Geometric Algorithms for Geographic Information Systems

Geometrische Algoritmen voor Geografische Informatiesystemen
(met een samenvatting in het Nederlands)

PROEFSCHRIFT

ter verkrijging van de graad van
doctor aan de Universiteit Utrecht
op gezag van de Rector Magnificus, Prof. Dr. H.O. Voorma
ingevolge het besluit van het College voor Promoties
in het openbaar te verdedigen
op maandag 31 mei 1999 des middags te 12:45 uur

door

René Willibrordus van Oostrum

geboren op 7 november 1965,
te Utrecht

promotor:	Prof. Dr. M.H. Overmars Faculteit Wiskunde en Informatica
co-promotor:	Dr. M.J. van Kreveld Faculteit Wiskunde en Informatica

ISBN 90-393-2049-7
This research was partially supported by the ESPRIT IV LTR Project No. 21957 (CGAL).

Contents

CHAPTER 1

Introduction

A geographic information system (GIS) is a software package for storing geographic data and performing complex operations on the data. Examples are the reporting of all land parcels that will be flooded when a certain river rises above some level, or analyzing the costs, benefits, and risks involved with the development of industrial activities at some place. A substantial part of all activities performed by a GIS involves computing with the geometry of the data, such as location, shape, proximity, and spatial distribution. The amount of data stored in a GIS is usually very large, and it calls for efficient methods to store, manipulate, analyze, and display such amounts of data. This makes the field of GIS an interesting source of problems to work on for computational geometers.

In this thesis we give new geometric algorithms to solve four selected GIS problems. This introduction provides the necessary background, overview, and definitions to appreciate the following chapters of this thesis.

Section 1.1 introduces the field of geography, the phenomena it studies, the kind of data that is used to model these phenomena, and the types of maps that are available for displaying the data. Section 1.2 introduces GIS, its functionality, and the data structures used to deal with geographic data. Readers with a geographic or GIS background may want to skip or skim the first two sections.

Section 1.3 introduces computational geometry and the basic data structures and techniques from this field that are used extensively in the four following chapters: the doubly-connected edge list, plane sweep, a number of search structures, and Voronoi diagrams. This section may be skipped by computational geometers and others familiar with these structures and techniques.

Section 1.4 introduces the four problems that are studied in Chapters 2–5:

- *Subdivision traversal*: we give a new method to traverse planar subdivisions without using mark bits or a stack.

- *Contour trees and seed sets*: we give a new algorithm for generating a *contour tree* for d-dimensional meshes, and use it to determine a *seed set* of minimum size that can be used for isosurface generation. This is the first algorithm that guarantees a seed set of minimum size. Its running time is quadratic in the input size, which is not fast enough for many practical situations. Therefore, we also give a faster algorithm that gives small (although not minimal) seed sets.

- *Settlement selection*: we give a number of new models for the *settlement selection problem*. When settlements, such as cities, have to be displayed on a map, displaying all of them may clutter the map, depending on the map scale. Choices have to be made which settlements are selected, and which ones are omitted. Compared to existing selection methods, our methods have a number of favorable properties.

- *Facility location*: we give the first algorithm for computing the *furthest-site Voronoi diagram* on a polyhedral terrain, and show that its running time is near-optimal. We use the furthest-site Voronoi diagram to solve the facility location problem: the determination of the point on the terrain that minimizes the maximal distance to a given set of sites on the terrain.

1.1 Geography: preliminaries

1.1.1 Geography and related disciplines

Geography is the science that describes and analyzes phenomena and processes on the surface of the earth and the relations between them. We can distinguish between physical geography and human geography, both of which are broad fields of science that can be subdivided further into several sub-fields [91].

Physical geography studies the physical state of the earth surface and changes of this state over time as a result of natural processes, such as formation and deformation of mountains, and changes in the flow of a river. Subfields include geomorphology, hydrology, and meteorology [109].

Human geography focuses on the spatial aspects of human activities, and the development and consequences of these activities in relation to geographic location. A human geographer may for example investigate why people in a certain region can expect to live longer than people in some other region. Subfields of human geography include political geography, economic geography, and urban geography [60, 81].

Cartography is one of the spatial sciences related to geography. It doesn't deal with describing and analyzing, but with issues involved in the design and reproduction of maps. Geographic information can be displayed in many different ways, and decisions have to be made about what information to display, scale, projection, coordinate system, the use of colors, and the size and location of text. Other important matters are abstraction, classification, and generalization [34, 63, 90].

Geographers and cartographers are not the only professionals dealing with geographic information. For instance, when a new railroad is planned, decision makers need information about impact on the existing landscape, the type of soil, and the costs of the various possible locations. A biologist may be interested in the geographic distribution of species, or the statistical relations between the presence of a species in certain areas and the height above sea level of those areas.

1.1.2 Geographic data

Geographic objects can have many properties, but from the geographer's point of view, one of these properties plays a special role: the location of the object [55, p. 19 ff.]. Coordinate data is referred to as spatial or geo-referenced data; the other properties of the object are non-spatial or attribute data. Locations are usually specified in a two-dimensional coordinate system [127, p. 182]. In some cases that is not enough; for instance, computer models of the atmosphere as used by meteorologists, contain data about temperature, air pressure and wind speed at various altitudes, so three-dimensional coordinates are needed there [88]. The same holds for geological data in different subsurface layers. The geographic objects themselves, whether specified in two or three dimensions, can be zero-dimensional (points), one-dimensional (lines and line-segments), two-dimensional (area), or three-dimensional (volumes; only in the case of a three-dimensional coordinate system) [67, 90].

Apart from discriminating between spatial and non-spatial data, there is an other important classification of the data stored in a GIS or any other information system. We can distinguish between categorial data, which can be either nominal or ordinal, and continuous data, which can be either ratio or interval [63, 67, 90]. Nominal variables are names without implicit or explicit order, such as names of countries or kinds of vegetation. Ordinal variables are names with an implicit order, such the university ranks of assistant professor, associate professor, and full professor. Interval variables are also ordered, but in contrast with ordinal variables, there is a meaning to distances between categories. A typical example is temperature in °C. Although it makes sense to talk about the difference in temperature measured in degrees Celsius, the statement "2 °C is twice as warm as 1 °C" is meaningless: the degrees Celsius scale does not have a proper zero. Ratio variables are variables that do have a natural starting point, such as degrees Kelvin, annual rainfall, and number of elephants per square kilometer.

Geographic data is not static, but may change over time. On some occasions it is not sufficient to maintain only the most recent data; it may be necessary to record the changes as well. In agriculture for instance, certain kinds of vegetation can only be grown a limited number of successive years, to prevent soil exhaustion or the spread of diseases. If a GIS is used in this kind of situation, then time has to be modeled as well [65, 67, 127].

Two important issues in data quality are *accuracy* and *precision* [127, p. 148] [67, p. 300]. Accuracy is the degree in which errors are present, while precision or resolution is the degree of refinement in which a measurement is stated. For example, if the width of a road is expressed in millimeters, but the value is 20 percent too small, then we have that

high precision combined with low accuracy. Accuracy may change over time [127, p. 16]: ten-year old population figures are unlikely to accurately reflect the present situation, but may have been very accurate when they were acquired.

1.1.3 Geographic maps

Geographic maps are two-dimensional graphic representations of the earth surface, either on a sheet of paper or on a computer screen. The sizes of geographic objects on a map are for obvious reasons usually smaller than the sizes in reality; the ratio between these sizes is called the *scale* of the map. Trying to depict every aspect of reality on the map would render it useless, because it would be unreadable. Instead, most maps are *thematic maps*, showing only what is relevant for the intended use of the map. To display the different kinds of data described in the previous section, the geographer or cartographer has several types of maps at her disposal [25, 34, 90]: choropleth maps, isoline maps, and network maps are the most noteworthy types (see Figure 1.1).

Choropleth maps are subdivisions of the displayed part of the earth surface into shaded or colored regions. Each color or shade represents a value or range of values of the displayed phenomenon. Higher values are usually darker. Cartograms are a special kind of choropleth maps, where the regions are deformed such that the ratios of the areas of the regions conform with the ratios of some numerical aspect of the regions to be displayed. On a cartogram showing population size for instance, if one region has twice as many inhabitants as some other, possibly larger region, the regions are deformed such that the area of the former region is twice the area of the latter region.

Isoline maps are used to display continuous data. Like choropleth maps, they are also subdivisions of the displayed area into regions, but the emphasis here is on the boundaries between regions, not on the regions themselves. These boundaries, or isolines, show the points in the domain that map to a specified set of function values. Examples of isoline maps are air pressure maps as used in meteorology, and contour maps, showing curves of equal height in mountainous area.

Network maps, such as railroad maps and maps of piping systems, are in fact graphs showing geographic objects and the connections between them; the regions in a network map are meaningless. Precise locations of the objects are not necessarily preserved, although they should globally correspond to the locations of the objects in the real world to be readable for humans.

1.2 GIS: preliminaries

1.2.1 GIS functionality

The functionality offered by a GIS should at least contain the following elements [55, 107]: data input, preprocessing, data management, basic spatial queries, geographic anal-

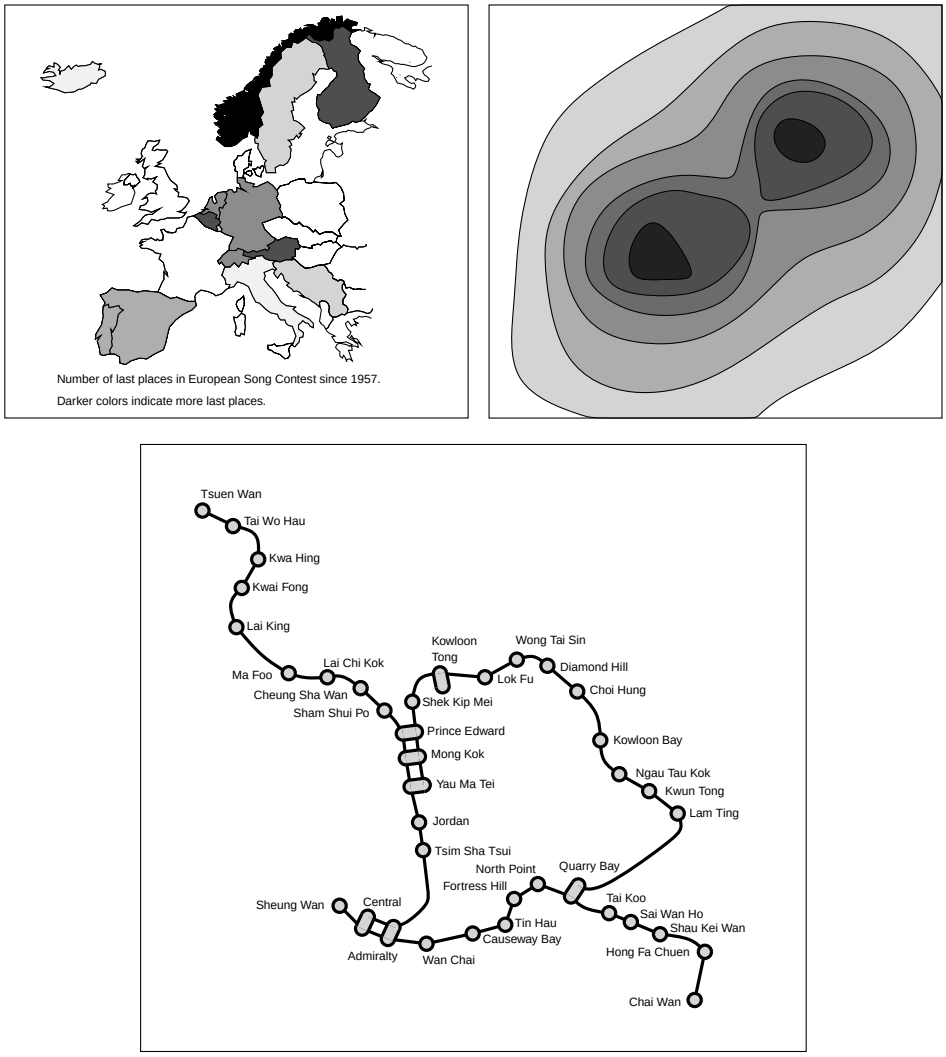


Figure 1.1: A choropleth map (top-left), an isoline map (top-right), and a network map (bottom).

ysis, and output generation. We will describe these elements briefly in the next paragraphs, and expand on some selected topics in the next subsection.

Data input There are several ways to acquire data and to put it into a GIS. Most often, existing data will be used as a source, such as paper maps and public domain or commercial data files. Paper maps will have to be digitized, and although there are methods to automate this process, in practice most of the digitizing is done manually [29].

