EMPTY-SHAPE TRIANGULATION ALGORITHMS

By

TIMOTHY LAMBERT

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

Department of Computer Science
University of Manitoba
Winnipeg, Manitoba

ISBN 0-612-13278-1

Name ___

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

COMPUTER SCIENCE

SUBJECT TERM

0984

SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523
Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language
- General 0679
- Ancient 0289
- Linguistics 0290
- Modern 0291

Literature
- General 0401
- Classical 0294
- Comparative 0295
- Medieval 0297
- Modern 0298
- African 0316
- American 0591
- Asian 0305
- Canadian (English) 0352
- Canadian (French) 0355
- English 0593
- Germanic 0311
- Latin American 0312
- Middle Eastern 0315
- Romance 0313
- Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion
- General 0318
- Biblical Studies 0321
- Clergy 0319
- History of 0320
- Philosophy of 0322

Theology 0469

SOCIAL SCIENCES

American Studies 0323
Anthropology
- Archaeology 0324
- Cultural 0326
- Physical 0327

Business Administration
- General 0310
- Accounting 0272
- Banking 0770
- Management 0454
- Marketing 0338

Canadian Studies 0385
Economics
- General 0501
- Agricultural 0503
- Commerce-Business 0505
- Finance 0508
- History 0509
- Labor 0510
- Theory 0511

Folklore 0358
Geography 0366
Gerontology 0351
History
- General 0578
- Ancient 0579
- Medieval 0581
- Modern 0582
- Black 0328
- African 0331
- Asia, Australia and Oceania 0332
- Canadian 0334
- European 0335
- Latin American 0336
- Middle Eastern 0333
- United States 0337

History of Science 0585
Law 0398
Political Science
- General 0615
- International Law and Relations 0616
- Public Administration 0617

Recreation 0814
Social Work 0452
Sociology
- General 0626
- Criminology and Penology ... 0627
- Demography 0938
- Ethnic and Racial Studies 0631
- Individual and Family Studies 0628
- Industrial and Labor Relations 0629
- Public and Social Welfare 0630
- Social Structure and Development 0700
- Theory and Methods 0344

Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture
- General 0473
- Agronomy 0285
- Animal Culture and Nutrition 0475
- Animal Pathology 0476
- Food Science and Technology 0359
- Forestry and Wildlife 0478
- Plant Culture 0479
- Plant Pathology 0480
- Plant Physiology 0817
- Range Management 0777
- Wood Technology 0746

Biology
- General 0306
- Anatomy 0287
- Biostatistics 0308
- Botany 0309
- Cell 0379
- Ecology 0329
- Entomology 0353
- Genetics 0369
- Limnology 0793
- Microbiology 0410
- Molecular 0307
- Neuroscience 0317
- Oceanography 0416
- Physiology 0433
- Radiation 0821
- Veterinary Science 0778
- Zoology 0472

Biophysics
- General 0786
- Medical 0760

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996
Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences
- General 0566
- Audiology 0300
- Chemotherapy 0992
- Dentistry 0567
- Education 0350
- Hospital Management 0769
- Human Development 0758
- Immunology 0982
- Medicine and Surgery 0564
- Mental Health 0347
- Nursing 0569
- Nutrition 0570
- Obstetrics and Gynecology .. 0380
- Occupational Health and Therapy 0354
- Ophthalmology 0381
- Pathology 0571
- Pharmacology 0419
- Pharmacy 0572
- Physical Therapy 0382
- Public Health 0573
- Radiology 0574
- Recreation 0575
- Speech Pathology 0460
- Toxicology 0383

Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences

Chemistry
- General 0485
- Agricultural 0749
- Analytical 0486
- Biochemistry 0487
- Inorganic 0488
- Nuclear 0738
- Organic 0490
- Pharmaceutical 0491
- Physical 0494
- Polymer 0495
- Radiation 0754

Mathematics 0405
Physics
- General 0605
- Acoustics 0986
- Astronomy and Astrophysics 0606
- Atmospheric Science 0608
- Atomic 0748
- Electronics and Electricity 0607
- Elementary Particles and High Energy 0798
- Fluid and Plasma 0759
- Molecular 0609
- Nuclear 0610
- Optics 0752
- Radiation 0756
- Solid State 0611

Statistics 0463

Applied Sciences

Applied Mechanics 0346
Computer Science 0984
Engineering
- General 0537
- Aerospace 0538
- Agricultural 0539
- Automotive 0540
- Biomedical 0541
- Chemical 0542
- Civil 0543
- Electronics and Electrical 0544
- Heat and Thermodynamics ... 0348
- Hydraulic 0545
- Industrial 0546
- Marine 0547
- Materials Science 0794
- Mechanical 0548
- Metallurgy 0743
- Mining 0551
- Nuclear 0552
- Packaging 0549
- Petroleum 0765
- Sanitary and Municipal 0554
- System Science 0790

Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

EMPTY-SHAPE TRIANGULATION ALGORITHMS

BY

TIMOTHY LAMBERT

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

DOCTOR OF PHILOSOPHY

Abstract

The Delaunay triangulation of a set of sites (points in the plane) can be defined as the triangulation with the property that the circumcircle of each triangle is empty (contains no site). I generalize this to define *empty-shape triangulations*. An empty-shape triangulation is defined by a set of shapes with the property that any triangle has a unique circumscribing shape. The Delaunay triangulation is the empty-shape triangulation where the shapes consist of the set of all circles.

In this thesis I develop a taxonomy for triangulation algorithms, describe and implement a plane sweep algorithm for empty-shape triangulations, describe algorithms for constrained empty-shape triangulations and an algorithm for higher-dimensional empty-shape triangulations. I implement an algorithm for computing convex-distance-function Delaunay triangulations by extending them to empty-shape triangulations and then extracting the appropriate subtriangulation.

Two properties of the Delaunay triangulation are necessary for the correctness of the known efficient algorithms. I prove that the only triangulations with these properties are empty-shape triangulations.

I analyze, implement and measure the performance of Delaunay triangulation algorithms on random convex polygons.

There is no generally accepted definition of what a random convex polygon is. I give several operational definitions, design efficient algorithms and implement some of them.

Acknowledgements

I thank Bill Hoskins and Dereck Meek for their supervision and useful feedback without which this thesis would never have been completed.

I thank Judy Goldsmith for prodding me to get the thesis into shape.

Thanks to Robert Thomas for his careful reading of this manuscript.

Thanks to David Kirkpatrick and Barry Schaudt for useful discussions. (And apologies to Barry for finding a hole in his algorithm.)

And special thanks to all my friends who never doubted that I would one day finish my thesis.

And extra special thanks to Donald Knuth for TEX.

Contents

List of Tables

List of Figures

Imagine a vast sheet of paper on which straight Lines, Triangles, Squares, Pentagons, Hexagons, and other figures, instead of remaining fixed in their places, move freely about, on or in the surface, but without the power of rising above or sinking below it, very much like shadows—only hard with luminous edges—and you will then have a pretty correct notion of my country and countrymen...

Our Women are Straight Lines.

Our Soldiers and Lowest Classes of Workmen are Triangles with two equal sides, each about eleven inches long, and a base or third side so short (often not exceeding half an inch) that they form at their vertices a very sharp and formidable angle. Indeed when their bases are of the most degraded type (not more than the eighth part of an inch in size), they can hardly be distinguished from Straight Lines or Women; so extremely pointed are their vertices.

...a wise ordinance of Nature has decreed that, in proportion as the working-classes increase in intelligence, knowledge and all virtue, in that same proportion their acute angle (which makes them physically terrible) shall increase also and approximate to the comparatively harmless angle of an Equilateral Triangle. Thus, in the most brutal and formidable of the soldier class—creatures almost on a level with women in their lack of intelligence—it is found that, as they wax in the mental ability necessary to employ their tremendous penetrating power to advantage, so they wane in the power of penetration itself.

Edwin A. Abbott *Flatland* [1]

Chapter 1

Introduction

A set of sites (points) in the plane can be connected into a network of triangles by joining pairs of sites with line segments so that no segments cross and no more segments can be added. There are many applications of triangulations:

- A surveyor measures the height of the ground at a set of a sites, and then wants to find a surface interpolating those sites (for example, to construct a contour map). Triangulating the sites gives a polyhedral surface [329].
- A mechanical engineer who wishes to analyze a mechanical component can divide it up into finite elements and solve the resulting equations. Triangular elements are a popular choice [341].
- If the sites represent an image, we might wish to cluster the sites into groups that are close together. Triangulating the sites connects each site to others that are close to it [86].
- If the sites represent post offices, we might wish to find the post office closest to a given query location (this is called the post office problem [185]). For each post office, the set of sites that it is closest to is known as its Voronoi polygon. The set of all Voronoi polygons is called the Voronoi diagram [318]. The Voronoi diagram is dual to the Delaunay triangulation (see figure 1.1) and can be constructed from it in time $O(n)$. To solve the post-office problem it is only necessary to locate the Voronoi polygon containing the query point [295].

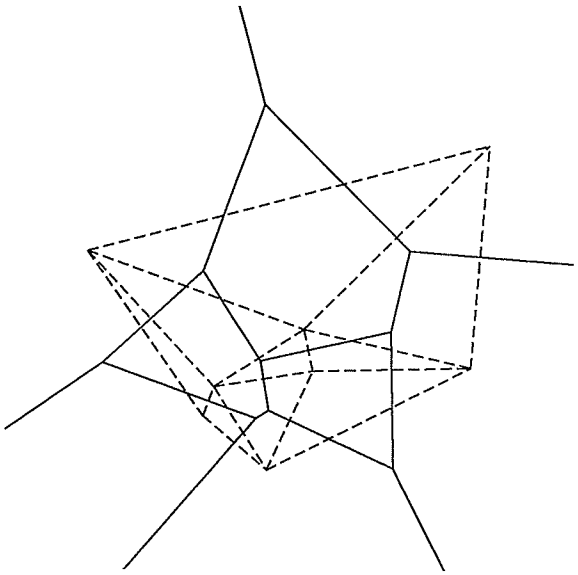


Figure 1.1: Voronoi diagram (solid) and Delaunay triangulation (dotted)

- If the sites represent cities that we wish to connect in a power grid by joining cities with straight power lines, then the grid with the shortest total line length is the Euclidean minimum spanning tree of the sites. Using a general minimum-spanning-tree algorithm to compute this requires examining $O(n^2)$ edges. The Euclidean minimum spanning tree is a subgraph of the Delaunay triangulation [78] (see figure 1.1) and can be constructed from it in time $O(n)$ [263].

- Other applications of triangulations include such fields as robotics [245], pattern recognition [311], surface fitting [19], computer-aided geometric design [289], geography [255], geology [327], architecture [233], medicine [20], computer graphics [10], forestry [324], VLSI [208], remote sensing [81], image processing [143], computer vision [114], and hydrology [167]. Aurenhammer [15], Bern and Eppstein [25], Okabe *et al.* [248], and De Floriani [74] survey many of these applications.

The most commonly constructed triangulation is the Delaunay triangulation. There are two reasons for this:

1. there are fast ($O(n \log n)$) algorithms for its construction:
   - divide and conquer [202],
   - randomized incremental [148], or
   - plane sweep [122].
2. it has useful geometric properties:
   - Amongst all triangulations, it optimizes various triangulation measures. These include
     - maximizing the minimum angle [298],
     - minimizing the maximum circumscribed circle [71],
     - minimizing the maximum smallest enclosing circle[1] [71, 266],
     - minimizing the integral of the gradient squared [261, 272], and
     - maximizing the mean inradius (proved in appendix A).
   - The circumcircle of each triangle contains no other site [78].
   - It contains as subgraphs the convex hull, the Euclidean minimum spanning tree, the Gabriel graph [130], and the relative neighbourhood graph [312] of the sites.
   - The length of the shortest path between two sites along edges of the Delaunay triangulation is within a constant factor of the straight-line distance [53, 90, 174].
   - If we define a relation on the triangles by their order along rays from a given site, then that relation is a partial order [75, 95].

   The fact the Delaunay triangulation optimizes so many measures has led some authors astray in claiming that the Delaunay triangulation minimizes the sum of the minimum triangle angles [206, 257, 260, 294], that it minimizes the standard deviation of the triangle angles [329], and that it minimizes the total edge length [89, 295]. (Counterexamples to these claims can be found in appendix B.)

[1] The smallest enclosing circle differs from the circumscribing circle when the triangle is obtuse.

1.1 Triangulation Algorithm Taxonomy

There have been about one hundred papers published describing triangulation algorithms (see table 2.2 on 24). The surveys by Aurenhammer [15], De Floriani [74], Fortune [120] and Okabe *et al.* [248] list some of these, but the only attempt at classification is made by De Floriani who divides them into "static" (off-line) and "dynamic" (on-line) algorithms.

In chapter 2 I provide a taxonomy for triangulation algorithms. I hope that my taxonomy will prove useful to programmers faced with choosing from a hundred alternatives for a triangulation algorithm by helping them realize that there are really only a half dozen choices; to researchers by discouraging them from publishing a microscopic variation on a previous method and encouraging them to find a new algorithm; and to anyone who wants to understand triangulation algorithms by helping them to apply their intuition about sorting algorithms to this problem.

The one-dimensional analogue of triangulation is sorting; so I have adapted Knuth's taxonomy for sorting algorithms [185]. I classify triangulation algorithms into Incremental (*e.g.* [145]), Selection (*e.g.* [229]), Flip (*e.g.* [19]) and Divide-and-Conquer algorithms. Just as with sort algorithms, straightforward algorithms of the first three types have worst-case complexity of $O(n^2)$, cleverer algorithms can improve on this (for example Fortune's sweepline algorithm [122] is a triangulation by selection algorithm with worst-case complexity $O(n \log n)$), the Divide-and-Conquer algorithm [202] has worst-case complexity $O(n \log n)$, and the randomized incremental algorithm of Guibas *et al.* [148] has average (taken over all insertion orders) complexity $O(n \log n)$).

1.2 Locally Optimized Triangulations

A locally optimized triangulation can be defined by a flip rule that determines which diagonal of a convex quadrilateral should be included in the triangulation. The Flip algorithm repeatedly applies the flip rule to adjacent triangles in the triangulation until there are no flippable edges left.

The rule is generally chosen to select the "better" of the two triangulations of the quadrilateral. About twenty different such rules have been published, including:

- maximize the minimum angle in both triangles [194] (this leads to the Delaunay triangulation [298]),

- minimize the maximum angle [19],
- select the shorter diagonal [233],
- maximize the sum of the minimum angles [43, 294],
- maximize the minimum altitude [138, 332], and
- minimize the maximum inradius [288].

For each flip rule we can also consider a globally optimized triangulation. For the maximize-minimum angle flip rule this is just the Delaunay triangulation. For the shorter-diagonal rule this is the triangulation with minimum total edge length, this is commonly called the MWT (Minimum Weight triangulation). No polynomial-time algorithm is known for the MWT. (It is one of the remaining open problems from Garey and Johnson [133].) Polynomial algorithms have been published for only a few globally optimized triangulations (other than those that are the Delaunay triangulation):

- the minimize-maximum-angle rule [103] ($O(n^2 \log n)$),
- the minimize-maximum-edge-length rule [100] ($O(n^2)$),
- the maximize-minimum-triangle-height rule [24] ($O(n^2 \log n)$), and
- the minimize-maximum-eccentricity rule [24] (eccentricity is distance from circumcentre to triangle) ($O(n^3)$).

If we use the Delaunay flip rule, the Flip algorithm terminates with the Delaunay triangulation [194] using at most $O(n^2)$ flips [102]. If the flips are done in the right order then on average only $O(n)$ flips are required [148]. Do fast algorithms exist for the triangulations defined by other flip rules?

Two key properties of the Delaunay triangulation are needed to prove the correctness of the algorithms described in chapter 2:

- A Systematic property—there is a unique triangulation.
- A Local property—when a site is added to the triangulation, the only new edges are those adjacent to the new site.

Nielson and Franke [243] claim that the min-max angle rule has the systematic property. It is easy to construct a five site counterexample,[2] but it would be tedious to do so for other flip rules for which counter-examples exist, so in chapter 3 I generalized all the dozen or so flip rules that I had seen published to get 120 different flip rules and tested each of these against random convex pentagons to see if I could find counterexamples to the two properties. I found counterexamples to all of them, except those that were equivalent to the Delaunay triangulation. This led me to search for the proof described in the next paragraph.

I prove that the only systematic local flip rule that is invariant under rotations and translations of the quadrilateral is the Delaunay rule (section 3.6.3).

I prove that the only systematic local flip rules invariant under scaling and translation correspond to generalizations of convex-distance-function Delaunay triangulations which I call *empty-shape triangulations* (section 3.6.4).

This suggests that algorithms as fast as Delaunay triangulation algorithms for triangulations other than empty-shape triangulations are unlikely to be found. However, for most site sets, these other triangulations do not differ by much from the Delaunay triangulations, so computing a Delaunay triangulations and then applying the flip algorithm should work well in practice.

1.3 Convex-Distance-Function Delaunay triangulations

The Voronoi diagram and Delaunay triangulation can be defined for non-Euclidean metrics. Algorithms to compute the Voronoi diagram have been published for the l_1 metric by Hwang [158], for the l_1 and l_∞ metrics by Lee and Wong [203], for the l_p metric by Lee [200], for metrics where paths are limited to a fixed number of orientations by Widmayer *et al.* [333] and for a metric where paths can only go in a certain connected range of directions by Chang *et al.* [45]. These metrics are all special cases of Minkowski convex distance functions, where the ball ("unit circle") can be any convex shape. Chew and Drysdale [55] give a Voronoi diagram algorithm for convex distance functions. The corresponding Delaunay triangulation can be easily computed from the Voronoi diagram, but it is simpler to calculate it directly. Drysdale [91] implements a Divide-and-Conquer algorithm for convex-distance-function Delaunay triangulation.

[2] Nielson gives a six point counter-example in a note correcting this [242].

The above papers and the book by Okabe, Boots and Sugihara [248] give many applications for these Voronoi diagrams and Delaunay triangulations including defining response areas for emergency units in urban areas, scheduling head movement in a two-dimensional secondary storage system, analyzing market areas, finding minimum spanning trees, Steiner trees and nearest neighbours in these metrics, finding largest empty homothetic convex shapes, testing polygon containment, and planning robot motion.

The algorithms[3] described in chapter 2 can be implemented so that they use only two geometric tests:

- Is a point inside the circle passing through three other points?
- Is a point to the left of a directed line?

If the ball for the distance function is smooth (no corners) and strictly convex (no flat spots) then for any three points there is a unique circumball [187]. If we replace "circle" in the first test above by "ball" the algorithm will work for this distance function with essentially the same proof of correctness.

If this is not the case (for example, the Manhattan metric), the outer face of the triangulation may have concavities, and the algorithms will break down. It is possible to fix up the algorithms—this is what Drysdale does for the Divide-and-Conquer algorithm, but has only been done in the metrics l_1 and l_∞ for a sweepline algorithm [297] and Drysdale lists the implementation of a general sweepline algorithm as an open problem.

In chapter 4 I show that a simpler approach, using empty-shape triangulations (which generalize convex-distance-function Delaunay triangulations), is to fix the distance function and round off the corners at an infinitesimal scale. This produces a supertriangulation from which the desired triangulation can be easily extracted, leading to the first known Flip, Selection and Sweepline algorithms for convex-distance-function Delaunay triangulations (section 4.1.2).

I present a complete working implementation of my new sweepline algorithm for empty-shape triangulations and convex-distance-function Delaunay triangulations, and implementation of the geometric primitives required for the other algorithms (section 4.1.1).

I give examples showing that algorithms for higher-dimensional Delaunay triangulation

[3] with the sole exception of the sweepline algorithm

do not work in general for higher-dimensional convex-distance-function Delaunay triangulation (section 4.2.1). I give a Selection algorithm for higher-dimensional convex-distance-function Delaunay triangulation (section 4.2.2).

1.4 Constrained Delaunay triangulation

A constrained triangulation is one where certain edges are forced (for example, triangulating a simple polygon). In a constrained Delaunay triangulation, sites can occur in the circumcircle of a triangle if they are hidden from a triangle vertex by a constraint edge.

Algorithms have been published for constrained Delaunay triangulation, using Incremental (*e.g.* [63]), Selection (*e.g.* [216]), Flip (*e.g.* [33]), Divide-and-Conquer (*e.g.* [57]) and Sweepline algorithms (*e.g.* [291]).

Applications of constrained Delaunay triangulations include constructing finite-element meshes for polygonal shapes [164], dividing polygons into triangles while avoiding small angles [54], finding shortest paths that avoid line obstacles [56], finding the greedy triangulation (this is formed by adding edges that do not intersect previously added edges in order from shortest to longest) [139, 211] and fitting a surface to a scattered set of sites and line segments [76].

Using the ideas in the previous section, I can create algorithms for constrained Delaunay triangulation using arbitrary convex distance functions, a problem not previously considered in the literature (section 4.1.4).

1.5 Delaunay Triangulation of Convex Polygons

The worst-case lower bound for constructing the Delaunay triangulation is $\Omega(n \log n)$ in[4] the real-RAM model [263]. If the sites to be triangulated form a convex polygon this lower bound does not apply and Aggarwal *et al.* have found an $O(n)$ worst-case algorithm [5]. Unfortunately, it is too complicated to be practical. Devijver and Maybank [83] present a very simple algorithm that is $O(n^3)$ in the worst case and $O(n^2)$ if we take the average over all possible triangulations of the polygon. Chew [52] gives a randomized incremental algorithm that is $O(n)$ if we take the average over all insertion orders.

[4] $\Omega\big(g(n)\big)$ is the set of functions $f(n)$ such that $|f(n)| \geq C|g(n)|$ for some $C > 0$ [144].

In chapter 5 I calculate the average time complexity of general Delaunay triangulation algorithms over all possible triangulations of the polygon. This is $O(n)$ for all types of algorithm, except for the simple selection algorithm, which is $O(n^{3/2})$ and the sweepline algorithm, which is still $O(n \log n)$.

Surprisingly, when tested on random convex polygons generated by the methods described in chapter 6, each of the algorithms exhibited worst-case performance, rather than the performance expected from assuming that all triangulations were equally likely. For example, the incremental algorithm took time $O(n^2)$. To improve this to $O(n)$ it is necessary to insert the sites in a random order. A similar randomization is required to make Divide-and-Conquer take time $O(n)$.

I implemented both of these randomizations and the resulting algorithms ran in the expected time on my test polygons.

1.6 Generating Random Convex Polygons

To test the analysis described in the preceding section, it is necessary to be able to generate random convex n-gons. Unfortunately, there is no accepted definition of what a random convex polygon is. For example, Sylvester's problem [277] is to find the probability that the convex hull of four random sites is a quadrilateral. Even for sites drawn from the uniform distribution, this turns out to depend on the shape of the region from which they are drawn.

Random convex polygons have been generated on the computer by Crain [67], who used Voronoi polygons defined by a Poisson point process, by Crain and Miles [68], who examined polygons defined by a Poisson line process, by Devroye [84], De Pano *et al.* [80] and Abrahamson [2], who took the convex hull of random points, and by May and Smith [226], who took the intersection of random half-spaces. However, none of these methods let you specify the number of sides of the polygon.

In chapter 6 I describe efficient algorithms for each of the following methods:

- Pick n points from some distribution. Reject if their convex hull is not an n-gon. (We can generate a convex n-gon in time $O(n \log n)$ using this method.)

- Select points from some distribution until their convex hull has n vertices. (This takes time $O(h \log n)$ where h is the number of times times the hull changes, which seems in practice to be proportional to n.)

- The n vectors comprising the sides of the polygon can be regarded as a point in $2n$-dimensional space. For the polygon to close, the vectors must sum to zero. This means that the point must lie on a $2n-2$ dimensional flat; so pick from some distribution on this flat. (This takes time $O(n \log n)$ since it is necessary to sort the vectors to construct the polygon.)

- Start with an arbitrary convex polygon and give each vertex a random velocity. If a vertex is ever about to become concave, we "bounce" it from that constraint. If we perform $O(n)$ bounces the resulting polygon should be "random". (Each bounce will take time $O(\log n)$ since the event queue will have $O(n)$ elements, giving $O(n \log n)$ time in total.)

- Choose a random topological triangulation of a polygon. Use Dillencourt's constructive proof of the realizability of such triangulations as Delaunay triangulations [87] to construct a convex polygon.

- We can take the dual of polygons produced by the above methods. For example, for the first method, this amounts to taking the intersection of half-spaces containing the origin.

I have implemented the first three methods above and used them for testing the performance of the convex polygon triangulation algorithms described in chapter 5.

Some other uses for my random convex polygons might be to determine how often random convex polygons were unimodal [6] and how often the minimum-area and minimum-perimeter-enclosing rectangles are different [80].

1.7 Contributions of this thesis

I develop a taxonomy of Delaunay triangulation algorithms that allows us to use our intuitions about sorting algorithms to understand triangulation algorithms and show that this classification scheme deals with constrained Delaunay triangulation algorithms as well (chapter 2).

Sweepline algorithms for Delaunay triangulation and constrained Delaunay triangulation have been presented in terms of the dual Voronoi diagram. I give a clearer and simpler presentation showing how the sweepline algorithm is a direct search for Delaunay triangles (section 2.5).

I implement and illustrate all the Delaunay triangulation algorithms described in sections 2.2.1,2.3.1,2.4.1,2.5.1, and 2.6.1.

I prove that the only systematic local flip rule that is invariant under rotations and translations of the quadrilateral is the Delaunay flip rule (section 3.6.3).

I prove that the only systematic local flip rules invariant under scaling and translation correspond to generalizations of convex-distance-function Delaunay triangulations which I call *empty-shape triangulations* (section 3.6.4). I show how to modify Delaunay triangulation algorithms to produce empty-shape triangulations and constrained empty-shape triangulations (section 4.1). This also provides new algorithms for convex-distance-function Delaunay triangulations and constrained convex-distance-function Delaunay triangulations.

I present a complete working implementation of the new sweepline algorithm for empty-shape triangulations and convex-distance-function Delaunay triangulations (solving a problem posed in [91]) and implementation of the geometric primitives required for the other algorithms (section 4.1).

I show that that this approach does not generalize to three-dimensional convex-distance-function Delaunay triangulation and design an algorithm for this problem (section 4.2). I also prove some results bounding the complexity of three-dimensional convex-distance-function Delaunay triangulation (theorems 14 and 15).

I compute the average (taken over all possible triangulations) execution time for three algorithms for computing the Delaunay triangulation of a convex polygon (section 5.4). I measure the performance of the algorithms on random convex polygons and show that randomization of the algorithms is necessary to obtain the expected execution times (section 5.5). I also give an $O(1)$ space algorithm for convex-polygon Delaunay triangulation (solving a problem incorrectly solved in [83]).

I give several operational definitions of "random" convex polygons, design efficient ($O(n \log n)$ or better) algorithms to compute them and implement some of them (chapter 6). Two algorithms I developed as part of algorithms for convex polygon generation are interesting in their own right—a data structure that allows generation of variates in time $O(\log n)$ from a dynamically changing discrete distribution (section 6.2) and an $O(n)$ algorithm for realizing a Delaunay triangulation of a convex polygon (section 6.6.2).

I prove that the Delaunay triangulation optimizes several geometrical properties of the triangulation including maximizing the mean inradius (appendix A).

Chapter 2

Triangulation Algorithms

2.1 Introduction

There have been over a hundred papers published on various algorithms for Delaunay and non-Delaunay triangulation problems (table 2.2 on page 24).

We can classify Delaunay triangulation problems using two orthogonal axes:

constraint properties What is the nature of the constraint edges? Figure 2.1 shows a lattice of the constraint properties described in section 2.1.1.

metric properties What is the shape of the "circle" in this metric? Figure 2.2 shows a lattice of the metrics described in section 2.1.2.

Not included in this framework are non-Delaunay triangulations such as the Greedy triangulation and the Minimum Weight triangulation. These are discussed in chapter 3.

Section 2.1.3 describes the classification for the algorithmic paradigms used to classify Delaunay triangulation algorithms.

Sections 2.2 to 2.6 survey how each paradigm has been applied to each Delaunay triangulation problem. Where it has not been applied, I design an algorithm to demonstrate that it can be so applied.

2.1.1 Constraint properties

Note from figure 2.1 that all the other Delaunay triangulation problems are subsets of the constrained Delaunay triangulation problem. Simpler triangulation algorithms are possible for these problems, so it is worthwhile to consider them separately.

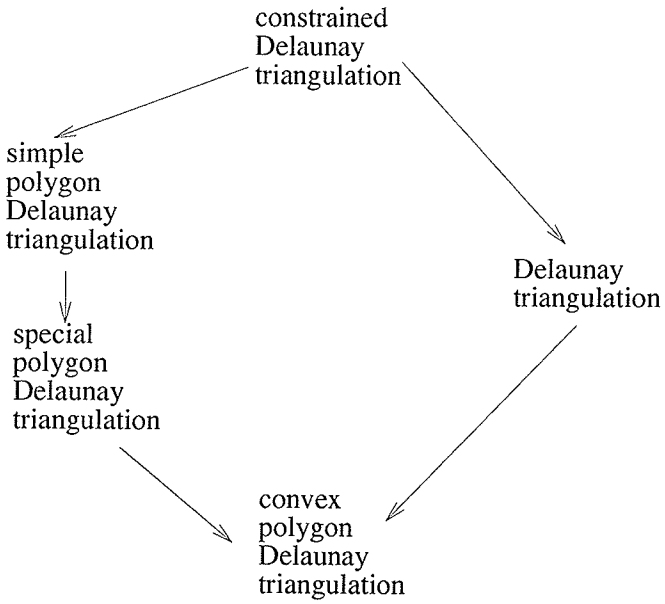


Figure 2.1: Constraint properties

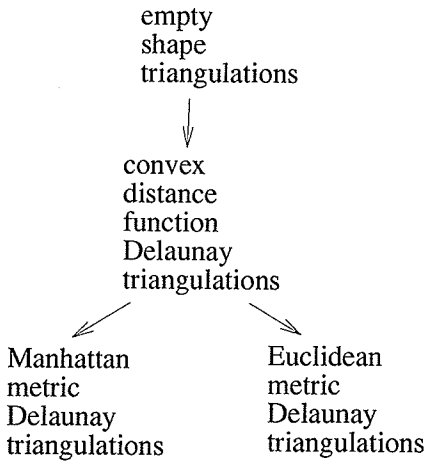


Figure 2.2: Metric properties

The Delaunay triangulation

Let S be a set of points in the plane. We will call the points in S *sites*.

To simplify the discussion we will assume that the set of sites is *non-degenerate*, that is, no three sites are collinear and no four sites are cocircular. Degenerate site sets can be made non-degenerate by a small perturbation of the sites. This can be done by modifying the computation of the geometric primitives rather than actually moving any sites [97].

The *Delaunay triangulation* of S is the unique triangulation of S such that the circumcircle of each triangle contains no site in its interior. Sites s and t are connected by a Delaunay edge iff there exists a circle through s and t which has no other site on its boundary or in its interior.

The *Voronoi polygon* of a site $s \in S$ is the set of points that are closer to s than to any other site in S. The Voronoi polygons of all the sites form a partition of the plane known as the *Voronoi diagram* of S. The Voronoi diagram is the dual of the Delaunay triangulation. Figure 1.1 shows the Voronoi diagram and Delaunay triangulation of a set of sites.

There is a strong relationship between the Delaunay triangulation and three-dimensional convex hulls [94, 147]. The *lifting map* sends each point (x, y) to the three-dimensional point $(x, y, x^2 + y^2)$. The lifting map sends the base plane to the paraboloid $z = x^2 + y^2$. To *lift* a triangulation we just apply the lifting map to its sites to get a triangulation embedded in three dimensions. The *lower convex hull* of a point set consists of those faces visible from $(0, 0, -\infty)$. The lift of the Delaunay triangulation is just the lower convex hull of the lifted sites. (This follows from the fact that the lift of a circle is the intersection of a plane with the paraboloid.)

If all except one of the sites are on a line, then any triangulation sorts the sites in the order in which they occur on that line. This means that there is a worst-case lower bound of $\Omega(n \log n)$ for any Delaunay triangulation algorithm.

Shamos and Hoey [295] were the first to develop an algorithm that attained this bound.

See the surveys by Aurenhammer [15], Fortune [120] and Okabe *et al.* [248] for more information on the properties of Voronoi diagrams and Delaunay triangulations.

Constrained Delaunay triangulation

Let S be a set of sites in the plane and E be a set of straight-line edges (*constraints*) connecting sites in S. A point A is *visible* from a point B if the segment AB does not cross

an edge of E.

The *constrained Delaunay triangulation* of (S, E) is the unique triangulation of S such that the circumcircle of each triangle contains no site visible from all three vertices of the triangle in its interior and the vertices of each triangle are mutually visible. Sites s and t are connected by a constrained Delaunay edge iff s is visible from t, and there exists a circle through s and t which has no other site on its boundary or in its interior visible to s and t.

The *bounded Voronoi polygon* of a site $s \in S$ is the set of points whose closest visible site is s. The bounded Voronoi polygons of all the sites form the *bounded Voronoi diagram* of S. Figure 2.3 shows the bounded Voronoi diagram and constrained Delaunay triangulation of a set of sites and constraints. Note that AB is a Delaunay edge but that the bounded Voronoi polygons of A and B are not adjacent, *i.e.* the bounded Voronoi diagram and constrained Delaunay triangulation are *not* dual.

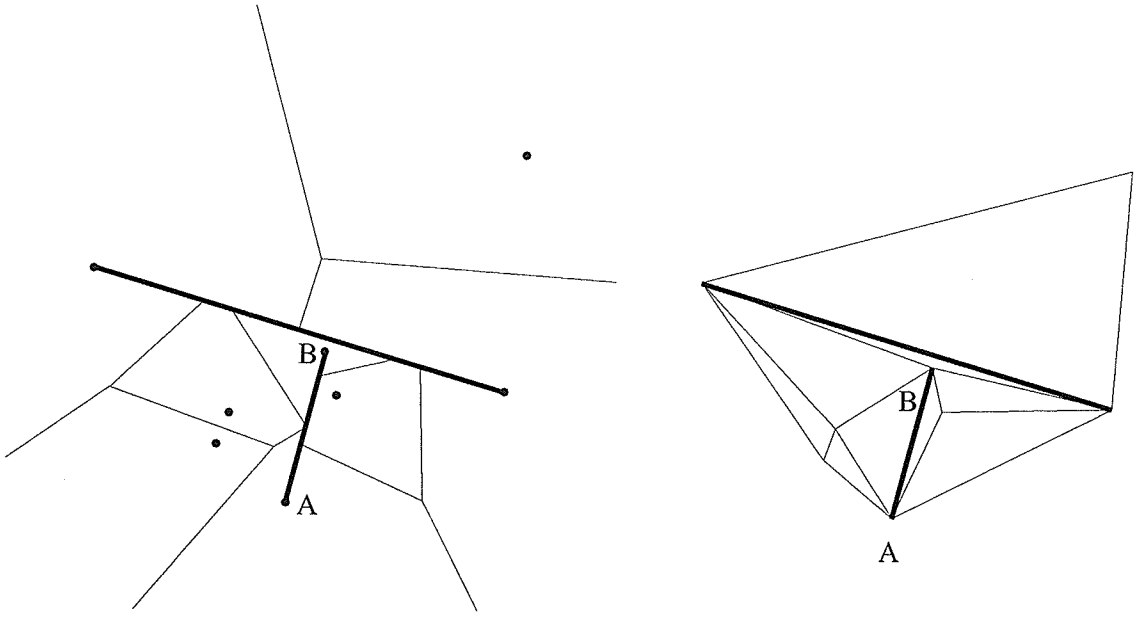


Figure 2.3: Bounded Voronoi diagram and constrained Delaunay triangulation

The dual of the constrained Delaunay triangulation is the *constrained Voronoi diagram.* This is formed by taking the bounded Voronoi diagram and gluing an extra sheet to it along each constraint edge so that if you cross the constraint edge, you move from the base plane to the associated extra sheet. If a Voronoi polygon is adjacent to a constraint in the

base plane, then it extends into the associated extra sheet. Figure 2.4 shows the two extra sheets that are glued to the bounded Voronoi diagram in figure 2.3 to form the constrained Voronoi diagram.

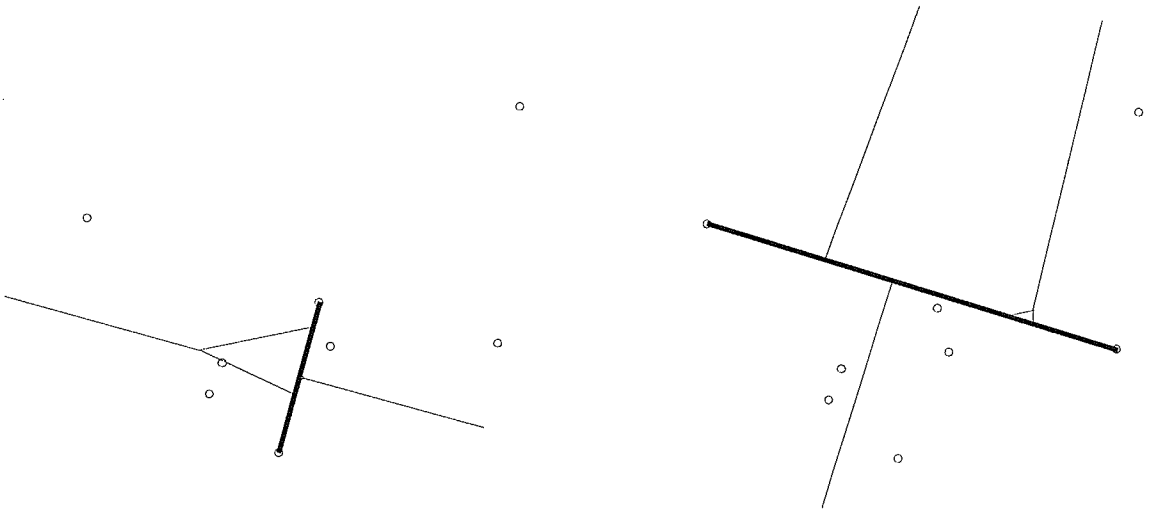


Figure 2.4: Extra sheets in constrained Voronoi diagram

The lift of the constrained Delaunay triangulation is just the lowest triangulated surface which contains all the constraint edges.

Chew [57] and Wang and Schubert [320] were the first to develop worst-case optimal $O(n \log n)$ algorithm for constrained Delaunay triangulation.

Some applications of constrained Delaunay triangulations are listed in section 1.4. For more on the properties of constrained Delaunay triangulations see Joe and Wang [165]. (They say "constrained Voronoi diagram" instead of "bounded Voronoi diagram" and "extended constrained Voronoi diagram" instead of "constrained Voronoi diagram".)

A related construction is the *conforming Delaunay triangulation.* The conforming Delaunay triangulation of (S, E) is the Delaunay triangulation of $S' \supset S$, where S' is chosen such that no edge in the Delaunay triangulation of S' crosses an edge in E. Figure 2.5 shows the conforming Delaunay triangulation corresponding to the constrained Delaunay triangulation shown in figure 2.3.

The usual approach used to construct a conforming Delaunay triangulation is to repeatedly add sites on constraint edges that are crossed by Delaunay edges until the triangulation

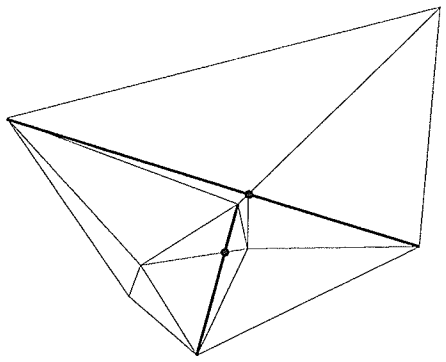


Figure 2.5: Conforming Delaunay triangulation. Added points are marked with bullets.

is conforming [29, 30, 150, 239, 249, 276, 278, 287, 314, 330]. This approach seems to work well in practice, though it is possible that vast numbers of extra sites would have to be added. Edelsbrunner and Tan [101] show that with n sites and m edges, $\Omega(mn)$ extra sites can be required and give an algorithm to find a conforming triangulation with at most $O(m^2 n)$ extra sites.

Convex-Polygon Delaunay triangulation

If the sites form a convex polygon, and we are given the order in which they occur around the boundary, the $\Omega(n \log n)$ lower bound no longer applies, and Aggarwal *et al.* [5] have developed a worst-case $O(n)$ algorithm. Chapter 5 is devoted to the analysis, implementation and performance measurement of several algorithms for this problem.

Simple polygon Delaunay triangulation

Definition. A polygon with vertices $p_0, p_1, \ldots, p_{n-1}$, and edges $e_i = p_i p_{i+1}$ (define p_n as p_0) is a *simple polygon* if

- adjacent segments intersect only at their shared vertex: $e_i \cap e_{i+1} = p_{i+1}$.
- non-adjacent segments do not intersect: $e_i \cap e_j = \emptyset$ if $j \neq i + 1$.

This is a special case of constrained Delaunay triangulation where the constraints form a simple polygon and the sites are the endpoints of the constraint edges. Furthermore, we are only interested in the part of the triangulation inside the polygon. The incremental

algorithm for constrained Delaunay triangulation (section 2.3.2) requires this computation as one step.

The $\Omega(n \log n)$ lower bound does not apply in this case either, and recently Klein and Lingas [181] have presented a randomized algorithm that takes expected linear time.

In contrast, calculating the Delaunay triangulation of the vertices of a simple polygon still requires time $\Omega(n \log n)$ [4, 290]. That is, the constraints make the problem "easier".

Special polygon Delaunay triangulation

Triangulations of particular kinds of polygons are required as steps in other geometric problems.

A polygon is *monotone* if there is a line l such that all lines parallel to l intersect the polygon at most twice. Yeung [338] provides a $O(n \log n)$ algorithm for the Delaunay triangulation of a monotone polygon.

If we delete a site from a Delaunay triangulation it is necessary to retriangulate the polygon formed by the union of all the triangles adjacent to the deleted site. This *Delaunay deletion polygon* is characterized by having the intersection of the circumcircles of all triangles formed from three vertices be nonempty. Aggarwal *et al.*'s convex-polygon Delaunay triangulation algorithm can be generalized to triangulate Delaunay deletion polygons in linear time [5].

A polygon P is a *Delaunay monotone polygon* if there is a line which intersects every internal edge of the Delaunay triangulation of P [321]. These arise when a single constraint is inserted into a constrained Delaunay triangulation, and their Delaunay triangulations can be found in linear time [204, 321].

A polygon P with vertices $p_0, \ldots, p_n$ is a *normal histogram* if the p_i have ascending x-coordinates, p_0 and p_n have the same y-coordinate, and all other vertices have larger y-coordinates. Klein and Lingas present a linear algorithm for Delaunay triangulation of normal histograms as a step in their linear algorithm for the Delaunay triangulation of a simple polygon [181].

2.1.2 Metric properties

Definition. A *homothety* $h(t_x, t_y, k)$ is a product of a translation by (t_x, t_y) and a scaling by k.

$$h(t_x, t_y, k)(x, y) = (t_x + kx, t_y + ky).$$

If $k \neq 1$ then the homothety has a fixed point P, so we will also call it $H(P,k)$. A set A is a *homothet* of a set B of there is a homothety H such that $H(B) = A$.

Lay [195] and Coxeter [66] (who calls them dilatations) cover some of the properties of homotheties.

I will use *shape set* to refer to an equivalence class under homotheties. For example, the set of all axis-parallel squares is a shape set.

Definition. A convex body is *strictly convex* if it contains no straight line segments in its boundary [317].

Definition. A directed line l is a *support line* of a set K iff l contains a boundary point (a *support point*) of K and K is contained in the closed halfplane to the left of l.

We can associate directed lines with the half-planes to their left. A convex set can be seen to be equal to the intersection of the half spaces associated with its support lines.

Definition. A convex body is *smooth* if there is a unique support line at each boundary point [195].

Definition. The *polar set* K^* of K is defined by

$$K^* \equiv \{(x,y) | ax + by \leq 1 \text{ for all } (a,b) \in K\}.$$

If K contains a single point $(a,b) \neq (0,0)$ then K^* is the closed half-plane $ax + by \leq 1$. The *dual* of such a point is its polar set K^*. The *dual* of a closed convex set K with $(0,0)$ in its interior is its polar set K^*. See Lay [195] for more details.

Boundary points and support lines are dual—that is, the directed line associated with the dual of a boundary point p of K is a support line of the dual K^*. Smoothness and strict convexity are dual—if there are two support lines at a boundary point of K, then, in the dual there are two boundary points incident on the same support line, and the boundary of the dual will contain the segment joining these two points. In other words, K is smooth if and only if the dual of K is strictly convex.

Definition. Given a closed convex set K with $(0,0)$ in its interior, then the *convex distance function* $f : \mathbf{R}^2 \rightarrow \mathbf{R}$ of K is defined by

$$d(x,y) = \inf\{k | (x-y) \in h(0,0,k)K\}.$$

K is called the *ball* for the convex distance function.

Note that we have generalized the definition given by Chew and Drysdale [55] by allowing the convex set to be unbounded.

If we let P' be the point where the ray OP (O is the origin) intersects the boundary of K then $d(P,O) = |OP|/|OP'|$ (see figure 2.6). The Euclidean metric is the convex distance function of the unit disc $\{(x,y)|x^2+y^2 \leq 1\}$. The Manhattan metric is the convex distance function of the square with corners $(0,1)$, $(1,0)$, $(0,-1)$, and $(-1,0)$.

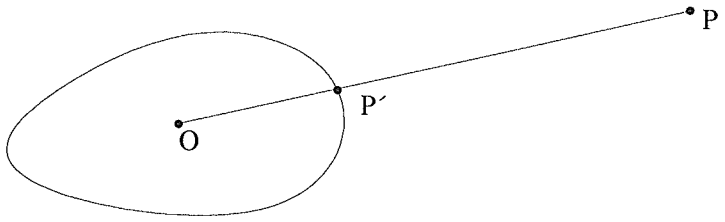


Figure 2.6: Ball for a convex distance function

Given a ball K, a *circumball* of a triangle T is a homothet of that ball with the vertices of T on its boundary. We say that K *circumscribes* T. If the ball is smooth and strictly convex then every non-degenerate triangle has a unique circumball [187].

The convex-distance-function Voronoi diagram is defined just as in section 2.1.1, but using the convex distance function to measure distance. In the dual convex-distance-function Delaunay triangulation each triangle has a circumball with no site in its interior.

If the ball is not smooth then some triangles will not have circumballs and the outer face of the Delaunay triangulation is not the convex hull but the *support hull* [91]. (See figure 4.6 for an example.) An edge PQ is part of the support hull iff there is an infinitely large homothet of the ball with P and Q on its boundary and no site in its interior.

Similarly, if the ball is unbounded, some triangles will not have circumballs and the outer face is the support hull. In addition, some sites will be outside the the support hull (See figure 4.8 for an example.). We can regard an unbounded ball as having a corner at infinity.

If the ball is not strictly convex and the line through two sites is parallel to a straight-line segment on the boundary, then there may be an infinite number of circumballs for a

triangle with the two sites as two vertices. Consequently, the Delaunay triangulation is ambiguously defined. A small perturbation of the sites will solve this problem, so we can deal with it in the same way that other degeneracies are dealt with.

In this thesis I further generalize convex-distance-function Delaunay triangulation to define the *empty-shape triangulation*. If we are given a set of balls $\mathcal{K}$ such that for any triangle T there is exactly one ball in $\mathcal{K}$ which circumscribes T, in the empty-shape triangulation each triangle has an empty circumball. Figure 4.6 shows an empty-shape triangulation where $\mathcal{K}$ consists of all homothets of a triangle and three hyperbolae.

Table 2.1 shows the Delaunay triangulation problems for which algorithms have been published. Chapter 4 gives algorithms for constrained empty-shape triangulations, which are a superset of all the problems in table 2.1.

	empty-shape triangulation	convex distance function	Euclidean	Manhattan
constrained			[57]	
no constraints		[55]	[295]	[158]
simple polygon			[181]	[180]
convex polygon			[5]	

Table 2.1: Delaunay triangulation problems

2.1.3 Algorithm paradigms

The one-dimensional analogue of triangulation is sorting. Much has been written on the taxonomy of sorting algorithms. One commonly used classification scheme is that of Knuth [185], who classifies sorting algorithms into *insertion*, *selection*, *exchange*, and *divide and conquer*. If all except one of the input sites are on a line, then the triangulation algorithm functions as a sorting algorithm, so it should be no surprise that we can adapt Knuth's scheme to classify triangulation algorithms.

Flip

Bubble sort repeatedly exchanges adjacent elements that are out of order until the sequence is sorted. The number of exchanges is equal to the number of inversions in the list which is $O(n^2)$ in the worst and average case.

By exchanging non-adjacent elements it is possible to obtain a $O(n \log n)$ algorithm.

Analogously, a *flip algorithm* for triangulation repeatedly modifies a triangulation by "exchanging" diagonals of convex quadrilaterals in the triangulation that are "out of order" until the triangulation is "sorted". In this context to "exchange" means to flip[1] the diagonal, "out of order" means that the triangulation of the convex quadrilateral is not Delaunay and "sorted" means that the triangulation is Delaunay.

Incremental

Insertion sort inserts each point into the sorted sequence in turn. This takes time $O(n)$ in the worst and average cases, giving $O(n^2)$ time to sort n points.

Using a more sophisticated data structure, (such as an AVL tree) to store the sorted sequence enables the insertion to be carried out in time $O(\log n)$, leading to a $O(n \log n)$ algorithm.

Analogously, an *incremental algorithm* for triangulation maintains a triangulation of the sites processed so far. Each new site is inserted in the triangulation in turn.

Selection

Selection sort outputs a sorted sequence by selecting the smallest element in a sequence, then the next to smallest and so on. Finding the smallest element takes time $O(n)$, so the total time taken is $O(n^2)$.

By using a priority queue data structure (such as a heap) that enables the selection to made in time $O(\log n)$ we can obtain a $O(n \log n)$ algorithm (heapsort).

Analogously, a *selection algorithm* for triangulation finds Delaunay triangles one at a time from the set of sites.

Sweepline

The sweepline paradigm is an important computational geometry paradigm that can be used to create a selection triangulation algorithm by finding Delaunay triangles in the order that a sweepline crosses the rightmost point of their circumball. Although sweepline algorithms are selection triangulation algorithms, they are important enough to be given their own category.

[1] defined on page 72

The sweepline algorithm is analogous to heap sort in that it uses a priority queue data structure to select Delaunay triangles in time $O(\log n)$ per triangle.

Divide and Conquer

Merge sort divides the sequence to be sorted into two equal sized sequences. These are sorted recursively and the results merged to produce a final sorted sequence. Since the merge step takes $O(n)$ time, merge sort takes $O(n \log n)$ time overall.

Analogously, the Divide-and-Conquer triangulation algorithm divides the sites into two equal sized sets, recursively triangulates each set, and merges the two triangulations.

Published algorithms

Table 2.2 classifies published triangulation algorithms using the above scheme. The great popularity of the incremental algorithm is evident.

2.2 Flip Triangulation Algorithms

2.2.1 Delaunay triangulation

The flip algorithm for the Delaunay triangulation constructs an initial triangulation and then does Delaunay flips until no more flips are possible.

Lawson [194] proved that the flip algorithm will converge to the Delaunay triangulation, no matter what order the flips are done in.

The initial triangulation can be made by constructing a *star triangulation* by connecting a site to all others and then filling in the concavities [233], finding a spiral path through the sites and then filling in between whorls of the spiral [222], by repeatedly dividing the sites into two by finding a path connecting sites [207], or by connecting sites to other visible sites [19].

We can repeatedly make passes over all the edges of the triangulation, flipping any eligible edges and stopping when a pass is made without any flips [233]. Or, if we use a list of triangles to store the triangulation, we can check each triangle against its three neighbours. Once a triangle has been tested it need not be tested again, so one pass over the triangle list, with new triangles being added at the end, will suffice. Figure 2.7 shows the sequence of triangulations that this method produces, starting with a star triangulation.

	Flip	Incremental	Selection	Sweepline	Divide and Conquer
Delaunay triangulation	[19, 119, 161, 207, 222, 269, 280]	[7, 8, 31, 34, 35, 36, 42, 58, 64, 65, 82, 85, 115, 116, 119, 123, 131, 134, 135, 145, 148, 150, 152, 163, 168, 194, 197, 202, 213, 221, 225, 234, 246, 249, 253, 260, 294, 296, 300, 302, 303, 308, 314, 327, 339, 340]	[20, 22, 37, 40, 69, 81, 105, 112, 113, 124, 154, 175, 183, 198, 224, 227, 229, 271, 281, 305, 307]	[122, 199]	[39, 70, 92, 104, 147, 172, 173, 199, 202]
constrained Delaunay triangulation	[33, 231, 252, 309]	[14, 27, 63, 73, 76, 142, 160, 169, 171, 181, 219, 220, 301, 320, 321]	[201, 216, 218, 228, 241]	[291]	[57, 165, 236, 237, 279]
simple polygon Delaunay triangulation	[76, 301]	[63]			[201]
special polygon Delaunay triangulation		[181, 182]			[171, 338]
convex polygon Delaunay triangulation	[161]	[55]	[83]		
convex-distance-function Delaunay triangulation				[45, 77, 121, 299, 297]	[91, 158, 170, 200, 203, 205]
non-Delaunay triangulations	[109, 136, 138, 156, 233, 254, 325]	[139, 211, 204]	[235]		

Table 2.2: Published Triangulation Algorithms

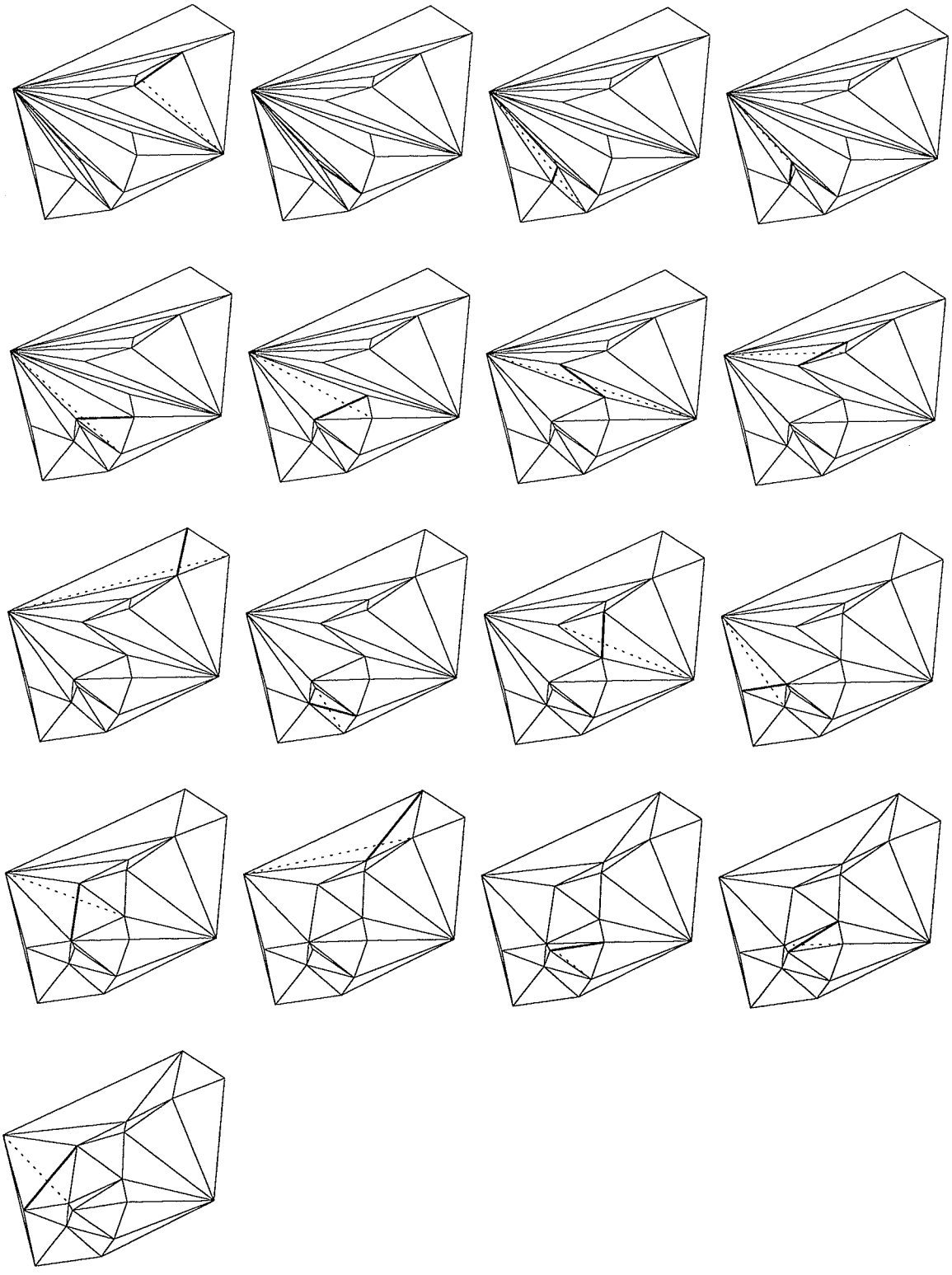


Figure 2.7: Flip algorithm

Efficiency of the flip algorithm

In the lifted triangulation each flip causes the surface to move downwards. Consequently, once an edge is deleted it can never come back. Since there are $\binom{n}{2}$ possible edges there can be at most $O(n^2)$ flips.

2.2.2 Constrained Delaunay triangulation

The flip algorithm is the conceptually simplest algorithm for constructing the constrained Delaunay triangulation. Construct an initial triangulation that contains all the constraint edges. Do constrained Delaunay flips until no more are possible. A constrained Delaunay flip differs from a Delaunay flip in that it will not delete a constraint edge. The proof of the correctness is almost identical to that for the unconstrained flip algorithm and the worst case is still $O(n^2)$.

The only difficulty is that it is no longer trivial to construct the initial constrained triangulation. Garey *et al.* [132] give a $O(n \log n)$ for this problem. Implementing this algorithm is the most complicated part of implementing the flip algorithm for the constrained Delaunay triangulation, since it requires two stages: first a plane sweep from left to right that adds edges so that each site has an edge to its left and an edge to its right, and then triangulation of the resulting monotone polygons.

There have been no published implementations using this method.

In many applications, the constraint edges form a simple polygon. We can triangulate the simple polygon and complete the triangulation by triangulating the sites that fall into each triangle by any of the above methods. There exists a very simple algorithm to triangulate the simple polygon. An *ear* of a polygon is a triangle ABC where A, B and C are successive vertices along the boundary, $\angle ABC$ is convex, and $\triangle ABC$ contains no polygon vertex in its interior. Meisters [230] has proved that any polygon has at least two ears. The *ear-cutting algorithm* triangulates a polygon in time $O(n^3)$. It finds an ear in time $O(n^2)$ by testing (in time $O(n)$) each of $O(n)$ possibilities, cutting this ear from the polygon and recursively triangulating the resulting polygon.

This algorithm has been proposed in [337] and [336], while [231] and [252] use it with the flip algorithm.

A slightly more complicated algorithm, also with $O(n^3)$ worst-case complexity, directly searches for a triangle standing on a side that does not intersect a polygon edge [51].

Borgers [33] uses it with the flip algorithm.

Implementors have used these simple $O(n^3)$ algorithms because in their applications the number of constraints is small relative to the number of sites, so the flipping time dominates.

We should also note that there has been much research into finding $o(n \log n)$ algorithms[2] for polygon triangulation. Such algorithms were found for special cases [50, 107, 153, 310, 313, 335], then general $O(n \log \log n)$ algorithms [178, 306] and $O(n \log^* n)$ randomized algorithms [62, 60, 292] were developed before Chazelle found a deterministic $O(n)$ algorithm [48].

2.2.3 Simple polygon Delaunay triangulation

We construct an arbitrary triangulation of the simple polygon and then apply the flip algorithm. We have already discussed triangulating simple polygons in section 2.2.2.

2.2.4 Convex-Polygon Delaunay triangulation

In this case it is particularly easy to construct an initial triangulation. For example, connecting one vertex to all others will suffice. Joe [161] describes a flip algorithm for convex polygon Delaunay triangulation.

2.2.5 Convex-Distance-Function Delaunay triangulation

If the convex-distance-function ball is smooth and bounded then the flip algorithm will work in the same way that it does in the Euclidean metric. If not, two difficulties are encountered:

1. We must first find the support hull and construct an initial triangulation of that.
2. The convex-distance-function Delaunay triangulation of a convex quadrilateral may not contain any triangles, making it unclear which way to flip the diagonal.

In section 4.1.2 I present a way to deal with these difficulties.

[2] $o\big(g(n)\big)$ is the set of functions $f(n)$ such that $|f(n)| \leq \epsilon |g(n)|$ for all $\epsilon > 0$ [144].

2.3 Incremental Triangulation Algorithms

2.3.1 Delaunay triangulation

There are two main ways that a new site can be inserted into a triangulation. Watson's method [326], is to scan through all the triangles, deleting all those triangles whose circumcircles contain the new site. The deleted triangles form a star-shaped polygon.[3] New triangles are then added by connecting the new site to each boundary edge of this polygon. The list of boundary edges can be computed by collecting all the edges of the deleted polygons and discarding all the edges that occur twice. For this method, we do not need a data structure that keeps track of triangle adjacencies. A simple list of triangles will suffice.

Another method, due to Lawson [194] connects the new site to all those sites visible from it. This triangulation is then converted to a Delaunay triangulation by repeated flips. The only possible candidate triangles for flipping are the new triangles that have just been created by connecting the new site.

If the new site is inside the convex hull of the sites triangulated so far, then the only sites visible from it will be the three corners of the triangle that it is inside, so to simplify programming you can start with a triangulation of some dummy sites whose convex hull includes all the real sites [202].

If the new site is outside the convex hull of the sites triangulated so far, then the only sites visible from it will be a sequence of sites on the convex hull. To ensure that this is the only case that you need to consider in your triangulation algorithm you can sort all the sites in order of distance from the origin [194], or sort them by their x coordinate [65, 308].

Efficiency of Incremental Algorithms

In the worst case, an incremental algorithm will require $O(n)$ time to insert a site in the triangulation, since the new site might have to be connected to all $O(n)$ other sites. This means that the total time will be $O(n^2)$ in the worst case. A sequence of sites along a half-parabola [202] is an example of this worst case.

Most implementations of incremental algorithms perform much better than this in practice for reasons explained below:

[3] A *star-shaped polygon* is a polygon with a point in its interior from which every other interior point is visible.

Insertion by Flipping When a site is inserted into a Delaunay triangulation the only new edges are those adjacent to the new site, so the number of flips required is d, the degree of the new vertex. If the sites are uniformly distributed, he expected value of d is 6 and the expected number of flips is $O(n)$. Guibas, Knuth and Sharir [148] generalize this result to show that the expected number of flips is $O(n)$ for any site set, provided that the sites are inserted in random order.

The other time-consuming part of incremental algorithms, is finding the sites to initially connect the new site to.

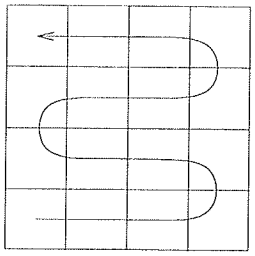
If we reorder the sites such that the new site is outside the convex hull of the sites triangulated so far, it is necessary to search the convex hull of the sites. If the sites are uniformly distributed and we order the sites by the distance from the origin, the expected number of sites in the convex hull is $O(n^{1/3})$ [268], giving a total search time of $O(n^{4/3})$ [194]. If sites are uniformly distributed over a rectangle, then the expected number of sites in the convex hull is $O(\log n)$ [270], so sorting the sites by x coordinate will lead to a total search time of $O(n \log n)$ [308].

If the new site is inside the convex hull, it is necessary to find the containing triangle.

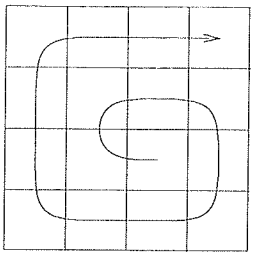
We can use the partial Delaunay triangulation for this. We test the new site against each side of a given triangle. If it is inside each side, we have found the containing triangle, otherwise the next triangle to be checked is the one on the other side of a side that the site is outside. We can walk across the triangulation from one triangle to an adjacent triangle until we find the containing triangle. If the sites are uniformly distributed, the average number of triangles intersected by a line is $O(\sqrt{n})$ (see below), so in this case the search time is $O(\sqrt{n})$ and the search time for the entire algorithm is $O(n^{3/2})$ [145].

We can improve on this by reordering the sites so that successive sites are close together, and starting the search for the containing triangle at the site of last insertion.

A fast way to do this sorting is to divide an enclosing rectangle into b buckets and in time $O(n + b)$ reorder the sites by the buckets they fall into. Lee and Schachter [202] proposed using a serpentine or spiral order (see figure 2.8) and about $\sqrt{n}$ buckets. If the sites are uniformly distributed, there will be $O(\sqrt{n})$ sites in each bucket and an average search time of $O(n^{1/4})$ per site, and a total search time of $O(n^{5/4})$. Sloan [300] reports that for an implementation of the above scheme using serpentine ordering the observed run time was $O(n^{1.06})$ for sites uniformly distributed on the unit square. He suggests that this is because for the values of n tested ($< 10,000$) the time for the $O(n)$ flip operations dominates. Ohya,



Serpentine



Spiral

23	22	32	33
21	20	30	31
01	00	10	11
03	02	12	13

Quatenary

Figure 2.8: Proposed bucket orderings

Iri and Murota [247] got similar results for an implementation using a spiral ordering (about $O(n^{1.05})$ for $n < 32,000$).

Ohya, Iri and Murota [246] also tried using n buckets so that there is an average of one site per bucket and a quaternary ordering for the buckets. (The numbers in figure 2.8 are in base 4 to make the pattern clearer.) The observed execution time was $O(n)$, for $n < 32,000$.

Agishtein and Migdal [7] sort the sites by the order that they occur on a space-filling fractal curve. This takes time $O(n \log n)$ for any distribution of sites. The observed execution time to compute the triangulation (not counting the initial sort) was $O(n)$ for $n < 90,000$. This approach is similar to using quaternary bucketing without a fixed number of buckets, and subdividing any bucket with more than one site.

Note that for all the schemes that involve re-ordering the sites, the Guibas-Knuth-Sharir guarantee (that the expected number of flips is $O(n)$ regardless of the site distribution) is voided. Indeed, for sites on a half-parabola, sorting by x coordinate, or spiral or serpentine bucket ordering can lead to $\Omega(n^2)$ flips since the order of insertion could be the order that sites occur on the half-parabola. The quaternary ordering scheme is designed to be more robust by mixing up the ordering of the buckets. If the sites are on a half parabola through the bottom row of buckets, with one site per bucket, the total number of flips to produce the triangulation satisfies the recurrence

$$f(1) = 0 \tag{2.1}$$

$$f(2) = 1 \tag{2.2}$$

$$f(2^{j+1}) = f(2^j) + 2^j(j+1), \tag{2.3}$$

The solution to this is $\Theta(n \log n)$,[4] so while better than the other schemes, the number of flips is still suboptimal for this particular distribution of sites.

By keeping all the intermediate triangulations, Guibas, Knuth and Sharir [148] are able to locate the insertion triangle in average time $O(\log n)$ (average over all insertions and all insertion orderings, regardless of site distribution). Whenever a triangle is flipped, or subdivided by a new site, instead of deleting it, they keep it around, with pointers to the two (if flipped) or three (if subdivided) new intersecting child triangles. Since the expected number of flips to construct the triangulation is $O(n)$, the total space required is $O(n)$ on average. To find the insertion triangle for a new site it is just necessary to start at the root triangle that contains all others, and move to the child which contains the new site until a leaf triangle is reached. The total search time to construct the triangulation is then $O(n \log n)$.

Watson's algorithm If we do not keep track of triangle adjacencies and use Watson's method of deleting all triangles whose circumcircles include the new site, we can avoid having to search all the triangles by pre-sorting the sites by x coordinate. When searching the triangles, if we discover one whose circumcircle does not intersect the vertical line through the new site, then we know that no later site can be inside the circumcircle of that triangle, and it need not be checked when inserting later sites. So, when inserting a new site, it is only necessary to check those triangles whose circumcircles intersect the vertical line.

If the sites are uniformly distributed inside the unit square, we can estimate this number. A circle of radius R has a probability of $2R$ of intersecting a random vertical line (ignoring edge effects). The expected value of R is $3/(4\sqrt{n})$ [232], and there are $2n$ Delaunay triangles (again ignoring edge effects), so a random vertical line will intersect $3\sqrt{n}$ Delaunay circumcircles on average.

