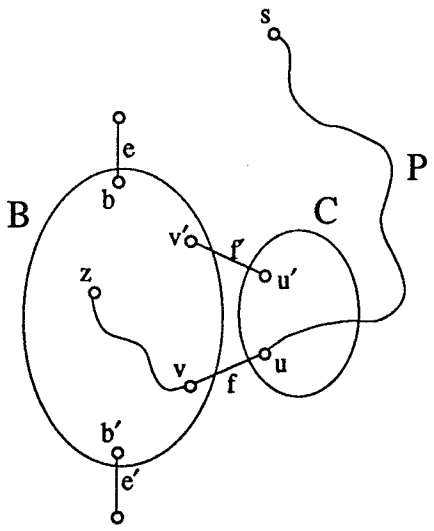
This is just a corollary of property 3. uv is a search edge so u = PrevPt[v]; thus if $v \in T$ then $v' = v^{T'}$ is the base of C. If $v \in T'$ then $v' \in T$ and, since v′u′ is also a search edge, $u' \in T$ by property 0. Thus, by property 3, $u = u^{T'}$ is the base of B. □

Property 5

Let B be a blossom with base b′, and let e be the search edge for $b \in T$ (so SwitchEdge[b] = e). Every valid path in A from s to $z \in B$ first enters B along e.

Proof:

Suppose there is a valid path P from s to z∈ B that first enters B along f=uv, where f≠e, and let B be the first blossom on P for which property 5 does not hold. By property 1 uv is a search edge. If v∈ T then f is the search edge for v and, by property 3, v′ is the base of B; hence v=b and, since the search edge for a vertex is clearly unique, e=f, a contradiction. Thus v∈ T′. If u∈ T we contradict property 0, so u∈ T′ too, and we have the following situation:



uv therefore goes up T′, since search edges in T′ are only unsaturated in this direction, and since it is a search edge v = PrevPt[u]. Thus v′ = PrevPt[u′] and, since u′∈ T and u′ and v′ are in different blossoms, we can apply property 3 to establish that u is the base of the blossom C in which it lies. v′u′ is not on P (since v′∈ B and P first enters B on the edge uv), so P enters C on some other edge. Thus C is a blossom for which property 5 does not hold, and since u∈ C precedes the first vertex in B (namely v) on P, this contradicts the fact that B was the first such blossom on P. □

It is now easy to see why the algorithm need never record an edge between two vertices in the same blossom as a switch edge. If uv is an edge between two vertices in

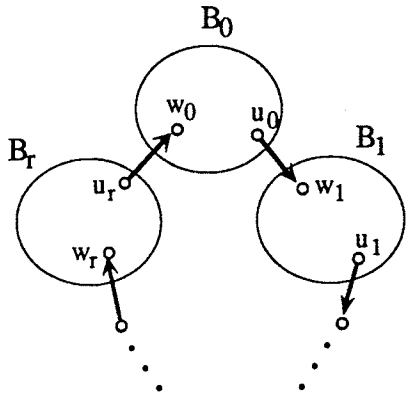
the same blossom B, then ab is on the explicit su-path in A and b′a′ is on the implicit vt-path. Since every vertex in B is already s-reachable, the only valid paths that uv can introduce into A are ones that contain a pair of complementary edges. If ab and b′a′ have a residual capacity of one, then uv can not be a switch edge. Otherwise, b′ will eventually be searched, and the relevant valid paths found. This latter case illustrates when uv could be, but is not, recorded as the switch edge for a vertex.

Property 6

There is no valid path in a mirror network A that leaves and then re-enters a blossom.

Proof:

Suppose that such a path P exists and follows the sequence of blossoms $B_0B_1B_2...B_rB_0$. By property 5 P enters each new blossom B_i at w_i, where w'_i is the base of B_i. B_i is then departed along the edge u_iw_{i+1}, where u_i is any vertex in B_i. By property 1 u_iw_{i+1} is a search edge, so since $w_{i+1} \in T$, SwitchEdge[w_{i+1}] = u_iw_{i+1}.



Thus we have:

$$\begin{aligned}
\text{Depth}[w_0] &\leq \text{Depth}[u_0] && \text{(by property 2)}\\
&< \text{Depth}[w_1] && \text{(since SwitchEdge}[w_1] = u_0w_1)\\
&\leq \text{Depth}[u_1] && \text{(by property 2)}\\
&\ldots\\
&< \text{Depth}[w_r] && \text{(since SwitchEdge}[w_r] = u_{r-1}w_r)\\
&\leq \text{Depth}[u_r] && \text{(by property 2)}\\
&< \text{Depth}[w_0] && \text{(since SwitchEdge}[w_0] = u_rw_0),
\end{aligned}$$

a contradiction. □

Property 7

There is at most one pair of complementary edges between two different blossoms B and C.

Proof:

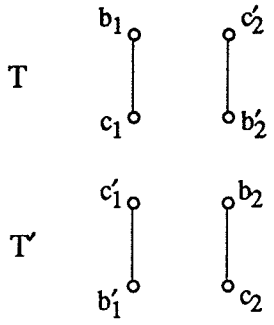
Suppose that there are two pairs of complementary edges that go between two different blossoms B and C, and let these two pairs of edges be $\{b_1c_1,c'_1b'_1\}$ and $\{b_2c_2,c'_2b'_2\}$, where $b_i \in B$ and $c_i \in C$ for all i. By property 1 all four of these edges are search edges. We have the following possibilities:

i) c'_1 and c'_2 are both in T'.

This also handles the case where c'_1 and c'_2 are both in T, since we can rename c_1 to c'_1 and c'_1 to c_1, and similarly for c_2 and c'_2. If c'_1 and c'_2 are both in T' then c_1 and c_2 are both in T, $\text{PrevPt}[c_1] = b_1 \notin C$ and $\text{PrevPt}[c_2] = b_2 \notin C$. Hence, by property 3, c'_1 and c'_2 are both the base of C. But C has a unique base, so $c'_2=c'_1$, and c'_1 has a unique PrevPt, so $b'_2=b'_1$; there is therefore really only one pair of complementary edges between B and C.

ii) $c'_1 \in T'$ and $c'_2 \in T$.

The case where $c'_2 \in T'$ and $c'_1 \in T$ is also handled here, since we can rename c_1 to c_2, and conversely. Arguing as above, c'_1 is the base of C. Also, since $c'_2 \in T$ we know by property 0 that $b'_2 \in T$ (since $c'_2 b'_2$ is a search edge), and so by property 3 that b_2 is the base of B.



We therefore have:

$Depth(b_1)$	$< Depth(c_1)$	(since $c_1 \in T$, so $SwitchEdge[c_1] = b_1c_1$)
	$< Depth(c_2)$	(by property 2)
	$= Depth(c'_2)$	(complementary vertices have same Depth)
	$< Depth(b'_2)$	(since $b'_2 \in T$, so $SwitchEdge[b'_2] = c'_2b'_2$)
	$= Depth(b_2)$	(complementary vertices have same Depth)
	$< Depth(b_1)$	(by property 2)

which is a contradiction. □

3.1.8. Proof of Correctness

Lemma 3.5

A vertex in T′ is marked s-reachable if and only if it is in a non-trivial blossom.

Proof:

The first part of the proof is easy; vertices are marked s-reachable in one of two ways: when they are added to T (statement 23 in the mainline), or when they are put into a non-trivial blossom (statements 15,23 and 28 in function ExtendLongBase).

The converse is proved by induction on the number of times ELB has been called. The lemma is true when ELB has been called 0 times, since then there are no non-trivial blossoms in the mirror network A.

Now suppose that ELB is called to create a new blossom B. ELB amalgamates some set of blossoms $C_1,C_2,...,C_r$ to create B; if C_i is a non-trivial blossom then every vertex in $C_i \cap T'$ is s-reachable by the induction hypothesis. If C_i is a trivial blossom then c′, the base of C_i, must be visited by ELB at one of statements 9,17 and 25. Thus, if it is not yet marked s-reachable then it is so marked by statement 23 or 28. Hence every vertex in $B \cap T'$ is marked s-reachable. □

Since all the vertices in T are marked s-reachable, we have the following corollary:

Corollary 3.6

Every vertex in a non-trivial blossom is marked s-reachable.

Lemma 3.7

If a vertex v in A is marked s-reachable, then there is a valid sv-path in A.

Proof:

By induction on the number of calls ExtendLongBase(uv).

Vertices are marked s-reachable in only two ways: when they are added to T (statement 23 in the mainline), or when they are put into a non-trivial blossom (statements 15,23 and 28 in function ExtendLongBase). The following lemma is useful in completing this proof:

Lemma 3.7.1

If lemma 3.7 is true at either statement 12 or statement 30 in the mainline, then it remains true until ELB is called again.

Proof:

First let the time period during which this lemma is applicable be τ; that is, τ is the time from when either statement 12 or statement 30 is executed, until ELB is called again. The only place vertices are marked s-reachable during τ is at statement 23 in the mainline. Thus A acquires another s-reachable vertex v during τ only if a vertex u is visited, where u is already marked s-reachable, and the edge e=uv is unsaturated in N. If there is a valid su-path P when this edge is discovered then there is a valid sv-path after e, e′ and v are added to A, since P existed before e and e′ were added to A, so P cannot contain e′. Hence if lemma 3.7 is true at the start of τ, then, by induction on the number of vertices marked s-reachable during τ, it is true at the end of τ. □

Lemma 3.7 is true when statement 12 in the mainline is executed; there are no edges in A, only s is marked s-reachable, and A contains a vacuous ss-path of length 0.

Now look at the k+1st call, ExtendLongBase(uv); by the induction hypothesis and lemma 3.7.1, we can say that lemma 3.7 is true at the start of this call. {u,u′} and {v,v′} are in different blossoms, say C_1 based at w'_1 and C_2 based at w'_2 respectively.

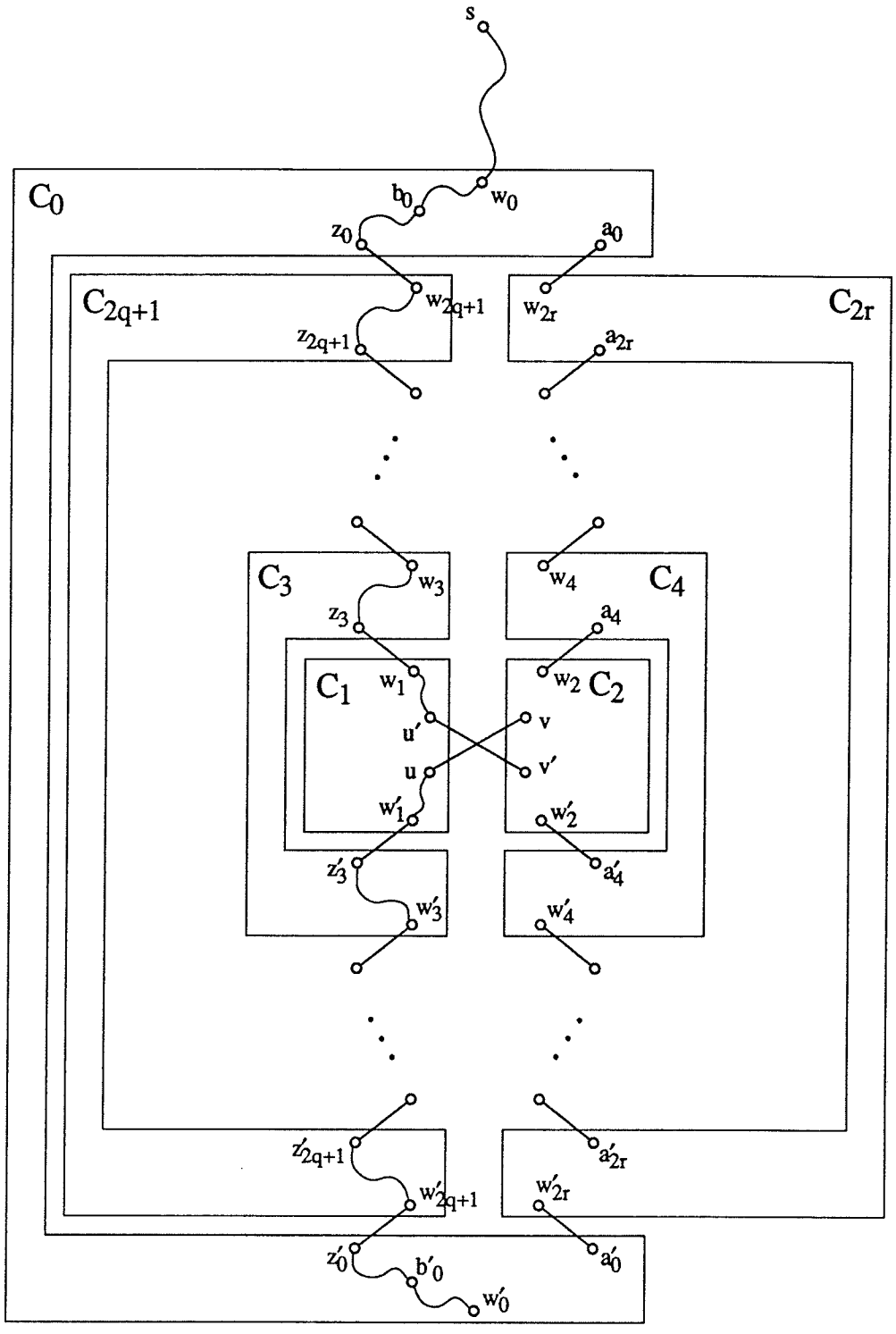


Figure 3.13
A detailed look at the path of blossoms followed by function ExtendLongBase.

As described in the proof of property 2, ELB follows a path P^* of blossoms up T', beginning at C_1. The blossom following C_1 on P^* is C_3, where $PrevPt[w'_1] = z'_3 \in C_3$, and in general the blossom following C_{2i+1} (based at w'_{2i+1}) on P^* is C_{2i+3} where $PrevPt[w'_{2i+1}] = z'_{2i+3} \in C_{2i+3}$. Thus $P^* = C_1C_3...C_{2q+1}C_0$, where $C_0 = C_{2q+3}$ (see figure 3.13). ELB also follows a similar path of blossoms up T' beginning at C_2, and this path is $Q^* = C_2C_4...C_{2r}C_0$, where $C_0 = C_{2r+2}$. The last blossom on both P^* and Q^* is C_0, the first blossom that is common to both. By property 5 every valid sq-path P, where $q \in C_1$, uses the edge z_3w_1, and by property 6 the portion of P from w_1 to q stays inside C_1. Both of these statements are true in general; hence a valid path P_u from s to u must eventually enter C_0 and thereafter follow blossoms $C_{2q+1}, C_{2q-1},...,C_1$, and by the induction hypothesis such a path does exist. Also, v is t-reachable so v' is s-reachable; hence a similar path P_v from s to v' also exists, and P_v eventually enters C_0 and thereafter follows blossoms $C_{2r}, C_{2r-2},...,C_2$.

Now let $z_0 = PrevPt[w_{2q+1}]$, $a_0 = PrevPt[w_{2r}]$, and b_0 be the last vertex on the sz_0-path that is also common to the sa_0-path. Also, let Q_u be the sub-path of P_u from b_0 to u, and Q_v be the subpath of P_v from b_0 to v'. Q_u and Q_v are vertex disjoint, except for b_0 (or C_0 would not be the first blossom on P^* and Q^* that is common to both), and are therefore edge disjoint; thus Q'_u and Q'_v are also edge disjoint and so $Q = Q_uuvQ'_v$ is a valid path, since if $e \in Q_u$ and $e' \in Q'_v$ then, since $e' \in Q'_u$ by definition, Q'_u and Q'_v are not edge disjoint, a contradiction.