This is a time-consuming but relatively low-cost process, since the techniques can be taught to users within hours. Automated digitizing by scanning is only worth considering for aerial photographs or maps that contain very simple and uncluttered data; otherwise, the advantages of the automated input will be nullified by the need for clean-up post-processing [29].

Preprocessing If we want to use data from various sources in a GIS, several preprocessing steps have to be made. Firstly, it is required that all data use the same geographic reference system, such as latitude-longitude or Universal Transverse Mercator coordinates [90, p. 101 ff.] [107, p. 98 ff]. Secondly, the data has to be converted to a common raster or vector format [107, p. 77 ff.] (see also Section 1.2.2). Thirdly, errors in the source data or errors introduced in the digitizing process have to be corrected [107, p. 93 ff.]. For instance, if in the digitizing of a paper choropleth map two neighboring regions are processed separately, then their common boundary may not exactly match (see Figure 1.2, left). Also, inconsistencies between different data sources have to be dealt with. Aerial photos may show a perspective distortion as well as a distortion due to the curvature of the earth surface. Objects crossing the boundary between two such photos of adjacent regions may be distorted differently on each photo, such that the edges of the object that cross the boundary don't match (see Figure 1.2, right). The correction of this kind of error is called *edge matching*, and it can be done manually or automated [107, p. 96 ff]. Finally, if the detail in the source data is greater than we actually need in the GIS, a data reduction and generalization step has to be made to prevent excessive storage requirements and processing costs [107, p. 91 ff].

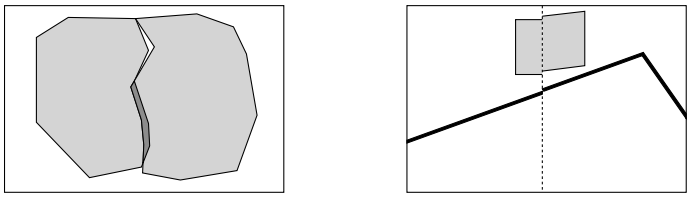


Figure 1.2: Errors introduced in the digitizing process.

Data management Just like any other information system, a GIS should provide the basic database functionality [55, p. 66 ff.]. The system must perform integrity checks when data elements are entered or removed from the database [107, p. 129]. For example, when in a topological data structure two neighboring polygons merge into one by removing a common edge, pointers to that edge have to be updated. Also, there should be mechanisms to allow for concurrent access to the database. The efficiency of data storage and retrieval is dependent on many factors; one of them is the layout of the data on the disk. In GISs it may be worthwhile to consider storage schemes that exploit the *spatial coherence* of the queries: geographical objects that are accessed in subsequent queries are often close to one another in the real world, and they should also be close together on the disk or on the same disk page, to reduce disk access time [8].

Basic spatial queries Apart from the basic queries that must be supported by any database management system, a GIS must also be able to answer spatial queries such as "what is at location (x,y)?", and "what is the area of region such-and-so?". Also, the calculation of buffers around objects, to answer queries like "how many houses lie within 100 meter from that river?", must be supported [107, p. 157] (see Figure 1.3). When a categorial vector model (see Section 1.2.2) is used for a choropleth map, it is often necessary to traverse all cells in some specified part of the subdivision to collect data for further calculations; this is facilitated by the use of topological data structures. Subdivision traversal is handled in detail in Chapter 2.

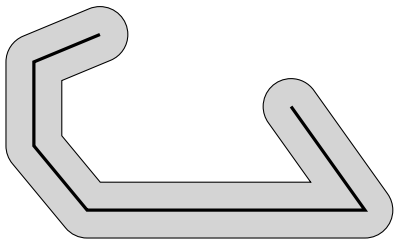


Figure 1.3: A buffer around a polyline.

Geographic analysis The main advantage of a GIS over a collection of paper maps is the ability to easily combine different kinds of data and to infer relations between them by overlaying two or more layers of thematic data. No self-respecting GIS can do without map overlay algorithms [67, 90]. After the map overlay has been done, the data can be subjected to all kinds of statistical analysis such as the calculation of correlation between variables [58]. Other statistical operations include the plotting of histograms and the calculation of the nearest-neighbor index [73, p. 119 ff.] of a set of point objects on a map. The nearest-neighbor index is a measure of regularity of the data; it is the ratio of the actual mean distance to the nearest neighbor and the expected mean distance to the nearest neighbor of a uniformly distributed point set of the same size. A different kind of geographic analysis is facility planning: the determination of one or more locations that are suitable for some purpose [55, p. 138 ff.]. To perform this kind of analysis it must be possible to specify constraints on the locations and optimization factors. For example, when a new factory is to be built, it should ideally be close to main roads or railway terminals to facilitate transportation of products and raw materials, close to a river if cooling water is needed in the production process, but at least at some minimum distance from urban area. See also Chapter 5, where we give a new algorithm for a facility location problem in mountainous area.

Output generation The output of a GIS consists mainly of maps, charts, and tables, either on paper or screen [55, 107]. It is usually not possible to display all the available data on a specific theme in full detail on the map, for that would render the map unreadable. Important cartographic generalization operations that should be supported by a GIS include selection, simplification, and aggregation [19, 90] (see Figure 1.4). Selection is

the process of choosing what data to display, and what to discard, in order to keep the map readable (see Chapter 4 for settlement selection). Depending on the target scale of the map, the level of detail of the displayed data has to be tuned. A road with a large number of small bends may be simplified to a polyline with fewer bends, and a cluster of small areas displaying single houses may be aggregated into a single region of urban area. Another form of complexity reduction is classification of the data. The placement of labels with names of cities, rivers, and regions on the map is a difficult problem; labels must not overlap, and it must be clear what label corresponds with what object. A GIS should be able to place text labels automatically. For references to label-placement literature, see Christensen et al. [23] and Doerschler and Freeman [39].

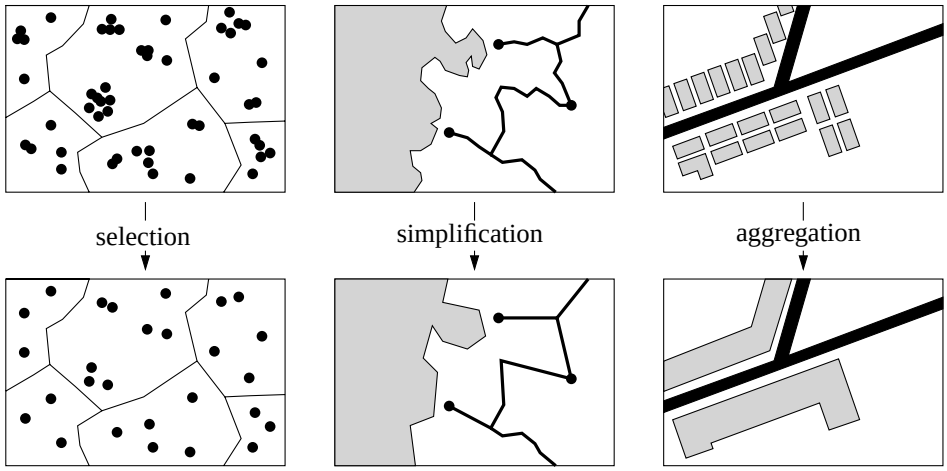


Figure 1.4: Some cartographic generalization operations.

Other examples of GIS output are pie charts showing annual gross income of various countries, tables of population of all cities in some region, and even single numbers, such as the nearest-neighbor index for a set of points on a map [73, p. 119 ff.].

1.2.2 GIS data models and structures

In this section we will discuss various data structures for storing and searching the different kinds of data that we described in Section 1.1.2. Basically, there are two ways to store both categorial and continuous data: *raster-based* and *vector-based* [127, p. 15].

Raster and vector models for categorial data A (planar) raster is a two-dimensional array of equal-sized square cells, discretizing the real world [127, p. 15]. Boundaries between regions do not cross the interiors of the cells; each cell stores a single value for each attribute.

Raster structures are very simple to implement, because arrays are well-supported by computer architectures and programming languages. For instance, displaying a rectangular portion of a raster-based map on the computer screen is simply a matter of directly accessing those cells whose x- and y-coordinates fall between the rectangle boundaries. The major drawback of raster structures is their size: cell sizes should be small enough to show the desired detail, and if a high resolution is required, the raster size becomes very large. Using run-length encoding, or its two-dimensional counterpart block-encoding, one can reduce the storage requirements of raster structures [127, p. 254 ff.], but at the same time this greatly reduces their simplicity, since individual cells are no longer directly accessible.

In contrast with raster structures, which are image-based, vector models take the object-based approach [127, p. 16]. Points are stored with their coordinates according to some reference system, line segments by their two end points, and regions are defined by their bounding line segments. When incidence relations between points, segments, and regions are stored explicitly, we speak of a *topological data structure* [127, p. 193]. In a non-topological data structure, each polygon is stored independently as a sequence of vertices. As we have seen in Section 1.2.1 this may result in errors such as gaps and slivers, when two adjacent polygons do not match exactly. In a topological data structure these problems are avoided, because each feature (in this case, the common edge between two adjacent polygons) is stored only once. Also, topological data structures make it possible to efficiently access adjacent features and to traverse the subdivision.

Raster and vector models for continuous data Just like categorial data, real-valued functional data can be represented by raster as well as vector models. Although the function values in continuous data do not necessarily represent height, the term *digital elevation model*, or *DEM*, is commonly used to denote these models [127, p. 162]. We can distinguish between the raster-based *elevation matrix* [127, p. 162] and the vector-based *isoline model* [127, p. 163] and *triangulated irregular network* [127, p. 206], or *TIN* (see Figure 1.5).

The *elevation matrix* [127, p. 162] is a two-dimensional array of square cells, each cell storing a height value or elevation. The stored height can be the height at the center of the cell, or the average height of the cell. In fact, the only difference between an elevation matrix and a raster structure for categorial data is that the set of values stored in the latter structure is usually taken from a finite set, while a cell in an elevation matrix may contain any real (or at least, any number representable by a computer).

The *isoline model* [127, p. 163] is a way to represent and visualize a real-valued function defined over the plane. Isolines are curves in the plane that map to the same function values, and by choosing an appropriate set of isolines for various function values, humans can get a good overview of the "behavior" of the function. For instance, when using this model to represent a mountainous terrain, it is easy to determine the pits, peaks, and passes of the terrain, or to distinguish steep areas from less steep areas. Isolines can be represented by polylines, and they are either closed loops or they end at the boundary. To determine the function value of a point in between two isolines, one would have to interpolate in some way. However, different ways of interpolating may give different

17	11	7	7	4
16	18	14	18	9
5	14	13	20	11
4	4	5	11	10

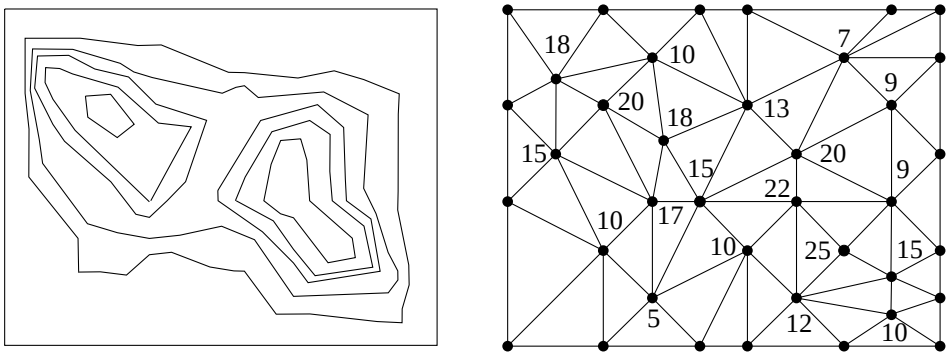


Figure 1.5: An elevation matrix, isoline model, and TIN for the same source data.

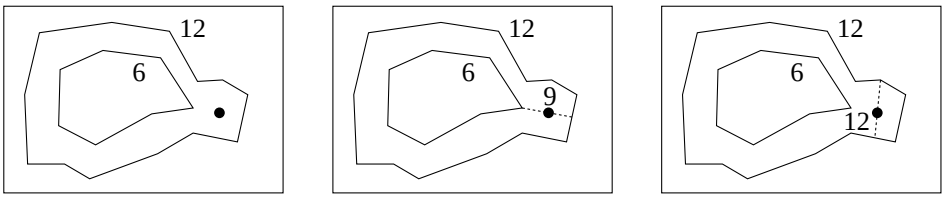


Figure 1.6: The function value at the dotted location can be obtained by interpolation, but results are not unique.

results, and it is not always clear how to interpolate (see Figure 1.6). Since the isoline model is a planar subdivision, we can use a topological data structure such as the doubly-connected edge list [31, 79] (see also Section 1.3.1) to represent it.