We also need to consider the triangles that are Delaunay triangles of the sites to the left of the line, but whose circumcircles contain a site to the right of the line. An edge of such a triangle must be intersected by a Delaunay edge that crosses the vertical line, so we can bound this number by considering the number of Delaunay edges that cross a vertical line. A line of length l has probability $2l/\pi$ of intersecting a random vertical line [277]. The expected length of a Delaunay edge is $32/9\pi\sqrt{n}$ [232], and there are $3n$ Delaunay edges

[4] $\Theta\big(g(n)\big)$ is the set of functions $f(n)$ such that $C|g(n)| \leq |f(n)| \leq D|g(n)|$ for some $C, D > 0$ [144].

(ignoring edge effects), so a random vertical line will intersect $(64/3\pi^2)\sqrt{n}$ Delaunay edges.

Hence, it only necessary to search $O(\sqrt{n})$ triangles, giving time $O(n^{1.5})$ for uniformly distributed sites. Sloan and Houlsby [302] report execution times growing at $O(n^{1.5})$ for an implementation of the above approach.

If we record triangle adjacencies, then the same triangulation walking, site reordering, and keeping old triangulations methods can be used as in flip insertion.

The Delaunay tree, proposed by Boissonnat and Teillaud [31], by keeping old triangulations around allows us to find the triangles whose circumcircles contain the new site in average time $O(\log n)$. The main difference from the Guibas-Knuth-Sharir method is that fewer triangles are kept—the temporary triangles created while flipping that are not in a partial Delaunay triangulation are not included. This saves space, but makes the data structure a little more complicated since triangles have a variable number of children.

A similar scheme was proposed by Palacios-Velez and Renaud [253]. The main difference is that for deleted triangles we just store a pointer to the site whose insertion caused the deletion of this triangle, since each new triangle that intersects it must be adjacent to this site. When searching for an enclosing triangle for a new site, we can walk on the triangulation, starting at a triangle adjacent to the deletion causing site. They measured a total search time of $O(n \log n)$ for an implementation of this method for $n < 15,000$ and uniformly distributed sites. They also compared this method with a simple triangulation walk scheme. This proved to be slower for $n > 500$ but sorting the sites made a dramatic difference—in this case fewer than 4 triangles needed to be searched for each insertion, as compared with 24 for the hierarchical method and 35 for unsorted triangulation walking. (This is for $n = 2000$.)

Table 2.3 further classifies the incremental Delaunay triangulation algorithms from table 2.2 by the way insertions are carried out.

Watson's algorithm	[31, 82, 85, 115, 116, 135, 152, 197, 253, 260, 302, 314, 327, 339]
Flip	[34, 163, 296]
inside	[7, 36, 42, 58, 119, 123, 131, 134, 148, 150, 168, 202, 221, 234, 246, 294, 300, 303]
outside	[8, 65, 194, 308]

Table 2.3: Incremental Delaunay Triangulation Algorithms

2.3.2 Constrained Delaunay triangulation

Given a constrained Delaunay triangulation we can insert a new site or a new constraint. Inserting a new site is done in the same way as in the incremental Delaunay triangulation algorithm—we connect it to all the sites that it can see and then use the constrained flip algorithm to get the new constrained Delaunay triangulation.

To insert a constraint it is necessary to delete the k edges that it intersects and then insert the edge. If $k > 1$ the result will contain one or two non-triangular faces—a p-gon and an r-gon,where $p+r = k+5$. We need to compute a simple polygon Delaunay triangulation. This has been done by the selection algorithm [63] (section 2.4.3), the flip algorithm [76, 301] (section 2.2.3), and the divide-and-conquer algorithm [169] (section 2.6.3).

If we use a worst case $O(k^2)$ (*e.g.* selection) algorithm, then since k is $O(n)$ and there could be $O(n)$ constraints, the worst-case execution time is $O(n^3)$. However, in many applications, k tends to be small (*i.e.* the constraint edges are short), or there are not many constraints, so this approach can give reasonable performance.

In fact, the faces to be retriangulated are Delaunay monotone polygons (section 2.1.1) and hence can be triangulated in time $O(k)$, yielding $O(n^2)$ worst-case algorithms [204, 321].

If we do not require an on-line algorithm, we can insert all the constraints before doing any retriangulating. This leaves us with a set of simple polygons to Delaunay triangulate. If we use an $O(n \log n)$ algorithm for this, we obtain a $O(n \log n)$ for constrained Delaunay triangulation [169, 320].

2.3.3 Simple polygon Delaunay triangulation

The natural way to apply this paradigm is to add vertices one at a time to the polygon, maintaining a simple polygon triangulation at all times. In particular, if the polygon has vertices $p_1 p_2 \ldots p_n$ and $\{p_{k_1}, p_{k_2}, \ldots, p_{k_i}\}$ is a subset of the vertices such that $k_1 < k_2 < \ldots < k_i$ we want the Delaunay triangulation of the polygon $p_{k_1} p_{k_2} \ldots p_{k_i}$. Unfortunately, this polygon may not be simple (its edges might cross). We could also try adding the constraints one at a time, but a subset these might not form a simple polygon either.

Consequently, we must insert points in an order such that $p_{k_1} p_{k_2} \ldots p_{k_i}$ is a simple polygon. One way to achieve this is to first triangulate the simple polygon (section 2.2.2). Pick any triangle to start, and then add triangles that share an edge with previously added triangles, one at a time. This introduces one new vertex at each step. Each intermediate

polygon is simple because its edges are edges of the polygon triangulation, and hence do not intersect. The Delaunay triangulation can be updated by flipping in the usual manner.

Note that since sites are not inserted in a random order the Guibas-Knuth-Sharir [148] result does not apply: we cannot say that the expected number of flips is $O(n)$.

2.3.4 Convex-Polygon Delaunay triangulation

If the sites are the vertices of a convex polygon then the two sites to connect a new site to can be found in constant time. If the sites are inserted in a random order the expected total number of flips is $O(n)$ and the total time is $O(n)$. This result was first proved by Chew [52].

2.3.5 Special polygon Delaunay triangulation

Why were we able to find a linear randomized insertion algorithm for convex polygons but not for simple polygons? The difference was that for convex polygons, we were guaranteed that polygons like $p_{k_1}p_{k_2}\ldots p_{k_i}$ were simple. So we also have a linear randomized insertion algorithm for polygons with the property that a line joining any two vertices does not cross a polygon side.

Klein and Lingas [182] show that the same applies for polygons with the property that sides of polygons like $p_{k_1}p_{k_2}\ldots p_{k_i}$ were edges of the Delaunay triangulation or of the furthest-site Delaunay triangulation[5] of the sites $\{p_{k_1}, p_{k_2}, \ldots, p_{k_i}\}$. Such polygons include Delaunay deletion polygons and monotone histograms.

2.3.6 Convex-Distance-Function Delaunay triangulation

If the convex-distance-function ball is smooth and bounded then the incremental algorithm will work in the same way that it does in the Euclidean metric. If not, two difficulties are encountered:

1. We don't always know how to flip diagonals (see section 2.2.5).

2. Connecting a new site to all visible sites may add edges outside the support hull.

Drysdale [91] presents an algorithm that deals with these difficulties in the case of non-smooth balls. In section 4.1.2 I present a way to deal with these difficulties in all cases.

[5] Triangles of the furthest-site Delaunay triangulation contain all other sites in their circumcircles.

2.4 Selection Triangulation Algorithms

2.4.1 Delaunay triangulation

The naive selection algorithm just considers all $\binom{n}{3}$ possible triangles. We check each triangle to see if its circumcircle is empty in time $O(n)$, leading to a $O(n^4)$ algorithm.

Given a Delaunay edge we can find the Delaunay triangle on a given side of that edge by a simple scan through the sites on that side. We start with a candidate site for the third site of that triangle. If another site is inside the circumcircle of the candidate triangle, then that site becomes the new candidate.

The *circumcircle algorithm* [229] starts by finding a Delaunay edge (for example, by finding the closest site to a particular site). We place this edge and its reverse on a stack. The algorithm proceeds by popping an edge from the stack, finding the Delaunay triangle on that edge, and pushing the two new Delaunay edges onto the stack. If we push an edge onto the stack and its reverse is already on the stack, we remove both edges.

Figure 2.9 shows the the sequence of triangles constructed. Triangles around the convex hull are found first, then it spirals inward. This is a depth first search of the dual graph of the triangulation.

If a queue is used instead of a stack (figure 2.10), the boundary tends to sweep across the triangulation, and the size of the boundary is likely to be smaller. A queue gives a breadth first search of the dual graph of the triangulation. Figure 2.11 shows average and maximum boundary sizes for each data structure, with sites taken from the uniform distribution over the unit square.

For each Delaunay edge we do a $O(n)$ search through all the sites, so this algorithm takes time $O(n^2)$. The edges stored on the stack form the boundary of the area triangulated so far, so the stack will have maximum size $O(n)$. The only data structure needed by this algorithm is the stack since Delaunay triangles can be output as they are computed. If we wish to construct the adjacencies for the triangles it is necessary to store with each edge the associated triangle.

If we lift our two-dimensional triangulation problem to a three-dimensional convex hull problem, then the gift-wrapping algorithm [263] is just the circumcircle algorithm in disguise.

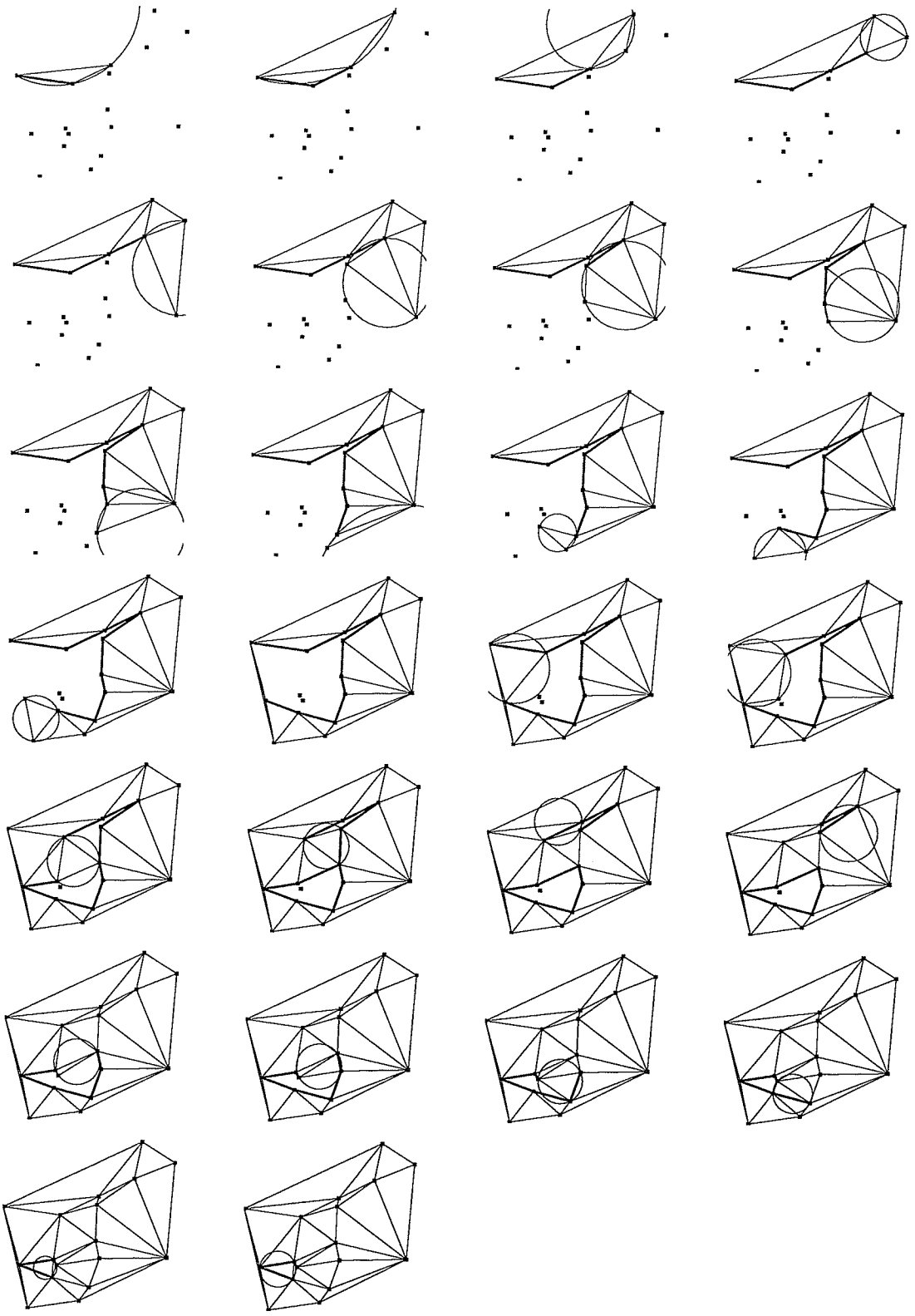


Figure 2.9: Circumcircle algorithm. Stack is in bold

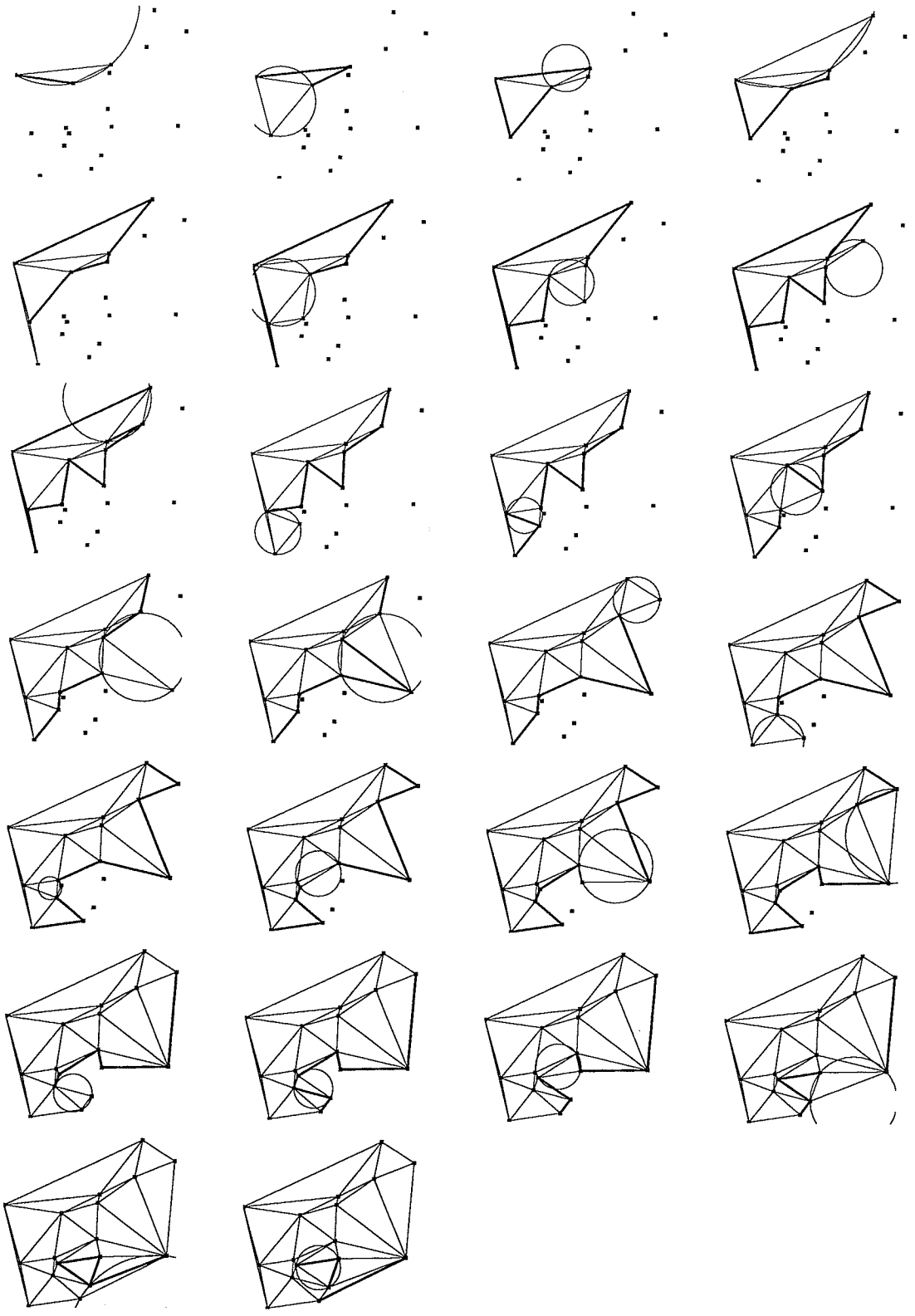


Figure 2.10: Circumcircle algorithm. Queue is in bold

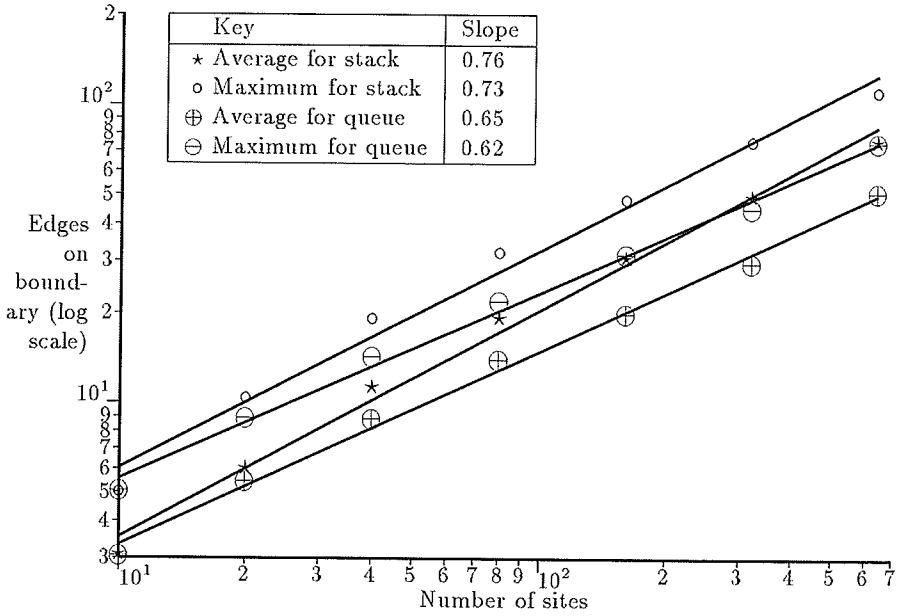


Figure 2.11: Boundary size in circumcircle algorithm (log scale)

Speeding up the circumcircle algorithm

Sorting on x A simple way to speed up the search for the Delaunay triangle is to sort the sites by x coordinate. All the sites in a vertical strip will be contiguous in the sorted list. If C is a candidate site for the third vertex of the triangle on AB, then it is only necessary to test those sites in the vertical strip with sides tangent to the part of $\bigcirc ABC$ on the same side of AB as C. If we find a site inside the circle ABC, then we have a new candidate and the circumcircle and consequently the vertical strip to be checked will shrink.

If we test the sites in the right order it is unnecessary to test sites outside the vertical strip tangent to the actual Delaunay arc[6] on AB. The optimal order is the order in which the sites are touched by the expanding vertical strip (see figure 2.13).

If the sites are uniformly distributed in a unit square then the expected radius of a Delaunay circumcircle is $3/(4\sqrt{n})$ and the expected number of sites searched is less than $\frac{3}{2}\sqrt{n}$. Hence the total search time to construct the Delaunay triangulation is $O(n^{3/2})$.

Fang and Piegl [112] uses an approach similar to the above, though without the optimum search ordering.

[6] the part of the Delaunay circle on the same side of AB as C

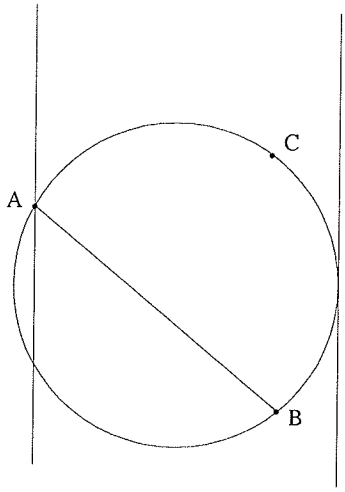


Figure 2.12: Strip tangent to the part of $\bigcirc ABC$ on the same side of AB as C

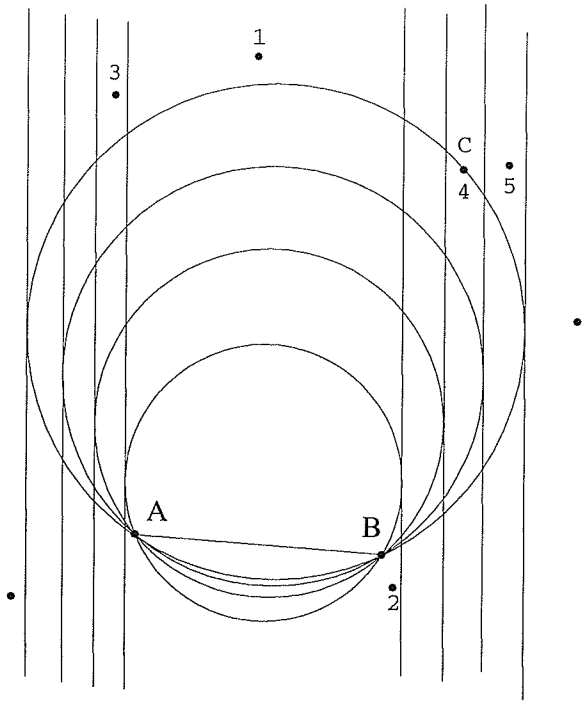


Figure 2.13: Search for a Delaunay triangle using an x sorted list

Bucketing If the sites to be triangulated are uniformly distributed, we can use bucketing to improve the execution time to $O(n)$ in the average case [113, 224, 227, 307].

We need to use $O(n)$ buckets so that each bucket will contain $O(1) = \rho$ sites on the average. To find the third site of the Delaunay triangle to the right of the edge AB (see figure 2.14) we need to first search the buckets intersected by the edge AB. To ensure that buckets are not searched unnecessarily, the order in which the remaining buckets should be searched is the order in which they are encountered by an expanding arc through the sites A and B. The buckets in figure 2.14 are numbered in the order that they will be searched. We can stop the search when we have searched all the buckets to the right of AB that intersect the circumcircle of the candidate triangle. (In figure 2.14 we would stop after searching bucket 11.)

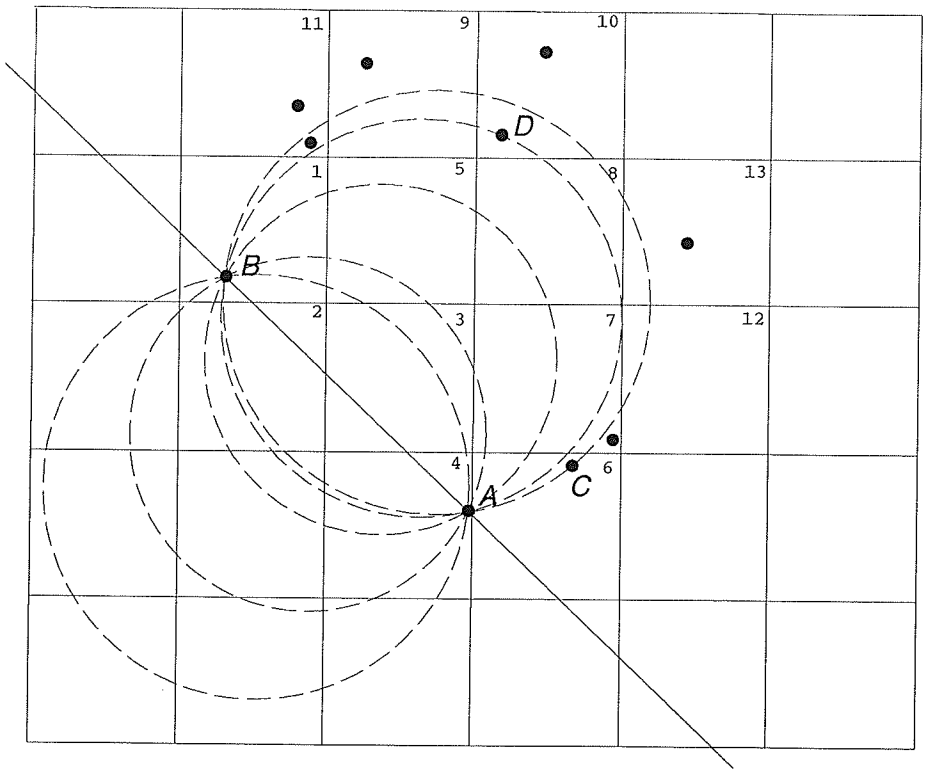


Figure 2.14: Bucket search for a Delaunay triangle

A circle with radius R will contain an average of $\pi R^2 = c$ bucket corners, and intersect an average of $2R = h$ horizontal and $2R = l$ vertical lines. Consider the planar graph formed

by the intersection of the buckets with the circle (see figure 2.15). This has $c+2h+2l=v$ vertices and $(4c+3(2h+2l))/2=e$ edges since the vertices inside the circle have degree 4 and the ones on the boundary have degree 3. Euler's formula says

$$\begin{aligned} f &= 2+e-v \\ &= 2+2c+3(h+l)-(c+2h+2l) \\ &= 2+c+h+l. \end{aligned}$$

(This is one more than the number of buckets, since it counts the exterior face.)

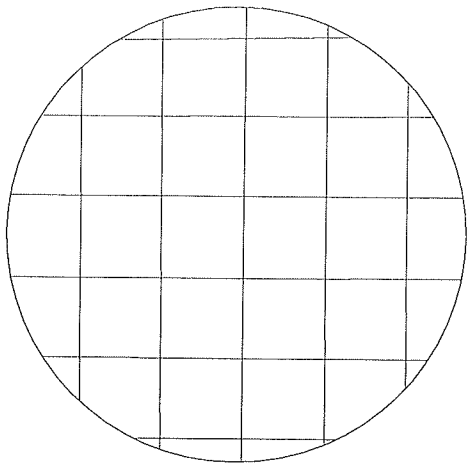


Figure 2.15: Graph formed by intersection of buckets with a circle

Miles [232] proves that the the moments of the circumradius of a random Delaunay triangle are $E[R^k]=\Gamma(k/2+2)/(\pi\rho)^{k/2}$. Therefore, the average number of buckets intersected by a Delaunay circumcircle will be

$$\begin{aligned} b(\rho) &= E[\pi R^2+4R+1] \\ &= \pi\frac{\Gamma(3)}{\pi\rho}+4\frac{\Gamma(5/2)}{(\pi\rho)^{1/2}}+1 \\ &= \frac{2}{\rho}+\frac{3}{\sqrt{\rho}}+1. \end{aligned}$$

The number of buckets searched will hence be at most $b(\rho)$ and at most $\rho b(\rho)$ sites will be tested. The optimum value of ρ will depend on the amount of time it takes to test a site

and the amount of time it takes to compute the next bucket and test if it is empty. If these times are equal the optimum value is about 1, giving about 6 sites tested and 6 buckets searched.

The total search time for the whole triangulation will hence be $O(n)$.

To ensure that the total running time is $O(n)$ we must make sure that the updating of the stack takes $O(1)$ time. We cannot just search the entire stack for the reverse of the edge to be added, since this could take $O(n)$ time. The stack is just the edges making up the boundary of the triangulated region and almost all sites will appear on this boundary at most once, so we just need to maintain a table that contains for each site a list of pointers into the edges on the stack that it is the first site of.

Maus [224] and McCullagh [227] do not search the buckets in the optimum order defined above. Instead they search all the buckets intersecting a circle through AB. If these are all empty, they try progressively larger circles. Fang and Piegl [113] adopt a similar approach, except that they search the buckets in a bounding square of the circle. They report an optimum value of ρ of 16/9. Tarvydas [307] just searches the nine buckets adjacent to the bucket containing the centre of AB.

2.4.2 Constrained Delaunay triangulation

When searching for a new triangle on an edge AB it is sufficient just to test AC and BC to see if they intersect a constraint edge. This gives a very simple $O(n^2e)$ algorithm. Since e is $O(n)$ this is $O(n^3)$ in the worst case. A simple improvement, implemented by Lo [216] is to first construct the triangles on either side of the constraint edges. Once we have done this it is no longer necessary to test for intersection with the constraint edges, giving time $O(ne^2+n^2)$. In many applications there are only a few constraint edges—if e is $O(\sqrt{n})$ the total time is just $O(n^2)$.

Another possibility, suggested by Lee and Lin [201] is to construct (in time $O(e \log e)$) the visibility polygon from A. We can test in time $\log e$ whether a site is in this polygon. This gives a $O(n^2 \log e)$ algorithm. Lee and Lin also suggest using a $O(n^2)$ algorithm [12] to construct the visibility graph for the entire set of sites, which gives a $O(n^2)$ algorithm, but this is not practical since the space requirement is $O(n^2)$ and the visibility algorithm is much more complicated than the simple Selection triangulation algorithm.

Bucket Search

If the sites are uniformly distributed we can consider using bucketing. The obvious approach is to store each edge in the buckets that it intersects. When we search a bucket, we test each site in the bucket against each constraint edge intersecting that bucket. Furthermore, we do not search buckets on the far side of constrained edges that we find in a bucket.

There is a problem with this approach: it is possible to have $\Omega(n)$ constraint edges, each of which intersects $\Omega(\sqrt{n})$ buckets, leading to $\Omega(n^{3/2})$ storage requirements. Furthermore, even if we could, by some clever data structure, reduce this to $O(n)$, we have another problem. The constraint edges will divide each bucket up into an average of $\Omega(\sqrt{n})$ regions. When we search a bucket, we will only consider sites in one of these regions. This means that we will have to search $\Omega(\sqrt{n})$ buckets on average to find a site, so the total search time for the algorithm will be $\Omega(n^{3/2})$.

This problem may not be as bad as it seems—most applications will not want to force a lot of long edges, since this guarantees many skinny triangles. In this context a long edge is one that intersects $O(\sqrt{n})$ buckets. If the constraint edges intersect a total of $O(n)$ buckets (*e.g.* $O(\sqrt{n})$ long edges, or O(n) edges that intersect $O(1)$ buckets) then the total storage requirements are $O(n)$ In this case, each bucket is intersected by an average of $O(1)$ constraint edges, so we can still expect to find $O(1)$ candidate sites per bucket which gives $O(n)$ search time and $O(n)$ time to construct the constrained Delaunay triangulation.

This approach is adopted by Piegl and Richard [256] for the slightly less general problem that occurs when the constraints form multiply connected polygons.

2.4.3 Simple polygon Delaunay triangulation

The visibility graph (from a site) in a simple polygon can be constructed in time $O(n)$ [106] and so the search for a triangle standing on an edge will take time $O(n)$ and the circumcircle algorithm will take time $O(n^2)$. In fact, since in this case the Delaunay triangulation divides the polygon into two smaller polygons we can do better than this on average.

Let us consider the two extreme cases for the structure of the triangulation of the polygon: a *thin triangulation* (where the dual graph is a path) and a *bushy triangulation* (where the dual graph is a complete ternary tree).[7]

In the case of a thin triangulation if we randomly choose a side of the polygon to

[7]These names come from Chatopadhyay and Das [46].

construct a Delaunay triangulation then the average time taken satisfies

$$T(n) = n + \frac{2}{n}\sum_{i=1}^{n-1} T(i)$$

giving time $O(n \log n)$. On the other hand, if we always choose the newly constructed side, we get

$$T(n) = n + T(n-1),$$

giving a total of $O(n^2)$.

In the case of a bushy triangulation if we randomly choose a side we always get a worst-case split and take time $O(n^2)$. On the other hand, if we always choose the newly constructed side, in $O(\log n)$ steps we get to the middle of the triangulation and after that, each Delaunay triangle that we find divides the polygon exactly into two, so run time is $O(n \log n)$.

In the case where the triangulation is a random one from all possible polygon triangulations, the run time is $O(n^{3/2})$ (see section 5.4.1.)

If we have no *a priori* information about which sorts of triangulations are most likely it would seem best to alternate between a randomly chosen side and a newly constructed side.

2.4.4 Convex-Polygon Delaunay triangulation

This is similar to simple polygon Delaunay triangulation except that it is no longer necessary to test for intersections with polygon edges. See section 5.4.1 for more details.

2.4.5 Convex-Distance-Function Delaunay triangulation

If the convex-distance-function ball is smooth and bounded then the selection algorithm will work in the same way that it does in the Euclidean metric. If not, two difficulties are encountered:

1. The dual graph of the triangulation may not be connected.

2. The triangulation may contain edges that are not part of any triangle.

In section 4.1.2 I present a way to deal with these difficulties.

2.5 Sweepline Triangulation Algorithms

2.5.1 Delaunay triangulation

Fortune's sweepline algorithm for the Voronoi diagram [122] works by deforming the plane so that each site appears at the bottom of its Voronoi region. Guibas and Stolfi [149] give a three-dimensional interpretation. An infinite cone extending upwards is placed at each site so that the view from below is just the Voronoi diagram and space is swept with a plane that makes the same angle with the base plane as the cones.

We can think of Fortune's algorithm as a selection Delaunay triangulation algorithm in the following way: The circumcircle algorithm extends the triangulated region by searching through all the sites for each edge on the boundary of the triangulated region. The sweepline algorithm makes one pass from left to right over the sites. As each site is encountered, it is connected to the appropriate part of the boundary. Each triangle is added as soon as we are sure that its circumcircle is empty.

We will maintain the subset of the Delaunay triangulation that is guaranteed to be present no matter what sites are to the right of the sweepline. We will be able to include an edge if we can find an empty circumcircle touching that edge. Let AC be the Delaunay edge shared by Delaunay triangles ABC and ACD (see figure 2.16).

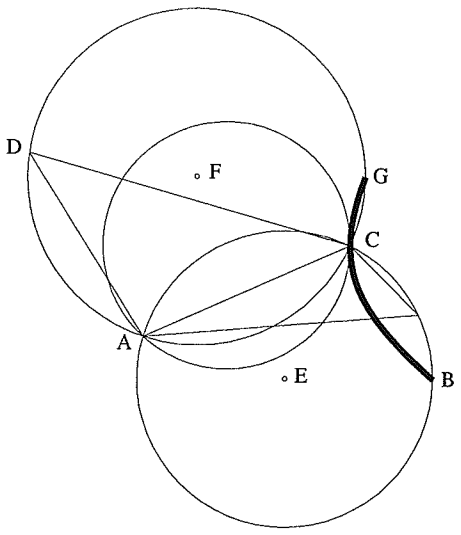


Figure 2.16: Type I edge AC is added when sweepline reaches C

Consider the pencil of circles that pass through the sites A and C. The locus formed by the rightmost points of these circles is a branch of a rectangular hyperbola[8] with leftmost point C. The circles whose centres lie on the segment joining the circumcentre of ABC (E in figure 2.16) and the circumcentre of ACD (F in figure 2.16) are the only empty ones. The rightmost points of these empty circles form a segment of the hyperbola (the part whose y coordinates are between those of E and F). We are interested in the first empty circle that the sweepline passes over. This is the one corresponding to the leftmost point of the hyperbola segment. There are two possibilities:

Type I edge The y coordinate of C is between that of E and that of F (figure 2.16). The edge AC must be added when the sweepline reaches C.

Type II edge The y coordinates of E and F are both greater or both less than C's (figure 2.17). The edge AC must be added when the sweepline reaches the rightmost point of the circumcircle ACD.

From the above it might seem that we need to know the entire Delaunay triangulation in order to determine the edge to add next, but in fact we only need to keep track of the boundary of the partial triangulation, those edges that could potentially have a Delaunay triangle on their right hand side. We can order the edges of the boundary in an anticlockwise order, starting from the leftmost site.

Define STC(AB) (sweep tangent circle) of an edge AB to mean the circle through A and B and tangent to the sweepline at a point to the right of AB. (Note that there are two circles through an edge, tangent to a line.) The sweep tangent circle for a boundary edge will be empty, since if it contained a site, a Delaunay triangle to the right of the edge with a circumcircle to the left of the sweepline would exist. The order of the edges around the boundary is the order in which their tangent circles touch the sweepline (see figure 2.18), since if ABC are three consecutive sites on the boundary and the tangent

[8] Put the origin at the midpoint of AC, and let C have coordinates (a, b). The circle centres lie on the line $ax + by = 0$. Let the circle centre be (x_0, y) (so $x_0 = -\frac{b}{a}y$) and its rightmost point be (x, y). Then

$$\begin{aligned} (x - x_0)^2 &= (x_0 - a)^2 + (y - b)^2 \\ (x + \frac{b}{a}y)^2 &= (-\frac{b}{a}y - a)^2 + (y - b)^2 \\ x^2 + 2\frac{b}{a}xy - y^2 - (a^2 + b^2) &= 0 \end{aligned}$$

which is a rectangular hyperbola.

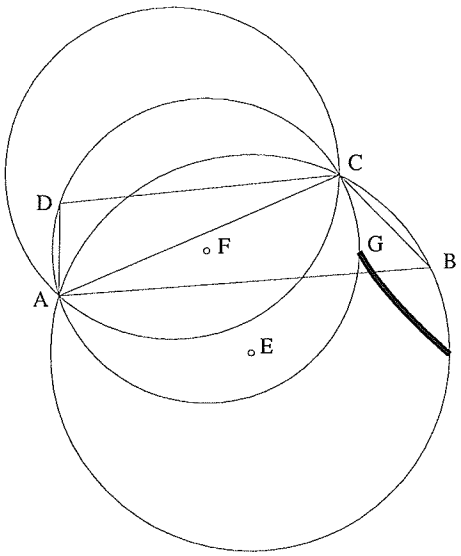


Figure 2.17: Type II edge AC is added when sweepline reaches G

point for STC(AB) is above the tangent point for STC(BC), then $C \in$ STC(AB) (see figure 2.19), which is impossible.

Initially the boundary will contain just the leftmost site. When we add a type I edge AC where C is the new site and A occurs on the boundary in the sequence $\alpha XAY\beta$ (α and β are sequences of vertices), the boundary becomes $\alpha XACAY\beta$ (see figure 2.20). When we add a type II edge AC (see figure 2.17) the edges DC and AD will already have been included in the partial triangulation (since we can find an empty circle entirely to the left of the sweepline that touches CD with a centre just a little above F). The boundary will contain $\alpha ADC\beta$ since all new edges are added to the boundary and edges are only deleted when their Delaunay triangle is found. The boundary becomes $\alpha AC\beta$. At termination, the boundary contains the convex hull of the sites.

If ABC are three consecutive vertices on the boundary and C is to the right of AB (*i.e.* B is a concave vertex), then ABC is a potential Delaunay triangle, and AC can be added to the triangulation when the sweepline reaches the rightmost point of $\bigcirc ABC$ (since then $\bigcirc ABC =$ STC(AB)). We will update the boundary whenever the sweepline reaches a site or the rightmost point of circumcircle. It is sufficient to maintain a priority queue for these events, ordered by x coordinate. The priority queue will contain a site event for each site

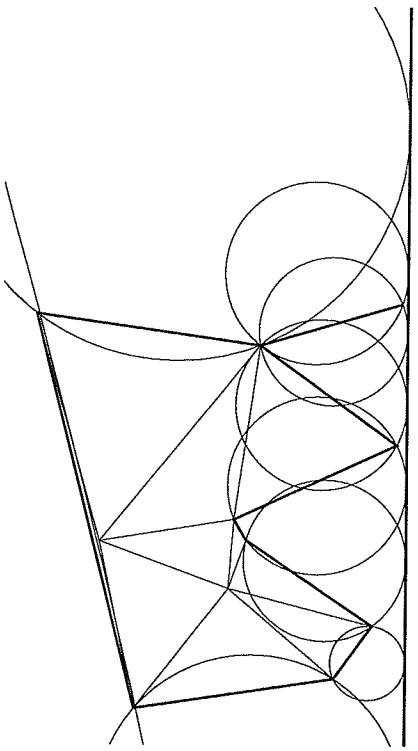


Figure 2.18: Circles through boundary edges tangent to sweepline

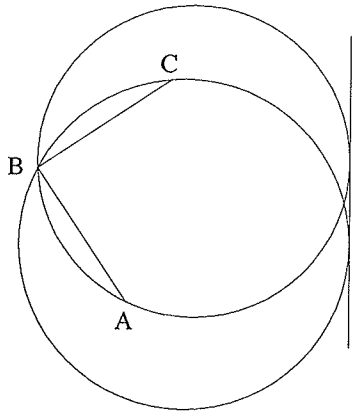


Figure 2.19: Sweep tangent circles in the wrong order

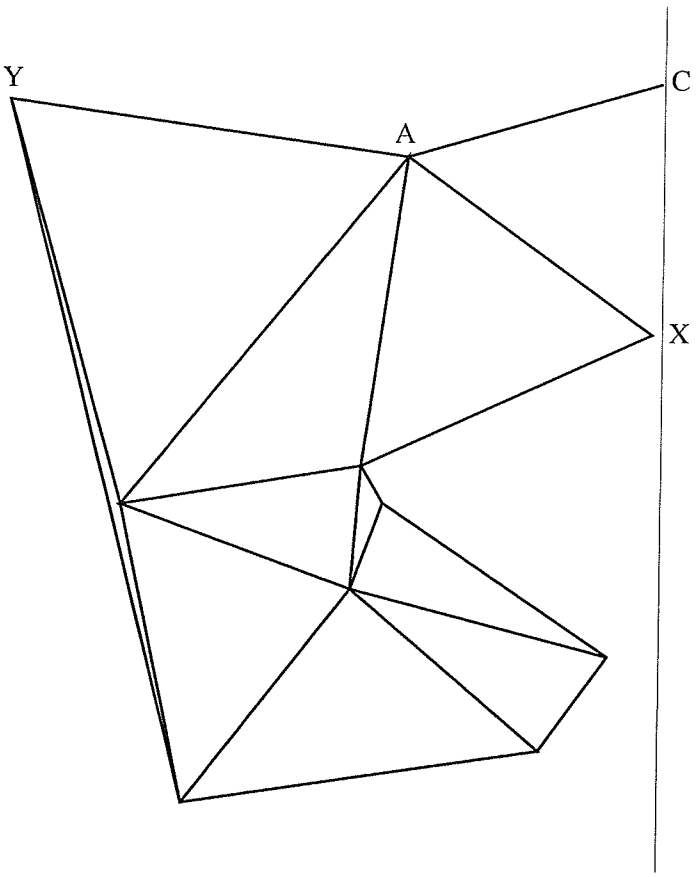


Figure 2.20: Boundary changes from $\alpha X A Y \beta$ to $\alpha X A C A Y \beta$

to the right of the sweepline, and a circumcircle event for concavity ABC on the boundary.

We initialize the queue to contain all the sites except the leftmost. We then proceed by removing an event from the queue.

If this is a circumcircle event for triangle ADC we can output the triangle ADC and update the boundary from $\alpha XADCY\beta$ to $\alpha XACY\beta$. We remove events (if any) for XAD and DCY and add events for XAC (if A is concave) and ACY (if C is concave).

If this is site event for site C, we need to find the boundary site to connect it to. Such a site is guaranteed to exist—consider an expanding circle through C, tangent to the sweepline. The first site that it touches, say A, is the site we want. A could occur on the boundary more than once (see figure 2.21). At the correct spot on the boundary $\alpha XAY\beta$, we have $C \in \text{wedge}(XAY)$.[9]

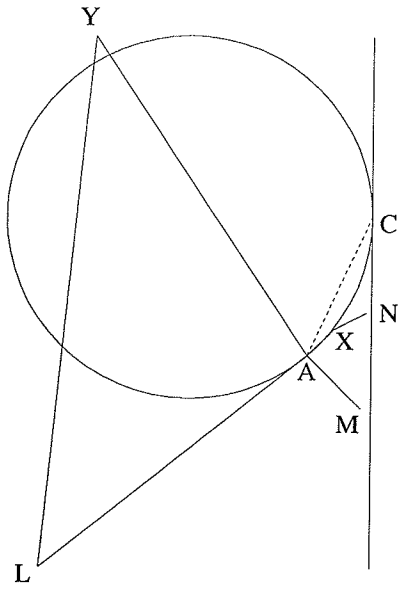


Figure 2.21: Boundary is $LAMAXNXAYL$

We update the boundary from $\alpha XAY\beta$ to $\alpha XACAY\beta$, remove the event (if it exists) for XAY and add events for XAC and CAY (if they are concave).

We could search the entire boundary to find the spot with the desired properties, but

[9] wedge(XAY) is the intersection of the half-plane to the right of XA and the half-plane to the right of AY.

there is a better way. Since the tangent points of the STCs of boundary edges are ordered, we can use binary search to find adjacent boundary edges XA and AY such that C is between the tangent points of STC(XA) and STC(AY). Now, STC(AC) $\subset$ STC(XA)$\cup$STC(AY)$\cup R$ where R is the shaded region in figure 2.22.

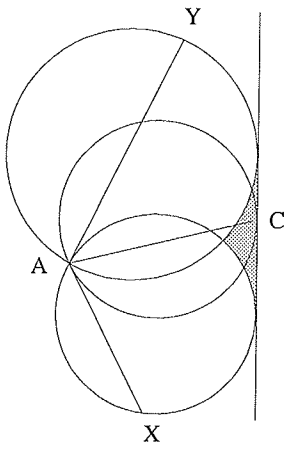


Figure 2.22: CA is a Delaunay edge

If R is empty then STC(AC) is empty and, since the tangent points of STC(XA) and STC(AY) are in wedge(XAY) and C is between them, then $C \in$ wedge(XAY).

If R is not empty, then let C' be the leftmost point in R. The circle through A and C' with a vertical tangent at C' is empty. By the same reasoning as in the previous paragraph, C' would be connected to A when the sweepline reached C', which contradicts XA and AY being adjacent.

When the queue becomes empty, we will have constructed the Delaunay triangulation and the boundary will contain the convex hull of the sites.

Figures 2.23 and 2.24 show the sequence of events that occur. Circumcircle events added to the queue are shown as dashed circles, while those deleted are shown as dotted circles.

Efficiency of Sweepline

The total number of events that occur is equal to the number of edges in the Delaunay triangulation, which is $O(n)$. The boundary is a subset of the Delaunay triangulation, so its size is $O(n)$. The event queue contains at most one event for each site, and one event for

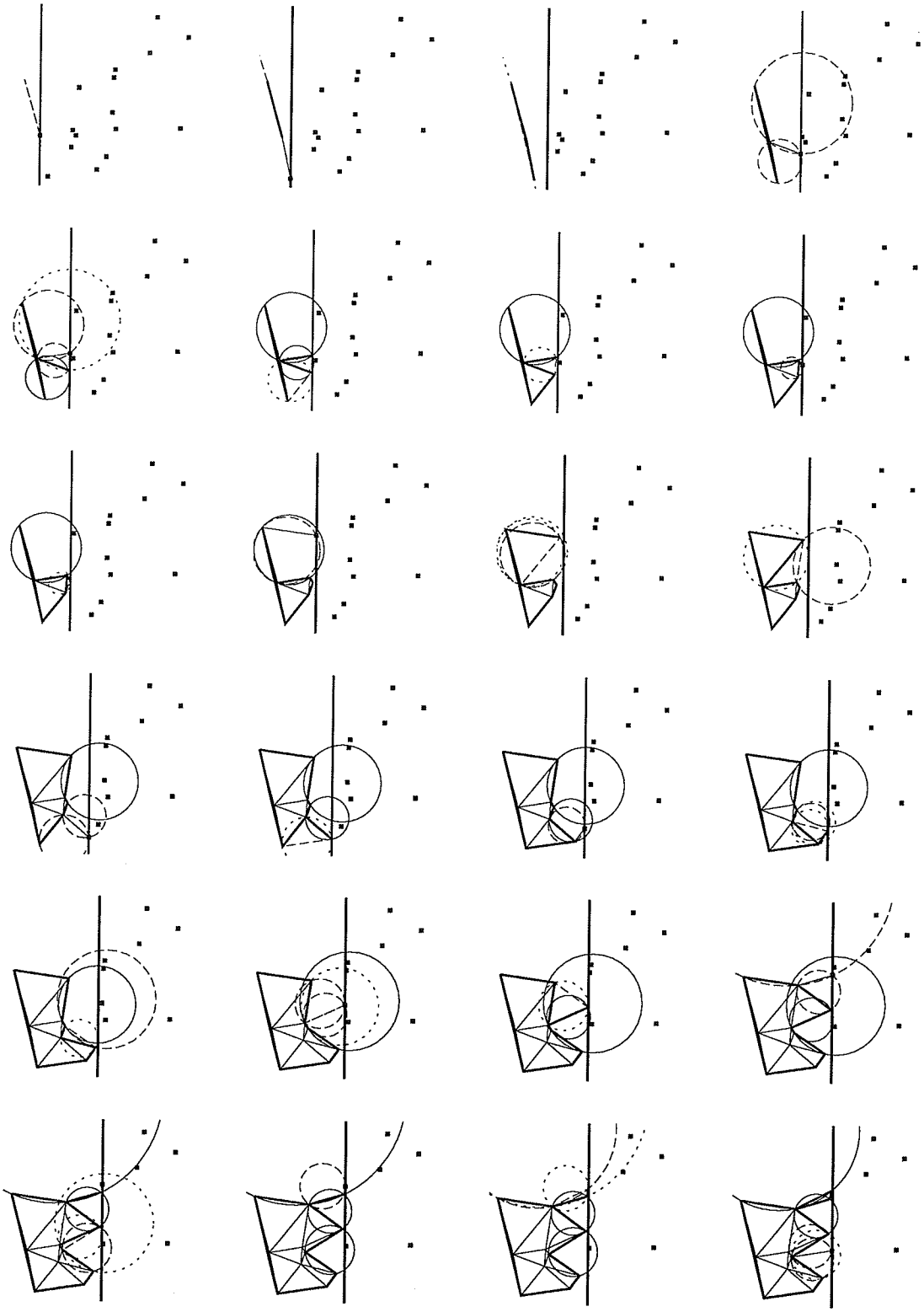


Figure 2.23: Sweep algorithm

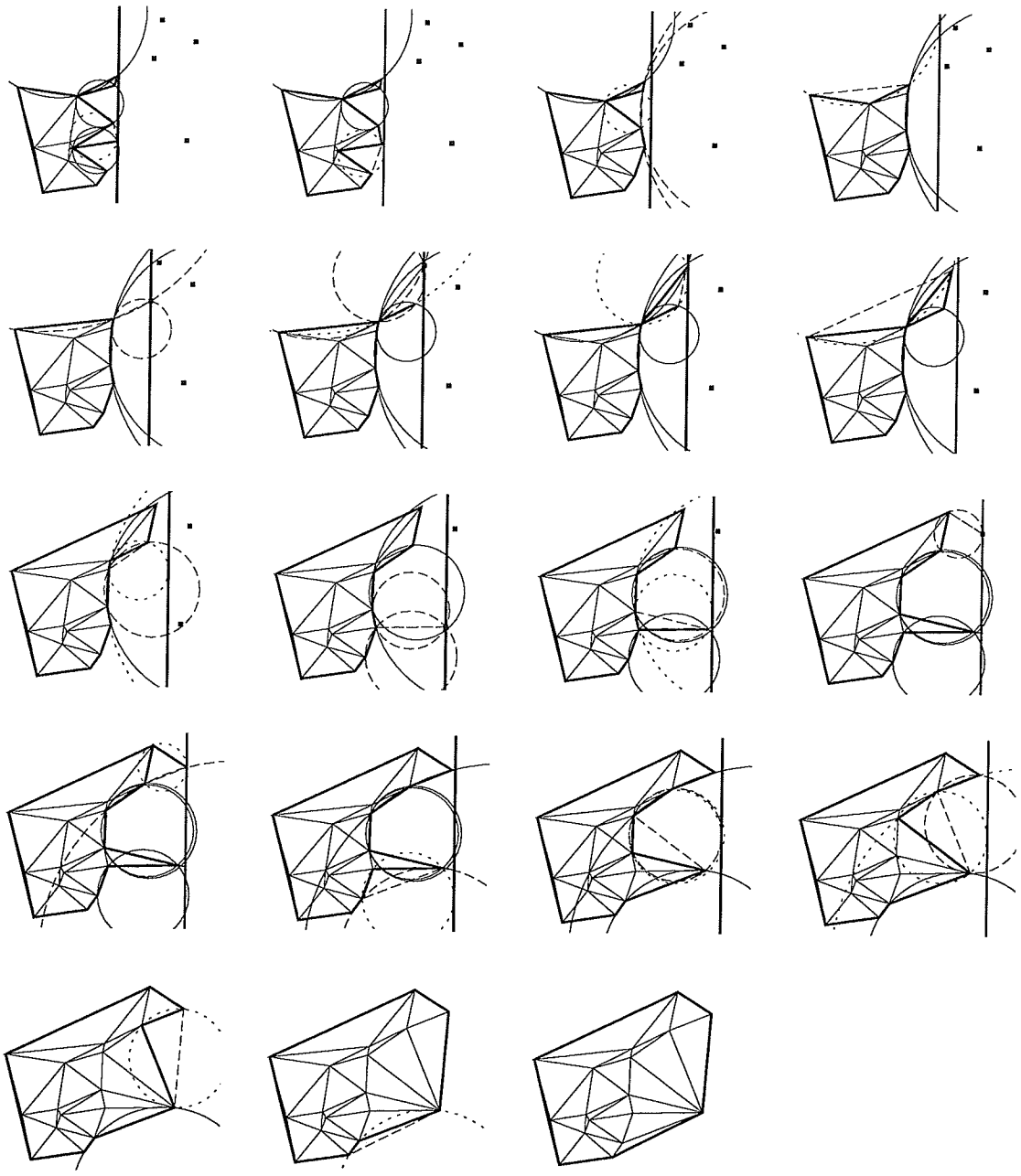


Figure 2.24: Sweep algorithm

each boundary edge, so its size is $O(n)$. By using suitable data structures (for example, a heap for the queue and a balanced tree for the boundary) updates and searches on the event queue and the boundary can be done in time $O(\log n)$. The time to process each event is $O(\log n)$ and the total time to compute the triangulation is $O(n \log n)$.

2.5.2 Constrained Delaunay triangulation

The sweepline algorithm can be modified to compute the constrained Delaunay triangulation. We still maintain a subset of the constrained Delaunay triangulation that is guaranteed to be present no matter what sites are to the right of the sweepline.[10] This obviously includes all the constraint edges. Constraint edges that lie entirely to the left of the sweepline can be treated just like any other edges. Constraint edges that lie entirely to the right of the sweepline can be ignored until the sweepline reaches them. Constraint edges that cross the sweepline divide the area between the partial constrained Delaunay triangulation and the sweepline into a number of regions (see figure 2.25). We will call these constraint edges *active constraints*.

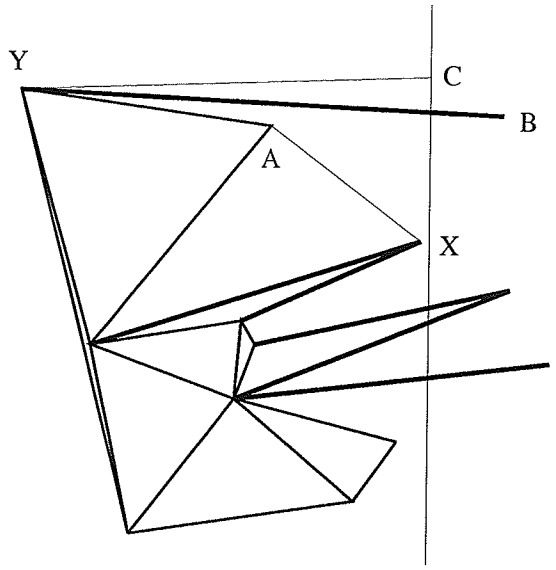


Figure 2.25: Partial constrained Delaunay triangulation—Constraint edges are in bold

[10] Though not the maximal such subset as we will see below.

Definition. If a site is the leftmost point of a constraint edge then that edge is an *incoming edge* of that site. If a site is the rightmost point of a constraint edge then that edge is an *outgoing edge* of that site.

When a site event for a site with no incoming edges is processed the search for the boundary site to connect it to is confined to the piece of the boundary that borders the region the new site belongs to since these are the only sites visible from the new site. For example, in figure 2.25 when processing the site event for C, the boundary to be searched is just XY and C must be connected to Y (compare with figure 2.20).

Each piece of the boundary can be stored as a balanced tree just as in section 2.5.1. If there are k active constraints there will be $k+1$ pieces. These pieces can be organized into a balanced tree with the constraints stored in the internal nodes and the boundary pieces in the leaves. We will call this tree the constraint tree. An in-order traversal of the constraint tree puts each boundary piece between the two constraints that form its top and bottom. The number of constraints is $O(n)$, so the constraint tree only requires $O(n)$ space in the worst case.

The queue is exactly the same as it is in the original algorithm.

Circumcircle events are processed just as in the original algorithm. If the rightmost point of the circumcircle is on the far side of a constraint from the inscribed triangle, then we could be sure that the circumcircle was empty of visible sites before the sweepline reached the rightmost point (see figure 2.26).

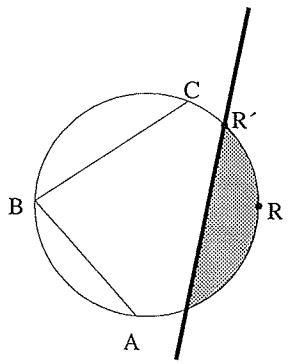


Figure 2.26: ABC can be added to the triangulation when the sweepline reaches R'

If we wanted to maintain the maximal subset of the constrained Delaunay triangulation that is guaranteed to be present no matter what sites are to the right of the sweepline we would have to schedule the circumcircle event when the sweepline passes the rightmost point of the circumcircle modified by removing all the parts on the far side of constraints (the unshaded portion in figure 2.26). However, it is rather difficult to compute the rightmost point of the modified circumcircle since we would have to find the potentially $O(n)$ constraints that intersect it.

Fortunately, this is not necessary. We can schedule the circumcircle event when the sweepline passes the rightmost point of the unmodified circumcircle. The only way this could cause a problem is if we encountered a site inside the circumcircle on the far side of a constraint (in the shaded region in figure 2.26). This site could only cause the removal of the circumcircle event if the new site ended up being connected to one of the vertices of the inscribed triangle. But this is impossible since these vertices do not belong to the piece of the boundary adjacent to the region that the new site falls into.

There are three stages to processing a site event for a site C: first we must deal with the incoming edges for a site, then find a boundary site to connect it to, then deal with the outgoing edges.

Incoming Edges Let i be the number of incoming edges. These edges will occur in succession in an in-order traversal of the constraint tree, so we can find them (and the boundary pieces that they separate) in time $O(i + \log n)$. The in-order traversal of the constraint tree will look like this:

$$\alpha b_0 X_1 C b_1 X_2 C b_2 \ldots b_{i-1} X_i b_i \beta$$

where the b_j denote pieces of the boundary and the X_j are left ends of constraint edges (see figure 2.27). We just need to modify the constraint tree so that in the in-order traversal the above sequence is replaced by the single boundary piece $b_0 C b_i$. Each boundary piece is represented by a balanced tree, so that we can construct this new piece in time $O(\log n)$ by merging two balanced trees by creating a tree with C at the root and b_0 and b_i as its children and rebalancing if necessary. To modify the constraint tree we just need to delete the i incoming edges in time $O(i \log n)$.

Find a boundary site to connect it to If the site has an incoming edge this is unnecessary. Otherwise, the search is confined to the piece of the boundary that borders

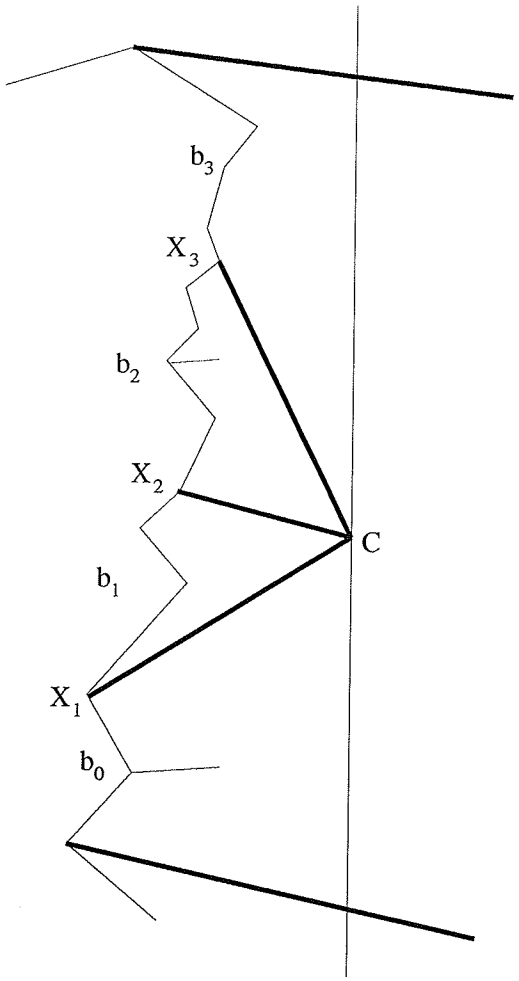


Figure 2.27: Incoming edges for a site event

the region the new site belongs to. We first do a search in the constraint tree to find this piece. To compare the new site to a constraint, we just have to test whether it is above or below. Then we do a search within the piece and modify the event queue just as in the original algorithm. Each of these take time $O(\log n)$ for a total of $O(\log n)$.

Outgoing edges Let j be the number of outgoing edges. Let the boundary piece that C belongs to be $b = b_0 C b_j$. The in-order traversal of the constraint tree will look like this:

$$\alpha F G b D E \beta$$

where FG and DE are the constraint edges that bound b (see figure 2.28).

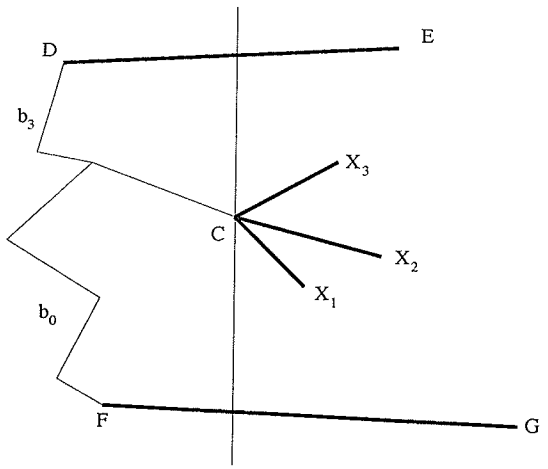


Figure 2.28: Outgoing edges for a site event

The constraint tree must be modified so that its in-order traversal looks like:

$$\alpha F G b'_0 X_1 C b'_1 X_2 C b'_2 \ldots b'_{j-1} X_i b'_j D E \beta$$

where $b'_0 = b_0 C$, $b'_j = C$ for $0 < k < j$, and $b'_j = C b_j$. We need to split the boundary piece b into two pieces at C. This can be accomplished in time $O(\log n)$ [185]. We need to insert j constraints into the constraint tree which will take time $O(j \log n)$.

The total time to process a site event is therefore $O(\log n + (j + o) \log n)$. The total time to process all the site events is $O(n \log n)$ since the total number of incoming and outgoing

edges is $O(n)$.

The efficiency of processing site events can be improved somewhat by noticing that if a site has incoming and outgoing edges we merge two boundary pieces and then split them apart at the same spot. This is unnecessary, so we only need to merge when a site has incoming but no outgoing, and split when it has outgoing edges but no incoming edges.

The total time to construct the constrained Delaunay triangulation is $O(n \log n)$ just as in the original algorithm.

Seidel [291] gives another sweepline algorithm for constrained Delaunay triangulation. He proceeds by modifying Fortune's algorithm [122] for the Voronoi diagram to compute the constrained Voronoi diagram. It is simplest to think of it as operating in a base plane (which holds the sites), plus one plane per constraint edge. Each extra plane is glued to the base plane along its constraint edge so that crossing the constraint edge goes from one plane to the other. All the planes are swept in parallel and by appropriate data structures each event can be processed in time $O(\log n)$, giving a $O(n \log n)$ algorithm.

2.5.3 Simple polygon Delaunay triangulation

Some simplification of the sweepline algorithm for constrained Delaunay triangulation (section 2.5.2) is possible if the constraints form a simple polygon—it is not necessary to compute the part of the triangulation inside the polygon and each site has exactly two incident constraints.

2.5.4 Special polygon Delaunay triangulation

If the polygon is monotone, each vertex has exactly one incoming and one outgoing edge. This means that we will never have to search the boundary for a site to connect a new edge to, so we do not need a fancy data structure to store the boundary. Nor do we need to implement the geometric primitive that finds the tangent point of a sweep tangent circle. Furthermore, we no longer need to schedule site events so that they are processed when the boundary is correct. Instead, we can just process all the site events first. This amounts to setting the boundary (which can now be represented by a double-linked list) to be the polygon. Then we schedule a circumcircle event for each triangle formed by a convex corner of the polygon. To process an event, we cut off the appropriate corner of the polygon, delete up to two events (for corners containing the vertex that has been removed), and schedule up to two events (for the two new corners created). It takes time $O(\log n)$ to process an

event, giving us a very simple $O(n \log n)$ algorithm.

We can use this algorithm to build a $O(n \log n)$ algorithm for simple polygon Delaunay triangulation. Use a sweepline algorithm to divide the polygon into monotone polygons [263], Delaunay triangulate each monotone polygon with the above algorithm, and merge the triangulations together with technique used in the Divide-and-Conquer algorithm. Each of these steps takes time $O(n \log n)$.

2.5.5 Convex-Polygon Delaunay triangulation

Convex polygons are monotone, so the algorithm described in the previous section will work for this case.

2.5.6 Convex-Distance-Function Delaunay triangulation

If the convex-distance-function ball is strictly convex, smooth and bounded then the sweepline algorithm will work in the same way that it does in the Euclidean metric [299]. If not, two difficulties are encountered:

1. The partial triangulation (and hence the boundary) may be disconnected.

2. The sweep tangent circle may not exist.

In section 4.1.2 I present a way to deal with these difficulties. Dehne and Klein [77] also solve this problem, though in a different way by creating two more types of events.

Shute *et al.* [297] give a sweepline algorithm for the Manhattan metric. They use a sweepline parallel to a side of the square ball for this metric. This ensures that the boundary of the partial triangulation is monotone, making site insertion easy. This approach does not generalize to other metrics.

Chang *et al.* [45] give a sweepline algorithm for the oriented Voronoi diagram. This is a convex distance function where the ball is a circle sector, with the origin at the centre. They use a sweepline that reaches the origin of the ball first. This approach does not generalize to other metrics.

2.6 Divide-and-Conquer Triangulation Algorithms

2.6.1 Delaunay triangulation

The hard part about the divide-and-conquer algorithms is the merge step.

While it is possible to merge two arbitrary Delaunay triangulations in linear time [176], the merge step is much simpler if the two triangulations are separated by a line. In this case, all the new edges in the merged triangulation will cross the separating line. We can find these edges in the order in which they cross this line. Given one of these cross edges LR (see figure 2.29), we can quickly find the next one, LR' say. LRR' is a Delaunay triangle, so RR' must be a Delaunay edge in the triangulation of the set of sites to the right of the line. We could find the site R' by searching all the sites adjacent to R and all the sites adjacent to L, but there is a better way.

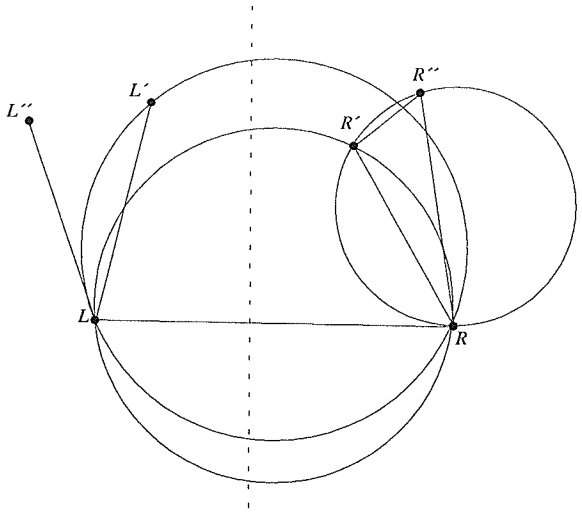


Figure 2.29: Finding the next cross edge

Suppose L, R' and R'' are three successive neighbours of R (see figure 2.29). If R'' is outside $\bigcirc LRR'$ then since $RR'R''$ is a Delaunay triangle of the right hand sites, $\bigcirc RR'R''$ contains none of the right hand sites, and consequently no other neighbour of R could be inside LRR'. If R'' is inside $\bigcirc LRR'$, then RR' cannot be a Delaunay edge of the merged triangulation, so it can be deleted and the next two neighbours of R considered. We can similarly find L' adjacent to L such that $\bigcirc LRL'$ contains none of the left hand sites.