If R is the portion of P_u from s to b_0, then P=RQ is a valid path from s to w'_0, as is RQ'. This is so because if it were false then, since Q and the su-path are both valid, there would have to be an edge e on R such that e' is on either Q'_u or Q'_v. But every edge e on R is on the sb_0-path, and is therefore on the su-path P_u and the sv-path P_v, and precedes b_0 on both of these paths; thus e' follows b'_0 on P'_u and P'_v, and is therefore not on Q'_u or Q'_v.

But only vertices that are on Q or Q′ are marked s-reachable by ELB, so if there is a path P in A from s to any such vertex q, then P is certainly valid. There is a valid sq-path in A if there is an su-path and a vq-path in A, where SwitchEdge[q] = uv. Every vertex such as q has its switch edge recorded as uv at one of statements 14, 22 and 27 in ELB, and there is an su-path by the induction hypothesis. v′ is s-reachable because v is t-reachable, so by the induction hypothesis there is an explicit sv′-path V, and hence an implicit vt-path. Property 5 can be used in a simple inductive argument to show that q′ is on V. But q′ is not t-reachable, so by lemma 3.2 there is an explicit q′v′-path, and hence an implicit vq-path. Hence there is an sq-path in A, and this path is valid. □

Lemma 3.8

If a valid sv-path appears in A, then v is eventually marked s-reachable.

Proof:

Suppose there is a valid path P in A to a vertex that does not get marked s-reachable, and let v′ be the first such vertex on P. Every vertex in T gets marked s-reachable, so by corollary 3.6 v′ must be the base of a trivial blossom {v,v′}. Every edge incident on v′ is therefore a search edge (if a link edge was incident on v′, then v would be in a non-trival blossom); hence every edge that is unsaturated into v′ is a search edge, including the edge w′v′ on P, where w′=pred[v′]. w′ is marked s-reachable so it is searched, and statement 27 is executed when the edge w′v′ is discovered. Hence the conditional statement 28 is executed, since statement 27 clearly evaluates to true, and ELB is called and thus marks v′ s-reachable unless statement 28 evaluates to false; thus ResCap[vw]=1 and v=pred[w]. Also, if w′∈ T then we violate property 0, so w′∈ T′; thus w∈ T and so by property 3 w′ is a blossom base. But then by property 5 every sw′-path in A contains the edge vw; since this includes P, P is not a valid path to v′, a contradiction.

Corollary 3.9

There is a valid path in A from s to v if and only if v is eventually marked s-reachable.

Theorem 3.10

A valid sv-path eventually appears in A if and only if there is a valid path in N from s to v.

Proof:

Suppose there is a valid sv-path in A. Every edge in A is an edge in N, so this is also a valid path in N.

Now suppose there is a valid sv-path P in N, and further suppose that no such path appears in A. Let u' be the first vertex on P that is not marked s-reachable, and let w' be the vertex preceding u' on P (so all vertices preceding u' on P are searched and therefore in A). As noted in the proof of Lemma 3.8, w' is the base of a blossom B, ResCap[uw]=1 and u=pred[w]. Now consider the first edge zb on P such that $b \in B$ in A; neither z nor b comes after w' on P, since $w' \in B$, and zb is not an edge of A, or by property 5 it must be the edge uw and so P is not a valid su'-path, a contradiction. z is s-reachable so it was searched by the algorithm and the edge zb was examined but not added to A; hence b was not t-reachable at that time. B is a non trivial blossom, by lemma 3.5 and the fact that it contains a vertex in T' that is marked s-reachable, so every vertex in B is searched; this includes b', so the edge $b'z'$ was examined by the algorithm and not added to A; hence z' was not t-reachable at that time. But either zb or $b'z'$ was examined first, so suppose it was zb; at this time z' was marked t-reachable, a contradiction. □

Theorem 3.11

The balanced network max-flow algorithm works.

Proof:

The flow in a balanced network N is maximum if and only if there is a valid augmenting path in N. This algorithm finds an augmenting path in a balanced network N if and only if t is marked s-reachable (see statement 30 in the mainline). But by corollary 3.9 and theorem 3.10 this happens if and only if there is an augmenting path in N. □

3.2. Complexity Analysis

In the following analysis, a **step** is considered to be anything that takes a constant amount of time, regardless of the size of the constant. However, the sizes of the constants are actually quite small; that is, this algorithm is certainly of practical use. Let ε and n be the respective number of edges and vertices in the balanced network N. Let us first determine the complexity of finding and augmenting on a single augmenting path, which involves one iteration of the body of statement 2 in the mainline. The complexity of procedure calls that are made during this single iteration of statement 2 will be analysed by calculating their complexity *over all calls*, rather than calculating the complexity of a single call. This enables us to add the complexity of each of these procedures to the complexity of the mainline, instead of multiplying the number of calls by the complexity of a single call.

In the mainline, the maximum number of steps executed by one of statements 3–12 is n (for array initialisation), so their complexity is O(n). Statement 13 can be executed at most n times, once for each vertex in N. It contains exactly one loop, statement 16. In the worst possible case every edge in N may be unsaturated in both directions; thus statement 16 may be executed at most 2ε times. But this is *over all iterations* of statement 13, not per iteration. All other statements in the body of statement 13 require constant time, so the complexity of statement 13 is $O(\varepsilon + n)$.

Procedure PullFlow is called exactly once from the mainline, but will be called recursively during the execution of this call. Every call to PullFlow, recursive or otherwise, requires a constant amount of time, so an upper bound on the number of calls will give us the complexity of this procedure. Every call to PullFlow that executes the recursive case augments the flow on a pair of complementary edges; since the length of an augmenting path in N cannot exceed n, there are no more that n such calls. But each call to PullFlow may make at most two recursive calls, so we get a trivial upper bound of 2n on the number of calls to PullFlow that execute the easy case. Thus there are no more than 3n calls to PullFlow altogether, so PullFlow has complexity O(n).

Now consider the number of steps executed by procedure ExtendLongBase (ELB) *over all calls* during a single iteration of statement 2 in the mainline. The mirror network A that the algorithm builds can have at most n/2 blossoms, and each call to ELB reduces the number of blossoms in A by at least one. Hence there are no more than n/2 calls to ELB. The first loop in ELB is at statement 4. In the body of this statement, an assignment at statement 6 or 8 is a movement from one blossom into another. This is how ELB traverses a path of blossoms. For each assignment made, another blossom is merged into the new blossom, so the number of blossoms in A is reduced by 1. Thus there can be no more that n/2 such assignments *over all calls* to ELB. ELB has only two other loops, at statements 11 and 19, and the body of each of these statements requires constant time. But it is clear that over all calls to ELB, statement 11 will be executed exactly once for each assignment made in statement 6, and similarly for statements 19 and 8. Thus, *over all calls* to ELB, these two loops execute no more than n/2 steps between them. All other statements in ELB require constant time, and therefore also execute no more than n/2 steps between them over all calls to ELB, since ELB is called no more than n/2 times. Thus, *over all calls*, ELB executes n/2 steps and therefore has complexity O(n).

Now all that remains is to analyse the complexity of FindBase. The calls to FindBase at statements 2,3,10 and 18 in ELB are included in the pseudo code purely for the purpose of clarity, and will therefore not be considered; they are clearly not necessary because the values FindBase(u) and FindBase(v) are known in the mainline and can therefore be passed to ELB as parameters. From the analysis of ELB, we can see that FindBase will be called in ELB no more than n/2 times at statements 6 and 8 combined, and no more than n/2 times at statements 17 and 25 combined. This gives at most n calls in ELB. In the mainline FindBase is called at most n times at statement 15 (once for each vertex) and at most 2ε times at statement 17 (at most once for each edge incident on a vertex). This gives a total of at most $2\varepsilon + 2n$ non-recursive calls to FindBase; that is, calls to FindBase from outside FindBase. Hence there are no more than $2n + 2\varepsilon$ calls to FindBase that execute the easy case, since every such call must have a corresponding non-recursive call.

We can now place an upper bound on the total number of calls to FindBase by finding bounds for the number of such calls that do and do not alter the LongBase array. Let us first examine the latter. This clearly includes all calls that execute the easy case. If an invocation of FindBase executes the recursive case and does not change a value in the LongBase array, then from statements 5 and 6, FindBase(LongBase[v]) = LongBase[v]. Hence LongBase[v] is the base of the blossom in which it lies, so the recursive call at statement 5 executes the easy case and the recursion ends. There are therefore no more than $2n + 2\varepsilon$ such calls, for a total of no more than $4n + 4\varepsilon$ calls to FindBase that do not alter the LongBase array.

Since blossoms are always merged, and never split apart, each element of the LongBase array may trivially change no more than n times. Since each call to FindBase changes at most one such element, there are therefore no more than n^2 calls to FindBase that change an element in the LongBase array. Thus the complexity of FindBase over all calls is $O(n^2 + 4\varepsilon + 4n) = O(n^2 + \varepsilon)$.

Summarising the above results, we see that the complexity of finding and augmenting on a single augmenting path is $O(\varepsilon + n) + O(n) + O(n) + O(n^2 + \varepsilon) = O(n^2 + \varepsilon)$. However, the complexity of FindBase can very easily be improved to $O(n \cdot \log n)$. To do this, when blossoms are merged always make the base of the largest blossom (i.e. the one with the most vertices) the base of the new blossom. Then a given vertex v is always merged into a blossom that is at least twice as big, so LongBase[v] can change no more than log(n/2) times. Hence the total work done by FindBase is $O(\varepsilon + n \cdot \log n)$, and the complexity of finding a single augmenting path is then improved to $O(\varepsilon + n \cdot \log n)$.

This can be implemented with only trivial changes to the algorithm given herein. LongBase can no longer be used to hold the representative of the blossom containing a vertex, since that representative will now usually not be the base of the blossom. Instead, an array BlossomRep will be needed for this purpose, and having determined that u is the blossom representative, LongBase[u] will then give the base of the blossom. Also, we will need to store the size of each blossom. This can be done with an extra array, but in a clever implementation this is not necessary. One merely stores a negative number –x in BlossomRep[u] for every vertex u that is a blossom representative, where x is the number of vertices in the blossom. FindBase then knows it has an easy case when it finds a negative number in the BlossomRep array.

This last improvement was mentioned because of the simplicity of the changes required to the given algorithm in order to implement it. However, it is not the best we can do. The use made of the merge–find data structure in this algorithm is a special case in which the "union tree" [8] is known in advance. That is, we only merge blossoms that are already in the tree. This differs from the general merge find situation, where the performance of a merge operation defines some sub-tree of what will later become the "union tree", but the placement of this sub-tree in the final union tree is not yet known. [8] describes a method of implementing this special case in $O(\varepsilon)$ time and O(n) space by

pre-computing some of the FindBase operations, and augmenting the normal FindBase operation with some table lookups, which are performed in constant time.

This is really only of theoretical importance, because the extra programming required is definitely non-trivial, and the first improved algorithm given above will perform the merge–find operations in time that is very close to $O(\varepsilon)$ anyway. However, it does allow us to say that *the complexity of finding a single augmenting path in the algorithm given in this thesis is* $O(\varepsilon + n)$, even though that will not be the complexity of an implementation from the given pseudo code.

Since finding one augmenting path P guarantees that the value of the balanced flow will be increased by at least two units (we augment by at least one unit on both P and P′), we may need to find such a path at most m/2 times, where m is the value of a maximum balanced flow in N. Thus the complexity of the maximum balanced flow algorithm is $O(m) \bullet O(\varepsilon + n) = O(m\varepsilon + mn)$.

Of particular interest is the relationship between the number of edges and vertices in a graph G and the corresponding network N. If we now let n and ε be the respective number of edges and vertices *in G* rather than N, then N has $2n + 2\varepsilon$ edges and $2n + 2$ vertices. In terms of the number of vertices and edges in G, the complexity of the maximum balanced flow algorithm is therefore $O(m\bullet(2n + 2\varepsilon) + m\bullet(2n + 2)) = O(m\varepsilon + mn)$.

One final note should be made here about the complexity of the BNS. The BNS does not make a BFS of the balanced network, since a vertex v may be visited before a vertex u, even though the sv-path is longer than the su-path. The advantage of a BFS is that it gives us shortest paths, but the first switch edge we find for a vertex v may not be the one that gives us the shortest path to v, even if we do perform a BFS of N. Adding to the algorithm another level of complexity so that it executes a BFS is therefore not worthwhile, since something more would be needed to ensure that

shortest paths are found, and thus improve the complexity. The important thing is that all s-reachable vertices get searched, and this does happen.

3.3. Implementation

Appendix A contains a program that uses an implementation of the maximum balanced flow algorithm to find an f–factor in a simple graph. This program is coded in Think's Lightspeed Pascal [9] for the Macintosh computer. The edges in the balanced network that are not incident on the source or the target therefore all have capacity one, but this implementation can easily be extended to multiple graphs, where this restriction does not apply (see section 4.3.3.5).

In the given implementation an augmenting path P always has a residual capacity of one, and procedure PullFlow always augments the flow in the balanced network by exactly one unit on both P and P′. In general an augmenting path may have a residual capacity that is greater than one. It is extremely difficult to keep track, for each vertex v, of the residual capacity of the sv-path in A, because this path often contains *subpaths* of other su-paths. One simple answer to this problem is for PullFlow to first augment the flow on both P and P′ by one unit, and at the same time calculate the residual capacity of P. Then PullFlow can augment on P a second time, by the appropriate amount, so that P is no longer unsaturated. (Be careful if P contains a pair of complementary edges with a residual capacity that is greater than one, since then the flow on P can only be augmented by $\lfloor \text{rescap}(P)/2 \rfloor$, and similarly for P′.) This will basically double the number of steps required to augment the flow on an augmenting path, and will therefore not adversely affect the complexity of the algorithm.

The given implementation contains an optimization that should be clarified. A pair of complementary vertices always have the same LongBase and the same Depth, so it is only necessary to store this value once. The type "GraphVertexType" is used to denote the range of values $0 \leq i \leq n$, where N contains the pairs of complementary vertices

$\{x_i,y_i \mid 0 \leq i \leq n\}$. Both LongBase and Depth are indexed by an element of this type; thus to look up Depth[x_i] or Depth[y_i] one actually looks up Depth[i], and similarly for LongBase[x_i].

This is the only significant optimization that is made though, since the code was designed with clarity and readability, rather than efficiency, as the main priority.

4. Applications of Balanced Networks

4.1. An Alternate Proof of Tutte's Theorem

In section 2.1 it was noted that given any graph G there is a unique balanced network N that corresponds to G, and that a balanced flow in N corresponds to a subgraph H of G. By the conservation condition, it is clear that every vertex i in H has degree flow(sx_i) = flow(y_it). Suppose we would like to know if there is a perfect matching in a graph G. We can give every edge in N capacity 1, and if there is a balanced flow f in N that saturates all of the edges incident on s then it defines a perfect matching in G. In this case val(f) = n, the number of vertices in G. Also, any perfect matching in G defines a balanced flow f in N such that val(f) = n. Thus, by the Max-Balanced-Flow–Min-Balanced-Cut theorem, there is a perfect matching in G if and only if balcap(K) ≥ n, for every balanced edge cut K in N.

Let |A| denote the number of pairs of complementary vertices in A, so |A| = |A*|, and [A,B] denote the set of edges directed from A to B in N. |A,B| is used to denote |[A,B]|. This notation is also extended to the sets B,C and D in the obvious way. Thus balcap(K) = 2|A| + |B,B| + |B,C| + |B,D| + |C| + |D| – k and n = |A| + |B| + |C| + |D|, so balcap(K) ≥ n–m implies that |A| + |B,B| + |B,C| + |B,D| ≥ |B| + k. This proves:

Theorem 4.1

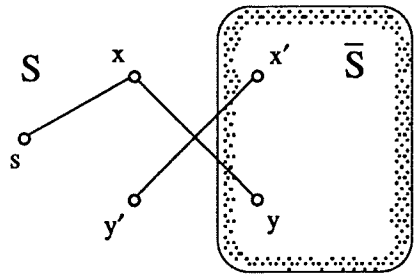
G has a perfect matching if and only if |A| + |B,B| + |B,C| + |B,D| ≥ |B| + k, for every balanced edge cut K in N.