The *triangulated irregular network* [127, p. 206], or *TIN*, is a subdivision of the plane in triangles. The height of the vertices of the triangles is stored explicitly, and the height of any point in the interior of a triangle can be derived by linear interpolation of the height values at the vertices of the triangle. The size of the triangles varies (hence the term "irregular"), and this makes the TIN a memory-efficient data structure for modeling the surface of the earth: in relatively flat areas we can do with a small number of large triangles, while smaller triangles are used in areas with a lot of variation in height. Like the isoline model, we can store a TIN using the doubly-connected edge list [31, 79], or other topological data structures, tailored towards triangulations.

Indexing structures for raster data A widely-used structure for storing and searching planar raster data is the quad tree [92–96]. The quad tree recursively subdivides the raster into four equally-sized quadrants. A quadrant in which all cells have the same attribute value is not subdivided any further. So, the quad tree is a tree with as root the whole raster, and each non-leaf node has four children (see Figure 1.7). The main advantage of the quad tree is that it only recurs in regions with much detail, while large homogeneous regions are stored efficiently.

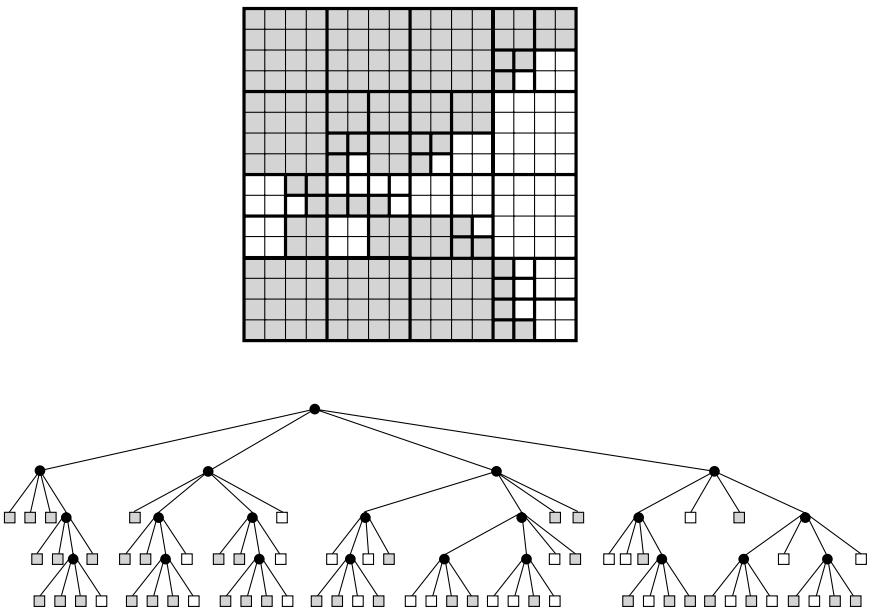


Figure 1.7: A raster subdivision, and a quadtree for it. The order of the quadrants is north-west, north-east, south-west, and south-east.

Indexing structures for vector data Perhaps the most important structures supporting rectangular search queries in categorial vector data are the R-tree [52] and its relatives. The R-tree may be seen as a two-dimensional version of the B-tree, and it is designed with the reduction of disk access and dealing with non-uniformly distributed data in mind. The

leaf nodes of the R-tree correspond to the smallest enclosing (axis-parallel) rectangles of the objects to be stored. Each non-leaf node stores the smallest rectangle containing the rectangles of its children, and the number of children lies between some predefined minimum and maximum. The root of the tree is the smallest axis-parallel rectangle that encloses all objects (see Figure 1.8). A rectangular search query is performed by recursively searching the R-tree, descending only in those nodes for which their stored rectangles intersect the query rectangle. It follows that the rectangles stored in an R-tree should be as small as possible; the difficult part of constructing an R-tree lies in finding a grouping of the rectangles that fulfills this requirement. The rectangles stored in the internal nodes of an R-tree may overlap, and this is undesirable, because it makes searching less efficient. In the R^+-tree [108], overlapping of non-leaf nodes is not allowed, and rectangles are partitioned when necessary to avoid overlapping. The drawback is that it is difficult to guarantee a minimum on the number of children of a non-leaf node. The R^*-tree [15] is another variant on the R-tree, where the grouping of rectangles is guided by some weighted balancing between overlapping of the resulting rectangles and smallest possible area and perimeter.

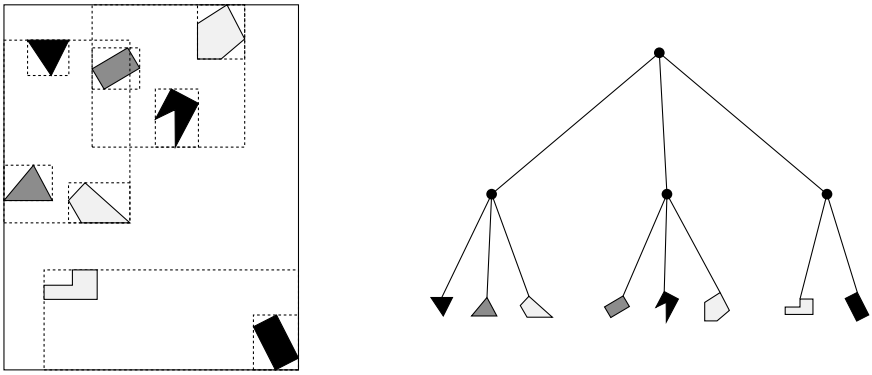


Figure 1.8: Vector objects, and an R-tree on them.

1.3 Computational geometry: preliminaries

Computational geometry arose from the field of design and analysis of algorithms and developed into a discipline of its own in the late seventies. The basic objects of interest for the computational geometer are points, lines, spheres, rectangles, and the like. Computational geometers are interested in algorithmic versions of questions like “How many of these line segments intersect?”, “What is the largest circle that will fit in this polygon?”, “Does this point lie in the interior of that rectangle?”, and “Which one of this set of points is nearest to this query point?”. One reason for the growth of the field is that

it has applications in many domains. A great many of the basic problems from robot motion planning, computer aided design, computer graphics, and GIS, to name a few, are of geometric nature. Another reason of the popularity of computational geometry is the beauty of many of the problems and the elegance of their solutions. Geometric problems are often simple to express and to explain to outsiders of the field, and in many cases even the solutions to these problems are relatively simple to understand for people without a strong geometric background.

Just like the field of algorithms design and analysis, computational geometry is concerned with the efficiency of the solutions to its problems: how much time and computer memory are needed to solve a particular problem? Rather than comparing exact values of time and memory consumption, algorithms researchers tend to look at the *asymptotic* behavior of algorithms, expressed in O- ("big-oh") notation: "this algorithm runs in $O(f(n))$ time" is a shorthand for "there is a constant N and a constant c such that for input size n greater than N, the running time is *at most* $cf(n)$ time units." Similarly, lowerbounds are expressed with Ω-notation: an $\Omega(f(n))$ memory bound means that there are constants N and c such that the memory requirements of the algorithm is *at least* $cf(n)$ memory units for certain inputs of size $n > N$. Finally, when there exist constants c_1, c_2 and N such that the running time or memory usage of the algorithm lies between $c_1 f(n)$ and $c_2 f(n)$ units for input sizes of at least N, the bounds are denoted with $\Theta(f(n))$.

In the next four sections we discuss some important data structures and concepts that we will use in the rest of this thesis. In the description of the algorithms we assume that the input objects are in *general position*. This means that no two lines or line segments are parallel, no three points are collinear, and no four points lie on one circle. This assumption greatly simplifies the description of the algorithms.

1.3.1 The DCEL structure

A planar subdivision $\mathcal{S}$ is a partition of a two-dimensional manifold into three finite collections of disjoint parts: the vertices, the edges, and the cells. A subdivision is *connected* if its edges and vertices form a connected set. This implies that all bounded cells are simple polygons without holes.

When working with vector-based subdivisions, one often needs information about the topological relations between vertices, edges, and cells. A data structure that explicitly stores this information is the *doubly-connected edge list (DCEL)* (see Muller and Preparata [79], de Berg et al. [31]). In the DCEL-structure, edges in the subdivision are represented by two directed half-edges, such that one of the half-edges bounds one cell incident to the edge and the other half-edge bounds the other cell. If $\vec{e}$ is a half-edge, then $twin(\vec{e})$ denotes the half-edge that is part of the same edge. Every half-edge is oriented in such a way that the cell to which it is incident lies to its left. In this way, for all bounded cellsof a connected subdivision, the incident half-edges form a counterclockwise cycle around the cell. The half-edges that bound the one unbounded cell (the *outer cell*) form a clockwise cycle. This is illustrated in Figure 1.9.

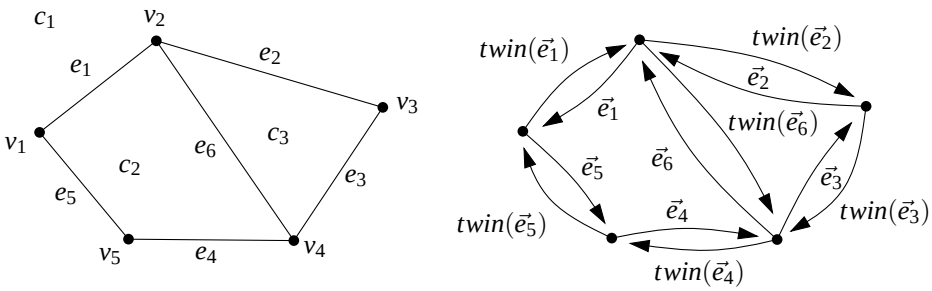


Figure 1.9: A subdivision and the corresponding orientation of half-edges.

The DCEL consists of a table of half-edge records, a table of vertex records, and a table of cell records.

Any vertex record stores the coordinates of the vertex and a single reference to a half-edge record of a half-edge that has the vertex as its origin; the other half-edges incident to this vertex can be accessed by first accessing the referenced half-edge. The primitives *Coordinates*(v) and *Incident-edge*(v) are used for the values of these fields for a vertex v. Also, non-geometric information may be stored. For the subdivision in Figure 1.9 the vertex table may look something like this:

Vertex	*Coordinates*	*Incident-edge*
v_1	$(\cdots,\cdots)$	$twin(\vec{e}_1)$
v_2	$(\cdots,\cdots)$	$\vec{e}_1$
$\cdots$	$(\cdots,\cdots)$	$\cdots$

If the edges and vertices of the subdivision form a connected subset of the plane, then every cell is incident to one cycle of half-edges. In this case, a cell record stores one reference to a half-edge record of a half-edge incident to that cell, and possibly some non geometric information. The primitive *Outer-incident-edge*(c) is used to access the half-edge for a bounded cell c, and the primitive *Inner-incident-edge*(c_1) is used for the unbounded cell c_1. The table below shows the cell table for the subdivision in Figure 1.9.

Cell	*Outer-incident-edge*	*Inner-incident-edge*
c_1		$twin(\vec{e}_4)$
c_2	$\vec{e}_1$	
c_3	$\vec{e}_2$	

For any half-edge $\vec{e}$, its record stores one reference to the record of the vertex at which it originates, a reference to the record of the incident cell c (which lies to its left), a reference

to the half-edge that precedes $\vec{e}$ in the cycle of edges around c, a reference to the record of the half-edge that follows after $\vec{e}$ in this cycle, and a reference to the record of the complementary half-edge $twin(\vec{e})$:

Half-edge	*Origin*	*Incident-cell*	*Prev*	*Next*	*twin*
...	...	...	...	...	...
e_5	v_1	c_2	$\vec{e_1}$	$\vec{e_4}$	$twin(\vec{e_5})$
$twin(\vec{e_5})$	v_5	c_1	$twin(\vec{e_4})$	$twin(\vec{e_1})$	$\vec{e_5}$
...	...	...	...	...	...

The DCEL-primitives can be combined to answer more complex queries. For example, suppose that we want to know the first outgoing half-edge of vertex v_4 that we encounter if we rotate counterclockwise around v_4 and start at half-edge $\vec{e_3}$ (Figure 1.9). This query is answered by $twin(prev(\vec{e_3}))$.