If L' is outside $\bigcirc LRR'$ then $\bigcirc LRR'$ is an empty circle, and LR' is the next cross edge and can be inserted in the triangulation. If L' is inside $\bigcirc LRR'$ then $\bigcirc LRL'$ is an empty circle, and $L'R$ is the next cross edge.

The first cross edge is just the bottom hull edge that crosses the vertical line. Figure 2.30 shows the sequence of partial triangulations as all the cross edges are found.

The number of steps to do the merge is $O(n)$, since there is one step for each new edge added and one step for each edge deleted.

If the sites are sorted by x coordinate, it is easy to divide them into two equal sets separated by a vertical line. Figure 2.31 shows how the triangulations are merged together to form the final triangulation.

This is the algorithm described by Lee and Schachter in [202]. Guibas and Stolfi give a more detailed description in [147].

Efficiency of Divide-and-Conquer

If the sites are uniformly distributed on a unit square, the expected number of Delaunay edges crossing a vertical line is $O(\sqrt{n})$ (see the discussion in section 2.3.1). From this it might appear that (not counting the initial sort), the run time should satisfy the equation $T(n) = O(\sqrt{n}) + 2T(n/2)$ which has solution $T(n) = O(n)$, but this is not the case. A look at figure 2.31 reveals the problem. The typical merge is not along a line splitting a square, but one dividing a rectangle along its long axis and most Delaunay edges will cross this line, leading to $\Omega(n)$ merge steps. Ohya, Iri and Murota [246] report that the divide and conquer algorithm does indeed take time $\Theta(n \log n)$ for $n < 32,000$ uniformly distributed sites.

However, if the splitting is done on both horizontal and vertical lines, Dwyer [92] shows that expected running time $O(n \log \log n)$ is obtained. Katajainen and Koppinen [173] improve this to $O(n)$.

2.6.2 Constrained Delaunay triangulation

Chew [57] generalizes the Divide-and-Conquer algorithm to construct constrained Delaunay triangulations.

Intermediate triangulations that are to be merged together triangulate all the sites in a vertical strip. Constraint edges that cross a vertical strip divide it up into regions. Chew keeps track of only those regions that contain sites (the unshaded regions in figure 2.32),

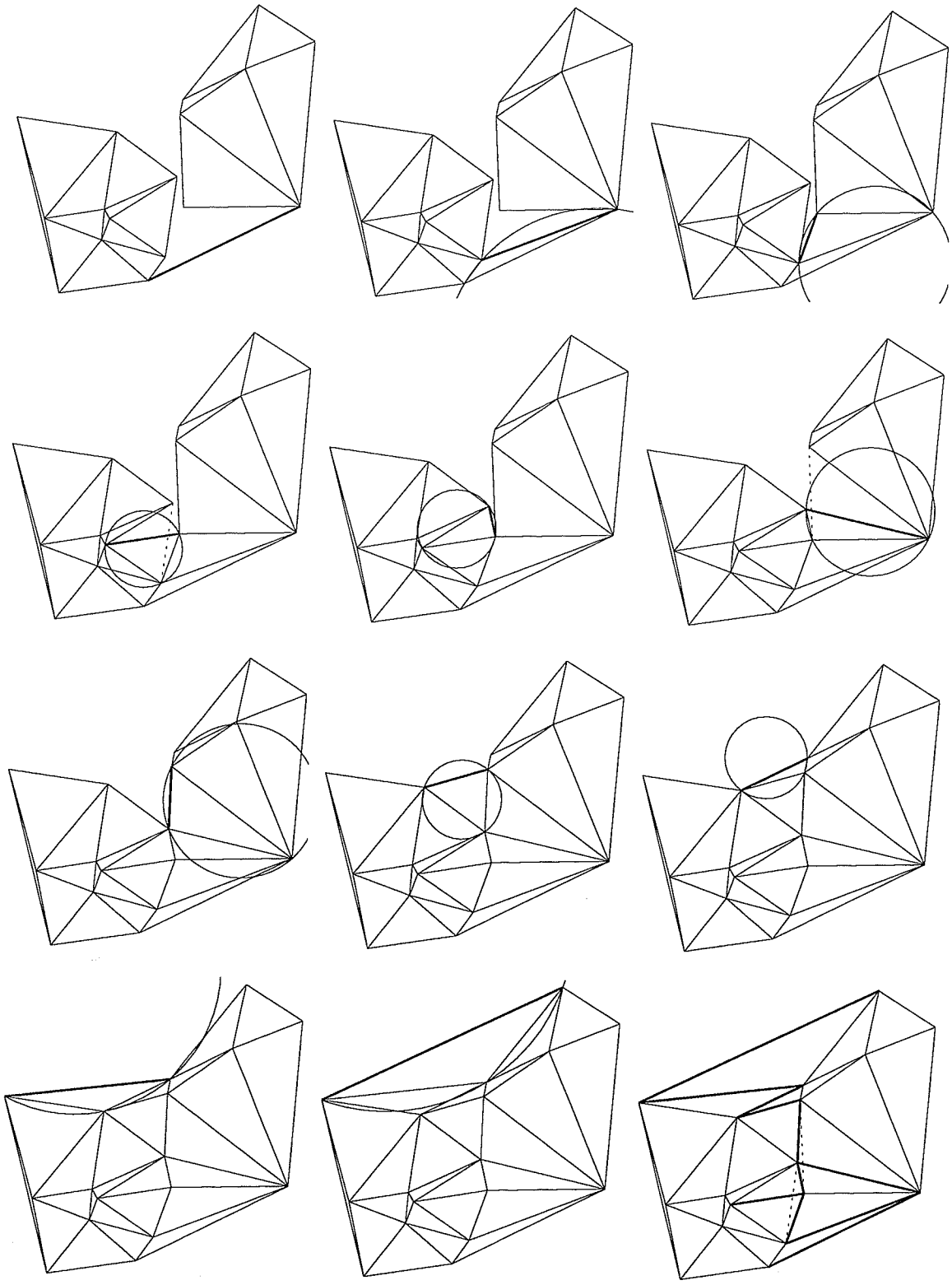


Figure 2.30: Merging two triangulations

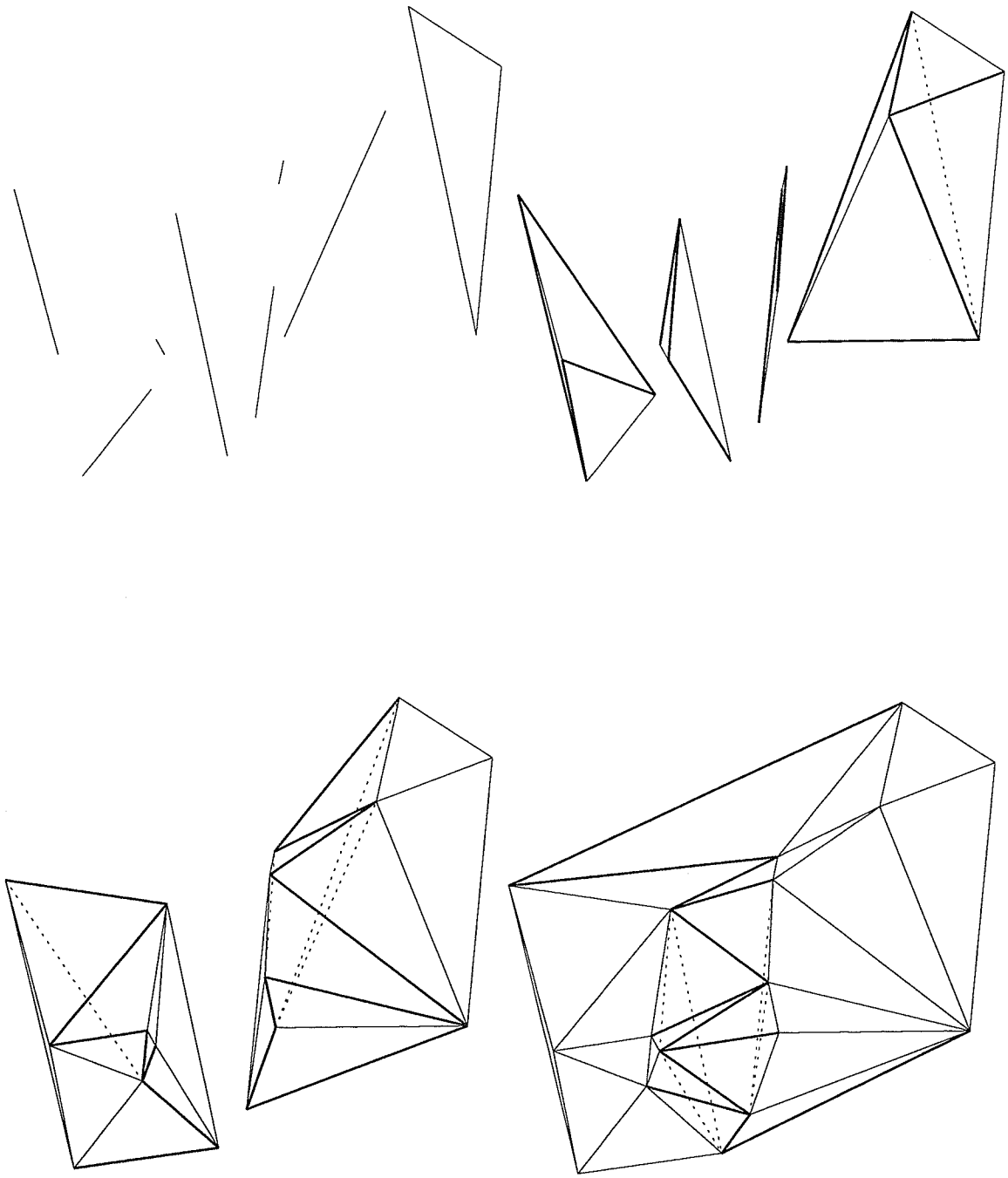


Figure 2.31: Divide-and-Conquer triangulation

since otherwise $\Omega(n^2)$ space could be required.

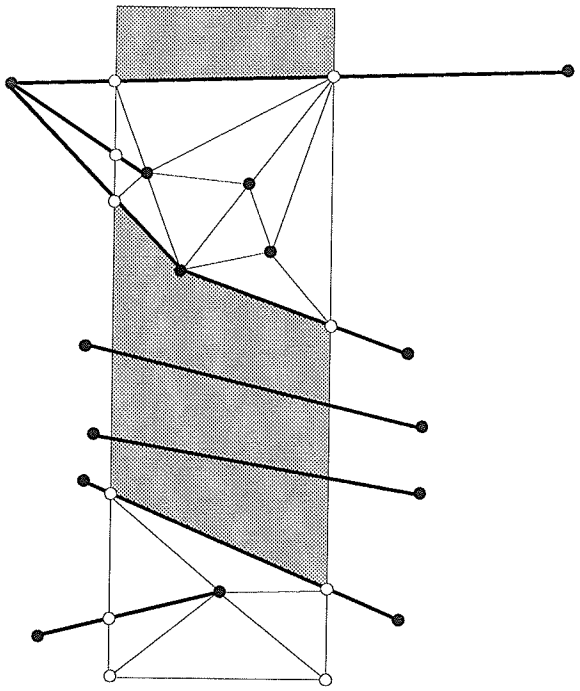


Figure 2.32: A vertical strip. Constraint edges are in bold.

He also adds virtual sites where the constraint edges cross the strip edges. These are the white sites in figure 2.32. The virtual sites are considered to be infinitely far away for the purposes of constructing the triangulation. (So they are never considered to be inside the circumcircle of three real sites.) When two strips are merged, the virtual sites that are on their common edge allow adjacent regions to be matched up. These virtual sites are then eliminated. The adjacent regions are then merged independently, by the same technique as in the original algorithm. Two strips can be merged in time linear in the number of sites they contain, so the algorithm takes time $O(n \log n)$ in total.

Moreau and Volino [237] have implemented Chew's algorithm. They modified the algorithm to remove the virtual vertices. Instead, an AVL tree is used to keep track of the active regions and match them up for merging.

Joe and Wang [165] sketch a Divide-and-Conquer algorithm for the extended constrained Voronoi diagram. This could be used to construct the constrained Delaunay triangulation

by duality.

2.6.3 Simple polygon Delaunay triangulation

A *diagonal* of a polygon is an edge connecting two vertices that is interior to the polygon.

The divide step of a Divide-and-Conquer algorithm will divide the polygon at two vertices. The edge connecting these two vertices must be a diagonal if the two smaller polygons are to be simple.

Chazelle [47] showed that a diagonal of a simple polygon where each subpolygon has $n/3$ vertices can be found in time $O(n)$. Lee and Lin [201] use this result to do the division for a Divide-and-Conquer algorithm. The merging of the triangulated subpolygons is done the same way as in the Delaunay triangulation algorithm. This algorithm takes time $O(n \log n)$.

An alternative to Chazelle's algorithm for diagonal finding is to triangulate the polygon (see section 2.2.2). Using the triangulation it is easy to find in linear time a diagonal where each subpolygon has $n/3$ vertices.

2.6.4 Special polygon Delaunay triangulation

It is easy to find a diagonal in time $O(n)$ that splits a monotone polygon in two roughly equal subpolygons. Yeung [338] used this idea to design a Divide-and-Conquer algorithm for the Delaunay triangulation of convex polygons. Kao and Mount [171] use a similar approach for Delaunay monotone polygons.

2.6.5 Convex-Polygon Delaunay triangulation

Splitting a convex polygon in two can be done in constant time. The merge step could still take $O(n)$ in the worst case, so the divide and conquer algorithm for the Delaunay triangulation of a convex polygon is $O(n \log n)$ in the worst case. Section 5.4.2 analyzes the average time over all triangulations of a convex polygon.

2.6.6 Convex-Distance-Function Delaunay triangulation

If the convex-distance-function ball is smooth and bounded then the Divide-and-Conquer algorithm will work in the same way that it does in the Euclidean metric. If not, two difficulties are encountered:

Reference	n	Flip	Random Incremental	Quaternary Incremental	Sweepline	Divide and Conquer
[120]	10000	59	51		24	42
	50000	317	285		128	222
[199]	2^{19}				164	195
	2^{20}				333	410
[246]	2^{14}			4.2		7.5
	2^{15}			8.5		16.4

Table 2.4: Execution time (seconds) for some implementations

1. It is necessary to start the merge with the bottom support hull edge (instead of bottom convex hull edge).

2. Deleting non-Delaunay edges could cause one of the subtriangulations being merged to become disconnected.

Drysdale [91] presents an algorithm that deals with these difficulties in the case of non-smooth balls.

In section 4.1.2 I present a way to deal with these difficulties in all cases.

2.7 Conclusion

Although the worst-case performance of the incremental, selection and flip algorithms for the Delaunay triangulation is $O(n^2)$, the use of bucketing can give average-case performance of $O(n)$ with selection and incremental algorithms for most site sets, while randomized insertion can attain average-case performance of $O(n \log n)$ for any site set. Furthermore, the flip algorithm tends to require only $O(n)$ flips when measured on sample data. That is, all five classes of algorithms have comparable asymptotic performance. Table 2.4 summarizes some measurements of the actual performance implementations of some of the algorithms on uniformly distributed data.

Needless to say, the numbers are not comparable across the boxes of table 2.4—only the relative sizes are important since the measurements are for different computers at different times.

The actual execution times are roughly comparable—any of the algorithms should be suitable for triangulating large site sets. The quaternary incremental is one of the simplest

to implement and one of the best performers.

It would be interesting to make similar measurements for some of the other Delaunay triangulation problems such as constrained Delaunay triangulation.

Chapter 3

Local Optimization of Triangulations

3.1 Introduction

3.1.1 Optimal triangulations

There are many different possible triangulations of a set of sites. Which one is optimal will depend on the application. For example:

- If the triangulation is to be used as finite-element mesh we wish to avoid ill-conditioned equations. This means that we wish to avoid triangles with angles close to 180° [17].
- If we are using the triangulation to linearly interpolate functions with a bounded second derivative, then the error is minimized by minimizing the maximum circumradius of any triangle [250].
- If the triangulation represents a three-dimensional surface which is to be rendered on a raster display, then we want to avoid triangles less than one pixel wide as these can cause undesirable artifacts [118].
- If the triangulation contains no obtuse angles then it can be used to discretize partial differential equations in a way such that the resulting matrix is a Stieltjes matrix, a desirable property for computation and theoretical analysis [18].

Many alternative definitions of optimality can be found in table 3.1. The only point of agreement seems to be that a triangulation consisting entirely of equilateral triangles is

optimal.

This does not necessarily apply to two-and-a-half-dimensional triangulations, where a triangulated surface is embedded in three dimensions. These are often used for scattered-data interpolation in $\mathbf{R}^2$, where a function value is specified at a set of sites and must be interpolated.

Nadler [240] showed that the best shaped triangles for best l_2 approximation[1] of a quadratic F by linear polynomials are long in the direction of minimum second directional derivative of F and narrow in the direction of maximum second directional derivative of F.

Dazevedo and Simpson [71] showed that the triangulation that gave the best l_∞ approximation was a convex-distance-function Delaunay triangulation where the "circle" was an ellipse with the long axis in the direction of minimum second derivative. Rippa [273] extended this to the best l_p approximation, where $1 \leq p \leq \infty$.

Dyn, Levin and Rippa [93] consider several optimality conditions including minimizing the angle between the triangle normals (also considered by Choi *et al.* [59]) and minimizing the jump in the normal derivatives. Quak and Schumaker [265] try to minimize the energy (thinking of the surface as an elastic membrane). Brown [38] computes a surface normal at each vertex by taking the average of the normals of the adjacent triangles and then tries to minimize the sum of the squares of the angles between vertex normals and adjacent triangle normals.

There are also definitions of optimality applicable to two-dimensional triangulations where it is permitted to add extra (Steiner) points and to higher-dimensional triangulation. See Bern and Eppstein's survey [25] for more details.

3.1.2 Systematic Triangulations

Definition. A triangulation has the *systematic property* if it depends only on the positions of the sites, not on the order in which they are presented.

That is, the triangulation is a well defined function from sets of sites to triangulations. If a triangulation is not systematic, the results of a computation using the triangulation can be difficult to reproduce.

[1] An l_2 approximation minimizes the integral of the squares of the deviations of the approximation from the true function.

In a survey on triangulations Watson and Philip [328] found only three systematic triangulations:

- the *Minimum Weight triangulation* (MWT) which is the triangulation which minimizes the total edge length of all triangles [215].
- the *Greedy triangulation* which is formed by considering edges in order from shortest to longest and adding them to the triangulation if they don't intersect any edge already present [223].
- the Delaunay Triangulation, the dual of the Voronoi diagram [263].

3.1.3 Local triangulations

Definition. A triangulation has the *local property* when the only edges added when a site is added to the triangulation are those adjacent to the new site.

The Delaunay triangulation is the only one of the three triangulations mentioned above with the local property.

The local property is particularly useful for surface-fitting applications—adding a new vertex will produce only local changes to the surface [72].

The local property of the Delaunay triangulation is also the key to some of the fast algorithms for its computation. The incremental algorithm [202] will make at most $O(n)$ changes to the triangulation when adding a new site, giving a worst case of $O(n^2)$ time to compute the Delaunay triangulation. Guibas, Knuth and Sharir [148] show that the average[2] number of sites adjacent to a new site is $O(1)$, so the incremental algorithm will make an average[2] of $O(n)$ changes to the triangulation.

The divide-and-conquer algorithm [202, 147] for the Delaunay triangulation also depends on the local property, since it guarantees that the new edges added when merging two triangulations separated by a line are just those that cross the line and of course are ordered by their intersection with that line.

In contrast, the Minimum Weight Triangulation does not possess any sort of local property. Kirkpatrick [177] shows that it is possible for the MWT of n sites to share no edges (apart from hull edges which every triangulation must share) with each of its $n-1$ site subsets.

[2]The average here is taken over all $n!$ insertion orders.

3.1.4 Locally Optimized Triangulations

A flip rule determines which triangulation of a set of four sites is optimal. If the sites form a convex quadrilateral, this amounts to making a choice between the diagonals.

Given a flip rule, a locally optimized triangulation (LOT) is one where each quadrilateral formed by adjacent triangles is optimally triangulated. It can be constructed by the flip algorithm, which repeatedly flips the diagonals of non-optimal quadrilaterals.

A flip rule is systematic (local) if its LOT is systematic (local). If a flip rule is systematic (local) then the flip algorithm generates a unique triangulation (does $O(n^2)$ flips in the worst case).

If a triangulation is locally Delaunay then it is globally Delaunay [298], so the flip rule that selects the Delaunay triangulation of the quadrilateral (DT) is systematic and local. Nielson and Franke [243] claimed that the flip rule "choose the triangulation that minimizes the maximum angle in both triangles" is systematic. Which flip rules are systematic and local?

Nielson [242] has given a six site counterexample that disproves his claim above. One of the points in his counterexample can be deleted to yield a five site counterexample. The main result of this chapter is to prove that such a five site counterexample exists for *any* flip rule that is not a generalization of the Delaunay triangulation flip rule.

3.2 Flips

Lawson [194] introduced the idea of local optimization of a triangulation.

Definition. In a triangulation, if two adjacent triangles, ABC and ACD, form a convex quadrilateral $ABCD$[3] it is possible to perform a *flip* (figure 3.1) and replace the diagonal AC with BD to get the triangles ABD and BCD.

Any triangulation can be transformed into any other triangulation on the same set of sites by a sequence of flips [193, 319].

Definition. The *flip graph* of a set of sites: The nodes consist of all possible triangulations of that set. Two nodes are connected by an edge if one can be transformed into the other by a single flip.

[3]Throughout this thesis it is assumed that the vertices of quadrilaterals and triangles are given in anti-clockwise order.

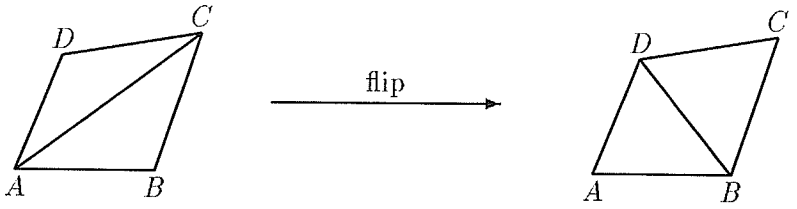


Figure 3.1: A flip

Figure 3.2 shows the flip graph for a set of seven sites.

Definition. A *flip rule* is a function $F(ABCD) \rightarrow \{AC, BD, \textbf{either}\}$ where $ABCD$ is a quadrilateral, which tells us whether the triangulation should include the diagonal AC or the diagonal BD. We must have $F(ABCD) = F(BCDA) = F(CDAB) = F(DABC)$.

For example, Mirante [233] suggested a flip rule to select whichever of AC and BD is shorter. The flip rules that have been proposed (see table 3.1) are invariant under translation, scaling, and (usually) rotation and reflection of $ABCD$. (This is why we must allow the value **either** for a flip rule, for if $ABCD$ is a square, a rotation of 90° maps diagonal AC to BD.)

A flip rule that frequently returns the value **either** is not very useful.

Definition. either(F) is the set of quadrilaterals (regarded as points in $\mathbf{R}^8$) that flip rule F returns **either** for, and similarly for $AC(F)$ and $BD(F)$.

For example, if F is the flip rule "choose the shorter diagonal" then

$$AC(F) = \{(x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4) | (x_1 - x_3)^2 + (y_1 - y_3)^2 > (x_2 - x_4)^2 + (y_2 - y_4)^2\}.$$

If F is a good flip rule then $AC(F)$ and $BD(F)$ should be open sets and **either**(F) the boundary between them.

Often a flip rule can be defined by a function $f : \mathbf{R}^8 \rightarrow \mathbf{R}$ such that $AC(F) = \{\mathbf{x} | f(\mathbf{x}) > 0\}$ **either**$(F) = \{\mathbf{x} | f(\mathbf{x}) = 0\}$. For example, with the "choose shorter diagonal" flip rule we can use

$$f(x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4) = (x_1 - x_3)^2 + (y_1 - y_3)^2 - (x_2 - x_4)^2 - (y_2 - y_4)^2.$$

If f is continuous then $AC(F)$ will be an open set

For the rest of this thesis we shall assume that the sites are "in general position" with respect to a good flip rule and it never gives the value **either**.

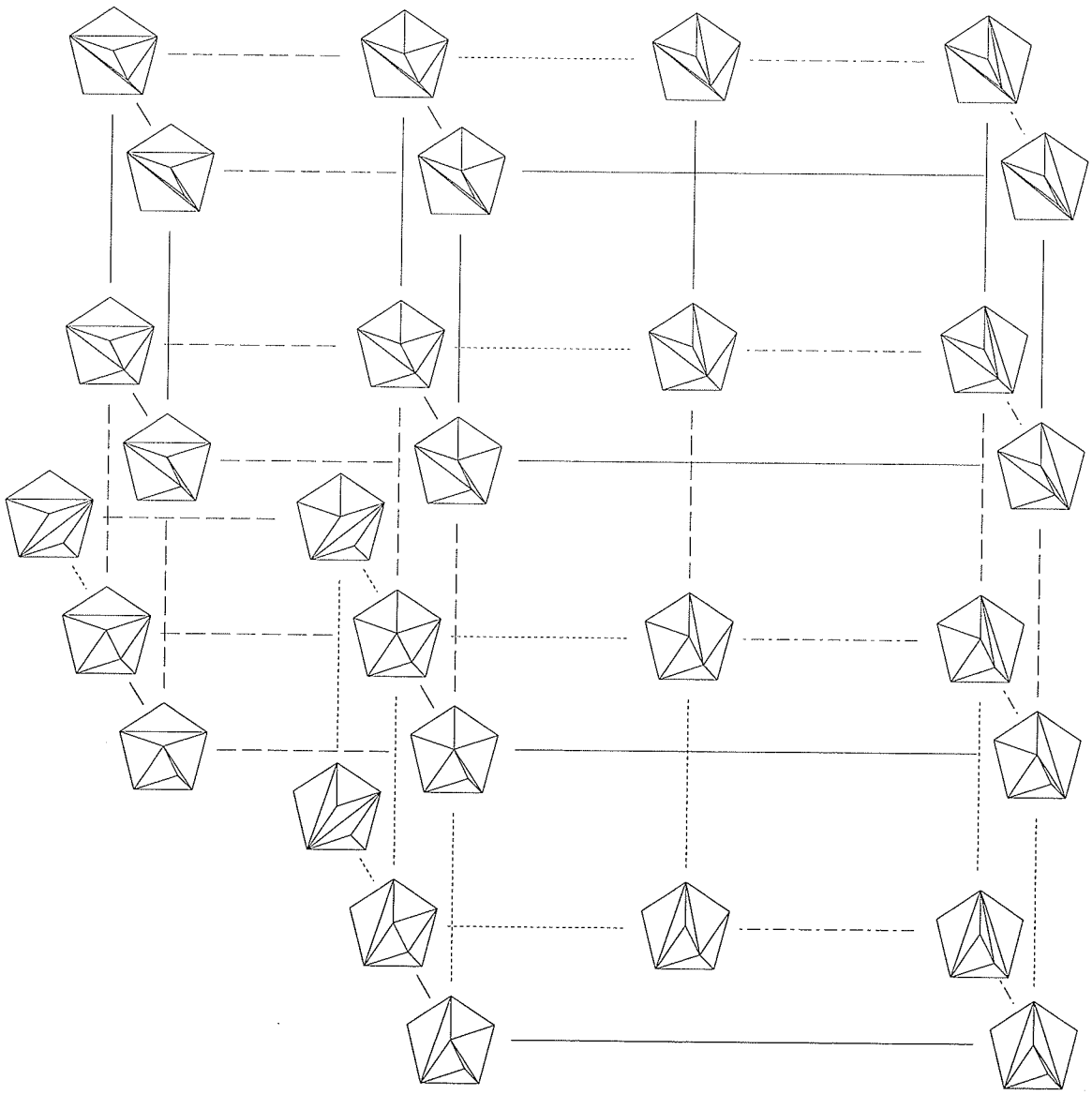


Figure 3.2: Flip graph for a seven site set

A flip rule amounts to putting directions on the edges of the flip graph, pointing to the triangulation that the flip rule prefers. This produces a *directed flip graph*. Figure 3.3 shows a directed flip graph using the "choose the shorter diagonal" flip rule.

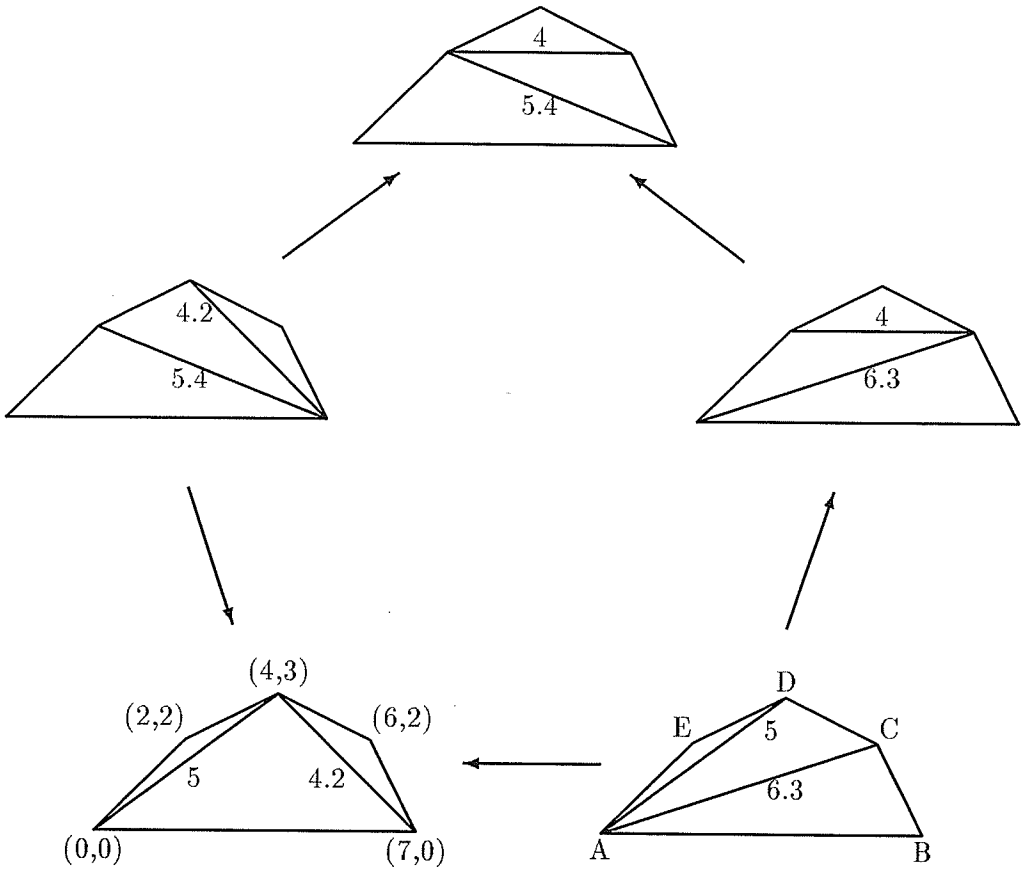


Figure 3.3: Directed flip graph using shorter diagonal

Note that a flip rule cannot assign arbitrary directions to all the edges of the flip graph, since the same convex quadrilateral can occur in many different triangulations. In figure 3.2 parallel lines with the same dash pattern correspond to flips of the same quadrilateral, so must have the same direction.

Definition. A *locally optimized triangulation* (LOT) with respect to a flip rule F is one where the flip rule would not change any diagonal.

This corresponds to a sink (a node with no outgoing edges) in the directed flip graph (*e.g.* the top and bottom left triangulations in figure 3.3). Of course, with a bad choice of flip

rule the flip graph might not have a sink and a locally optimized triangulation will not exist.

The main reason for studying LOTs is that very simple algorithms exist for computing them.

Definition. The *flip algorithm* for constructing a LOT takes an initial triangulation and repeatedly performs flips using the flip rule until no more are possible [207, 233].

The initial triangulation could be part of the problem definition or might have to be computed by some other method.

Definition. The *incremental algorithm* adds sites to the triangulation one site at a time. The new site is connected to all the sites visible from it. The flip algorithm is then used to construct a LOT of all the sites considered so far. [194, 202]

A program using either of these algorithms will go into an endless loop if there is no locally optimized triangulation. In the next section we will prove that LOTs exist for a broad class of flip rules.

3.3 Triangle-Based Flip Rules

There have been many different flip rules proposed (see table 3.1). All except the last two rules in the table fall into the class of triangle-based flip rules.

Definition. A *badness function* is a function $b(ABC) \rightarrow \mathbf{R}^+$ or $\mathbf{R}^-$ which measures the "badness" of a triangle ABC, that is, we would prefer a triangle with a small value of $b(ABC)$.

When ABC is an equilateral triangle $b(ABC)$ should be minimized. Unless $b(ABC)$ is a dimensionless quantity scaling can affect its value. For example, if $b(ABC)$ is the perimeter of ABC then halving the length of each side will halve the value of $b(ABC)$. However, if the area of a triangle is kept fixed, the minimum value of the perimeter occurs when the triangle is equilateral. Since, in triangulating a site set, the total area is fixed, there are no difficulties caused by using badness measures that are not dimensionless.

Sometimes the triangle property is maximized for equilateral triangles (for example, the size of the smallest angle). In this case we just use the negation for our badness measure.

There are many possible choices for a badness measure (table 3.2). The only require-

DESCRIPTION	REF	NAME
Let the "badness" of a triangle be the quantity $\lvert a'/a - \sqrt{3/4}\rvert$ where a' is the length of the minimum altitude and a is that of the longest side. Choose triangulation that minimizes maximum badness.	[41]	$(a'/a)_\infty$
Similar to above, except minimize root mean square badness.	[304]	$(a'/a - \sqrt{\frac{3}{4}})_2$
Maximize minimum angle in both triangles	[194]	$-\alpha_\infty$
Minimize maximum angle in both triangles	[19, 243]	γ_∞
Choose shorter diagonal of quadrilateral	[233]	P_1
Maximize the sum of the minimum angles	[43, 294]	$-\alpha_1$
Let "quality" of triangle be the inradius divided by the circumradius. Choose the triangulation which maximizes the harmonic mean of quality.	[209]	$(-r/R)_{-1}$
Similar to above, except maximize geometric mean	[140]	$(-r/R)_0$
Similar to above, except maximize arithmetic mean	[140]	$(-r/R)_1$
Similar to above, except maximize minimum quality	[126]	$(-r/R)_\infty$
Minimize the minimum circumradius	[231]	$R_{-\infty}$
Maximize the minimum inradius	[214]	$-r_\infty$
Minimize the maximum inradius	[288]	r_∞
Maximize the minimum altitude	[138, 332]	$-a'_\infty$
Maximize the minimum value of area of incircle divided by area of triangle	[214]	$(-r^2/\Delta)_\infty$
Maximize the minimum area	[214]	$-\Delta_\infty$
Minimize the arithmetic mean of "sliveriness", the quantity Perimeter^2/Area	[282, 283]	$(P^2/\Delta)_1$
Minimize the maximum "sliveriness", the quantity Perimeter^2/Area	[284]	$(P^2/\Delta)_\infty$
Let the "goodness" of a triangle be the area divided by the square of the perimeter. Maximize the minimum goodness.	[325]	$(-\Delta/P^2)_\infty$
Let triangle "quality" be the area divided by the sum of the squares of the lengths of each side. Maximize the geometric mean of the qualities of each triangle.	[217]	$\left(\frac{-\Delta}{a^2+b^2+c^2}\right)_0$
Choose the triangle with closest circumcentre.	[328]	d_∞
Minimize the standard deviation of the triangle angles	[329]	s_2
Choose the shorter diagonal provided that the minimum angle is above some threshold.	[156]	
Minimize "irregularity", the quantity $\sum_{p=A,B,C,D}(d(p)-6)^2$, where $d(p)$ is the degree of p	[127]	
Let I be the point where the diagonals of $ABCD$ intersect. If AC is the shorter diagonal we select it provided $1/4 < AI/IC < 4$.	[109]	

Table 3.1: Some proposed flip rules

NAME	DESCRIPTION
$-\alpha$	Smallest angle
β	Median angle
γ	Largest angle
a	Length of longest side
$-c$	Length of shortest side
$P = a + b + c$	Perimeter
abc	Product of sides
$-\Delta$	Area
$R = abc/(4\Delta)$	Radius of circumcircle
$-r = -2\Delta/P$	Radius of incircle
$-a' = -2\Delta/a$	Length of shortest altitude
$-r/R$	Ratio of inradius and circumradius
$-\pi r^2/\Delta$	Area of incircle on area of triangle
$-c/a$	Ratio of shortest and longest sides
$-c/P$	Fraction of perimeter taken by shortest side
$a - c$	Difference between longest and shortest sides
$b/(c - a)$	Median side divided by difference in other two sides
$A = a/a'$	Aspect ratio (Largest width divided by smallest width)
$d = \pm\sqrt{R^2 - (c/2)^2} - R$	Eccentricity. Signed distance from circumcentre to triangle. Take the negative root if circumcentre is outside the triangle.
$s = \sqrt{\sum_{i\in\{\alpha,\beta,\gamma\}} (i - 60°)^2/3}$	Standard deviation of the triangle angles.
P^∞	Perimeter measured by l_∞ metric
R^∞	Radius of circumcircle in l_∞ metric
$-r^\infty = -2\Delta/P^\infty$	Radius of incircle in l_∞ metric (see appendix C)

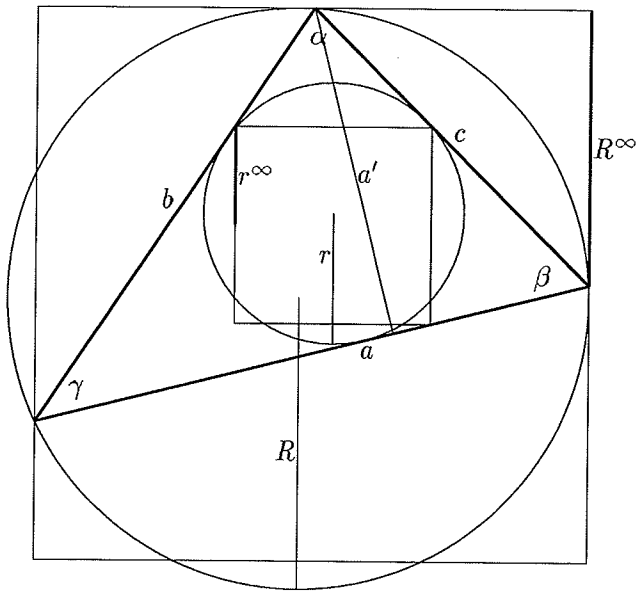


Table 3.2: Some possible badness measures for triangles

ments are that it be unaffected by translation and (usually) rotation and that it be minimized (in the sense mentioned above) for equilateral triangles.

Now we need a way to measure the joint badness of two triangles.

Definition. A *joint function* is a function $f : \mathbf{R}^{+2} \to \mathbf{R}^{+}$ and $\mathbf{R}^{-2} \to \mathbf{R}^{-}$ that is commutative ($f(x,y) = f(y,x)$), associative ($f(x, f(y,z)) = f(f(x,y),z)$) and monotonic (if $x > y$ then $f(x,z) > f(y,z)$).

Definition. A *triangle-based flip rule* b_f is a flip rule

$$b_f(ABCD) = \begin{cases} AC & \text{if } f(b(ACD), b(ABC)) < f(b(ABD), b(BCD)) \\ BD & \text{if } f(b(ACD), b(ABC)) > f(b(ABD), b(BCD)) \\ \textbf{either} & \text{otherwise} \end{cases} .$$

where b is a badness function and f is a joint function.

For example, if we take $b(ABC)$ to be $|AB|+|BC|+|AC|$ (the perimeter) and $f(x,y) = x+y$ we get

$$b_f(ABCD) = \begin{cases} AC & \text{if } |AC| < |BD| \\ BD & \text{if } |AC| > |BD| \\ \textbf{either} & \text{otherwise} \end{cases} ,$$

which is the shorter-diagonal flip rule we encountered earlier.

Definition. If b is a badness function b_* is the set of flip rules b_f where f is any joint function.

For example, R_* is the set of triangle-based flip rules with the circumradius as the badness measure.

Definition. The *total badness* $B(\mathbf{T})$ of a triangulation using b_f consisting of triangles $T_1, T_2, \ldots, T_n$ is $f(b(T_1), f(b(T_2), \ldots, f(b(T_{n-1}), b(T_n)) \cdots))$. Note that this does not depend on the ordering of the triangles.

Theorem 1 *LOTs always exist for triangle-based flip rules.*

PROOF. If the flip rule prefers $\mathbf{T}'$ to $\mathbf{T}$ we can take $\mathbf{T}'$ as $T_1', T_2', T_3, \ldots, T_n$.

$$B(\mathbf{T}') \quad = \quad f(b(T_1'), f(b(T_2'), K)) \quad \text{where } K = f(b(T_3), \ldots, b(T_n)\ldots)$$

$$\begin{aligned} &= f(f(b(T_1'), b(T_2')), K) \\ &< f(f(b(T_1), b(T_2)), K) \\ &= B(\mathbf{T}) \end{aligned}$$

That is, every time a flip is performed the total badness is decreased. Since there are a finite number of triangulations there must be at least one triangulation where no flips will be performed. □

The Minkowski distance function $l_p(x, y) = \pm \sqrt[p]{|x|^p + |y|^p}$ when $p \neq 0, \pm\infty$ satisfies the conditions of a joint function. (We take the sign of l_p to be the same as x and y.) $l_\infty(x, y) = \max(x, y)$ is not monotone, but we can still prove that LOTs exist for flip rules using l_∞.

Theorem 2 *LOTs always exist for flip rules using* l_∞.

PROOF. We order the triangles of $\mathbf{T}$ such that $b(T_1) \geq b(T_2) \geq \cdots \geq b(T_n)$. Define the total badness $B(\mathbf{T})$ to be the sequence $\langle b(T_1), b(T_2), \ldots, b(T_n)\rangle$. We use lexicographic ordering for total badness.

Now, if $\mathbf{T}'$ is preferred to $\mathbf{T}$ by the flip rule and is formed by replacing T_i and T_j $(i < j)$ with T_k' and T_l' $(k < l)$ then $\max(b(T_i), b(T_j)) = b(T_i) > \max(b(T_k'), b(T_l')) = b(T_k')$. Hence, $i \leq k$ so $B(\mathbf{T}') = \langle b(T_1), \ldots, b(T_{i-1}), b(T_{i+1}), \ldots, b(T_k'), \ldots\rangle$. Now, $b(T_i) \geq b(T_{i+1}) \geq \cdots b(T_k) \geq b(T_k')$ but $b(T_i) > b(T_k')$ so at least one of the $\geq$ signs must be a $>$, say $b(T_m) > b(T_{m+1}) = b(T_m')$ and $b(T_k) \geq b(T_k')$ for $k < m$. Hence, $B(\mathbf{T}) > B(\mathbf{T}')$ and by the same argument as before a LOT must exist. □

A similar argument shows that LOTs exist if we use $l_{-\infty}(x, y) = \min(x, y)$. We shall extend our definition of joint functions to include l_∞, $l_{-\infty}$ and l_0. (We can regard them as the limits of the joint function l_p as $p \to \infty$, $p \to -\infty$ and $p \to 0$ respectively.) We will write b_p instead of b_{l_p}. Some possible joint functions are listed in table 3.3. The names of the triangle-based flip rules in table 3.1 can be found in the last column. (The first entry is $(a'/a)_\infty$ instead of $|a'/a - \sqrt{3/4}|_\infty$. The reader can check that these two rules are identical.)

Definition. $LOT(b_f)$ is a locally optimized triangulation using flip rule b_f.

Definition. The flip rule b_f is *systematic* if $LOT(b_f)$ is unique (that is, has the systematic property).

NAME	DESCRIPTION
$l_{-\infty}$	$\min(x, y)$
l_{-1}	$1/(1/\|x\| + 1/\|y\|)$
l_0	$\|xy\|$
l_1	$\|x\| + \|y\|$
l_2	$\sqrt{x^2 + y^2}$
l_∞	$\max(x, y)$

Table 3.3: Some possible joint functions

The example in figure 3.3 showed that $LOT(P_1)$ was not systematic.

Definition. $GOT(b_f)$ is the triangulation with the minimum total badness with respect to flip rule b_f.

GOT stands for globally optimized triangulation. If b_f is systematic then $LOT(b_f) = GOT(b_f)$.

Definition. The flip rule b_f is *local* if $LOT(b_f)$ has the local property.

The Greedy triangulation is a $LOT(P_1)$. The Minimum Weight triangulation is the $GOT(P_1)$. Neither the Greedy triangulation nor the Minimum Weight triangulation has the local property: The Greedy triangulation and the Minimum Weight triangulation of the set $\{A, B, C, D\}$ in figure 3.4 is ABC, BCD. Now we add the site E. The Greedy triangulation and the Minimum Weight triangulation of $\{A, B, C, D, E\}$ is ABE, BDE, BCD. BD is an edge of this triangulation that was not in the triangulation of $\{A, B, C, D\}$, and of course BD is not adjacent to E.

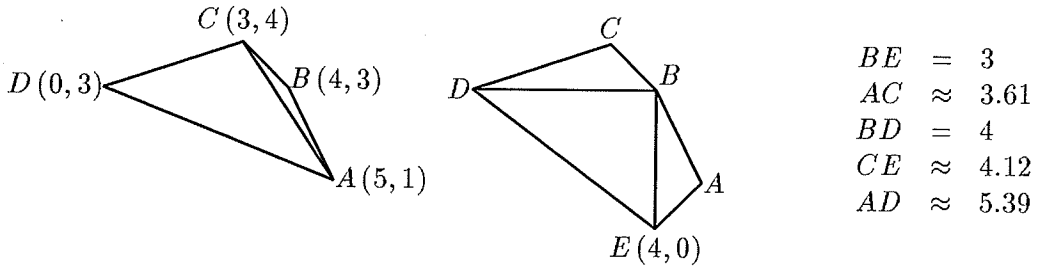


Figure 3.4: Neither Greedy nor Minimum Weight triangulation is local

3.3.1 Algorithms for *GOT*s

No polynomial-time algorithm is known for the Minimum Weight triangulation [263].

Plaisted and Hong [258] describe a $O(\log n)$ approximation to the MWT. Baszenski and Schumaker [21] have used simulated annealing to approximate the MWT.

Polynomial algorithms are known for only a few GOTs. Edelsbrunner *et al.* [103] generalize flips to insert arbitrary edges and develop an $O(n^2 \log n)$ algorithm for $GOT(\gamma_\infty)$. Bern *at al* [24] have generalized this approach to compute $GOT(b_\infty)$ in the cases where b_∞ has the *anchor property*. b_∞ has the anchor property if each triangle with the worst badness has an *anchor*. An anchor is a vertex that must be incident to a new edge crossing the opposite side of the triangle in any improved triangulation. In particular, γ_∞, $-a'_\infty$, and d_∞ have the anchor property [24].

Edelsbrunner and Tan [100] give a $O(n^2)$ algorithm for $GOT(a_\infty)$.

The brute-force algorithm for the Greedy triangulation takes $O(n^3)$ time and $O(n^2)$ space. In 1979 Gilbert [137] improved the time bound to $O(n^2 \log n)$. It was almost ten years before any further improvement was made. First, the space requirement was reduced to $O(n)$ [139, 210] and then the time requirement was reduced to $O(n^2)$ [204, 322] and finally to the optimal $O(n \log n)$ [323].

All the fast algorithms listed above are rather complicated.

For any badness function b, $GOT(b_{-\infty})$ can be computed by searching the $O(n^3)$ triangles that contain no other site, finding the one that minimizes b, and arbitrarily completing the triangulation. The naive implementation of this algorithm takes $O(n^4)$ time (since it takes $O(n)$ time to test if a triangle contains a site). This can be improved to $O(n^3)$: for a given site P, sort the other sites by their angles relative to P getting $Q_1, Q_2, \ldots, Q_{n-1}$. The triangle PQ_1Q_i will contain no other site if $\angle PQ_1Q_i$ is smaller than $\angle PQ_1Q_j$ for $1 < j < i$. Consequently, all site-free triangles with PQ_1 as a side can be found in time $O(n)$ by scanning through the Q_i while keeping track of the minimum $\angle PQ_1Q_i$. All site-free triangles involving P can be found in time $O(n^2)$ (including the time to sort the Q_i), and all site-free triangles can be found in time $O(n^3)$.

For some badness functions this can be improved—Edelsbrunner and Guibas [96] show that the minimum area triangle (and hence $GOT(\Delta_{-\infty})$) can be found in time $O(n^2)$ and space $O(n)$ by topologically sweeping the dual arrangement of lines. The *dual* of a point is a line and vice versa, in such a way that incidence is preserved [49]. The dual of the sites is a set of of lines that partition the plane into a subdivision known as an *arrangement* [146]. An edge in this dual arrangement that is a subset of the line Q from its intersection with line P to the line Q is dual to an angle PQR that contains no other site. The smallest

angle is dual to an edge in the arrangement, so the smallest angle and hence $GOT(\alpha_{-\infty})$ can also be found in time $O(n^2)$ by topologically sweeping the dual arrangement.

Eppstein [108] proves that in the case where the sites form a convex polygon $GOT(\alpha_{-\infty})$ is the farthest site Delaunay triangulation, which can be computed in time $O(n)$ [5].

For badness measures, b, that are minimized by degenerate triangles (such as Δ, α and r), the problem of computing $GOT(b_{-\infty})$ falls into the class of n^2-hard problems for which no sub-quadratic algorithms are known [251]. This is because the problem of determining whether three sites are collinear is n^2-hard.

If the sites form a convex polygon, Klincsek [184] describes a dynamic programming algorithm for the Minimum Weight triangulation ($GOT(P_1)$). We can generalize this to compute $GOT(b_f)$ for any triangle-based flip rule b_f for sites on a convex polygon.

Algorithm 1 *$GOT(b_f)$ of a convex polygon.*

Let $p_1, p_2, \ldots, p_n$ be the vertices (in order) of the convex polygon. We will consider p_{n+i} to be the same as p_i. The basic idea is to compute $B(i,j)$, defined to be the total badness of the $GOT(b_f)$ for $p_i, p_{i+1}, \ldots, p_j$.

for $i := 1$ **to** n **do**
  $B(i, i+1) := identity_f$
for $k := 3$ **to** $n-1$ **do**
  for $i := 1$ **to** n **do**
    $B(i, i+k) := \min_{i<j<i+k} f(B(i,j), f(b(p_i p_j p_k), B(j,k)))$

$identity_f$ is defined by $f(identity_f, x) = f(x, identity_f) = x$. For example, $identity_0 = 1$ and $identity_{-1} = \infty$.

This computes the badness of the GOT—the actual GOT can be extracted by the usual dynamic programming backtracking technique.

The algorithm uses $\Theta(n^3)$ time and $\Theta(n^2)$ space.

3.4 The Delaunay Triangulation

Let P be the set of points $p_1, p_2, \ldots, p_n$.

Definition. The *Voronoi polygon* V_{p_i} is the set of sites closer to p_i than to any other site.

Definition. The *Voronoi diagram* $Vor(P)$ is the set of Voronoi polygons for all sites.

It can be shown [263, 274] that if no four sites are co-circular, the straight-line dual of $Vor(P)$ forms the Delaunay triangulation.

Since there is only one Voronoi diagram of a set of sites it follows that the Delaunay triangulation has the systematic property. Furthermore, if a new site p_{n+1} is added to P the Voronoi polygons $\{V_{p_i} | i = 1, \ldots, n\}$ can only get smaller as $V_{p_{n+1}}$ takes territory away from them. Two Voronoi polygons that were not adjacent in $Vor(P)$ cannot be adjacent in $Vor(P + \{p_{n+1}\})$. So edges in the Delaunay triangulation of $P + \{p_{n+1}\}$ that are not in the Delaunay triangulation of P do not connect two sites in P—they must be adjacent to p_{n+1}. That is, the Delaunay triangulation has the local property.

Now, a triangle ABC in the Delaunay triangulation corresponds to three mutually adjacent Voronoi polygons, V_A, V_B and V_C. The boundary between two adjacent Voronoi polygons V_A and V_B consists of points equidistant from A and B and closer to to A and B than to any other site. The point common to all three boundaries is equidistant from A, B and C so it is the circumcentre of ABC. It is closer to A, B and C than to any other site, so the circumcircle of ABC contains no other site of P. This is called the empty-circle property. Contrariwise, any triangle with the empty-circle property must be a triangle of the Delaunay triangulation.

Definition. The flip rule DT is the rule that chooses the Delaunay triangulation of $ABCD$. That is,

$$DT(ABCD) = \begin{cases} AC & \text{if the Delaunay triangulation is } ABC, ACD \\ BD & \text{if it is } ABD, BCD \\ \textbf{either} & \text{if } ABCD \text{ is a cyclic quadrilateral} \end{cases}.$$

The equation of the circumcircle through $A = (x_A, y_A)$, $B = (x_B, y_B)$ and $C = (x_C, y_C)$ is

$$g(x, y) = \begin{vmatrix} x^2 + y^2 & x & y & 1 \\ x_A^2 + y_A^2 & x_A & y_A & 1 \\ x_B^2 + y_B^2 & x_B & y_B & 1 \\ x_C^2 + y_C^2 & x_C & y_C & 1 \end{vmatrix} = 0.$$

The point (x, y) is outside the circumcircle if and only if $g(x, y) > 0$. So,

$$DT(ABCD) = \begin{cases} AC & \text{if } g(x_D, y_D) > 0 \\ BD & \text{if } g(x_D, y_D) < 0 \\ \textbf{either} & \text{if } g(x_D, y_D) = 0 \end{cases} .$$

Theorem 3 $R_* = \{DT\}$. *(That is, if f is a joint function R_f=DT.)*

PROOF. Let $ABCD$ be a convex quadrilateral with the Delaunay triangulation ABC, ACD (figure 3.5). We can assume that $\angle ACB$ and $\angle CAD$ are acute angles. For, if $\angle ACB$ is obtuse

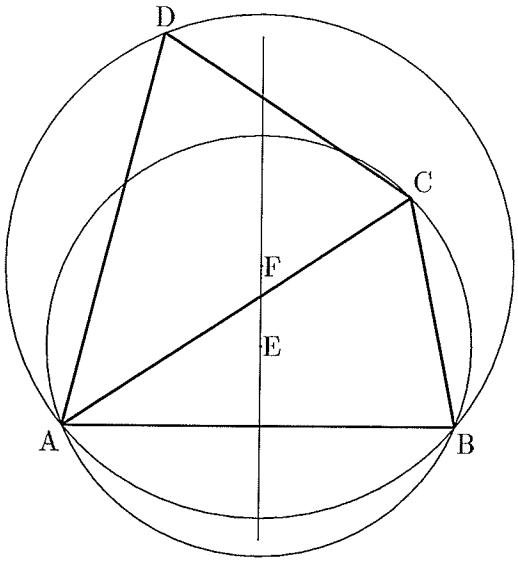


Figure 3.5: Delaunay triangulation of $ABCD$ is ABC, ACD

then $\angle BAC$ is acute (part of same triangle as $\angle ACB$) and $\angle ACD$ is acute (quadrilateral is convex); so we can interchange A and C to make the statement true. A similar argument works if $\angle CAD$ is obtuse.

The circumcentres of ABD (say F) and ABC (say E) both lie on the perpendicular bisector of AB. Consider the family of circles through A and B. As the circle centre moves along the bisector from infinity above AB (the side with C and D on) to infinity below, the part of the circle above AB gets smaller and the smaller pieces are contained in the larger ones. The circumradius gets smaller as the centre approaches AB, has a minimum when it reaches AB, and then gets larger again. Since C lies in the circumcircle of ABD, E is lower than F. By a plane geometry theorem [110] if $\angle ACB$ is acute E is above

AB and $R(ABC) < R(ABD)$. Similarly $R(CDA) < R(CDB)$. (R is the radius of the circumcircle—see table 3.2.)

So, for any joint function f, $f(R(ABC), R(ACD)) < f(R(ABD), R(CDB))$. Hence $R_f(ABCD) = AC$. □

Since R_* is a singleton set, with a slight abuse of notation we also use R_* to mean the single element of this set.

Theorem 4 *For any site set $LOT(DT)$ is the Delaunay triangulation.*

PROOF. Let ABC be a non-Delaunay triangle of a $LOT(DT)$. $\bigcirc ABC$ (the circumcircle of ABC) must contain another site, say X, on the opposite side of BC from A (we can relabel to make this true). A line from A to X will cross a finite number of triangles. Let BCD be the second of these (after ABC). $DT(ABCD) = BC$; so D must be outside $\bigcirc ABC$. So the the part of $\bigcirc ABC$ above BC is included in the part of $\bigcirc BCD$ above it. Hence X is in $\bigcirc BCD$. Similarly X is in the circumcircle of the third, fourth and every triangle crossed by AX. Let the last and second last of these be QRX and PQR respectively. Then $DT(PQRX) = QR$; so X is outside $\bigcirc PQR$. Contradiction. □

Since the Delaunay triangulation has the systematic and local properties this also shows that $R_* = DT$ is systematic and local.

$LOT(-\alpha_\infty)$ was one of the first triangulations considered [192] and can also be proved to be the same as the Delaunay triangulation (see appendix A).

The Delaunay triangulation can be calculated quickly, with very simple $O(n^2)$ algorithms and reasonably simple $O(n \log n)$ algorithms. The reason for this is the local property of the Delaunay triangulation.

Theorem 5 *If F is a local flip rule then the incremental algorithm to construct $LOT(F)$ of a set of n sites will require at most $\frac{1}{2}(n-2)(n-3)$ flips.*

PROOF. Consider what happens when the incremental algorithm is used to construct $LOT(F)$ of the site set P, where F is a local flip rule. When we add p_n the only flips that are performed are those that add a new edge adjacent to p_n and each such flip adds such an edge. So the number of flips to add p_n is just the degree of p_n in $LOT(F)$ minus the number of sites it is first connected to. p_n is initially connected to at least two sites because p_n will always be visible to two sites in a triangulation of $p_1, \ldots, p_{n-1}$. Hence the

number of flips is at most $n-3$ and the total number of flips needed by the incremental algorithm to construct $LOT(F)$ is at most $\frac{1}{2}(n-2)(n-3)$. □

Theorem 6 *The incremental algorithm can require $\frac{1}{2}(n-2)(n-3)$ flips to construct a $LOT(F)$ (F a local flip rule) of a set of n sites.*

PROOF. We will show that there is a site set where the incremental algorithm for the Delaunay triangulation requires $\frac{1}{2}(n-2)(n-3)$ flips. If $A=(a,a^2), B=(b,b^2), C=(c,c^2), D=(d,d^2)$ lie on the half parabola $x=\sqrt{y}$ and $0 \le a < b < c < d$ then

$$\begin{vmatrix} d^2+d^4 & d & d^2 & 1 \\ a^2+a^4 & a & a^2 & 1 \\ b^2+b^4 & b & b^2 & 1 \\ c^2+c^4 & c & c^2 & 1 \end{vmatrix} = (d-a)(d-b)(d-c)(c-a)(c-b)(b-a)(a+b+c+d) > 0$$

so $DT(ABCD) = AC$.

Now, consider the site set $P = \{p_1=(n,n^2), p_2=(n-1,(n-1)^2), \ldots p_n=(1,1)\}$. $DT(p_ip_3p_2p_1) = p_ip_2$ so p_3 is outside $\bigcirc p_ip_2p_1$. Similarly, $p_4, p_5, \ldots, p_{i-1}$ are also outside $\bigcirc p_ip_2p_1$; so $p_ip_2p_1$ is a triangle of the Delaunay triangulation of $p_1, \ldots, p_i$. By the same reasoning $p_ip_3p_2, p_ip_4p_3, \ldots, p_ip_{i-1}p_{i-2}$ are also triangles in the Delaunay triangulation of $p_1, \ldots, p_i$. That is, p_i is connected to every other site in the Delaunay triangulation of $p_1, \ldots, p_i$. When the incremental algorithm adds p_i to the triangulation it is initially connected to p_1 and p_{i-1} only (these are all it can see because P is a convex polygon), so $i-3$ flips must be performed. Hence the incremental algorithm requires a total of $\frac{1}{2}(n-2)(n-3)$ flips for this site set. □

Theorem 7 *If F is a local flip rule then the flip algorithm can require at most $\frac{1}{2}(n-2)(n-3)$ flips to construct $LOT(F)$ from any triangulation of a set of n sites. It does not matter in what order the flips are done.*

PROOF. Consider what happens after we perform a flip on a quadrilateral $ABCD$, replacing diagonal AC with diagonal BD. We now have the $LOT(F)$ for $ABCD$; so when we go on to consider other sites in the triangulation, because of the local property the only edges that can be added are those with at least one end a site other than A, B, C or D. Consequently the edge AC can never be added. That is, an edge deleted by a flip will never be added by

another flip. Each flip deletes exactly one edge. There are $\binom{n}{2}$ possible edges and at least $2n-3$ edges in the final triangulation, so the maximum number of flips is $\binom{n}{2} - (2n-3) = \frac{1}{2}(n-2)(n-3)$. □

Theorem 8 *There exists a set of n sites so that for any systematic flip rule F there is an initial triangulation for which the flip algorithm requires $O(n^2)$ flips to construct $LOT(F)$, regardless of the order in which the flips are done.*

PROOF. The site set P is defined as follows (we will take n as even): The sites $p_1, \ldots, p_{n/2}$ lie on the circle with centre $(-100, 0)$ and radius 99.5 with

$$p_i = (-100 + \sqrt{(99.5)^2 - (i/n)^2}, i/n) \approx (-0.5, i/n).$$

The sites $p_{(n/2)+1}, \ldots, p_n$ lie on the circle with centre $(100, 0)$ and radius 99.5 with

$$p_{i+n/2} = (100 - \sqrt{(99.5)^2 - (i/n)^2}, i/n) \approx (0.5, i/n).$$

(See figure 3.6.)

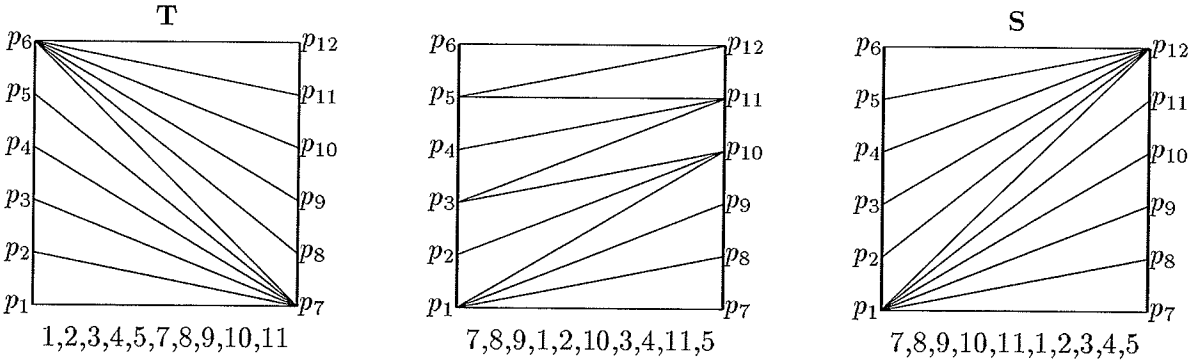


Figure 3.6: Some possible triangulations of P

Any triangulation of P must include the edge p_1p_2 since it is not intersected by any other possible edge. Similarly edges $p_2p_3, \ldots, p_{(n/2)-1}p_{n/2}, p_{n/2}p_{(n/2)+1}, \ldots, p_{n-1}p_n$ must be in every triangulation. The sequences of edges $p_1p_2 \ldots p_{n/2}$ and $p_{(n/2)+1} \ldots p_n$ divide the triangulation into three independent pieces. We shall only be concerned with the middle piece. Each triangle of the middle piece has two edges that cross the y axis and one edge of the form p_ip_{i+1}.

If we move a point up the y axis from $(0,0)$ to $(0,1)$ we cross each triangle (of the middle piece) in turn. This defines an ordering among these triangles. Associated with

this ordering is a sequence of the numbers 1 to $n-1$, excluding $n/2$. This sequence is formed by associating the number i with a triangle with an edge p_ip_{i+1} (the associated sequences are written below each triangulation in figure 3.6). There are two possible types of quadrilaterals in the triangulation of the middle piece:

- those with three corners in one piece and one in the other. These are of the form $p_ip_{i+1}p_{i+2}p_j$ with associated subsequence $i, i+1$. These quadrilaterals are not convex so no flip is possible.

- those with two corners in each piece. These are of the form $p_ip_{i+1}p_jp_{j+1}$ with two possible associated subsequences i, j and j, i depending on which way the diagonal of the quadrilateral is drawn. A flip in the triangulation corresponds to exchanging a pair of adjacent numbers in the sequence.

Therefore, a flip in the triangulation corresponds to changing the number of inversions (pairs of elements that are out of order) in the sequence by one.

Now consider the triangulation $\mathbf{T}$ formed by joining $p_{(n/2)+1}$ to all the sites in the left half and $p_{n/2}$ to all the sites in the right half. It has associated sequence $1, 2, \ldots, (n/2)-1, (n/2)+1, \ldots, n-1$ with 0 inversions. Now consider the triangulation $\mathbf{S}$ formed by joining p_n to all the sites in the left half and p_1 to all the sites in the right half. It has associated sequence $(n/2)+1, \ldots, n-1, 1, 2, \ldots, (n/2)-1$ with $((n/2)-1)^2$ inversions.

Therefore $((n/2)-1)^2$ flips are required to transform $\mathbf{T}$ into $\mathbf{S}$. Now, if it were possible to find a sequences of flips that transformed $\mathbf{T}$ and $\mathbf{S}$ into the $LOT(F)$ in less than $\frac{1}{2}((n/2)-1)^2$ flips we could transform $\mathbf{T}$ into $\mathbf{S}$ in less than $((n/2)-1)^2$ flips merely by transforming $\mathbf{T}$ into the $LOT(F)$ and reversing the flips that transform $\mathbf{S}$ into the $LOT(F)$. Hence either $\mathbf{T}$ or $\mathbf{S}$ requires at least $\frac{1}{2}((n/2)-1)^2 = O(n^2)$ flips. □

Theorem 8 might seem to suggest that an $O(n^2)$ algorithm is optimal for constructing the Delaunay triangulation but Delaunay triangulation algorithms do not have to use flips. The local property also lets us merge two disjoint Delaunay triangulations of n sites each in $O(n)$ time. This leads to an $O(n \log n)$ Divide-and-Conquer algorithm to construct the Delaunay triangulation [202].

In practice the $O(n^2)$ worst case does not seem to occur and the performance of the flip algorithm is competitive with that of the Divide-and-Conquer algorithm [120].

If we use the incremental or flip algorithm to construct $LOT(b_f)$, where b_f is not local, although we know that the process of repeatedly performing flips will eventually terminate,

there is no reason why exponentially many flips may not be necessary to construct $LOT(b_f)$.

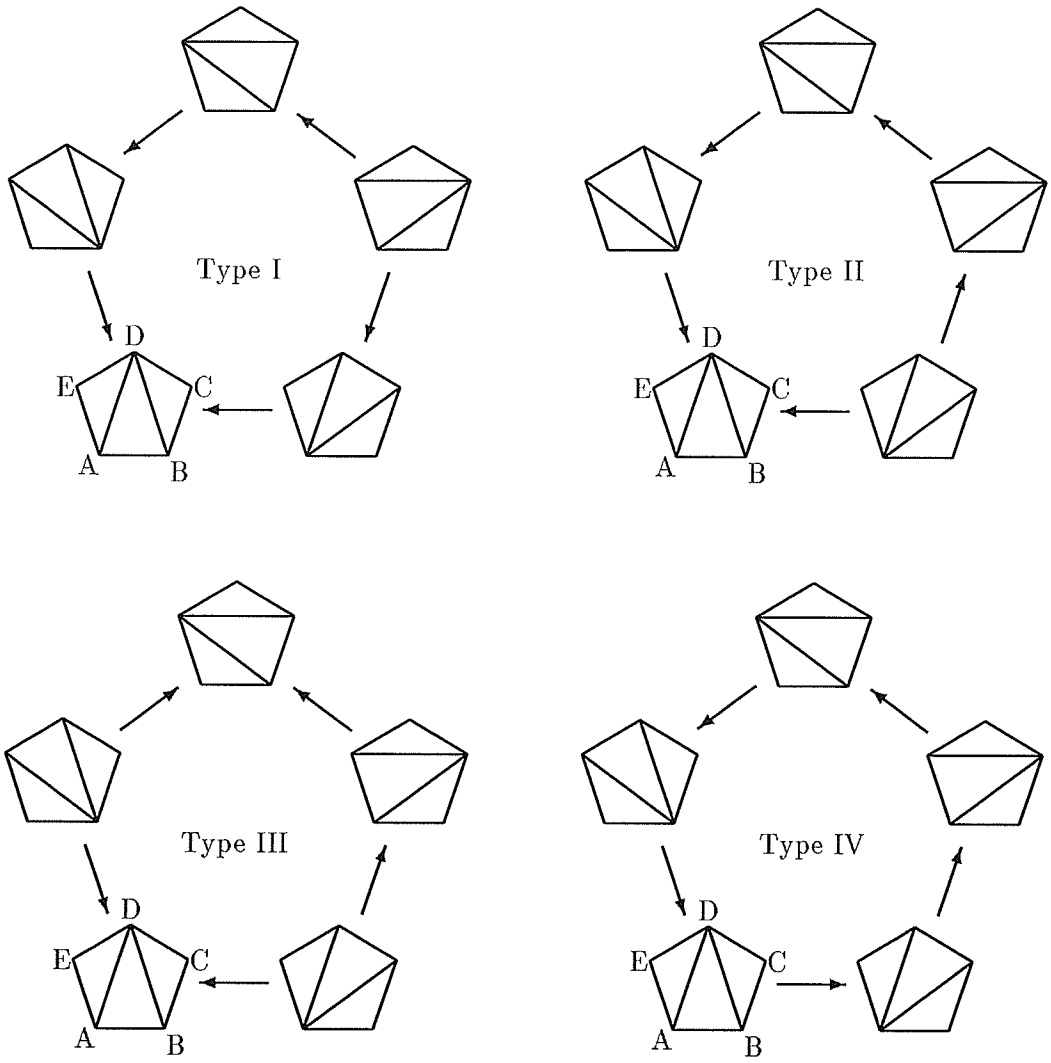


Figure 3.7: Possible directed flip graphs

3.5 Testing Flip Rules

By using the badness measures in table 3.2 each combined with the joint functions in table 3.3 we can create 120 flip rules. We would like to find out which ones are systematic and local.