Now suppose that G does not have a perfect matching. Let f be a maximum balanced flow in N, and S be the set of all vertices that are s-reachable in (N,f). The edge cut K = [S,$\bar{S}$] can be used to divide the pairs of complementary vertices in N into the 4 sets A,B,C and D, as described in section 2.2, and by corollary 2.20 K is a balanced edge cut in N.

Lemma 4.2

[B,B] = [B,D] = Ø.

Proof:

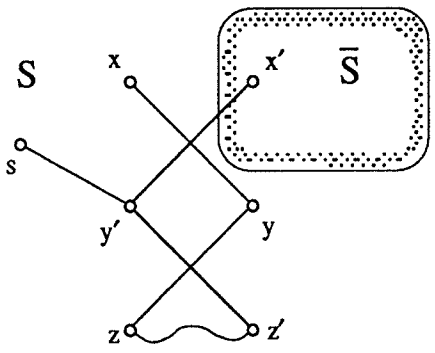


Suppose xy is an edge in either [B,B] or [B,D]. xy cannot be unsaturated or $y \in \overline{S}$ would be s-reachable, so suppose xy is saturated. Then by the conservation condition sx is also saturated, and since cap(sx) = 1, xy is the only edge out of x that carries flow. Thus there is there is no valid sx-path in N, a contradiction. □

Lemma 4.3

[B,C] = Ø.

Proof:



Suppose xy is such an edge. If xy is saturated then so too is y′x′ (by complementarity) and thus sy′ (by the conservation condition). But then there is no

valid sy′-path in N, a contradiction. Thus xy is unsaturated, so every valid sy′-path must use the edge xy, or $x' \in \overline{S}$ would be s-reachable. But every such valid path P must begin with yz, the one saturated edge in N directed into y, and end with the edge z′y′. Thus P is invalid, a contradiction. □

Let an **odd component** be a connected component with an odd number of vertices, and **odd(G)** denote the number of odd components in a graph G. If $Z \subseteq V(N)$ and consists entirely of pairs of complementary vertices, define in G the set $Z^* = \{z \in V(G) \mid x_z, y_z \in Z\}$. This defines the sets A^*, B^*, C^* and D^* in G.

Tutte's Theorem

G has a perfect matching if and only if $odd(G-U) \leq |U|$ for all sets of vertices U in G.

Proof:

First suppose G does not have a perfect matching, and consider (N,f), as defined previously. By lemmas 4.2 and 4.3 the vertices in B^* will be isolated vertices in $G-A^*$, and the C_i^* will be odd components, since $cap(K_i) = X_{IB} + X_{II} = X_{II} = |C_i| = |C_i^*|$ is odd. Thus $G-A^*$ has at least $|B^*| + k$ odd components; that is, $|B^*| + k \leq odd(G-A^*)$. But G has no perfect matching, so by theorem 4.1 $|A| < |B| + k$. Thus $|A^*| = |A| < |B| + k = |B^*| + k \leq odd(G-A^*)$.

If there is a set of vertices A^* in G such that $|A^*| < odd(G-A^*)$, then: let B^* be all isolated vertices in $G-A^*$; C^* be all vertices that are in an odd component consisting of more that one vertex in $G-A^*$, where there are k such components; and D^* be the remaining vertices of G. Now construct a balanced network N from G. Form the edge cut K in N defined by the sets A,B,C and D, which are related to A^*, B^*, C^* and D^* respectively in the manner described previously. Then $|B| + k = |B^*| + k = odd(G-A^*) > |A^*| = |A| = |A| + |B,B| + |B,C| + |B,D|$, since there are

no edges in [B,B], [B,C] and [B,D], or the vertices in B^* would not be isolated in $G-A^*$. Hence, by theorem 4.1, G does not have a perfect matching. □

The maximum balanced flow algorithm will clearly find a perfect matching in G if one exists. However, if a perfect matching does not exist, then the algorithm will still find the sets A,B,C and D. This is extremely useful, because it means that the algorithm will actually *find* a set A^* of vertices that, by Tutte's Theorem, proves that G does not contain a perfect matching.

It is easy to generalise the above results. Let M be a maximum matching in G, and suppose that r vertices in G are unsaturated with respect to M. Then we say that r is the **deficiency** of G, **def(G)**. If a graph has a perfect matching then def(G) = 0. As stated earlier, it is clear that a balanced flow in N with value n–r, $r \geq 0$, defines a matching in G that saturates n–r vertices, and conversely. Hence def(G) = n – val(f), where f is a maximum balanced flow in N. We therefore have:

Theorem 4.4

G has a matching that saturates n–r vertices if and only if $|A| + |B,B| + |B,C| + |B,D| \geq |B| + k - r$, for every balanced edge cut K in N.

This gives us theorem 4.5, the proof of which is similar to that of Tutte's theorem, but with an extra term included in the calculations.

Theorem 4.5

Let G be a graph with n vertices. G has a matching that saturates n–r vertices if and only if $|U| \geq odd(G-U) - r$ for all sets of vertices U in G.

Proof:

First suppose G does not have a matching which saturates n–r vertices, and consider (N,f), as defined previously. As in the proof of Tutte's Theorem, $k \leq$

odd(G–A^*). But G has no matching that saturates n–r vertices, so by theorem 4.4 $|A| < |B| + k - r$. Thus $|A^*| = |A| < |B| + k - r = |B^*| + k - r \leq \text{odd}(G-A^*) - r$.

If there is a set of vertices A^* in G such that $|A^*| < \text{odd}(G-A^*) - r$, then form the edge cut K in N defined by the sets A,B,C and D, as described in the proof of Tutte's Theorem. Then $|B| + k - r = |B^*| + k - r = \text{odd}(G-A^*) - r > |A^*| = |A| = |A| + |B,B| + |B,C| + |B,D|$. Hence, by theorem 4.4, G does not have a matching that saturates n–r vertices. □

If there is no matching in G that saturates n–r vertices, then as before the maximum balanced flow algorithm will find a set A^* of vertices that, by theorem 4.5, proves that G does not contain such a matching.

4.2. Conditions for the Existence of a Simple Graph

Let $\Delta = (d_1, d_2, \ldots, d_n)$, where $d_i \geq d_{i+1}$ for $0 \leq i \leq n-1$, and $0 \leq d_i \leq n-1$ for $0 \leq i \leq n-1$. Δ is called a **degree sequence**, because the degrees of the vertices in a graph can always be arranged in such an ordered n-tuple. Thus every graph has a degree sequence, but the converse does not necessarily hold. If there is a simple graph that has degree sequence Δ, then we say that Δ is **graphic**.

Suppose we are given a degree sequence Δ, and would like to determine whether or not Δ is graphic. The following well known theorem of Erdös and Gallai [4] gives us one way to make this determination:

Theorem 4.6

A degree sequence $\Delta = (d_1, d_2, \ldots, d_n)$, where $d_i \geq d_{i+1}$ for $0 \leq i \leq n-1$, and $0 \leq d_i \leq n-1$ for $0 \leq i \leq n-1$, is graphic if and only if:

1. $\sum_{i=1}^{n} d_i$ is even, *and*
2. $\sum_{i=1}^{j} d_i \leq j(j-1) + \sum_{i=j+1}^{n} \min\{j, d_i\}$, for $1 \leq j \leq n$

It is easy to see that this is necessary. For a proof that it is sufficient see [4].

A graph G with degree sequence Δ is a subgraph of K_n, the simple graph on n vertices in which every possible edge occurs. We can therefore observe that if we construct a balanced network N from K_n, and give edge sx_i in N capacity d_i, $1 \leq i \leq n$, then Δ is graphic if and only if a maximum balanced flow in N saturates all the edges incident on s.

By the Max-Balanced-Flow–Min-Balanced-Cut theorem, a maximum balanced flow in N saturates all the edges incident on s if and only if the value of a minimum balanced edge cut in $N = \sum_{i=1}^{n} cap(sx_i)$. Now suppose we have found a maximum flow in N, and let S be the set of all vertices in N that are reachable on a valid path. Then as noted in section 2.4.2 $K^* = [S,\bar{S}]$ is a minimum balanced edge cut in N, and has the general structure described in section 2.2. By lemma 2.11 there are no edges between C and D; but N was constructed from K_n, so if there is a vertex in C and a vertex in D then there is an edge between them; hence either $C=\emptyset$ or $D=\emptyset$. In either case we get $balcap(K^*) = 2 \cdot \sum_{x_i \in A} cap(sx_i) + b(b-1) + b(n-b-a) + \sum_{\substack{x_i \notin A \\ x_i \notin B}} cap(sx_i) - k$, where a and b denote the number of pairs of complementary vertices in A and B respectively. If a simple graph with the required degree sequence does exist, then we must have $balcap(K) \geq \sum_{i=1}^{n} cap(sx_i) = \sum_{i=1}^{n} d_i$ for all balanced edge cuts K in N. We may assume this latter sum is even, since adding an edge to a graph H adds 2 to $\sum_{v \in H} deg(v) = \sum_{i=1}^{n} d_i$; thus if the aforementioned sum is odd then the degree sequence is certainly not graphic.

Since every vertex x_i is joined to every vertex y_j, $i \neq j$, in N, the number of connected components in C is exactly 1, if $C \neq \emptyset$. Thus we have exactly two sets of edges in K, namely K_r and K_i, and so $cap(K) = cap(K_r) + cap(K_i)$. But edges appear in K_r only in complementary pairs, so $cap(K_r)$ is even. This shows that if K is a balanced

edge cut then k=1 and cap(K) is odd, because cap(K_i) is. So, if K is a balanced edge cut and $cap(K) \geq \sum_{i=1}^{n} cap(sx_i)$, then $cap(K) > \sum_{i=1}^{n} cap(sx_i)$, and balcap(K) = cap(K) – 1 $\geq \sum_{i=1}^{n} cap(sx_i)$. This shows us that it suffices to check the condition for cap(K) instead of balcap(K). That is, for all edge cuts K in N we require

$$cap(K) = 2 \cdot \sum_{x_i \in A} cap(sx_i) + b(b-1) + b(n-b-a) + \sum_{\substack{x_i \notin A \\ x_i \notin B}} cap(sx_i) \geq \sum_{i=1}^{n} cap(sx_i),$$

which, after simplification, becomes:

$$\sum_{x_i \in A} cap(sx_i) + b(n-1-a) \geq \sum_{x_i \in B} cap(sx_i)$$

To find out if this is ever false, we must examine what happens when the L.H.S. is made as small as possible. This occurs when the b largest d_i are in B and the a smallest d_i are in A, where the phrase "d_i is in A" means that x_i is in A. For a given value of b there is a particular value of a that makes this as small as possible; we put the a vertices x_i for which $d_i < b$ into A.

Thus if $b(n-1-a) + \sum_{d_i<b} d_i \geq \sum_{i=1}^{b} d_i$, where a is the number of terms in the summation on the L.H.S., for all possible values of b, then a maximum balanced flow in N saturates all the edges incident on s. If this condition is violated for any value of b, then no such flow exists. But D is graphic if and only there is such a flow, which proves:

Theorem 4.7

A degree sequence $\Delta = (d_1, d_2, \ldots, d_n)$, where $d_i \geq d_{i+1}$ for $0 \leq i \leq n-1$, and $0 \leq d_i \leq n-1$ for $0 \leq i \leq n-1$, is graphic if and only if $b(n-1-a) + \sum_{d_i<b} d_i \geq \sum_{i=1}^{b} d_i$, where a is the number of terms in the summation on the L.H.S., for $0 \leq b \leq n$.

In actual fact the conditions given in theorem 4.7 are the same as those given in theorem 4.6, so we can once again use balanced network theory to give us an alternate proof of a known result. First recall that a and b are the number of pairs of complementary vertices in the sets A and B, so that $n = a + b + c$, $c \geq 0$. Now we can say that Δ is graphic if and only if

$$\sum_{i=1}^{b} d_i \leq b(n-1-a) + \sum_{d_i<b} d_i \qquad \text{(a terms in the summation)}$$

$$= b((a+b+c)-1-a) + \sum_{d_i<b} d_i \qquad \text{(since } n = a+b+c\text{)}$$

$$= b(b-1) + bc + \sum_{d_i<b} d_i$$

$$= b(b-1) + \sum_{i=b+1}^{n} \min\{b, d_i\} \qquad \text{(because } c \geq 0\text{)}$$

Hence, by theorem 4.7 and an earlier note about the parity of a degree sequence, degree sequence Δ exists if and only if the conditions stated in theorem 4.6 hold; this proves theorem 4.6 in both directions.

Now suppose we ask whether or not there is a multigraph G with degree sequence Δ, where m is the maximum allowable edge multiplicity in G. Using the previous method, and using the fact that the capacity of an edge between X and Y is now m instead of 1, it is easy to work out that G exists if and only if:

$$\sum_{x_i \in A} \text{cap}(sx_i) + mb(n-1-a) \geq \sum_{x_i \in B} \text{cap}(sx_i), \text{ for all balanced edge cuts K.}$$

Proceeding as before, this is true if and only if:

$$mb(n-1-a) + \sum_{d_i<mb} d_i \geq \sum_{i=1}^{b} d_i, \qquad 0 \leq b \leq n,$$

where a is the number of terms in the summation on the L.H.S. Thus:

$$\sum_{i=1}^{b} d_i \leq mb(n-1-a) + \sum_{d_i<mb} d_i$$

$$= mb((a+b+c)-1-a) + \sum_{d_i<mb} d_i \qquad (n = a+b+c,\ c \geq 0)$$

$$= mb(b-1) + mbc + \sum_{d_i<mb} d_i$$

$$= mb(b-1) + \sum_{i=b+1}^{n} \min\{mb, d_i\} \qquad \text{(because } c \geq 0\text{)}$$

We have therefore proved the more general Erdös Gallai conditions for a multigraph, which are summarised in the following theorem:

Theorem 4.8

There is a multigraph G with degree sequence $\Delta = (d_1, d_2,...,d_n)$, where $d_i \geq d_{i+1}$ for $0 \leq i \leq n-1$, $0 \leq d_i \leq n-1$ for $0 \leq i \leq n-1$, and m is the maximum allowable edge multiplicity in G, if and only if:

$$\sum_{i=1}^{b} d_i \leq mb(b-1) + \sum_{i=b+1}^{n} \min\{mb, d_i\}, \quad 0 \leq b \leq n.$$

4.3. The Factor Problem

4.3.1. Introduction

First let us define $\mathbf{deg_G(v)}$ to be the degree of a vertex v in a graph G. If the graph under consideration is unambiguous, then **deg(v)** may be used.

Given a graph G and a non negative integer valued function d defined on V(G), consider the following question: Is there a subgraph H of G such that every vertex $v \in V(G)$ has degree d(v) in H? This is called the factor problem, and if H exists it is called a factor of G. (This is usually called the f–factor problem, where f is the function defined on V(G), but since f has been consistently used to denote a flow, a slight change from the usual terminology seems appropriate.) One should observe that the factor problem is the natural generalisation of the maximum matching problem, since a maximum matching in G is also a factor of G, where d(v) = 1 for all $v \in V(G)$. Thus a maximum matching in G is also called a 1–factor of G, and in general a factor of G in which d(v) = k for all $v \in V(G)$, is called a k–factor of G. It is assumed that $d(v) \leq \deg(v)$ since if $d(v) > \deg(v)$ then the required factor clearly does not exist.

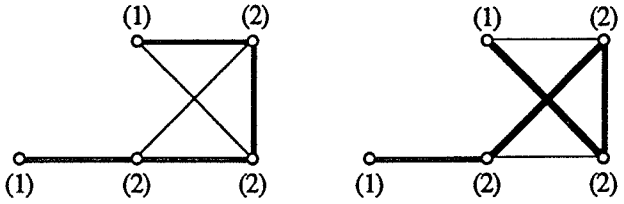


Figure 4.1
Two copies of a graph G, with d(v) shown in parentheses for each vertex v in G. Two possible factors of G are shown, one in each copy. The thicker edges are the edges of the factor.