The DCEL-structure can easily be extended to deal with unconnected subdivisions as well (Figure 1.10). Considering the geometric structure of a cell, we observe that a cell is either unbounded or bounded from the outside by one single cell or component. Furthermore, the cell may be bounded from the inside by zero or more components.

In the cell record that represents the cell it is convenient to have references to all incident components. Since the outer component, if it exists, plays a special role there will be a reference to some half-edge of this component. This reference is empty for the unbounded cell. For the other components the cell record stores a list of which each element contains one reference to some half-edge of a component. There is no natural order for these components so we let the list be unordered. To be able to access the cell record from the incident half-edges we add one more reference to every half-edge record to the record of the incident cell. The primitives *Outer-incident-edge*(c) and *Inner-incident-edges*(c) are used to obtain these features.

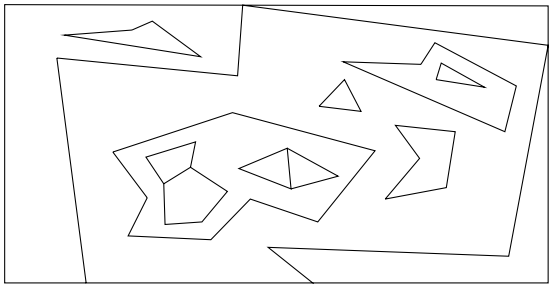


Figure 1.10: The DCEL-structure can be extended to deal with unconnected subdivisions.

1.3.2 The sweeping paradigm

A basic geometric operation is the determination of intersections between geometric objects. Consider for instance the problem of finding all intersections between pairs of segments from some set S of n line segments in the plane. A simple approach would be to check all pairs of line segments. This takes $\Theta(n^2)$ time, regardless of the actual number of intersections, which can be any number between zero and $\frac{1}{2} \cdot n(n-1)$. Using a plane sweep approach however, determining all k intersections between pairs of segments from S takes only $O((n+k)\log n)$ time. The first algorithms that take this approach are by Shamos and Hoey [99], Lee and Preparata [68], and Bentley and Ottmann [16].

To simplify the description of the plane sweep algorithm, we will first consider the one-dimensional variant to the line-segment intersection problem, namely, the determination of all intersections, or overlaps, between pairs of intervals from some set S' of n intervals. Again, it is possible to solve this in $\Theta(n^2)$ time by testing all pairs of intervals, but it is not difficult to improve on this: if we sort the endpoints of the intervals, we can treat them in order from left to right. Imagine that we "sweep" the interval $(-\infty, \infty)$, that encloses our set S', by moving a point p from left to right. An interval in S' becomes *active* when p passes its left endpoint, and it becomes inactive again when p passes its right endpoint. The key observation is that two intervals intersect if and only if they are both active at some time during the sweep with p. So if we keep track of which intervals are active, we simply report the intersections of all these active segments with a new one upon its activation. In sweeping terminology, the list of active intervals is called the *status structure*; it is used to determine the intersections by looking only at intervals that really matter. The status structure changes upon certain *event points*; these are the left endpoints of the intervals, where an interval becomes active, and the right endpoints, where an interval becomes inactive. All event points are known in advance in this case, so the event list, a sorted list of left and right endpoints, can be computed before starting the actual sweep. If we maintain pointers between the events in the event list and the status structure, the insertion and removal of an interval in or from the status structure can be dealt with in $O(1)$ time. The total running time becomes $\Theta(n\log n + k)$, where k is the number of reported intersections: $\Theta(n\log n)$ time for the sorting, $\Theta(n)$ time for the insertion and removal of intervals in or from the status structure, and $\Theta(k)$ time for the reporting. The running time can still be as high as $\Theta(n^2)$, but only if the actual number of intersections is $\Theta(n^2)$; the algorithm is what we call *output sensitive*.

Back to our original problem, the determination of intersections between line segments in the plane. The one-dimensional sweeping algorithm can be adapted for use in two dimensions in the following way: instead of sweeping the interval $(-\infty, \infty)$ with a point, we sweep the plane with a line from top to bottom (see Figure 1.11).

The status structure is a list of active segments, i.e., those segments that are intersected by the line. As in the one-dimensional case, two line segments can only intersect if they are both active at some moment (i.e., if there is a horizontal line intersecting both segments), but the reverse is not necessarily true in two dimensions: two active segments need not intersect. So testing all active segments for intersection with a newly active one may not

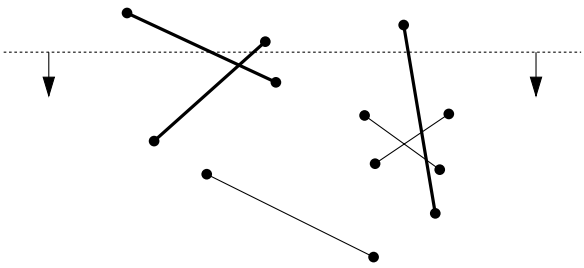


Figure 1.11: Sweeping the plane in downward direction. Active segments are fat.

be very efficient, and in fact it is possible to trigger $\Omega(n^2)$ such tests while there are no intersections at all, as is illustrated in Figure 1.12.

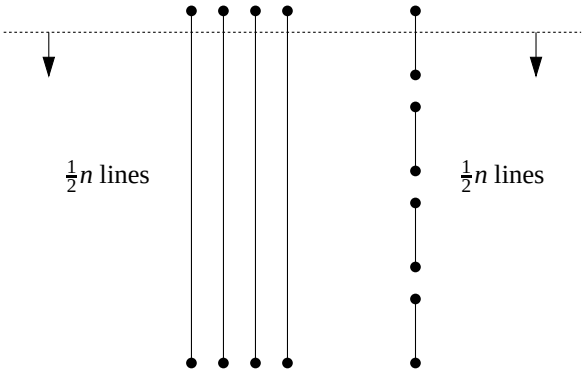


Figure 1.12: Testing all active segments for intersection with a newly active one can lead to an $\Omega(n^2)$ running time, even when there are no intersections.

It appears that we need another observation: just before the sweep line reaches an intersection between two segments s_i and s_j in S, these two segments are neighbors on the sweepline. This suggests the following approach: we maintain the status structure as a *sorted* list of line segments; the sorting is on the x-coordinate of the intersection of the active segments with the sweepline. We only test for intersections between active segments that are neighbors on the sweepline. The status structure changes on three kinds of events, which we handle as follows:

- The sweepline reaches the topmost endpoint of a segment. The segment becomes active, and has to be inserted in the proper place of the status structure. We test for intersections between the newly active segment and its (at most two) neighbors. New intersections are reported and inserted in the event list. See Figure 1.13, left.

- The sweepline reaches an intersection between two active segments. Their order in the status structure is changed, and they are tested for intersection with their (at most two) new neighbors. See Figure 1.13, middle.
- The sweepline reaches the bottommost endpoint of an active segment. The segment becomes inactive and is removed from the status structure. Its neighbors are tested for intersection. See Figure 1.13, right.

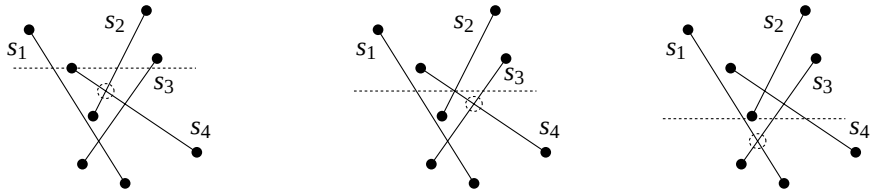


Figure 1.13: Three kinds of sweep events.

If we maintain the status structure as a binary search tree, all changes of the status structure can be done in $O(\log n)$ time, since at any time during the sweep at most n segments are active.

The event queue is also maintained as a binary search tree; initially it contains only the endpoints of all segments; intersections are inserted when we find them. Although the total number of events can be $\Theta(n^2)$, we can keep the size of the event queue linear by maintaining only the intersections of segments that are neighbors on the sweepline (see Brown [17] and Pach and Sharir [83]). This means that some intersections are removed and reinserted again if the order of the active segments changes (see Figure 1.14). Hence, all operations on the event queue can be done in $O(\log n)$ time.

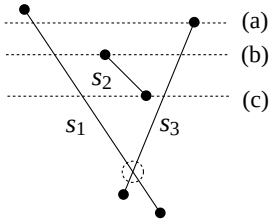


Figure 1.14: An intersection is added to the event queue at (a), removed again at (b), and reinserted at (c).

The total running time of this sweep algorithm is $O((n+k)\log n)$ time, where k is the number of intersections: there are $O(n+k)$ events, each of which takes $O(\log n)$ time to handle. Details and a more in-depth analysis can be found in any textbook on computational geometry (e.g. de Berg et al. [31]).

1.3.3 Search structures

Another important basic operation in computational geometry is searching for geometric objects. This problem comes in two varieties. The first is the location of some query object, for instance, the reporting of the face of a planar subdivision that contains some query point; this is the *point location problem* [26, 40, 62, 80, 86, 97, 98]. We can also reverse the problem: given a set of geometric objects, such as points in the plane, and a query range (for example, a rectangle), report all the objects that lie inside the query range. This is what we call *range searching* (see for instance chapters 5, 10, and 16 of de Berg et al. [31]). For both searching problems there is an obvious brute-force solution: simply test all the stored objects with the query object. But there are more efficient solutions, all of which are in fact based on the same principle as the binary search in the dictionary, as described above: they all recursively narrow the region in which the search takes place.

Planar point location is used extensively in this thesis, so we will treat it here in somewhat more detail. Let $\mathcal{S}$ be a planar subdivision of the with n edges in total. It is not at all straightforward to determine without preprocessing whether a query point p lies inside a given polygonal region of $\mathcal{S}$; the region can have a linear number of edges. The problem becomes easier if the region is a triangle or a convex quadrilateral; three sidedness tests, or four, respectively, suffice in that case to answer the question: if the DCEL-structure is used to represent the subdivision, then the point lies in the region if and only if it lies to the left of the half-edges incident to the region. This motivates us to refine the subdivision by breaking up the polygonal regions in the following way (see Figure 1.15): from each vertex we draw vertical segments upward and downward, until we hit an edge of the subdivision. In the resulting refined subdivision, which is known as the *trapezoidal map* [98], every region is either a triangle or a quadrilateral. The total number of edges we add is twice the number of vertices in the original subdivision, which is $O(n)$, since the subdivision is planar. The total number of vertices, edges, and faces in the trapezoidal map is therefore also $O(n)$. If the edges of the original subdivision are in general position, then each region in the trapezoidal map has one or two vertical sides, and exactly two non-vertical sides.

A search structure for the trapezoidal map is a directed acyclic graph, and it can be built with a randomized algorithm (see Seidel [98]). We first determine a bounding box of the edges in the subdivision, and initialize a trapezoidal map structure and search structure (consisting of a single node representing the bounding box). Next, we add the edges of the subdivision in random order. Each time that we add an edge, we update the trapezoidal map by identifying which trapezoids are replaced by new ones, and these modifications are reflected in the search structure. The leaf nodes of the resulting search structure represent the trapezoids of the final trapezoidal map; the internal nodes are either *x-nodes*, representing an endpoint of some segment, or *y-nodes*, representing a segment itself. Searching this structure is done by descending it starting from the root (see Figure 1.16, taken from [31, p. 127]). At an x-node we compare the x-coordinate of the stored endpoint with that of the query point to decide whether we have to descend to the left or right child of the x-node. At an y-node, we test whether the query point lies above or below the stored segment.