Consider a set of sites that forms a convex pentagon. If the flip rule does not return

either for any of the 5 subsets of size 4 (quadrilaterals), there are only four possible different directed flip graphs for such a set (see figure 3.7).

- Type IV: No sink. No LOT exists. We proved this was impossible for triangle-based flip rules.
- Type III: Two sinks. LOT is not unique. The flip rule is not systematic.
- Type II: One sink. The flip rule is not local: The (unique) LOT of $ACDE$ is ACE, CDE since EC is preferred to DA but ADE, ABD, BD is the LOT of $ABCDE$. The edge AD is a new edge in this triangulation although it is not adjacent to B.
- Type I: One sink. A flip rule which is systematic and local will always give this result.

If the flip rule returns the value **either** for any quadrilateral, we classify the flip rule as type V. Since **either** should result with probability zero in a good flip rule, a type V flip graph is a witness to the fact that a flip rule is not good.

To test our triangle-based flip rules we generate "random" pentagons and then for each triangle-based flip rule classify the type of the directed flip graph. If any type III graphs are found, then we know that that triangle-based flip rule is not systematic. If any type II graphs are found, then we know that that triangle-based flip rule is not local. If any type IV graphs are found, we have made mistake somewhere.

Of course, even if our flip rule gives us only type I graphs this doesn't mean it is local and systematic. Perhaps our set of test pentagons was unlucky, or perhaps the flip rule is local and systematic for pentagons but not for other site sets.

For example, consider the flip rule $-R_1$ (*maximize* the sum of the circumradii). The directed flip graph for this rule can be obtained by reversing the edges of the directed flip graph for R_1 (minimize the sum of the circumradii). Reversing the edges of the type I graph in figure 3.7 gives another type I graph so $-R_1$ will always produce a type I graph for a pentagon.

However, $-R_1$ is not systematic as figure 3.8 shows. Figure 3.8 is the directed flip graph for the four corners of a square and a fifth site inside the square using $-R_1$. (The fifth site is inside the circumcircle of any three of the others so the middle triangulation in figure 3.8 is the Delaunay triangulation of the set. So, in the directed flip graph using $DT = R_1$[4] both arrows point to the middle triangulation. Reversing them gives figure 3.8.)

[4]See theorem 3 in section 3.4.

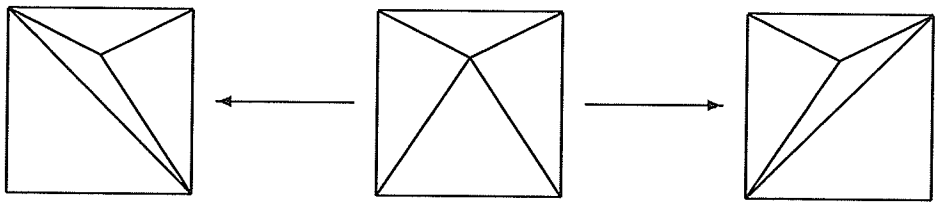


Figure 3.8: Directed flip graph using $-R_1$

We have already seen that $LOT(R_*)$ and $LOT(-\alpha_\infty)$ are the same as the Delaunay triangulation. Other flip rules might also be the same as DT; so if the flip graph is type I we check to see if the LOT is the same as the Delaunay triangulation.

5000 pentagons were generated by taking points from the uniform distribution within the unit circle until their convex hull formed a pentagon. (This is the iteration method for generating convex polygons described in section 6.3.) The results are summarized in table 3.4.

The first thing to be noted about table 3.4 is the set of flip rules that produced type V flip graphs.

The reason why Δ_1 always gave a type V flip graph is quite simple. The sum of the areas of the triangles making up a quadrilateral is just the area of that quadrilateral, and this will be the same whichever way the diagonal is drawn. Consequently Δ_1 will always give the value **either**—not a very useful flip rule.

a_∞ does not always return the value **either**—it will only do so when the quadrilateral has a side longer than both diagonals. However, it is easy to see that this will always be the case for at least one sub-quadrilateral of a pentagon since the longest diagonal or side of the pentagon will be a side of at least one sub-quadrilateral.

Similar explanations can be found for the other rules that gave type V flip graphs.

The second thing to be noted is the set of rules that always produced the Delaunay triangulation. As well as R_* and $-\alpha_\infty$, as we would expect from section 3.4, this set includes the rules $-r_1$, $(-r/R)_1$, $(-r^2/\Delta)_0$ and abc_1. Proofs that this is always the case for some of these rules can be found in appendix A.

This discovery might prove useful in programs that compute the Delaunay triangulation since it is possible that one of these rules could be calculated in less time than it takes to test to see if a site is inside the circumcircle of three other sites. Or one of these rules may be less vulnerable to numerical error when the four sites are almost co-circular.

The third and most important thing to be noticed is the remaining set of rules. *None* of

Joint Function	Badness Measure (see table 3.2)																				
		R	α	γ	a'	Δ	r	P	a	c	$\frac{r}{R}$	$\frac{r^2}{\Delta}$	$\frac{a}{a'}$	$\frac{c}{a}$	abc	$\frac{c}{P}$	$a-c$	$\frac{b}{c-a}$	R^∞	P^∞	r^∞	
$l_{-\infty}$	DT	100	49	14	16	3	9	40	60	2	34	38	32	41	37	36	48	34	49	38	9	DT
	I	100	73	56	56	54	52	71	85	9	66	69	66	68	69	63	73	66	79	71	53	I
	II	0	7	12	12	16	13	7	4	1	10	9	9	11	8	10	8	10	4	6	14	II
	III	0	20	32	32	30	35	22	11	6	24	23	25	22	24	26	18	25	17	24	33	III
l_{-1}	DT	100	91	67	70	21	65	54	60	2	86	97	78	48	48	41	50	67	49	48	58	DT
	I	100	99	84	89	54	86	82	85	9	98	100	95	77	78	71	77	87	79	78	84	I
	II	0	0.7	5	4	16	5	5	4	1	0.6	0.1	2	5	5	6	5	5	4	4	5	II
	III	0	0.4	11	7	30	8	14	11	6	1	0.1	4	18	17	23	18	8	17	18	11	III
l_0	DT	100	85	72	81	21	75	57	60	2	91	100	97	48	60	43	50	54	49	50	64	DT
	I	100	97	88	94	54	93	83	85	9	99	100	100	76	85	73	78	78	79	79	88	I
	II	0	1	3	1	16	2	4	4	1	0.2	0	0.1	5	4	6	5	6	4	4	4	II
	III	0	2	9	4	30	5	13	11	6	0.9	0	0.1	19	11	21	18	16	17	17	9	III
l_1	DT	100	65	75	79	0	100	60	60	2	100	94	96	49	100	47	50	47	49	52	75	DT
	I	100	85	90	95	0	100	85	85	9	100	100	100	76	100	74	78	74	79	81	94	I
	II	0	5	3	2	0	0	4	4	1	0	0.2	0.1	5	0	6	4	6	4	4	2	II
	III	0	10	8	3	0	0	11	11	6	0	0.3	0.3	19	0	20	17	20	17	16	5	III
l_2	DT	100	56	75	31	3	36	64	60	2	70	79	91	48	67	47	51	43	49	54	32	DT
	I	100	79	91	66	54	69	88	85	9	89	95	99	74	87	72	79	71	79	82	68	I
	II	0	6	2	7	16	7	4	4	1	4	2	0.3	7	3	8	5	7	4	4	7	II
	III	0	15	7	26	30	25	9	11	6	6	3	0.5	19	10	20	16	21	17	14	25	III
l_∞	DT	100	100	71	55	21	50	63	0	0	79	88	83	48	43	36	48	77	0	48	47	DT
	I	100	100	91	79	54	75	85	0	0	95	98	96	77	69	67	77	94	0	76	74	I
	II	0	0	4	9	16	11	7	0	0	3	1	2	8	8	8	8	3	0	3	10	II
	III	0	0	5	12	30	14	8	0	0	3	0.9	2	16	23	24	16	4	0	9	16	III

There were no type IV flip graphs

The only flip rules to produce type V flip graphs									
rule	Δ_1	a_∞	$c_{-\infty}$	c_{-1}	c_0	c_1	c_2	c_∞	R_∞^∞	P_∞^∞
% type V	100	100	84	84	84	84	84	100	100	11

Table 3.4: % of flip graphs of each type for flip rules

them were systematic or local. This contradicts Nielson and Franke [243] who, because of the close similarity between the descriptions of $-\alpha_\infty$ and γ_∞ (see table 3.1) assumed that γ_∞ was systematic.[5]

Another rule that might be thought to be systematic and local is R_*^∞. At first sight it would appear that the proof in section 3.4 should work if the word 'circle' is replaced throughout by 'square parallel to the axes'. Unfortunately, it is not always possible to draw such a square through three sites. For example, no square parallel to the axes can be drawn through the points $(0,0)$, $(1,3)$ and $(4,4)$. So, while we can find a smallest enclosing square for this triangle with radius 2, this square does not have the "empty square property" and the proof in section 3.4 does not apply.

If $1 < p < \infty$ then it is possible to find a ball in the l_p metric through any three points, so R_*^p is systematic and local in these cases. (Tests confirm that it always produces type I flip graphs for pentagons.) However, the triangulations that these rules produce are duals of the Voronoi diagram under the l_p metric so these triangulations are generalized Delaunay triangulations. (Though these rules are not rotation invariant.)

If we examine the scatter plot of type I and Delaunay percentages shown in figure 3.9 (rules that gave any type V flip graphs have been excluded from this plot) there is a strong correlation between these two percentages, which suggests that a rotation invariant rule that always gives a type I flip graph will always give the Delaunay triangulation. This result is proved in the next section. In section 3.4 we noted that constructing a locally optimized triangulation for a flip rule that is not local could take exponential time. The authors cited in table 3.1 managed to construct LOTs for several different non-local flip rules, presumably without needing an exponential number of flips. How is this possible? The answer lies in the fact that all of these rules agreed with DT more than 50% of the time on a convex pentagon. Consequently a LOT for one of these rules of a typical set of sites will be identical to the Delaunay triangulation except for a number of isolated 'islands' of a few adjacent triangles. The flip algorithm will perform quite quickly on the part of the triangulation that is the same as the Delaunay and exponential behaviour on a small set of non-Delaunay triangles will not cause any problems. Nielson and Franke [243] compared $LOT(DT)$ and $LOT(\gamma_\infty)$ for the same set of sites and found 90% of the triangles to be the same.

[5] Nielson has written a note correcting this error [242].

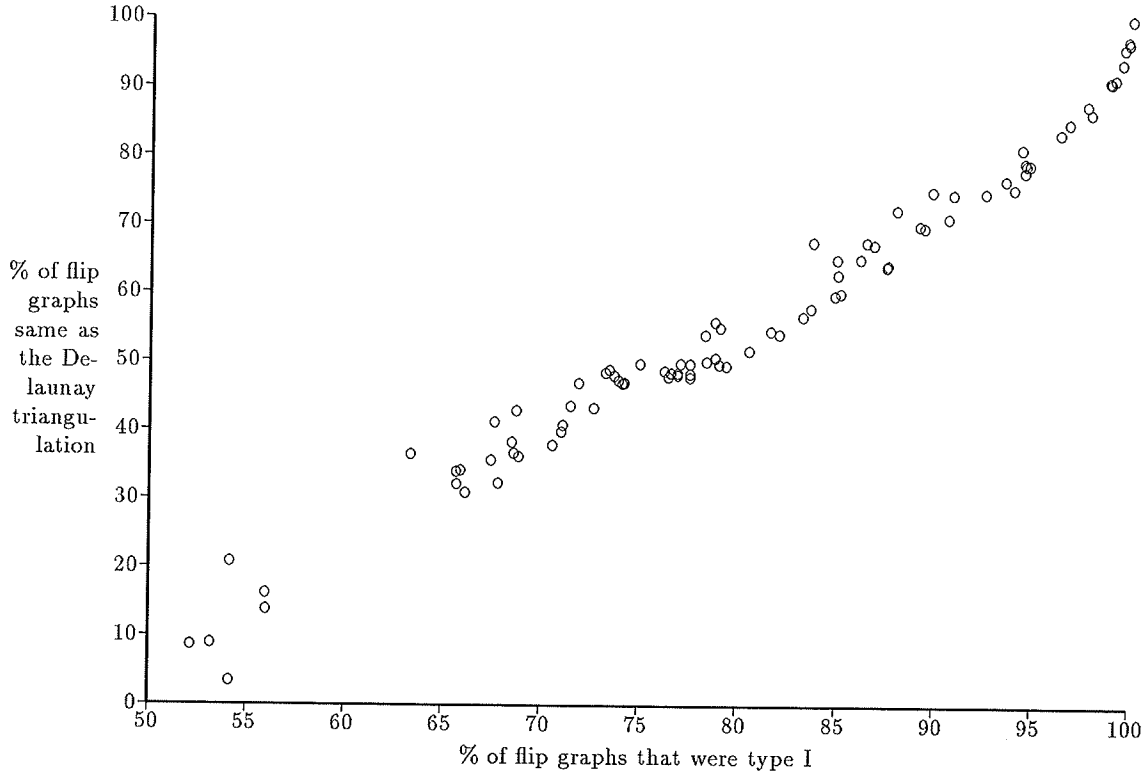


Figure 3.9: Scatter plot for flip rules

3.6 Systematic and Local Flip Rules are Generalized Delaunay rules

Definition. The lines making up the sides of a triangle ABC divide the plane into seven regions (see figure 3.10). We will denote by $AB\overline{C}$ the region that is adjacent to the sites A and B, but not C. The regions are open sets. Their union is the plane except for the lines AB, BC, and AC.

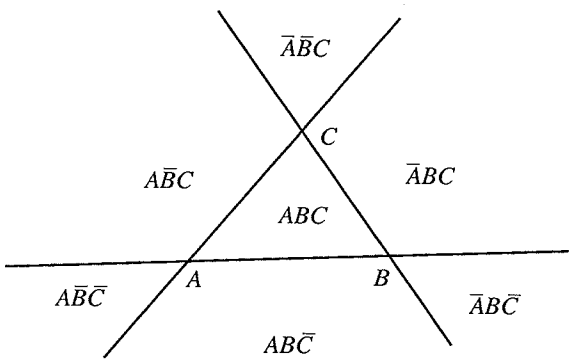


Figure 3.10: Regions around a triangle

Definition. $F^+(ABC)$ is the set of points in the plane for which the flip rule F would not choose the triangle ABC. That is, if $D \in A\overline{B}C$ then $D \in F^+(ABC)$ iff $F(ABCD) = BD$. We can similarly define $F^+(ABC)$ in the regions $\overline{A}BC$ and $AB\overline{C}$. In the remaining regions, the quadrilateral is not convex, so the flip rule must choose the interior diagonal. This means that ABC is included in $F^+(ABC)$ and $\overline{A}\overline{B}C$, $\overline{A}B\overline{C}$, $A\overline{B}\overline{C}$ are excluded. $F^0(ABC)$ is the boundary of $F^+(ABC)$, and $F^-(ABC) = \mathbf{R}^2 - (F^+(ABC) + F^0(ABC))$

Definition. Let $\mathcal{F}^0$ be the set of curves $F^0(ABC)$ for all possible triangles ABC.

If $F = DT$, $\mathcal{F}^0$ is the set of all circles. This is intimately related to the empty-circle property of the Delaunay triangulation. The curves in $\mathcal{F}^0$ for some flip rule F act in a way similar to circles for DT.

Figure 3.11 shows $\mathcal{F}^0$ for $F = \gamma_\infty$. We have rotated and scaled so that the longest triangle side is $(0,0)(1,0)$. The black dot marks the position of the third triangle vertex.

This plot was produced numerically (and some of the jaggedness of the curves are sampling artifacts), Hansford [151] and Powar [262] give analytical descriptions of γ^0_∞.

The interesting thing to note is that some of the curves in figure 3.11 (the two dashed curves in particular) intersect at points other than $(0,0)$ and $(1,0)$. We will prove that this is not possible for a systematic local flip rule.

The basic idea behind the proof is illustrated by figure 3.12 which shows just the two dashed curves from figure 3.11. The flip graphs for the pentagons $ABCDE$ and $A'BCDE$ are shown in figure 3.13. The bold arrows in figure 3.13 follow from the placement of the sites in figure 3.12. For example, the direction of the leftmost bold arrow in figure 3.13 is a consequence of the fact that $A \in \gamma^-_\infty(BDE)$ ($\gamma^-_\infty(BDE)$ is the region outside $\gamma^0_\infty(BDE)$ in figure 3.12.)

If γ_∞ is systematic and local, both of the flip graphs in figure 3.13 must be type I—this means that the remaining edges must be as shown in the figure. This is impossible since it requires both $\gamma_\infty(BCDE) = EC$ (in the left graph) and $\gamma_\infty(BCDE) = BD$ (in the right graph).

This proves that γ_∞ cannot be systematic and local. Section 3.6.1 generalizes this result.

Definition. A curve *circumscribes* a polygon if each vertex of the polygon lies on the curve.

Definition. We say that F has the *circumscribing property* if, given any triangle, there is exactly one curve in $\mathcal{F}^0$ circumscribing that triangle.

3.6.1 Systematic local rules have the circumscribing property

Lemma 1 *If F is a systematic local rule and $C \in F^0(ABE) \cap \overline{A}BE$ then $F^+(ABE) = F^+(ACE)$ in the region $\overline{A}CE - \overline{ABC}$ (the shaded region in figure 3.14).*

PROOF. If $F^+(ABE) \neq F^+(ACE)$ then there is a point $D \in F^+(ABE) \setminus F^+(ACE)$ or a point $D \in F^+(ACE) \setminus F^+(ABE)$ (see figure 3.14). Note that $ABCDE$ is strictly convex. Let's consider the first case. If $D \in F^0(ACE)$ then because $F^+(ABE)$ is open and $F^0(ACE)$ is the boundary of $F^-(ACE)$ we can find a new $D' \in F^+(ABE) \cap F^-(ACE)$.

- $ABCD'E$ is strictly convex
- $F(ABCE) =$ **either**
- $F(ABD'E) = AD'$

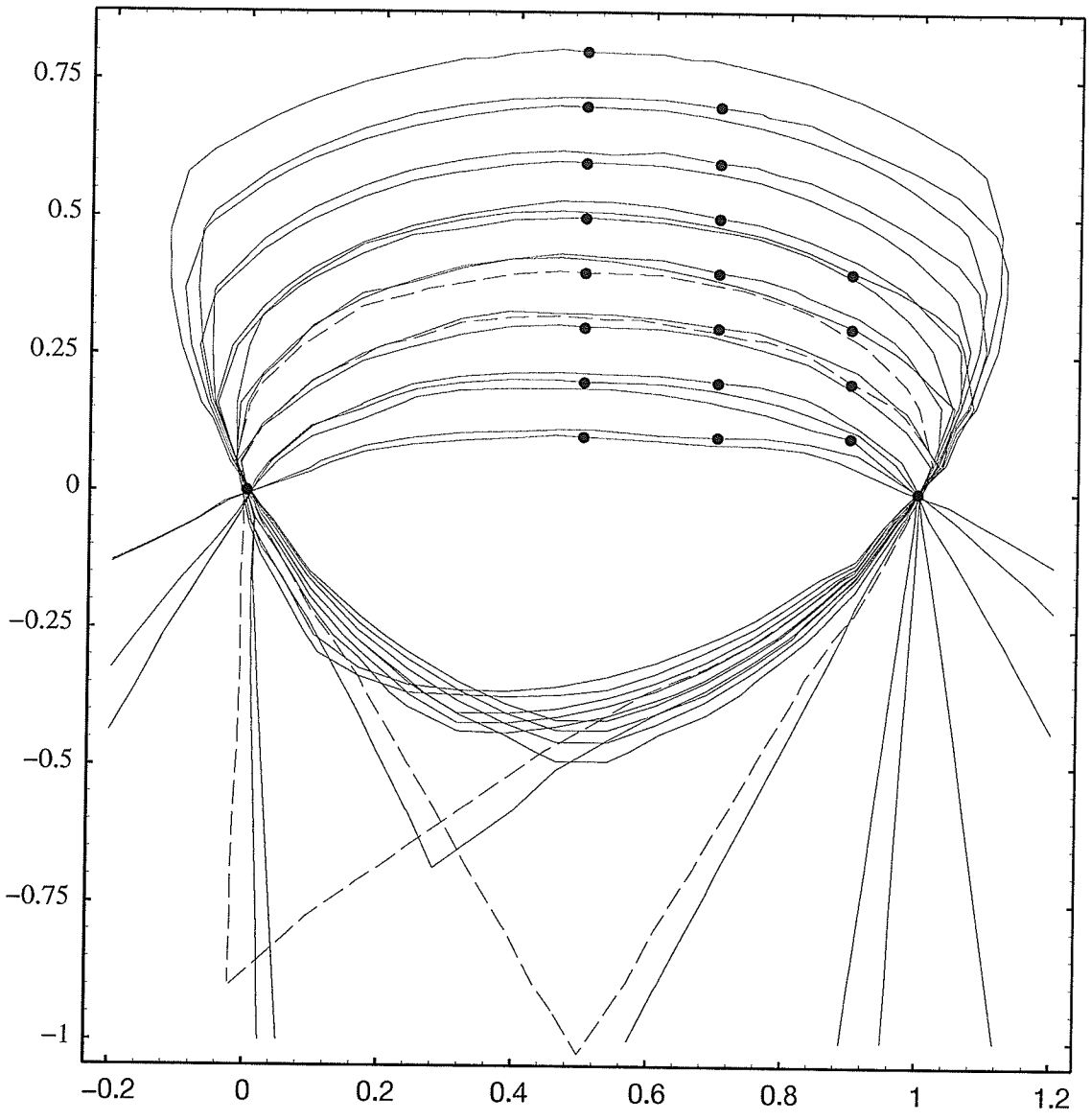


Figure 3.11: γ_∞^0

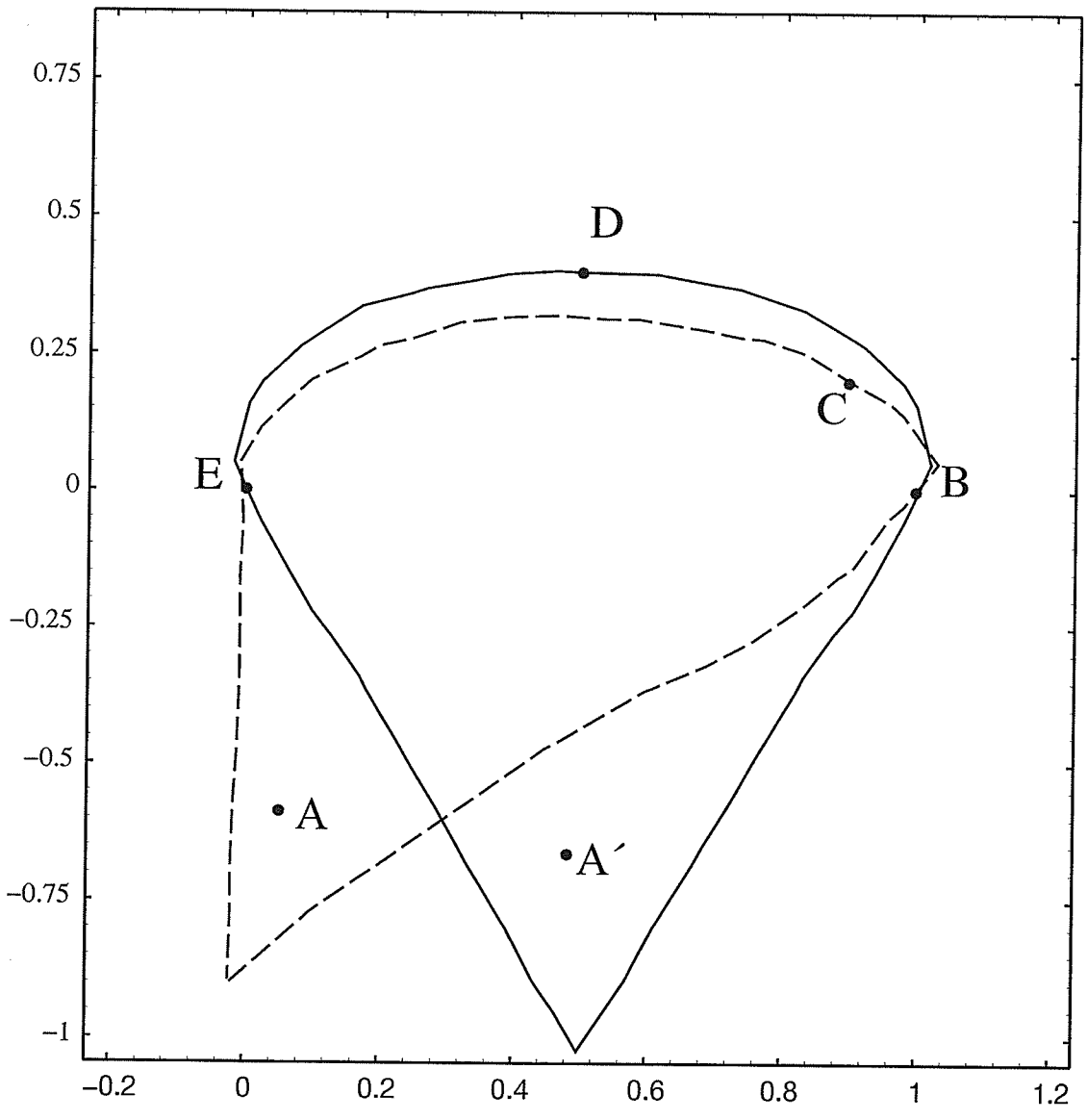


Figure 3.12: $\gamma^0_\infty(BDE)$ and $\gamma^0_\infty(BCE)$

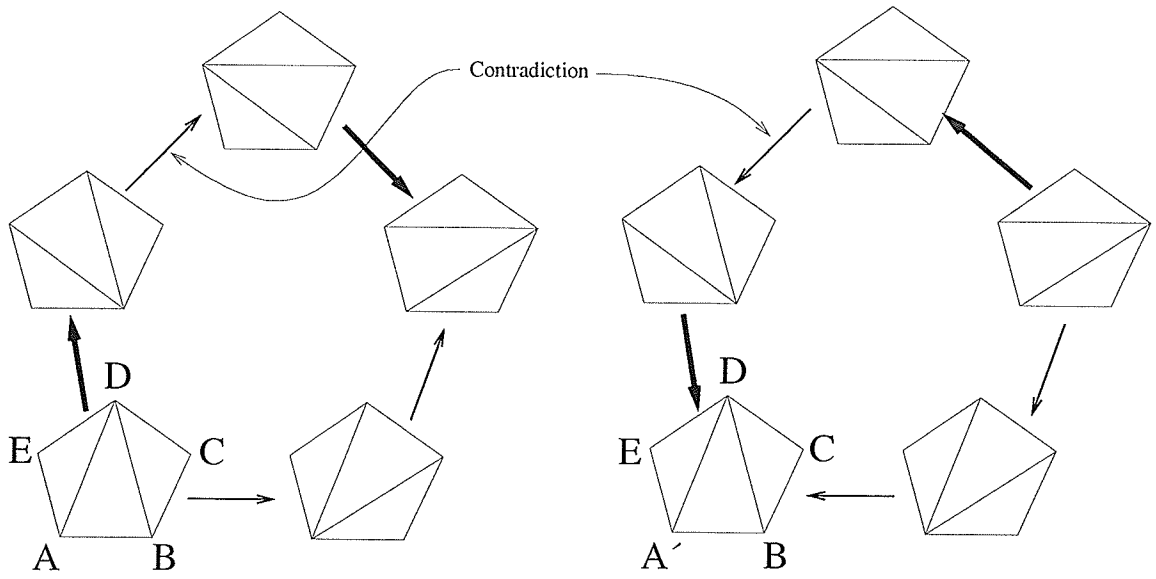


Figure 3.13: Flip graphs for $ABCDE$ and $A'BCDE$.

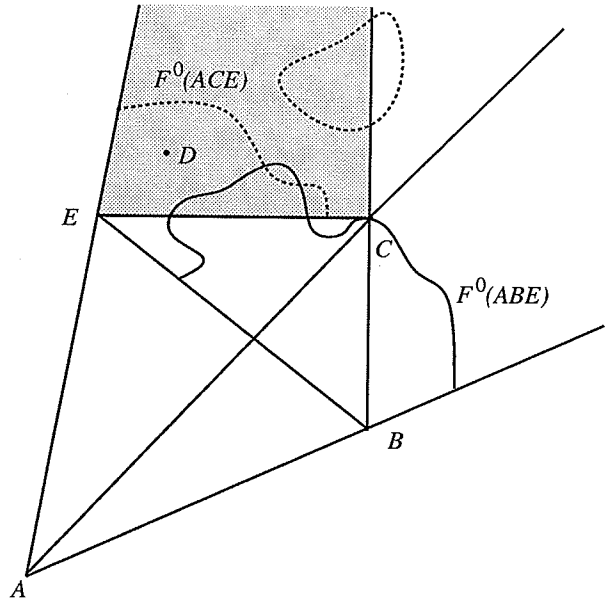


Figure 3.14: $F^0(ABE)$ and $F^0(ACE)$

- $F(ACD'E) = CE$

Now, because $AD'(F)$ and $CE(F)$ are open sets we find a ball around $ABCE$ such that for all $A'B'C'E'$ in that ball $F(A'B'D'E') = A'D'$, $F(A'C'D'E') = C'E'$ and $A'B'C'D'E'$ is convex. And since $ABCE$ is on the boundary between $AC(F)$ and $BE(F)$ we can find $A'B'C'E'$ in that ball such that:

- $A'B'C'D'E'$ is strictly convex
- $F(A'B'C'E') = B'E'$
- $F(A'B'D'E') = A'D'$
- $F(A'C'D'E') = C'E'$

There is no way that we can pick the directions of the two remaining edges in the flip graph for $A'B'C'D'E'$ so that it is type I (see figure 3.15). This contradicts F being local and systematic.

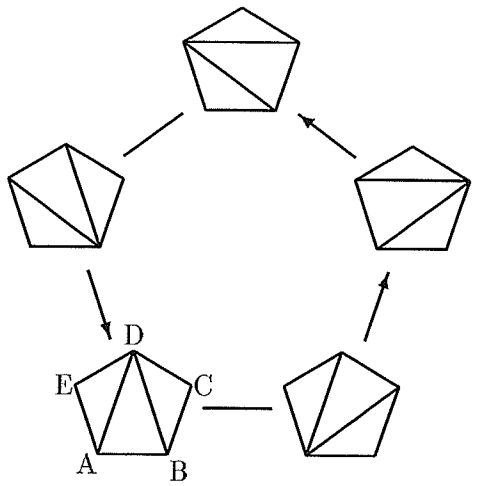


Figure 3.15: Not a type I flip graph

If $D \in F^+(ACE) \setminus F^+(ABE)$, the proof is the same, except that we find $A'B'C'E'$ such that $F(A'B'C'E') = A'C'$. This leads to a flip graph that is figure 3.15 with all the arrows reversed. This still can't be type I. □

Lemma 2 *If F is a systematic local rule and $C \in F^0(ABE) \cap \overline{A}BE$ then $F^+(ABE) = F^+(BCE)$ in the region $\overline{A}CE - \overline{AB}C$ (the shaded region in figure 3.14).*

PROOF. If there is a point $D \in F^+(ABE) \setminus F^+(BCE)$, then just as in lemma 1 we can find $A'B'C'D'E'$ such that

- $A'B'C'D'E'$ is strictly convex
- $F(A'B'C'E') = B'E'$
- $F(A'B'D'E') = A'D'$
- $F(B'C'D'E') = C'E'$

The flip graph for $A'B'C'D'E'$ cannot be type I (see figure 3.16).

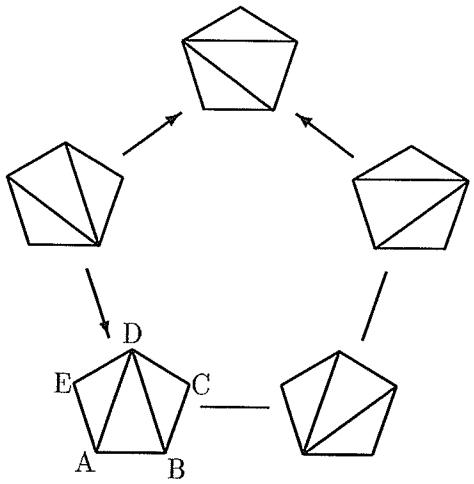


Figure 3.16: Not a type I flip graph

If $D \in F^+(BCE) \setminus F^+(ACE)$ then we can find a flip graph that is figure 3.16 with the arrows reversed. This still can't be type I. □

Lemma 3 *If F is a systematic local rule and $C \in F^0(ABE) \cap \overline{A}BE$ then $F^+(ABE) = F^+(ACE) = \emptyset$ in the region $\overline{AB}C$ (the shaded region in figure 3.17).*

PROOF. If there is a $D \in F^+(ACE) \cap \overline{AB}C$ then, just as in the proof of Lemma 1 we can find $A'B'C'E'$ such that

- $A'B'C'E$ is convex
- $D \in \overline{A'B'}C'$.

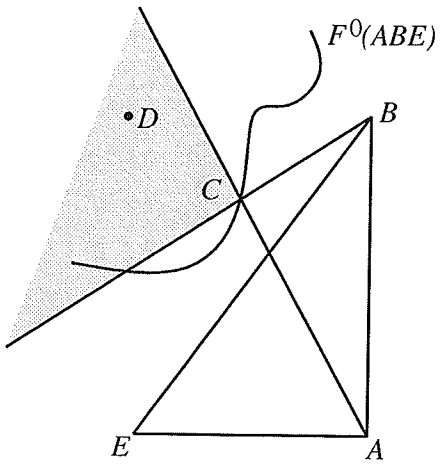


Figure 3.17: $D \in \overline{ABC}$

- $F(A'C'DE') = A'D$
- $F(A'B'C'E') = B'E'$

The flip graph for $A'B'C'DE'$ (see figure 3.18) contradicts F being systematic. Hence $F^+(ABE) \cap \overline{ABC} = \emptyset$.

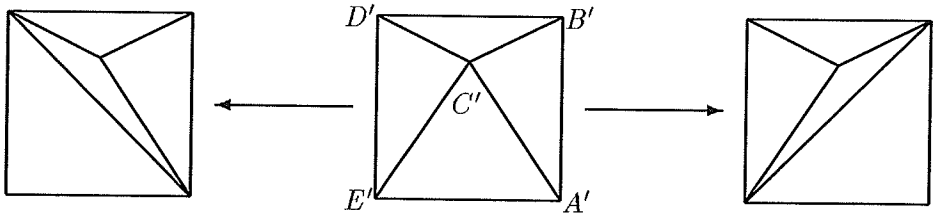


Figure 3.18: Flip graph of $A'B'C'DE'$

If there is a $D \in F^+(ACE) \cap \overline{ABC}$, then we find $D' \in F^-(ACE) \cap F^+(ABE) \cap \overline{ABC}$ and $A'B'C'E'$ such that

- $A'B'C'E'$ is convex
- $D' \in \overline{A'B'C'}$.
- $F(A'B'D'E') = A'D'$
- $F(A'C'D'E') = C'E'$

- $F(A'B'C'E') = B'E'$

The triangulation of $A'B'D'E'$ is $A'B'D', A'D'E'$. If we add C' the resulting triangulation of $A'B'C'D'E'$ is $A'B'E', B'C'E', B'D'C', C'D'E'$ (see figure 3.19). We have a new edge $B'E'$ that is not adjacent to C', which contradicts F being local. Hence $F^+(ACE) \cap \overline{ABC} = \emptyset$. □

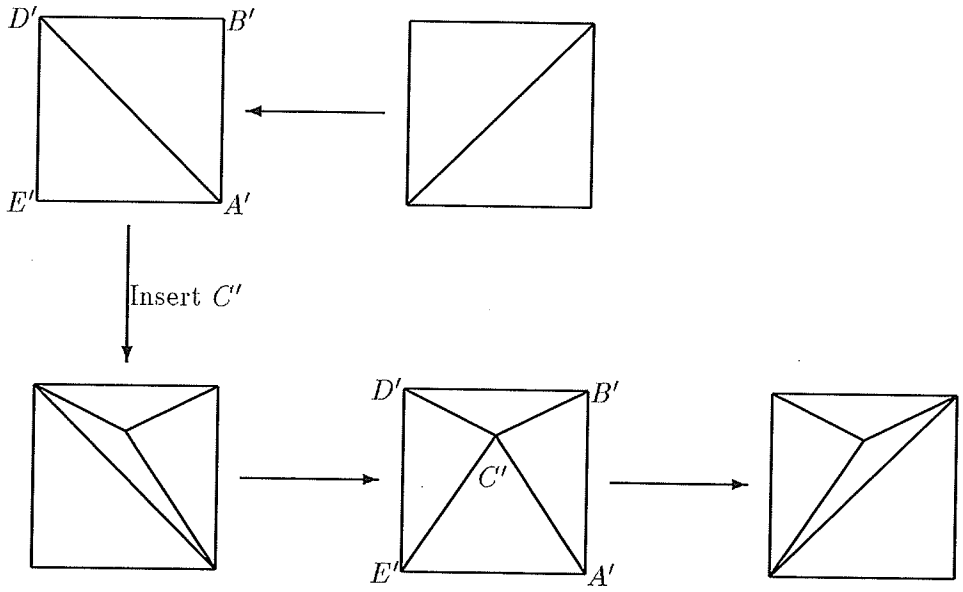


Figure 3.19: Inserting C' in the triangulation of $A'B'D'E'$

Lemma 4 *If F is a systematic local rule and $D \in F^0(ABE) \cap \overline{A}BE$ then $(BDE \cap ABD) \subset F^+(ABE)$ ($BDE \cap ABD$ is the shaded region in figure 3.20).*

PROOF. Suppose $C \in (BDE \cap ABD) \setminus F^+(ABE)$. If $C \in F^0(ABE)$ then we can find a $C' \in BDE \cap ABD \cap F^-(ABE)$. We can now find $A'B'D'E'$ such that

- $A'B'D'E'$ is convex
- $C' \in B'D'E' \cap A'B'D'$
- $F(A'B'D'E') = A'D'$
- $F(A'B'C'E') = B'E'$

Depending on the choice we make for the remaining edge direction in the flip graph for $A'B'C'D'E'$ (see figure 3.21), we end up either with figure 3.18 or figure 3.19, contradicting F being systematic and local. □

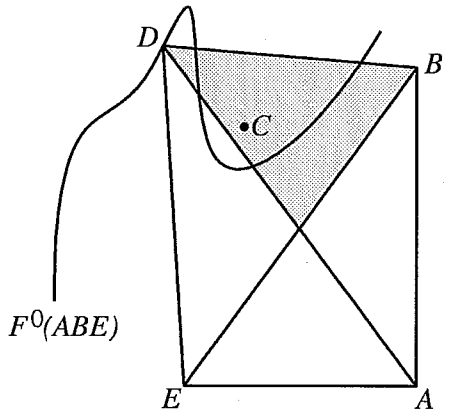


Figure 3.20: $C \in BDE \cap ABD$

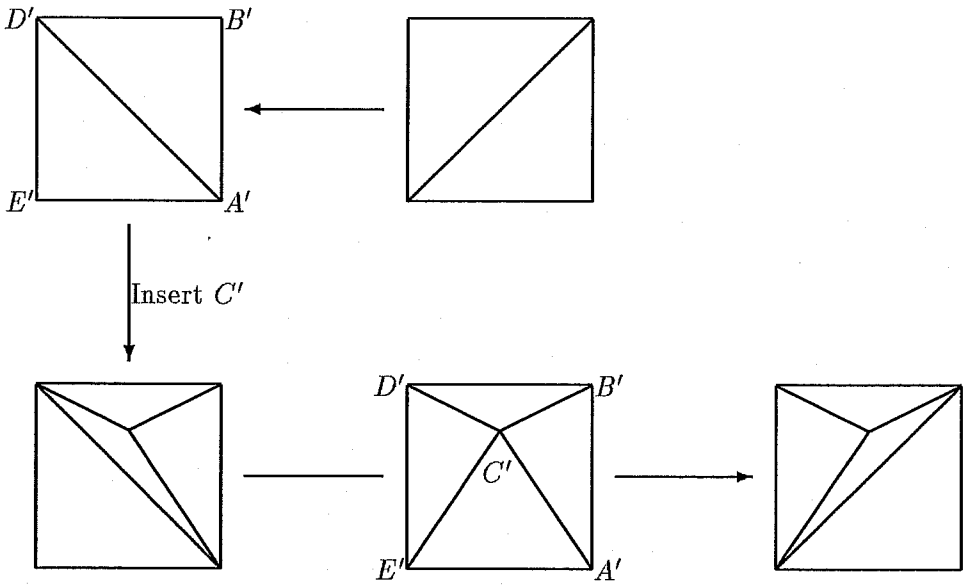


Figure 3.21: Either nonlocal or nonsystematic

Lemma 5 *If F is a systematic local flip rule and $D \in F^0(ABC)$ then $F^+(ABC) = F^+(BCD) = F^+(ACD) = F^+(ABD)$.*

PROOF. D cannot be in $ABC \cup A\overline{BC} \cup \overline{A}B\overline{C} \cup \overline{AB}C$. Relabel the points if necessary, so that $D \in A\overline{B}C$ (see figure 3.22). Note that $C \in F^0(ABD)$, $B \in F^0(ACD)$ and $A \in F^0(BCD)$.

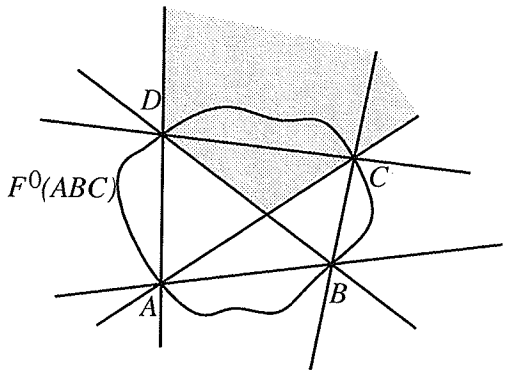


Figure 3.22: $D \in A\overline{B}C$

We need to prove the result in the regions $\overline{A}CD$ and $ACD \cap BCD$ (shaded in figure 3.22). The rest will follow by symmetry.

- In the region $\overline{A}CD \cap \overline{B}CD$, applying lemma 1 with ABE replaced by ABC shows that $F^+(ABC) = F^+(BCD)$ and with ABE replaced by ABD shows that $F^+(ABD) = F^+(ACD)$. Applying lemma 2 with ABE replaced by ABC shows that $F^+(ABC) = F^+(ACD)$.

- In the region $\overline{AB}C$, lemma 3 shows that $F^+(ABC) = F^+(ABD) = \emptyset$. Because F is a flip rule $F^+(ABC) = \emptyset$, and $F^+(BCD) = \emptyset$ because $\overline{AB}C \subset \overline{B}C\overline{D}$.

- in the region $BCD \cap ACD = I$, applying lemma 4 with ABE replaced by ABC shows that $I \subset F^+(ABC)$ and with ABE replaced by ABD shows that $I \subset F^+(ABD)$. $I \subset F^+(BCD)$ because $I \subset BCD$ and $I \subset F^+(ACD)$ because $I \subset ACD$.

Finally we note that we have left out the lines AB, BC and AC in the proof, but there is only one way to complete $F^+(ABC)$ onto these lines. □

Theorem 9 *If F is a systematic local rule and $P, Q, R \in F^0(ABC)$ then $F^+(ABC) = F^+(PQR)$. (That is, F has the circumscribing property.)*

PROOF. By lemma 4, $F^0(ABC) = F^0(ABR) = F^0(AQR) = F^0(PQR)$. □

Theorem 10 *If F is a systematic local rule then $F^+(ABC)$ is convex.*

PROOF. If $F^+(ABC)$ is not convex, then there are points $P, Q \in F^+(ABC)$ and $R \notin F^+(ABC)$ such that R lies on the line segment $\overline{PQ}$ (see figure 3.23). If $R \in F^0(ABC)$ then

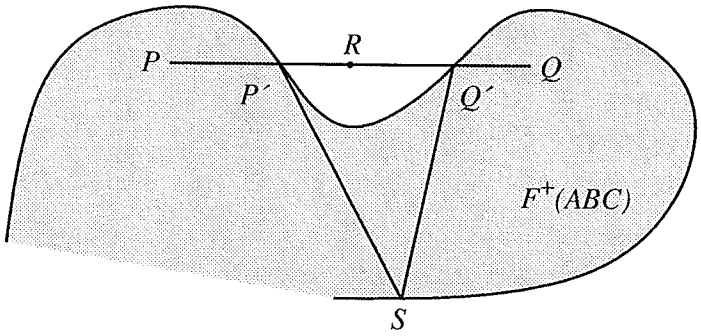


Figure 3.23: $F^+(ABC)$ is convex

we can find new PQR such that $P, Q \in F^+(ABC)$ and $R \in F^-(ABC)$. The segment $\overline{PR}$ goes from inside $F^+(ABC)$ to outside, so let P' be a point where it intersects $F^0(ABC)$, and Q' a point where $\overline{QR}$ intersects $F^0(ABC)$. Let $S \in F^0(ABC) \setminus PQ$. (If we can't find such a S then $F^0(ABC) = PQ$, which contradicts $R \in F^-(ABC)$.) Since R is on the edge of $P'Q'S$ and $R \in F^-(ABC) = F^-(P'Q'S)$ (by theorem 9) we can find an $R' \in P'Q'S \cap F^-(P'Q'S)$ which contradicts F being a flip rule. □

3.6.2 Rules with the circumscribing property are systematic and local

The proof that DT is systematic and local relies on the following geometric fact: If two circles share a common chord, then on each side of the chord, the interior of one circle is a subset of the interior of the other circle. We shall call this the *nesting property*.

If F has the circumscribing property then the same fact is true, provided we replace "circle" with "curve from $\mathcal{F}^0$". Consequently the same proof proves that rules with the circumscribing property are systematic and local.

Also, just as for the Delaunay triangulation we have the "empty-circle property"—the circumcircle of each Delaunay triangle contains no other site, for $GOT(F)$, F systematic

and local, we have the "empty-shape property"—the circumscribing curve for each triangle $GOT(F)$ contains no other site. We will call triangulations like $GOT(F)$ *empty-shape triangulations* in such cases.

3.6.3 The only rotation and translation-invariant systematic local flip rule is DT.

Definition. The two *asymptotes* of an unbounded curve are the limits of the support lines as the support point goes to infinity.

Theorem 11 *The only rotation and translation-invariant systematic local flip rule is DT.*

PROOF. Let F be such a rule and $K \in \mathcal{F}^0$.

Case 1 K is bounded. Fujiwara [128] and Bol [32] have shown that if K is a compact convex set which is not a disc, then it is possible to find an infinite number of congruent copies K' of K such that K and K' have at least four points in common on their boundaries. It follows from theorem 9 that $K = K'$. Hence K has an infinite number of symmetries and must be a disc.

Case 2 K is unbounded. Let A_1 and A_2 be the asymptotes to K, P their point of intersection, and α the angle between them. Rotate K by $\alpha/2$ about P, translate by d in direction A_1 and by d in direction $A_2 + \alpha/2$ to get K' (figure 3.24). By making

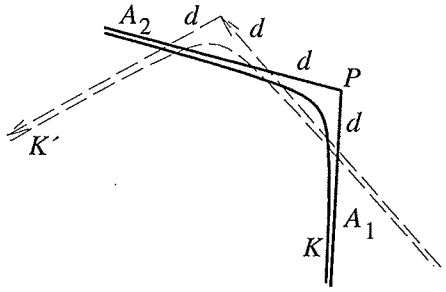


Figure 3.24: K and K' intersect three times

d sufficiently large we can ensure that A_2 is a chord of K'. Then the boundaries of K and K' intersect three times and by theorem 9 $K = K'$, *i.e.* $\alpha = 0$ and K is a half-plane, which we can regard as infinitely large disc.

Hence $F^0(ABC)$ is a circle for any three points A, B, and C. By theorem 9 this circle is the circumcircle of ABC and so $F = DT$. □

3.6.4 The only systematic local homothetic flip rules are generalized Delaunay rules.

A *homothetic flip rule* is one that is invariant under homotheties of the quadrilateral.

Notice that if F is a homothetic flip rule, $\mathcal{F}^0$ must be closed under homotheties, and since **either**(F) is a closed set, $\mathcal{F}^0$ is a closed set.

At this point it might seem that a systematic local homothetic rule F must necessarily be that of a Delaunay triangulation based on the convex distance function induced by the "circle" $F^0(ABC)$ [55, 91], but this is only true if $\mathcal{F}^0$ contains only one shape set.

Suppose $F^0(ABC)$ is a square (see figure 3.25). This is what you would expect in the

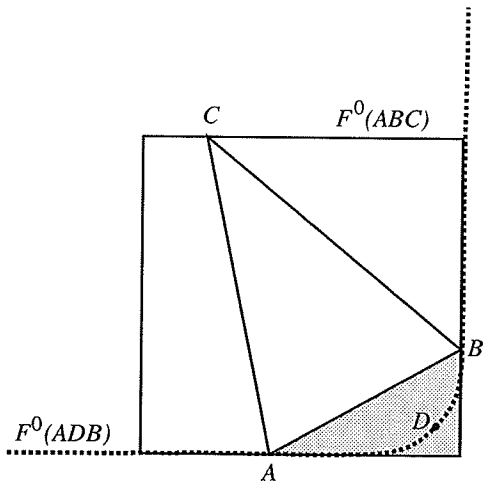


Figure 3.25: Two different F^0 curves

l_∞ metric. If D is in the shaded region in figure 3.25 then it is not possible to draw a scaled, translated copy of the square $F^0(ABC)$ through the points ADB. $F^0(ADB)$ must be a different shape, for example, the bottom right corner of a circle joined to an upwards ray and a leftwards ray.

In general, the $\mathcal{F}^0$ curves will be a collection of shape sets. We can complete the example with three more copies of $F^0(ABD)$ rotated through 90°, 180° and 270°. This corresponds

to a convex-distance-function Delaunay triangulation where the circle is a square with infinitesimally rounded corners.

Definition. A *cone* is a convex set with a boundary consisting of two rays. We will also consider an infinite strip with boundary a pair of antiparallel[6] lines a cone.

Note that if a convex set is invariant under a homothety, it must be a cone.

Theorem 12 *Let F be a homothetic systematic local flip rule. Then $K \in \mathcal{F}^0$ is either strictly convex or a cone.*

PROOF. Suppose K is not strictly convex. Let XY be a line segment on the boundary of K. Take A and B interior points of XY and C a point on the boundary of K not on the line XY. Then we can find a C' sufficiently close to C such that the line through C' parallel to CA intersects the segment XY at A' and the line through C' parallel to CB intersects the segment XY at B' (see figure 3.26).

Since the sides of the triangles ABC and $A'B'C'$ are parallel, there is a homothety that transforms ABC to $A'B'C'$. By theorem 9, K must be invariant under this homothety and hence must be a cone. □

Definition. We can order the support lines of a convex set by the angle they make with the x axis. If $0 \leq \alpha(l) < 2\pi$ is the angle between l and the x axis then we say $l < m$ if $0 < \alpha(m) - \alpha(l) < \pi$ or $0 < \alpha(m) - \alpha(l) + 2\pi < \pi$. If $\alpha(m) - \alpha(l) = \pm\pi$ then l and m are antiparallel and we do not define $<$ in this case.

Definition. If K is a convex set then $T_K^-(P)$ (the pretangent) is the minimum support line through P, and $T_K^+(P)$ (the posttangent) is the maximum support line through P.

Definition. P is a *corner* of K if $T_K^+(P) \neq T_K^-(P)$.

Definition. The *support cone* to K at a point P is the cone with sides $T_K^-(P)$ and $T_K^+(P)$. Note that this is $\lim_{k\to\infty} H(P,k)K$.

Definition. If K is an unbounded convex set, the *asymptote cone* to K is the cone with sides A_K^- and A_K^+, the asymptotes to K. Note that the asymptotes are the maximum and minimum support lines to K and if P is the intersection point of the asymptotes the asymptote cone is $\lim_{k\to 0} H(P,k)K$.

[6]Two parallel lines with opposite directions are *antiparallel.*

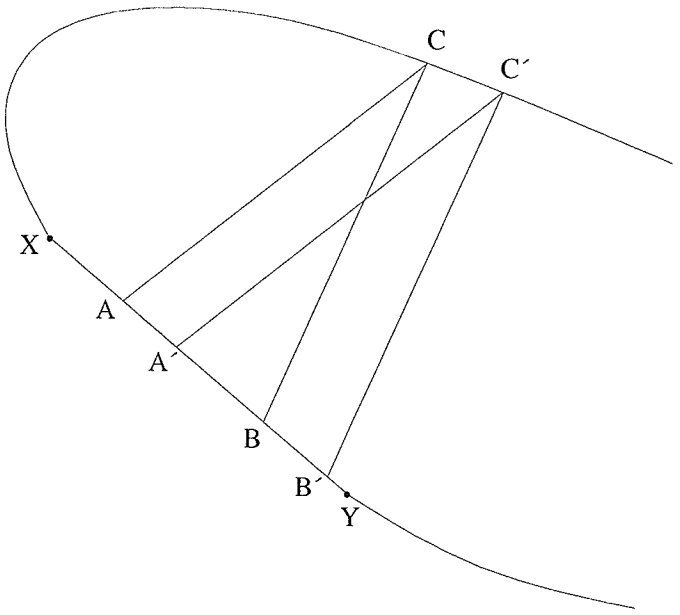


Figure 3.26: ABC is homothetic to $A'B'C'$

Definition. Two sets have a *common support line* if they each have a support line with the same direction.

Lemma 6 *Let F be a translation-invariant systematic local flip rule. If $K, K' \in \mathcal{F}^0$ have a common support line l at a point P and $T_K^-(P) < T_{K'}^-(P)$ then $T_K^+(P) \geq T_{K'}^+(P)$.*

PROOF. Suppose $T_K^+(P) < T_{K'}^+(P)$.. Then

$$T_K^-(P) < T_{K'}^-(P) \leq l \leq T_K^+(P) < T_{K'}^+(P)$$

Translate K' by a distance d in direction l (see figure 3.27).

For any value of d, the support cones of K and K' intersect. Since K and K' get arbitrarily close to their support cones as they get closer to P, for sufficiently small values of d K and K' will also intersect three times. By theorem 9, K' is a translate of K which contradicts $T_K^-(P) < T_{K'}^-(P)$. □

Definition. If P is on K then in a neighbourhood of P we define $P^+(K)$ to be the part of K that follows P as we traverse K in an anti-clockwise direction and $P^-(K)$ as the part

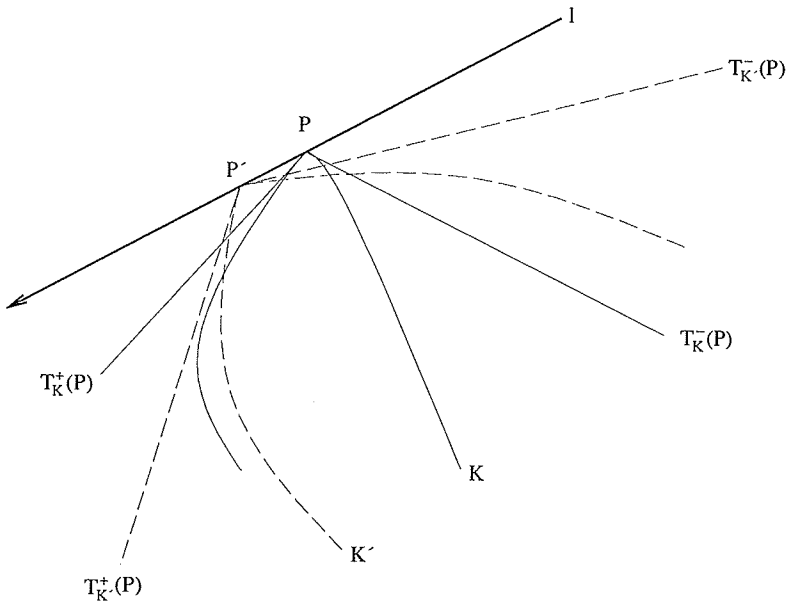


Figure 3.27: K and K' intersect three times.

that precedes P.

Definition. If K and K' are curves with a common support line l and K is a subset of the convex hull of K' when translated so that the support points coincide then we say $K \subset_l K'$.

Lemma 7 *Let F be a homothetic systematic local flip rule and $K, K' \in \mathcal{F}^0$. If $T_K^+(P) = T_{K'}^+(P)$ or $T_K^-(P) = T_{K'}^-(P)$ then K is homothetic to K', or one of K and K' is a cone. Furthermore, $K \subset_l K'$ or $K' \subset_l K$ where l is a common support line at P.*

PROOF.

Case 1 $T_K^-(P) > T_{K'}^-(P)$. In some neighbourhood of P, $P^-(K)$ is between $P^-(K')$ and $T_K^-(P)$ (see figure 3.28). If $I = P^+(K) \cap T_K^+(P) \neq \emptyset$ then K is not strictly convex and by theorem 12 must be a cone. Otherwise, we can find a k large enough so that $P^+(H(P,k)K')$ is between $P^+(K)$ and $T_K^+(P)$. $P^-(K)$ will still be between $P^-(H(P,k)K')$ and $T_K^-(P)$ for some neighbourhood of P. By a similar argument to that in lemma 6, K and K' are homothetic which contradicts $T_K^-(P) > T_{K'}^-(P)$. So if $T_K^-(P) > T_{K'}^-(P)$, K is a cone and $K' \subset_l K$.

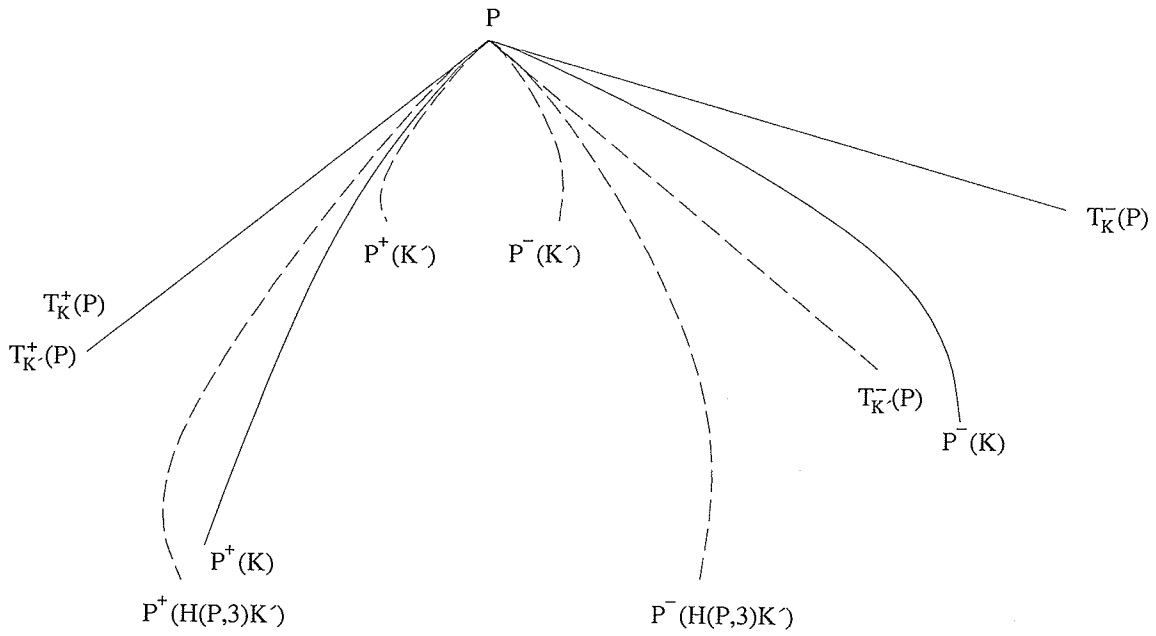


Figure 3.28: K and a translate of $H(P,3)K'$ intersect three times.

Case 2 $T_K^-(P) = T_{K'}^-(P)$. If both K and K' are cones, then obviously $K = K'$. If (say) K is a cone then $K' \subset_l K$ and K is the support cone of K' at P.

If neither are cones, relabel if necessary so that in some neighbourhood of P, $P^-(K)$ is between $P^-(K')$ and $T_K^-(P)$. As in case 1 we can find a k large enough so that $P^+(H(P,k)K')$ is between $P^+(K)$ and $T_K^+(P)$. However, in this case it is possible for $P^-(H(P,k)K')$ to also switch sides and be between $P^-(K)$ and $T_K^-(P)$. Nevertheless, unless $P^-(H(P,k)K')$ and $P^+(H(P,k)K')$ switch at the same value of k we can proceed as in case 1. If they do switch at the same value of k, then since $H(P,k)$ is continuous $P^-(H(P,k)K')$ intersects $P^-(K)$ and $P^+(H(P,k)K')$ intersects $P^+(K)$. So $H(P,k)K'$ intersects K three times and by theorem 9, K' is homothetic to K and $K' \subset_l K$. □

Lemma 8 *If two shapes in $\mathcal{F}'$ have two common support lines at different points and in different directions, then the shapes are homothetic.*

PROOF. Let the support lines be l and m with $m > l$ and support points P and

P'. We can translate and scale one of the shapes so that the support points coincide (see figure 3.6.4).

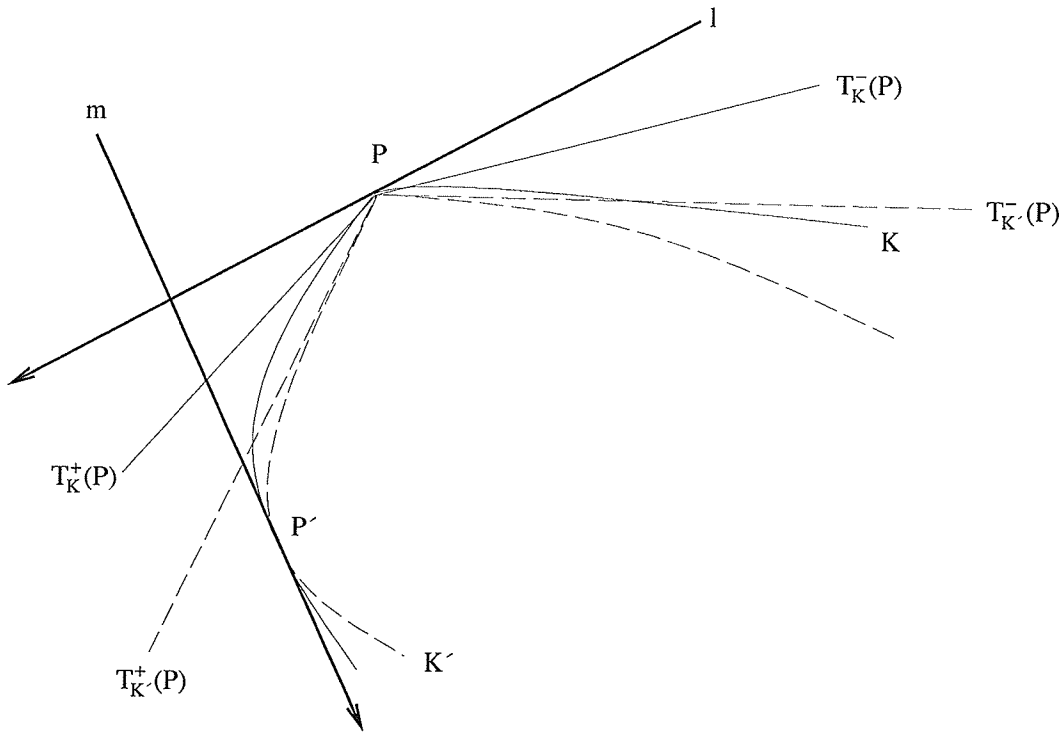


Figure 3.29: Two common support lines

Case 1 $T_K^-(P) = T_{K'}^-(P)$. By lemma 7 K and K' are homothetic or one is a cone.

If, say, K is a cone then PP' is one of the sides of the cone and one of the support lines. Since K' is convex, the segment PP' must be part of K'. K and K' have three common points and by theorem 9, $K = K'$ and the original shapes were homothetic.

Case 2 $T_K^-(P) > T_{K'}^-(P)$. Since $PP' > T_K^-(P)$ a small expansion of K' about P' produces two intersections of K' with K near P which combined with the intersection at P' means that by theorem 9 K is homothetic to K'. □

Definition. $S_F(l, P)$ is the set of curves in $\mathcal{F}^0$ through P with l as a support line.

Lemma 9 *Let F be a homothetic systematic local flip rule. If $K, K' \in \mathcal{F}^0$ have a common support line l then $K \subset_l K'$ or $K' \subset_l K$. That is, $S_F(l, P)$ is totally ordered by $\subset_l$.*

PROOF. From lemma 7 the result follows if $T_K^-(P) = T_{K'}^-(P)$. Otherwise, by lemma 6, $T_K^-(P) < T_{K'}^-(P) \leq l \leq T_{K'}^+(P) < T_K^+(P)$. So in a neighbourhood of P, $K \subset_l K'$. If K is not a subset of the convex hull of K' then K and K' must intersect at some point other than P. By a similar argument to case 2 in lemma 8, K is homothetic to K' and the result follows. □

Lemma 10 *Given a $P \in l$ and Q to the left of l, there exists $K \in S_F(l,p)$ such that $Q \in K$.*

PROOF. Let $K = \lim_{P \to P'} F^0(PP'Q)$, where $P \in l$. □

Lemma 11 *There is at most one bounded shape set in $\mathcal{F}^0$. If there is no bounded shape in $\mathcal{F}^0$, then $\mathcal{F}^0$ contains two antiparallel rays.*

PROOF. A closed bounded set has a support line parallel to any given direction; so lemma 8 shows that any two bounded shapes in $\mathcal{F}^0$ must be homothetic.

If there is no bounded shape in $\mathcal{F}^0$ then for any P and l the minimum (with respect to $\subset_l$) shape in $S_F(l,P)$ must be unbounded and pass through P and by lemma 10 must have an empty interior. Hence it must be a ray (degenerate cone) with direction r, say. This ray will be also be the minimum for l such that $-r < l < r$. The minimum for l such that $r < l < -r$ must be $-r$. □

Theorem 13 *If there is a finite number of shape sets in $\mathcal{F}^0$ then there is one bounded shape set with all the other shapes "rounding" off the corners of this shape.*

PROOF. Since there is only a finite number of shape sets in $S_F(l,P)$ it is meaningful to talk about the following shape set using the $\subset_l$ ordering. If K is not a cone then $H(P,k)K \subset_l H(P,k')K$ if $k < k'$; so the shape set preceding K is the asymptote cone of K and the following shape set is the support cone of K. The shape set following a cone cannot be another cone, since in that case a point in the region between the two cones will not have a K in $S_F(l,P)$ passing through it, contradicting lemma 10. Consequently the support cone of one shape is the asymptote cone of the following shape set (see figure 3.30.)