If H exists, it is clear that V(H) = V(G), so H is called a **spanning subgraph** of G. However, it is easy to see that this is not at all restrictive. If we are interested in only some proper subset U of the vertices in G, so that d(v) is non zero only for vertices that are in U, then H certainly cannot contain any edge that is not in the subgraph G[U] induced by U. Thus we simply apply the factor problem to G[U] instead of G.

4.3.2. Tutte's Transformation

An existing method of finding a factor in a graph G is called Tutte's Transformation. This works by transforming the factor problem into a maximum matching problem in such a way that there is a perfect matching if and only if there is a solution to the factor problem, and a perfect matching gives the solution to the factor problem. Much work has been done on methods of solving the maximum matching problem, so any one of these well known methods, such as Edmonds' Algorithm, can then be used to solve the resulting maximum matching problem, and thus the original factor problem.

The transformation is clearly the key to this method, and it is this transformation that is presented here. A new graph G^* is created from G using the following method:

For each edge e=uv in G, create two vertices e_u and e_v and an edge e_ue_v in G^*. For each vertex v in G, create in G^* a set Y_v of deg(v) – d(v) vertices. Join each of these to all vertices e_v, for all edges e incident on v. If we let X_v be the deg(v) vertices e_v, then

in G^* each $X_v \cup Y_v$ induces a complete bipartite graph with bipartition $\{X_v, Y_v\}$. G^* has $2\varepsilon + \sum_{v \in V(G)} (\deg(v) - d(v))$ vertices.

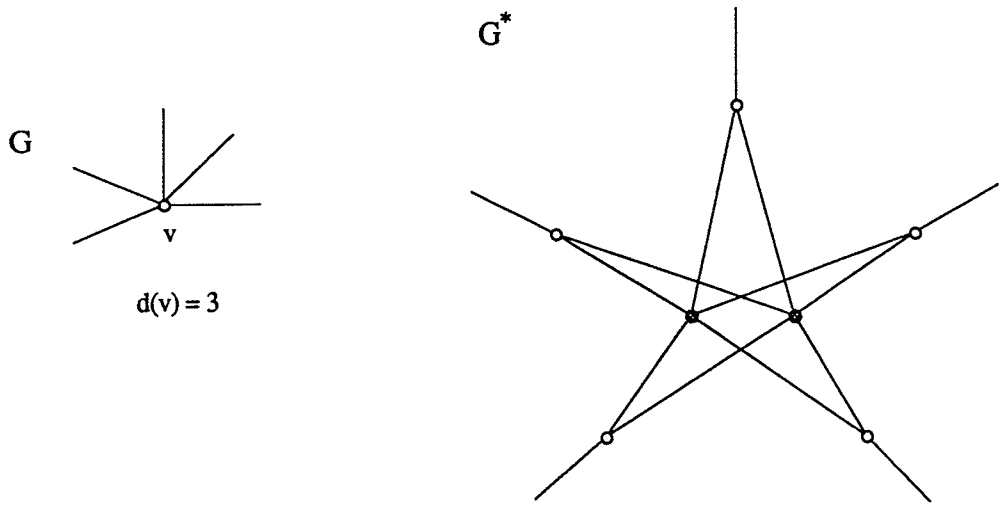


Figure 4.2
The edges and vertices in G^* for a single vertex in G.

Now find a maximum matching in G^*. If such a matching M exists, then form a spanning subgraph H of G as follows: uv is an edge of H if and only if $e_u e_v$ is an edge of M. H is then a factor of G.

To verify this, consider a single component $X_v \cup Y_v$ of G. Vertices in Y_v are only joined to vertices in X_v in G^*, so exactly $|Y_v| = \deg(v) - d(v)$ of the vertices in X_v are matched to a vertex in Y_v. Since $|X_v| = \deg(v)$, there are therefore d(v) remaining vertices in X_v matched to vertices that are not in Y_v. But these are edges of the form $e_u e_v$ and there are d(v) of them. Hence v has degree d(v) in H.

It is also clear that a factor H in G defines a maximum matching in G^*, so there is a maximum matching in G^* if and only if there is a factor in G.

Now a quick note on the complexity of this technique. Suppose G has n vertices and ε edges. The number of vertices in G^* is $2\varepsilon + \sum_{v \in V(G)} (\deg(v) - d(v)) \leq 4\varepsilon = O(\varepsilon)$. The number of edges in G^* is $\varepsilon + \sum_{v \in V(G)} \deg(v) \cdot (\deg(v) - d(v)) \leq n \cdot ((n-1)^2) + \varepsilon$ $= O(n^3)$, since $\deg(v) \leq n - 1$. Hence the complexity of this algorithm when used in

conjunction with one of the best known maximum matching algorithms, say the $O(\varepsilon\sqrt{n})$ algorithm described in [3], is $O(n^3\sqrt{\varepsilon})$. This is unquestionably a weak upper bound, but is does give the flavour of the orders of magnitude involved.

4.3.3. Using Flows to Find Factors in Graphs

4.3.3.1 Bipartite Graphs

The idea of using flows to find a factor in a bipartite graph is not new. If you refer back to the section that describes how to use flows to find a maximum matching or 1–factor in a bipartite graph G, you see that it is easy to extend the idea contained therein to find an arbitrary factor, if one exists. In order to make this extension it is only necessary to notice that in the network N, which we create by transforming G, every edge sx_i or y_it has a capacity which is equal to the degree that x_i and y_i respectively require in H. When we are searching for a maximum matching this degree is of course one, but there is no reason whatsoever that it should not be higher. We can therefore find a factor by giving the edges sx_i capacity $d(x_i)$, and the edges y_it capacity $d(y_i)$. Hence the algorithm presented earlier is nothing more than a special case of this more general algorithm for solving the factor problem in bipartite graphs.

4.3.3.2. General Graphs – The Balanced Network Method

This is where the balanced networks come in. As noted at the end of this section, the method described here extends very easily to **multi–graphs**, which are graphs that are not simple, but for now let us suppose that G is a simple graph with $V(G) = \{1,2,...n\}$. Create a balanced network N from G as follows:

- $V(N) = \{s,t\} \cup \{x_i,y_i\}$, $1 \le i \le n$;
- $E(N) = \{sx_i,y_it \mid 1 \le i \le n\} \cup \{x_iy_j,x_jy_i \mid ij \in E(G)\}$;
- $cap(sx_i) = cap\ (y_it) = d(i)$;
- $cap(x_iy_j) = cap(x_jy_i) = 1$.

It is easy to see that a maximum balanced flow in N that saturates all the edges incident on s defines a factor H of G; E(H) is simply $\{v_iv_j \mid x_iy_j$ and x_jy_i carry flow$\}$. To verify that H is indeed a factor of G, consider an arbitrary x_i in N; $\text{flow}^+(x_i) = d(i)$, so since $\text{flow}^+(x_i) = \text{flow}^-(x_i)$ by the conservation condition, there are exactly d(i) pairs of complementary edges incident on x_i that carry flow. But there is exactly one edge incident on i in H for every such pair of complementary edges in N, so $\deg_H(i) = d(i)$.

Conversely, a factor H can be used to find a balanced flow f in N that saturates all the edges incident on s. One just puts flow on $\{x_iy_j, x_jy_i\}$ if ij is an edge of H, and then puts the appropriate amount of flow on the edges incident on s and t so that the conservation condition is satisfied. f is clearly a maximum flow in N, because $\text{cap}([S,\bar{S}]) = \text{val}(f)$, where $S = \{s\}$. Hence there is an f-factor in G if and only if a maximum balanced flow in N saturates all the edges incident on s.

The complexity of the algorithm is $O(m\varepsilon)$, where m is the number of edges in the required factor, since the complexity of finding a single augmenting path is $O(\varepsilon)$, and we must find m such paths. This is $O(\varepsilon^2)$ for simple graphs.

Let us now once again use the notation introduced in section 4.1, and let ΣU denote $\sum_{x_i \in U} \text{cap}(sx_i)$, for any $U \subseteq V(N)$, where U consists entirely of complementary pairs of vertices. Thus $\Sigma A = \sum_{a \in A^*} d(a)$, and similarly for B,C,D and every C_i. Also, let $V = V(N) - \{s,t\} = A \cup B \cup C \cup D$. From the foregoing discussion and the Max-Balanced-Flow–Min-Balanced-Cut theorem we can say that G is without a factor if and only if N contains a balanced edge cut K such that $\text{balcap}(K) < \sum_{i=1}^{n} \text{cap}(sx_i) = \Sigma V$. But $\text{balcap}(K) = 2\Sigma A + |B,B| + |B,C| + |B,D| + \Sigma C + \Sigma D - k$ and $\Sigma V = \Sigma A + \Sigma B + \Sigma C + \Sigma D$, which gives us:

Theorem 4.9

G is without a factor if and only if $\Sigma A < \Sigma B + k - |B,B| - |B,C| - |B,D|$.

Let $A^* \subseteq V(G)$ and $B^* \subseteq V(G) - A^*$. If C_i^* is a component of $G - (A^* \cup B^*)$ then $[B^*,C_i^*]$ is used to denote the set of edges of G with one endpoint in C_i^* and the other in B^*; thus $|B^*,C_i^*| = |B,C_i|$. $odd\Sigma(A^*,B^*)$ denotes the number of components C_i of $G - (A^* \cup B^*)$ such that $|B^*,C_i^*| + \sum_{c \in C_i^*} d(c)$ is odd.

Factor Theorem

G is without a factor if and only if there is a subset A^* of V(G) and a subset B^* of $G-A^*$ such that:

$$\sum_{a \in A^*} d(a) < odd\Sigma(A^*,B^*) + \sum_{b \in B^*} (d(b) - deg_{G-A^*}(b))$$

Proof:

Suppose G has no factor and let f be a maximum balanced flow in N. By corollary 2.20 f defines a balanced edge cut $K = [S,\bar{S}]$ in N, where S is the set of all vertices that are s-reachable in (N,f). By theorem 4.9 $\Sigma A < \Sigma B + k - |B,B| - |B,C| - |B,D|$.

It is clear that $\Sigma B - |B,B| - |B,C| - |B,D| = \sum_{b \in B^*} d(b) - \sum_{b \in B^*} deg_{G-A^*}(b) = \sum_{b \in B^*} (d(b) - deg_{G-A^*}(b))$. Now consider an arbitrary C_i in N. There are no edges from C_i to D or to any C_j, $i \neq j$, in N, so C_i^* is a component of $G - (A^* \cup B^*)$. By definition, $|B,C_i| + \Sigma C_i$ is odd; but $|B^*,C_i^*| = |B,C_i|$ and $\sum_{c \in C_i^*} d(c) = \Sigma C_i$, so C_i^* is a component of $G - (A^* \cup B^*)$ such that $|B^*,C_i^*| + \sum_{c \in C_i^*} d(c)$ is odd. Thus $odd\Sigma(A^*,B^*) \geq k$.

Combining the above results we have: $\sum_{a\in A^*} d(a) = \Sigma A < \Sigma B + k - |B,B| - |B,C| - |B,D| \leq odd\Sigma(A^*,B^*) + \sum_{b\in B} (d(b) - deg_{G-A}(b))$.

Now suppose the sets A^* and B^* exist. Let $C_1^*,C_2^*,...,C_k^*$ be the components of $G - (A^* \cup B^*)$ such that $|B^*,C_i^*| + \sum_{c\in C_i^*} d(c)$ is odd, $C^* = C_1^* \cup C_2^* \cup ... \cup C_k^*$ and $D^* = V(G) - (A^* \cup B^* \cup C^*)$. If we form the balanced network N that corresponds to G, and the edge cut K in N defined by A,B,C and D, then, from above, K is clearly a balanced edge cut. Furthermore,

$$\Sigma A = \sum_{a\in A^*} d(a) < odd\Sigma(A^*,B^*) + \sum_{b\in B^*} (d(b) - deg_{G-A^*}(b)) = k + (\Sigma B - |B,B| - |B,C| - |B,D|).$$

Hence, by theorem 4.9, G does not have a factor. □

Thus we have proved the factor theorem using the theory of balanced networks. As was the case with perfect matchings, in the case that no factor exists in G the maximum balanced flow algorithm actually *finds* A^* and B^*, the sets from the statement of the factor theorem. That is, it is an algorithm for either finding the factor or finding the two sets that prove, by the factor theorem, that G does not have a factor.

Now suppose we relax the requirement that H be a subgraph of G, and demand instead that if uv is an edge of H then uv is an edge of G, and that the multiplicity of an edge in H does not exceed m. We call such a graph H an **m–multifactor**. If G has an ∞–multifactor then we say that G is **soluble**. (Existing terminology calls G f–soluble, where f is the function defined on V(G). As was the case with f–factors, a slightly different terminology is used here.) From preceding arguments, we have the following theorem:

Theorem 4.10

H does not exist if and only if $\Sigma A < \Sigma B + k - m|B,B| - m|B,C| - m|B,D|$.

Let $odd\Sigma_m(A^*,B^*)$ denote the number of components C_i of $G - (A^* \cup B^*)$ such that $m|B^*,C_i^*| + \sum_{c \in C_i^*} d(c)$ is odd.

Theorem 4.11

G is without an m–multifactor if and only if there is a subset A^* of V(G) and a subset B^* of $G–A^*$ such that:

$$\sum_{a \in A^*} d(a) < odd\Sigma_m(A^*,B^*) + \sum_{b \in B^*} (d(b) - m \bullet deg_{G-A^*}(b))$$

Proof:

The proof is similar to that of the factor theorem. Suppose G has no m–multifactor and let f be a maximum balanced flow in N. By corollary 2.20 f defines a balanced edge cut $K = [S,\bar{S}]$ in N, where S is the set of all vertices that are s-reachable in (N,f). By theorem 4.10 $\Sigma A < \Sigma B + k - m|B,B| - m|B,C| - m|B,D|$.

It is clear that $\Sigma B - m|B,B| - m|B,C| - m|B,D| = \sum_{b \in B^*} d(b) - \sum_{b \in B^*} m \bullet deg_{G-A^*}(b)$

$= \sum_{b \in B^*} (d(b) - m \bullet deg_{G-A^*}(b))$. Now consider an arbitrary C_i in N. By a previous argument, C_i^* is a component of $G - (A^* \cup B^*)$ such that $|B^*,C_i^*| + \sum_{c \in C_i^*} d(c)$ is odd.

Thus $odd\Sigma_m(A^*,B^*) \geq k$.

Combining the above results we have: $\sum_{a \in A^*} d(a) = \Sigma A < \Sigma B + k - m|B,B| -$

$m|B,C| - m|B,D| \leq odd\Sigma_m(A^*,B^*) + \sum_{b \in B} (d(b) - m \bullet deg_{G-A}(b))$.

Now suppose the sets A^* and B^* exist. Let $C_1^*,C_2^*,...,C_k^*$ be the components of $G - (A^* \cup B^*)$ such that $m|B^*,C_i^*| + \sum_{c \in C_i^*} d(c)$ is odd, $C^* = C_1^* \cup C_2^* \cup ... \cup C_k^*$ and $D^* = V(G) - (A^* \cup B^* \cup C^*)$. If we form the balanced network N that

corresponds to G, and the edge cut K in N defined by A,B,C and D, then, from above, K is clearly a balanced edge cut. Furthermore,

$$\Sigma A = \sum_{a \in A^*} d(a) < odd\Sigma_m(A^*,B^*) + \sum_{b \in B^*} (d(b) - m \bullet deg_{G-A^*}(b)) = k + (\Sigma B -$$

$m|B,B| - m|B,C| - m|B,D|$). Hence, by theorem 4.10, G does not have an m–multifactor.□

Let us now extend the definition of def(G), so that $\mathbf{def(G)} = \sum_{v \in H} (d(v) -$ $deg_H(v)))$. Theorem 4.12 follows directly from the preceding discussion and theorem 4.13 is the generalisation of all the results in this section:

Theorem 4.12

When looking for an m–multifactor, def(G) > r if and only if $\Sigma A < \Sigma B + k -$ $m|B,B| - m|B,C| - m|B,D| - r$.