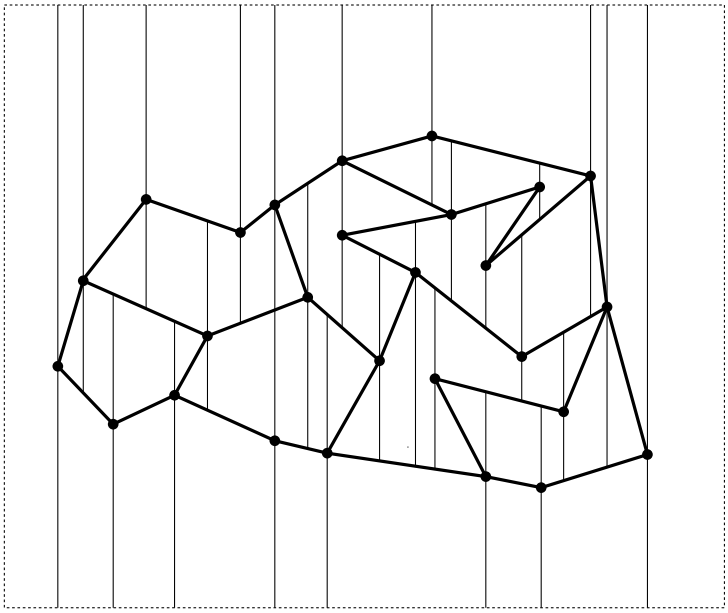


Figure 1.15: Trapezoidal decomposition of a planar subdivision

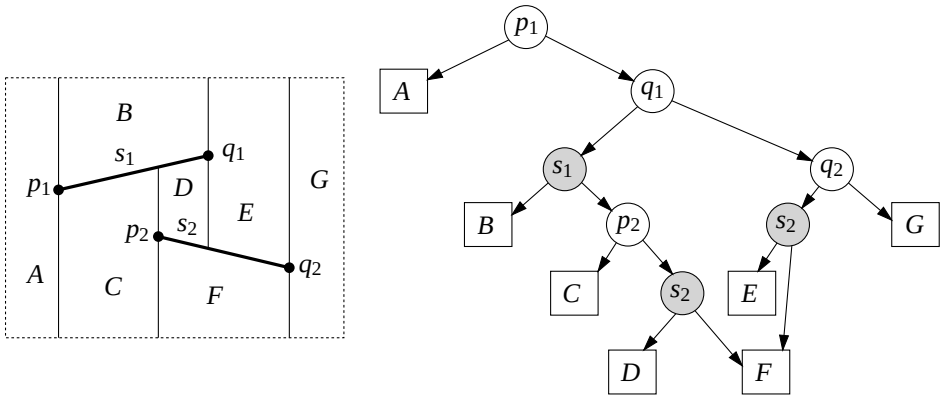


Figure 1.16: Search structure for the trapezoidal map.

The expected running time for the construction of the trapezoidal map and the accompanying search structure is $O(n \log n)$; the expected size of the search structure is linear in

n. Searching with a query point takes $O(\log n)$ expected time. Details on the construction of the trapezoidal map and the search structure, as well as an in-depth analysis of preprocessing time, query time, and memory requirements can be found in Seidel's paper [98].

For completeness we mention the most important structures that support range queries in a set of n geometric objects (see de Berg et al. [31]):

- A *kd-tree* for a set P of n points in d dimensions can be constructed in $O(d \cdot n \log n)$ time. It takes $O(d \cdot n)$ memory, and supports rectangular range queries in $O(n^{1-1/d} + k)$ time, where k is the number of reported points.
- A *range-tree* for a set P of n points in d dimensions takes $O(n \log^{d-1} n)$ storage and $O(n \log^{d-1} n)$ construction time. It supports range queries in $O(\log^d n + k)$ time, where k is the number of reported points. Using a technique called *fractional cascading* (see Chazelle and Guibas [20, 21]), the query time can be improved to $O(\log^{d-1} n + k)$.
- An *interval tree* stores a set of n intervals on a line in $O(n)$ memory, and can be used to report all the intervals that contain a query point in $O(\log n + k)$ time, where k is the number of reported intervals. Building an interval tree takes $O(n \log n)$ time.
- A *priority search tree* for a set P of n points in the plane uses $O(n)$ storage and can be built in $O(n \log n)$ time. It can be used to report all the points in P that lie in the unbounded query-range $(-\infty, q_x] \times [q_y, q'_y]$ in $O(\log n + k)$ time, where k is the number of reported points.
- A *segment tree* stores a set of n intervals on a line in $O(n \log n)$ memory, and can be used to report all the intervals that contain a query point in $O(\log n + k)$ time, where k is the number of reported intervals. Building an interval tree takes $O(n \log n)$ time.
- A *partition tree* can be used for triangular range queries. For a set of n points in the plane it takes $O(n)$ storage, and it can be constructed in $O(n^{1+\varepsilon})$ time for any $\varepsilon > 0$. Counting the number of points in a query rectangle takes $O(n^{1/2+\varepsilon})$ time using the partition tree; reporting the points takes an additional $O(k)$ time, where k is the number of reported points. The query time can be improved to $O(n^{1/2}(\log n)^{O(1)})$.
- a *cutting tree* can be used to report the number of lines of a set of n lines in the plane that lie below a certain query point in $O(\log n)$ time; the lines can be reported in an additional $O(k)$ time, where k is the number of reported lines. The structure takes $O(n^{2+\varepsilon})$ memory and can be constructed in $O(n^{2+\varepsilon})$ time for any $\varepsilon > 0$. The $O(n^{\varepsilon})$ factor in the storage requirements can be removed.

In spite of the query time of the partition tree and the storage requirements of the cutting tree, which may seem not very attractive, these are very powerful (but complex) structures. Their *multi-level* variants have about the same query time and storage requirements respectively, but can answer more complex range queries, some of which are impossible to solve efficiently with the simpler search structures.

1.3.4 Voronoi diagrams

Suppose that you are walking in the desert, equipped with a GPS device (so you know the coordinates of your current location) and a thematic map, showing all the n oases with wells in the desert. You are incredibly thirsty, so you would like to identify the nearest oasis as quickly as possible. Having heared about computational geometry, you suspect that that there must be a smarter way than to simply calculate your distance to all oases and pick the one with minimum distance, all in linear time.

You are right: you should have prepared your journey through the desert by subdividing the map with the oases into regions, such that for all locations within a single region, the nearest oasis is fixed. Such a subdivision is called the *Voronoi diagram* [9, 36, 122, 123] of the oases; an example is shown in Figure 1.17. Having computed the Voronoi diagram [9, 43, 51], you can determine the nearest oasis in $O(\log n)$ time by making use of the point location techniques described in the preceding subsection.

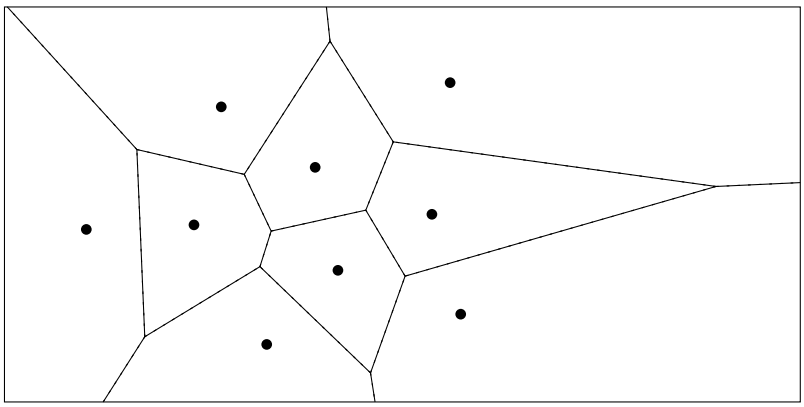


Figure 1.17: Voronoi diagram of a set of points.

More formally, let P be a set of n points in the plane, and let k be an integer such that $1 \leq k \leq n-1$. For a point $q \notin P$ in the plane, let $P(q,k) \subset P$ be any set of k points in P for which the Euclidean distance to q is less than or equal to the Euclidean distance to any point in $P \setminus P(q,k)$. The *order-k Voronoi diagram* [3, 11] of a set P of n points is a planar graph that subdivides the plane into open regions such that for any point q in a region, $P(q,k)$ is uniquely determined, and such that for any two points q and q' in the same region, $P(q,k) = P(q',k)$. The interior of the boundary between two adjacent regions is an *edge* of the order-k Voronoi diagram; it is easy to see that each edge lies on a bisector of two points in P. The non-empty intersections of the closures of three or more regions of the order-k Voronoi diagram are its *vertices*. We assume that all vertices have degree three; otherwise, a degeneracy is present.

The order-1 Voronoi diagram is known as the (standard) Voronoi diagram or *closest-site Voronoi diagram*, and the order-$(n-1)$ Voronoi diagram is also called the *furthest-site Voronoi diagram*. Voronoi diagrams have been well-studied, and they are covered in any textbook on computational geometry. Interesting varieties include Voronoi diagrams with different metrics, and Voronoi diagrams of other objects than points, such as line segments or circles of different sizes.

An interesting structure related to the Voronoi diagram is the Delaunay triangulation [49]: it is the dual of the Voronoi diagram, obtained by connecting all points whose Voronoi regions are adjacent (see Figure 1.18). The Delaunay triangulation has some interesting properties. One of these properties is the *empty circle* property: the circle through three points that are vertices of the same face in the triangulation contains no points of P in its interior, and for any two points in P that form an edge in the triangulation there is a closed disc that has the two points on its boundary and does not contain any other point of P. A second property is that the Delaunay triangulation maximizes the minimum angle over all triangulations of P [103]. This second property makes the Delaunay triangulation the triangulation of choice when long and skinny triangles are undesirable, which is the case in many applications.

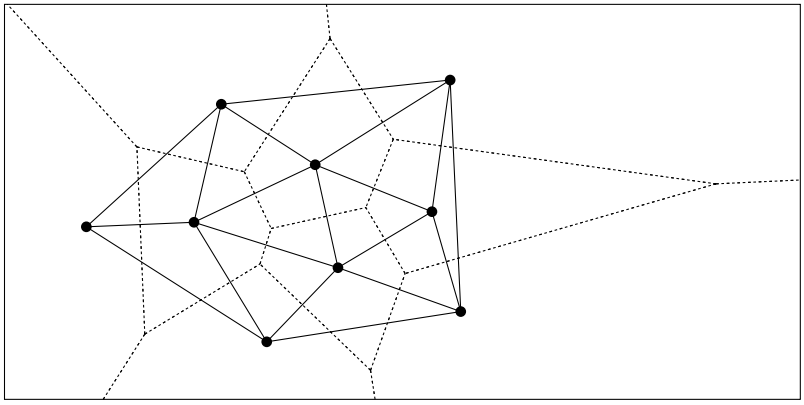


Figure 1.18: The Delaunay diagram is the dual of the Voronoi diagram

The Voronoi diagram can be computed by a sweep algorithm in $\Theta(n \log n)$ time (see Fortune [43]), and given the diagram, the Delaunay triangulation can be constructed easily in linear time. Another way to compute the Delaunay triangulation is by a randomized incremental algorithm (see Guibas et al. [49]).

1.4 Selected computational geometry problems from GIS

In the following four chapters of this thesis we will look in detail into four selected example problems that arise in the field of GIS. These four problems are introduced next.

1.4.1 Choropleth map traversal

Choropleth maps (see Section 1.1.3) are frequently dealt with in a GIS, in many different ways. All kinds of complex operations on these maps can often be carried out by repeatedly using a simpler operation that does the job for a single region of the map. Therefore, a common operation in a GIS is to traverse the map, to retrieve its regions, and to report their features or the attribute data stored with these features. For instance, if we want to draw a map showing population sizes of European countries on the screen, the polygons representing the European countries have to be retrieved and passed on to the graphics engine of the GIS. Or, if we want to determine the total number of inhabitants of Europe, we have to traverse all countries and collect information about the number of inhabitants of each individual country.

The spatial data structures in a contemporary vector-based GIS are basically planar subdivisions, and the topological relations between vertices, edges, and cells are stored explicitly. The underlying computational geometry problem of the GIS tasks sketched above is the traversal of such a topological subdivision, and the enumeration of all cells, edges, and vertices. It is clear that we have to report each relevant cell or its attribute data precisely once. Reporting a country more than once would lead to inefficient displaying of the map and an erroneous calculation of the total number of Europeans, while omitting one or more countries would yield incorrect results in both examples. Somehow we have to keep track of which cells have been reported, and which cells have not. A common solution is to use *mark bits*, which can be seen as auxiliary attributes for recording if the feature has been reported or not. However, this is not always applicable in practice; if the GIS at hand does not provide mark bits, we would either have to alter the data structures, or make a copy of the data in main memory or background storage, which is inefficient if not impossible. Another disadvantage is that in order to gather information–essentially a *read* operation–we'd have to *write* to the database for setting and resetting the mark bits, thus preventing concurrent access to the database. In Chapter 2 (based on a paper with De Berg *et al.* [32]) we show how to exploit the topological relations between the zero-, one-, and two-dimensional features, together with some basic geometric calculations, for traversing a subdivision without using mark bits. We also address the problem of reporting parts of the subdivision, such as windowing queries, where we are only interested in those cells that intersect a specified rectangle, or the reporting of connected subsets of cells with an equal-valued attribute. Both operations are common in GIS.