If $\mathcal{F}^0$ does not contain exactly one bounded shape set, lemma 11 shows that it contains a pair of antiparallel rays. In this case we can pretend that $\mathcal{F}^0$ contains a (bounded) digon (line segment) with sides parallel to the rays.

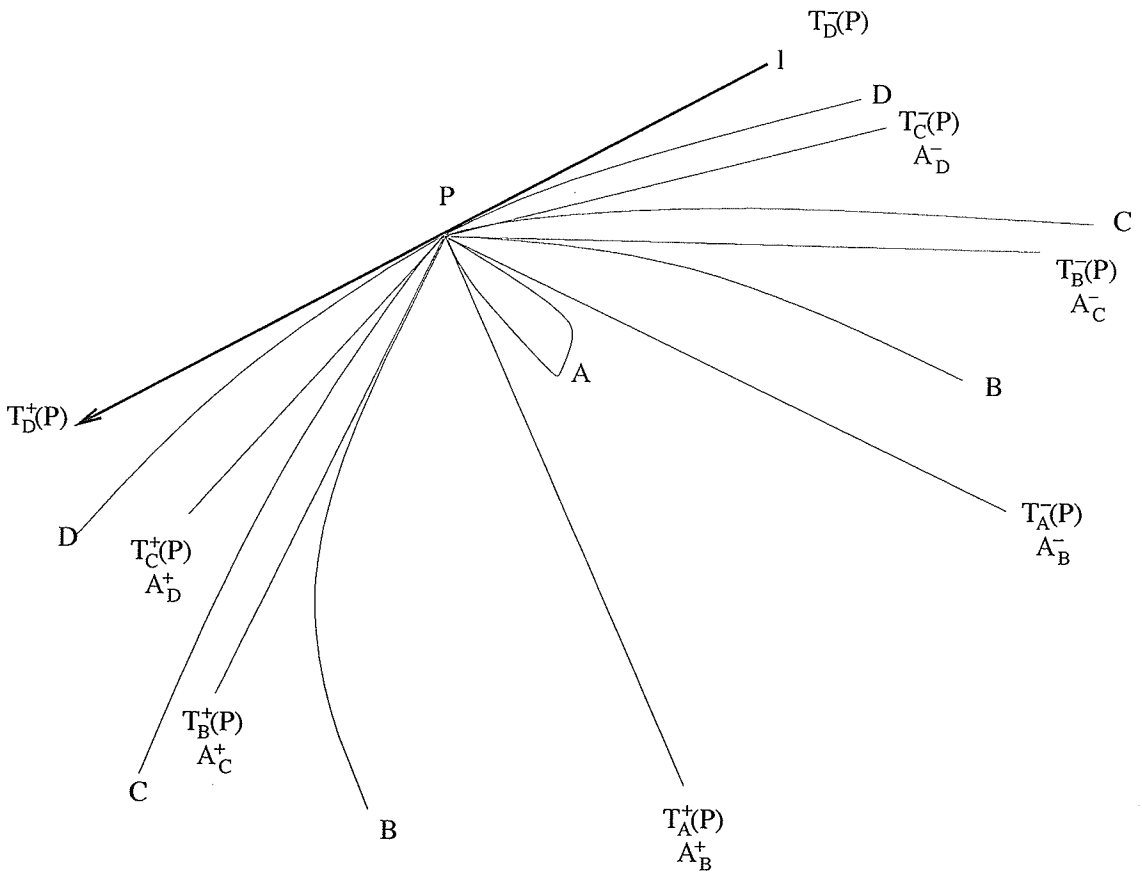


Figure 3.30: $S_F(l, P)$

So $\mathcal{F}^0$ contains exactly one bounded shape set. For each corner of this shape set there is a shape that "rounds" the corner, with asymptotes corresponding to the corner. If these shapes have corners then there are shapes to round their corners and so on. This describes all the shapes in $\mathcal{F}^0$. □

If $\mathcal{F}^0$ contains an infinite number of shape sets then the result is similar, except that it is possible to have an infinite number of cones to fill in the space between two cones.

How does this relate to convex-distance-function Delaunay triangulation? We can define the Euclidean Delaunay triangulation by the "empty-circle property"—the triangles of the Euclidean Delaunay triangulation are just those whose circumcircles contain no other site. Similarly, the convex-distance-function Delaunay triangulation can be defined by the "empty-ball property"—the triangles are those whose circumscribing balls are empty, where the ball is the unit circle for the convex distance function. Triangulations defined by systematic local flip rules further generalize this to the "empty-shape property"—the triangles are those whose circumscribing shapes are empty.

If the convex distance function is not strictly convex then theorem 12 tells us that the corresponding flip rule is not local and systematic. The problem here is that in this case triangles with a side parallel to a line segment on the boundary of the shape have more than one circumscribing shape and the Delaunay triangulation is not unambiguously defined. To resolve this ambiguity one particular circumscribing shape must be chosen (*e.g.* the bottom leftmost one [179]). This is effectively treating the flat part of the boundary as being very slightly curved, that is, the shape is strictly convex.

If the convex distance function has corners then the conditions of theorem 13 are violated because there is no shape that rounds the corners of the convex distance function. The problem here is that triangles with two sides that are support lines at the same corner of the convex distance function cannot be circumscribed and the convex-distance-function Delaunay triangulation does not completely triangulate the convex hull of the input sites. In this case we can add shapes to round the corners of the convex distance function and the convex-distance-function Delaunay triangulation is just a subset of the generalized one.

Finally we note that we can interpret our generalized convex-distance-function Delaunay triangulation as duals of Voronoi diagrams in the surreal [141] Cartesian plane, where the distance function is smooth (no corners) and strictly convex.

3.7 Conclusion

Locally optimized triangulations are simple to define and compute, while the systematic and local properties are important and useful properties for a flip rule to possess.

I have shown that the only translation and rotation invariant systematic local flip rule is the rule DT, and found that the Delaunay triangulation maximizes the mean inradius over all triangulations as well as several other geometric properties. This gives some more reasons to support the use of the Delaunay triangulation as the most natural and useful triangulation of a set of sites.

I have shown that the only homothetic systematic local flip rules correspond to empty-shape Delaunay triangulations, which generalize convex-distance-function Delaunay triangulations. Any Delaunay triangulation algorithm can be modified to produce empty-shape Delaunay triangulations and consequently convex-distance-function Delaunay triangulations.

These new simpler algorithms for convex-distance-function Delaunay triangulations are well suited for practical use.

Chapter 4

Computing Empty-Shape Triangulations

4.1 Two dimensions

Since empty-shape triangulations are systematic and local we can use the flip algorithm to compute empty-shape triangulations in worst-case time $O(n^2)$. Other algorithms for computing the Delaunay triangulation that use the incircle test, such as the divide-and-conquer algorithm [202] and the selection algorithm [229] will also compute empty-shape triangulations—it is only necessary to replace the incircle test with the F function.

The sweepline algorithm for Delaunay triangulation [122, 120] uses two different geometric tests:

1. Find the rightmost point of the circumcircle of three sites.

2. Determine whether a site is above or below the contact point of a circle through two other sites and tangent to the vertical line through the first site.

To implement a sweepline algorithm for empty-shape triangulation it is necessary to be able to compute these two primitives when "circle" is replaced by "curve from $\mathcal{F}^0$". The only difficulty arises for the first primitive when the curve is unbounded and has no rightmost point. This can be overcome in one of two ways:

1. By using surreal numbers [141] and using polynomials in ω to represent such points "at infinity".

2. By noticing that such events occur after all site events and "finite" circumcircle events. If we have events for ABC and BCD then ABC must occur before BCD just if $F(ABCD) = AC$. So F imposes a partial order on the infinite circumcircle events. It is sufficient to process the event queue in a way consistent with this partial order.

4.1.1 An implementation of the sweepline algorithm

Here is a complete Miranda [28, 315] implementation of the sweepline algorithm.[1]

Plane sweep algorithms operate by sweeping a vertical line across the plane from left to right.

Any plane sweep algorithm requires two data structures:

- The *sweep-line status* which represents the intersection of the sweepline with the geometry.
- A *priority queue* containing *events*, places where the status changes.

Priority Queue ADT

```
> abstype priority * **
> with push :: (priority * **) -> * -> (priority * **)
>      top :: (priority * **) -> *
>      pop :: (priority * **) -> (priority * **)
>      empty :: (*->**) -> (priority * **)
>      isempty :: (priority * **) -> bool
>      remove :: (priority * **) -> * -> (priority * **)
```

`empty` is given a function that defines the ordering of events and creates an empty queue. `isempty` determines if a queue is empty. `top` returns the smallest value from a queue, `pop` deletes that value. `push` inserts a value into the queue and `remove` deletes an arbitrary value from the queue.

The priority queue can be implemented[2] with a heap so that each operation takes time $O(\log n)$, where n is the number of values in the queue.

[1] The TEX source of this document is an executable Miranda program. Lines starting with > are Miranda source. All others are comments.

[2] appendix D contains an implementation.

Plane Sweep Algorithm

The algorithm processes events one at a time. The event processing function is given the current state and an event and returns a new state, a list of new events, a list of events to be deleted and a list of results.

```
> do_event_t * ** *** == * -> ** -> (*,[**],[**],[***])
```

The sweep function is also given a starting state, an initial list of events, and a function that defines the ordering of events. It returns the results of processing the events until there are no more events.

It uses an auxiliary function `sweep'` that is passed the current event queue.

```
> sweep :: do_event_t * ** *** -> * -> [**] -> (**->****) -> [***]
> sweep do_event start_state start_events event_priority
>  = sweep' do_event start_state q
>    where q = foldl push (empty event_priority) start_events

> sweep' :: do_event_t * ** *** -> * -> priority ** **** -> [***]
> sweep' do_event s q
>  = [], if isempty q
>  = r++sweep' do_event news newq, otherwise
>    where (news,newevent,delevent,r) = do_event s (top q)
>          newq = foldl push (foldl remove (pop q) delevent) newevent
```

Ordered Sequence ADT

We can use an ordered sequence ADT to represent the sweep-line status. Instead of being ordered by $<$, it is ordered by a transitive, irreflexive relation $\prec$ between elements and pairs of elements and also between two pairs of elements. If A, B and C are three successive elements in the sequence, the invariant is that $(A,B) \prec (B,C)$.

Elements can occur more than once in the sequence, but clearly a pair can occur once only.

```
> abstype ordered_seq_point
> with insert :: ordered_seq_point -> point
```

```
>                                    -> (ordered_seq_point,[point])
>      delete :: ordered_seq_point -> (point,point)
>                                    -> (ordered_seq_point,[point])
>      create :: ((point,point)-> point -> bool) -> [point]
>                                    -> ordered_seq_point
```

Inserting a new element X involves finding $\langle \ldots A, B, C \ldots \rangle$ such that $(A, B) \prec X$ and $(B, C) \not\prec X$ and replacing the sequence by $\langle \ldots, A, B, X, B, C, \ldots \rangle$. `insert` returns the new sequence and the list $[A, B, C]$.

Since elements can occur more than once in a sequence it is necessary to specify which occurrence by giving the successor of an element to be deleted, as well as that element. `delete` just deletes the first element of a pair (C, D) from the sequence $\langle \ldots, A, B, C, D, E, \ldots \rangle$ and returns the sequence $\langle \ldots, A, B, D, E, \ldots \rangle$ and the list $[A, B, D, E]$.

`create` creates the ordered sequence $\langle D, A, D' \rangle$ from $[D, A, D']$ and the $\prec$ relation.

The lists that `insert` and `delete` return contain just enough of the context of the insertion or deletion point to schedule events as will be seen below.

The ordered sequence can be implemented[3] using balanced trees so that each operation takes time $O(\log n)$ where n is the number of elements in the sequence.

Delaunay sweepline

Now we can apply the general sweepline algorithm to computing Delaunay triangulations.

Define STS(AB) (sweep tangent shape) of an edge AB to mean the shape through the two sites of the edge and tangent to the sweepline at a point to the right at the ends of the edge. (Note that there may be two shapes through an edge, tangent to a line.) The sweep tangent shape for a boundary edge will be empty, since if it contained a site, a Delaunay triangle to the right of the edge with a circumcircle to the left of the sweepline would exist.

Another way to think of the state is to consider the part of the Delaunay triangulation of the sites to the left of the sweepline that is guaranteed to be present, no matter how the sites to the right of the sweepline are arranged. The sweepline state is just the boundary of the external face of this partial triangulation.

The relation $\prec$ corresponds to the ordering of the contact points of the shapes tangent to the sweep line. $(A, B) \prec X$ means that the contact point for STS(AB) is below X on

[3] appendix D contains an implementation.

the sweepline.

Implementing this relation obviously depends on the shape. It can be done by computing the contact point, but it is simpler to find the circumscribing shape through A, B and X. Because of the nesting property, the contact point is below X iff the intersection of the circumscribing shape with the sweepline is below X. For example, if the shape is a circle, we can just test if the circumcentre of ABX is below X. (`circle_centre` computes the centre of the circle circumscribing three points.[4])

```
> point == (num,num)
> triangle == [point]
> below_func == triangle -> bool
> below_circle :: below_func
> below_circle [x,a,b] = snd(circle_centre [a,x,b])>snd x
```

Another way to implement this relation is to determine if the slope of the tangent to the circumscribing shape through A, B and X is positive or negative.

To turn a `below` function into a $\prec$ relation, we need to observe that `below` gives the right answer only if X is to the right of AB. Otherwise the contact point is below X if B is below A. `area` computes twice the signed area of a polygon,[5] and its sign is used to test if X is to the right of AB.

```
> before_func == (point,point) -> point -> bool
> makebefore :: below_func -> before_func
> makebefore below (a,b) x
>   = below [x,b,a], if area [x,b,a] > 0
>   = fst a > fst b, otherwise
```

There are two sorts of events that change the sweepline status.

- If ABC are three consecutive sites on the boundary, and the sweepline passes the rightmost point of the circumscribing shape for ABC, then ABC is a Delaunay triangle. The two edges on the boundary AB and BC must be replaced by the single edge AC. (This is what `delete` does.)

[4] Appendix D contains an implementation.
[5] Appendix D contains an implementation.

- If the sweepline passes over a site P, then we must find the site X on the boundary such that the shape passing through X and P and tangent to the sweepline is empty. The edges XP and PX must be added to the boundary. (This is what `insert` does.)

```
> delaunay_event ::= Site point| Circumshape triangle
```

We must ensure that there is a `Circumshape` event in the priority queue for each triple of consecutive sites ABC on the boundary where C is to the left of AB. So, whenever the boundary is changed, we must delete events for triples that are no longer consecutive and insert events for triples that are now consecutive.

```
> do_delaunay_event::
>            do_event_t (ordered_seq_point) delaunay_event triangle
> do_delaunay_event s (Site p)
>  = (news,acwC[[p,a,x],[y,a,p]],acwC[[y,a,x]],[[p,a]])
>    where (news,[x,a,y]) = insert s p
> do_delaunay_event s (Circumshape [c,b,a])
>  = (news,acwC[[c,a,x],[y,c,a]],acwC[[b,a,x],[y,c,b]],[[c,b,a]])
>    where (news,[x,a',c',y]) = delete s (b,c)
```

Notice that the list of results of `do_delaunay_event` is a singleton list of Delaunay triangles. (One for each `Circumshape` event.)[6]

`acwC` filters out triangles where the third point is not left of the first two (*i.e.* triangle vertices are not anticlockwise.)

```
> acwC :: [triangle] -> [delaunay_event]
> acwC ts = [Circumshape t | t <- ts; area t > 0]
```

`Site` events are scheduled when the scan line reaches the site. `Circumshape` events are scheduled when the scan line reaches the rightmost point of a circumscribing shape. `schedule` is polymorphic so that we can use real numbers (type `num`) or surreal numbers for ordering events.

```
> schedule_func * ==  delaunay_event -> *
```

[6]The code above also returns the Delaunay edge associated with each `Site` event—this is used to emphasize these edges in figure 2.23.

```
> schedule :: (point->*) -> (triangle->*) -> schedule_func *
> schedule rightmostp rightmost (Site p) = rightmostp p
> schedule rightmostp rightmost (Circumshape t) = rightmost t
```

If we use real numbers to order events, `rightmostp` will just take the x coordinate (and can be implemented with the standard function `fst`).

Implementing `rightmost` depends on the shape. For example, the rightmost Point of a circle can be computed by: (`d2` computes the Euclidean distance between two Points.[7])

```
> rightmost_circle :: triangle -> num
> rightmost_circle t
>  = cx+r
>    where (cx,cy) = circle_centre t
>          r = d2 (hd t) (cx,cy)
```

Finally, we can write a function to compute empty-shape Delaunay triangulation. It requires a list of sites, the $\prec$ relation and the `schedule` function. The only tricky part is creating the initial state. It simplifies the program to use sentinel sites directly above and below the leftmost site.

```
> sweep_geom_funcs * == (schedule_func *,before_func)
> sweep_delaunay :: sweep_geom_funcs * -> [point] -> [triangle]
> sweep_delaunay (sched,before) ps
>  = sweep do_delaunay_event
>          (create before [(ax,ay+1),(ax,ay),(ax,ay-1)])
>          (map Point rest)
>          sched
>    where (ax,ay):rest = sort ps
```

For example,

```
sweep_delaunay (schedule fst rightmost_circle, makebefore below_circle)
```

will calculate the Euclidean Delaunay triangulation.

[7] Appendix D contains an implementation.

Figures 4.1 and 4.2 show the sequence of events that occur when computing the empty-shape triangulation for a triangle and three hyperbolae that round off its corners. Circumshape events added to the queue are shown as dashed shapes, while those deleted are shown as dotted shapes.

These figures were produced by using a modified version of `sweep` that returns the sweepline state and the queue after each event and using the graphics system described in [190].

4.1.2 Computing convex-distance-function Delaunay triangulations

An algorithm for empty-shape Delaunay triangulation can be used to compute a convex-distance-function Delaunay triangulation. If the convex distance function has corners it is sufficient to add shapes that round the corners (see below) and compute the empty-shape Delaunay triangulation. This triangulation is a superset of the convex-distance-function Delaunay triangulation. The triangles in both triangulations are just those whose circumscribing shape is a homothet of the original convex-distance-function ball. The triangles that are in the empty-shape triangulation and not in the convex-distance-function triangulation are those whose circumscribing shape is one of the shapes used to round the corners. It is clear that a simple $O(n)$ pass over the empty-shape triangulation will produce the convex-distance-function triangulation.

It is interesting to compare these algorithms with previously published algorithms. The fact that the convex-distance-function may have corners complicates algorithms such as the Divide-and-Conquer algorithm given by Chew and Drysdale [55]. The merge step involves computing the bisecting curve between the two sets of sites. Corners can cause this curve to repeatedly head off to infinity and require the merge to be restarted. That is, the algorithm must be "fixed" to deal with the problems caused by corners. In contrast, using empty-shape Delaunay triangulations solves the problem by "fixing" the convex distance function to remove the corners, and leaves the algorithm unchanged.

Rounding Corners

How do we add shapes to round off the corners of a convex distance function? The simplest solution is to add an unbounded smooth shape with asymptotes equal to the support cone for each corner.

The simplest such shape is one branch of a hyperbola. The equation of the rectangular

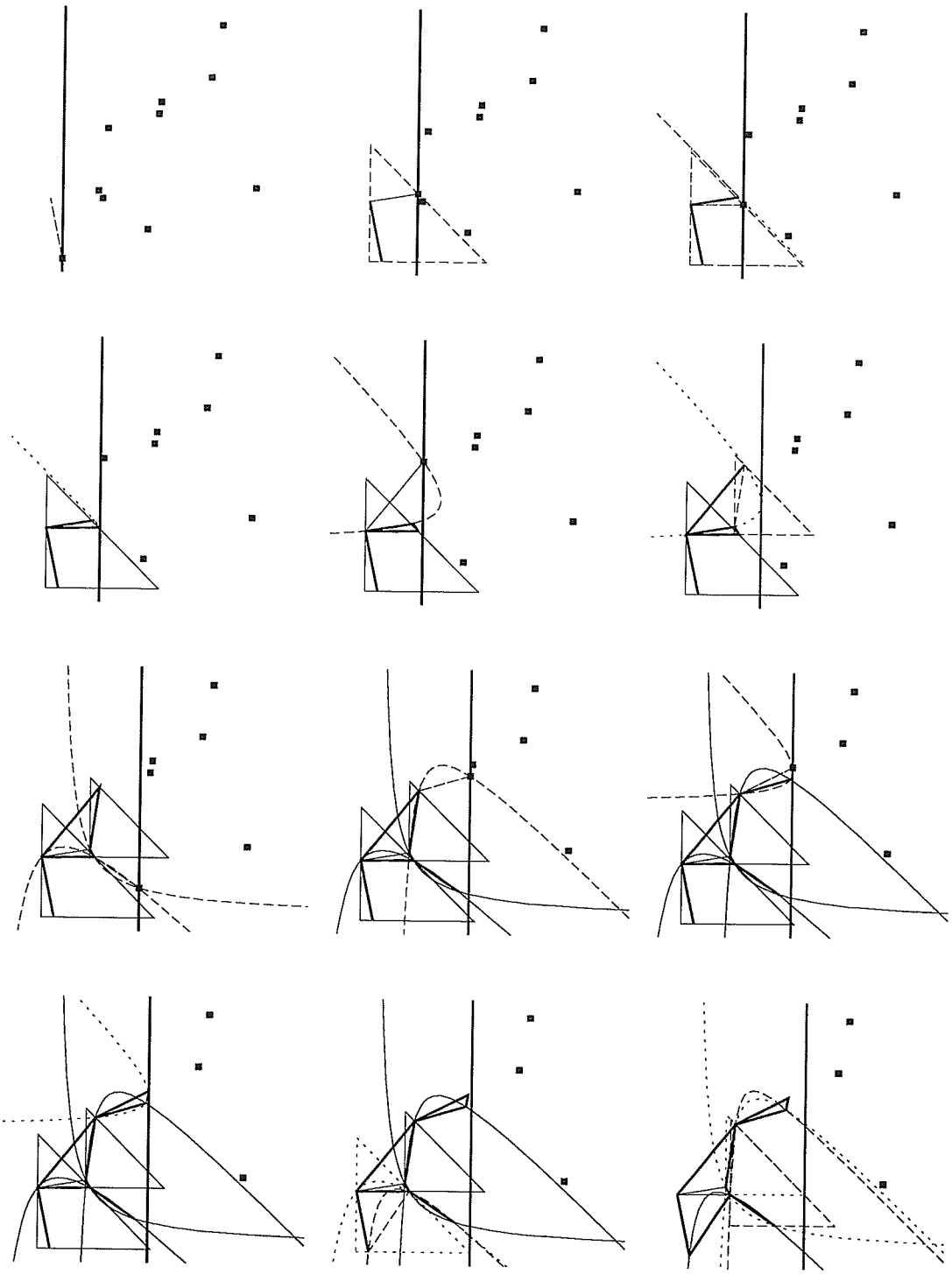


Figure 4.1: Sweep algorithm

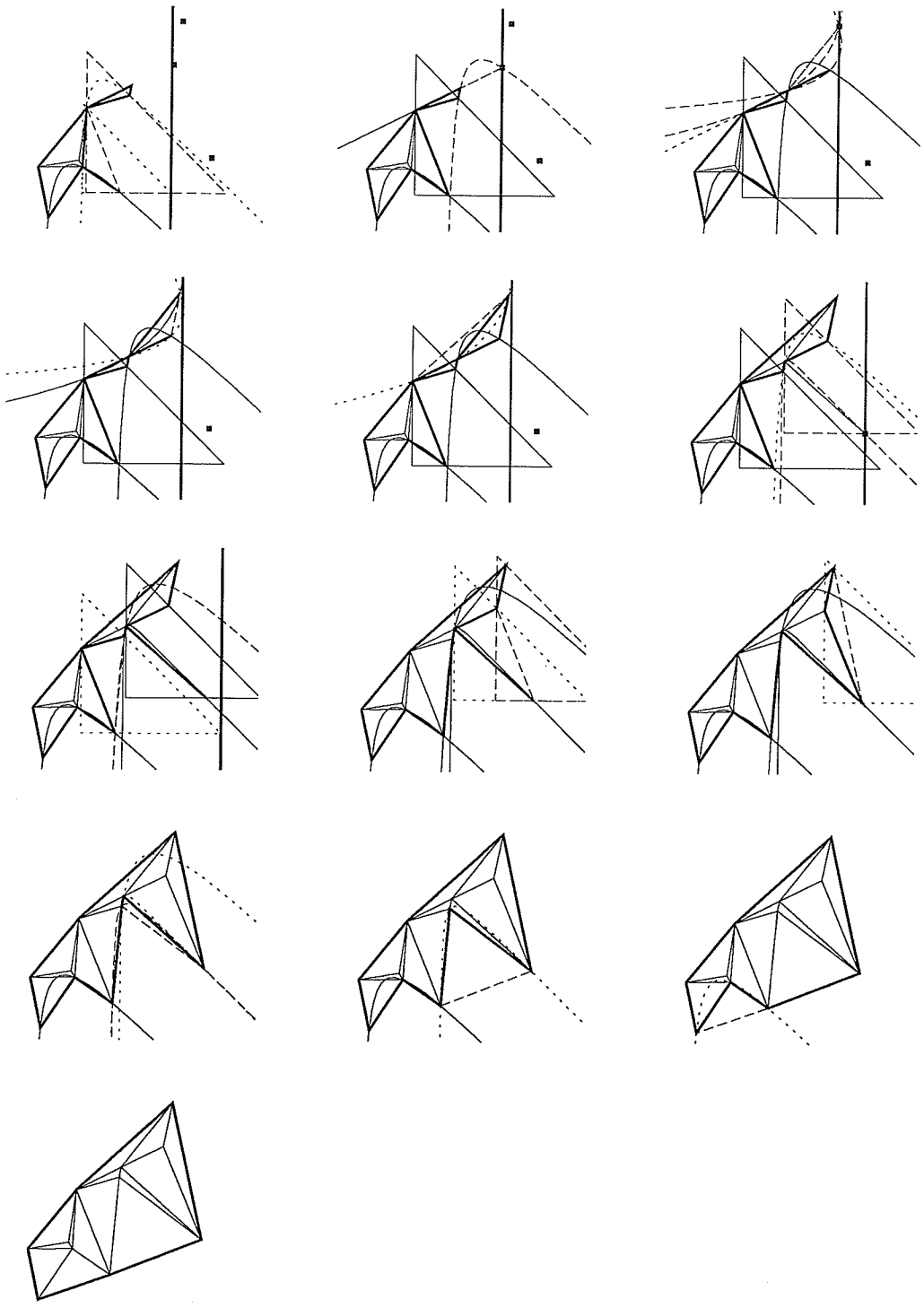


Figure 4.2: Sweep algorithm

hyperbola through $A = (x_A, y_A)$, $B = (x_B, y_B)$ and $C = (x_C, y_C)$ is

$$h(x,y) = \begin{vmatrix} xy & x & y & 1 \\ x_A y_A & x_A & y_A & 1 \\ x_B y_B & x_B & y_B & 1 \\ x_C y_C & x_C & y_C & 1 \end{vmatrix} = 0.$$

(Since this has the form $(x-a)(y-b) = r$ and passes through A, B and C.)

So, to round an arbitrary corner, it is sufficient to apply an affine transformation so that its support cone is mapped to the positive x and y axes and then test the sign of

$$\begin{vmatrix} x'_A y'_A & x'_A & y'_A & 1 \\ x'_B y'_B & x'_B & y'_B & 1 \\ x'_C y'_C & x'_C & y'_C & 1 \\ x'_D y'_D & x'_D & y'_D & 1 \end{vmatrix}$$

where A', B', C' and D' are the sites that the flip rule is being applied to after transformation.

There is one exception—this will not work if the sides of the corner are antiparallel rays. In this case the parabola with a vertical diameter

$$p(x,y) = \begin{vmatrix} x^2 & x & y & 1 \\ x_A^2 & x_A & y_A & 1 \\ x_B^2 & x_B & y_B & 1 \\ x_C^2 & x_C & y_C & 1 \end{vmatrix} = 0.$$

can be used instead of the hyperbola.

Implementation

If the sites A, B, and C cannot be circumscribed by the convex-distance-function "circle" the flip rule cannot return any of the values $\{AC, BD, \textbf{either}\}$. We must instead identify the corner of the "circle" that has caused the problem. This is the corner at which the support cone to the "circle" is contained in the support cone to the triangle ABC (see figure 4.3.)

Let us make this concrete with an example: we will implement the flip rule for the

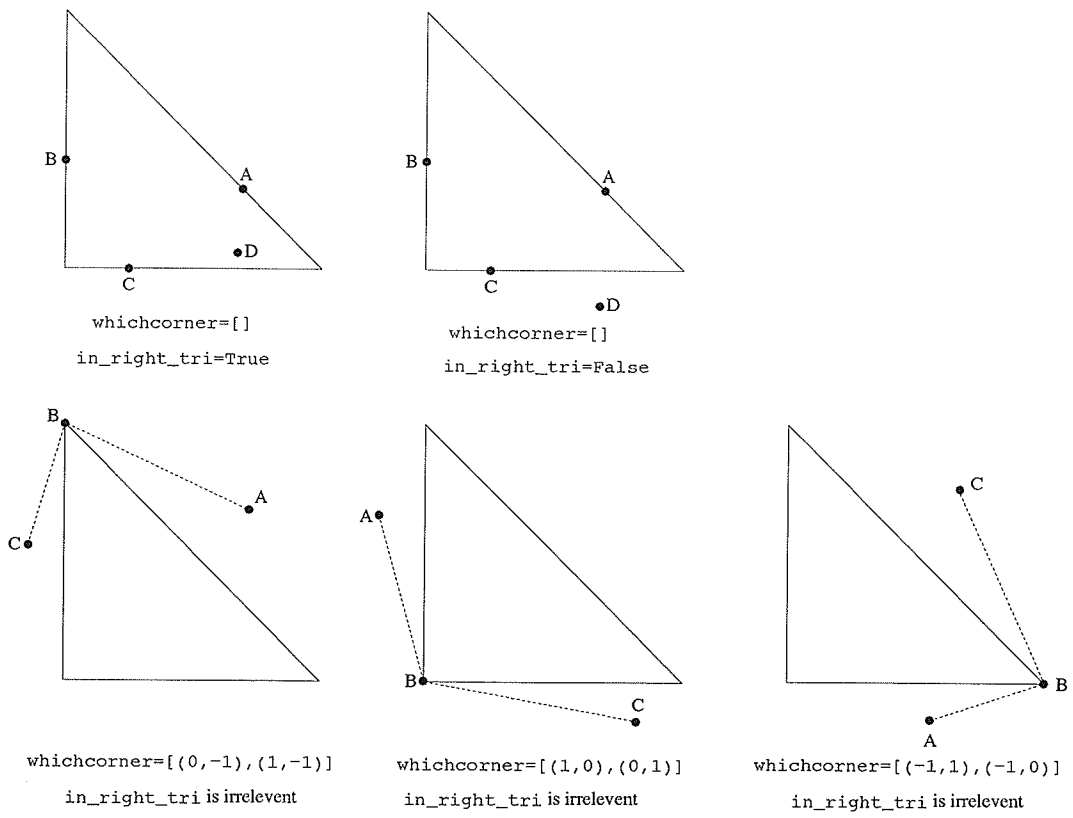


Figure 4.3: Possible values for `whichcorner[A,B,C]` and `in_right_tri[A,B,C,D]`

convex distance function where the "circles" are homothets of the right-angled triangle $X(0,0)Y(1,0)Z(0,1)$. Since we can find an affine transformation to transform any triangle into this right triangle, this lets us handle the convex-distance-function Delaunay triangulation for any triangular convex distance function.

First let us implement the flip rule for the case when the triangle ABC can be circumscribed by a homothet of the right triangle $X(0,0)Y(1,0)Z(0,1)$.

We merely find the smallest homothet that encloses ABC and test to see if D is inside.

```
> fliprule == [point] -> bool
> in_right_tri :: fliprule
> in_right_tri [(ax,ay), (bx,by), (cx,cy), (dx,dy)]
>  =  dx > w & dy > s & dx+dy < ne
>     where s = min [ay,by,cy] || bottom edge is y >= s
>           w = min [ax,bx,cx] || left edge is x >= w
>           ne = max [ax+ay,bx+by,cx+cy] || diag edge is x+y <= ne
```

Note that `in_right_tri` is not a well defined flip rule: figure 4.4 shows an example where `in_right_tri [a,b,c,d]=in_right_tri [d,a,b,c]`.

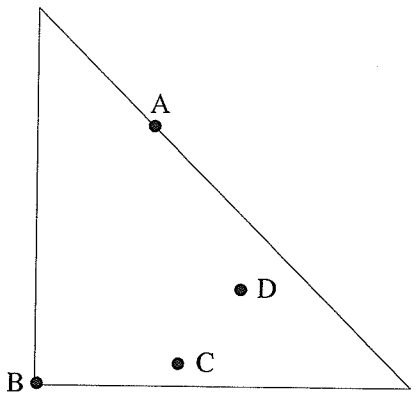


Figure 4.4: `in_right_tri` is not a flip rule

To test if ABC is circumscribable and to find the offending corner if it is not, requires another geometric primitive, `support_right_tri`. This finds the support cone to the right triangle at the support point of a given directed line AB. We will represent the cone by two vectors giving the direction of each ray.

```
> cone == [point]
```

support_right_tri can take on one of three possible values, depending on which of the three regions relative to A that B falls into (see figure 4.5). support_right_tri assumes that the triangle has a positive area.

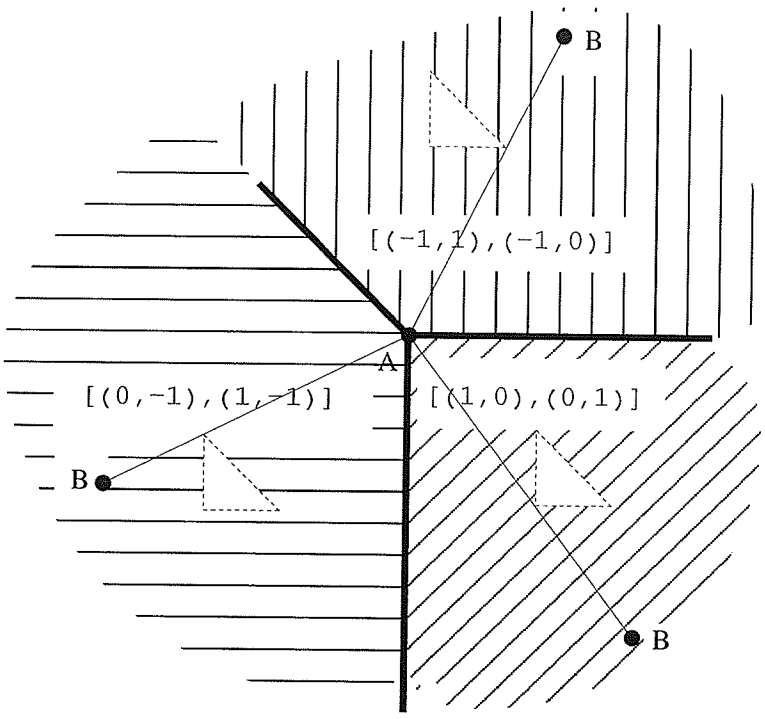


Figure 4.5: Regions for support_right_tri A B

```
> support_f == point -> point -> cone
> support_right_tri :: support_f
> support_right_tri (ax,ay) (bx,by)
>  = [(1,0),(0,1)], if bx>ax & by < ay
>  = [(0,-1),(1,-1)], if bx<ax & (bx+by<ax+ay)
>  = [(-1,1),(-1,0)], otherwise
```

For an arbitrary convex distance function with c corners, the `support` primitive can be implemented using binary search in time $\log c$. All that is needed is a list of the corners of the convex distance function.

It is interesting to compare our primitives with Drysdale's [91]. Instead of a `support` primitive he uses a `InInfCircle(p,a,b)` primitive. This returns `inside` if `p` is inside the homothet of the cone `support a b` that passes through `a` and `b`. The `support` primitive is more powerful since it can be used to implement the `InInfCircle` but not vice versa. However, it is unclear how `InInfCircle` could be implemented with computing the support cone.

Now we can identify the problem corner if ABC is not circumscribable. This will occur iff the support points for two sides of ABC are the same, and the duplicated support point will be the problem corner (see figure 4.3). We will use the empty list to represent the case where there is no problem corner. (We assume that the triangle has a positive area—our Delaunay triangulation algorithms produce positive triangles only.)

```
> whichcorner :: support_f -> triangle -> cone
> whichcorner support [a,b,c]
>   = supab, if supab = supbc \/ supab = supca
>   = supbc, if supbc = supca
>   = [], otherwise
>     where supab = support a b
>           supbc = support b c
>           supca = support c a
```

Now we can can use a hyperbola to round the corners. If the corner has support cone given by the vectors (α_x, α_y) and $\beta_x, \beta_y)$ then the affine transformation

$$T(x, y) = (\beta_y x - \beta_x y, -\alpha_y x + \alpha_x y)$$

will transform them so that they are parallel to the x and y axes. We then test if $T(D)$ lies inside the rectangular hyperbola through $T(A)T(B)T(C)$.

```
> round_corners :: support_f -> fliprule -> fliprule
> round_corners sup flip quad
>   = flip quad, corn=[]
```

```
>   = (affine_flip corn in_hyperbola) quad, otherwise
>     where corn = whichcorner sup (take 3 quad)
```

`affine_flip [a,b]` applies an affine transformation (the one that maps the vector **a** onto the x axis and **b** onto the y axis) to a flip rule.

```
> affine_flip :: cone -> fliprule -> fliprule
> affine_flip corn = ( . (map (affine corn)))
```

`affine` applies an affine transformation to a point.

```
> affine :: cone -> point -> point
> affine [(ax,ay),(bx,by)] (x,y) = (by*x-bx*y, -ay*x+ax*y)
```

Testing to see if a point is inside a rectangular hyperbola through three points just involves evaluating the sign of a determinant, and is very similar to testing to see if a point is inside a circumcircle.

Note that it is always possible to fit a rectangular hyperbola through three non-collinear points, but that the points may lie on different branches. If we label the points such that $x_A \leq x_B \leq x_C$ then for all the points to lie on the same branch we must have $y_A \geq y_B \geq y_C$. It is not hard to see that this will always be the case with the points used here.

`hdist` is the analogue of the Euclidean distance function.

```
> in_hyperbola :: fliprule
> in_hyperbola [a, b, c, d]
>   =  hdist hc d > hdist hc a
>      where hc = hyperbola_centre [a,b,c]

> hdist :: point -> point -> num
> hdist (ax,ay) (bx,by) = (ax-bx)*(ay-by)

> hyperbola_centre :: [point] -> point
> hyperbola_centre t
>   = (cx,cy)
>     where ds = map d t
>           xs = map fst t
```

```
>           ys = map snd t
>           divisor = area t
>           cx = area (zip2 xs ds)/divisor
>           cy = area (zip2 ds ys)/divisor
>           d (x,y) = x*y
```

Finally, we can put all the pieces together. `delaunay` is given a flip rule and a list of sites and computes the Delaunay triangulation using any algorithm that uses a flip rule as the geometric primitive (selection [229], incremental [202], flip [207] or Divide-and-Conquer [202] algorithms).

We round the corners, compute the empty-shape triangulation, and extract the triangles that can be circumscribed.

While this finds all the triangles of the convex-distance-function Delaunay triangulation, the triangulation may include edges that are not part of any triangle (see figure 4.6).

Referring back to figure 4.3, if we use B to label the corner of the triangle ABC whose support cone contains the support cone of the "circle", it is evident that every "circle" with AC as a chord must include B in its interior. This means that the edge AC is not in the Delaunay triangulation. Furthermore, the nesting property implies that all "circles" through AB contain no site on the same side of AB as C. Consequently it is only necessary to examine the triangles on either side of an edge to determine if that edge is Delaunay.

The edge that is not a Delaunay edge is the one whose support point is different from the other two.

```
> edge == [point]
> nondelaunay :: support_f -> triangle -> edge
> nondelaunay support [a,b,c]
>   = [c,a], if supab = supbc
>   = [b,c], if supab = supca
>   = [a,b], if supbc = supca
>   = [], otherwise
>     where supab = support a b
>           supbc = support b c
>           supca = support c a
```

The convex-distance-function Delaunay triangulation can now be calculated by removing

all the non-Delaunay edges from the augmented triangulation.

```
> cdf_delaunay :: support_f -> fliprule -> [point] -> [triangle]
> cdf_delaunay sup flip ps
>  = filter circumscribable ts ++
>    (potential_edges -- nonedges) -- map reverse nonedges
>    where circumscribable t = nondelaunay sup t = []
>          ts = delaunay (round_corners sup flip) ps
>          nontriangles = filter ((~).circumscribable) ts
>          nonedges = map (nondelaunay sup) nontriangles
>          potential_edges = concat(map explode nontriangles)
>          explode [a,b,c] = [[a,b],[b,c],[c,a]]
```

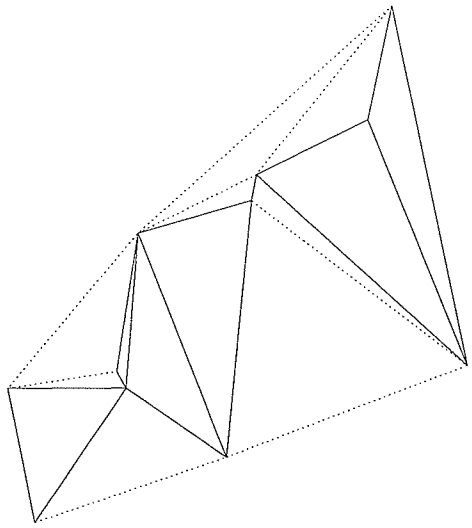


Figure 4.6: cdf Delaunay triangulation for a right triangle

For example, the Delaunay triangulation for our right-triangle distance function is computed with `cdf_delaunay support_right_tri in_right_tri`.

Figure 4.7 shows the working of the selection algorithm [229] for our example distance function. The selection algorithm starts with a hull edge and finds the Delaunay triangle standing on this edge by searching all the other sites. This gives two (possibly new) edges to which the algorithm can be recursively applied. Figure 4.7 also shows the empty circle

for each triangle as it is discovered.

Figure 4.6 shows why the selection algorithm will not work with the original (unrounded) convex distance function: hull edges may not belong to the triangulation, the dual graph of the triangulation may not be connected, and there may be edges that do not belong to any triangle.

Implementation of primitives for the sweepline algorithm

We need two primitives: find the x coordinate of the rightmost point of the "circle" through three points, and test whether a point is on top of the "circle" through that point and two other points.

Here they are for our right-triangle example: (It is simpler to write one function that returns the results of both primitives for a triangle.)

Naturally these are only well defined if the three points are circumscribable by the right triangle.

```
> sweep_primitives * == triangle -> (*,bool)
> sweep_prim_right_tri :: sweep_primitives num
> sweep_prim_right_tri [(ax,ay), (bx,by), (cx,cy)]
>  =  (ne-s, ay=s)
>     where s = min [ay,by,cy] || bottom edge is y >= s
>           ne = max [ax+ay,bx+by,cx+cy] || diag edge is x+y <= ne
```

Just as with flip rules, we can use hyperbolae to round the corners. We will use surreals for the rightmost points of hyperbolae with no real rightmost point. We only need to deal with surreals of the form $a\omega + b$, so we will just use ordered pairs like (a, b) to represent them. (This means that we can use Miranda's > operator which uses lexicographic ordering to compare surreals.)

```
> surreal == (num,num)
> to_surreal :: num -> surreal
> to_surreal x = (0,x)
```

round_corners_sweep takes a partial sweep-primitives function and creates a total one. If the triangle is circumscribable (*i.e.* there is no problem corner) then we just use the supplied partial function, converting the rightmost value to a surreal.

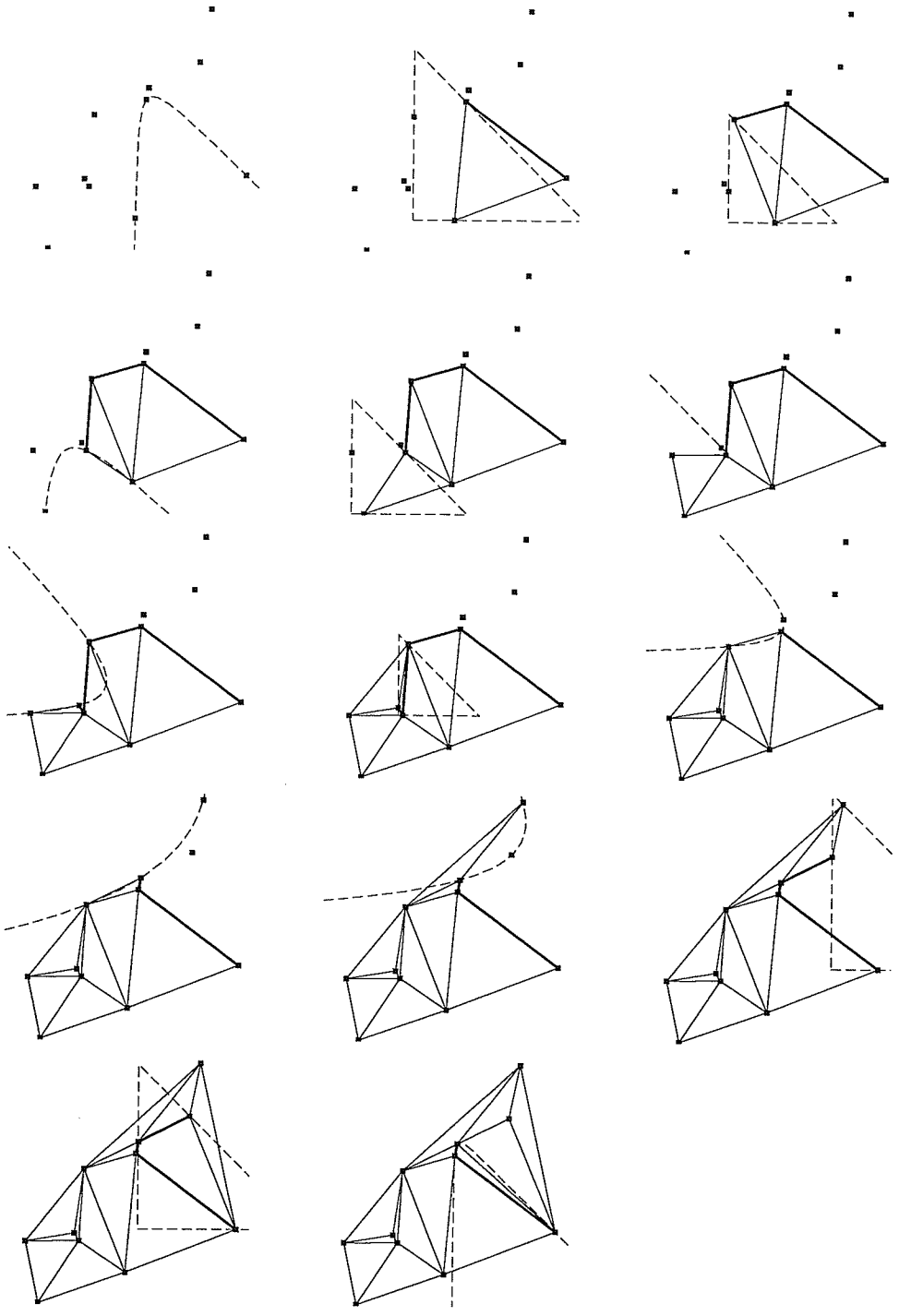


Figure 4.7: Selection algorithm for right triangle with rounded corners (solid lines)

```
> round_corners_sweep :: support_f -> sweep_primitives num
>                                        -> sweep_primitives surreal
> round_corners_sweep sup sweep_prim tri
>   = applypair (to_surreal,id) (sweep_prim tri), corn=[]
```

Otherwise, we need to circumscribe the triangle with a hyperbola with asymptotes parallel to the sides of the corner. We apply a transformation to the triangle (giving `tri'`) so that the asymptotes are transformed to x and y axes.

```
>   = (rightmost,tax>0), otherwise
>     where
>     corn = whichcorner sup tri
>     tri' = map (affine corn) tri
```

$(x-h_x)(y-h_y) = r$ is the equation of the rectangular hyperbola through the transformed points. We will call the hyperbola that it is the transformation of, the 'original hyperbola'.

```
>     (hx,hy) = hyperbola_centre tri'
>     r = hdist (hx,hy) (ax,ay)
>     (ax,ay) = hd tri'
```

The tangent vector to the rectangular hyperbola at the point (x, y) is

$$((x - h_x)^2, -r).$$

A point is below the original hyperbola if the tangent at that point lies in the first or fourth quadrant, that is, the x component is positive. We need to use the inverse transformation to convert a tangent to the rectangular hyperbola to a tangent to the original hyperbola.

```
>     (tax,tay) = iaffine corn ((ax-hx)^2,-r)
```

The rightmost point of the original hyperbola has a vertical tangent. The corresponding point on the rectangular hyperbola will have tangent (t_x, t_y), the transformed vertical tangent. This point satisfies $(x - h_x)^2 = -r * t_y/t_x$. If this equation has a solution, then the inverse transformation gives the rightmost point of the original hyperbola. Otherwise,

the original hyperbola does not have a real rightmost point. We can arbitrarily choose ω as the rightmost point of an original hyperbola[8] with centre $(0,0)$ and radius $r = 1$. Another original hyperbola with centre (x,y) and radius r, will have rightmost point $r\omega + x$. For the sweepline algorithm, we can just use $r\omega$ since this will only make a difference if two hyperbolae have the same radius, and the ordering of `Circumshape` events only matters for triangles with a common edge. Clearly, if two homothetic hyperbolae have two points in common and the same radius, then they are identical. By construction it is not possible for two non-homothetic hyperbolae to have two points in common.

```
>     (tx,ty) = affine corn (0,1)
>     x'2 = -r*ty/tx
>     (x,y) = (sqrt x'2 + hx,hy + r/(x-hx))
>     rightmost = (r,0), tx=0 \/ x'2<=0
>               = to_surreal (fst(iaffine corn (x,y))), otherwise
```

`iaffine` is just the inverse of `affine`.

```
> iaffine [(ax,ay),(bx,by)] (x,y)
>    = ((ax*x+bx*y)/det, (ay*x+by*y)/det)
>      where det = ax*by - bx*ay
```

Finally, we can use the primitives to produce the geometric functions required by `sweep_delaunay`.

```
> round_geom_funcs :: support_f -> sweep_primitives num
>                                      -> sweep_geom_funcs surreal
> round_geom_funcs sup sweep_prim
>    = (schedule (to_surreal.fst) rightmost, makebefore below)
>      where rightbelow = round_corners_sweep sup sweep_prim
>            rightmost = fst.rightbelow
>            below = snd.rightbelow
```

For example, the Delaunay triangulation for our right-triangle distance function (see figures 4.1 and 4.2) can be computed with

[8] Assuming that is, we are calculating the Delaunay triangulation of sites with real coordinates. If they have surreal coordinates, more careful calculation is required.

```
sweep_delaunay (round_geom_funcs support_right_tri sweep_prim_right_tri)
```

Looking at these figures should make it clear why it was necessary to round the corners. Consider what it would be necessary to do to use a sweep algorithm to directly calculate the convex-distance-function Delaunay triangulation. The triangles whose circumscribing shape have no real rightmost point can be safely discarded, but ignoring the ones that are unbounded on the left causes the partial Delaunay triangulation to be disconnected into potentially n pieces. The sweep algorithm uses binary search on the boundary of the partial Delaunay triangulation to find a Delaunay edge connecting newly encountered sites to the partial triangulation in time $O(\log n)$. In order to maintain this, the disconnected pieces would have to be organized into a data structure with similar properties. But this is precisely what the triangles with circumscribing shapes unbounded on the left do.

4.1.3 Bounding unbounded "circles"

If the "circle" for the convex distance function is unbounded there are further difficulties. Not all the sites may be included in the triangulation (see figure 4.8). No previous published algorithm correctly deals with this in the general case, though for the case where the "circle" is a right-angle cone with sides the positive x and y axes the Delaunay triangulation is the set of maxima of the sites [189], joined together in x-sorted order. (And there are no triangles in the triangulation, just edges!)

To deal with unbounded "circles", as well as adding shapes to round the corners, we need to add a shape with a corner whose support cone is equal to the asymptote cone of the unbounded "circle".

If we transform the sides of the asymptote cone onto the x and y axes, then a right triangle can be used as this shape.

The only exception occurs if the sides of the asymptote cone are antiparallel (for example, a parabola). In this case an infinite strip (a digon) with another shape to round the other corner will work.

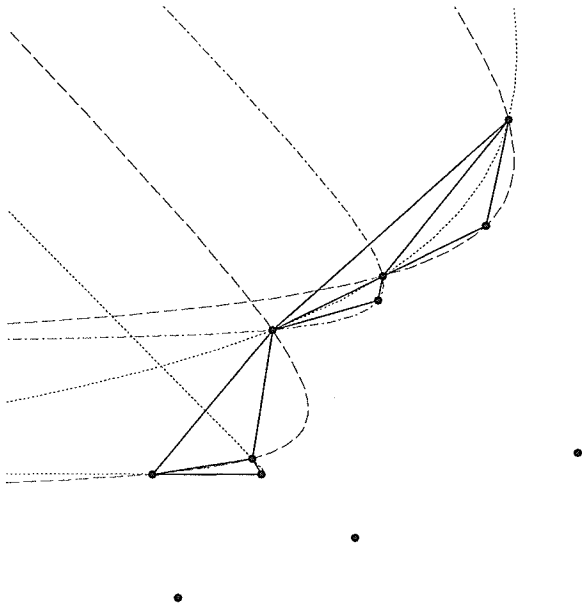


Figure 4.8: Delaunay triangulation where "circle" is a hyperbola with asymptotes $x + y = 0$ and $y = 0$.

4.1.4 Constrained empty-shape Delaunay triangulations

Constrained empty-shape Delaunay triangulations can be defined in exactly the same way as constrained Delaunay triangulations, and any algorithm that computes constrained Delaunay triangulations can be used to compute constrained empty-shape Delaunay triangulations.

Just as in the previous section it is also possible to compute constrained convex-distance-function Delaunay triangulations using Incremental (*e.g.* [63]), Selection (*e.g.* [216]), Flip (*e.g.* [33]), Divide-and-Conquer (*e.g.* [57]) or Sweepline algorithms (*e.g.* [291]). All that is required is changing the appropriate geometric primitive.

4.2 Three or More Dimensions

The results of section 3.6 extend into three or more dimensions. The triangulation of a set of sites in space consists of a division of the convex hull of the sites into tetrahedra whose vertices belong to the set. A pair of tetrahedra which form a convex pentahedron can be

exchanged for three tetrahedra that form a convex pentahedron and vice versa, so we can define flip rules in a natural manner (slightly complicated by the fact that the number of tetrahedra is not fixed).

Now consider a site set where all but one of the sites are co-planar. Every tetrahedron in the triangulation of this set will have as one vertex the non-planar site. The sides of the tetrahedra opposite this site form a triangulation in the plane and the space flip rule gives a flip rule in the plane, so if the flip rule is systematic and local it must be a generalized Delaunay rule.

In higher dimensions it is still true that if a triangulation is locally Delaunay then it globally Delaunay [94]. However, Joe [162] has shown than even in three dimensions DT is not systematic. This is because it is possible for a triangulation to not be locally Delaunay but impossible to improve by flipping. If the flipping is done incrementally, that is sites are added one at a time and flips used to construct a LOT after each site is added, then Edelsbrunner and Shah [99] show that the result is always the Delaunay triangulation.

4.2.1 Higher-Dimensional Convex Distance Functions

This result does not extend to higher-dimensional convex-distance-function Delaunay triangulations. It is possible for a triangulation to be locally Delaunay but not globally Delaunay. Here is an example in $\mathbf{R}^3$ using the l_∞ metric:

Take $A[6,6,2]$ $B[10,4,8]$ $C[0,2,6]$ $D[2,12,0]$ $E[4,8,10]$ $F[8,0,4]$ (see figure 4.9) and consider the diagram:

			distance from centre					
cell	centre	radius	A	B	C	D	E	F
ACEF	[4,4,6]	4	4	6	4	8	4	4
ACDE	[1,7,5]	5	5	9	5	5	5	7
ABEF	[6,4,6]	4	4	4	6	8	4	4
BCEF	[5,3,9]	5	7	5	5	9	5	5
BCDF	[6,6,2]	6	0	6	6	6	8	6

This is locally Delaunay but not Delaunay. (Since $BCDF$ has A in its circumball but is adjacent only to $BCEF$ and E is outside $BCDF$'s circumball.)

Schaudt and Drysdale [285] give an incremental algorithm that computes the Delaunay triangulation for convex distance functions in d dimensions provided the set of sites is "non-degenerate". They defined "non-degenerate" to mean that each set of $d+1$ sites had a unique

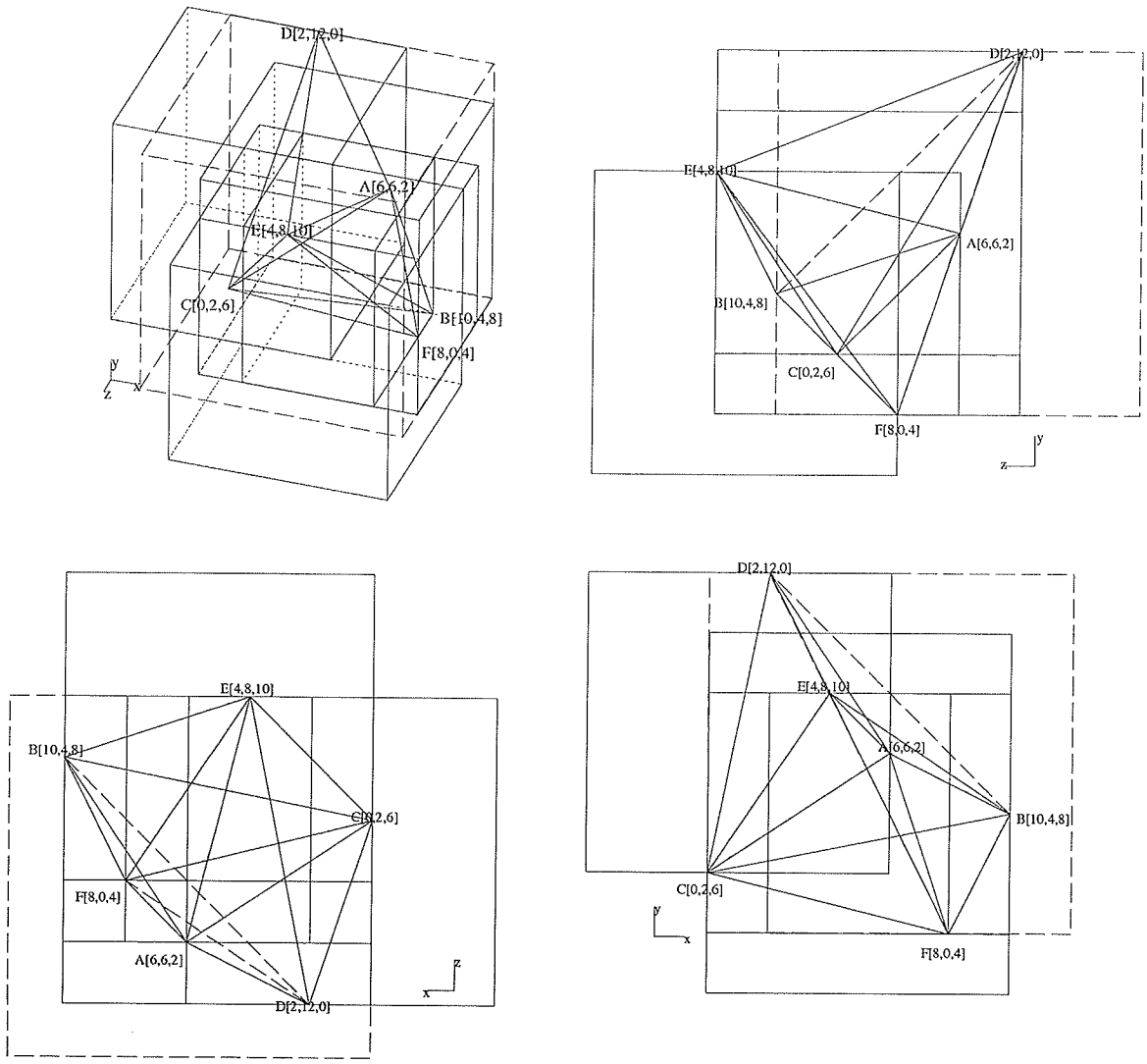


Figure 4.9: Locally Delaunay but not globally Delaunay

circumball. Unfortunately, this definition of "non-degenerate" is too restrictive. Consider the site set (again, in $\mathbf{R}^3$ using the l_∞ metric) $A[1,1,7]$ $B[5,3,0]$ $C[3,5,8]$ $D[8,6,1]$ (see figure 4.10). This has two circumballs, both of radius 4 with centres $[4,5,4]$ and $[5,2,4]$, so is "degenerate" by Schaudt and Drysdale's definition. However, any small perturbation of the sites $ABCD$ still has two circumballs, so the set of "degenerate configurations" does not have measure 0.[9] We will call a configuration like $ABCD$ *pseudo-degenerate*.

Lê [196] gives a formal definition of "non-degeneracy" for site configurations in $\mathbf{R}^d$ with respect to a convex distance function.

Note that the existence of a pseudo-degenerate configuration does not depend on the l_∞ ball being non-smooth and non-strictly convex. We can perturb the ball slightly so that it is smooth and strictly convex and still have a configuration like $ABCD$.

A large set of randomly chosen sites will almost certainly contain a pseudo-degenerate configuration and the Schaudt-Drysdale algorithm will fail.

Icking *et al.* [159] give balls in $\mathbf{R}^3$ such that there is a tetrahedron with c circumballs for any $c > 1$.

Theorem 14 *The maximum number of circumballs of a non-degenerate configuration of $d+1$ sites in $\mathbf{R}^d$ with the l_∞ metric is $2^{\lfloor \frac{d-1}{2} \rfloor}$.*

PROOF. Let $s_0, s_1, \ldots, s_d$ be the sites, and WLOG take s_0 and s_1 as the pair of sites that are the furthest from each other and $d(s_0, s_1) = s_{01} - s_{11} = 2r$, where r is the radius of the circumball. Each site lies on a different one of the $2d$ faces of the circumball. (If two sites lie on the same face, then the configuration is degenerate, since a small perturbation of the sites destroys this property.) We call the two faces orthogonal to the ith coordinate the min-i face and the max-i face, so s_0 lies on the max-1 face, and s_1 lies on the min-1 face. It is not possible for any other pair of faces to have sites on both faces of the pair. (Such a configuration would be degenerate since the sites in question would be distance $2r$ from each other.) So, one of each of the $d-1$ remaining sites lies on one of each of the remaining pairs of faces. This gives us a bound of 2^{d-1} on the number of circumballs through the sites.

To improve this to $2^{\lfloor \frac{d-1}{2} \rfloor}$, note that a site can only lie on the max-i face if it is the i-maximum (has the largest ith coordinate of the sites). The number of choices for the face that a site lies on is just the number of is for which the site is a maximum or a minimum.

[9] Numerical experiments indicate that four sites taken from the uniform distribution inside the unit cube have two l_∞ circumballs 2.6% of the time.

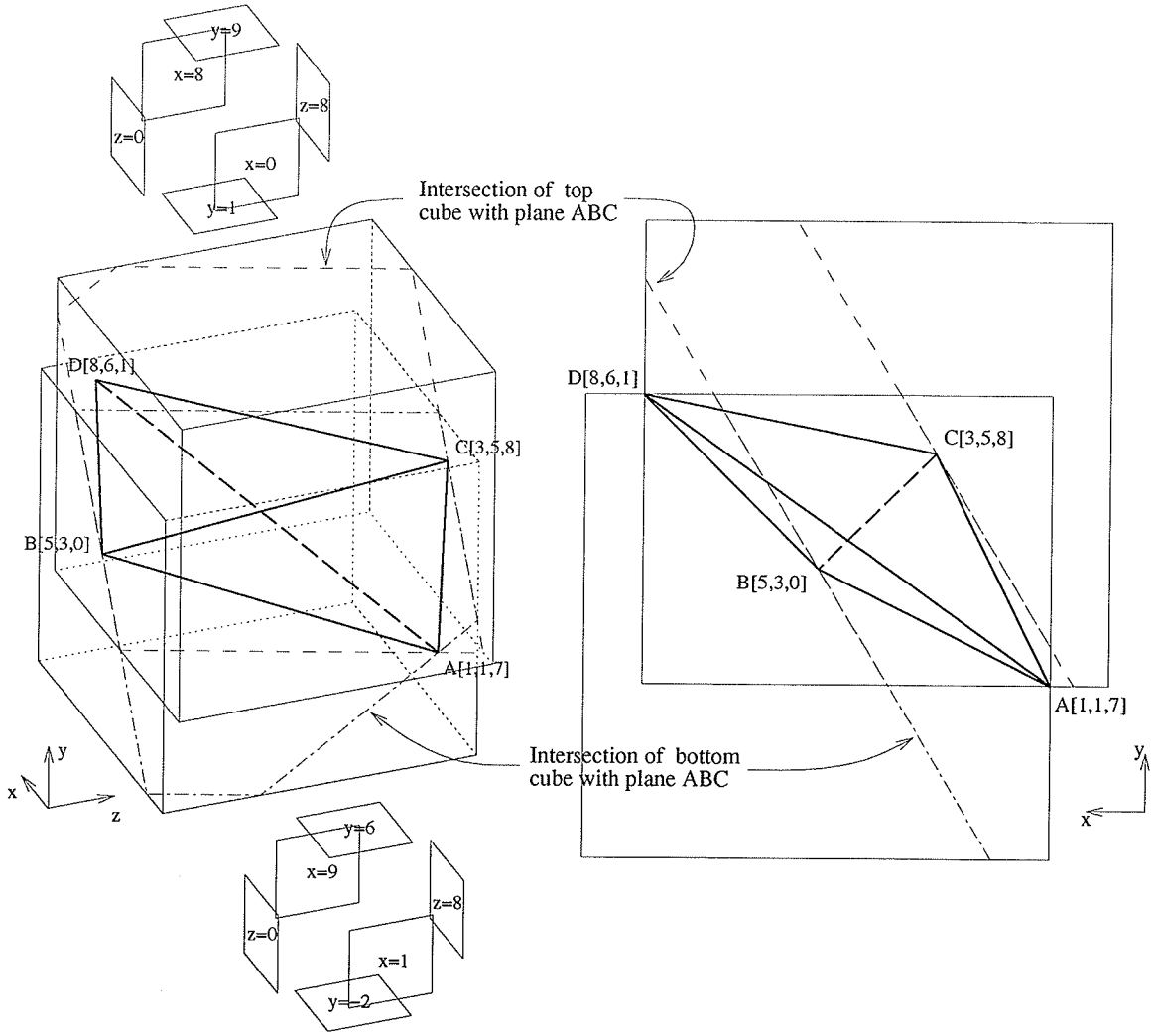


Figure 4.10: ABCD has two circumballs in the l_∞ metric

The total number of choices for all sites is $2(d-1)$. Now let us count the number of choices for the circumball. If a site has no choices, then there is no possible circumball through the sites. If a site has just one choice, then it is forced to lie on that face, the max-i face, say. Then the i-minimum cannot lie on the min-i face, so we have reduced by one the choices for the i-minimum site. Once we have dealt with all the forced sites, the remaining k sites will have at least two choices each. Each forced site reduces the total number of choices by 2, so the total number of choices left is $2k$, that is, each remaining site has exactly two choices. Note that $k = 1$ is impossible, since that site would have to be both a i-minimum and a i-maximum. (This would be degenerate.) Each time we make a choice of a face for one of these, one of the other remaining sites is forced. This means that we can make a choice between two faces at most $\lfloor k/2 \rfloor$ times. k is at most $d-1$, so the number of circumballs is at most $2^{\lfloor \frac{d-1}{2} \rfloor}$. □

We can construct configurations that attain this bound. For example, in $\mathbf{R}^5$, the site set

$$\{(0,1,1,1,1),(4,1,1,1,1),(1,0,0,1,1),(1,2,2,1,1),(1,1,1,0,0),(1,1,1,2,2)\}$$

has 4 circumballs, as does any configuration with l_∞ distance less than $1/2$ from this one. The generalization to $\mathbf{R}^d$ is obvious.

Configurations with large numbers of circumballs seem to be quite rare. I chose 6 points from the uniform distribution on the unit hypercube in $\mathbf{R}^5$, and found that the configuration had no circumballs in 75% of the trials, one circumball in 20%, two circumballs in 5%, and four circumballs in just 0.03% of the trials.[10]

Another interesting fact can be discovered by examining figure 4.10. The left-hand picture shows an orthogonal projection onto the plane through ABC. The thin dashed line shows the intersection between the top cube and the plane ABC while the thin dot-dashed line shows the intersection with the bottom cube. Looking at the part of the plane ABC "below" the line BC we see that the nesting property does not hold, even if we just consider the plane ABC. Consequently, it is possible for ABC to be a common facet of two Delaunay tetrahedra on the same side of ABC and of three Delaunay tetrahedra altogether.