Theorem 4.13

When looking for an m–multifactor, def(G) > r if and only if there is a subset A^* of V(G) and a subset B^* of $G–A^*$ such that:

$$\sum_{a \in A^*} d(a) < odd\Sigma_m(A^*,B^*) + \sum_{b \in B^*} (d(b) - m \bullet deg_{G-A^*}(b)) - r$$

Proof:

The proof is similar to that of the factor theorem, but contains the extra term r.

4.3.3.3. Balanced Flows and Soluble Graphs

We may determine whether or not a graph is soluble using the balanced network technique. We create the balanced network corresponding to G, give the edges sx_i and y_it capacity d(i) and all other edges capacity ∞. Then a balanced flow in N defines a solution H for G, and conversely. The previous definition of |A,B| is now extended to

denote $\sum_{e\in[A,B]} cap(e)$. Let $A^* \subseteq V(G)$ and B^* denote the set of vertices of $G-A^*$ which are joined only to vertices of A^*. $odd\Sigma(A^*)$ denotes the number of components C_i of $G-A^*$ such that C_i contains more that one vertex and $\sum_{c\in C_i^*} d(c)$ is odd.

The following theorem gives a known necessary and sufficient condition for a graph to be soluble [7]:

Solubility Theorem

G is not soluble if and only if there is a subset A^* of V(G) such that:

$$\sum_{a\in A^*} d(a) < odd\Sigma(A^*) + \sum_{b\in B^*} d(b),$$

where B^* is the set of vertices of $G-A^*$ which are joined only to vertices of A^*.

Proof:

Suppose that G is not soluble and let f be a maximum balanced flow in N. By corollary 2.20 f defines a balanced edge cut $K = [S,\bar{S}]$ in N, where S is the set of all vertices that are s-reachable in (N,f). By theorem 4.10, $\Sigma A < k + \Sigma B$, since it is clear that we must have $|B,B| = |B,C| = |B,D| = 0$ when $m=\infty$. Thus there are no edges from B^* to any C_i^*, so the C_i^* are all connected components in $G-A^*$. It also follows that $\sum_{c\in C_i^*} d(c)$ is odd, since $\sum_{c\in C_i^*} d(c) = \Sigma C_i = |B,C_i| + \Sigma C_i = cap(K_i)$, which is odd. Also, each C_i^* must contain more that one complementary pair of vertices, or it is not connected. Hence $odd\Sigma(A^*) \geq k$, and so $\sum_{a\in A^*} d(a) = \Sigma A < k + \Sigma B \leq odd\Sigma(A^*) + \Sigma B = odd\Sigma(A^*) + \sum_{b\in B^*} d(b)$.

Now suppose the sets A^* and B^* referenced in the theorem exist. Let $C_1^*, C_2^*, \ldots, C_k^*$ be the components of $G-A^*$ with more than one vertex such that $\sum_{c\in C_i^*} d(c)$ is odd, $C^* = C_1^* \cup C_2^* \cup \ldots \cup C_k^*$ and $D^* = V(G) - (A^* \cup B^* \cup C^*)$. If we

form the balanced network N that corresponds to G, and the edge cut K in N defined by A,B,C and D, then, from above, K is clearly a balanced edge cut. Furthermore, $[B^*,B^*] = [B^*,C^*] = [B^*,D^*] = \emptyset$, by the definition of B, so $|B,B| = |B,C| = |B,D| = 0$. Thus $\Sigma A + |B,B| + |B,C| + |B,D| = \Sigma A = \sum_{a \in A^*} d(a) < \text{odd}\Sigma(A^*)$ $+ \sum_{b \in B^*} d(b) = k + \Sigma B$, so by theorem 4.10, G is not soluble. □

As is the case with factors, it is easy to see that the maximum balanced flow algorithm will either find an f–solution or a set $A^* \subseteq V(G)$ that proves, by the solubility theorem, that G is not soluble.

4.3.3.4. Balanced Network Blossoms Generalise Edmonds' Blossoms

First we give some terminology. If H is a subgraph of G then G\H denotes the graph G^* with $V(G^*) = V(G)$ and $E(G^*) = E(G) - E(H)$. Now suppose there is a function d defined on the vertices of G, and that H is a subgraph of G such that $\deg_H(v) \leq d(v)$ for every vertex v in G. An **H–alternating walk** in G is a walk W in G such that: W begins at a vertex v in G such that $\deg_H(v) < d(v)$; W begins with an edge in G\H; the edges of W are alternately in G\H in H. An **H–augmenting walk** in G is an alternating walk W in G such that: if W begins and ends at the same vertex v then $\deg_H(v) \leq d(v) - 2$; W ends with an edge in G\H.

If we use 1,2,3,... to number the edges on an H–alternating walk W, then the edges in G\H on W always have an odd number; thus we call such edges **odd edges**, and for a similar reason the edges in H are called **even edges**. Observe that if W is an H–augmenting walk, then if we set $H^* = H \oplus W$, where $\oplus$ denotes the "exclusive or" operation, then H^* has more edges than H but still satisfies the constraint that $\deg_H(v) \leq d(v)$ for every vertex v in G. Performing this $\oplus$ operation is called **augmenting on W**. The notion of a saturated vertex is now extended; v is said to be **saturated** if $\deg_H(v) = d(v)$.

For the remainder of this section it is assumed, unless otherwise stated, that N is a balanced network created from a graph G in the manner described in section 4.1.4.1, and f is a balanced flow in N.

Let us now attempt to analyse what the balanced max–flow algorithm is doing in terms of the underlying graph G. When the balanced max–flow algorithm searches for an augmenting path in N, it is really searching for an H–augmenting walk in G, since there is a 1–1 correspondence between the two.

Before explaining this remark further, let us first elaborate on the relationship between G and N. N contains exactly one pair of complementary vertices $\{x_i,y_i\}$ for each vertex i in G, plus a source and a target. Thus we may think of x_i and y_i as the x-copy and the y-copy of i in N respectively. Hence a path in N defines a unique walk in G, if we disregard the copy of the vertex we are visiting. For example, the path $x_iy_jx_k$ in N uniquely defines the path ijk in G. Also, recall that edges in N between X and Y that carry flow correspond to edges in G that are in the subgraph H that the algorithm has discovered so far, and edges between X and Y that do not carry flow correspond to edges that are not in H.

Now let us first examine an augmenting path P in N. It starts with an edge sx_i, where $\text{flow}(sx_i) < \text{cap}(sx_i)$. Thus i is a vertex in G, such that $\deg_H(i) < d(i)$. P then follows an edge x_iy_j that has no flow, which is equivalent to following an edge ij in G, such that ij is not in H. The next edge that P follows is y_jx_k, and this edge does carry flow; this is equivalent to following the edge jk in G, where jk is in H. These last two steps are repeated until, after repeating the first step, P takes an edge y_jt instead of an edge y_jx_k. As before, j is a vertex in G such that $\deg_H(j) < d(j)$.

It should now be clear that an augmenting path in N corresponds to a unique H–augmenting walk in G, but we must do a little more work before deciding on the

converse. Observe that an H–augmenting walk does not contain any edge twice, and does not begin and end at the same vertex v if $\deg_H(v) + 1 = d(v)$.

Now let us determine the features of the walk W in G determined by an invalid st-path R in N. R contains a pair of complementary edges with residual capacity 1, so let these edges be e and e′. If e has an endpoint incident on either s or t so that, for example, $e = sx_i$ and $e' = y_it$, then W begins and ends at vertex i, and $\deg_H(i) + 1 = d(i)$. If e does not have an endpoint incident on either s or t then $e = x_iy_j$ and $e' = x_jy_i$, and W therefore contains the edge ij twice, but in a different direction the second time. In both cases it is clear that W is not an H–augmenting walk in G. Thus an H–augmenting walk in G cannot correspond to an invalid st–path in N.

However, it is clear that an H–augmenting walk W does correspond to an unsaturated path in N; if W begins ijk... then P begins $sx_iy_jx_k$... . But is P unique? The answer is no, and to see this just notice that if W ends with kji then $sx_iy_jx_k$ is once again the beginning of an unsaturated path in N. Thus if W = 1234 is an H–augmenting walk in G then both $P = sx_1y_2x_3y_4t$ and $Q = sx_4y_3x_2y_1t$ are augmenting paths in N. P corresponds to walking along W in one direction and Q corresponds to walking it in the other. But closer examination of Q reveals that Q = P′! This may seem to be a rather startling coincidence, but in fact it exemplifies the fundamental relationship between any path P and P′ in N.

Let us now look at a couple of examples. Figure 4.3 (a) shows a graph G_1 in which we are looking for a maximum matching. Notice that P = 123456 is an augmenting path in G_1, and is also the only augmenting path. Let us suppose that the Hungarian algorithm is applied to this graph in an attempt to find P, and that the search begins at vertex 1. 3 will be visited by the search, and if 5 is searched before 4 on this visit, and it may well be, then we get the search tree T shown in figure 4.3 (b). But only the outer vertices in T are put on the ScanQ; thus 5, which is an inner vertex, is never added to T, and the edge 56, which completes the only augmenting path in G_1, is never

searched. This illustrates why the Hungarian algorithm does not work for non–bipartite graphs.

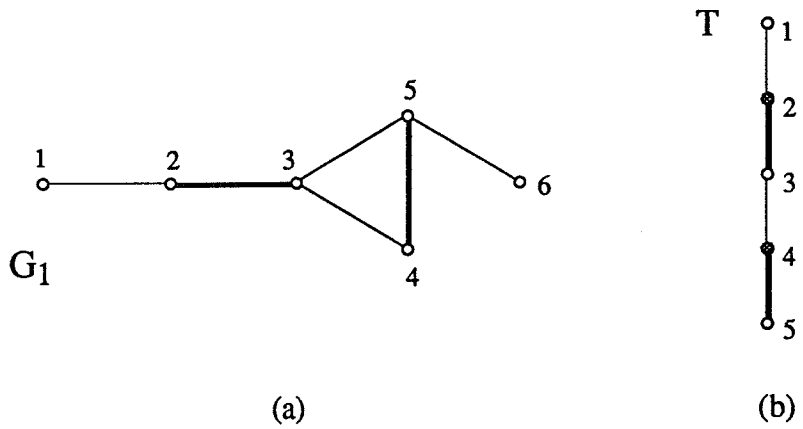


Figure 4.3
Part (a) shows a graph G_1 in which we are looking for a 1-factor. Indicated in G_1 is a subgraph H_1 such that $\deg_{H_1}(v) \leq d(v)$ for every vertex v in G_1 (the thicker edges belong to H_1). Part (b) shows a possible search tree T after the Hungarian algorithm has been applied to G_1. The inner vertices in T are shaded.

In developing an algorithm to find maximum matchings in general (i.e. possibly non–bipartite) graphs, Edmonds overcame this by recognising that the problem is the existence of an odd cycle in G_1, combined with the fact that if the search proceeds around this cycle in the wrong direction, an augmenting path may be missed. His algorithm makes use of the key observation that there is an augmenting path in a graph G if and only if there is an augmenting path in G/B, where G/B denotes the graph obtained from G by shrinking into a single vertex b all the vertices on an alternating odd cycle B. Thus all the vertices in B are put onto the ScanQ and eventually searched, so that all the edges incident on the new vertex b are examined. Of course it is possible to have odd cycles within odd cycles, and Edmonds called such a recursive collection of odd cycles a blossom. Thus B = {3,4,5} is a blossom, and is in fact itself a collection of blossoms, since Edmonds algorithm initially places each vertex v in the search tree into a blossom C = {v}.

Let us now look at another example, specifically the one depicted in Figure 4.4. G_2 contains exactly one H_2–augmenting walk, namely W = 123425. Thus in order to find the required factor, we must perform a search of G_2 that allows us to visit vertex 2 twice. It is worth elucidating the reason that this second visit is required. The first visit to 2 is on an edge in G\H, so we may extend an alternating path from 2 only on edges in H. But in order to find W the search must at some point extend an alternating path from 2 on an edge in G\H. Hence a second visit to 2 is required and, incidentally, this is why we must deal with H–augmenting walks when looking for general factors, rather than the H–augmenting paths that appear in the special case of maximum matchings or 1–factors. However, one should notice that there is a restriction on the kind of second visit that is allowed. If a vertex was first visited on an edge in G\H, then it may only be visited a second time on an edge in G, and conversely.

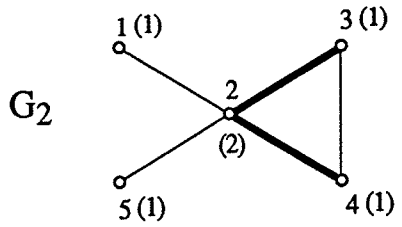


Figure 4.4
A graph G_2, with d(v) shown in parentheses for each vertex v in G_2. Also indicated in G_2 is a subgraph H_2 such that $\deg_{H_2}(v) \leq d(v)$ for every vertex v in G_2 (the thicker edges belong to H_2).

Balanced networks take care of this quite nicely. Allowing one visit to a vertex in a balanced network N formed from a graph G is equivalent to allowing two visits to a vertex i in G, since there are two copies in N of each vertex in G. Furthermore, provided that it is not first visited from s, the x–copy of i is always visited on an edge that carries flow, which corresponds to an edge in H, and the y–copy is always visited on an edge that does not carry flow, which corresponds to an edge in G\H. If the x–copy is visited on an edge from s, this just means that i does not yet have its required degree d(i) in H; then all the edges to y–copies of vertices in G are examined, which is

equivalent to examining all the edges in G\H that are incident on i. Thus no visit to i in G on an edge in H will be required, which is good because x_i will now be marked searched in N.

There is one other hitherto unmentioned point about searching for a factor, that balanced networks take care of automatically. In the maximum matching case we are free to search for an alternating path P that begins and ends at an unsaturated vertex; this is because if a vertex is unsaturated, then there is no edge in H incident on it, so P must begin and end with an edge in G\H. However, in the general factor case it is quite possible for an alternating walk to begin or end on an edge in H at an unsaturated vertex. We must ensure that we do not consider such a walk to be an H–augmenting walk, because it cannot be used to augment H. But balanced networks are bipartite graphs, and any path from Y to X (and an unsaturated path P in N is such a path) has an odd number of edges; this eliminates the possibility that exactly one of the first and last edges on P is an edge in H. Furthermore, P visits the x–copy of some vertex first, so the first edge on the path defined by P in G is an edge in G\H.

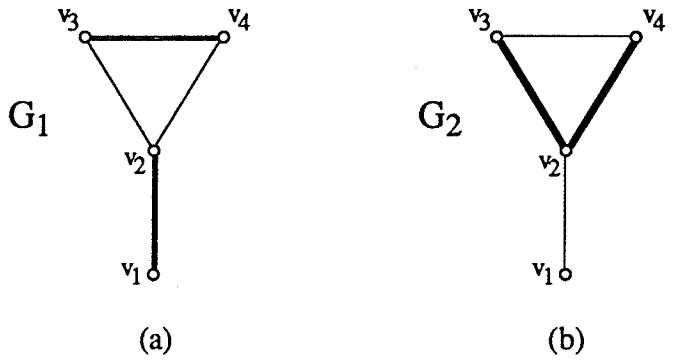


Figure 4.5
The two types of blossoms required for factors. The thick edges are edges in H.

Let us now take a closer look at Edmonds' blossoms. Edmonds' algorithm looks only for a 1–factor, and since there can never be two edges in H incident on the same vertex in a 1–factor, Edmonds' blossoms always have the form shown in figure 4.5

(a). The important feature is that the odd cycle has more edges in G\H than in H. However, when searching for a general factor, it may be necessary to create blossoms with more edges in H than in G\H; such a blossom is illustrated in figure 4.5 (b).