1.4.2 Isosurface generation

In Section 1.2.2 we have seen three models to represent continuous data in a GIS: the elevation matrix, the TIN, and the isoline model. To visualize continuous data, different methods are available, depending on the model used. For the elevation matrix and the TIN, a perspective view of the model can be generated by the GIS: in that case, the data is presented to the user as the picture of a mountainous landscape. But probably the most commonly used method for displaying continuous data is the isoline map (Section 1.1.3). Isoline maps can be stored directly, but often the data is stored as an elevation matrix or a TIN, and the isolines have to be generated by the GIS. Suppose that a GIS user wants to display an isoline map of a certain geographic area, showing the contours of her favorite set of height levels. A straightforward way to implement this is to traverse all cells of the elevation matrix or the TIN (perhaps using the techniques of Chapter 2), and to test for every cell if it is intersected by one or more of the requested contours. However, in many practical situations one can expect the number of cells intersected by a particular contour line to be roughly $O(\sqrt{n})$, where n is the number of cells. More efficient methods exist that make use of geometric data structures such as kd-trees or interval-trees (Section 1.3.3), but the storage requirements of these data structures are often too large to be worth considering in practical situations. A different approach is based on the observation that it suffices to find, for each individual contour that has to be displayed, one cell that is intersected by the contour. Once we have found such a *seed cell*, we can use it as a starting point for the traversal of only those cells that are intersected by that particular contour. It is important to realize that the contour lines of a given height value may consist of more than one connected component, each of which is either a closed curve or touches the boundary of the displayed area at both ends. In order to correctly generate all these contours for a given height value, we need a seed cell for each component. This leads to the following problem: given a regular or irregular mesh, find a minimum size seed set, such that every possible contour component intersects at least one of the cells in the seed set (see Figure 1.19). Once we have found such a seed set, we can solve the contour generating problem by searching the (hopefully small) seed set, either brute-force or with the use of the aforementioned search structures, instead of searching the whole set of cells.

In Chapter 3 (based on a paper with van Kreveld et al. [117]) we give an algorithm to compute a minimum-size seed set. Unfortunately, its running time is $O(n^2 \log n)$, where n is the number of cells, which makes it difficult or impossible to implement in many real-world situations. Therefore, we also give an approximation algorithm that runs in $O(n \log^2 n)$ time, uses sub-linear memory in practical situations, and yields a small seed set. Implementations show that that the approximation algorithm indeed is applicable in practical situations, and that it outperforms previous seed set generating algorithms in terms of seed set size. Our algorithms make use of the *contour tree*, a structure that captures the topology of the contours of certain specific values. Seed sets can be computed for three-dimensional meshes, such as medical images, as well.

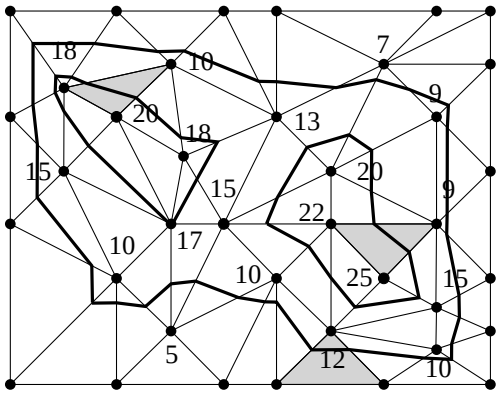


Figure 1.19: Any contour in the TIN intersects at least one of the seeds (shaded).

1.4.3 Selection for cartographic generalization

The generation of a map is a complex process involving many steps. First, it should be decided what kinds of thematic data is to be displayed on the map. Depending on the amount of data stored in these layers and the desired scale of the map, it may be the case that displaying all geographic objects would result in a heavily cluttered, unreadable map. One has to decide which objects will actually be displayed, and which ones will be omitted. After the selection step follows a generalization step, in which groups of small objects are aggregated into a single region, and the level of detail is reduced to match the scale of the map.

Chapter 4 (based on a paper with van Kreveld et al. [118]) addresses the problem of cartographic selection, where we are given a set of geographic objects and the task of deciding which ones of these to select for displaying. We limit ourselves to point objects, *settlements* (see Figure 1.20), for which we assume an importance factor to be given. For instance, the settlements could be cities, and the importance factor simply the number of inhabitants. Although in general less important settlements are dropped from the map in favor of more important ones, it is hardly ever the case that all settlements that are displayed are more important than all settlements that are omitted. For instance, on a small-scale map of the United States of America, it may not be possible or desirable to display both Philadelphia and New York. Firstly, there may simply be not enough space to display them both, and secondly, one may prefer displaying a small set of evenly distributed large cities, such as on weather forecast maps, where cities have a reference function. In both cases, Philadelphia will usually be omitted. Salt Lake City on the other hand, although smaller than Philadelphia, will be displayed, because there is no larger city close to Salt Lake City with which to compete for space. So we will have to use some notion of *relative importance*, taking proximity of cities into account. In Chapter 4, we

first discuss three existing models for relative importance. Two of these have the property that when we increase the number of settlements to be displayed, settlements that were originally displayed may disappear, and reappear again later on in the process, and for all three models it is not always possible to make a selection of a specified size. We regard this as an undesirable property of the models, and we give four new models that don't suffer from this flaw. Our models compute a *ranking* of the settlements, which is a complete order of the relative-importance factors. The advantage of having a ranking is that, after computing it once, we can store the ranking in memory linear in the number of settlements. Displaying is then simply a matter of selecting the required number of settlements in order of rank; upon change of the desired number of settlements, no time-consuming re-evaluation of relative importance factors needs to be done, as opposed to the existing models that do not give a ranking. We show how to implement our methods efficiently by making use of computational geometry data structures and techniques, such as Voronoi diagrams and point location algorithms. We also implemented both the existing and the new models and tested them on two datasets, consisting of the 158 largest cities in the US, and of 139 municipalities in the Netherlands, respectively.

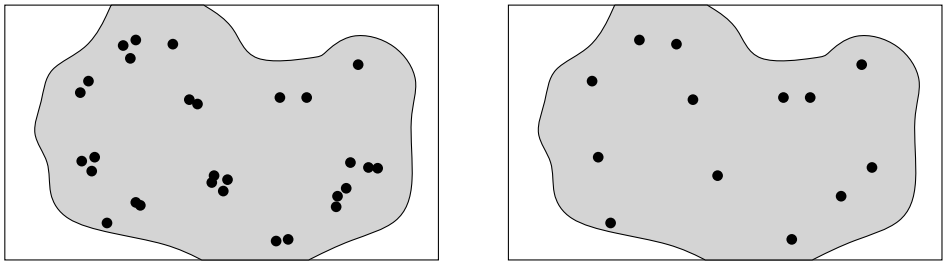


Figure 1.20: Displaying all settlements (left) leads to a cluttered map. Selecting fewer settlements (right) makes the map more readable.

1.4.4 Facility location in terrains

One of the functionalities of a GIS is geographic analysis. A GIS must be able to combine information stored in different thematic layers, and to calculate all kinds of statistics such as correlation between variables. A more complex kind of geographic analysis is the determination of locations that are suitable for some purpose. Suppose for instance that a new hospital is to be built for the benefit of several villages in mountainous area. The location of the hospital should be such that the maximum distance to any of the villages is minimized. This is an instance of the *facility location problem*, which we study in Chapter 5 (based on a paper with Aronov et al. [6]) in an abstract form. The mountainous area is modeled as a polyhedral terrain, the villages as point sites on the terrain, and distances are measured along the surface of the terrain.

In the plane, the facility location problem has been well-studied. The optimal placement of the facility is at the center of the smallest enclosing circle of the sites, and

the first deterministic linear-time algorithm to determine this circle has been given by Megiddo [74]. A much simpler randomized algorithm that runs in linear expected time, is due to Welzl [125]. Despite the simplicity of the planar version of the facility location problem, it appears to be very hard to solve it on a terrain. This is partly due to the fact that shortest paths between two points on a terrain are not necessarily unique, and distance calculations along the terrain are much more expensive than in the plane.

There is a well-known relation between the smallest enclosing circle for a set of points and the furthest-site Voronoi diagram for the same set. That is, the center of the smallest enclosing circle lies on an edge or vertex of the diagram. In Chapter 5 we show that the same relation holds on a terrain, and we give a new algorithm for computing the furthest-site Voronoi diagram on a terrain. After the diagram has been computed, the optimal location for the facility can be found by traversing it: the location on the diagram where the distance to the defining sites is minimized is the location for the facility.

CHAPTER 2

Subdivision Traversal Without Extra Storage

2.1 Introduction

The basic spatial vector data structure in any geographic information system is the one that stores the zero-, one- and two-dimensional features of planar subdivisions. Contemporary GISs like ARC/INFO [1] and the DIME file [2] use polygon structures that store the explicit topology as well. This means that from any feature, it is possible to access adjacent features efficiently. Essentially, these structures are similar to the doubly connected edge list [79] (see also Section 1.3.1) and quad edge structures [50] used in computational geometry.

A basic algorithmic routine that can be applied to any planar subdivision is its traversal. The objective of such a traversal can be to report all boundary segments on a map in order to display them on a computer screen, or to collect numerical information stored with the cells for further statistical analysis. Unfortunately, to traverse a topological polygon structure one needs to record what features have been visited, to avoid continuing forever. This means that the zero-, one-, and two-dimensional features must have a mark bit to capture this information. This is highly undesirable, because mark bits require extra storage space, or even worse, the data structure at hand may not have such mark bits with the features. Also, when background storage has to be used because the subdivision is too large to fit in main memory, using mark bits involves not only read operations, but also write operations. Another drawback of mark bits in concurrent situations is that subdivision traversal cannot be performed by two users at the same time.

An algorithm for the traversal of *triangulated subdivisions*, or *triangulated irregular networks*, that does not require mark bits to record what triangles have been visited and which

have not, was developed by Gold et al. [47] (see also Gold and Maydell [48] and Gold and Cormack[46]). Their method involves choosing one starting point, and defining for each triangle exactly one incident edge through which the triangle can be entered. With the correct definitions and choices one can make sure that every triangle in the subdivision is reported exactly once. In fact, an order is defined on the triangles and the triangles are reported according to this order. Other work on ordering of triangles was done by De Floriani et al. [33], who used the order for visibility purposes.

In the field of computational geometry, efficient traversal algorithms have also been studied. Edelsbrunner et al. [40] showed how to traverse a directed acyclic graph without using mark bits. Avis and Fukuda [12] gave an algorithm to report the vertices of an *arrangement* or a *convex polytope* without using mark bits in the data structure that represents the arrangement or polytope. Their description is a rather abstract one and does not address non-convex subdivisions; see also Fukuda and Rosta [45]. A generic algorithm for traversing graph-like data structures without storing any information about the visited parts of the data structure was developed by Avis and Fukuda [13].

In this chapter we extend the result of Gold et al. [47] and Edelsbrunner et al. [40] to traverse subdivisions without using mark bits. Our ideas are similar to those of Avis and Fukuda [13] and of Edelsbrunner et al. [40]. Unlike their methods, our algorithm can be applied to any subdivision of which the vertices and edges form a connected graph. We prove the correctness of our algorithms for both convex and non-convex subdivisions. The algorithms are extremely simple; we implemented the algorithm for straight-line subdivisions in about 100 lines of C-code and it works fine. We also extend the results to subdivisions with curved arcs, and we give extensions for the traversal of a connected part of a subdivision, or the part of a subdivision that lies inside a specified window. These operations are commonly used by geographic information systems. One usually doesn't need the whole subdivision to be traversed, but just some subregion in which the user is interested. For instance, when a GIS user needs to know the annual rainfall for a single province of some country represented by a choropleth map, only the cells for that province need to be traversed to collect the rainfall figures. Also, when she wants to display a rectangular map of a part of the country, only the cells that intersect the rectangle need to be displayed. We also address the traversal of the surface of a convex polyhedron in three-dimensional space, and the traversal of a convex subdivision in three-dimensional space. In all cases, no mark bits are required in the data structure. Some of our results have independently been obtained by Snoeyink [105].

We present our algorithms using the *doubly-connected edge list structure* [31, 79, 85], a standard data structure used in computational geometry that stores topology explicitly (see section 1.3.1). This is not a restriction; simple adaptations to the algorithms can be made so that they apply to the quad edge structure [50], the fully topological network structure [18], the ARC/INFO structure [1], the DIME file [2], or any other vector data structure that stores the topology explicitly.

In the next section we describe the simple traversal algorithm for subdivisions embedded in the Euclidean plane $\mathbb{E}^2$ and prove its correctness. In Section 2.2.3 we show how to adapt the algorithm such that it can handle TINs and surfaces of three-dimensional polyhedra

and subdivisions with curved edges. We address the traversal of connected subsets of the cells in a subdivision and windowing queries in Section 2.2.4. Finally, in Section 2.3 we extend our results to connected subdivisions in three dimensions.