[10] Just 17 times in 50,000 trials.

4.2.2 An Algorithm for Higher-Dimensional Convex-Distance-Function Delaunay Triangulation

Bearing this possibility in mind, we can now sketch a selection algorithm for computing the cdf Delaunay triangulation of n sites that works for pseudo-degenerate configurations. We assume that the sites are not truly degenerate (that is, each set of $d+1$ sites has a finite number of circumballs). The symbolic perturbation technique of Edelsbrunner and Mücke can be used to simulate this [97] if necessary. The algorithm is similar to the gift-wrapping algorithm for computing the convex hull [44]. The geometric primitives required are one to find the set of circumballs of $d+1$ sites, and one that tests if a site is inside a circumball. We proceed from simplex to simplex, at each step constructing a simplex that shares a facet, (a d dimensional simplex) with a previously constructed simplex. To construct a simplex on a facet, we select any other site as a candidate and find the circumballs of the d sites in the facet and the candidate site. We then test each other site against each of the circumballs, eliminating circumballs found to be non-empty. If the last circumball of a candidate is eliminated, then the site responsible becomes the new candidate.

Now, if the nesting property holds, it is only necessary to keep a single candidate and make one pass to find the new simplex. This is because if we find a site inside the circumball of a candidate we are guaranteed that the circumball of the new site does not contain any of the sites that we have already checked, and if a site is outside the circumball of the candidate then its circumball will contain the candidate.

Since the nesting property does not hold it is necessary to maintain a set of candidate sites. Each new site is tested to see if it is inside each candidate circumball, and circumballs found to be non-empty are eliminated. If the last circumball of a candidate is eliminated, it is removed from the candidate set. It also necessary to test the circumball(s) of the new site against each candidate. If at least one circumball survives, then the new site is added to the candidate set. Finally, it is necessary to make two passes over the sites so that each candidate circumball is tested against all sites.

Clearly, any simplex found by this algorithm will have an empty circumball. To ensure that we find all of the Delaunay simplices we can use the techniques of section 4.1 to round the corners of the convex distance function and guarantee that the dual graph of the Delaunay triangulation is connected. This also means that any facet of the convex hull will also be a facet of a Delaunay simplex; so the normal gift-wrapping algorithm can be used to find an initial facet.

The algorithm performs a traversal of the dual graph of the triangulation; so a set ADT is needed to keep track of the facets already encountered and a stack (for depth first) or queue (for breadth first) to help us find the next facet to search from.

Analysis In the unlikely event that the set of sites is not pseudo-degenerate then each facet is incident on only two simplices and there are at most $O(n^{\lfloor d/2 \rfloor})$ simplices (since in this case the Delaunay triangulation is a projection of a $d+1$-dimensional polytope [94]) and $O(n^{\lfloor d/2 \rfloor})$ facets. Each facet requires a test to see if it is in the set of visited facets ($O(\log n)$ if a balanced tree is used to implement the set ADT) and $O(n)$ circumball tests to construct a new simplex on that facet. The total execution time is $O(n^{\lfloor (d+2)/2 \rfloor})$, the same as that for the Schaudt-Drysdale algorithm.

More likely, the site set is pseudo-degenerate. Then $O(csn)$ circumball tests will be required to find the potentially s simplices adjacent to a facet, where c is the maximum possible number of circumballs through a set of $d+1$ sites, and s is the maximum size of the candidate set.

c is a constant that depends only on the distance function. (For example, in $\mathbf{R}^d$ using the l_∞ metric we proved above that $c = 2^{\lfloor \frac{d-1}{2} \rfloor}$.) Furthermore, $c \leq s$ since having c circumballs passing through $d+1$ sites forms a candidate set of size c.

Theorem 15 *If the ball for the convex distance function is polyhedral then s is at most $(f-1)d+1$, where f is the number of faces of the polyhedron.*

PROOF. To see why this is true consider the pencil of balls passing through the d sites that form a facet.

These balls are all homothets to each other, so apply a homothety to each one and also to the hyperplane through the d sites so that they are all transformed to the unit ball. The hyperplanes are transformed to a pencil of parallel hyperplanes that intersect the ball. The sites are transformed to points on the intersection of the surface of the ball and the appropriate hyperplane. Imagine sweeping a hyperplane across the ball, starting at vertex F and ending at vertex L. (These are the support points for the hyperplane.) As we sweep across the ball, each site moves across the surface of the ball from F to L, changing faces at most $f-1$ times, so there are at most $(f-1)d$ occasions when the faces that the sites lie on change.

We can therefore partition the original pencil of balls into at most $(f-1)d+1$ subpencils, according to which sites lie on which faces. Within each subpencil all the balls are mutually

homothetic (with the same centre of homothety), namely the intersection of the d support planes of the faces that hold sites. Consequently, the nesting property holds within each subpencil and there can be only one candidate circumball within each subpencil. Since there are $(f-1)d+1$ subpencils, this is the maximum size of the candidate set. □

So, if the ball has f faces the total execution time of the algorithm is $O(nkfdc)$, where k is the size of the output.

4.3 Conclusion

It is simpler to compute empty-shape triangulations than to directly compute convex-distance-function Delaunay triangulations. In this chapter I have given a complete implementation of a sweepline algorithm for empty-shape triangulations, showing that the only things that had to be changed from a Euclidean sweepline algorithm were the geometric primitives. Other Delaunay triangulation algorithms (flip, incremental, selection, Divide-and-Conquer) can also be easily modified to produce empty-shape triangulations. (I so modified the selection algorithm to produce figure 4.7.)

Empty-shape triangulations also provide a simple way to compute convex-distance-function Delaunay triangulations. Given the corners of the convex-distance-function ball and the geometric primitives for the convex distance function I create the geometric primitives for an empty-shape triangulation that is a superset of the convex-distance-function Delaunay triangulation. Extracting the convex-distance-function Delaunay triangulation from this superset is straightforward.

These ideas also allow the computation of convex-distance-function Delaunay triangulations when the ball is unbounded and computation of constrained convex-distance-function Delaunay triangulations.

Turning to higher dimensions, I show that algorithms for higher-dimensional Euclidean Delaunay triangulation such as the incremental algorithm do not generalize to convex-distance-function Delaunay triangulation. The only one that can be easily modified to work is the selection algorithm, and I describe how to do this.

Chapter 5

Delaunay triangulation of convex polygons

5.1 Introduction

The case where the sites to be triangulated are the vertices of a convex polygon has been previously considered by Devijver and Maybank [83], by Joe [161], by Chew [52] and by Aggarwal *et al.* [5]. This case is of special interest because of the insight it gives us into the general case. Lee and Schachter [202] show that the worst case for incremental algorithms occurs when the sites lie on a parabola (and hence form a convex polygon). Algorithms that use bucketing [11, 22] will also perform poorly in this case since they require the sites to be distributed approximately uniformly.

Also, if we delete a site from a general Delaunay triangulation it is necessary to retriangulate a star-shaped polygon. Convex-polygon Delaunay triangulation algorithms can be generalized to work in this case too.

The $\Omega(n \log n)$ lower bound for constructing the Delaunay triangulation [295] (based on sorting) does not apply in this case. We shall show that some algorithms use linear expected time and implementations of these algorithms run in linear time.

It would be nice to calculate the expected run time for a 'random convex polygon', but there is no commonly accepted definition of what this means. For example, one could define it to mean 'the convex hull of a set of points from the uniform distribution'. However, one cannot take points from the uniform distribution over the entire plane. Instead, we must take them from the uniform distribution over some bounded subset, and then the convex

hull of those points will tend to approximate the shape of that subset. This contradicts our intuitive idea of randomness.

We shall calculate the average execution time over all possible triangulations of a polygon. This requires us to assume that all possible Delaunay triangulations are equally likely, but gets around the difficulty of defining a 'random convex polygon' and doesn't require us to know anything about the location of the points.

5.2 Previous work

Devijver and Maybank [83] give an algorithm that they claim satisfies a "minimum space complexity constraint", that is, requires $O(1)$ space in addition to the input. They choose an arbitrary side of the n-gon and find and test each of the $n-2$ possible triangles that could stand on that side for the empty-circle property. When they find a triangle with an empty circumcircle, it splits the polygon into two smaller polygons which are recursively triangulated. The worst case for this algorithm is $O(n^3)$ and it actually requires more than $O(1)$ space since it must use a stack for recursion. The stack will require $O(\log n)$ if the smaller piece is triangulated first. In section 5.4.1 I describe an algorithm that requires $O(1)$ additional space.

Joe [161] describes a flip algorithm for convex-polygon Delaunay triangulation. The polygon is decomposed into two chains $u_1u_2\ldots u_{nu}$ and $v_1v_2\ldots v_{nv}$ where $u_1 = v_1$ and $u_{nu} = u_{nv}$ are the endpoints of a diameter of the polygon. An initial triangulation is constructed by adding edges of the form u_iv_j. If the last edge added was u_iv_j then the next edge added is either u_iv_{j+1} or $u_{i+1}v_j$ depending on which of these two edges is present in the Delaunay triangulation of $u_iv_ju_{i+1}v_{j+1}$. Flips (see section 2.2.1) are then used to transform the triangulation into a Delaunay one. This algorithm has worst-case complexity of $O(n^2)$. Joe suggests that it will take $O(n)$ time for most polygons.

Aggarwal, Guibas, Saxe and Shor [5] describe an $O(n)$ algorithm for computing the convex hull of a polygon in 3-space with a convex projection onto a plane. By using Guibas and Stolfi's [147] lifting map $\mu(x,y) = (x, y, x^2+y^2)$ which maps the Delaunay triangulation of a set of sites S to the lower part of the convex hull of $\mu(S)$ they are able to obtain the Delaunay triangulation of a convex polygon in linear time. Unfortunately, the algorithm is rather involved and difficult to follow and seems to involve large constants.

Chew [52] describes a much simpler randomized algorithm that runs in linear expected

time. The sites are inserted into the triangulation one at a time in a random order. The amount of time required to insert a site is proportional to its degree in the resulting triangulation which is $4 - 6/n$ on average. Hence the average (taken over all insertion orders) execution time is $O(n)$. Seidel [293] observes that Chew's algorithm and analysis was the first example of backwards analysis of randomized geometric algorithms.

Djidjev and Lingas [88] showed that Aggarwal *et al.*'s algorithm allows the construction the Voronoi diagram of the vertices of a monotone histogram (*i.e.* the sites are sorted by x-coordinate and have in this ordering monotone y-coordinates). Klein and Lingas [182] show that Chew's algorithm can be generalized to compute the convex hull of the same site sets that Aggarwal *et al.*'s algorithm can (and hence it can compute the Voronoi diagram of monotone histograms).

5.3 Preliminaries

Let P be an $(n+2)$-gon with vertices $p_0p_1 \ldots p_{n+1}$. The number of different ways of triangulating P is given by the Catalan number

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

with generating function

$$c(x) = \sum_{n=0}^{\infty} C_n x^n = \frac{1-\sqrt{1-4x}}{2x} = 1 + xc^2(x)$$

and satisfying the recurrence

$$C_{n+1} = \sum_{k=0}^{n} C_k C_{n-k}.$$

C_n is $\Theta(4^n n^{-3/2})$. See [23, 264] for details.

5.4 Analysis

The fact that the sites lie on a convex polygon has been used to simplify the algorithms in this section.

5.4.1 The Circumcircle Algorithm

This algorithm uses the fact that in the Delaunay triangulation, there are no sites in the interior of the circumcircle of any triangle [202]. So, given an edge of the Delaunay triangulation (in particular, the edge $p_n p_{n+1}$of P), we can find the Delaunay triangle on that edge, $p_k p_n p_{n+1}$ by scanning through all the sites (this part is the same as in the general case).

```
radius := infinity;  c := p[n + 1];
for i := 1 to n do
  begin if distance(c, p[i]) < radius then
    begin k := i;
    c := circumcirclecentre(p[n], p[n + 1], p[i]);
    radius := distance(c, p[i]);
    end;
  end;
```

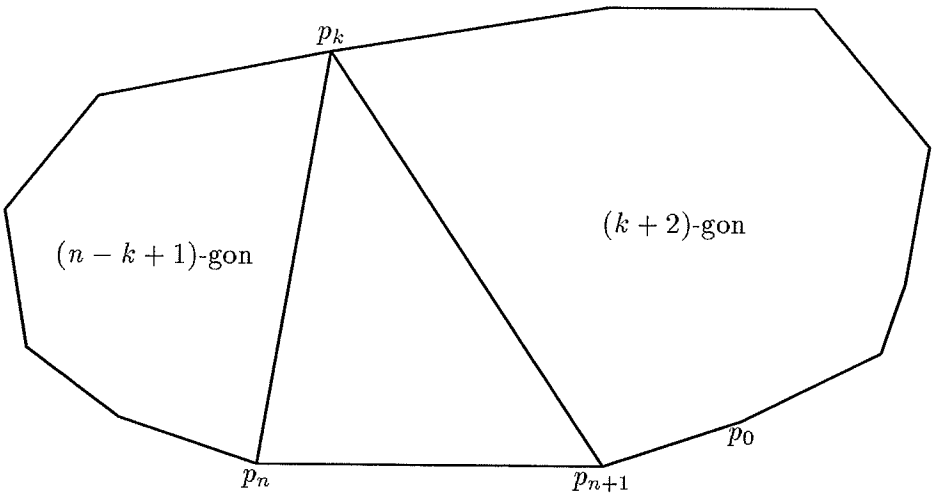


Figure 5.1: How $\triangle p_k p_n p_{n+1}$ divides P

This triangle divides P into a $(k+2)$-gon $p_{n+1}p_0p_1 \ldots p_k$ and a $(n-k+1)$-gon $p_k p_{k+1} \ldots p_n$ (see figure 5.1); so we recursively apply the algorithm to each of these smaller polygons. (This is where the fact that the sites are convex enables us to simplify things.) So, if $scan(a, b)$ returns the third point of the Delaunay triangle on side $p_a p_b$ of the polygon $p_a p_{a+1} \ldots p_b$ we can triangulate the polygon with:

```
procedure circumtri(a, b : pointindex);
var k : pointindex;
begin if b − a > 1 then
  begin k := scan(a, b);
  print_triangle(a, k, b);
  circumtri(a, k); circumtri(k, b);
  end;
end;
```

Devijver and Maybank [83] pose the problem of computing the Delaunay triangulation using only $O(1)$ additional space. This algorithm can be modified to remove the recursion and solve their problem (note that $O(n)$ stack space could be required by the algorithm above). The dual graph of the Delaunay triangulation is a tree (see figure 6.10). We merely traverse the outer face of this tree until we return to our starting point. The *scan* procedure outlined above lets us cross any edge of the Delaunay triangulation using a constant amount of space. The time requirement is increased by a constant factor of 2 since each edge must now be crossed twice. The modified algorithm is similar to the Avis-Fukuda algorithm for the enumeration of the facets of a convex hull [16].

Analysis

Let $T_c(n)$ be the number of vertices scanned by the circumcircle algorithm when triangulating an $(n+2)$-gon. Then $T_c(n+1) = n + 1 + T_c(k) + T_c(n-k)$, since we must scan $n+1$ vertices and then we have a $(k+2)$-gon and a $(n-k+2)$-gon to triangulate. The worst case for this algorithm occurs when the division is always most unequal, that is when k is always 0. Then $T_c(n+1) = n + 1 + T_c(0) + T_c(n)$, and since $T_c(0) = 0$, the solution is $T_c(n) = \frac{1}{2}n(n+1)$, that is $T_c(n)$ is $\Theta(n^2)$.

The best case occurs when k is always $\lfloor (n-1)/2 \rfloor$. Then $T_c(2n+1) = 2n + 1 + 2T_c(n)$ and if $n = 2^j - 1$ the solution is $T_c(n) = (j-1)n + j$, so $T_c(n)$ is $\Theta(n \log n)$. Now, the expected time complexity is given by

$$T_c(n+1) = n + 1 + \sum_{k=0}^{n} P_{n+1}^{k}(T_c(k) + T_c(n-k)),$$

where $P_{n+1}^{k}$is the probability that in an $(n+3)$-gon $p_k p_{n+1} p_{n+2}$ is a triangle of the Delaunay

triangulation. P_{n+1}^k is just the number of triangulations which include $p_k p_{n+1} p_{n+2}$ divided by the the total number of triangulations. This is $C_k C_{n-k}/C_{n+1}$. So

$$\begin{aligned} T_c(n+1) &= n+1+\sum_{k=0}^{n}\frac{C_k C_{n-k}}{C_{n+1}}(T_c(k)+T_c(n-k)), \\ T_c(n+1)C_{n+1} &= (n+1)C_{n+1}+\sum_{k=0}^{n} C_k C_{n-k}T_c(k)+\sum_{k=0}^{n} C_k C_{n-k}T_c(n-k) \\ &= (n+1)C_{n+1}+2\sum_{k=0}^{n} C_k C_{n-k}T_c(k). \end{aligned}$$

Let

$$g(x)=\sum_{n=0}^{\infty} T_c(n)C_n x^n.$$

Now

$$x^n T_c(n)C_n = x^n n C_n + 2x^n \sum_{k=0}^{n-1}(T_c(k)C_k)C_{n-k-1}.$$

So summing from 1 to ∞

$$\begin{aligned} \sum_{n=1}^{\infty} x^n T_c(n)C_n &= x\sum_{n=1}^{\infty} nC_n x^{n-1}+2x\sum_{n=1}^{\infty} x^{n-1}\sum_{k=0}^{n-1}(T_c(k)C_k)C_{n-k-1}, \\ g(x)-T_c(0)C_0 &= xc'(x)+2xc(x)g(x). \end{aligned}$$

Since $T_c(0)=0$,

$$g(x)=\frac{xc'(x)}{1-2xc(x)}.$$

Now, $c(x)=1+xc^2(x)$; so, differentiating, $c'(x)=c^2(x)+2xc(x)c'(x)$ and

$$c'(x)=\frac{c^2(x)}{1-2xc(x)}.$$

Therefore

$$\begin{aligned} g(x) &= \frac{xc^2(x)}{(1-2xc(x))^2} \\ &= \frac{c(x)-1}{1-4x} \\ &= \frac{1-\sqrt{1-4x}-2x}{2x(1-4x)} \end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{1-4x} + \frac{1}{2x}\left(1 - \frac{1}{\sqrt{1-4x}}\right) \\
&= \sum_{n=0}^{\infty} 4^n x^n + \frac{1}{2x}\left(1 - \sum_{n=0}^{\infty}\binom{2n}{n}x^n\right) \\
&= \sum_{n=0}^{\infty} 4^n x^n + \frac{1}{2x}\left(1 - 1 - x\sum_{n=1}^{\infty}\binom{2n}{n}x^{n-1}\right) \\
&= \sum_{n=0}^{\infty} 4^n x^n - \frac{1}{2}\left(\sum_{n=0}^{\infty}\binom{2n+2}{n+1}x^n\right).
\end{aligned}
$$

So, taking the coefficient of x^n,

$$
\begin{aligned}
T_c(n)C_n &= 4^n - \tfrac{1}{2}\tbinom{2n+2}{n+1}, \\
T_c(n) &= \frac{4^n}{C_n} - \frac{\binom{2n+2}{n+1}}{2\frac{1}{n+1}\binom{2n}{n}} \\
&= \frac{4^n}{C_n} - (2n+1) \\
&= \frac{4^n}{\Theta(4^n n^{-3/2})} - (2n+1) \\
&= \Theta(n^{3/2}).
\end{aligned}
$$

We see that the performance of the algorithm on average is much better than the worst case suggests.

5.4.2 The Divide-and-Conquer Algorithm

Divide the polygon into the two smaller polygons, $p_{\lfloor n/2\rfloor+1}\cdots p_n p_{n+1}$ and $p_0 p_1 \ldots p_{\lfloor n/2\rfloor}$ and recursively triangulate each piece. The triangulations are then merged. (This could involve deleting some edges.) See section 2.6.1 for details.

Analysis

Let $T_d(n)$ be the time taken by the Divide-and-Conquer algorithm to triangulate an $(n+2)$-gon. The divide step takes constant time since the vertices of the polygon are in order. The merge step must add to the triangulation all edges going from one piece to the other, that is those crossing the dotted vertical line in figure 5.2, say a edges. The final triangulation *contains* $n-1$ edges, the two pieces *contain* $\lfloor n/2\rfloor - 1$ and $\lceil n/2\rceil - 1$ edges so we must

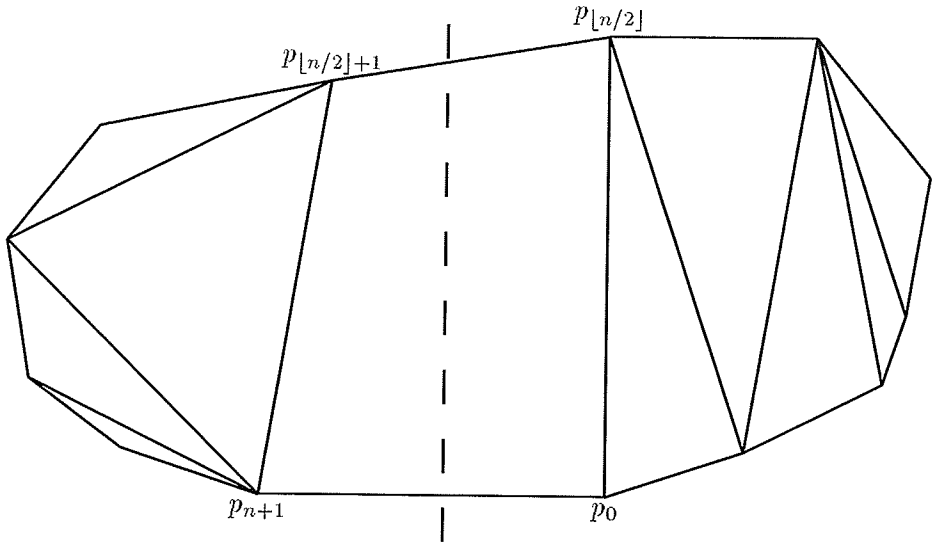


Figure 5.2: Merging the triangulations of two sub-polygons

delete $a + \lfloor n/2 \rfloor - 1 + \lceil n/2 \rceil - 1 - (n-1) = a - 1$ edges. So the total number of additions and deletions is $\Theta(a)$. Lee and Schachter [202] show how to structure the merge step so that the total work done is $\Theta(a)$. Hence

$$T_{\mathrm{d}}(n) = \Theta(a) + 2T_{\mathrm{d}}(n/2) \qquad (n \text{ even})$$

(assuming that all possible Delaunay triangulations of each piece are equally likely). Now, in the worst case all edges of the triangulation cross the vertical line and $a = n - 1$, and so $T_{\mathrm{d}}(n) = \Theta(n \log n)$. In the best case $a = 1$ and $T_{\mathrm{d}}(n) = \Theta(n)$. Let A_n^k be the average number of edges crossing the line from the centre of $p_{n+1}p_0$ to the centre of p_kp_{k+1} (the dotted line in figure 5.3). Now we shall count the number of edges that cross the line in all possible triangulations. The edge p_ip_j crosses the line when $0 \le i \le k$ and $k+1 \le j \le n+1$ (excepting $i = 0, j = n+1$ and $i = k, j = k+1$). This edge divides P into a $(j-i+1)$-gon and a $(n-j+i+3)$-gon (see figure 5.3) which can be triangulated in C_{j-i-1} and $C_{n-(j-i-1)}$ ways respectively. Hence p_ip_j occurs in $C_{j-i-1}C_{n-(j-i-1)}$ triangulations. Therefore the total number

$$C_nA_n^k = \sum_{i=0}^{k} \sum_{j=k+1}^{n+1} C_{j-i-1}C_{n-(j-i-1)} - 2C_0C_n$$

$$
\begin{aligned}
&= \sum_{i=0}^{k} \sum_{m=k-i}^{n-i} C_m C_{n-m} - 2C_0 C_n \\
&= \sum_{m=1}^{k-1} (m+1) C_m C_{n-m} + (k+1) \sum_{m=k}^{n-k} C_m C_{n-m} + \sum_{m=n-k+1}^{n-1} (n-m+1) C_m C_{n-m} \\
&= 2 \sum_{m=1}^{k-1} (m+1) C_m C_{n-m} + (k+1) \sum_{m=k}^{n-k} C_m C_{n-m}.
\end{aligned}
$$

We are interested in

$$
\begin{aligned}
A_{2n}^{n} &= \frac{1}{C_{2n}} \left(2 \sum_{m=1}^{n-1} (m+1) C_m C_{2n-m} + (n+1) C_n^2 \right) \\
&= O\left(\sum_{m=1}^{n} (m+1) C_m \frac{C_{2n-m}}{C_{2n}} \right) \\
&= O\left(\sum_{m=1}^{n} \frac{(m+1) C_m}{4^m} \right) \\
&= O\left(\sum_{m=1}^{n} \frac{1}{\sqrt{m}} \right) \\
&= O(\sqrt{n}).
\end{aligned}
$$

Hence $T_{\mathrm{d}}(n) = O(\sqrt{n}) + 2T_{\mathrm{d}}(n/2)$ and the solution is $T_{\mathrm{d}}(n) = O(n)$.

5.4.3 The Incremental Algorithm

Let D_i be the Delaunay triangulation of $p_0 \ldots p_i$. We construct D_2, D_3, ..., D_{n+1} in turn by merging the triangulation of p_i with that of D_{i-1}. We could use the method used in the Divide-and-Conquer algorithm, but the use of 'flips' [192] simplifies the procedure. We make use of the following facts: The edges added in constructing D_i from D_{i-1} are just those incident to p_i. The edges deleted are those that intersect the edges added. We connect p_i to p_0 and p_{i-1} (see figure 5.4). If p_i is outside the circumcircle of the triangle $p_0 p_k p_{i-1}$ then $p_0 p_k p_{i-1}$ is a triangle of D_i. There can be no more edges from p_i in D_i since they would cross $p_i p_{i-1}$. We can stop since we have constructed D_i. If p_i is inside the circumcircle of the triangle $p_0 p_k p_{i-1}$, then $p_0 p_{i-1} \notin D_i$ so we perform a 'flip', that is, we delete $p_0 p_{i-1}$ and insert $p_i p_k$. Now, if $p_i p_k \notin D_i$ then some edge $p_a p_b$ must intersect it. This is impossible, since no edge $p_i p_c \in D_i \setminus D_{i-1}$ can do this and we have already eliminated $p_0 p_{i-1}$, the

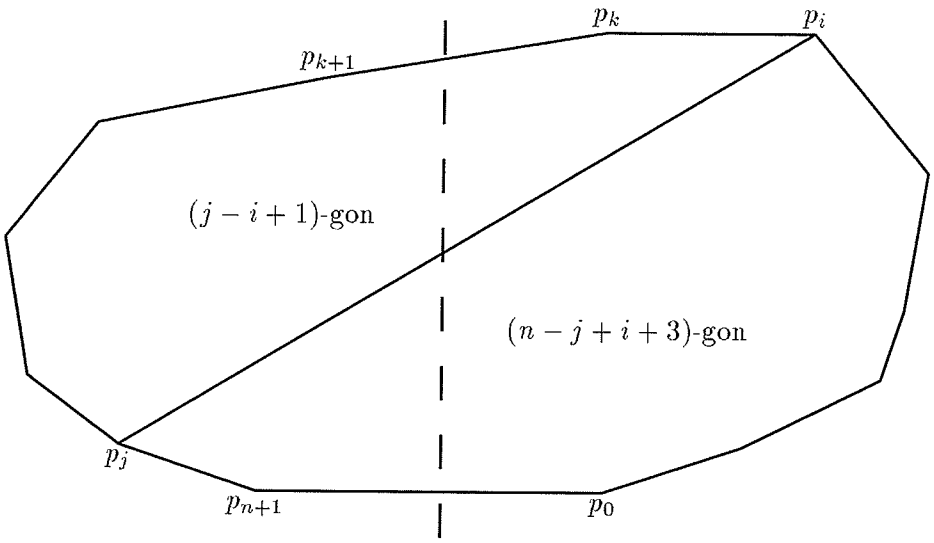


Figure 5.3: Division of P by $p_i p_j$

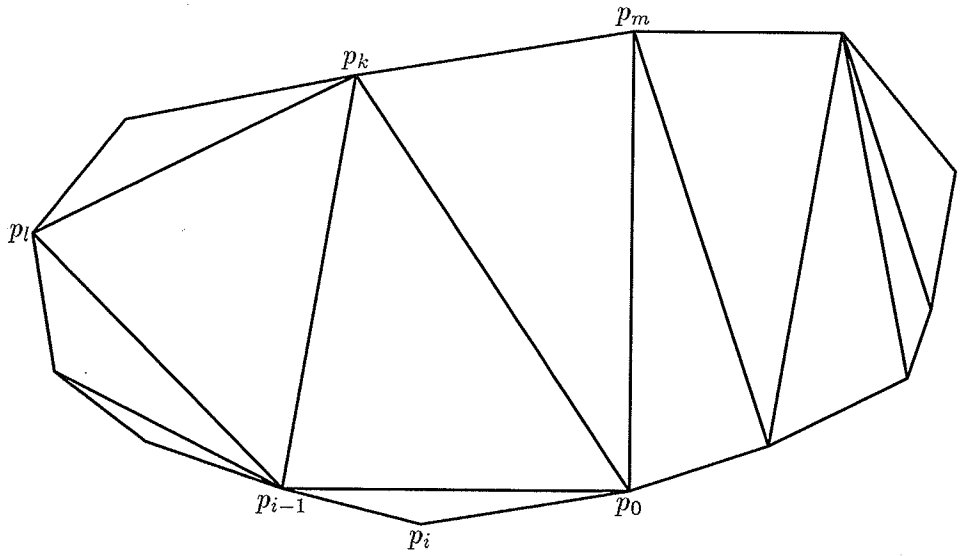


Figure 5.4: Adding p_i to D_{i-1}

only possibility in D_{i-1}. Hence $p_ip_k \in D_i$. We now continue by considering the triangles $p_{i-1}p_kp_l$ and $p_0p_mp_k$ in turn. If p_i is outside the circumcircle we can stop since we will have found a triangle of D_i. Otherwise we perform a flip and consider two more triangles. We continue this process, stopping when we find a Delaunay triangle or reach the polygon edge. Since only Delaunay edges are added and we only stop when we find an edge that no p_i-edge could cross, this process will construct D_i.

Analysis

Let $T_{\mathrm{i}}(n)$ be the number of flips performed by the incremental algorithm in triangulating an $(n+2)$-gon. The number of flips in constructing D_i from D_{i-1} is just the number of edges incident on p_i, say d_i flips. Hence

$$T_{\mathrm{i}}(n) = T_{\mathrm{i}}(n-1) + d_i = \sum_{i=1}^{n} d_i$$

since $T_{\mathrm{i}}(1) = 0$. Now in the worst case all the edges of D_i are incident on p_i and so $d_i = i-1$, and $T_{\mathrm{i}}(n) = \frac{1}{2}n(n-1) = \Theta(n^2)$. In the average case $d_i = A_i^0 = 2(i-1)/(i+2)$ (since there are $i-1$ edges and $i+2$ sites) in an $(i+2)$-gon; so $T_{\mathrm{i}}(n) = 2n - 3H_{n+2} + 4\frac{1}{2} = \Theta(n)$. ($H_n$ is the nth harmonic number.)

5.5 Empirical Tests

To test the analysis described in the preceding section, it is necessary to be able to generate random convex n-gons. Unfortunately, there is no accepted definition of what a random convex polygon is. Chapter 6 discusses the difficulty and presents several operational definitions.

The three algorithms were implemented in Pascal and tested on convex polygons of sizes from 31 to 992 generated by the rejection method (see section 6.2) with sites taken from the uniform distribution on the unit square. Execution times (averages of 70 trials) are plotted in figure 5.5. A least-squares line of best fit has been drawn through each set of points. The results are surprising. Although the slope of the line for the circumcircle algorithm suggests $\Theta(n^{1.61})$ behaviour it can be seen that the points are curving upwards. The slope is tending towards 1 (the slope of the line through the last two points is 0.9), implying worst case

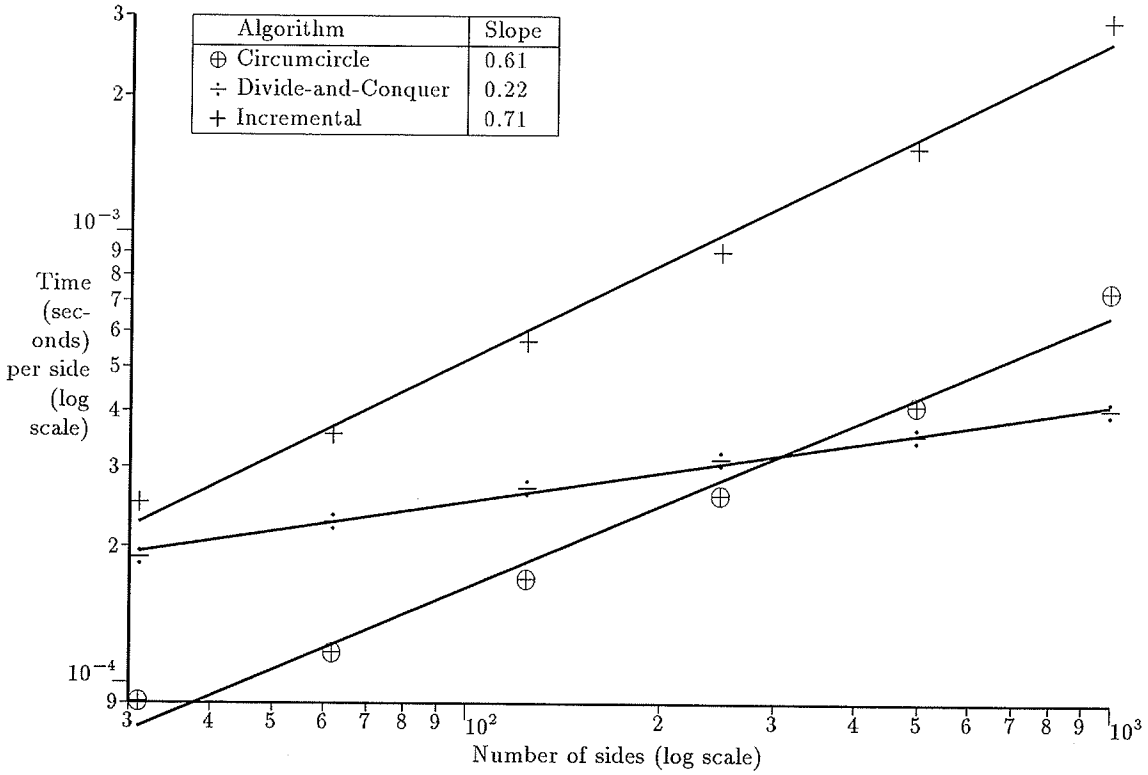


Figure 5.5: Average triangulation time

($\Theta(n^2)$) behaviour. The divide-and-conquer and incremental algorithms also show worst-case behaviour. This suggests that all possible triangulations of the polygons generated by the above method are not equiprobable. Now, we would expect the Divide-and-Conquer algorithm to exhibit worst-case behaviour if $\overline{A}^n_{2n}$ (the average value of A^n_{2n} for the generated polygons) is $O(n)$. The circumcircle algorithm will perform poorly when the p_k that it finds tends to be close to p_n or p_{n+1}. This will occur if the triangulation tends to have 'short' edges. Define L_n to be the average length of all edges over all possible triangulations (we will say p_1p_3 has length 2) and $\overline{L}_n$ to be the average in our generated polygons. The edge p_kp_{n+1} has length $\min(k+1, n-k+1)$ and occurs in $C_{n-k}C_k$ triangulations, so

$$L_n = \frac{\sum_{k=1}^{n-1} C_{n-k}C_k \min(k+1, n-k+1)}{\sum_{k=1}^{n-1} C_{n-k}C_k}.$$

Now, $\sum_{k=1}^{n-1} C_{n-k}C_k = C_{n+1} - 2C_n = \frac{2n-2}{n+2}C_n$ so,

$$\begin{aligned} L_{2n} &= \frac{2n+2}{(4n-2)C_{2n}}\left(2\sum_{k=1}^{n-1}(k+1)C_kC_{2n-k} + (n+1)C_n^2\right) \\ &= \frac{n+1}{2n-1}A^n_{2n} \end{aligned}$$

Similarly, $\overline{L}_{2n} = \frac{n+1}{2n-1}\overline{A}^n_{2n}$. This is quite remarkable.

- If $\overline{L}_n > L_n$ we would expect the circumcircle algorithm to perform better than expected and the Divide-and-Conquer algorithm to do worse.
- If $\overline{L}_n < L_n$ we would expect the opposite.

How did both algorithms manage to do so badly?

It is instructive to consider the maximum and minimum possible average edge length for a particular triangulation, $L_n^{\max}$ and $L_n^{\min}$.

Let D be a triangulation of an $(n+2)$-gon, P. The average length of the edges of D is

$$L(D) = \frac{1}{n-1}\sum_{\substack{j>i \\ ij\in D}} \min(j-i, n+2-j+i).$$

Form D', the dual of the triangulation. This will be a tree with n vertices, each having a maximum degree of three. Edges of D' correspond to edges of D which divide P into two pieces. Add directions to the edges of D' to point from the triangle in the larger piece to

the triangle in the smaller piece. The length of an edge in D is one more than the number of triangles in the smaller of the pieces it divides P into. The length of an edge in D' is one more than the number of its descendant vertices.

Now, D' must have a source, say s, and every vertex can be reached from s (since we will always be going from a larger piece to a smaller piece). Each vertex other than s has exactly one incoming edge. (If there were two, there would be two paths from s to it, contradicting D' being a tree.); so we can associate with each vertex the length of the incoming edge (figure 5.6).

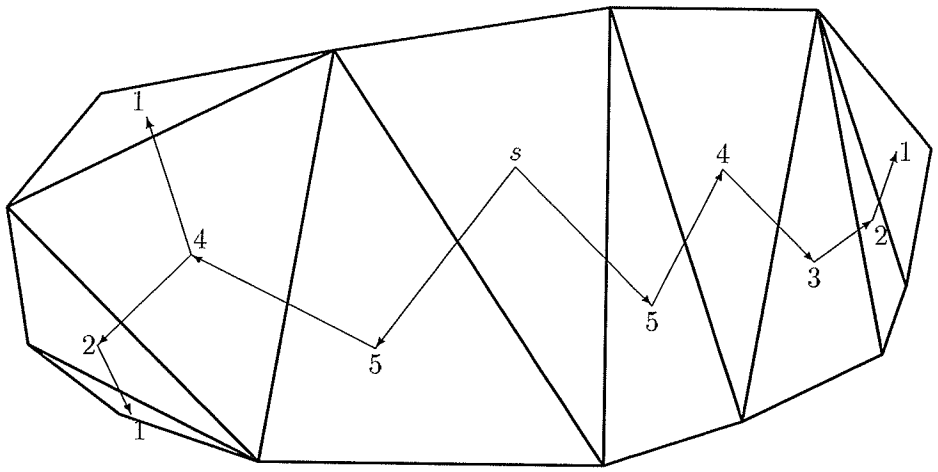


Figure 5.6: The dual of a triangulation

In the total length of all the vertices in D', each vertex will contribute one to the sum for each ancestor it has, so $L(D) = 1 + (\sum_{v \in D'} depth(v))/(n-1)$ where $depth(v)$ is the number of edges on the path from s to v (the number of ancestors of v).

This is a similar expression to that for the average internal path length of a binary tree [186] and the minimum possible value occurs when the maximum possible number of nodes are at depths $1, 2, 3, \ldots, k$. That is, we have 3 depth 1 nodes, 6 depth 2, 12 depth 3, $\ldots$, $3 \cdot 2^{k-1}$ depth k, $n + 2 - 3 \cdot 2^k$ depth $k+1$, where $k = \lfloor \log_2((n+2)/3) \rfloor$. Hence

$$\begin{aligned} L_n^{\min} &= 1 + \frac{1}{n-1}\left(\sum_{i=0}^{k} i \cdot 3 \cdot 2^{i-1} + (k+1)(n+2-3\cdot 2^k)\right) \\ &= 1 + \frac{1}{n-1}\left(3 \cdot 2^k(k-1) + 3 + (k+1)(n+2-3\cdot 2^k)\right) \end{aligned}$$

$$
\begin{aligned}
&= 1 + \frac{1}{n-1}\Big((k+1)(n+2) - 3\cdot 2^{k+1} + 3\Big) \\
&= \Theta(\log n).
\end{aligned}
$$

The maximum possible value of $L(D)$ occurs when D' is just a path. Then s is in the centre of this path; so if n is odd

$$
\begin{aligned}
L_n^{\max} &= 1 + \frac{1}{n-1}\Big(2\sum_{i=1}^{(n-1)/2} i\Big) \\
&= 1 + \frac{1}{n-1}\Big(\frac{n-1}{2}\frac{n+1}{2}\Big) \\
&= \tfrac{1}{4}(n+5).
\end{aligned}
$$

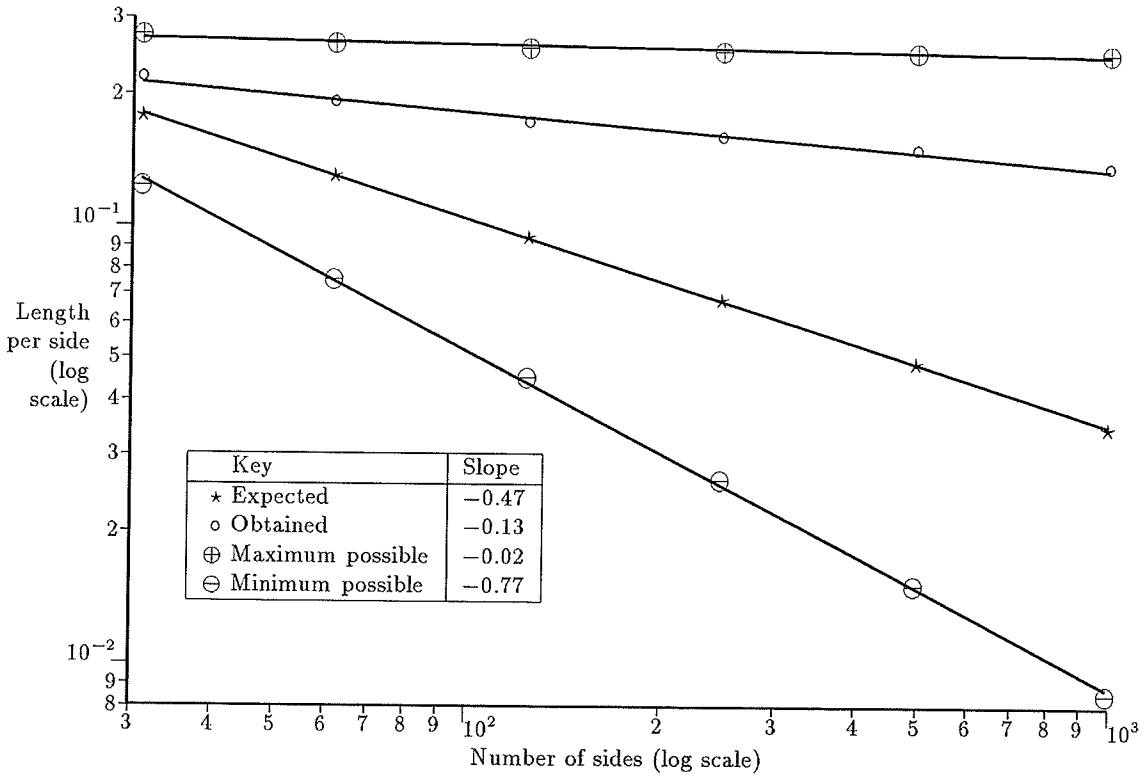


Figure 5.7: Average edge length

$L_n, \overline{L}_n$ are plotted in figure 5.7 for the same polygons that execution times were measured for. We see that $\overline{L}_n$ is $O(n)$. This explains the performance of the Divide-and-Conquer algorithm, but not that of the circumcircle algorithm.

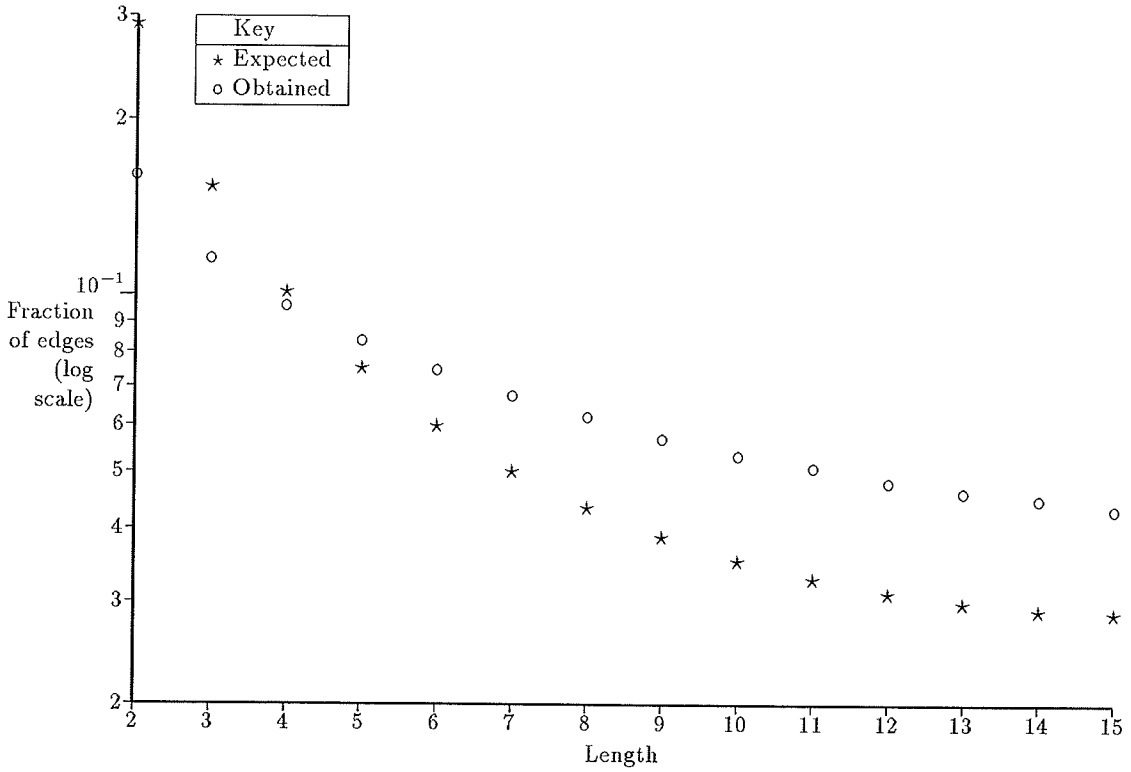


Figure 5.8: Distribution of 31-gon edge lengths

Figure 5.8 plots the distribution of edge lengths for 2000 31-gons and the expected distribution. (We would expect $2C_kC_{n-k}/\sum_{k=1}^{n-1} C_kC_{n-k} = \frac{4n+2}{n-1}P_{n+1}^k$ of the edges to be of length $k+1$ in an $(n+2)$-gon if $n \neq 2k$.) We see that there are more long edges than expected. This also suggests that the circumcircle algorithm would perform better than expected, not worse.

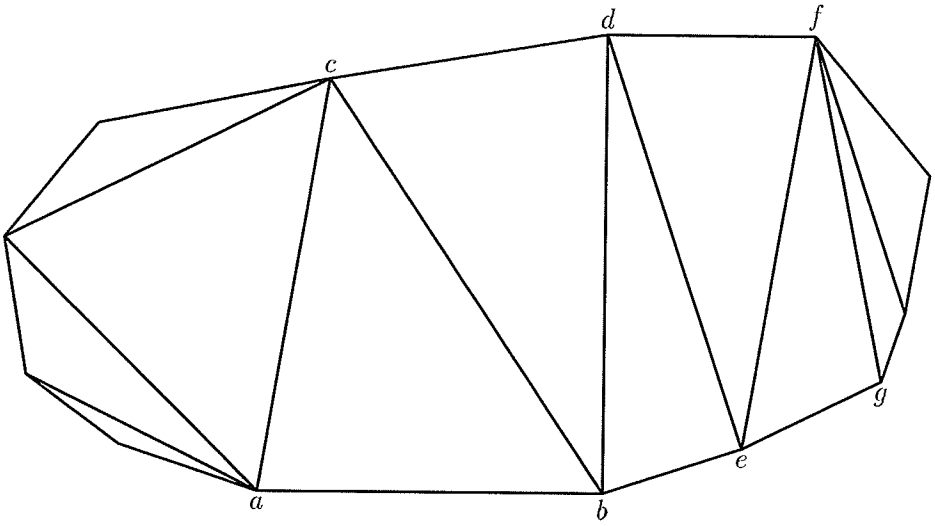


Figure 5.9: A polygon with long edges

Looking at the triangulation of a convex polygon with a large value of $\overline{L}_n$ suggests a reason for this behaviour. (Figure 5.9 shows a polygon with $\overline{L}_{11} = 3.8$, whilst $L_{11} \approx 3.42$.) Consider the behaviour of the circumcircle algorithm on this polygon. If we start with edge $ab = p_0p_{n+1}$, the algorithm will find $p_k = c$ and split the polygon using triangle abc. When we continue to triangulate the left piece, we construct triangles bcd, bde, def, efg and so on. Each of these triangles represents the worst case for splitting the polygon. We see that although the polygon has a high average edge length, the algorithm exhibits worst-case behaviour.

The problem is that the new side formed by splitting the polygon to be triangulated is special. The third point of a triangle constructed on this new side is very likely to be adjacent to one of the endpoints of the new side.

Fortunately, there is a simple way to solve this problem. Instead of constructing the triangle on the new side, we pick a side at random to build the triangle on.

Figure 5.10 plots the number of vertices scanned by the original and randomized algorithm, along with the expected number. We see that while the unmodified algorithm

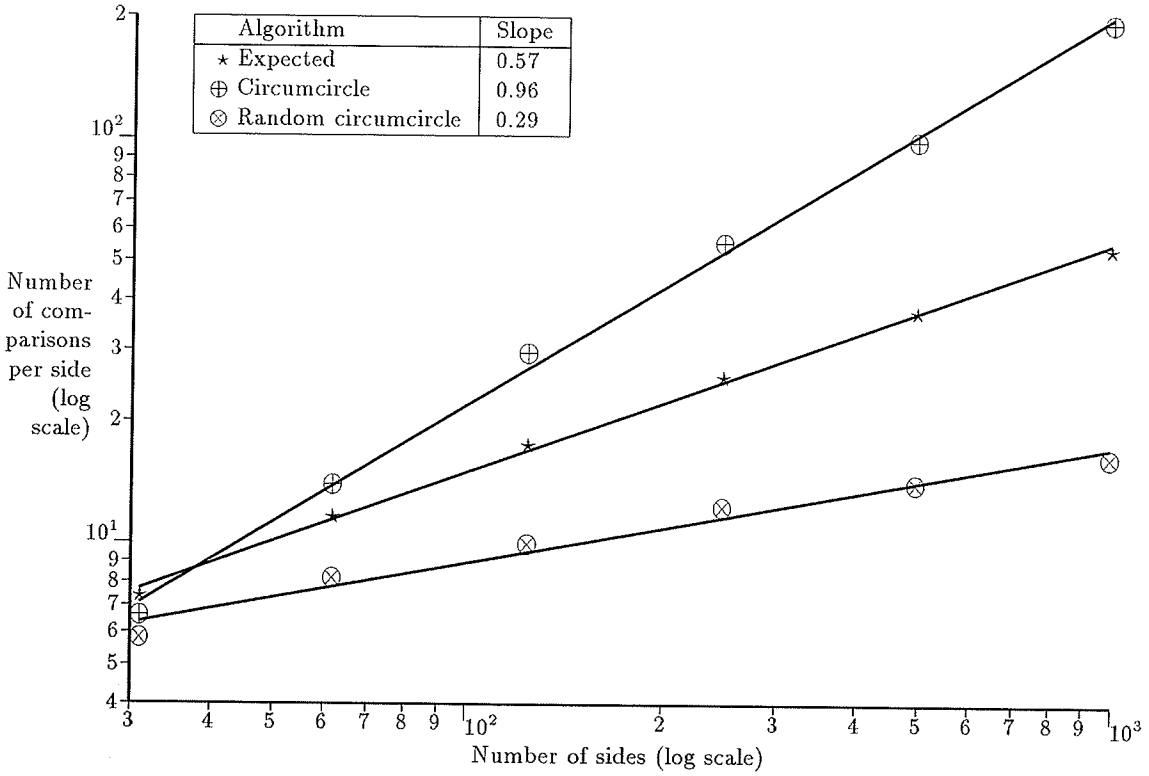


Figure 5.10: Average number of distance comparisons

is $O(n^2)$ the random algorithm performs better than expected. It appears to be at most $O(n^{5/4})$ and could even be $O(n \log n)$.

The behaviour of the incremental algorithm is also anomalous.

Let G_n^d be the number of triangulations of an $(n+2)$-gon in which p_n has degree d. By the degree of a vertex we mean the number of internal edges joined to it. Let p_k be the third vertex of the triangle on $p_n p_{n+1}$. We divide the triangulations where p_n has degree $d > 0$ into two sets: those where $k = 0$ and those where $0 < k < n-1$. (If $k = n$ then p_n has degree 0.)

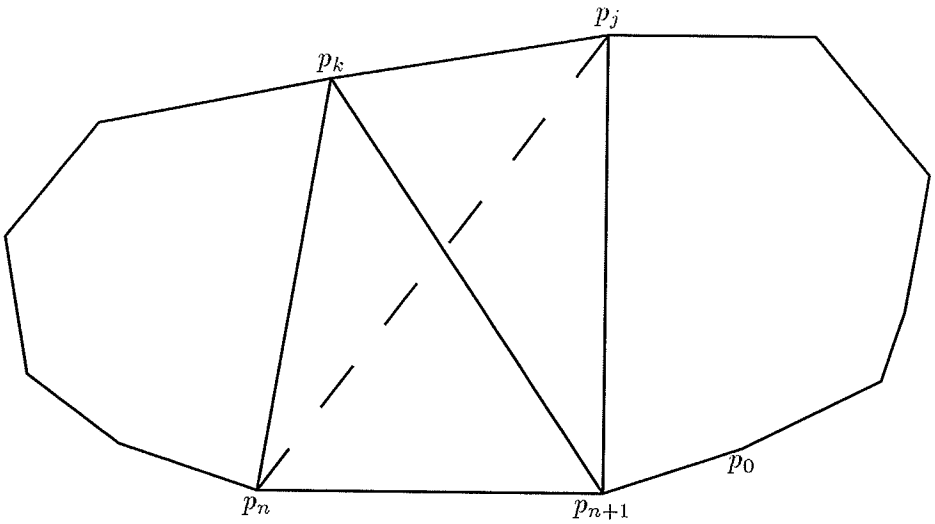


Figure 5.11: Derivation of G_n^d

If $k = 0$ the remainder of the polygon is an $(n+1)$-gon where p_0 has degree $d-1$. There are G_{n-1}^{d-1} such triangulations.

If $0 < k < n-1$ let p_j be the vertex on the other side of $p_{n+1}p_k$ (figure 5.11). If we delete $p_{n+1}p_k$ and insert $p_n p_j$ we have a triangulation where p_n has degree $d+1$. There are G_n^{d+1} such triangulations.

Hence, $G_n^d = G_n^{d+1} + G_{n-1}^{d-1}$ if $d, n > 0$. We also have $G_n^d = 0$ if $d = n$. The solution to this recurrence is

$$G_n^d = \binom{2n-3-d}{n-2} - \binom{2n-3-d}{n}.$$

Figure 5.12 shows G_{496}^d / C_{496} and the distribution of degrees for the generated 496-gons. We see that vertices of high degree are more likely than expected. Now, when the incremental algorithm adds p_k of degree d in D_k to the triangulation it must perform d flips.

Now, since p_{k-1} is close to p_k the sites that are connected to p_k in D_k will probably

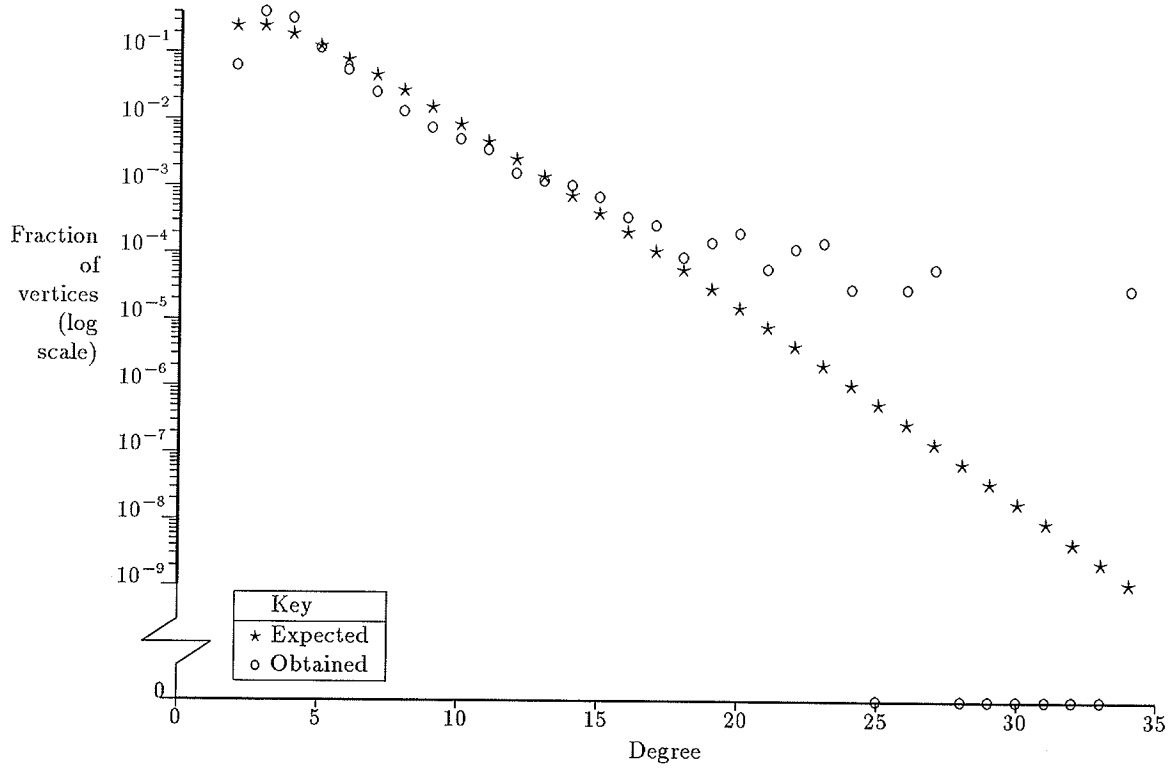


Figure 5.12: 496-gon vertex degrees

be connected to p_{k-1} in D_{k-1}; so adding p_{k-1} will probably take at least d flips. Similarly adding p_{k-2}, p_{k-3}, ... will also tend to require d flips.

Clearly, if d is large, the average number of flips per vertex added will be much larger than $2(k-1)/(k+2)$ as predicted by our theory. (In fact, for the generated polygons it seems to be about $k/7$.) This is why the incremental algorithm performed so poorly.

Fortunately, the same method that worked for the circumcircle algorithm also works here. Instead of adding sites in the order $p_1, p_2, p_3, \ldots$, we add them in a random order.

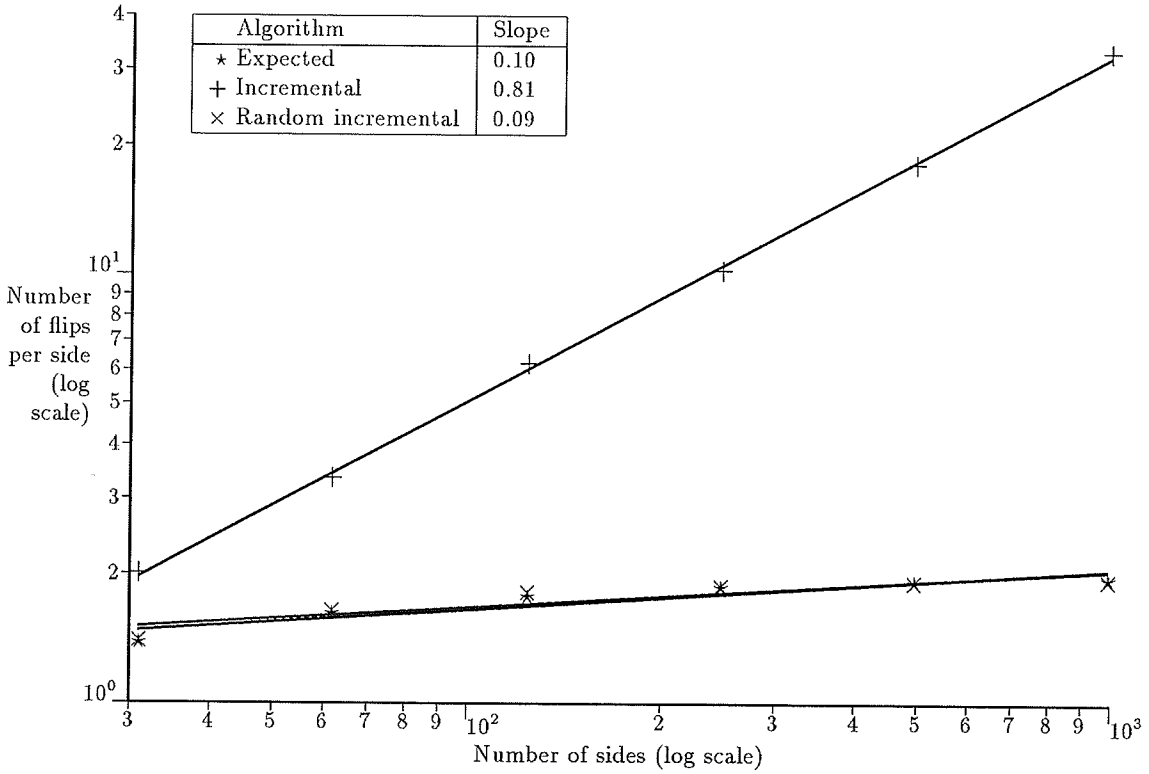


Figure 5.13: Average number of flips

Figure 5.13 shows the number of flips for the two versions of the incremental algorithm, and the expected number. We see that the original algorithm is $O(n^2)$ while the random algorithm is $O(n)$.

Figure 5.14 shows execution times for the original and randomized algorithms.

These experiments were repeated using the iteration method (section 6.3) and the vector method (section 6.4) to generate convex polygons. The results were similar to those obtained using the rejection method.

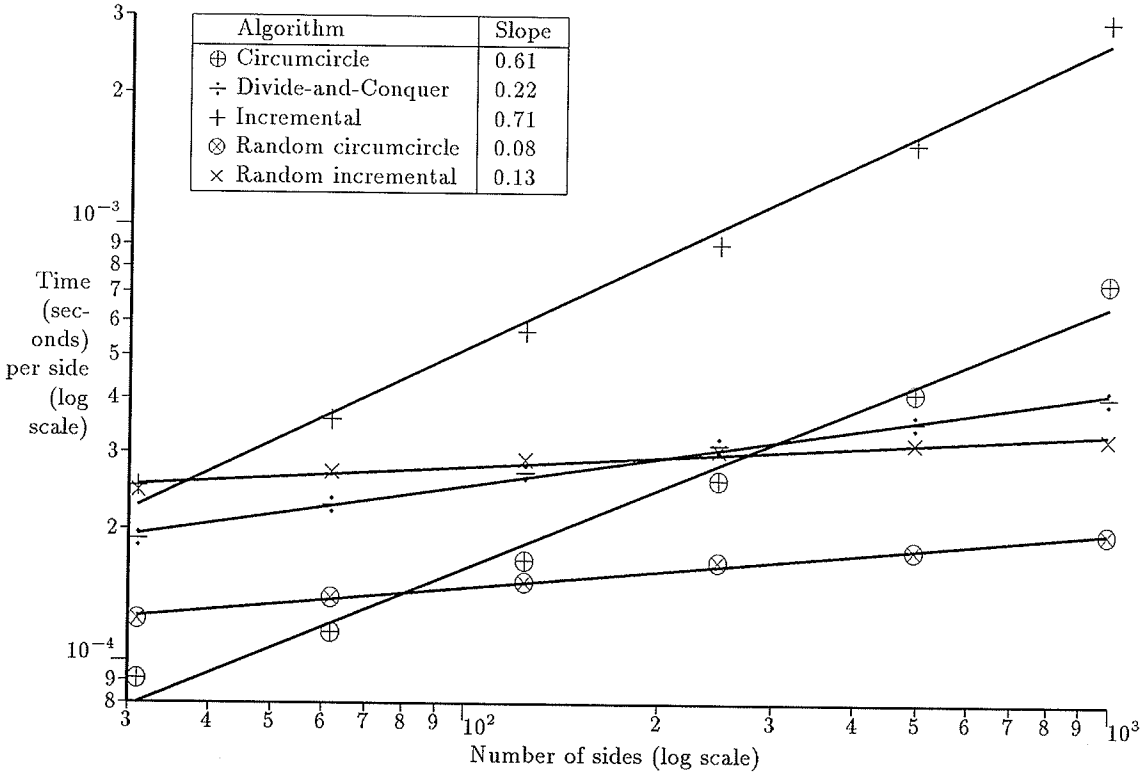


Figure 5.14: Average triangulation time

5.6 Conclusion

All the algorithms exhibited worst-case behaviour on the generated polygons rather than the behaviour predicted by our analysis. There were two causes for this worst-case behaviour.

Firstly, all triangulations of our generated polygons were not equally likely—there was a bias towards long edges. Secondly, the overall triangulation and the subtriangulations considered by the various algorithms were not 'independent'. The circumcircle and incremental algorithms were vastly speeded up by randomizing to ensure this independence.

The worst-case behaviour has some interesting implications for general Delaunay triangulation algorithms. A major step in the incremental algorithm is finding the first edge from a new site (either by finding the closest site to the new site [145], or finding the triangle the site is in [202]). In the convex polygon case, this search is unnecessary. Finding this edge could take $O(n)$ time; so one approach [11] has been to sort the sites in such a way that successive sites are close together and to start the search at the previous point. However, this chapter has shown that if the sites form a convex polygon this approach leads to $O(n)$ update time. The worst case for such algorithms may be much more probable than previously thought.

It would be nice to generalize the analysis of these algorithms to the case where the sites do not form a convex polygon. Unfortunately, while combinatorial results for the number of possible triangulations of a set of sites exist [267, 316], these allow curved edges. If we restrict the edges to being straight, then the number of possible triangulations depends on the position of the sites, and the techniques used in this chapter do not apply.

Finally, observation of figure 5.14 reveals how misleading considering just asymptotic behaviour can be. The random incremental algorithm has the best asymptotic behaviour ($O(n)$) of the five algorithms in figure 5.14, and yet is the slowest for 32-gons. Even for 1000-gons the random circumcircle algorithm is faster by a factor of almost 2.

Chapter 6

Generating Random Convex Polygons

6.1 Introduction

To test the analysis described in section 5.4, it is necessary to be able to generate random convex n-gons. Unfortunately, there is no accepted definition of what a random convex polygon is. For example, Sylvester's problem [277] is to find the probability that the convex hull of four random points is a quadrilateral. Even for points drawn from the uniform distribution, this turns out to depend on the shape of the region from which they are drawn.

Random convex polygons have been generated on the computer by Crain [67], who used Voronoi polygons defined by a Poisson point process, by Crain and Miles [68] who examined polygons defined by a Poisson line process, by Devroye [84], De Pano *et al.* [80] and Abrahamson [2] who took the convex hull of random points, and by May and Smith [226] who took the intersection of random half-spaces. However, none of these methods let you specify the number of sides of the polygon.

The only published algorithm that allows the number of sides of the polygon to be specified is that of Roussille and Dufour [275]. They present an algorithm that also allows constraints on the range of values a given polygon angle will take to be specified. The algorithm works its way around the boundary of the polygon, randomly choosing an angle for each corner and a length for each edge. The requirement that the polygon be convex and that it be possible to satisfy the constraints on the remaining angles imposes further

constraints on the angles and lengths chosen. The authors do not discuss what probability distribution the angle should be chosen from. If the uniform distribution is used then the first few angles chosen will use up most of the slack available and the remaining angles will be very close to straight angles. This does not seem like a very random convex polygon. It seems desirable that the probability distribution for each angle in the resulting polygon be the same, and it is unclear how to make this happen.