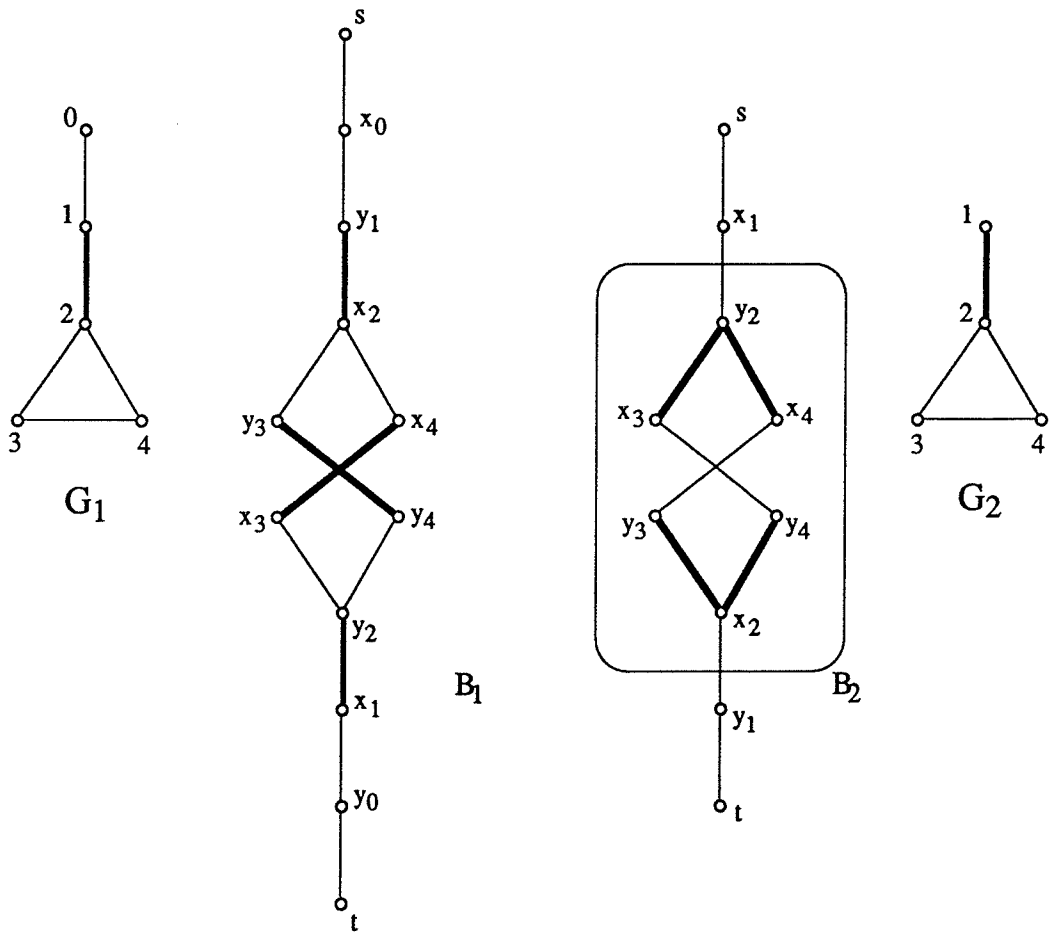


Figure 4.6
Two graphs, and the mirror networks created by a search beginning at vertex 0 in G_1 and vertex 1 in G_2. It is assumed that $d(v) = deg_H(v)$ for every vertex except 0 in G_1 and 1 in G_2.

But consider what these blossoms represent in a balanced network. If we add the edge 01 to G_1, then the mirror networks created by a search of G_1 from v_0 and G_2 from v_1 are shown in parts (a) and (b) respectively of figure 4.6. The balanced network blossoms created are indicated. Notice that they consist of exactly both copies of every vertex in the blossom in the original graph. Thus, as one would expect, the max–

balanced–flow algorithm for balanced networks handles both kinds of blossoms indicated in figure 4.5.

There is an interesting observation that can be made by examining the paths in these blossoms from w′ to w, the base of the blossom. Consider B_1. The base is $w' = y_2$, and there are two paths in B_1 from $w = x_2$ to y_2: $P = x_2y_3x_4y_2$ and $Q = x_2y_4x_3y_2$. But $Q=P'$, so these paths represent searching the blossom in G_1 in opposite directions from the base. At first glance it appears that this may enable us to do away with blossoms entirely, but the simple example in figure 3.1 illustrated that this is not the case, since an augmenting path could not be found in the network using the standard techniques.

There is one other point that is of interest – if P is an invalid st–path in N, then what does the walk defined by P in G look like? Let e and e′ be the pair of complementary edges with capacity 1 on P; if e and e′ have an endpoint incident on s or t then let P be called a **type 1** invalid path, otherwise let it be a **type 2** invalid path. The residual capacity of the edge sx_i tells us the amount by which we are allowed to increase $\deg_H(i)$, and still have $\deg_H(i) \leq d(i)$. Clearly, augmenting on an H–augmenting walk W increases by 1 the degree in H of the first and last vertices on W, and does not change the degree of any other vertex in H. Hence a type 1 invalid st–path corresponds to an alternating walk in G that starts and ends at a vertex v, where there can be only one more edge incident on v in H. In figure 4.7 (a), both 1231 and 4564 are such walks.

A type 2 invalid path P contains some pair of complementary edges x_iy_j and x_jy_i, and thus follows some edge ij in G twice. If ij is in H then if we try to augment on W this means that we will try to take ij out of H twice, thus introducing a 'negative' edge into H. In figure 4.7 (b) 12345326 is such a walk. Interestingly, if we allow negative edges in a graph, whatever they may be, then W is not really an invalid walk at all, since everything works out when we augment on W! Also, if we allow negative edges in G then we must really allow a vertex to have a negative degree, and if we do this then type 1 invalid paths are not invalid either. Finally, if ij is in G\H then we try to add

it to G twice. This may seem alright for multi–graphs, but we will see that it violates the problem boundary, just as the previous two examples violate the inherent problem boundary that limits deg(v) to the non negative integers.

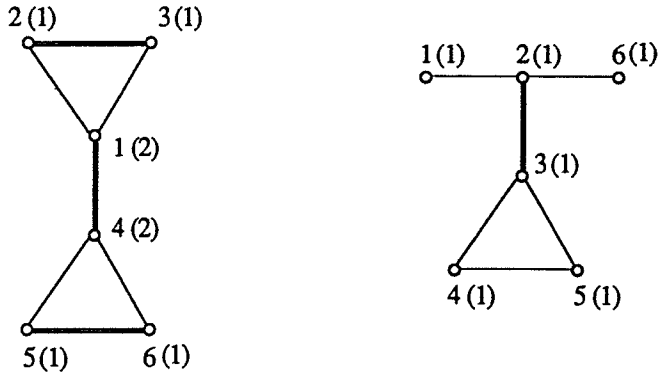


Figure 4.7
Two graphs with walks defined by invalid paths in N. d(v) is in parentheses.

4.3.3.5. The Multi–Graph Case

The extension of the simple graph case to the multi–graph case is easy. If V(G) = {1,2,...n}, then create a balanced network N from G as follows:

- $V(N) = \{s,t\} \cup \{x_i,y_i\}$, $1 \leq i \leq n$;
- $E(N) = \{sx_i, y_it \mid 1 \leq i \leq n\} \cup \{x_iy_j, x_jy_i \mid ij \in E(G)\}$;
- $cap(sx_i) = cap(y_it) = d(i)$;
- $cap(x_iy_j) = cap(x_jy_i) = m(ij)$, where m(ij) is the maximum allowed multiplicity for the edge ij in G.

Again, it is easy to see that a maximum balanced flow in N that saturates all the edges incident on s defines a factor H of G; E(H) is simply $\{ij \mid x_iy_j, x_jy_i$ carry flow$\}$, and $m(ij) = flow(x_iy_j) = flow(x_jy_i)$.

The only point that has changed from the simple graph case is the fourth. This is really just a generalisation of the simple graph case, where m(ij) = 1 for every edge ij in G.

There is one point of interest with regard to the algorithm when it operates on this general case. Edges may now be unsaturated in *both directions*, which raises the possibility that an unsaturated path P may contain a pair of complementary edges, where one is a forward edge and the other is a backward edge; for example, both x_iy_j and y_ix_j could appear on P. However, this is not a problem; traversing x_iy_j on P is equivalent to adding $m_a = \text{flow}(x_iy_j)$ edges ij into G, and traversing y_ix_j is equivalent to subtracting $m_s = \text{flow}(y_ix_j)$ edges ij from G. The net result will therefore be to either add $0 \geq m_a - m_s > m_a$ edges or subtract $0 \geq m_s - m_a > m_s$ edges; in either case, if m_a and m_s were alright, then so is the net result.

However, with the algorithm given in this thesis, this is all purely academic, since it will never record such a path in the mirror network. This is illustrated by the multi–graph example shown in figure 4.8.

Only vertices 1 and 2 are unsaturated, and each requires one more edge incident on it in H. Notice that 1,3,6,10,11,7,3,8,12,9,5,2 is an H augmenting walk in G. Also notice that we may move from vertex 8 to vertex 12, and conversely, on an H–augmenting walk, whether we arrive at 8 on an odd or even edge; this is only possible in the multi graph case.

The non-trivial blossoms created by the algorithm as it forms the mirror network A are as follows, provided sx_1 is the first unsaturated edge examined: Discovery of the switch edge $y_{10}x_{11}$ causes the blossom $\{x_iy_i \mid i = 3,6,7,10,11\}$ based at x_3, or simply $B_1 = \{3,6,7,10,11\}$, base=x_3, to be formed; the next switch edge found by the algorithm is x_3y_8, and this causes formation of the blossom $B_2 = B_1 \cup \{8,4,1\}$, base = y_1. Thus vertex y_8 will eventually be searched. The multiple edge y_8x_{12} will thus be added to A, so vertex x_{12} will eventually be examined. Then the switch edge $x_{12}y_9$ will be found, and a blossom based at t and containing every vertex in the graph will be created, indicating that an augmenting path has been found.

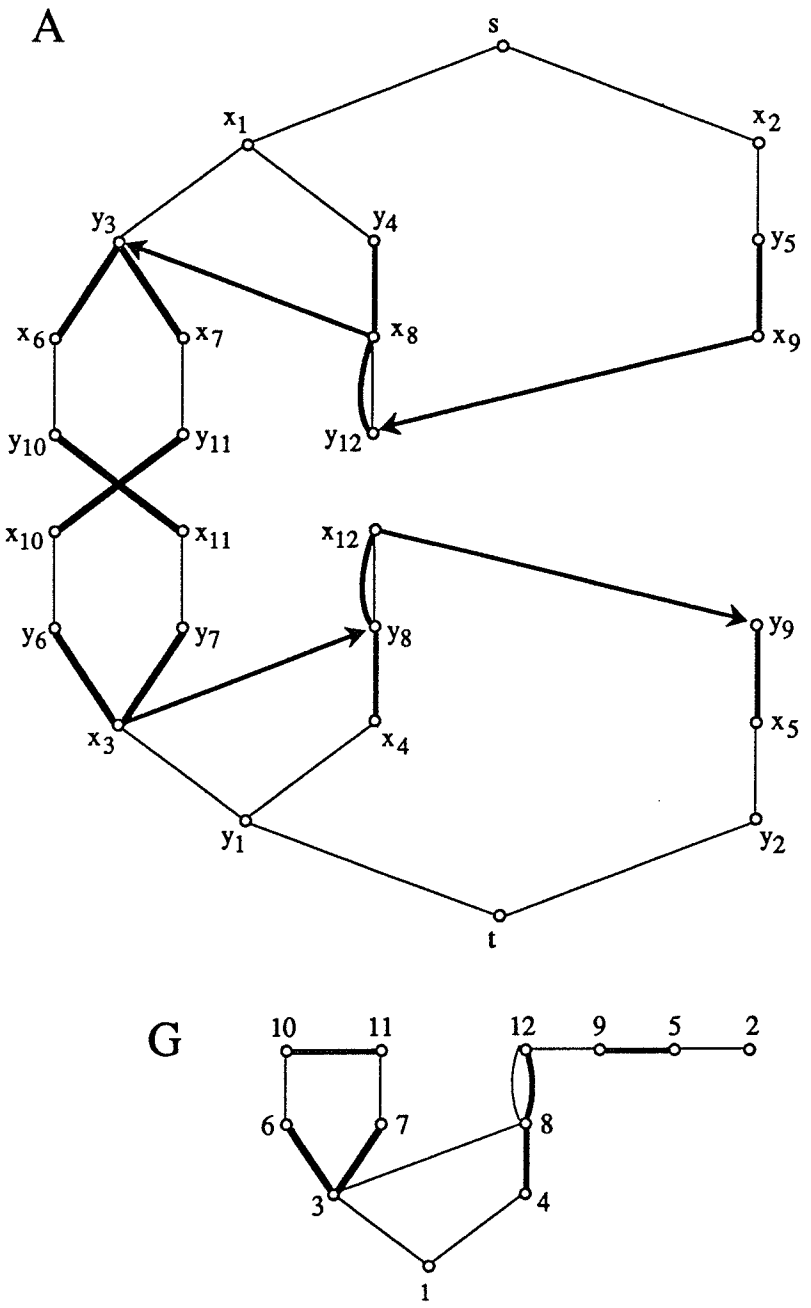


Figure 4.8
A Graph G in which we are looking for a factor. Only vertices 1 and 2 are unsaturated, and d(1) = d(2) = 1. The thicker edges belong to H.

The main purpose of this example was just to illustrate the balanced network max–balanced–flow algorithm operating on a multi graph but, as mentioned above, one can observe that the valid st-path found by the algorithm does not contain a pair of

complementary edges with opposite direction, and it is fairly easy to deduce that this is true in general.

As in the simple graph case, the complexity of the algorithm is $O(m\varepsilon)$, where m is the number of edges in the required factor. However, this is no longer $O(\varepsilon^2)$, since multiple edges are now allowed.

4.4. Future Research

Let G be a graph, and w be an integer valued function defined on E(G). Then G, together with w, is called a **weighted graph**, and w(e) is called the **weight of an edge** e. The problem of finding a matching M in G such that w(M) is as small as possible is called the **minimum weighted matching problem.** All the applications of balanced networks considered in this thesis use capacities rather than weights. However, this does seem like a natural extension of the work herein.

Another interesting research problem is to improve the complexity of the balanced network maximum balanced flow algorithm. One of the fastest maximum matching algorithms known, described in [3], explicitly uses the concepts of odd and even length alternating paths, and performing a breadth first search of the graph beginning simultaneously at all unsaturated vertices. But these are both things that the balanced network maximum balanced flow algorithm does *implicitly*. The problem that must be overcome is that of *efficiently* choosing switch edges so that the shortest augmenting paths are recorded in the mirror network. Micali and Vazirani [3] use a technique that delays the shrinking of blossoms, and one expects that a similar technique would work for balanced networks.