2.2 Traversing a planar subdivision

2.2.1 The local method

We assume in this section and in Section 2.2.2 that the subdivision $\mathcal{S}$ our algorithm operates on is embedded in the Euclidean plane $\mathbb{E}^2$, that its edges are straight line segments, and that the edges and vertices of $\mathcal{S}$ form a connected set. In Section 2.2.3 we will show how to overcome the first two restrictions.

The idea behind our algorithm is to define an order on the cells of the subdivision and to visit the cells in this order. Let S be the set of cells in $\mathcal{S}$. For every cell in S we define a unique predecessor, such that the predecessor relationship imposes a directed graph $G(V,E)$ on the subdivision, with $V = \{c \mid c \in S\}$ and $E = \{(c',c) \mid c' \textit{ is the predecessor of } c\}$. Our algorithm reports the cells of S in depth-first order, that is, in the order corresponding to a depth-first order in the graph G. To facilitate this, the predecessor relationship on the cells of $\mathcal{S}$ needs to be defined such that G is a tree, rooted at the node of the starting cell c_{start}, and all arcs are directed away from the root. Graph G never is explicitly determined or stored, but it is used to prove the correctness of the algorithm. The most important step of the proof is showing that G is a tree. Analogous to graph terminology, a path π in the subdivision from one cell c' to another cell c is a sequence of cells $(c' = c_0, c_1, \ldots, c_k = c)$ such that c_{j-1} is the predecessor of c_j for each $j, 1 \leq j \leq k$. To show that G is a directed rooted tree, we must show that for every cell c there is exactly one path π from c_{start} to c. The following definition will prove to be helpful (see also Figure 2.1):

Definition 1 *Let c be a cell of $\mathcal{S}$, $\vec{e}$ a half-edge incident to c, and p a point not in the interior of c. We say that $\vec{e}$ is* exposed *to p if there is a point p' on the interior of $\vec{e}$ such that the interior of the segment $\overline{pp'}$ does not intersect any half-edge of c.*

It is straightforward to verify that for any cell c of S with a point p in its exterior or on its boundary, at least one half-edge is exposed to p. We will use this to define the predecessor relationship on the cells of S. First, we choose an arbitrary point p in the starting cell c_{start}. Using p we identify for each cell c except for c_{start} a special half-edge among the ones that are exposed to p, called $entry(c)$. The cell c' incident to $twin(entry(c))$ is defined as the predecessor of c. For any cell c of S, except for c_{start}, we determine its entry as follows:

- We calculate the Euclidean distance between p and the closures of all the half-edges of c. We define the half-edge $\vec{e}$ of c that has minimum distance to be the entry of

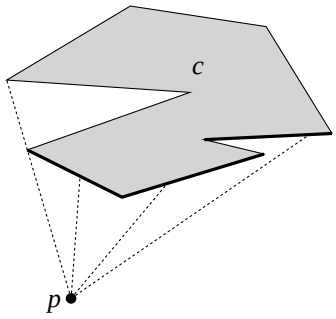


Figure 2.1: The fat half-edges of c are exposed to p, the other half-edges are not.

c. In some cases ties have to be broken. Consider the minimum radius circle C centered at p that intersects the boundary of the cell c. If C intersects the boundary of c in more than one point, we choose the first of these points on C, clockwise around p, starting in some fixed direction θ (Figure 2.2). Let this point be p'.

– If p' lies in the interior of a half-edge $\vec{e}$ incident to c, then $\vec{e}$ is the entry of c. Note that $\vec{e}$ is exposed to p.

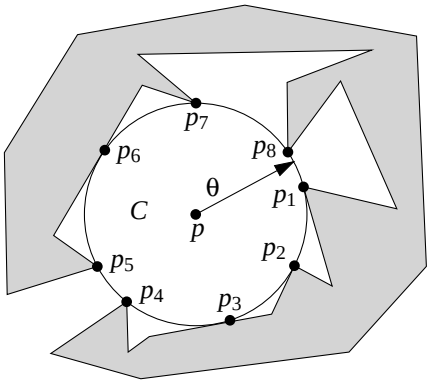


Figure 2.2: p_1 is the first point on C, starting in direction θ.

– If p' lies on a vertex v of the boundary of c, then we must choose between the two half-edges $\vec{e}$ and $\vec{e'}$ incident to c that have v as destination and source, respectively. If $\vec{e}$ is exposed to p we choose it as the entry of c; otherwise, $\vec{e'}$ is exposed to p, and we we choose it as the entry of c.

Testing whether a half-edge $\vec{e}$ is the entry of its incident cell can be done in $O(1)$ time for convex subdivisions. Note that an edge $\vec{e}$ of a convex cell is exposed to p if and only if p lies strictly to the right of the directed line induced by $\vec{e}$ (see Figure 2.3, left). Hence, some simple tests involving only p, $\vec{e}$, and the predecessor and the successor of $\vec{e}$ suffice for convex subdivisions:

- if $\vec{e}$ is not exposed to p, then it is not the entry of its incident cell.
- Otherwise, if the predecessor $\vec{e'}$ of $\vec{e}$ is exposed to p and the distance from p to $\vec{e'}$ is less than or equal to the distance from p to $\vec{e}$, then $\vec{e}$ is also not the entry of its incident cell.
- Otherwise, if the successor $\vec{e''}$ of $\vec{e}$ is exposed to p and the distance from p to $\vec{e''}$ is strictly less than the distance from p to $\vec{e}$, then $\vec{e}$ is also not the entry of its incident cell.
- Otherwise, $\vec{e}$ is the entry of its incident cell.

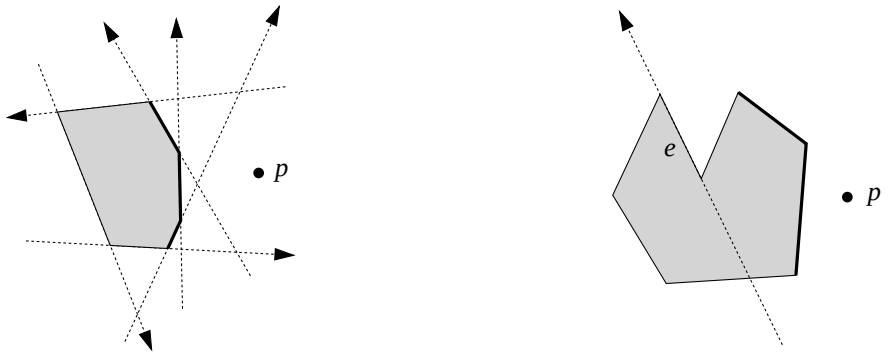


Figure 2.3: If p lies to the right of the directed line induced by a half-edge of a convex cell, then the half-edge is exposed to p. For non-convex cells, this condition is not sufficient.

For non-convex subdivisions, determining whether a half-edge $\vec{e}$ is the entry of its incident cell c involves the comparison of the distance between p and the closure of $\vec{e}$ with the distance between p and the closures of all other half-edges incident to c; this takes time linear in the number of half-edges incident to c.

Lemma 1 *With the predecessor relationship defined as above, for every cell c in S there is exactly one path π from c_{start} to c.*

Proof: It is easy to verify that there cannot be more than one path from c_{start} to a cell c, since every cell has exactly one entry. Now suppose that there exists a non-empty set $S' = \{c \mid c \in S \text{ and no path } \pi \text{ exists from } c_{start} \text{ to } c\}$. Define R' to be the set of closures

of the cells in S'. Let C be a circle centered at the point p as defined above, and let C have radius such that it intersects the boundary of R', but not its interior. C intersects the boundary of at least one cell c in S'.

Let $p' \in R'$ be the first point on C, clockwise around p, starting in direction θ. Choose one of the cells in S' that has p' on its boundary (observe that there is at least one such cell), and let this cell be c. We will show that there is a path from c_{start} to this cell c, thus deriving a contradiction, which will prove the lemma.

For the cell c, $entry(c)$ is defined as above. Observe that p' lies in the closure of $entry(c)$. If p' lies in the interior of $entry(c)$, then the predecessor c' of c, i.e. the cell incident to $twin(entry(c))$, intersects C, and a path π' from c_{start} to c' exists. Since c' is the predecessor of c, there is a path π from c_{start} to c via c' (Figure 2.4).

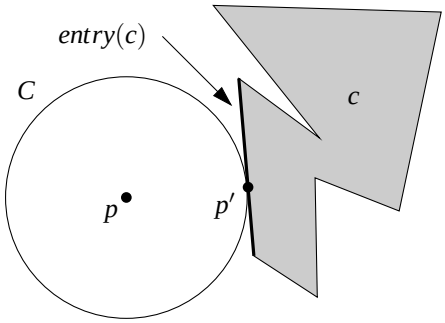


Figure 2.4: The cell incident to $twin(entry(c))$ intersects C.

On the other hand, suppose that p' is one of the endpoints of $entry(c)$. Assume p' is the destination vertex of $entry(c)$; the case where p' is the source vertex of $entry(c)$ is analogous. Let $\vec{\ell}$ be the directed line tangent to C at p' and with p to its right. Consider all half-edges that have p' as their destination and lie to the left of $\vec{\ell}$ (or on the line), as illustrated in Figure 2.5. Let these half-edges be $\vec{e_1}, \ldots, \vec{e_k} = entry(c), \vec{e_{k+1}}, \ldots$, in cyclic order as in Figure 2.5. The cell incident to $\vec{e_i}$ is labeled c_i for $1 \leq i \leq k$, and the cell incident to $twin(\vec{e_1})$ is labeled c_0. Observe that c_{i-1} is incident to $twin(\vec{e_i})$ for $1 \leq i \leq k$. Also note that a single cell can have more than one label, as illustrated in Figure 2.6. Using induction on i, we will show that there is a path π_k from c_{start} to c_k. Since $c_k = c \in S'$, this again leads to a contradiction.

- Since no edges lie between $\vec{e_1}$ and the tangent to C, c_0 must intersect C. It follows that there is a path π_0 from c_{start} to c_0.
- Assume there is a path π_i from c_{start} to all cells $c_0, c_1, \ldots, c_i$ for some $0 \leq i < k$. We can distinguish three cases, which together cover all possibilities:

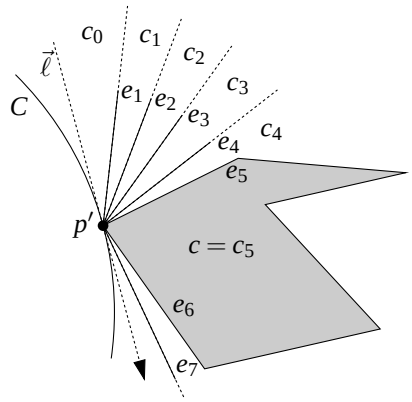


Figure 2.5: Half-edges having p' as their destination.

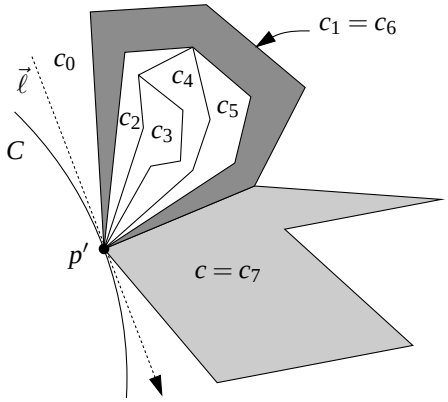


Figure 2.6: Cells can have multiple labels.

– If $c_{i+1} \notin S'$ then by definition of S' there is a path from c_{start} to c_{i+1} (note that it is possible for a cell incident to p' to intersect C even if its two half-edges incident to p do not intersect C, as is illustrated in Figure 2.7). Otherwise, If $c_{i+1} \in S'$, we can conclude by definition of C that the cell c_{i+1} doesn't intersect the interior of C, nor can its boundary have a point on C before p', starting clockwise from θ Figure 2.8.

– c_{i+1} has another label c_j, with $j < i$, as illustrated in Figure 2.6. In this case, there is a path $\pi_{i+1} = \pi_j$ from c_{start} to $c_j = c_{i+1}$ by the induction hypothesis.

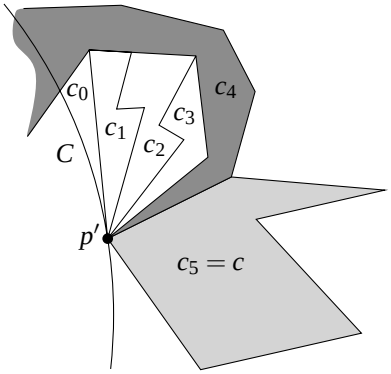


Figure 2.7: Cells intersecting the circle C are reachable from c_{start}.