XYZ Geobench [286, 244] contains an algorithm for generating a random convex polygon with a specified number of sides. It can best be described if we use polar coordinates $p_i = (r_i, \theta_i)$, where $1 \leq i \leq n$, for the corners of the convex polygon. The angles θ_i are chosen by taking n values from the uniform distribution on $[0, 2\pi)$ and sorting them so that the corners are given in anti-clockwise order. An initial cyclic convex polygon is created by setting all the r_is to the same value. Then, a randomly chosen r_i is given a new randomly chosen value r'_i, subject to the constraint that the resulting polygon remain convex. That is, p'_i must lie to the left of $p_{i-1}p_{i+1}$ and to the right of $p_{i-2}p_{i-1}$ and $p_{i+1}p_{i+2}$ (see figure 6.1). This last step is repeated n times.

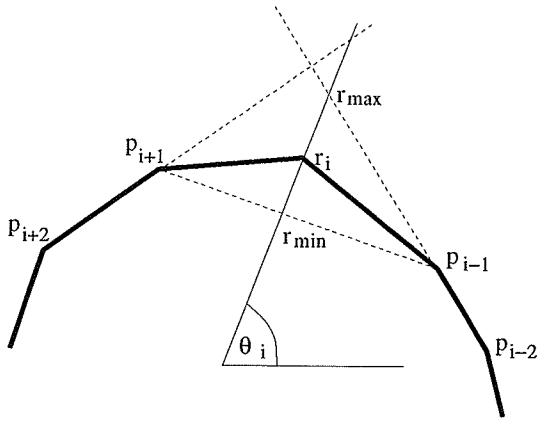


Figure 6.1: r'_i is chosen from the uniform distribution on $[r_{\min}, r_{\max}]$

There are some problems with this approach. The distribution of the number of times that a corner is moved is approximately Poisson with mean 1 if n is large, so that the fraction of vertices moved i times is approximately $i^1 e^{-1}/i!$. Approximately $e^{-1}/0! \approx 37\%$ of the corners will not be moved at all and will all lie on the same circle, which doesn't seem particularly random. Furthermore, if n is large it is not possible to move the corners

very far at each step and the final polygon closely approximates a circle.

I wish to be able to generate polygons quickly—in close to the optimal $O(n)$ time. After all, it doesn't seem to right to test an $O(n \log n)$ algorithm with data that takes time $\Omega(n^2)$ to generate.

In this chapter I consider the following methods:

Rejection Pick n points from some distribution. Reject if their convex hull is not an n-gon.

Iteration Select points from some distribution until their convex hull has n vertices.

Vector The n vectors comprising the sides of the polygon can be regarded as a point in $2n$ dimensional space. For the polygon to close, the vectors must sum to zero. This means that the point must lie on a $(2n-2)$-dimensional flat, so pick from some distribution on this flat (for example, the uniform distribution over a $(2n-2)$-dimensional unit hypersphere).

Bounce Start with an arbitrary convex n-gon and give each vertex a random velocity. If a vertex is ever about to become concave, we "bounce" it from that constraint. If we perform $O(n)$ bounces the resulting polygon should be "random".

Triangulation Choose a random topological triangulation of a polygon. Construct a convex polygon with Delaunay triangulation homeomorphic to this.

Dual We can take the dual of polygons produced by the above methods. For example, The Dual Rejection method takes the intersection of n half-spaces containing the origin and rejects the resulting polygon if it has fewer than n sides.

6.2 Rejection

Pick n points from some distribution. Reject if their convex hull is not an n-gon.

Another way of thinking about this method is to consider convex n-gons with bounded integer coordinates (*e.g.* those expressible as 32 bit integers). Randomly choosing one such n-gon is equivalent to the Rejection method with the points coming from the uniform distribution on a square.

The naive implementation of this method is obviously not feasible. The probability of the convex hull of n points from most distributions having n vertices is extremely small, even for moderate values of n.

We will therefore use an iterative rejection method to generate a convex polygon, rejecting points that would cause the convex hull to have fewer than n points. If we have a convex n-gon P we generate another point p from the distribution. If $|\text{conv}(p+P)| < n+1$ we reject p, otherwise we form $\text{conv}(p+P)$ to get a convex $(n+1)$-gon. ($\text{conv}(A)$ denotes the convex hull of the set A.) In other words, we accept the point if it lies in one of the shaded regions in figure 6.2. (This is called the 2-level of the arrangement formed by the sides of P [98].) We repeat this process until we have a convex polygon with the desired

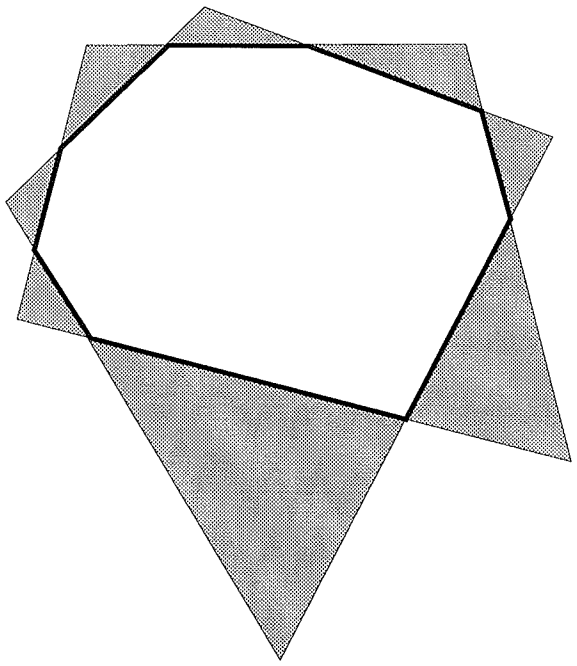


Figure 6.2: Acceptance regions for adding a point to P

number of sides.

If we represent P as a circular list of points (oriented anti-clockwise)

```
type polygon = ↑record
                   a : point;
                   next : polygon;
               end;
```

we can generate a convex n-gon with

```
procedure makepolygon(var P : polygon; n : integer);
var u, right, temp : polygon;
  p : point;
  size, noright : integer;
begin maketriangle(P); size := 3;
while size < n do
  p := randompoint;
  {count number of edges of P that p is to the right of}
  u := P; noright := 0;
  repeat if ¬left(u↑.a, u↑.next↑.a, p) then
      begin noright := noright + 1;
      right := u;
      end;
    u := u↑.next;
    until (u = P) ∨ (noright > 1);
  if noright = 1 then {add p to P}
    begin new(temp); temp↑.a := p; temp↑.next := u↑.next;
    u↑.next := p;
    end;
  end;
end;
```

where *maketriangle* makes a triangle by taking three points from the distribution and $left(x, y, z)$ returns *true* iff z is to the left of the line from x to y. The only drawback of this procedure is that it takes time $\Theta(n^4)$ to generate a convex n-gon.

To improve this to $O(n \log n)$ we need to compute the probability that a random point falls into each of the shaded regions in figure 6.2. Let a_i denote the probability that a point falls into acceptance region i, δ_i the area of acceptance region i and μ_i the minimum value of the probability density function over region i.

If the points are being chosen from the uniform distribution over some shape, then a_i is just the area of the intersection of the shaded region with the distribution shape (this intersection will probably be a triangle). If some other distribution is being used (for example, a normal distribution), then a_i is just the integral of the probability distribution over acceptance region i.

To randomly select a point from the acceptance regions we select region i with probability a_i and randomly choose a point from this region. For large values of n these regions will be small triangles. If the points are being chosen from a uniform distribution, then it is sufficient to choose a point from the uniform distribution over the triangle in time $O(1)$. Most other distributions will be almost constant over the acceptance region, that is, $\mu_i\delta_i$ will be almost as big as a_i. So with probability $(\mu_i\delta_i)/a_i$ (most of the time) we choose a point from the uniform distribution over the triangle in time $O(1)$. Otherwise we can subdivide the region and repeat the process.

To get $O(n \log n)$ total time we need to be able to select an acceptance region in time $O(\log n)$. The alias method for generating random variables [188] could randomly select an acceptance region with the required probability in time $O(1)$. Unfortunately, it takes time $O(n)$ to construct the alias table and this would be required after each point is generated (since the probabilities of all acceptance regions change). The alias method is therefore not suitable.

Instead, we use a tree structure. The leaves of the tree in order from left to right correspond to the acceptance regions in order around the polygon. In each internal node we store a probability given by the sums of all the a_is of the leaves in the associated subtree.

We can define it in Miranda [28, 315] like this:

```
> tree ::= Leaf num region | Internal num tree tree
> probability :: tree -> num
> probability (Leaf p reg) = p
> probability (Internal p l r) = p
```

To select a leaf with the required probability we begin by generating a random number uniformly between 0 and the probability of the root.

If the number is less than the probability of the left subtree, we recursively select from the left subtree with the same number, otherwise select from the right subtree using the number less the left subtree probability.

```
> select :: tree -> num -> region
> select (Leaf p reg) n = reg
> select (Internal p l r) n
>     = select l n, if n < probability l
>     = select r (n - probability l), otherwise
```

The optimal (in the sense of fastest selection of an acceptance region on average) tree is a Huffman tree [157]. Unfortunately, rebuilding the Huffman tree after a point is inserted will take $O(n)$ time; so we cannot use a Huffman tree.

Instead we will use some sort of balanced tree (height balanced or B tree or some similar scheme). Since the tree is balanced, the `select` function will select a region in time $O(\log n)$.

Now we need to show that the tree can be updated in time $O(\log n)$. Figure 6.3 shows that when a new point is inserted, its region is split into two and the two adjacent regions (only) change. So all that is required is to split the relevant leaf node, rebalance the tree ($O(\log n)$) and then recompute the probabilities of all the ancestors of the four changed regions ($O(\log n)$ since each node has $O(\log n)$ ancestors). To simplify the computation we

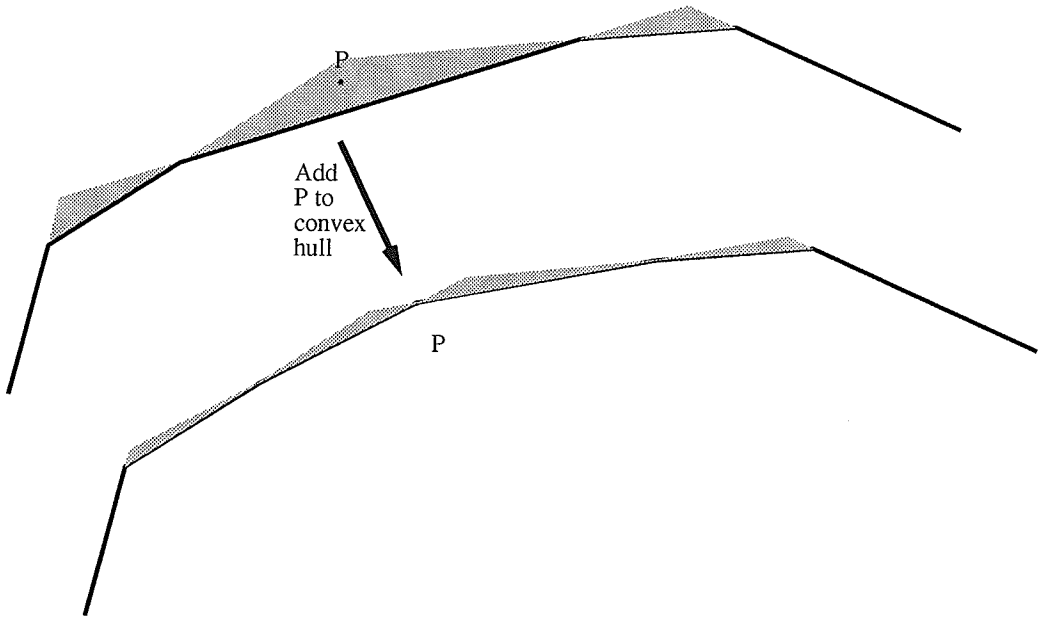


Figure 6.3: How the acceptance regions change when a new point P is added

store a parent pointer for each node (this saves us having to store the path from the root to the leaf) and link all the leaves together in a double linked list (simplifies finding the adjacent region).

Figure 6.4 shows how the tree rotation(s) necessary for rebalancing take time $O(1)$. We need to recalculate the probabilities at only two internal nodes.

Putting it all together, we can now generate a "random" convex n-gon in time $O(n \log n)$.

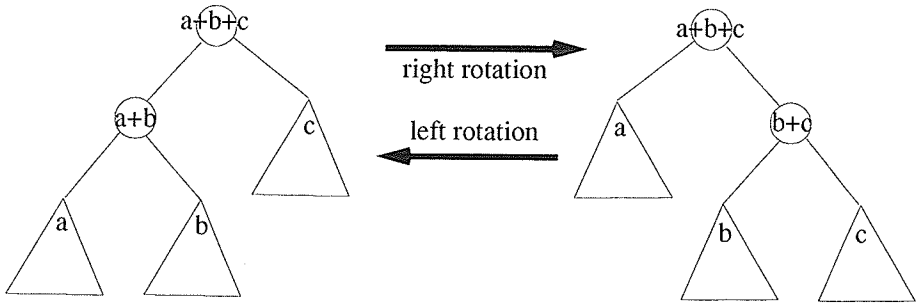


Figure 6.4: Tree rotation takes time $O(1)$

There is one remaining difficulty—our selection is biased. Fortunately, correcting for this bias is easy. At each iteration there is a probability $A = \sum_i a_i$ of getting a point in an acceptance region and a probability $1 - A$ of getting one outside. In the case that it is outside, we might as well stop, because there is no chance the the convex hull can have n vertices. Instead of doing this we always pick a point in the acceptance region and weight the result by A (this value is conveniently available at the root of the tree). The weight of the final polygon is given by the product of all the weights A during its construction. We use this weight when computing any statistics using this polygon (for example, the execution times in chapter 5).

6.3 Iteration

In the iteration method we repeatedly take points from our distribution until the convex hull has n points. If k points are taken from the unit disc, the expected number of points in the convex hull is $\Theta(k^{1/3})$ [268], so the naive implementation of this method will require generating $\Theta(n^3)$ points to produce an n-gon.

However, it is unnecessary to generate points that fall inside the convex hull of the preceding points. We can divide the area outside the convex hull into regions as shown in figure 6.5 and generate a point in one of these regions by the method described in section 6.2.

It takes time $O(\log n)$ to select a region, and then $O(1)$ to generate a point in that region and update the convex hull (we may have to delete $O(n)$ vertices, but this can be charged to the vertices when they are created), and $O(\log n)$ to update the tree used for region selection. If h points are generated while creating our n-gon, the total time is $O(h \log n)$.

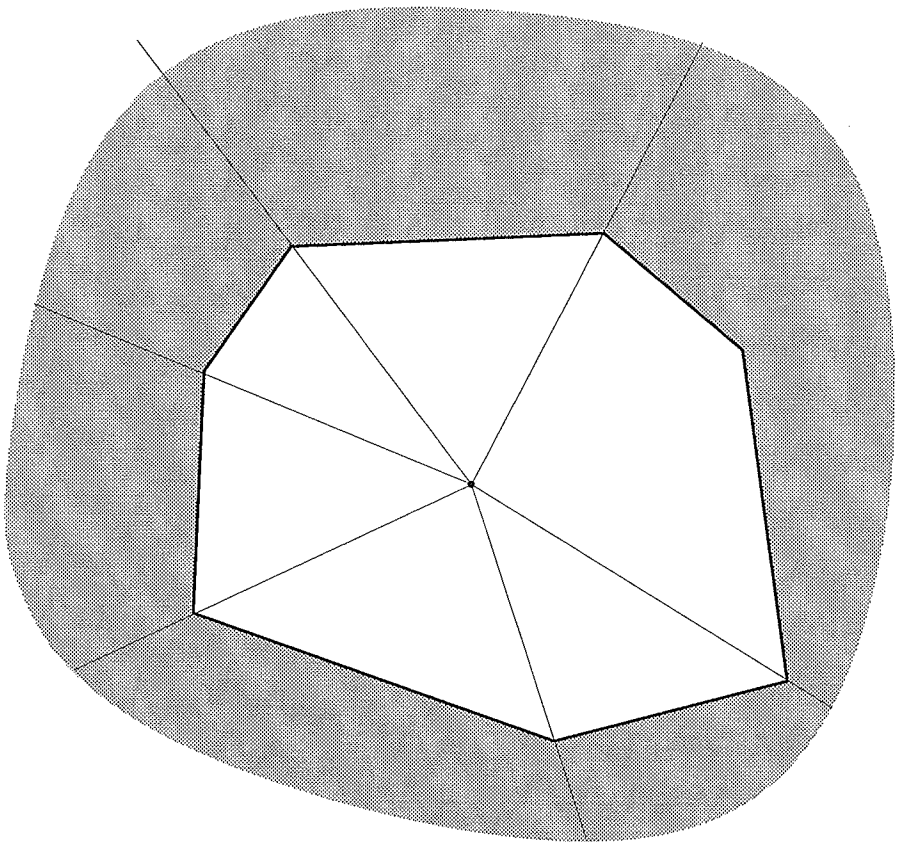


Figure 6.5: Dividing exterior of the hull into regions

When implemented using a uniform distribution on the unit disc, h turned out to be roughly $2n$ and the total time to generate an n-gon was $O(n \log n)$.

6.4 Vector

We can regard the sides of the convex polygon as vectors (figure 6.6).

Let the vectors be (x_1, y_1), (x_2, y_2), ..., (x_n, y_n). Since the polygon must close up, the sum of the vectors is 0. That is,

$$\sum_{i=1}^{n} x_i = 0.$$

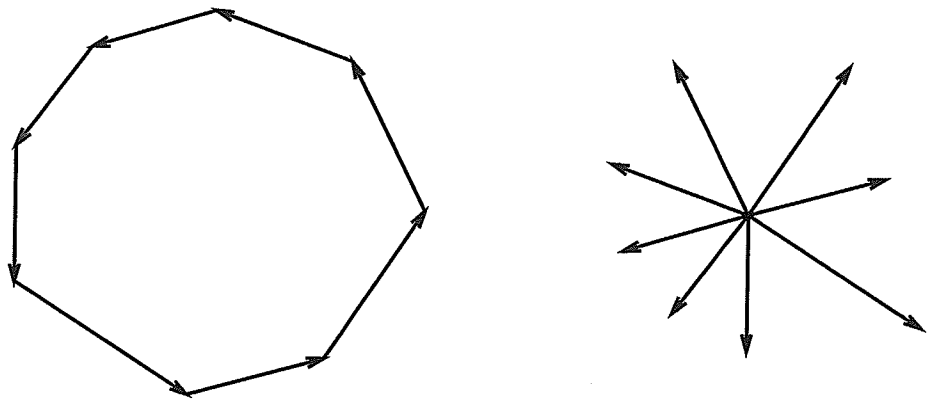


Figure 6.6: Convex polygon edges regarded as vectors

If we think of $(x_1, x_2, \ldots, x_n)$ as a point in n-dimensional space, this says that this point must lie on the hyperplane $\sum_i x_i = 0$. So all we need to do is choose a point from some distribution on this hyperplane. I have chosen the uniform distribution on the $n-1$ dimensional unit hypersphere in my implementation.

We can generate a point on the surface of a $n-1$ dimensional unit hypersphere by making a vector from $n-1$ normally distributed variates and normalizing it [238]. To get a point from the uniform distribution on the interior of this hypersphere we just scale this point by a factor $k = u^{1/n}$ where u is a uniform variate between 0 and 1.

If we extend this vector with a 0, we have a point on the hyperplane $x_n = 0$. This hyperplane can be rotated onto the $\sum_i x_i = 0$ hyperplane by constructing a orthonormal basis for $\mathbf{R}^n$ containing $(1, 1, \ldots, 1)/\sqrt{n}$, the normal to this hyperplane. This basis forms the columns of the transformation matrix for the rotation. The transformation is

$$\begin{bmatrix} x'_1 \\ x'_2 \\ x'_3 \\ \vdots \\ x'_{n-1} \\ x'_n \end{bmatrix} = \begin{bmatrix} \frac{-1}{\sqrt{2\cdot1}} & \frac{-1}{\sqrt{3\cdot2}} & \frac{-1}{\sqrt{4\cdot3}} & \cdots & \frac{-1}{\sqrt{n(n-1)}} & \frac{-1}{\sqrt{n}} \\ \frac{1}{\sqrt{2\cdot1}} & \frac{-1}{\sqrt{3\cdot2}} & \frac{-1}{\sqrt{4\cdot3}} & \cdots & \frac{-1}{\sqrt{n(n-1)}} & \frac{-1}{\sqrt{n}} \\ 0 & \frac{2}{\sqrt{3\cdot2}} & \frac{-1}{\sqrt{4\cdot3}} & \cdots & \frac{-1}{\sqrt{n(n-1)}} & \frac{-1}{\sqrt{n}} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & \frac{-1}{\sqrt{n(n-1)}} & \frac{-1}{\sqrt{n}} \\ 0 & 0 & 0 & \cdots & \frac{n-1}{\sqrt{n(n-1)}} & \frac{-1}{\sqrt{n}} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix}.$$

The following Pascal fragment performs the transformation in time $O(n)$.

```
sum := x[n]/sqrt(n);
for i := n downto 2 do begin
   sum := sum + x[i-1]/sqrt(i*(i-1));
   x[i] := i*x[i-1]/sqrt(i*(i-1)) - sum;
end; {for}
x[1] := -sum;
```

This generates the x coordinates of our vectors. The y coordinates are generated exactly the same way. Finally the n vectors are sorted by direction to create the convex polygon. The sorting is the only part that requires $O(n \log n)$ time.

6.5 Bounce

This idea is due to Thurston [26].

Start with an arbitrary convex n-gon and give each vertex a random velocity. If a vertex is ever about to become concave, we "bounce" it from that constraint. If we perform $O(n)$ bounces the resulting polygon should be "random".

We can use discrete event simulation [117] techniques to implement this method. We maintain a priority queue containing all potential bounces (events). This queue enables us to identify the next bounce to occur. We then modify velocities so that the polygon does not become concave and adjust the priority queue accordingly. We need to be able to calculate when the bounces occur and how to modify velocities to avoid concavities.

Let $A = (a_x, a_y)$, $B = (b_x, b_y)$ and $C = (c_x, c_y)$ be three successive vertices on the boundary of the convex polygon, with velocities (a_{vx}, a_{vy}), (b_{vx}, b_{vy}) and (c_{vx}, c_{vy}). The position of the point A at time t is given by $A(t) = (a_x + ta_{vx}, a_y + ta_{vy})$. For the polygon to remain convex the area of ABC must be positive, so that a "bounce" will occur whenever the area becomes zero. Let

$$\begin{aligned}
(a'_x, a'_y) &= (a_x - b_x, a_y - b_y), \\
(a'_{vx}, a'_{vy}) &= (a_{vx} - b_{vx}, a_{vy} - b_{vy}), \\
(c'_x, c'_y) &= (c_x - b_x, c_y - b_y), \\
(c'_{vx}, c'_{vy}) &= (c_{vx} - b_{vx}, c_{vy} - b_{vy}).
\end{aligned}$$

Then

$$\begin{aligned} 2\Delta(t) &= (c'_x + tc'_{vx})(a'_y + ta'_{vy}) - (c'_y + tc'_{vy})(a'_x + ta'_{vx}) \\ &= c'_x a'_y - c'_y a'_x + \\ &\quad t(c'_{vx} a'_y + c'_x a'_{vy} - c'_y a'_{vx} - c'_{vy} a'_x) + \\ &\quad t^2(c'_{vx} a'_{vy} - c'_{vy} a'_{vx}). \end{aligned}$$

So, if we start with $t = 0$ a bounce will occur at the smallest positive root of the quadratic equation $\Delta(t) = 0$. If this equation has no positive roots then no bounce is possible.

One natural way to modify the velocity of B to prevent a concavity when a bounce occurs at time t_b is to imagine that vertices are physical particles and to conserve the total energy of the system. This means that we change the direction of B but not its speed. Unfortunately, this is not always possible: for example, if B has a velocity of zero and the motion of A and C is causing the concavity.

One solution is just to randomly choose a new velocity for B such that $\frac{d\Delta}{dt}(t_b) > 0$. We can just keep randomly choosing until one has the desired property, or if we are selecting velocities from the unit disc we just need to select from the uniform distribution on the region formed by the intersection of the unit disc and the halfplane defined by $\frac{d\Delta}{dt}(t_b) > 0$. Then we just need to modify the events (if any) in the priority queue for the neighbours of B and insert the new event for B.

An alternative solution that does not require generating a new random number each bounce, is to choose a frame of reference moving with the velocity of A (so that A is stationary in this frame) and rotating about A such that the motion of C is along the line AC in this frame. We can then bounce B by reversing the component of its velocity perpendicular to AC. Note that this is done in a non-inertial frame of reference so that the energy of the system is not conserved.

If we use a heap to implement the priority queue then finding the minimum, insertion and deletion operations can be performed in time $O(\log n)$ and each event processed in time $O(\log n)$. If we perform $O(n)$ events, the total time to generate a convex polygon is $O(n \log n)$.

6.6 Triangulation

Choose a random topological triangulation of a polygon. Construct a convex polygon with Delaunay triangulation homeomorphic to this.

Atkinson and Sack [13] give a $O(n)$ algorithm for choosing a random triangulation of a convex polygon.

Dillencourt [87] gives a constructive proof for the realizability as a Delaunay triangulation of any triangulation of the interior of a simple polygon. A naive implementation of the construction will take $O(n^2)$ time. I show below how the construction can be performed in $O(n)$ time.

The total time to generate a convex polygon by this method is $O(n)$.

6.6.1 Realizing a Delaunay triangulation in $O(n^2)$ time.

Dillencourt's construction shows how to compute each angle of each triangle in the triangulation. If the values of the angles (measured in some arbitrary units such that s units form a straight angle) are a_i, then the a_is must satisfy the following properties:

1. For each vertex, $a_{i_1} + \ldots + a_{i_k} \leq s$ where $a_{i_1}, \ldots, a_{i_k}$ are the angles at that vertex. This says that the polygon is convex.

2. For each interior edge, $a_i + a_j < s$ where a_i and a_j are the two angles facing the edge. This says that the triangulation is Delaunay.

3. For each i, $a_i > 0$.

4. For each triangle, $a_i + a_j + a_k = s$ where a_i, a_j and a_k are the angles of the triangle.

The construction proceeds incrementally, computing values satisfying the above properties for progressively larger subtriangulations of the triangulation to be realized.

Initially we start with any triangle, set each a_i to 1 and s to 3, clearly satisfying the four properties above. Each step adds any triangle which shares a common edge (AB in figure 6.7) with one of the triangles in the subtriangulation.

For each triangle in a triangulation of a simple polygon we can define the *opposite corner* with respect to a triangle T in the triangulation as follows: starting at T follow a path within the triangulation to that the triangle. The opposite corner is the one opposite the last edge crossed.

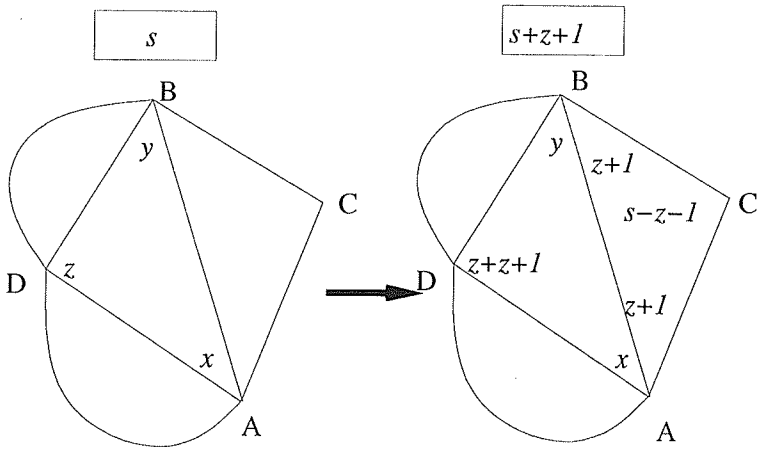


Figure 6.7: Adding triangle ACB

Let z be the value of the opposite corner to the new triangle (see figure 6.7). The new triangle is given values $z+1$, $z+1$ and $s-z-1$ at vertices A, B and C respectively. The value of s is replaced by $s' = s+z+1$. The value of the opposite corner with respect to ABC in every other triangle is increased by $z+1$.

Property 1 remains true for vertices other then A, B and C: since exactly one of the angles adjacent to each of these vertices is an opposite corner, both sides of the inequality $a_{i_1} + \ldots + a_{i_k} \leq s$ are increased by $z+1$. The totals at A and B are also increased by $z+1$, while there is only one angle at C, and it is clearly less than s'.

Property 2 remains true because for edges other than AB only one of the angles facing the edge is an opposite corner, while for AB we have $\angle BDA + \angle ACB = z+z+1+s-z-1 = s+z < s'$.

Property 3 is obviously still true.

Property 4 is true for $\triangle ABC$. For the other triangles it is still true since there is exactly one opposite corner per triangle.

Figure 6.8 gives an example of the steps in the construction.

Once all the angles of all the triangles have been computed we just need to pick positions for the endpoints of one edge and then trigonometry determines the positions of of all the other vertices.

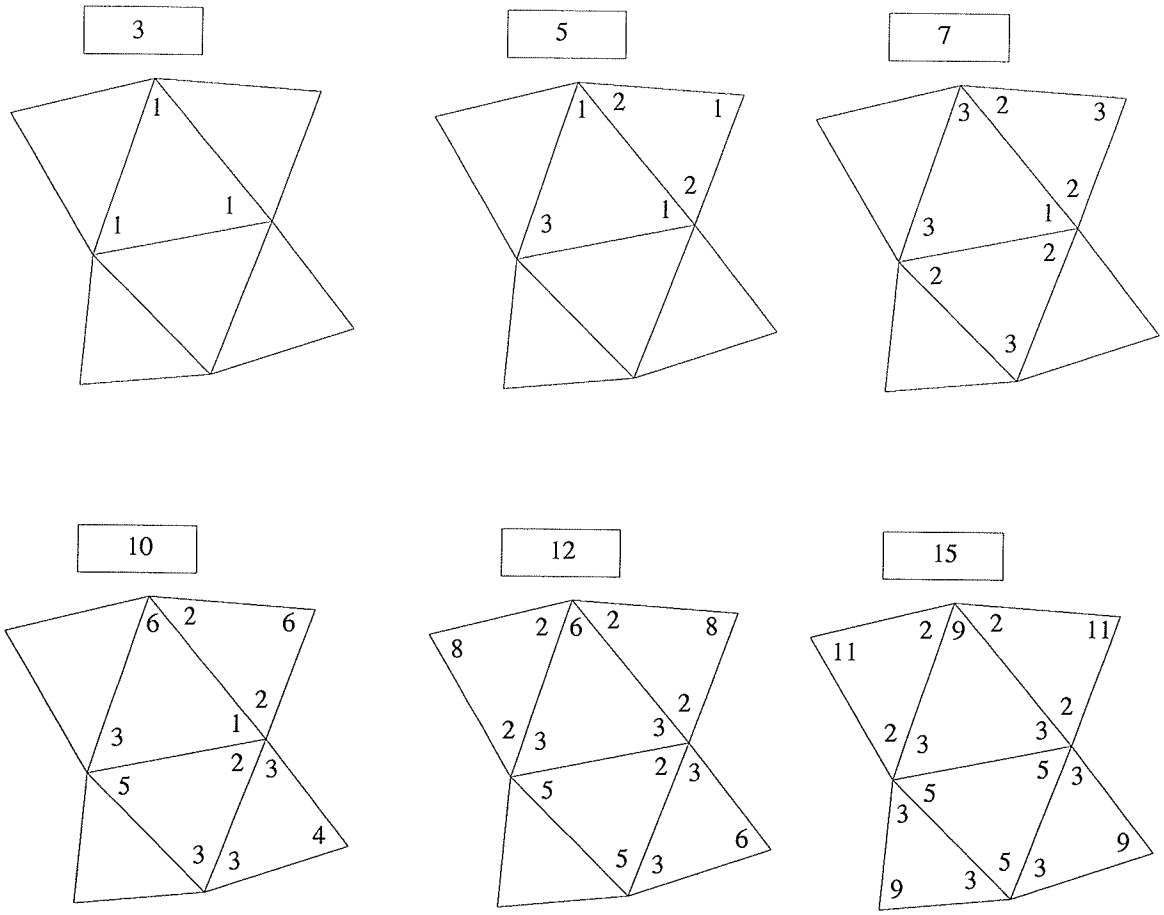


Figure 6.8: Realizing a Delaunay triangulation

Since an angle in each triangle must be updated at each step in the computation of the angles the total time required is $\Theta(n^2)$.

6.6.2 Realizing a Delaunay triangulation in $O(n)$ time.

The key to improving the execution time to $O(n)$ is the following observation: Since the angles of a triangle add to s, it doesn't matter if the value of one of the angles is incorrect, as long as we know which one it is.

Any given triangle ABC divides the triangulation into three pieces: those triangles whose opposite corner is A, those whose opposite corner is B and those whose opposite corner is C (see figure 6.9). If we add all the triangles whose opposite corner is A without updating the angles of ABC only the value of A will be incorrect. Its value can then be computed from the value of s and the other angles. Then the triangles opposite B can be added and the value of B then corrected and similarly for C.

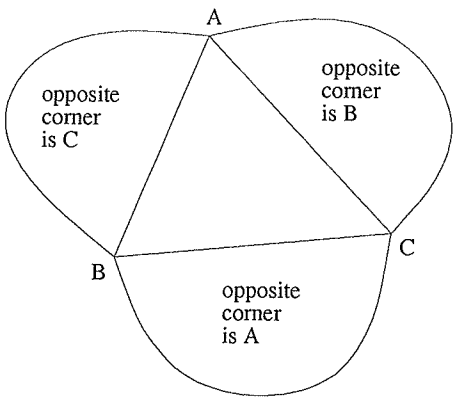


Figure 6.9: ABC divides the triangulation into three components

We wish to do this for *all* triangles, so the appropriate order is given by traversing the outer face of the dual graph of the polygon triangulation (see figure 6.10).

To describe the invariant for this algorithm we need one more bit of state—the *current triangle* of the traversal. By the opposite corner of a triangle we just mean the opposite corner with respect to the current triangle. (The current triangle does not have an opposite corner.)

For each angle i we store a value a_i'. The relationship between the a_i' and the a_i of

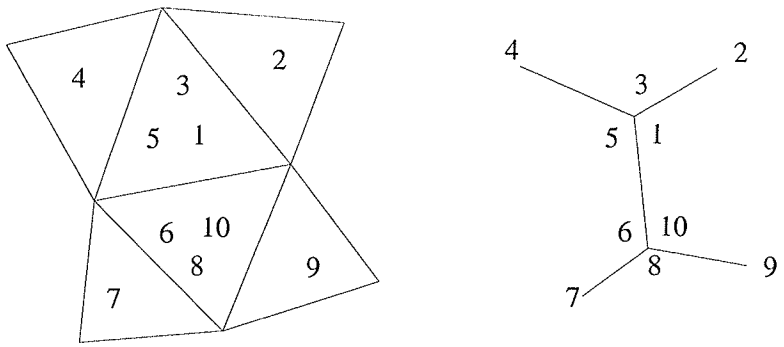


Figure 6.10: Dual of triangulation and traversal order

the previous section is quite simple: If i is not an opposite corner then $a_i = a'_i$. If i is an opposite corner then $a_i = s - a'_j - a'_k$, where j and k are the two other corners of the triangle.

The traversal order ensures that we only ever cross a single edge when moving from one current triangle to a new one. If the new current triangle is one we haven't visited before then we can compute the values for its angles and update s just as in section 6.6.1 (see figure 6.11). This will be correct since we know that the values for the current triangle are correct. The difference from section 6.6.1 is that we do not add $z + 1$ to all the opposite corners with respect to ABC. Since ABC is now the current triangle, these are all opposite corners and their values do not matter.

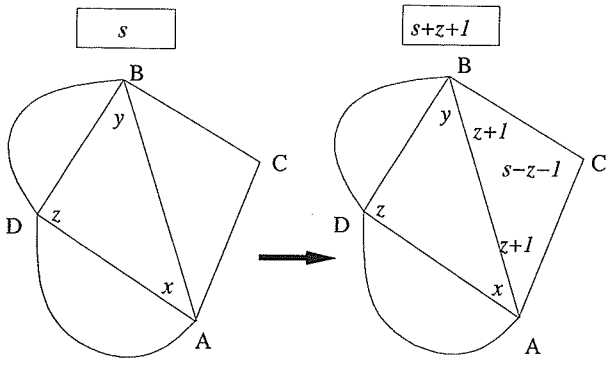


Figure 6.11: New current triangle is previously unvisited

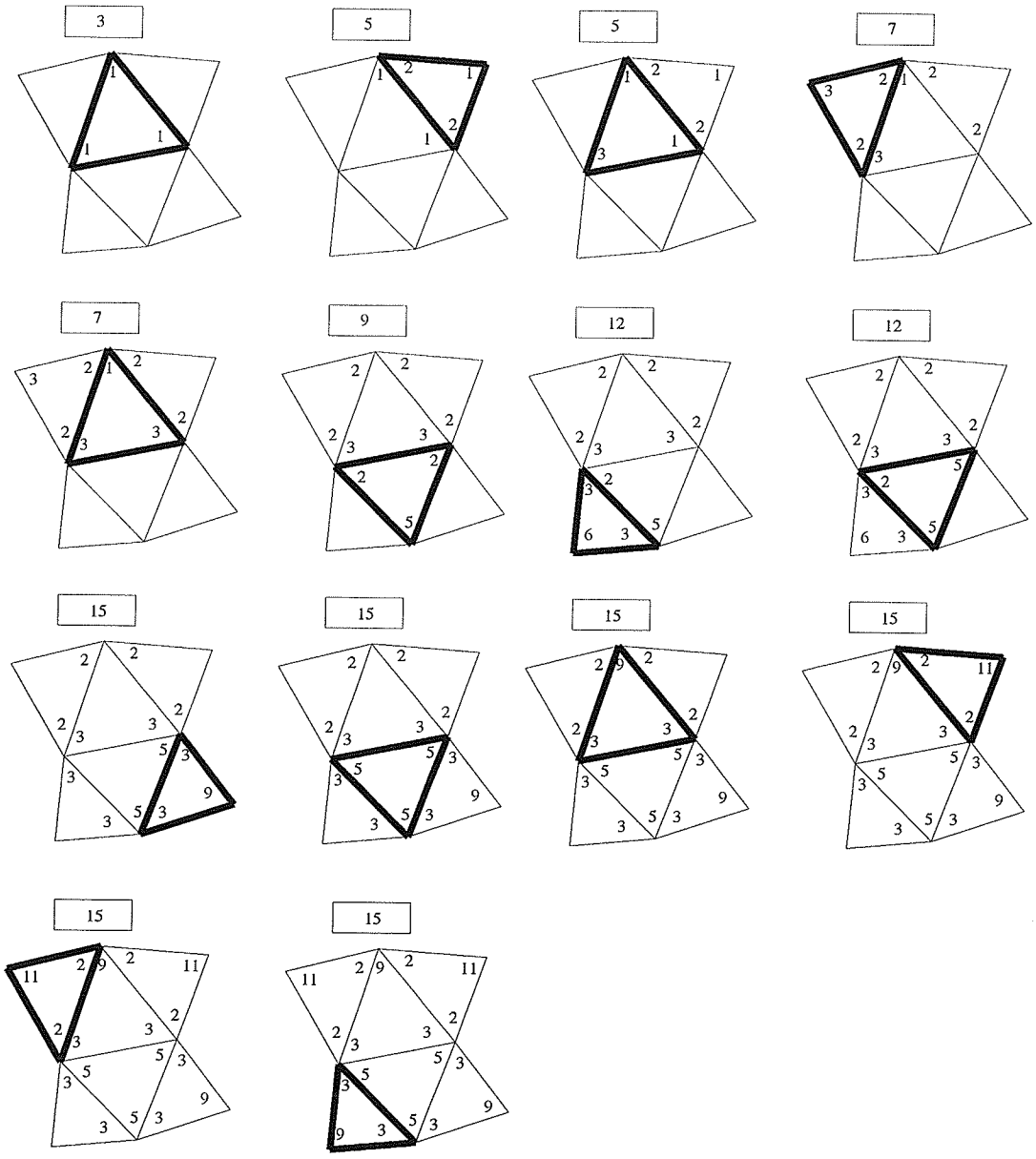


Figure 6.12: Realizing a Delaunay triangulation—$O(n)$ algorithm

If the new current triangle has been visited before, the corner opposite the edge crossed to enter this triangle is no longer an opposite corner, so we compute its value from s and the values of the two other corners.

One traversal of the outer face ensures that all the triangles are visited. A second one ensures that all the angles are correct. Figure 6.12 shows the steps for our example triangulation. The current triangle is highlighted in bold. The steps where no values change have not been shown.

Each step takes $O(1)$ time. The traversal crosses each edge exactly twice, so there are $O(n)$ steps and a total run time of $O(n)$.

6.6.3 Implementation

The following fragment of C implements the algorithm described above.

The topology of the triangulation is represented by three functions on the angles. `prev` and `next` give the next angle in the same triangle in the anticlockwise and clockwise directions respectively. `adj` is the successor in the clockwise ordering of angles which share a common vertex. The last angle in this ordering has `adj(i)=-1` In figure 6.13 we have `next(1)=5`, `prev(1)=9`, `adj(1)=2`, and `adj(2)=-1`.

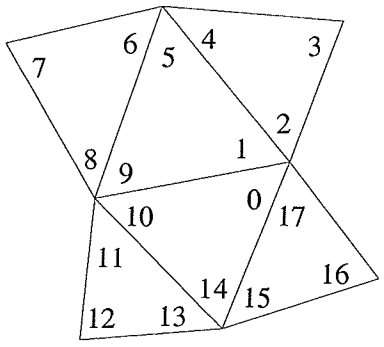


Figure 6.13: Traversal order for angles

We traverse the angles in the order in which they occur on the outer face (see figure 6.13). Let `i` is the current angle. If `adj(i)=0` then the successor to `i` in the ordering is `next(i)` in the same triangle. Otherwise, we cross a triangle edge to `adj(i)` and must update the angles of the new triangle. We store the values of the angles in array `a`, which is initially zero (so that we can test to see if we have entered a triangle for the first time by looking at

a[i]). We assume that the triangles are numbered from 0 to 3*n-1, where n is the number of triangles.

```
for (i = 0; i < 3*n; ++i)
  a[i] = 0;
i = 0;
a[i] = a[next(i)] = a[prev(i)] = 1;
s = 3;
for (j = 0; j < 2; ++j) {
  do {
    if (adj(i) == -1) {
      i = next(i);
    } else {
      i = adj(i);
      if (a[i] == 0) {
        z = a[next(adj(prev(i)))];
        a[prev(i)] = a[i] = z + 1;
        a[next(i)] = s - z -1;
        s = s + z + 1;
      } else {
        a[next(i)] = s - a[i] - a[prev[i]];
      }
    }
  } while (i != 0);
}
```

6.7 Dual

We can take the dual of the convex polygons produced by any of the above methods. For example, The Dual Rejection method takes the intersection of n half-spaces containing the origin and rejects the resulting polygon if it has fewer than n sides.

6.8 Conclusion

I have implemented the Rejection, Iteration and Vector methods described above.

Some other uses for my random convex polygons can be to determine how often random convex polygons were unimodal [6] and how often the minimum-area and minimum-perimeter-enclosing rectangles are different [80].

Chapter 7

Further Work

7.1 Performance of Delaunay triangulation algorithms

Not much has been done on actually measuring the performance of Delaunay triangulation algorithms. Implement all the constrained Delaunay triangulation algorithms described in chapter 2, the bucketing versions of the algorithms described in sections 2.3.1 and 2.4.1 and convex-distance-function versions of these algorithms 4.1.2 and measure the performance on a variety of distributions of sites and constraints.

7.2 Is Locality Necessary?

It may be possible to generalize the proof in section 3.6 to prove that systematic flip rules are generalized Delaunay rules, since I have not found a flip rule that is systematic but not local.

7.3 More Powerful Flip Rules

A generalization of the work in chapter 3 that might be considered is that of more powerful flip rules. For example, we could look at sets of three triangles that formed a convex pentagon and replace the triangulation of the pentagon with $GOT(P_1)$ of that pentagon. Such a rule would always be systematic for convex pentagons, (but not local since $GOT(P_1)$ (the Minimum Weight triangulation is not local), so would pass the tests in section 3.5.

The proofs in section 3.6 are obviously not applicable.

For this rule we could experiment using convex hexagons and a similar technique to that of section 3.5 could be used (although the directed flip graph is considerably more complicated).

7.4 Higher-Dimensional Convex-Distance-Function Delaunay triangulation

The structure of higher-dimensional convex-distance-function Delaunay triangulations is very different from that of higher-dimensional Euclidean Delaunay triangulations. It would be very interesting to investigate this structure and perhaps develop a more efficient algorithm than that described in section 4.2.1. Visualization tools such as IRIS Explorer would prove useful in visualizing the shape of the associated Voronoi diagram.

7.5 Robustness of Delaunay triangulation algorithms

When implemented using floating point arithmetic Delaunay triangulation algorithms sometimes fail because computations are inexact. This failure occurs because the approximated *DT* flip rule is not systematic and local. The tests used in section 3.5 can be used to examine this and compare various different implementations of the *DT* flip rule. (The results in appendix A suggest a variety of alternative implementations.)

7.6 Convex-Polygon Delaunay triangulation

It would be interesting to implement Aggarwal *et al.*'s deterministic linear algorithm as well as a sweepline and a flip algorithm and compare execution times with the other algorithms described in chapter 5.

7.7 Random convex polygons

Implement the other algorithms designed in chapter 6 and use them to test the algorithms in chapter 5 and also to measure how often random convex polygons were unimodal [6] and how often the minimum-area and minimum-perimeter-enclosing rectangles are different [80].

7.8 Prove linear number of points generated by the iteration algorithm

The iteration algorithm (section 6.3) for generating a random convex n-gon generated roughly $2n$ points. It would be interesting to prove that this number was $O(n)$.

Chapter 8

Conclusion

A useful heuristic in problem solving is to look for problems that you can use your favourite technique on. Faced with a large number of alternative definitions of optimality, in chapter 3 I looked for optimal triangulations defined by flip rules for which there were fast algorithms like those for the Delaunay triangulation rather than trying to find fast algorithms for each different definition. The "systematic" and "local" properties capture the properties of the Delaunay triangulation that are needed for the fast Delaunay triangulation algorithms to work.

The experiments described in section 3.5 convinced me that that there were no systematic local flip rules other than the Delaunay rule. I also discovered several more optimality properties of the Delaunay triangulation (proved in appendix A).

I proved that the Delaunay rule is the only systematic local flip rotation and translation-invariant rule (section 3.6.3). When I tried the natural extension of this result to convex-distance-function Delaunay rules, I discovered empty-shape triangulations which generalize convex-distance-function Delaunay triangulations and are the only systematic local homothetic rules (proved in section 3.6.4).

All of the algorithms I describe in chapter 2 can be used to compute empty-shape triangulations. To demonstrate this I implemented a sweepline algorithm for empty-shape triangulations using Miranda (section 4.1.1). The greater expressive power of Miranda as compared to a language like C allowed a very compact implementation of the sweepline algorithm—100 lines of Miranda as opposed to 900 lines of C [120] or 2500 lines of C [199]. This enabled me to present the complete implementation, together with extensive commentary in a few pages of this thesis.

It is easier to compute empty-shape triangulations than convex-distance-function Delaunay triangulations. (After all, I discovered them while looking for triangulations that were easy to compute.) So, extending the convex distance function by adding shapes to "round" its corners allows me to compute an empty-shape triangulation that is a superset of the convex-distance-function Delaunay triangulation (and from which the convex-distance-function Delaunay triangulation is easily extracted) (section 4.1.2). This provides a way for all the algorithms described in chapter 2 to be used to compute convex-distance-function Delaunay triangulations. Previously published algorithms for convex-distance-function Delaunay triangulations dealt with the problems caused by corners in the distance function by modifying the algorithm—my approach modifies the distance function. This approach yields simple algorithms for constrained convex-distance-function Delaunay triangulations as well.

Naturally I wanted to consider if this approach generalized to three dimensions. When looking at the published algorithm for three dimensional convex-distance-function Delaunay triangulation [285], I found something very disturbing—the authors observed that Delaunay tetrahedra could intersect. I wrote some simple programs to test the properties of random configurations and found the counterexamples presented in section 4.2.1. These show that the previously published algorithm is incorrect, so I devised a correct one.

In my literature review I found a vast number of published Delaunay triangulation algorithms (table 2.2). I found that a standard taxonomy for sorting algorithms applied to triangulation algorithms proved useful for classifying them and thinking about them. My work in section 4.1.2 amounts to filling in the empty boxes in the "convex distance function" row of table 2.2, while the proofs in section 3.6 show why the empty boxes in the "non-Delaunay" row are likely to remain empty.

Devijver and Maybank [83] analyze an algorithm for convex-polygon Delaunay triangulation that they claim satisfies a "minimum space complexity constraint". They computed the average execution time over all possible triangulations of the convex polygon. Unfortunately, their algorithm was very inefficient and did not satisfy their "minimum space complexity constraint". I did a similar analysis for more efficient algorithms and designed an algorithm that satisfies the "minimum space complexity constraint" (section 5.4).

Algorithms should be implemented and tested, so I had to have some way of generating "random" convex polygons, in order to test the algorithms analyzed in section 5.4. Since the concept of a "random" convex polygon is not well defined, I used a variety of operational

definitions (chapter 6). To get efficient algorithms for generating convex polygons by my definitions, I developed a data structure that allows generation of variates in time $O(\log n)$ from a dynamically changing discrete distribution (section 6.2) and a $O(n)$ algorithm for realizing a Delaunay triangulation of a convex polygon (section 6.6.2).

The results of testing the algorithms analyzed in section 5.4 on the random convex polygons generated by the methods of chapter 6 were surprising—all the algorithms had worst-case execution times rather than that that predicted by my analysis, thus showing that each triangulation of the polygons produced by my methods were not equally likely. In order to achieve the expected execution times predicted by my analysis it was necessary to randomize the algorithms. This is an interesting result. Randomization has found many applications in computational geometry [61] in recent years and in many cases it may be unnecessary to add an explicit randomization step if the the input is already "random" in some sense. I have discovered a case here where it definitely is necessary.

Appendix A

$$DT = -r_1 = -\alpha_\infty = (rR)_1 = abc_1 = (Rr^2/\Delta)_0$$

We will prove that each of the rules R_*, $-r_1$, $-\alpha_\infty$, $(rR)_1$, abc_1 and $(Rr^2/\Delta)_0$ produces the same result as the Delaunay triangulation on a convex quadrilateral.

Let $ABCD$ be a convex quadrilateral with Delaunay triangulation ABC, ACD (so $DT(ABCD) = AC$). D is outside the circumcircle of ABC and B is outside the circumcircle of ACD. We have already proved in theorem 3 that $DT = R_*$.

Theorem 16 $-r_1 = DT$.

PROOF. Let P be the point where the diagonals of $ABCD$ intersect. Let $r_A = r(DAB)$, $r_{AB} = r(PAB)$, $r_{DA} = r(PDA)$ and h_A be the length of the altitude at A in triangle DAB (see figure A.1).

Demir [79] has proved the following relation between these quantities:

$$r_{DA} + r_{AB} - r_A = 2\frac{r_{DA}r_{AB}}{h_A}. \tag{A.1}$$

Applying this relation to triangles ABC, BCD, and CDA yields:

$$r_{AB} + r_{BC} - r_B = 2\frac{r_{AB}r_{BC}}{h_B}, \tag{A.2}$$

$$r_{BC} + r_{CD} - r_C = 2\frac{r_{BC}r_{CD}}{h_C}, \tag{A.3}$$

$$r_{CD} + r_{DA} - r_D = 2\frac{r_{CD}r_{DA}}{h_D}. \tag{A.4}$$

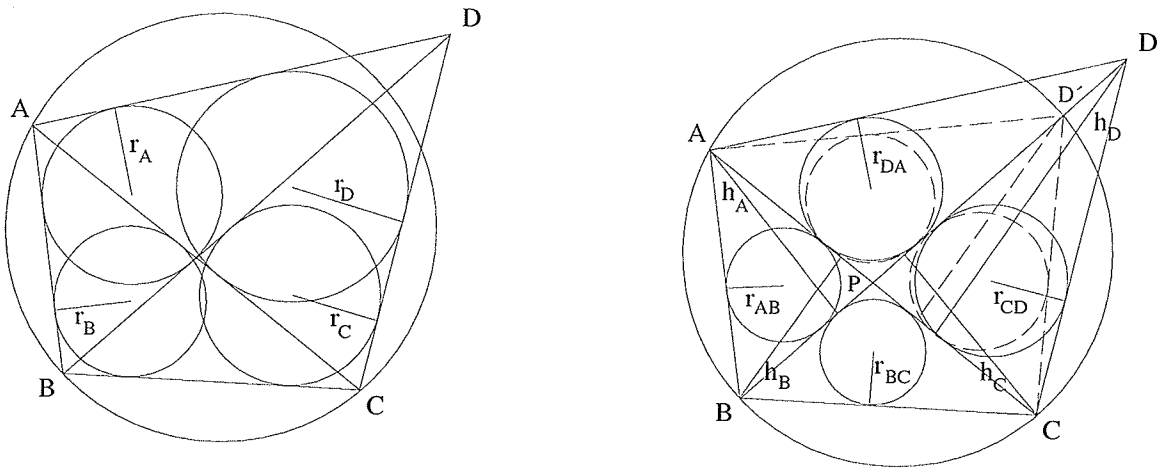


Figure A.1: Inscribed Circles

Adding A.1 and A.3 and subtracting A.2 and A.4 we get:

$$r_B + r_D - r_A - r_C = 2\left(\frac{r_{DA}r_{AB}}{h_A} - \frac{r_{AB}r_{BC}}{h_B} + \frac{r_{BC}r_{CD}}{h_C} - \frac{r_{CD}r_{DA}}{h_D}\right). \qquad \text{(A.5)}$$

Now let D' be the point where BD intersects the circumcircle of ABC, and define $r'DA$, $r'CD$ and h'_D appropriately.

$PD'A$ is similar to PCB; so

$$\frac{r'_{DA}}{h_A} = \frac{r_{BC}}{h_B} \quad \text{and} \quad \frac{r_{BC}}{h_C} - \frac{r'_{DA}}{h'_D}.$$

Clearly $r'_{DA} < r_{DA}$. Also,

$$\begin{aligned}
\frac{r_{DA}}{h_D} &= \frac{2\Delta(PDA)}{h_D(|PD| + |DA| + |AP|)} \\
&= \frac{|AP|}{(|PD| + |DA| + |AP|)} \\
&< \frac{|AP|}{(|PD'| + |D'A| + |AP|)} \\
&= \frac{r'_{DA}}{h'_D}.
\end{aligned}$$

Using these results in equation A.5 we get

$$r_B + r_D - r_A - r_C > 2\left(r_{AB}(\frac{r'_{DA}}{h_A} - \frac{r_{BC}}{h_B}) + r_{CD}(\frac{r_{BC}}{h_C} - \frac{r'_{DA}}{h'_D})\right) = 0.$$

That is, $-r_1(ABCD) = AC$. □

Theorem 17 $-\alpha_\infty = DT$ *(also proved in [192, 202, 298]).*

PROOF. Because D is outside $\bigcirc ABC$, a plane geometry theorem [111] states that $\alpha = \angle ADB < \angle ACB = \alpha'$ (see figure A.2). Similarly, the unprimed angles are less than

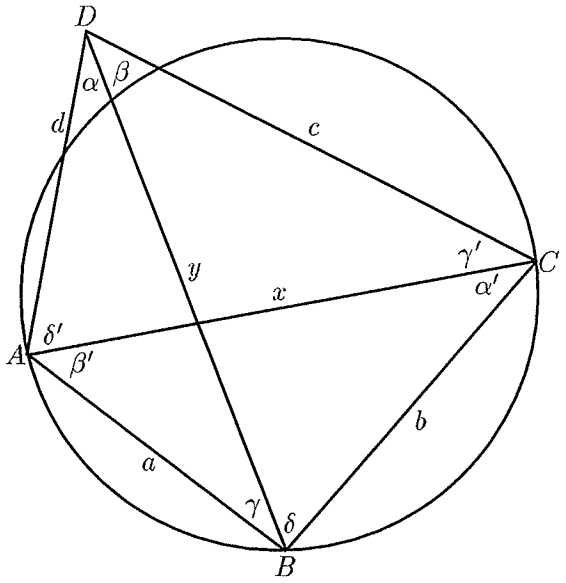


Figure A.2: Quadrilateral $ABCD$

the corresponding primed angles in figure A.2. Also, $\min(\angle ABD, \angle CBD) < \angle ABC$ and $\min(\angle ADB, \angle BDC) < \angle ADC$. Hence

$$\begin{aligned}\max(-\alpha(ABC), -\alpha(ACD)) &= -\min(\angle ABC, \angle ACB, \angle BAC, \angle ACD, \angle ADC, \angle CAD) \\ &< -\min(\angle ABD, \angle CBD, \angle ADB, \angle BDC) \\ &\leq \max(-\alpha(ABD), -\alpha(BCD)).\end{aligned}$$

That is, $-\alpha_\infty(ABCD) = AC$. □

Theorem 18 $abc_1 = DT$.

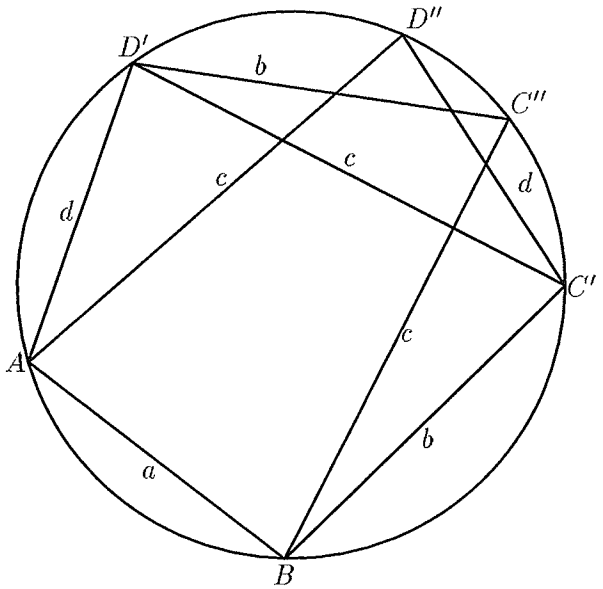


Figure A.3: Cyclic quadrilaterals

PROOF. Let $a = |AB|$, $b = |BC|$, $c = |CD|$, $d = |AD|$, $x = |AC|$ and $y = |BD|$. (See figure A.2). There is a unique cyclic quadrilateral $ABC'D'$ with $|BC'| = b$, $|C'D'| = c$ and $|AD'| = d$ (figure A.3). Let $x' = |AC'|$ and $y' = |BD'|$.

If $\angle ABC' \leq \angle ABC$ then $x' \leq x$ (Cosine law). Consequently $\angle AD'C' \leq \angle ADC$. Hence, $\angle ABC + \angle ADC \leq ABC' + \angle AD'C' = 180°$ (opposite angles of a cyclic quadrilateral are supplementary). Since $\angle ABC + \angle BCD + \angle CDA + \angle DAB = 360°$ we have $\angle BCD + \angle DAB \leq 180° \leq \angle ABC + \angle ADC$. But (see figure A.2)

$$\begin{aligned} \angle BCD + \angle DAB &= \alpha' + \beta' + \gamma' + \delta' \\ &> \alpha + \beta + \gamma + \delta \\ &= \angle ABC + \angle ADC. \end{aligned}$$

This is a contradiction. Hence $\angle ABC' > \angle ABC$, $x' > x$ and $y' < y$.

Ptolemy's theorem [9] gives us $x'y' = ac + bd$. If we construct two more cyclic quadrilaterals $ABC'D''$ and $ABC''D'$ (figure A.3) by setting $|D''A| = c$ and $|BC''| = c$ and let $z' = |AC''| = |BD''|$ then Ptolemy's theorem applied to these two triangulations gives us $x'z' = ad + bc$ and $y'z' = ab + cd$.

$$abc(ABC) + abc(ACD) = abx + cdx$$

$$\begin{aligned} &< x'(ab+cd) \\ &= x'y'z' \\ &= y'(ad+bc) \\ &< ady+bcy \\ &= abc(ABD)+abc(BCD) \end{aligned}$$

That is, $abc_1(ABCD) = AC$. □

Theorem 19 $rR_1 = DT$.

PROOF. If a triangle T has side lengths p, q and r and area Δ then badness function $rR(T) = (2\Delta/(p+q+r))(pqr/(4\Delta)) = pqr/(2(p+q+r))$ (see table 3.2). Note that if $r' > r$ and $p, q > 0$ then $pqr'/(p+q+r') > pqr/(p+q+r)$.

$$\begin{aligned} 2(rR(ABC)+rR(ACD)) &= \frac{abx}{a+b+x}+\frac{cdx}{c+d+x} \\ &< \frac{abx'}{a+b+x'}+\frac{cdx'}{c+d+x'} \\ &= \frac{x'(abc+abd+abx'+acd+bcd+cdx')}{x'(a+b+c+d+x')+(a+b)(c+d)} \\ &= \frac{x'(abc+abd+acd+bcd+x'(ab+cd))}{x'(a+b+c+d+x')+x'y'+x'z'} \\ &= \frac{abc+abd+acd+bcd+x'y'z'}{a+b+c+d+x'+y'+z'} \\ &= \frac{bcy'}{b+c+y'}+\frac{ady'}{a+d+y'} \quad \text{by symmetry} \\ &< \frac{bcy}{b+c+y}+\frac{ady}{a+d+y} \\ &= 2(rR(ABD)+rR(BCD)) \end{aligned}$$

That is, $rR_1(ABCD) = AC$. □

Theorem 20 $(Rr^2/\Delta)_0 = DT$.

PROOF. If a triangle T has side lengths p, q and r let

$$g(r) = \frac{Rr^2}{\Delta}(T) = \frac{pqr}{(p+q+r)^2}.$$

Now, because T is a triangle, $p, q > 0$ and $0 < r < p + q$; so

$$g'(r) = \frac{pq((p+q+r)^2 - 2r(p+q+r))}{(p+q+r)^4} = \frac{pq(p+q-r)}{(p+q+r)^3} > 0.$$

Hence if T' has side lengths p, q, r' and $r' > r$ then $g(r') > g(r)$.

$$\begin{aligned}
\frac{Rr^2}{\Delta}(ABC)\frac{Rr^2}{\Delta}(ACD) &= \frac{abx}{(a+b+x)^2}\frac{cdx}{(c+d+x)^2} \\
&< \frac{abcdx'x'}{((a+b+x')(c+d+x'))^2} \\
&= \frac{abcd}{(a+b+c+d+x'+y'+z')^2} \\
&< \frac{ady}{(a+d+y)^2}\frac{bcy}{(b+c+y)^2} \\
&= \frac{Rr^2}{\Delta}(ABD)\frac{Rr^2}{\Delta}(BCD).
\end{aligned}$$

That is, $\dfrac{Rr^2}{\Delta}(ABCD) = AC.$ □

If $DT(ABCD) =$ **either**, then $ABCD$ is a cyclic quadrilateral and all the inequalities in the above theorems become equalities.

In particular, from theorem 16 we have

Theorem 21 *$r(ABC) + r(CDA) = r(DAB) + r(BCD)$ if and only if $ABCD$ is cyclic.*

The first known statement of the "if" part of this theorem was on a tablet hung in a Japanese temple in 1800 [129]. It is the most celebrated Japanese temple geometry theorem, mentioned or proved in [129, 155, 166, 331]. None of these proofs can be easily modified to prove the converse; so the "only if" part would appear to be a new result in elementary geometry.

The results in this section do not generalize to three dimensions as the following counterexample shows.

The points $A = (0,0,0)$, $B = (1,0,0)$, $C = (0,1,0)$, $D = (0,0,1)$, $E = (1,1,1)$ lie on a common sphere. The convex polyhedron $ABCDE$ can be divided into tetrahedra in two ways: the two tetrahedra $ABCD$ and $BCDE$, or the three tetrahedra $AEBC$, $AECD$, and $AEDB$.

tetrahedron	volume	area	inradius	circumradius
$ABCD$	$1/6$	$(3+\sqrt{3})/2$	$\frac{1}{2}-\frac{1}{6}\sqrt{3}$	$1/2$
$BCDE$	$1/3$	$2\sqrt{3}$	$\frac{1}{6}\sqrt{3}$	$1/2$
$AEBC$	$1/6$	$(1+2\sqrt{2}+\sqrt{3})/2$	$1/(1+2\sqrt{2}+\sqrt{3})$	$1/2$
$AECD$	$1/6$	$(1+2\sqrt{2}+\sqrt{3})/2$	$1/(1+2\sqrt{2}+\sqrt{3})$	$1/2$
$AEDB$	$1/6$	$(1+2\sqrt{2}+\sqrt{3})/2$	$1/(1+2\sqrt{2}+\sqrt{3})$	$1/2$

Clearly $r(ABCD)+r(BCDE) \neq r(AEBC)+r(AECD)+r(AEDB)$. Similar calculations show that this is also a counterexample for $(rR)_1$, $(Rr^2/\Delta)_0$, and $-\alpha_\infty$ (no matter whether we use face angle, dihedral angle or solid angle).

Appendix B

$DT \neq P_1$, $DT \neq s_2$, $DT \neq -\alpha_1$

It is sufficient to give a single counterexample for each.

Consider the points $A = (1,0)$, $B = (1/2, \sqrt{3}/2)$, $C = (-1,0)$ and $D = (1/2, -\sqrt{3}/2)$. These points all lie on the unit circle, so $DT(ABCD) =$ **either**. $|AC| = 2 > \sqrt{3} = |BD|$, so $P_1(ABCD) = BD$. The triangle angles are as shown in figure B.1. The sum of the minimum triangle angles is 60° for the triangulation $\{ABC, CDA\}$ and 90° for the triangulation $\{ABD, BCD\}$. Hence, $-\alpha_1(ABCD) = BD$. The sum of the squares of the differences of the angles form 60° is $4(30)^2 = 3600$ for the triangulation $\{ABC, CDA\}$ and $2(30)^2 + 60^2 = 5400$ for the triangulation $\{ABD, BCD\}$. Hence, $s_2(ABCD) = AC$.

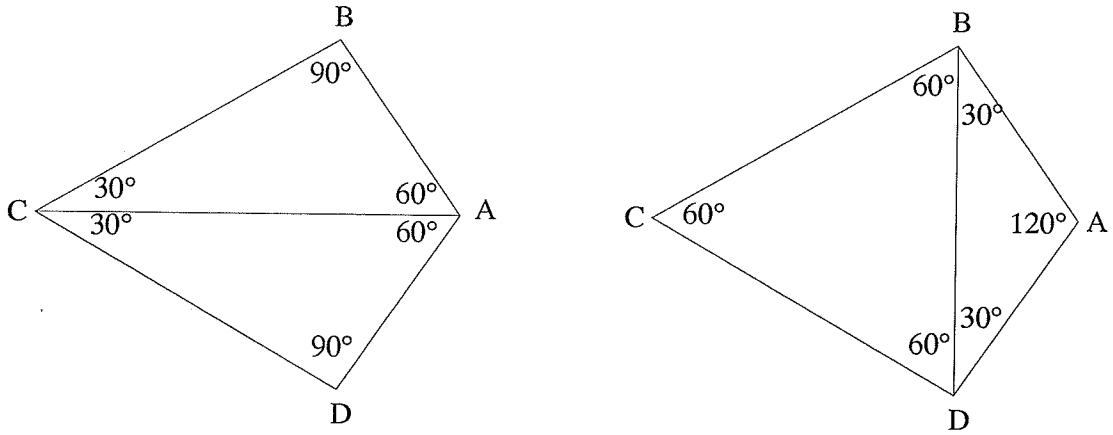


Figure B.1: A counterexample

Clearly, by perturbing the point A slightly so that it lies inside $\bigcirc BCD$, we can produce

$A'BCD$ such that its Delaunay triangulation does not minimize the total edge length, contradicting Shamos and Hoey [295] and Dobkin [89]. This was first noted by Lloyd [215].

The same $A'BCD$ also contradicts the claim by several authors [150, 206, 257, 260, 294] that the Delaunay triangulation minimizes the sum of the minimum triangle angles.

By perturbing the point A slightly so that it lies outside $\bigcirc BCD$ we can produce $A'BCD$ such that its Delaunay triangulation does not minimize the standard deviation of the triangle angles, contradicting Watson [329].