Appendix A

```
{*************************************************************************}
{*                                                                       *}
{*                          PROGRAM MaxBalFlow                           *}
{*                                                                       *}
{*************************************************************************}
{*                                                                       *}
{* Author :     Doug Stone                                               *}
{* Date :       July, 1991.                                              *}
{*                                                                       *}
{* Purpose:                                                              *}
{*    To find an f-factor of a graph, if one exists. More formally,      *}
{*    given a graph G and a function d defined on V(G), to find a        *}
{*    spanning sub-graph H of G, such that every vertex v in H has       *}
{*    degree ≤ d(v), and H has as many edges as possible.                *}
{* Algorithm:                                                            *}
{*    The balanced network maximum balanced flow algorithm.              *}
{* Remarks:                                                              *}
{*    This program gives an implementation of the above algorithm.       *}
{* Input :                                                               *}
{*    An initial graph G and d(v) for each vertex v in G. The input      *}
{*    file has the following format:                                     *}
{*          name$                                                        *}
{*          n$                                                           *}
{*          -1 1(1) 1(2) ... 1(k1) -d(1)$                                *}
{*          -2 2(1) 2(2) ... 2(k2) -d(2)$                                *}
{*                  ...                                                  *}
{*          -n n(1) n(2) ... n(kn) -d(n)$                                *}
{*    where:                                                             *}
{*        • $ denotes an end of line character;                          *}
{*        • i(j) denotes the vertex vj, where vertex i is adjacent to    *}
{*          vertices u1,u2,...,ur,v1,v2,...,vk, u1≤u2≤...≤ur≤i≤v1≤...    *}
{*          ≤vk;                                                         *}
{*        • 'name' is the name of the graph (alpha-numeric)              *}
{*        • 'n' is the number of vertices in G (positive integer);       *}
{*    The vertices in G must be numbered from 1 to n.                    *}
{* Output:                                                               *}
{*    1. confirmation of the input graph,                                *}
{*    2. the subgraph found by the algorithm.                            *}
{*                                                                       *}
{*************************************************************************}

program MaxBalFlow (input, output);

const
    MaxGraphVert = 50; { max # of verticies allowed in the input graph }
    MaxNetVert = (MaxGraphVert * 2) + 2; {max # verts allowed in network}
    source = 1;     { the source vertex is always vertex 1 }
    s = 1;          { alias for source }
    target = 0;     { the target vertex is always vertex 0 }
    t = 0;          { alias for target }
```

```
type
     GraphVertexType = 0..MaxGraphVert;  { a vertex in the graph }
     NetVertexType = -1..MaxNetVert; { a vertex in the network }
     DirectionType = (OutEdge, InEdge);  { type of an edge  }
     IntPtrType = ^integer;       { ptr to an integer variable }
     AdjPtrType = ^AdjNodeType;  { ptr to node in adjacency list }
     AdjNodeType = record        { node in adjacency list }
              vert: NetVertexType;{ adjacent to this vertex }
              dir: DirectionType; { direction of an edge }
              cap: integer;       { max capacity of edge }
              flow: IntPtrType;   { ptr to flow on this edge }
              next: AdjPtrType;   { ptr to next node in adjacency list }
          end; { record }
     NetworkType = record
              AdjList: array[NetVertexType] of AdjPtrType; { a network }
              NoVerts: integer;   { number of vertices in the network }
              SeqTot: integer;    { sum of degrees of all edges incident }
                                    { on the source }
          end; { record }

var
     Network: NetworkType; { the balanced network }
     infile: text;         { the input file }
     flow: integer;        { the maximum balanced flow }


{************************************************************************}
{*                                                                      *}
{*                        Function ComplementVert                       *}
{*                                                                      *}
{************************************************************************}
{*                                                                      *}
{* PURPOSE    : Given a vertex xi in the network, this function returns*}
{*    the vertex yi, and conversely.                                    *}
{* PASSED TO  :                                                         *}
{*    ui - the vertex to be complemented.                               *}
{* PASSED BACK:                                                         *}
{*    the complement of ui.                                             *}
{*                                                                      *}
{************************************************************************}
function ComplementVert (ui: NetVertexType): NetVertexType;

     begin
     if ui mod 2 = 0 then  { this is an x-vertex }
          ComplementVert := ui + 1
     else { this is a y-vertex }
          ComplementVert := ui - 1;
     end; { Function ComplementVert }
```

```
{*************************************************************************}
{*                                                                       *}
{*                          Function MapNtoG                             *}
{*                                                                       *}
{*************************************************************************}
{*                                                                       *}
{* PURPOSE    : Given a vertex in the network (xi or yi), this function*}
{*    returns the corresponding vertex (i) in the graph.                 *}
{* PASSED TO  : The vertex in the network.                               *}
{* PASSED BACK: The corresponding vertex in the graph.                   *}
{*                                                                       *}
{*************************************************************************}
function MapNtoG (u: NetVertexType): GraphVertexType;

    begin
    MapNtoG := u div 2;
    end; { Function MapNtoG }

{*************************************************************************}
{*                                                                       *}
{*                          Function MapGtoNx                            *}
{*                                                                       *}
{*************************************************************************}
{*                                                                       *}
{* PURPOSE    : Given a vertex (i) in the graph, this function returns *}
{*    the x copy (xi) of the vertex in the network.                      *}
{* PASSED TO  : The vertex in the graph.                                 *}
{* PASSED BACK: The x copy of the vertex in the network.                 *}
{*                                                                       *}
{*************************************************************************}
function MapGtoNx (u: GraphVertexType): NetVertexType;

    begin
    MapGtoNx := 2 * u;
    end; { Function MapGtoNx }

{*************************************************************************}
{*                                                                       *}
{*                          Function MapGtoNy                            *}
{*                                                                       *}
{*************************************************************************}
{*                                                                       *}
{* PURPOSE    : Given a vertex (i) in the graph, this function returns *}
{*    the y copy (yi) of the vertex in the network.                      *}
{* PASSED TO  : The vertex in the graph.                                 *}
{* PASSED BACK: The y copy of the vertex in the network.                 *}
{*                                                                       *}
{*************************************************************************}
function MapGtoNy (u: GraphVertexType): NetVertexType;

    begin
    MapGtoNy := (2 * u) + 1;
    end; { Function MapGtoNy }
```

```
{*************************************************************************}
{*                                                                       *}
{*                        Procedure PrintSubgraph                        *}
{*                                                                       *}
{*************************************************************************}
{*                                                                       *}
{* PURPOSE    : To print out the subgraph defined by the flow in a      *}
{*    balanced network.                                                  *}
{* PASSED TO  :                                                          *}
{*    Network - the balanced network in question.                        *}
{* PASSED BACK:                                                          *}
{*    Nothing.                                                           *}
{*                                                                       *}
{*************************************************************************}
procedure PrintSubgraph (var Network: NetworkType);

    var
        i: GraphVertexType; { the next vertex in the graph }
        xi: NetVertexType;  { x copy of i in the network }
        curr: AdjPtrType;   { ptr to adjacency node for current edge }

    begin
    writeln('Vertex    Adjacent to');
    for i := 1 to ((Network.Noverts - 2) div 2) do begin
        xi := MapGtoNx(i);
        write(i : 4, ' ' : 4);
        curr := Network.AdjList[xi];
        while curr <> nil do begin
            if curr^.flow^ > 0 then
                write(MapNtoG(curr^.vert) : 4);
            curr := curr^.next;
            end; { while }
        writeln;
        end; { for i }
    end; { Procedure PrintSubgraph }

{*************************************************************************}
{*                                                                       *}
{*                           Procedure MakeEdge                          *}
{*                                                                       *}
{*************************************************************************}
{*                                                                       *}
{* PURPOSE    : To create in the balanced network the pair of           *}
{*    complementary edges that correspond to a particular edge in the    *}
{*    input graph.                                                       *}
{* PASSED TO  :                                                          *}
{*    i,j - the two endpoints of the edge,                               *}
{*    capacity - the initial capacity of the edge,                       *}
{* PASSED BACK:                                                          *}
{*    Network - the updated network.                                     *}
{*                                                                       *}
{*************************************************************************}

procedure MakeEdge (i, j: GraphVertexType;
                                    capacity: integer;
                                    var Network: NetworkType);
```

```
var
     NuAdj: AdjPtrType;  { ptr to newly alloc'd adjacency list nodes }
     NuFlow: IntPtrType; { ptr to the flow on an edge }
     xi, xj, yi, yj: NetVertexType; { vertices in the network }

begin
xi := MapGtoNx(i);
yi := MapGtoNy(i);
xj := MapGtoNx(j);
yj := MapGtoNy(j);
new(NuFlow);
NuFlow^ := 0; { network initially contains the zero flow }

    { make x-node for directed edge xu-yv in network }
new(NuAdj);
with NuAdj^ do begin
     vert := yj;
     dir := OutEdge;
     cap := capacity;
     flow := NuFlow;
     next := Network.AdjList[xi];
     Network.AdjList[xi] := NuAdj;
     end; { with }

    { make y-node for directed edge xu-yv in network }
new(NuAdj);
with NuAdj^ do begin
     vert := xi;
     dir := InEdge;
     cap := capacity;
     flow := NuFlow;
     next := Network.AdjList[yj];
     Network.AdjList[yj] := NuAdj;
     end; { with }

    { make x-node for directed edge xv-yu in network }
new(NuAdj);
with NuAdj^ do begin
     vert := yi;
     dir := OutEdge;
     cap := capacity;
     flow := NuFlow;
     next := Network.AdjList[xj];
     Network.AdjList[xj] := NuAdj;
     end; { with }

    { make y-node for directed edge xv-yu in network }
new(NuAdj);
with NuAdj^ do begin
     vert := xj;
     dir := InEdge;
     cap := capacity;
     flow := NuFlow;
     next := Network.AdjList[yi];
     Network.AdjList[yi] := NuAdj;
     end; { with }
end; { Procedure MakeEdge }
```

```
{*************************************************************************}
{*                                                                       *}
{*                          Procedure MakeDegEdge                        *}
{*                                                                       *}
{*************************************************************************}
{*                                                                       *}
{* PURPOSE    : To create in the balanced network an edge that is       *}
{*    incident on the source vertex, and it's complementary edge which  *}
{*    is incident on the target vertex. The capacity of each of these   *}
{*    edges is the degree of some vertex in the subgraph that we are    *}
{*    trying to find.                                                   *}
{* PASSED TO  :                                                         *}
{*    i - some vertex in the graph,                                     *}
{*    degree - degree of u in the subgraph that we are trying to find,  *}
{* PASSED BACK:                                                         *}
{*    Network - the updated network.                                    *}
{*                                                                       *}
{*************************************************************************}
procedure MakeDegEdge (i: GraphVertexType;
                                    degree: integer;
                                    var Network: NetworkType);
    var
        NuAdj: AdjPtrType;  { ptr to newly alloc'd adjacency list nodes }
        NuFlow: IntPtrType; { ptr to the flow on an edge }
        xi, yi: NetVertexType; { verticies in the graph }

    begin
    xi := MapGtoNx(i);
    yi := MapGtoNy(i);
    new(NuFlow);
    NuFlow^ := 0; { network initially contains the zero flow }

    { make s-node for directed edge s-xi in network }
    { ( we don't need an x-node for these edges     }
    {   since we never follow them backwards )      }
    new(NuAdj);
    with NuAdj^ do begin
        vert := xi;
        dir := OutEdge;
        cap := degree;
        flow := NuFlow;
        next := Network.AdjList[s];
        Network.AdjList[s] := NuAdj;
        end; { with }

       { make y-node for directed edge yi-t in network }
       { ( we don't need a t-node for these edges      }
       {   since we never follow them backwards)       }
    new(NuAdj);
    with NuAdj^ do begin
        vert := target;
        dir := OutEdge;
        cap := degree;
        flow := NuFlow;
        next := Network.AdjList[yi];
        Network.AdjList[yi] := NuAdj;
        end; { with }
    end; { Procedure MakeDegEdge }
```

```
{************************************************************************}
{*                                                                      *}
{*                    Procedure CreateNetworkGGformat                   *}
{*                                                                      *}
{************************************************************************}
{*                                                                      *}
{* PURPOSE    : To read in (and echo) the graph for which a subgraph is*}
{*    to be found, and create the corresponding balanced network. The   *}
{*    graph will be in the standard format used by the 'Groups and      *}
{*    Graphs' program.                                                  *}
{* PASSED TO  :                                                         *}
{*    infile - the input file.                                          *}
{* PASSED BACK:                                                         *}
{*    Network - the updated network.                                    *}
{* REMARKS: The formatting of the listing of the input graph will be    *}
{*    spoiled if any vertex has degree greater than MaxDegree, but the  *}
{*    program will still execute properly.                              *}
{*                                                                      *}
{************************************************************************}
procedure CreateNetworkGGFormat (var infile: text;
                                 var Network: NetworkType);

    const
        MaxDegree = 10;     { max degree of a vertex in input graph }

    var
        i, j: integer;      { vertices in the graph }
        degree: integer;    { degree of a vertex in graph }
        capacity: integer;  { the capacity of an edge }
        count: integer;     { output formatting variable }
        NoGVerts: integer;  { number of vertices in the graph }
        k: integer;         { loop control }

    begin
    readln(infile);  { skip over graph name }
    readln(infile, NoGVerts);
    Network.NoVerts := (2 * NoGVerts) + 2;  { # of verts in network, }
                                            { including s and t }
    for i := 0 to Network.NoVerts do
        Network.AdjList[i] := nil;
    capacity := 1; { this program is for simple graphs only }

    { read in the input graph and set up the balanced network }
    Network.SeqTot := 0;
    writeln('The input graph was as follows:');
    writeln('Vertex    Adjacent to', ' ' : 24, 'Degree in Subgraph');
```

```
    while not eof(infile) do begin
        read(infile, i);  { skip over i in input file }
        i := -i;  { vertex is always negative value }
        write(i : 4, ' ' : 4);
        count := MaxDegree; { for output formatting }
        read(infile, j);
        while (j > 0) do begin
            write(j : 4);
            count := count - 1;
            if (i < j) then  { don't duplicate pair of comp edges }
                MakeEdge(i, j, capacity, Network);
            read(infile, j);
            end; { while }
        degree := -j;  { last value read was minus degree in subgraph }
        Network.SeqTot := Network.SeqTot + degree;
        for k := count downto 1 do
            write(' ' : 4);
        write(degree : 6);
        MakeDegEdge(i, degree, Network);
        readln(infile);
        writeln;
        end; { while }
    end; { Procedure CreateNetworkGGFormat }

{*************************************************************************}
{*                                                                       *}
{*                          Function MaxBalFlow                          *}
{*                                                                       *}
{*************************************************************************}
{*                                                                       *}
{* PURPOSE    : To find a maximum balanced flow in a balanced network.  *}
{* PASSED TO  :                                                          *}
{*    Network - network in which a max balanced flow is to be found.     *}
{* PASSED BACK:                                                          *}
{*    Network - the updated network, which now contains a maximum        *}
{*       balanced flow.                                                  *}
{*                                                                       *}
{*************************************************************************}

function MaxBalFlow (var Network: NetworkType): integer;

    var
        TotFlow: integer;  { total flow in the network }
        TempFlow: integer; { amount by which flow was just increased }
```

```
{*********************************************************************}
{*                                                                   *}
{*                        Function AugmentBalFlow                    *}
{*                                                                   *}
{*********************************************************************}
{*                                                                   *}
{* PURPOSE    : To augment the balanced flow in a balanced network,*}
{*    if possible.                                                   *}
{* PASSED TO  :                                                      *}
{*    Network - the balanced network in which the balanced flow is *}
{*       to be augmented.                                            *}
{* PASSED BACK:                                                      *}
{*    Network - the updated balanced network, which may now contain*}
{*       an increased balanced flow.                                 *}
{*                                                                   *}
{*********************************************************************}
function AugmentBalFlow (var Network: NetworkType): integer;

    const
        MaxQueue = MaxGraphVert; { capacity of queue }

    type
        QueueType = record
                data: array[0..MaxQueue] of NetVertexType;
                { the vertices in the queue }
                front, back: integer;
                        { ptrs to front and back of queue }
            end; { record }
        EdgeType = record      { type for an edge in the network }
                head, tail: NetVertexType; { endpoints of edge }
            end; { record }
        SwitchEdgeType = record
                e: EdgeType;  { endpoints of the switch edge }
                node: AdjPtrType;
                    { ptr to adj node for e or complement(e) }
            end; { record }

    var
        ScanQ: QueueType; { queue of vertices to be searched }
        SwitchEdge: array[NetVertexType] of SwitchEdgeType;
            { edge that allowed us to search vj }
        LongBase: array[GraphVertexType] of NetVertexType;
            { base of blossom containing a vertex }
        sReach: array[NetVertexType] of boolean;
            { is there a valid unsaturated path to a vertex ? }
        Depth: array[GraphVertexType] of NetVertexType;
            { depth of xi and yi in their respective trees }

        ui, vj: NetVertexType; { vertices in the network }
        iBase, jBase: NetVertexType; { blossom bases for ui and vj }
        i, j: GraphVertexType; { vertices in the graph }
        previ: GraphVertexType;{ prevpt of vertex ui }
        ResCap: integer;       { the residual capacity on an edge }
        augmented: boolean;    { was the flow in N just increased ? }
        curr: AdjPtrType;      { ptr to node in adjacency list }
```

```
{*****************************************************************}
{*                                                               *}
{*                        Procedure InitQ                        *}
{*                                                               *}
{*****************************************************************}
{*                                                               *}
{* PURPOSE    : To initialise a queue for use.                   *}
{* PASSED TO  :                                                  *}
{*    Queue - the queue to be initialised.                       *}
{* PASSED BACK:                                                  *}
{*    Queue - the initialised queue.                             *}
{*                                                               *}
{*****************************************************************}
 procedure InitQ (var Queue: QueueType);

     begin
     Queue.front := 0;
     Queue.back := 0;
     end; { Procedure InitQ }

{*****************************************************************}
{*                                                               *}
{*                       Function EmptyQ                         *}
{*                                                               *}
{*****************************************************************}
{*                                                               *}
{* PURPOSE    : To determine whether or not the queue is empty   *}
{* PASSED TO  :                                                  *}
{*    Queue - the queue in question.                             *}
{* PASSED BACK:                                                  *}
{*    a boolean value indicating whether or not the queue is     *}
{*    empty.                                                     *}
{*                                                               *}
{*****************************************************************}
 function EmptyQ (var Queue: QueueType): boolean;

     begin
     if Queue.front = Queue.back then
          EmptyQ := true
     else
          EmptyQ := false;
     end; { Function EmptyQ }

{*****************************************************************}
{*                                                               *}
{*                          Procedure EnQ                        *}
{*                                                               *}
{*****************************************************************}
{*                                                               *}
{* PURPOSE    : To put a vertex at the end of the queue.         *}
{* PASSED TO  :                                                  *}
{*    ui - The vertex to be placed into the queue.               *}
{* PASSED BACK:                                                  *}
{*    Queue - the updated queue.                                 *}
{*                                                               *}
{*****************************************************************}
 procedure EnQ (ui: NetVertexType;
                                 var Queue: QueueType);
```

```
     begin
     if ((Queue.back + 1) mod MaxQueue) <> Queue.front then begin
          Queue.data[Queue.back] := ui;
          Queue.back := (Queue.back + 1) mod MaxQueue;
          end { then }
     else
          writeln('*** ERROR ***  queue overflow.');
     end; { Procedure EnQ }

{*****************************************************************}
{*                                                               *}
{*                          Function DeQ                         *}
{*                                                               *}
{*****************************************************************}
{*                                                               *}
{* PURPOSE    : To remove and return the vertex at the front of*}
{*    the queue.                                                 *}
{* PASSED TO  :                                                  *}
{*    Queue - the queue from which the vertex is to be removed.*}
{* PASSED BACK:                                                  *}
{*    Queue - the updated queue.                                 *}
{*                                                               *}
{*****************************************************************}
 function DeQ (var Queue: QueueType): NetVertexType;

     begin
     if not EmptyQ(Queue) then begin
          DeQ := Queue.data[Queue.front];
          Queue.front := (Queue.front + 1) mod MaxQueue;
          end { then }
     else
          writeln('*** ERROR ***  queue underflow');
     end; { Function DeQ }

{*****************************************************************}
{*                                                               *}
{*                      Function FindResCap                      *}
{*                                                               *}
{*****************************************************************}
{*                                                               *}
{* PURPOSE    : To find the residual capacity on an edge.       *}
{* PASSED TO  :                                                  *}
{*    AdjPtr - ptr to the adjacency list node for the edge.      *}
{* PASSED BACK:                                                  *}
{*    The value of the residual capacity of the edge.            *}
{*                                                               *}
{*****************************************************************}
 function FindResCap (AdjPtr: AdjPtrType): integer;

     begin
     if (AdjPtr^.dir = OutEdge) then
          FindResCap := AdjPtr^.cap - AdjPtr^.flow^
     else
          FindResCap := AdjPtr^.flow^;
     end; { Function FindResCap }
```

```
{*****************************************************************}
{*                                                               *}
{*                        Function FindBase                      *}
{*                                                               *}
{*****************************************************************}
{*                                                               *}
{* PURPOSE    : To find the highest vertex in the lower tree    *}
{*    that can be reached from a particular vertex.              *}
{* PASSED TO  :                                                  *}
{*    i - the vertex in question.                                *}
{* PASSED BACK:                                                  *}
{*    LongBase - updated globally.                               *}
{*                                                               *}
{*****************************************************************}
 function FindBase (i: GraphVertexType): NetVertextype;

     var
          vj: NetVertexType;  { the LongBase for graph vertex i }
          j: GraphVertexType; { vertex in G corresponding to vj }

     begin
     vj := LongBase[i];
     if vj <> -1 then begin
          j := MapNtoG(vj);
          if i <> j then begin
               vj := FindBase(j);
               LongBase[i] := vj; { path compression }
               end; { if }
          end; { if }
     FindBase := vj;
     end; { Function FindBase }

{*****************************************************************}
{*                                                               *}
{*                       Procedure PullFlow                      *}
{*                                                               *}
{*****************************************************************}
{*                                                               *}
{* PURPOSE    : To put flow on the edges that lie on the path   *}
{*    from one vertex to another (i.e.to pull flow from u to     *}
{*    v).                                                        *}
{* PASSED TO  :                                                  *}
{*    u,v - the vertices in question. We want to pull flow down*}
{*        from u to v along the unsaturated path found by the   *}
{*        algorithm.                                             *}
{*    amount - the amount of flow that is to be put onto those  *}
{*        edges.                                                 *}
{* PASSED BACK:                                                  *}
{*    Network - the updated network (implicitly).                *}
{*                                                               *}
{*****************************************************************}
 procedure PullFlow (ui, vj: NetVertexType;
                                    amount: integer);

     var
          uh: NetVertexType; { head of SwitcEdge for u }
          vh: NetVertexType; { head of SwitcEdge for v }
          i, j: GraphVertexType; { vertices in the graph }
```

```
{************************************************************}
{*                                                          *}
{*                    Procedure Augment                     *}
{*                                                          *}
{************************************************************}
{*                                                          *}
{* PURPOSE    : To augment the flow on an edge by a         *}
{*    specified amount.                                     *}
{* PASSED TO  :                                             *}
{*    AdjPtr - ptr to adjacency node for edge in question,  *}
{*    amount - the amount by which the flow is to be        *}
{*       increased.                                         *}
{* PASSED BACK:                                             *}
{*    the updated network (implicitly).                     *}
{*                                                          *}
{************************************************************}
 procedure Augment (AdjPtr: AdjPtrType;
                                    amount: integer);

      begin
      if AdjPtr^.dir = OutEdge then
           { we followed edge forwards }
           AdjPtr^.flow^ := AdjPtr^.flow^ + 1
      else  { we followed edge backwards }
           AdjPtr^.flow^ := AdjPtr^.flow^ - 1;
      end; { Procedure Augment }

{************************************************************}
{*               Procedure PullFlow Mainline                *}
{************************************************************}
 begin
 if ui <> vj then begin
      vh := SwitchEdge[vj].e.head; { head of SwitchEdge[vj] }
      Augment(SwitchEdge[vj].node, amount);
      PullFlow(ui, SwitchEdge[vj].e.tail, amount);
      PullFlow(ComplementVert(vj),
                   ComplementVert(vh), amount);
      end; { if }
 end; { Procedure PullFlow }
```

```
{*****************************************************************}
{*                                                               *}
{*                    Function ExtendLongBase                    *}
{*                                                               *}
{*****************************************************************}
{*                                                               *}
{* PURPOSE    : To make the changes that allow a valid path to  *}
{*    be found to some vertices in the lower tree that were      *}
{*    hitherto unreachable on such a path.                       *}
{* PASSED TO  :                                                  *}
{*    ui,vj - the endpoints of the edge that allows extension    *}
{*       of search                                               *}
{*    iBase,jBase - the LongBase of i and j respectively,        *}
{*    AdjPtr - ptr to adj node for edge (ui,vj).                 *}
{* PASSED BACK:                                                  *}
{*    the updated network (implicitly).                          *}
{*                                                               *}
{*****************************************************************}
 function ExtendLongBase (ui, vj, iBase, jBase: NetVertexType;
                          AdjPtr: AdjPtrType): NetVertexType;

      var
           pg, qh: NetVertexType; { temp vertices in the network }
           g, h: GraphVertexType; { temp vertices in G }
      { corresponding to pg and qh in N respectively }
           wk: NetVertexType;
                { common base for i and j (in lower tree) }

      begin
      { find common base }
      pg := iBase;
      qh := jBase;
      while pg <> qh do begin
           g := MapNtoG(pg);
           h := MapNtoG(qh);
           if Depth[g] > Depth[h] then
                pg := FindBase(MapNtoG(
                     SwitchEdge[ComplementVert(pg)].e.tail))
           else
                qh := FindBase(MapNtoG(
                     SwitchEdge[ComplementVert(qh)].e.tail));
           end; { while }
      wk := pg;
```

```
{ deal with bases from ui back to common base }
pg := iBase;
while pg <> wk do begin
     g := MapNtoG(pg);
     LongBase[g] := wk;
     if not sReach[pg] then begin
          SwitchEdge[pg].e.tail := ComplementVert(vj);
          SwitchEdge[pg].e.head := ComplementVert(ui);
          SwitchEdge[pg].node := AdjPtr;
          sReach[pg] := true;
               { also sets tReach for comp. vertex }
          EnQ(pg, ScanQ);
          end; { if }
     pg := FindBase(MapNtoG(
          SwitchEdge[ComplementVert(pg)].e.tail));
     end; { while }
{ deal with bases from vj back to common base }
qh := jBase;
while qh <> wk do begin
     h := MapNtoG(qh);
     LongBase[h] := wk;
     if not sReach[qh] then begin
          SwitchEdge[qh].e.tail := ui;
          SwitchEdge[qh].e.head := vj;
          SwitchEdge[qh].node := AdjPtr;
          sReach[qh] := true;
               { also sets tReach for comp. vertex }
          EnQ(qh, ScanQ);
          end; { if }
     qh := FindBase(MapNtoG(
          SwitchEdge[ComplementVert(qh)].e.tail));
     end; { while }
{ deal with common base - no need to set LongBase }
if not sReach[wk] then begin
     SwitchEdge[wk].e.tail := ui;
     SwitchEdge[wk].e.head := vj;
     SwitchEdge[wk].node := AdjPtr;
     sReach[wk] := true;
          { also sets tReach for complementary vertex }
     EnQ(wk, ScanQ);
     end; { if }
ExtendLongBase := wk;
end; { ExtendLongBase }
```

```
{*****************************************************************}
{*                Function AugmentBalFlow Mainline               *}
{*****************************************************************}
 begin
 for ui := 0 to Network.NoVerts do
     sReach[ui] := false;
 for i := 0 to ((Network.NoVerts - 2) div 2) do
     LongBase[i] := -1;  { an invalid value }
 InitQ(ScanQ);
 SwitchEdge[s].e.tail := s;  { really just a flag }
 SwitchEdge[s].e.head := t;
 Depth[s] := 0;
 sReach[s] := true;
 LongBase[0] := 0;
 EnQ(s, ScanQ);
 augmented := false;
```

```
repeat
     ui := DeQ(ScanQ);
     i := MapNtoG(ui);
     iBase := FindBase(i);  { may change dynamically }
     curr := Network.AdjList[ui];
     while (curr <> nil) and not augmented do begin
          vj := curr^.vert;
          ResCap := FindResCap(curr);
          if ResCap > 0 then begin
               j := MapNtoG(vj);
               jBase := FindBase(j);
               if jBase = -1 then begin { vj not in either tree }
                    Depth[j] := Depth[i] + 1;
                    EnQ(vj, ScanQ);
                    sReach[vj] := true;
                    LongBase[j] := ComplementVert(vj);
                    SwitchEdge[vj].e.tail := ui;
                    SwitchEdge[vj].e.head := vj;
                    SwitchEdge[vj].node := curr;
                    end { if }
               else if (ibase <> jBase) and
                      (sReach[ComplementVert(vj)]) then begin
                    { edge is from an s-reachable to a }
                    { t-reachable vertex and allows us to go }
                    { farther up bottom tree }
                    previ := MapNtoG(
                         SwitchEdge[ComplementVert(ui)].e.tail);
                    if (iBase <> ui) or (previ <> j)
                                  or (ResCap > 1) then
                         begin
                         { not extending an invalid unsat path }
                         iBase :=
                              ExtendLongBase
                                   (ui, vj, iBase, jBase, curr);
                         if iBase = t then begin
                              { augmenting path found}
                              PullFlow(s, t, ResCap);
                              augmented := true;
                              end; { if }
                         end; { if }
                    end; { if }
               end; { if }
          curr := curr^.next;
          end; { while }
until EmptyQ(ScanQ) or augmented;
if augmented then
     AugmentBalFlow := 2 { 2•1, simple graphs only }
else
     AugmentBalFlow := 0;
end; { Function AugmentBalFlow }
```

```
    {*********************************************************************}
    {*                    Function MaxBalFlow Mainline                   *}
    {*********************************************************************}
    begin
    TotFlow := 0;
    repeat
         TempFlow := AugmentBalFlow(Network);
         TotFlow := TotFlow + TempFlow;
    until TempFlow = 0;
    MaxBalFlow := TotFlow;
    end; { Function MaxBalFlow }

{************************************************************************}
{*                                                                      *}
{*                          M A I N L I N E                             *}
{*                                                                      *}
{************************************************************************}
begin
showtext;
reset(infile, 'data3.dodec.3reg');
CreateNetworkGGFormat(infile, Network);
writeln;
flow := MaxBalFlow(Network);
writeln;
writeln('Flow was increased by', flow : 3, ' units.');
writeln;
writeln('The subgraph found by the algorithm is:');
PrintSubGraph(Network);

writeln;
writeln;
writeln('Processing successfully completed.');
end. { program MaxFlow }
```

References

[1] Bondy, J. A. and Murty, U. S. R., (1976). Graph Theory with Applications. North Holland.

[2] Aho, Alfred V., Hopcroft, John E., Ullman, Jeffrey D. (1983). Data Structures and Algorithms. Addison Wesley.

[3] Micali, Silvio and Vazirani, Vijay V. (1980). An $O(\sqrt{|V|} \cdot |E|)$ Algorithm for Finding Maximum Matchings in General Graphs. 21st Annual Symposium on Foundation Computer Science, IEEE.

[4] Erdos, P. and Gallai, T. (1960). Graphs with Prescribed Degrees of Vertices (Hungarian). Mat Lapok 11, 264–74.

[5] Papadimitriou, Christos H. and Steiglitz, Kenneth (1982). Combinatorial Optimization. Prentice Hall.

[6] Lovasz, L. and Plummer, M. D. (1986). Matching Theory. North Holland.

[7] Selected Papers of W. T. Tutte, Volume 1 (1979). The Charles Babbage Research Centre, St. Pierre, Manitoba.

[8] Gabow, Harold N. and Tarjan, Robert Endre (1984). A Linear–Time Algorithm for a Special Case of Disjoint Set Union. Journal of Computer and System Sciences 30, 209–221.

[9] Think's Lightspeed Pascal, Version 3.0.2 (1986). Think Technologies Inc, 135 South Road, Bedford, Massachusetts, U.S.A.