Appendix C

Calculation of badness measures

Let $A = (x_A, y_A)$, $B = (x_B, y_B)$ and $C = (x_C, y_C)$. Calculation of most of the measures in table 3.2 is straightforward. For example,

$$\Delta(ABC) = \tfrac{1}{2}(x_A y_B + x_B y_C + x_C y_A - x_A y_C - x_B y_A - x_C y_B).$$

Definition. $\text{range}(a_1, \ldots, a_n) = \max(a_1, \ldots, a_n) - \min(a_1, \ldots, a_n)$

$$R^{\infty}(ABC) = \tfrac{1}{2}\max(\text{range}(x_A, x_B, x_C), \text{range}(x_A, x_B, x_C)).$$

Calculating $r^{\infty}(ABC)$ is a little more complicated. $r^{\infty}(ABC)$ is the radius of the largest square parallel to the axes that can fit inside ABC. Call this square S. S must touch all three sides of ABC. (For if it did not touch a given side it could be moved toward that side so that it did not touch any side and then made larger.)

Relabel the points so that $x_C \leq x_B \leq x_A$. There are two possibilities: y_B is between y_A and y_C or it is not.

Case 1 y_B is not between y_A and y_C (see figure C.1). By reflecting about the x and y axes if necessary we can get $y_C \leq y_A \leq y_B$. Let D, E and F be the points where S touches BC, AC and AB respectively.

AB has a negative slope and S is below it so F must be the upper right corner of S. CB and AC have positive slopes and S is below CB and above AC so D is the upper left and E the lower right corner of S (see figure C.1).

Let H be the intersection of a vertical line through B and AC, and let G be the intersection of a horizontal line through A and BC. Because BH is parallel to EF,

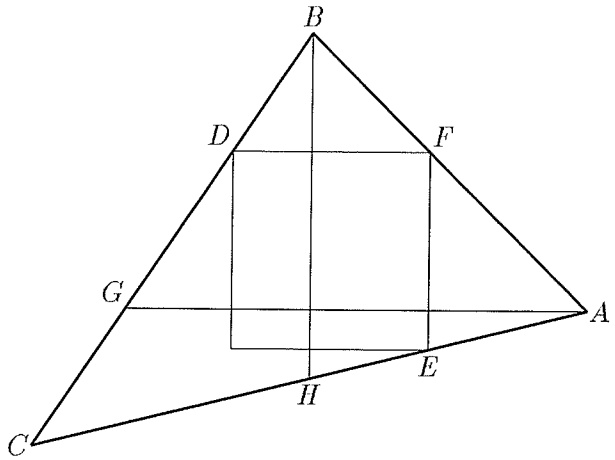


Figure C.1: Case 1: y_B not between y_A and y_C

the triangles AFE and ABH are similar. Therefore,

$$\frac{AF}{AB} = \frac{FE}{BH}. \tag{C.1}$$

Also BDF and BGA are similar; so

$$\frac{BF}{AB} = \frac{DF}{AG}. \tag{C.2}$$

Since $AF + BF = AB$, if we add equations C.1 and C.2 we get

$$\begin{aligned} 1 &= \frac{FE}{BH} + \frac{FE}{AG}, \\ 2r^{\infty}(ABC) = FE &= \frac{1}{\frac{1}{BH} + \frac{1}{AG}}. \end{aligned}$$

Now, $\Delta(ABH) = \frac{1}{2} \cdot BH \cdot (x_A - x_B)$ and $\Delta(HBC) = \frac{1}{2} \cdot BH \cdot (x_B - x_C)$; so $\Delta(ABC) = \Delta(ABH) + \Delta(HBC) = \frac{1}{2} \cdot BH \cdot (x_A - x_C)$, and consequently $BH = 2\Delta(ABC)/(x_A - x_C)$. Similarly $AG = 2\Delta(ABC)/(y_B - y_C)$, and so

$$r^{\infty}(ABC) = \frac{\Delta(ABC)}{(x_A - x_C + y_B - y_C)}.$$

Case 2 y_B is between y_A and y_C (see figure C.2). If necessary we can reflect about the x axis to get $y_C \leq y_B \leq y_A$. As in case 1, D and E are the upper left and lower

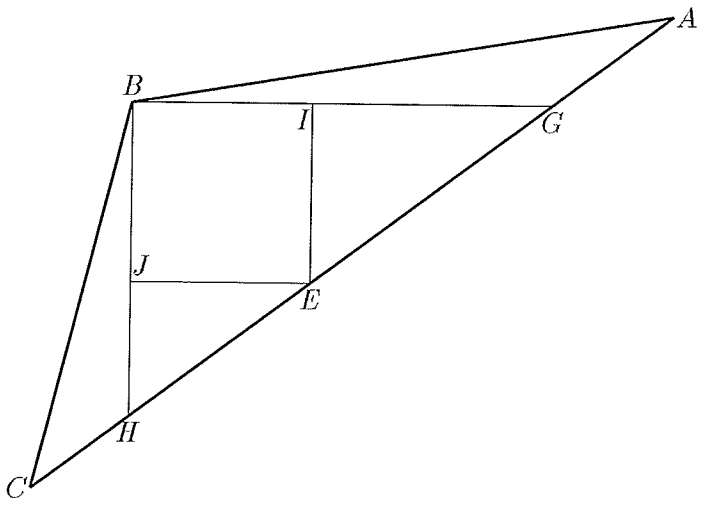


Figure C.2: Case 2: y_B is between y_A and y_C

right corners of S. However, AB now has a positive slope so F is the upper left corner of S. This means $D = F = B$. Let the upper right corner of S be I and G be the intersection of BI and AC. Let the lower left corner of S be J and H be the intersection of BJ and AC (see figure C.2). Triangles BHG, IEG and JHE are similar; so just as in case 1 we get $BI = 1/(1/BH + 1/BG)$ and

$$r^{\infty}(ABC) = \frac{\Delta(ABC)}{x_A - x_C + y_A - y_C}.$$

Combining the formulæ for each case and allowing for relabelling and reflection gives

$$\begin{aligned} r^{\infty}(ABC) &= \frac{\Delta(ABC)}{\text{range}(x_A, x_B, x_C) + \text{range}(y_A, y_B, y_C)} \\ &= 2\Delta(ABC)/P^{\infty}(ABC). \end{aligned}$$

Appendix D

Miscellaneous function definitions

Here are some Miranda functions required by the programs in chapter 4 that were not included in chapter 4.

Priority Queue ADT

First, an implementation of the priority queue ADT specified on page 120.

To make it easier to understand the operations, this implementation is very simple, using a list and the push operation takes time $O(n)$ instead of the optimal $O(\log n)$ possible with a heap-ordered tree.

We represent the priority queue by a pair containing the function that defines the ordering of events, and a list of pairs of the event order and the event. For example, if `f a < f b` then the priority queue

```
push (push (empty f) a) b)
```

will be represented by the pair

```
(f,[(f a, a), (f b, b)])
```

The type definition:

```
> priority * ** == (*->**,[(**,*)])
```

The implementation of the operations is straightforward:

```
> top = snd.hd.snd
> pop (p,x:xs) = (p,xs)
```

```
> empty p = (p,[])
> isempty (p,xs) = xs = []

> push (p,xs) y
>  = (p,push' xs (p y,y))
> push' [] y = [y]
> push' (x:xs) y = y:x:xs, if y<=x
>                = x:(push' xs y), otherwise

> remove (p,xs) y
>  = (p,remove' xs (p y,y))
> remove' [] y = []
> remove' (x:xs) y = xs, if y=x
>                  = (x:xs), if y<x
>                  = x:(remove' xs y), otherwise
```

Ordered Sequence ADT

A simple implementation of the ordered sequence ADT specified on page 121.

As with the priority queue implementation above we use a list and the `insert` operation takes time $O(n)$ instead of the optimal $O(\log n)$ possible with a balanced tree.

We represent the ordered sequence as a triple containing the `before` function, the list of sites, and a history list of all the operations that have been performed on the sequence. The history list is not needed for the `insert`, `delete` and `create` operations, but was found useful for debugging purposes. Since Miranda is a lazy language, there is no overhead in leaving it in.

```
> ordered_seq_point == ((point,point) -> point -> bool,[point],[op])
> op ::= Insert point [point] |
>        Delete (point,point) [point] |
>        Create [point]
```

In our implementation of `insert`, we are careful to check that the ordered sequence invariant holds.

```
> insert (before,as,h) x
```

```
>   = ((before,prefix++[a,b,x,b,c]++suffix,Insert x as:h),[a,b,c])
>     where befores = tl(init(map ((converse before) x) (pairs as)))
>                     || ignore dummy elements
>           pos = #(takewhile (=False) befores), check
>               = error invariant failure in insert, otherwise
>           prefix = take (pos) as
>           (a:b:c:suffix) = drop (pos) as
>           check = and(dropwhile (=False) befores)

> delete (before,as,h) p
>  = ((before,prefix++[a,b,d,e]++suffix,Delete p as:h)
>     ,[a,b,d,e]), if #(drop (pos-2) as)>=5
>  = error (delete++show (pos, #(drop (pos-2) as),as,p)
>           ++lay (map show h)), otherwise
>    where pos = position (pairs as) p
>          prefix = take (pos-2) as
>          (a:b:c:d:e:suffix) = drop (pos-2) as

> create before as
>  = (before,as,[Create as])
```

Miscellaneous functions

These were considered standard, so were not included in section 4.1 and are included here for the sake of completeness.

The Euclidean distance between two points.

```
> d2 (a,b) (c,d) = sqrt((a-c)^2+(b-d)^2)
```

Twice the signed area of a polygon.

```
> polygon == [point]
> area :: polygon -> num
> area ps = sum [x*y'-y*x' |
>                ((x,y),(x',y')) <- zip2 ps (tl ps ++ [hd ps])]
```

The centre of the circle through a set of co-circular points.

```
> circle_centre :: [point] -> point
> circle_centre t
>   = (cx,cy), divisor ~= 0
>   = error (show t), otherwise
>     where ds = map d t
>           xs = map fst t
>           ys = map snd t
>           divisor = 2*area t
>           cx = area (zip2 ds ys)/divisor
>           cy = area (zip2 xs ds)/divisor
>           d (x,y) = x^2 + y^2
```

Pairs of elements

```
> pairs xs = zip2 xs (tl xs)
```

Bibliography

[1] Edwin A. Abbott. *Flatland*, pages 3–4, 8, 10–11. Dover, New York, 1952. (Originally published in 1884).

[2] Karl Abrahamson. On the modality of convex polygons. *Discrete and Computational Geometry*, 5:409–419, 1990.

[3] Adobe Systems Incorporated. *PostScript Language Reference Manual.* Addison-Wesley, Reading, Massachusetts, 1985.

[4] A. Aggarwal, H. Edelsbrunner, P. Raghavan, and P. Tiwari. Optimal time bounds for some proximity problems in the plane. *Inform. Process. Lett.*, 42:55–60, 1992.

[5] Alok Aggarwal, Leonidas J. Guibas, and James Saxe. A linear-time algorithm for computing the Voronoi diagram of a convex polygon. *Discrete and Computational Geometry*, 4:591–604, 1989.

[6] Alok Aggarwal and Robert C. Melville. Fast computation of the modality of polygons. *Journal of Algorithms*, 7:369–381, 1986.

[7] Michael E. Agishtein and Alexander A. Migdal. Smooth surface reconstruction from scattered data points. *Computers & Graphics (Pergamon)*, 15(1):29–39, 1991.

[8] H. Akima. A method of bivariate interpolation on smooth surface fitting for irregularly distributed data points. *ACM Transactions on Mathematical Software*, 4:148–159, 1978.

[9] N. Altshiller-Court. *College Geometry.* Barnes & Noble, New York, 1952.

[10] D. B. Arnold and W. J. Milne. The use of Voronoi tessellations in processing soil survey results. *IEEE Computer Graphics and Applications*, pages 22–28, March 1984.

[11] T. Asano, M. Edahiro, H. Imai, M. Iri, and K. Murota. Bucketing techniques in computational geometry. In G. T. Toussaint, editor, *Computational Geometry*, pages 153–195. North-Holland, 1985.

[12] Ta. Asano, Te. Asano, L. J. Guibas, J. Hershberger, and H. Imai. Visibility of disjoint polygons. *Algorithmica*, 1:49–63, 1986.

[13] M. D. Atkinson and J.-R. Sack. Generating binary trees at random. *Information Processing Letters*, 41:21–23, 1992.

[14] S. Auerbach and H. Schaeben. Surface representations reproducing given digitized contour lines. *Mathematical Geology*, 22(6):723–742, 1990.

[15] F. Aurenhammer. Voronoi diagrams—a survey of a fundamental geometric data structure. *Computing Surveys*, 23(3):345–405, 1991.

[16] D. Avis and K. Fukuda. A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra. In *Proc. 7th Annu. ACM Sympos. Comput. Geom.*, pages 98–104, 1991.

[17] I. Babuška and A. Aziz. On the angle condition in the finite element method. *SIAM J. Numer. Analysis*, 13:214–227, 1976.

[18] Brenda S. Baker, Eric Grosse, and Conor S. Rafferty. Nonobtuse triangulation of polygons. *Discrete and Computational Geometry*, 3(2):147–168, January 1988.

[19] R. E. Barnhill. Representation and approximation of surfaces. In J. R. Rice, editor, *Math. Software III*, pages 69–120. Academic Press, New York, NY, 1977.

[20] R. C. Barr, T. M. Gallie, and M. S. Spach. Automated production of contour maps for electrophysiology. *Computers in Biomedical Research*, 13:142–191, 1980.

[21] G. Baszenski and Larry L. Schumaker. Use of simulated annealing to construct triangular facet surfaces. In Pierre-Jean Laurent, Alain Le Méhauté, and Larry L. Schumaker, editors, *Curves and Surfaces*, pages 27–32, Boston, 1991. Academic Press.

[22] J. O. Bentley, B. W. Wiede, and A. C. Yao. Optimal expected time algorithms for closest point problems. *ACM Transactions on Mathematical Software*, 6:563–580, 1980.

[23] G. Berman and K. D. Fryer. *Introduction to Combinatorics*, pages 230–231. Academic Press, New York, 1972.

[24] M. Bern, H. Edelsbrunner, D. Eppstein, S. Mitchell, and T. S. Tan. Edge-insertion for optimal triangulations. Tech. Report EDC UILU-ENG-92-1702, Univ. Illinois, Urbana, IL, 1992.

[25] M. Bern and D. Eppstein. Mesh generation and optimal triangulation. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 1 of *Lecture Notes Series on Computing*, pages 23–90. World Scientific, Singapore, 1992.

[26] Marshall Bern. Personal Communication.

[27] J. Bernal. Constructing Delaunay triangulations for sets constrained by line segments. Technical Report NIST/TN-1252, Mathematical Analysis Div., National Inst. of Standards and Technology (NEL), Gaithersburg, MD, September 1988.

[28] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.

[29] J.-D. Boissonnat. Shape reconstruction from planar cross-sections. *Comput. Vision Graph. Image Process.*, 44(1):1–29, October 1988.

[30] J.-D. Boissonnat and B. Geiger. Three-dimensional reconstruction of complex shapes based on the Delaunay triangulation. Report 1697, INRIA Sophia-Antipolis, Valbonne, France, April 1992.

[31] Jean-Daniel Boissonnat and Monique Teillaud. An hierarchical representation of objects: The Delaunay tree. In *Proceedings of the Second Annual Symposium on Computational Geometry*, pages 260–268, New York, 1986. ACM Press.

[32] G. Bol. Zur kinematischen Ordnung ebener Jordan-Kurven. *Abh. Math. Sem. Univ. Hamburg*, 11:394–408, 1936.

[33] C. Borgers. Generalized Delaunay triangulations of nonconvex domains. *Computers & Mathematics With Applications*, 20(7):45–49, 1990.

[34] S. W. Bova and G. F. Carey. Mesh generation/refinement using fractal concepts and iterated function systems. *International Journal for Numerical Methods in Engineering*, 33(2):287–305, Jan 1992.

[35] A. Bowyer. Computing Dirichlet tessellations. *The Computer Journal*, 24(2):162–166, 1981.

[36] J. W. Brandt and V. R. Algazi. Continuous skeleton computation by Voronoi diagram. *CVGIP Image Understanding*, 55(3):329–338, 1992.

[37] K. Brassel and D. Reif. A procedure to generate Thiessen polygons. *Geographical Analysis*, 11(3):289–303, 1979.

[38] J. L. Brown. Vertex based data dependent triangulations. *Computer Aided Geometric Design*, 8:239–251, 1991.

[39] C. F. Bryant. Two-dimensional automatic triangular mesh generation. *IEEE Transactions On Magnetics*, MAG-21(6):2547–2550, 1985.

[40] T. D. Bui and V. N. Hanh. Automatic mesh generation for finite-element analysis. *Computing*, 44(4):305–329, 1990.

[41] J. C. Cavendish. Automatic triangulation of arbitrary planar domains for the finite element method. *International Journal for Numerical Methods in Engineering*, 8:679–696, 1974.

[42] Z. J. Cendes, D. N. Shenton, and H. Shahnasser. Magnetic field computation using Delaunay triangulation and complementary finite element methods. *IEEE Transactions on Magnetics*, MAG-19(6), 1983.

[43] Zoltan J. Cendes. Unlocking the magic of Maxwell's equations. *IEEE Spectrum*, 26(4):29–33, Apr 1989.

[44] D. R. Chand and S. S. Kapur. An algorithm for convex polytopes. *J. ACM*, 17:78–86, 1970.

[45] Maw Shang Chang, Nen-Fu Huang, and Chuan-Yi Tang. Optimal algorithm for constructing oriented Voronoi diagrams and geographic neighborhood graphs. *Information Processing Letters*, 35(5):255–260, Aug 1990.

[46] S. Chattopadhyay and P. P. Das. Counting thin and bushy triangulations. *Pattern Recognition Letters*, 12(3):139–144, 1991.

[47] B. Chazelle. A theorem on polygon cutting with applications. In *Proc. 23rd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 339–349, 1982.

[48] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete Comput. Geom.*, 6:485–524, 1991.

[49] B. Chazelle, L. J. Guibas, and D. T. Lee. The power of geometric duality. *BIT*, 25:76–90, 1985.

[50] B. Chazelle and J. Incerpi. Triangulating a polygon by divide and conquer. In *Proceedings of the 21st Allerton Conference on Communications, Control and Computing*, pages 447–456, 1983.

[51] Xiangping Chen and Daoning Ying. Polygon triangulation algorithm as a powerful core processor of plan-1. *Computer Graphics Forum*, 8(3):193–198, Sep 1989.

[52] L. P. Chew. Building Voronoi diagrams for convex polygons in linear expected time. Technical Report PCS-TR90-147, Dept. Math. Comput. Sci., Dartmouth College, Hanover, NH, 1986.

[53] L. P. Chew. There are planar graphs almost as good as the complete graph. *J. Comput. Syst. Sci.*, 39:205–219, 1989.

[54] L. P. Chew. Guaranteed-quality triangular meshes. Technical Report CU-CSD-TR-89-983, Cornell Univ., Ithaca, NY. Dept. of Computer Science., Apr 89.

[55] L. P. Chew and R. L. Drysdale, III. Voronoi diagrams based on convex distance functions. In *Proc. 1st Annu. ACM Sympos. Comput. Geom.*, pages 235–244, 1985.

[56] L. Paul Chew. There is a planar graph almost as good as the complete graph. In *Proceedings of the Second Annual Symposium on Computational Geometry*, pages 169–177, New York, 1986. ACM Press.

[57] L. Paul Chew. Constrained Delaunay triangulations. *Algorithmica*, 4(1):97–108, 1989.

[58] Raju Chithambaram, Renato Barrera, and Kate Beard. Skeletonizing polygons for map generalization. In *Technical Papers - 1991 ACSM-ASPRS Annual Convention*, pages 44–55, Bethesda, MD, USA, 1991. ACSM.

[59] B. K. Choi, H. Y. Shin, Y. I. Yoon, and J. W. Lee. Triangulation of scattered data in 3d space. *Computer-Aided Design*, 20(5):239–248, Jun 1988.

[60] K. Clarkson, R. E. Tarjan, and C. J. Van Wyk. A fast Las Vegas algorithm for triangulating a simple polygon. *Discrete Comput. Geom.*, 4:423–432, 1989.

[61] K. L. Clarkson. Randomized geometric algorithms. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 1 of *Lecture Notes Series on Computing*, pages 117–162. World Scientific, Singapore, 1992.

[62] K. L. Clarkson, R. Cole, and R. E. Tarjan. Randomized parallel algorithms for trapezoidal diagrams. In *Proc. 7th Annu. ACM Sympos. Comput. Geom.*, pages 152–161, 1991.

[63] A. K. Cline and R. J. Renka. A constrained 2-dimensional triangulation and the solution of closest node problems in the presence of barriers. *SIAM J. Numer. Anal.*, 27(5):1305–1321, 1990.

[64] A. K. Cline and R. L. Renka. A storage-efficient method for construction of a Thiessen triangulation. *Rocky Mountain Journal of Mathematics*, 14(1):119–138, 1984.

[65] Y. Correc and E. Chapuis. Fast computation of Delaunay triangulations. *Advances in Engineering Software*, 9(2):77–83, 1987.

[66] H. S. M. Coxeter. *Introduction to Geometry.* John Wiley & Sons, New York, 1961.

[67] I. K. Crain. Monte-Carlo simulation of the random Voronoi polygons: Preliminary results. *Search*, 3:220–221, 1972.

[68] I. K. Crain and R. E. Miles. Monte-Carlo estimates of the distributions of the random polygons determined by random lines in a plane. *Journal of Statistical Computation and Simulation*, 4:293–325, 1976.

[69] Robert G. Cromley and Daniel Grogan. A procedure for identifying and storing a Thiessen diagram within a convex boundary. *Geographical Analysis*, 17(2):167–175, April 1985.

[70] J. R. Davy and P. M. Dew. A note on improving the performance of Delaunay triangulation. In R. A. Earnshaw and B. Wyvill, editors, *New Advances in Computer*

Graphics: Proceedings of Computer Graphics International 89, pages 209–226, Tokyo, 1989. Springer-Verlag.

[71] E. F. Dazevedo and R. B. Simpson. On optimal interpolation triangle incidences. *SIAM J. Sci. Statist. Comput.*, 10(6):1063–1075, 1989.

[72] L. De Floriani, B. Falcidieno, and C. Pienovi. Delaunay-based representation of surfaces defined over arbitrarily shaped domains. *Computer Vision, Graphics, and Image Processing*, 32:127–140, 1985.

[73] L. de Floriani and E. Puppo. An on-line algorithm for constrained Delaunay triangulation. *CVGIP: Graphical Models and Image Processing*, 54(3):290–300, 1992.

[74] Leila De Floriani. Surface representation based on triangular grids. *The Visual Computer*, 3:27–50, 1987.

[75] Leila De Floriani, Bianca Falcidieno, George Nagy, and Caterina Pienovi. On sorting triangles in a Delaunay tessellation. *Algorithmica*, 6:522–532, 1991.

[76] Leila De Floriani and Enrico Puppo. Constrained Delaunay triangulation for multiresolution surface description. In *Proc. Ninth IEEE International Conference on Pattern Recognition*, pages 566–569, Los Alamitos, California, 1988. CS Press.

[77] Frank Dehne and Rolf Klein. "the big sweep": On the power of the wavefront approach to Voronoi diagrams. manuscript, December 1992.

[78] B. Delaunay. Sur la sphère vide. *Bull. Acad. Sci. USSR: Class. Sci. Mat. Nat.*, 7:793–800, 1934.

[79] Hüseyin Demir. Incircles within. *Mathematics Magazine*, 59:77–83, 1986.

[80] N. Adlai A. DePano, Farinaz D. Boudreau, Philip Katner, and Brian Li. Algorithmic paradigms. examples in computational geometry II. In *21st SIGCSE Technical Symposium on Computer Science Education*, pages 186–191, Fort Collins Computer Center, Fort Collins, CO, 1990. ACM.

[81] B. J. Devereux, R. M. Fuller, L. Carter, and R. J. Parsell. Geometric correction of airborne scanner imagery by matching Delaunay triangles. *International Journal Of Remote Sensing*, 11(12):2237–2251, 1990.

[82] P. A. Devijver and M. Dekesel. Insert and delete algorithms for maintaining dynamic Delaunay triangulations. *Pattern Recogn. Lett.*, 1:73–77, 1982.

[83] P. A. Devijver and S. Maybank. Computation of the Delaunay triangulation of a convex polygon under a minimum space complexity constraint. In *Proceedings of the 6th IEEE International Conference on Pattern Recognition*, pages 420–422, 1982.

[84] L. P. Devroye. On the computer generation of random convex hulls. *Comput. Math. Appl.*, 8:1–13, 1982.

[85] Tamal K. Dey, Kokichi Sugihara, and Chandrajit L. Bajaj. Delaunay triangulations in three dimensions with finite precision arithmetic. *Computer Aided Geometric Design*, 9:457–470, 1992.

[86] P. J. Diggle. *Statistical Analysis of Point Patterns.* Academic Press, 1983.

[87] Michael B. Dillencourt. Realizability of Delaunay triangulations. *Information Processing Letters*, 33(6):283–287, Feb 1990.

[88] H. Djidjev and A. Lingas. On computing the Voronoi diagram for restricted planar figures. In *Proc. 2nd Workshop Algorithms Data Struct.*, volume 519 of *Lecture Notes in Computer Science*, pages 54–64. Springer-Verlag, 1991.

[89] D. P. Dobkin. Computational geometry and computer graphics. *Proc. IEEE*, 80(9):1400–1411, September 1992.

[90] D. P. Dobkin, S. J. Friedman, and K. J. Supowit. Delaunay graphs are almost as good as complete graphs. *Discrete Comput. Geom.*, 5:399–407, 1990.

[91] Robert L. (Scot) Drysdale, III. A practical algorithm for computing the Delaunay triangulation for convex distance functions. In *Proc. 1st ACM-SIAM Sympos. Discrete Algorithms*, pages 159–168, 1990.

[92] Rex A. Dwyer. A faster divide-and-conquer algorithm for constructing Delaunay triangulations. *Algorithmica*, 2:137–151, 1987.

[93] Nira Dyn, David Levin, and Samuel Rippa. Data dependent triangulations for piecewise linear interpolation. *IMA Journal of Numerical Analysis*, 10:137–154, 1990.

[94] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, West Germany, 1987.

[95] H. Edelsbrunner. An acyclicity theorem for cell complexes in d dimensions. In *Proc. 5th Annu. ACM Sympos. Comput. Geom.*, pages 145–151, 1989.

[96] H. Edelsbrunner and L. J. Guibas. Topologically sweeping an arrangement. *J. Comput. Syst. Sci.*, 38:165–194, 1989. Corrigendum in 42 (1991), 249–251.

[97] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph.*, 9:66–104, 1990.

[98] H. Edelsbrunner, J. O'Rourke, and R. Seidel. Constructing arrangements of lines with applications. In *Proceedings 24th IEEE Symposium on Foundations of Computer Science*, pages 83–91, 1983.

[99] H. Edelsbrunner and N. R. Shah. Incremental topological flipping works for regular triangulations. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 43–52, 1992.

[100] H. Edelsbrunner and T. S. Tan. A quadratic time algorithm for the minmax length triangulation. In *Proc. 32nd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 414–423, 1991.

[101] H. Edelsbrunner and T. S. Tan. An upper bound for conforming Delaunay triangulations. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 53–62, 1992.

[102] Herbert Edelsbrunner. CS497: Triangulations. Course notes, UIUC, Spring 1991.

[103] Herbert Edelsbrunner, Tiow Seng Tan, and Roman Waupotitsch. $O(N^2 \log N)$ time algorithm for the minmax angle triangulation. *SIAM journal on scientific and statistical computing*, 13(4):994–1008, July 1992.

[104] M. Elbaz and J.-C. Spehner. Construction of Voronoi diagrams in the plane by using maps. *Theoretical Computer Science*, 77(3):331–343, 1990.

[105] M. H. Elfick. Contouring by use of a triangular mesh. *The Cartographic Journal*, 16(1):24–29, 1979.

[106] H. ElGindy and D. Avis. A linear algorithm for computing the visibility polygon from a point. *J. Algorithms*, 2:186–197, 1981.

[107] Hossam ElGindy and Godfried T. Toussaint. On geodesic properties of polygons relevant to linear time triangulation. *Visual Comput.*, 5(1):68–74, 1989.

[108] D. Eppstein. The farthest point Delaunay triangulation minimizes angles. *Comput. Geom. Theory Appl.*, 1:143–148, 1992.

[109] G. Erlebacher and P. R. Eiseman. Adaptive triangular mesh generation. *AIAA Journal*, 25(10):1356–1364, 1987.

[110] Euclid. Elements of geometry. Book III, Proposition 31.

[111] Euclid. Elements of geometry. Book III, A consequence of Proposition 21.

[112] T.-P. Fang and L. A. Piegl. Algorithm for Delaunay triangulation and convex-hull computation using sparse matrix. *Comput.-Aided Design*, 24(8):425–436, August 1992.

[113] T.-P. Fang and L. A. Piegl. Delaunay triangulation using a uniform grid. *IEEE Comput. Graph. Appl.*, 13(3):36–48, May 1993.

[114] Olivier Faugeras. *Three-Dimensional Computer Vision: A Geometric Viewpoint.* MIT Press, Cambridge, MA, 1993.

[115] D. Field. A flexible Delaunay triangulation algorithm. Research Publication GMR-5675, General Motors, 1987.

[116] David A. Field and Thomas W. Nehl. Stitching together tetrahedral meshes. In *Proceedings of the SIAM Regional Conference on Geometric Aspects of Industrial Design*, pages 25–38, Philadelphia, PA, 1992. Soc. for Industrial & Applied Mathematics.

[117] George S. Fishman. *Principles of discrete event simulation.* Wiley, New York, 1978.

[118] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, Reading, MA, 1990.

[119] S. Fortune. Numerical stability of algorithms for 2-d Delaunay triangulations and Voronoi diagrams. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 83–92, 1992.

[120] S. Fortune. Voronoi diagrams and Delaunay triangulations. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 1 of *Lecture Notes Series on Computing*, pages 193–233. World Scientific, Singapore, 1992.

[121] S. J. Fortune. A fast algorithm for polygon containment by translation. In *Proc. 12th Internat. Colloq. Automata Lang. Program.*, volume 194 of *Lecture Notes in Computer Science*, pages 189–198. Springer-Verlag, 1985.

[122] Steven Fortune. Sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2(2):153–174, 1987.

[123] R. J. Fowler and J. J. Little. Automatic extraction of irregular network digital terrain models. *Computer Graphics*, 13(2):199–207, August 1979.

[124] C. O. Frederick, Y. C. Wong, and F. W. Edge. Two-dimensional automatic mesh generation for structural analysis. *International Journal for Numerical Methods in Engineering*, 2:133–144, 1970.

[125] Free Software Foundation. Gnu graphics 0.17. Available by FTP from `qed.rice.edu:/pub/graphics.tar.Z`.

[126] William H. Frey and David A. Field. Mesh relaxation for improving triangulations. In *Proceedings of the SIAM Regional Conference on Geometric Aspects of Industrial Design*, pages 11–24, Philadelphia, PA, USA, 1992. Soc for Industrial & Applied Mathematics Publ.

[127] William H. Frey and David A. Field. Mesh relaxation for improving triangulations. In *Proceedings of the SIAM Regional Conference on Geometric Aspects of Industrial Design*, pages 11–24, Philadelphia, PA, 1992. Soc. for Industrial & Applied Mathematics.

[128] M. Fujiwara. Ein Satz über konvexe geschlossene Kurven. *Sci. Repts. Tôhoku Univ.*, 9:289–294, 1920.

[129] Hidetosi Fukagawa and Dan Pedoe. *Japanese Temple Geometry Problems.* The Charles Babbage Research Centre, Winnipeg, 1989.

[130] K. R. Gabriel and R. R. Sokal. A new statistical approach to geographic variation analysis. *Systematic Zoology*, 18:259–278, 1969.

[131] Luis F. Garcia and Osama A. Mohammed. Automatic finite element grid generation in electromagnetics. In *1988 IEEE Southeastcon, Conference Proceedings*, pages 571–575, New York, NY, USA, 1988. IEEE.

[132] M. R. Garey, D. S. Johnson, and F. P. Preparata. Triangulating a simple polygon. *Information Processing Letters*, 7:175–179, 1978.

[133] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* W. H. Freeman, San Francisco, 1979.

[134] P. L. George, F. Hecht, and M. G. Vallet. Creation of internal points in Voronoi's type method. Control adaptation. *Advances in Engineering Software and Workstations*, 13(5-6):303–312, Sep-Nov 1991.

[135] P. L. George and F. Hermeline. Delaunay's mesh of a convex polyhedron in dimension d. Application to arbitrary polyhedra. *International Journal for Numerical Methods in Engineering*, 33(5):975–995, Apr 1992.

[136] G. Ghione, R. D. Graglia, and C. Rosati. New general-purpose two-dimensional mesh generator for finite elements, generalized finite differences, and moment method applications. *IEEE Transactions on Magnetics*, MAG-24(1):307–310, Jan 1987.

[137] P. D. Gilbert. New results in planar triangulations. Report R-850, Coordinated Sci. Lab., Univ. Illinois, Urbana, IL, 1979.

[138] C. M. Gold, T. D. Charters, and J. Ramsden. Automated contour mapping using triangular element data structures and an interpolant over each irregular triangular domain. *Computer Graphics*, 11(2):170–175, 1977.

[139] S. Goldman. A space efficient greedy triangulation algorithm. *Information Processing Letters*, 31(4):191–196, 1989.

[140] N. A. Golias and T. D. Tsiboukis. Adaptive refinement in 2-d finite element applications. *International Journal of Numerical Modelling: Electronic Networks, Devices and Fields*, 4(2):81–95, Jun 1991.

[141] Harry Gonshor. *An introduction to the theory of surreal numbers*, volume 110 of *London Mathematical Society lecture note series.* Cambridge University Press, Cambridge, 1986.

[142] T. Gonzalez and M. Razzazi. Properties and algorithms for constrained Delaunay triangulations. In *Proc. 3rd Canad. Conf. Comput. Geom.*, pages 114–117, 1991.

[143] Ardeshir Goshtasby. Piecewise linear mapping functions for image registration. *Pattern Recognition*, 19(6):459–466, 1986.

[144] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science.* Addison-Wesley, Reading, MA, 1989.

[145] P. J. Green and R. Sibson. Computing Dirichlet tessellations in the plane. *The Computer Journal*, 21(2):168–173, 1978.

[146] B. Grünbaum. Arrangements and spreads. In *Regional Conf. Ser. Math., Amer. Math. Soc.*, page ??, Providence, RI, 1972.

[147] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4:74–123, 1985.

[148] Leonidas J. Guibas, Donald E. Knuth, and Micha Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. In M. S. Paterson, editor, *Automata, Languages and Programming: 17th International Colloquium*, pages 414–431, Berlin, 1990. Springer-Verlag. LNCS 443.

[149] Leonidas J. Guibas and Jorge Stolfi. Ruler, compass, and computer: The design and analysis of geometric algorithms. Technical Report 37, Digital Equipment Corporation Systems Research Center, 1989.

[150] A. J. Hansen and P. L. Levin. On conforming Delaunay mesh generation. *Adv. Engineering Software*, 14(2):129–135, 1992.

[151] Dianne Hansford. The neutral case for the min-max triangulation. *Computer Aided Geometric Design*, 7:431–438, 1990.

[152] F. Hermeline. Triangulation automatique d'un polyèdre en dimension N. *RAIRO-Mathematical Modelling And Numerical Analysis-Modelisation Mathematique Et Analyse Numerique*, 16(3):211–242, 1982.

[153] S. Hertel and K. Mehlhorn. Fast triangulation of simple polygons. In *Proceedings 1983 Foundations of Computation Theory Conference*, pages 207–218, 1983.

[154] A. L Hinde and R. E. Miles. Monte Carlo estimates of the distributions of the random polygons of the Voronoi tessellation with respect to a Poisson process. *Journal of Statistics and Computer Simulation*, 10:205–223, 1980.

[155] R. Honsberger. *Mathematical Gems III.* The Mathematical Association of America, Washington D.C., 1985.

[156] Yih-ping Huang. Triangular irregular network generation and topographical modelling. *Computers in Industry*, 12:203–213, 1989.

[157] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40:1098–1101, 1952.

[158] F. K. Hwang. An $O(n \log n)$ algorithm for rectilinear minimal spanning trees. *Journal of the ACM*, 26:177–18, 1979.

[159] C. Icking, R. Klein, N.-M. Lê, and L. Ma. Convex distance functions in 3-space are different. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, pages 116–123, 1993.

[160] Hiroshi Inagaki and Kokichi Sugihara. Numerically robust algorithm for constructing Delaunay triangulation. In *Proc. 6th Canad. Conf. Comput. Geom.*, pages 171–176, Saskatoon, Canada, 1994.

[161] B. Joe. Delaunay triangular meshes in convex polygons. *SIAM Journal On Scientific and Statistical Computing*, 7(2):514–539, 1986.

[162] B. Joe. 3-dimensional triangulations from local transformations. *Siam Journal On Scientific And Statistical Computing*, 10(4):718–741, 1989.

[163] B. Joe. Geompack. A software package for the generation of meshes using geometric algorithms. *Advances in Engineering Software and Workstations*, 13(5-6):325–331, Sep-Nov 1991.

[164] B. Joe and R. B. Simpson. Triangular meshes for regions of complicated shape. *International Journal for Numerical Methods in Engineering*, 23:751–778, 1986.

[165] Barry Joe and Cao An Wang. Duality of constrained Voronoi diagrams and Delaunay triangulations. *Algorithmica*, 9(2):149–155, 1993.

[166] R. A. Johnson. *Advanced Euclidean Geometry.* Dover, New York, 1960.

[167] N. L. Jones, S. G. Wright, and D. R. Maidment. Watershed delineation with triangle-based terrain models. *Journal Of Hydraulic Engineering-ASCE*, 116(10):1232–1251, 1990.

[168] Norman L. Jones and Stephen G. Wright. Algorithm for smoothing triangulated surfaces. *Journal of Computing in Civil Engineering*, 5(1):85–102, Jan 1991.

[169] Dz-Mou Jung. An optimal algorithm for constrained Delaunay triangulation. In *Proceedings. Twenty-Sixth Annual Allerton Conference on Communication, Control and Computing*, pages 85–86, Urbana, IL, USA, 1988. Univ. Illinois.

[170] T. C. Kao and D. M. Mount. An algorithm for computing compacted Voronoi diagrams defined by convex distance functions. In *Proc. 3rd Canad. Conf. Comput. Geom.*, pages 104–109, 1991.

[171] T. C. Kao and D. M. Mount. Incremental construction and dynamic maintenance of constrained Delaunay triangulations. In *Proc. 4th Canad. Conf. Comput. Geom.*, pages 170–175, 1992.

[172] Michael Karasick, Derek Lieber, and Lee R. Nackman. Efficient Delaunay triangulation using rational arithmetic. *ACM Transactions on Graphics*, 10(1):71–91, January 1991.

[173] Jyrki Katajainen and Markku Koppinen. Constructing Delaunay triangulations by merging buckets in quadtree order. *Annales Societatis Mathematicae Polonae, Series IV, Fundamenta Informaticae*, 11(3):275–288, 1988.

[174] J. M. Keil and C. A. Gutwin. The Delaunay triangulation closely approximates the complete Euclidean graph. In *Proc. 1st Workshop Algorithms Data Struct.*, volume 382 of *Lecture Notes in Computer Science*, pages 47–56. Springer-Verlag, 1989.

[175] J. P. Kermode and D. Weaire. 2D-Froth—a program for the investigation of 2-dimensional froths. *Computer Physics Communications*, 60(1):75–109, 1990.

[176] D. G. Kirkpatrick. Efficient computation of continuous skeletons. In *Proc. 20th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 18–27, 1979.

[177] D. G. Kirkpatrick. On the absence of local characterizations of minimum weight triangulations. In *Abstracts 1st Canad. Conf. Comput. Geom.*, page 16, 1989.

[178] D. G. Kirkpatrick, M. M. Klawe, and R. E. Tarjan. Polygon triangulation in $O(n \log\log n)$ time with simple data structures. In *Proc. 6th Annu. ACM Sympos. Comput. Geom.*, pages 34–43, 1990.

[179] R. Klein. *Concrete and Abstract Voronoi Diagrams*, volume 400 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[180] R. Klein and A. Lingas. Manhattonian proximity in a simple polygon. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 312–319, 1992.

[181] R. Klein and A. Lingas. A linear-time randomized algorithm for the bounded Voronoi diagram of a simple polygon. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, pages 124–132, 1993.

[182] R. Klein and A. Lingas. A note on generalizations of Chew's algorithm for the Voronoi diagram of a convex polygon. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 370–374, Waterloo, Canada, 1993.

[183] C. Kleinstreuer and J. T. Holdeman. A triangular finite element mesh generation for fluid dynamic systems of arbitrary geometry. *International Journal for Numerical Methods in Engineering*, 15:1325–1334, 1980.

[184] G. T. Klincsek. Minimal triangulations of polygonal domains. *Annals of Discrete Mathematics*, 9:121–123, 1980.

[185] D. E. Knuth. *The Art of Computer Programming: Vol 3, Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.

[186] D. E. Knuth. *The Art of Computer Programming: Vol 1, Fundamental Algorithms*, pages 399–401. Addison-Wesley, Reading, Mass., 1975.

[187] Horst Kramer. A characterization of boundaries of smooth strictly convex plane sets. *L'Analyse Numérique et la Théorie de l'approximation*, 7(1):61–65, 1978.

[188] Richard A. Kronmal and Arthur V. Peterson. On the alias method for generating random variables from a discrete distribution. *Amer. Stat.*, 33:214–218, 1979.

[189] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 22:469–476, 1975.

[190] Tim Lambert, Peter Lindsay, and Ken Robinson. Using Miranda as a first programming language. *Journal of Functional Programming*, 3(1):5–34, 1993.

[191] Leslie Lamport. LaTeX: *A Document Preparation System.* Addison-Wesley, Reading, Massachusett, 1985.

[192] C. L. Lawson. Generation of a triangular grid with application to contour plotting. Technical Memorandum 299, California Institute of Technology Jet Propulsion Laboratory, 1972.

[193] C. L. Lawson. Transforming triangulations. *Discrete Mathematics*, 3:365–372, 1972.

[194] C. L. Lawson. Software for C^1 surface interpolation. In J. R. Rice, editor, *Math. Software III*, pages 161–194, New York, NY, 1977. Academic Press.

[195] Steven R. Lay. *Convex Sets and their Applications.* John Wiley & Sons, New York, 1982.

[196] N.-M. Lê. On general properties of strictly convex smooth distance functions in R^d. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 375–380, Waterloo, Canada, 1993.

[197] E. Le Bras-Mehlman, M. Schmitt, O. D. Faugeras, and J. D. Boissonnat. How the Delaunay triangulation can be used for representing stereo data. In *Second International Conference on Computer Vision*, pages 54–63, New York, NY, USA, 1988. IEEE.

[198] G. Le Caër and J. S. Ho. The Voronoi tessellation generated from eigenvalues of complex random matrices. *Journal Of Physics A-Mathematical And General*, 23(14):3279–3295, 1990.

[199] G. Leach. Improving worst-case optimal Delaunay triangulation algorithms. In *Proc. 4th Canad. Conf. Comput. Geom.*, pages 340–346, 1992.

[200] D. T. Lee. Two-dimensional Voronoi diagrams in the l_p metric. *Journal of the ACM*, 27:604–618, 1980.

[201] D. T. Lee and A. K. Lin. Generalized Delaunay triangulation for planar graphs. *Discrete and Computational Geometry*, pages 201–217, 1986.

[202] D. T. Lee and B. J. Schachter. Two algorithms for constructing the Delaunay triangulation. *International journal of Computer and Information Sciences*, 9:219–242, 1980.

[203] D. T. Lee and C. K. Wong. Voronoi diagrams in l_1 (l_∞) metrics with two-dimensional storage applications. *SIAM Journal on Computing*, 9(1):200–211, 1980.

[204] Christos Levcopoulus and Andrzej Lingas. Fast algorithms for greedy triangulations. *BIT*, 32:280–296, 1992.

[205] D. Leven and M. Sharir. Planning a purely translational motion for a convex object in two-dimensional space using generalized Voronoi diagrams. *Discrete Comput. Geom.*, 2:9–31, 1987.

[206] Peter L. Levin and James R. Hoburg. Donor cell-finite element descriptions of wire-duct precipitator fields, charges and efficiencies. *IEEE Transactions on Industry Applications*, 26:662–670, 1990.

[207] B. A. Lewis and J. S. Robinson. Triangulation of planar regions with applications. *Comput. J.*, 21:324–332, 1978.

[208] Z. M. Lin and Hung C. Lin. A new placement algorithm for custom chip design. In *1990 IEEE International Symposium on Circuits and Systems Part 3 (of 4)*, pages 1672–1675, IEEE Service Center, Piscataway, NJ, USA, 1990. IEEE.

[209] P. A. Lindholm. Automatic triangle mesh generation on surfaces of polyhedra. *IEEE Transactions on Magnetics*, MAG-19(6):2539–2542, 1983.

[210] A. Lingas. A space efficient algorithm for the greedy triangulation. In *Proc. 13th IFIP Conf. System Modelling and Optimization*, volume 113 of *Lecture Notes in Control and Information Science*, pages 359–364. Springer-Verlag, 1988.

[211] A. Lingas. Voronoi diagrams with barriers and the shortest diagonal problem. *Inform. Process. Lett.*, 32:191–198, 1989.

[212] M. A. Linton, P. R. Calder, and J. M. Vlissides. Composing user interface with InterViews. *IEEE Computer*, 22(2):8–22, February 1989.

[213] Dani Lischinski. Incremental delaunay triangulation. In Paul Heckbert, editor, *Graphics Gems IV*, pages 47–59. Academic Press, Boston, 1994.

[214] F. F. Little and J. L. Schwing. Automatic generation of triangulations. cited in [19].

[215] E. L. Lloyd. On triangulations of a set of points in the plane. In *IEEE 18th Annual Symposium on the Foundations of Computer Science*, pages 228–240, 1977.

[216] S. H. Lo. Delaunay triangulation of non-convex planar domains. *International Journal for Numerical Methods in Engineering*, 28(11):2695–2707, Nov 1989.

[217] S. H. Lo. Volume discretization into tetrahedra. I. verification and orientation of boundary surfaces. *Computers and Structures*, 39(5):493–500, 1991.

[218] S. H. Lo. Generation of high-quality gradation finite element mesh. *Engineering Fracture Mechanics*, 41(2):191–202, Jan 1992.

[219] Michael K. Loze and R. Saunders. Two simple algorithms for constructing a two-dimensional constrained Delaunay triangulation. *Applied Numerical Mathematics*, 11:403–418, 1993.

[220] Yizhi Lu and Wayne Wei-Ming Dai. A numerical stable algorithm for constructing constained Delaunay triangulation and application to multichip module layout. In *China 1991 International Conference on Circuits and Systems, June 1991, Shenzhen China*, pages 644–647, 1991.

[221] G. Macedonio and M. T. Pareschi. An algorithm for the triangulation of arbitrarily distributed points: Applications to volume estimate and terrain fitting. *Computers & Geosciences*, 17(7):859–874, 1991.

[222] E. R. Magnus, C. C. Joyce, and W. D. Scott. A spiral procedure for selecting a triangular grid from random data. *Journal for Applied Mathematics and Physics (ZAMP)*, 34:231–235, March 1983.

[223] G. K. Manacher and A. L. Zobrist. Probablistic methods with heaps for fast average-case greedy algorithms. In *Advances in Computing Research Vol 1*, pages 261–278. JAI Press, 1983.

[224] A. Maus. Delaunay triangulation and the convex hull of n points in expected linear time. *BIT*, 24:151–163, 1984.

[225] D. J. Mavriplis. Adaptive mesh generation for viscous flows using Delaunay triangulation. *Journal Of Computational Physics*, 90(2):271–291, 1990.

[226] Jerrold H. May and Robert L. Smith. Random polytopes: Their definition, generation and aggregate properties. *Mathematical Programming*, 24:39–54, 1982.

[227] M. J. McCullagh and C. G. Ross. Delaunay triangulation of a random data set for isarithmic mapping. *The Cartographic Journal*, 17(2):93–99, 1980.

[228] David G. McKenna. The inward spiral method: An improved TIN generation technique and data structure for land planning applications. In *Proceedings, AUTO-CARTO 8*, pages 670–679, Baltimore, MD, March 29–April 3, 1987. Eighth International Symposium on Computer-Assisted Cartography.

[229] D. H. McLain. Two dimensional interpolation from random data. *The Computer Journal*, 19(2):178–191, 1976.

[230] G. Meisters. Polygons have ears. *Amer. Math. Monthly*, 82:648–651, 1975.

[231] David Meyers, Shelley Skinner, and Kenneth Sloan. Surfaces from contours. *ACM Transactions on Graphics*, 11(3):228–258, July 1992.

[232] R. E. Miles. On the homogeneous planar Poisson point process. *Mathematical Biosciences*, 6:85–127, 1970.

[233] A. Mirante and N. Weingarten. The radial sweep algorithm for constructing triangulated irregular networks. *IEEE Computer Graphics and Applications*, pages 11–21, May 1982.

[234] Osama A. Mohammed and Luis F. Garcia. Optimum finite element automatic grid generator for electromagnetic field computations. *IEEE Transactions on Magnetics*, MAG-24(6):3177–3179, Nov 1988.

[235] I. G. Moore. Automatic contouring of geological data. In *APCOM 77, 15th International Symposium on the Application of Computers and Operations Research in the Mineral Industries*, pages 209–220. Australasian Institute of Mining and Metallurgy, 1977.

[236] J-M. Moreau. Hierarchical Delaunay triangulation. In *Proc. 6th Canad. Conf. Comput. Geom.*, pages 165–170, Saskatoon, Canada, 1994.

[237] J.-M. Moreau and P. Volino. Constrained Delaunay triangulation revisited. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 340–345, Waterloo, Canada, 1993.

[238] Mervin M. Muller. A note on a method for generating points uniformly on n-dimensional spheres. *Commun. ACM*, 2(4):19–20, 1959.

[239] L. R. Nackman and V. Srinivasan. Point placement for Delaunay triangulation of polygonal domains. In *Proc. 3rd Canad. Conf. Comput. Geom.*, pages 37–40, 1991.

[240] E. J. Nadler. *Piecewise Linear Approximation on Triangulations of Planar Regions.* PhD. thesis, Brown University, Providence, RI, May 1985.

[241] J. M. Nelson. A triangulation algorithm for arbitrary planar domains. *Applied Mathematical Modelling*, 2:151–159, September 1978.

[242] G. M. Nielson. An example with a local minimum for the minmax ordering of triangulations. Computer Science Department Technical Report TR-87-014, Arizona State University, Tempe, Arizona, 1987.

[243] G. M. Nielson and R. Franke. Surface construction based upon triangulations. In R. Barnhill and W. Boehm, editors, *Surfaces in Computer-Aided Geometric Design*, pages 163–177. North Holland, 1983.

[244] J. Nievergelt, P. Schorn, C. Amman, A. Brüngger, and M. De Lorezi. XYZ: A project in experimental geometric computation. In *Computational Geometry: Methods, Algorithms and Applications. Proc. CG'91, International Workshop on Comp. Geometry, Bern, March 1991*, volume 553 of *Springer LNCS*, pages 171–186, 1991.

[245] C O'Dunlaing and C. K. Yap. A retraction method for planning the motion of a disc. *Journal of Algorithms*, 6:104–111, 1985.

[246] T. Ohya, M. Iri, and K. Murota. A fast Voronoi diagram algorithm with quaternary tree bucketing. *Information Processing Letters*, 18:227–231, 1984.

[247] T. Ohya, M. Iri, and K. Murota. Improvements of the incremental method for the Voronoi diagram with computational comparison of various algorithms. *Journal of the Operations Research Society of Japan*, 27(4):306–336, 1984.

[248] Atsuyuki Okabe, Barry Boots, and Kokichi Sugihara. *Spatial Tessellations : Concepts and Applications of Voronoi Diagrams.* Wiley series in probability and mathematical statistics. Wiley & Sons, Chichester, 1992.

[249] Amr A. Oloufa. Triangulation applications in volume calculation. *Journal of Computing in Civil Engineering*, 5(1):103–121, Jan 1991.

[250] Stephen M. Omohundro. The Delaunay triangulation and function learning. Technical Report TR-90-001, International Computer Science Institute, Berkeley, CA, 1990.

[251] J. O'Rourke. Computational geometry column 22. *Internat. J. Comput. Geom. Appl.*, 4:119–122, 1994. Also in SIGACT News 25:1 (1994), 31–33.

[252] A. Oxley. Surface fitting by triangulation. *The Computer Journal*, 28(3):335–339, 1985.

[253] Oscar Palacios-Velez and Baltasar Cuevas Renaud. Dynamic hierarchical subdivision algorithm for computing Delaunay triangulations and other closest-point problems. *ACM Transactions on Mathematical Software*, 16(3):275–292, Sep 1990.

[254] J. Penman and M. D. Grieve. Self-adaptive mesh generation technique for the finite-element method. *IEE Proceedings. A, Physical science, measurement and instrumentation, management and education, reviews*, 134(8):634–650, 1987.

[255] T. K. Peucker, R. J. Fowler, and J. J. Little. The triangulated irregular network. In *Proceedings ASP-ACSM Symposium on Digital Terrain Models*, pages 516–540, 1978.

[256] Les A. Piegl and Arnaud M. Richard. Algorithm and data structure for triangulating multiply connected polygonal domains. *Computers & Graphics*, 17(5):563–574, 1993.

[257] J. J. Pisano, P. K. Enge, and P. L. Levin. Using GPS to calibrate Loran-C. *IEEE Transactions On Aerospace And Electronic Systems*, 27(4):696–708, 1991.

[258] David A. Plaisted and Jiarong Hong. A heuristic triangulation algorithm. *Journal of Algorithms*, 8:405–437, 1987.

[259] PostScript Developer Tools & Strategy Group. Encapsulated POSTSCRIPT files specification. Mountain View, CA, 1989.

[260] M. Pourazady and M. Radhakrishnan. Optimization of a triangular mesh. *Computers and Structures*, 40(3):795–804, 1991.

[261] P. L. Powar. Minimal roughness property of the Delaunay triangulation: a shorter approach. *Computer Aided Geometric Design*, 9:491–494, 1992.

[262] P. L. Powar. The neutral case for the min-max angle criterion: a generalized concept. *Computer Aided Geometric Design*, 9:413–418, 1992.

[263] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, 1985.

[264] P. W. Purdom and C. A. Brown. *The Analysis of Algorithms*, page 307. Holt, Rinehart and Winston, New York, 1985.

[265] Ewald Quak and Larry L. Schumaker. Cubic spline fitting using data dependent triangulations. *Computer Aided Geometric Design*, 7(1-4):293–301, Jun 1990.

[266] V. T. Rajan. Optimality of the Delaunay triangulation in R^d. In *Proc. 7th Annu. ACM Sympos. Comput. Geom.*, pages 357–363, 1991.

[267] P. N. Rathie. A census of simple planar triangulations. *J. Comb. Theory B*, 16:134–138, 1974.

[268] H. Raynaud. Sur l'enveloppe convex nuages des points aléatoires dans R^n. *J. Appl. Prob.*, 7:35–48, 1970.

[269] K. Reichert, J. Skoczylas, and T. Tarnhuvud. Automatic mesh generation based on expert-system-methods. *IEEE Transactions On Magnetics*, 27(5):4197–4200, 1991.

[270] A. Rényi and R. Sulanke. Über die konvexe Hülle von n zufällig gerwählten Punkten I. *Z. Wahrsch. Verw. Gebiete*, 2:75–84, 1963.

[271] D. Rhynsburger. Analytic delineation of Thiessen polygons. *Geographical Analysis*, 5(2):133–144, April 1973.

[272] Samuel Rippa. Minimal roughness property of the Delaunay triangulation. *Computer Aided Geometric Design*, 7:489–497, 1990.

[273] Samuel Rippa. Long and thin triangles can be good for linear interpolation. *SIAM Journal on Numerical Analysis*, 29(1):257–270, Feb 1992.

[274] C. A. Rogers. *Packing and Covering.* Cambridge University Press, 1964.

[275] M. Roussille and P. Dufour. Generation of convex polygons with individual angular constraints. *Information Processing Letters*, 24(3):159–164, February 1987.

[276] A. Saalfeld. Delaunay edge refinements. In *Proc. 3rd Canad. Conf. Comput. Geom.*, pages 33–36, 1991.

[277] Luis A. Santalo. *Integral Geometry and Geometric Probability.* Addison-Wesley, 1976.

[278] Nickolas Sapidis and Renato Perucchio. Delaunay triangulation of arbitrarily shaped planar domains. *Computer Aided Geometric Design*, 8(6):421–437, Dec 1991.

[279] V. Sarin and S. Kapoor. Algorithms for relative neighbourhood graphs and Voronoi diagrams in simple polygons. In *Proc. 4th Canad. Conf. Comput. Geom.*, pages 292–298, 1992.

[280] D. G. Sawker, G. R. Shevare, and S. P. Koruthu. Contour plotting for scattered data. *Computers and Graphics*, 11(2):101–104, 1987.

[281] Sanjeev Saxena, P. C. P. Bhatt, and V. C. Prasad. Efficient VLSI parallel algorithm for Delaunay triangulation on orthogonal tree network in two and three dimensions. *IEEE Transactions on Computers*, 39(3):400–404, Mar 1990.

[282] Lori Scarlatos and Theo Pavlidis. Adaptive hierarchical triangulation. In *Technical Papers 1991 ACSM-ASPRS Annual Convention. Volume 6 Auto-Carto 10*, pages 234–246, 1991.

[283] Lori Scarlatos and Theo Pavlidis. Hierarchical triangulation using cartographics coherence. *CVGIP: Graphical Models and Image Processing*, 54(2):147–161, March 1992.

[284] Lori L. Scarlatos and Theo Pavlidis. Optimizing triangulations by curvature equalization. In *Visualization '92*, pages 333–339. IEEE Computer Society, 1992.

[285] B. F. Schaudt and R. L. Drysdale. Higher-dimensional Voronoi diagrams for convex distance functions. In *Proc. 4th Canad. Conf. Comput. Geom.*, pages 274–279, 1992.

[286] Peter Schorn et al. XYZ Geobench v4.4.5. Available by FTP from `neptune.inf.ethz.ch:/XYZ`, 1994.

[287] W. J. Schroeder and M. S. Shephard. Geometry-based fully-automatic mesh generation and the Delaunay triangulation. *International Journal For Numerical Methods In Engineering*, 26(11):2503–2515, 1988.

[288] L. L. Schumaker. Triangulation methods. In C. K. Chui, L. L. Schumaker, and F. I. Utreras, editors, *Topics in Multivariate Approximation*, pages 219–232, New York, 1987. Academic Press.

[289] Larry L. Schumaker. Triangulations in CAGD. *IEEE Comput. Graph. Appl.*, 13:47–52, January 1993.

[290] R. Seidel. A method for proving lower bounds for certain geometric problems. In G. T. Toussaint, editor, *Computational Geometry*, pages 319–334. North-Holland, Amsterdam, Netherlands, 1985.

[291] R. Seidel. Constrained Delaunay triangulations and Voronoi diagrams. In *Rep. 260*, pages 178–191. IIG-TU, Graz, Austria, 1988.

[292] R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Comput. Geom. Theory Appl.*, 1:51–64, 1991.

[293] R. Seidel. Backwards analysis of randomized geometric algorithms. Technical Report TR-92-014, International Computer Science Institute, Berkeley, CA, 1992.

[294] Hamid Shahnasser, Ward Morgan, and A. Raghuram. Dynamic date structure suitable for adaptive mesh refinement in finite element method. *Finite Elements in Analysis and Design*, 4(3):237–247, Nov 1988.

[295] M. I. Shamos and D. Hoey. Closest point problems. In *Proceedings of the 16th Annual Symposium on the Foundations of Computer Science*, pages 151–162. IEEE, 1975.

[296] M. Shapiro. A note on Lee and Schachter's algorithm for Delaunay triangulation. *International journal of Computer and Information Sciences*, 10(6):413–418, 1981.

[297] G. M. Shute, L. L. Deneen, and C. D. Thomborson (a.k.a. Thompson). An $O(N\,Log\,N)$ plane-sweep algorithm for L_1 and L_∞ Delaunay triangulations. *Algorithmica*, 6:207–221, 1991.

[298] R. Sibson. Locally equiangular triangulations. *The Computer Journal*, 21(3):243–245, 1978.

[299] Sven Skyum. A sweepline algorithm for generalized delaunay triangulations. Technical Report DAIMIPB–373, Computer Science Department, Aarhus University, November 1991.

[300] S. W. Sloan. A fast algorithm for constructing Delaunay triangulations in the plane. *Advances in Engineering Software*, 9(1):34–55, January 1987.

[301] S. W. Sloan. A fast algorithm for generating constrained Delaunay triangulations. Research Report 065.07.1991, The University of Newcastle Department of Civil Engineering and Surveying, New South Wales, 1991.

[302] S. W. Sloan and G. T. Houlsby. An implementation of Watson's algorithm for computing 2-dimensional Delaunay triangulations. *Advances in Engineering Software*, 6(4):192–197, 1984.

[303] Kokichi Sugihara and Masao Iri. Construction of the Voronoi diagram for "one million" generators in single-precision arithmetic. *Proceedings of the IEEE*, 80(9):1471–1484, 1992.

[304] J. Suhara and J. Fukuda. Automatic mesh generation for finite element analysis. In J. T. Oden, R. W. Clough, and Y. Yamamoto, editors, *Advances in Computational Methods in Structural Mechanics and Design*, pages 607–624. UAU Press, Huntsville, Alabama, 1972.

[305] M. Tanemura, T. Ogawa, and W. Ogita. A new algorithm for three-dimensional Voronoi tessellation. *Journal of Computational Physics*, 51:191–207, 1983.

[306] R. E. Tarjan and C. J. Van Wyk. An $O(n \log \log n)$-time algorithm for triangulating a simple polygon. *SIAM J. Comput.*, 17:143–178, 1988. Erratum in 17 (1988), 106.

[307] Albin Tarvydas. Terrain approximation by triangular facets. In *Technical Papers of the 44th Annual Meeting of the American Congress on Surveying and Mapping.*, pages 524–532, Falls Church, VA, US, 1984. American Congress on Surveying and Mapping.

[308] John C. Tipper. Straightforward iterative algorithm for the planar Voronoi diagram. *Information Processing Letters*, 34(3):155–160, Apr 1990.

[309] John C. Tipper. FORTRAN programs to construct the planar Voronoi diagram. *Computers & Geosciences*, 17(5):597–632, 1991.

[310] G. Toussaint. Efficient triangulation of simple polygons. *Visual Comput.*, 7:280–295, 1991.

[311] G. T. Toussaint. Pattern recognition and geometrical complexity. In *Proceedings 5th International Conference on Pattern Recognition*, pages 1324–1347, 1980.

[312] G. T. Toussaint. The relative neighbourhood graph of a finite planar set. *Pattern Recognition*, 12:261–268, 1980.

[313] G. T. Toussaint. A new linear algorithm for triangulating monotone polygons. Technical Report SOCS 83.9, McGill University, 1983.

[314] Victor J. D. Tsai. Delaunay trianagulation in TIN creation: An overview and a linear-time algorithm. *Int. J. Geographical Information Systems*, 7(6):501–524, 1993.

[315] David Turner. An overview of Miranda. *SIGPLAN Notices*, 21(12), 1986.

[316] W. T. Tutte. A census of planar triangulations. *Canadian J. Math.*, 14:21–38, 1962.

[317] Frederick A. Valentine. *Convex Sets.* McGraw-Hill, New York, 1964.

[318] G. M. Voronoi. Nouvelles applications des paramètres continus à la théorie des formes quadratiques. deuxième Mémoire: Recherches sur les parallélloèdres primitifs. *J. Reine Angew. Math.*, 134:198–287, 1908.

[319] K. Wagner. Bemerkungen zum vierfarbenproblem. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, 46:26–32, 1936.

[320] C. A. Wang and L. Schubert. An optimal algorithm for constructing the Delaunay triangulation of a set of line segments. In *Proc. 3rd Annu. ACM Sympos. Comput. Geom.*, pages 223–232, 1987.

[321] C. A. Wang and Y. H. Tsin. Efficiently updating constrained Delaunay triangulations. In *Proc. 4th Canad. Conf. Comput. Geom.*, pages 176–181, 1992.

[322] Cao An Wang. Efficiently updating the constrained Delaunay triangulation. *BIT*, 33:238–252, 1993.

[323] Cao An Wang. An optimal algorithm for greedy triangulation of a set of points. In *Proc. 6th Canad. Conf. Comput. Geom.*, pages 332–338, Saskatoon, Canada, 1994.

[324] D. Ward. Triangular tessellation—a new approach to forest inventory. *Forest Ecology And Management*, 44(2-4):285–290, 1991.

[325] Glen Y. Watabayishi and J. A. Galt. An optimized triangular mesh system from random points. In J Häuser and C. Taylor, editors, *Numerical Grid Generation in Computational Fluid Dynamics*, pages 437–448, Swansea, U.K., 1986. Pineridge Press.

[326] D. F. Watson. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. *The Computer Journal*, 24:167–172, 1981.

[327] D. F. Watson. Automatic contouring of raw data. *Computers and Geosciences*, 8(1):97–101, 1982.

[328] D. F. Watson and C. M. Philip. Survey: Systematic triangulations. *Computer Vision, Graphics, and Image Processing*, 26:217–223, 1984.

[329] David F. Watson. *Contouring: A Guide to the Analysis and Display of Spatial Data.* Pergamon, Oxford, 1992.

[330] N. P. Weatherill. Integrity of geometrical boundaries in the two-dimensional Delaunay triangulation. *Communications in Applied Numerical Methods*, 6(2):101–109, Feb 1990.

[331] David Wells. *The Penguin Dictionary of Curious and Interesting Geometry.* Penguin, London, 1991.

[332] Marvin S. White Jr. and Patricia Griffin. Piecewise linear rubber-sheet map transformation. *American Cartographer*, 12(2):123–131, Oct 1985.

[333] P Widmayer, F. Wu, and C. K. Wong. On some distance problems in fixed orientations. *SIAM Journal on Computing*, 16:728–746, 1987.

[334] Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer.* Addison-Wesley, Redwood City, 1988.

[335] T. C. Woo and S. Y. Shin. A linear time algorithm for triangulating a point visible polygon. *ACM Transactions on Graphics*, 4(1):60–70, 1985.

[336] F. Yamaguchi and T. Tokieda. A unified algorithm for Boolean shape operations. *IEEE Computer Graphics and Applications*, 4:24–27, June 1984.

[337] F. Yamaguchi and T. Tokieda. A solid modeller with a 4×4 determinant processor. *IEEE Computer Graphics and Applications*, pages 51–59, April 1985.

[338] T. M. Yeung. Generalization of Delaunay triangulation. M.Sc. thesis, Department of EE/CS, Northwestern University, May 1980.

[339] M. M. F. Yuen, S. T. Tan, and K. Y. Hung. A hierarchical approach to automatic finite-element mesh generation. *International Journal For Numerical Methods In Engineering*, 32(3):501–525, 1991.

[340] Jian-Ming Zhou, Ke-Ran Shao, Ke-Ding Zhou, and Qiong-Hua Zhan. Computing constrained triangulation and Delaunay triangulation: A new algorithm. *IEEE Transactions on Magnetics*, 26(2):694–697, Mar 1990.

[341] O. C. Zienkiewicz and R. L. Taylor. *The Finite Element Method.* McGraw-Hill, London, 1989.

Index of Definitions

Colophon

This thesis was produced with a small modification of the `suthesis` style for LaTeX.

The figures were produced by a variety of means:

- LaTeXpicture environment [191], extended by me to use PostScript[1] [3] for the lines and circles, so that all circle sizes and line orientations were possible (*e.g.* figure 3.7).
- Graphs were produced by a plotting program that I wrote which could produce a LaTeXpicture environment (*e.g.* figure 5.5).
- The program `idraw` which comes as part of the InterViews toolkit [212] (*e.g.* figure 3.2).
- Miranda[2] [315] programs using a graphics package I wrote [190] to produce encapsulated PostScript [259] (*e.g.* figure 2.30).
- A combination of writing a Miranda program to produce a picture and editing the resulting picture with `idraw`. I added UNIX[3] plotfile output to my Miranda graphics environment and then used `plot2ps` from the Gnu graphics system [125] to produce an `idraw` file (*e.g.* figure 4.10).
- Plotting from Mathematica [334], conversion to `idraw` format and then using `idraw` to edit the plot (*e.g.* figure 3.11).

[1] PostScript is a trademark of Adobe Systems Incorporated.
[2] Miranda is a trademark of Research Software Ltd.
[3] UNIX is a trademark of AT&T.