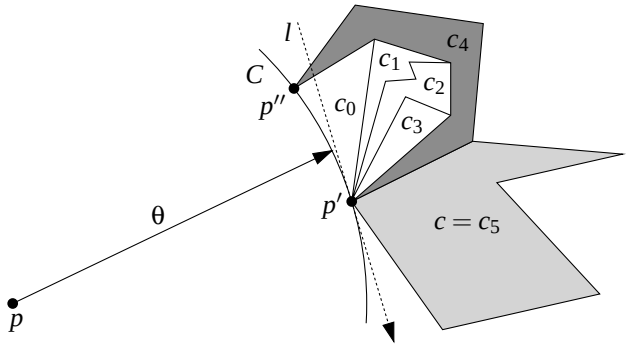


Figure 2.8: By choice of p', $entry(c_4)$ is incident to p'.

– $c_{i+1} \in S'$ and it has no other label c_j with $j < i$. In this case it's straightforward to verify that $entry(c_{i+1}) = \vec{e}_{i+1}$: C is the smallest circle centered at p intersecting the boundary of c_{i+1}, and p' is the first point on C and the boundary of c_{i+1} that is encountered when we rotate clockwise around p, starting in direction θ. Also note that since e_k (the entry of c_k) is exposed to p, $\vec{e}_{i+1}$ is also exposed to p.
  Since there is a path π_i from c_{start} to cell c_i by the induction hypothesis and c_i is the predecessor of c_{i+1}, there is a path π_{i+1} from c_{start} to c_{i+1} via c_i.

Since in all cases there is a path to c_{i+1}, we conclude by induction that there is a path to $c_k = c$. □

2.2.2 The algorithm

Using the methods of the previous section for finding the entry of a cell c, we can develop a very simple algorithm that traverses a subdivision S in a depth-first manner without using mark bits or a stack and which reports all cells of S. We assume that the subdivision is stored as a doubly-connected edge list; see Section 1.3.1 for the notation.

Algorithm *Traverse*(S, c)
Input: A planar subdivision $\mathcal{S}$ of which the edges and vertices form a connected set, and a bounded starting cell $c \in S$

```
1.  e ← Outer-incident-edge[c]
2.  Report Incident-cell[e]
3.  repeat if e = entry(Incident-cell[e])
4.        then e ← next[twin[e]] (* return from cell *)
5.        else if twin[e] = entry(Incident-cell[twin[e]])
6.                 then Report Incident-cell[twin[e]]
7.                      e ← next[twin[e]] (* explore new cell *)
8.                 else e ← next[e] (* next half-edge along current cell *)
9.  until e =Outer-incident-edge[c]
```

Of every half-edge of the counterclockwise cycle of edges around a cell c in S, algorithm *Traverse* inspects the corresponding twin half-edge, incident to a neighboring cell. If this half-edge is the entry of the neighboring cell, the algorithm continues in a depth-first manner in this cell after reporting it. Note that no stack or other memory resources are needed. Since we start with the successor of the entry-edge of a cell the algorithm is finished with the counterclockwise cycle of edges of that cell when it encounters the entry of the cell (line 3). Figure 2.9 shows some snapshots of a run the algorithm.

Let n be the number of edges in the subdivision. Since the subdivision is planar, both the number of vertices and cells are $O(n)$. In Algorithm *Traverse* the function *entry* is called at most $4n$ times, namely at most twice for each half-edge. For convex subdivisions we can determine in $O(1)$ time whether a half-edge is the entry of its incident cell; it follows that the running time of Algorithm *Traverse* is $O(n)$ for convex subdivisions. For non-convex subdivisions determining whether a half-edge is the entry of its incident cell takes time linear in the number of edges of that cell. If the cells have constant complexity, then the running time of our algorithm is $O(n)$; if the cells are non-convex polygons of which the complexity is not bounded by a constant, then the running time is bounded by the sum of the squares of the complexities of all cells, which is $O(n^2)$ in the worst case.

Theorem 1 *Algorithm Traverse reports all cells, vertices and edges of a connected subdivision $\mathcal{S}$ with n edges without using mark bits or a stack. The running time is $O(n)$ if all cells are convex or if the complexity of each cell is bounded by a constant. Otherwise, the running time is $O(n^2)$.*

Algorithm *Traverse* can easily be adapted to report the edges and the vertices of the subdivision as well: when a cell is reported in line 6 of Algorithm *Traverse*, we list all its

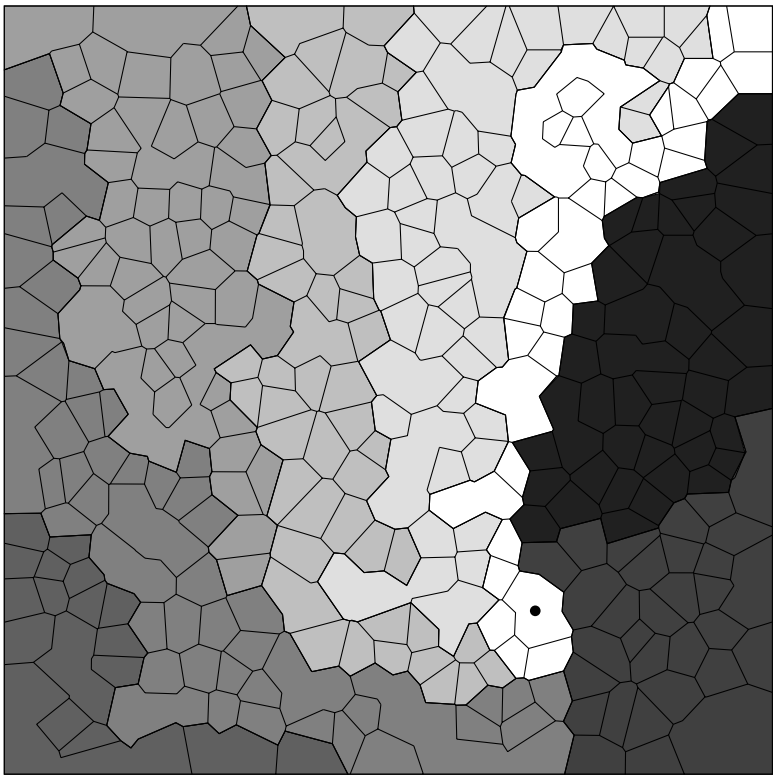


Figure 2.9: Snapshots of a run of the traversal algorithm. The traversal starts in the cell with the dot, and lighter cells are visited before darker cells.

half-edges. To prevent a half-edge pair from being reported twice, we only report the one that has its direction in the range $[0,\pi)$. Of every half-edge $\vec{e}$ that is reported, we report its source vertex v if and only if $\vec{e}$ is the first half-edge with v as the source, in a cyclic order starting in some fixed direction θ. These tests can be performed in constant time using standard DCEL-operations. The asymptotic running time of Algorithm *Traverse* is not affected by these adaptations.

2.2.3 Overcoming the restrictions

In Sections 2.2.1 and 2.2.2 we made the assumptions that the subdivision our algorithm operates on is embedded in $\mathbb{E}^2$ and that its edges are straight line segments. In this section we will show how to adapt the algorithm such that it can handle polyhedral terrains represented by TINs, surfaces of three-dimensional convex polyhedra, and subdivisions with curved arcs.

Adapting our algorithm so that it can handle a polyhedral terrain is quite simple, if it is represented by a *Triangulated Irregular Network* (TIN). A terrain is a two-dimensional surface in three-dimensional space, with the special property that every vertical line intersects it in a point, or not at all. This means that a point p with coordinates (x,y,z) on the terrain can be mapped to a point $p' \in \mathbb{E}^2$ with coordinates (x,y), and that the mapping of all points of the terrain to $\mathbb{E}^2$ is injective. Calculating the *entry* of a triangle in a polyhedral terrain can be done with the method of Section 2.2.1 if we project every edge and vertex of the triangles on $\mathbb{E}^2$ when it is examined; Algorithm *Traverse* needs no further adaptations. Since every projection takes $O(1)$ time, the asymptotic running time of the algorithm is not affected by the projections. For TINs our algorithm is similar to the algorithm of Gold and Cormack [46].

Corollary 1 *Algorithm Traverse can be adapted to report all n triangles of a TIN in $O(n)$ time without using mark bits or a stack.*

Boundaries of three-dimensional convex polyhedra can be dealt with in the same way, although mapping the vertices and edges to $\mathbb{E}^2$ requires a little more effort for a convex polyhedron than for a polyhedral terrain (see Figure 2.10). We assume that that the polyhedron is stored as a topological datastructure as described by Dobkin and Laszlo [38], such that we can find adjacent faces, edges and vertices efficiently. To traverse and report these features, we project them on a single face f of the polyhedron as follows: we determine a point p such that for all vertices v of the polyhedron, except those incident to f, the line segment between p and v intersects f. This intersection is the projection of v. Projections of edges are determined by projecting their incident vertices.

Finding a suitable point p involves taking a face f of the polyhedron, choosing a point in the interior of f, and translating it along the normal of f to the outside of the polyhedron. The faces incident to the *twin*-edges of the half-edges bounding f determine how little we should translate the point to the outside of the polyhedron. Testing this takes time linear in the complexity of f. After that, projections take O(1) time for each vertex and edge of the polyhedron, and we obtain a planar convex subdivision of f.

We now can run Algorithm *Traverse* on the polyhedron; determining whether an edge is the entry of a face or not is done by performing the calculations described in Section 2.2.1 on the projected version of the edge. Note that we don't need to project the whole subdivision in advance; it suffices to do the projections "on the fly". Again, the asymptotic running time of the algorithm is not affected by the projections.

Corollary 2 *Algorithm Traverse can be adapted to report all faces, edges, and vertices of a three-dimensional convex polyhedron with n faces in $O(n)$ time without using mark bits or a stack.*

Algorithm *Traverse* can also be used to report the cells, arcs, and vertices of subdivisions with curved arcs instead of straight edges, provided that the arcs have constant description size and that we can calculate the minimum distance from a point p to an arc. We assume

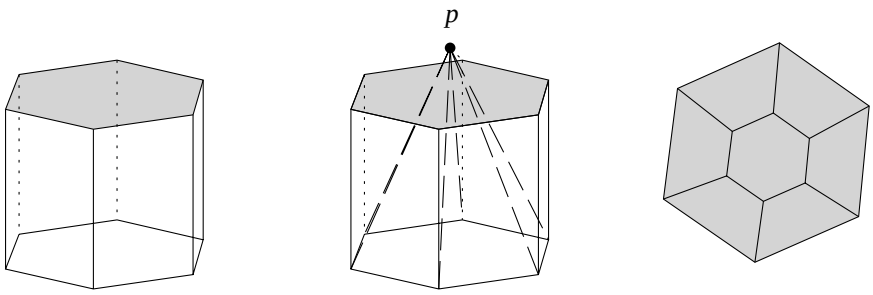


Figure 2.10: Projecting the edges and vertices of a convex polyhedron to one of its faces.

that the subdivision is stored as a doubly-connected edge list again, but with half-arcs instead of half-edges. The entry of a cell is defined almost the same as before: determine the point p' on the boundary of the cell that has minimum distance to the fixed point p in the starting cell. If p' is not uniquely defined, choose the one that is encountered first when we rotate around p in clockwise direction, starting in some fixed direction θ. If p lies on the interior of a half-arc $\vec{a}$, then $\vec{a}$ is the entry of the cell. Otherwise, if p coincides with a vertex v of the cell that is the destination of $\vec{a}$ and the origin of $\vec{a}'$, we have to choose between $\vec{a}$ and $\vec{a}'$. To this end, we need to adapt the notion of *exposed*. Intuitively, a curved half-arc $\vec{a}$ is exposed to a point p with respect to vertex v if close to v, the arc a has the cell to the one side and faces the point p to the other side. More formally, $\vec{a}$ is exposed to p with respect to v if for all sufficiently small positive real values ε there is a point $q \neq v$ on $\vec{a}$ in an ε-neighborhood of v such that the interior of the segment $\overline{pq}$ does not intersect the cell c incident to $\vec{a}$ in that ε-neighborhood of v (see Figure 2.11). If both half-arcs $\vec{a}$ and $\vec{a}'$ are exposed to p with respect to v, we choose $\vec{a}$, the one that has v as its destination, to be the entry of the cell. Otherwise, at least one of $\vec{a}$ and $\vec{a}'$ is exposed to p with respect to v, and we choose the one that is exposed.

Corollary 3 *Algorithm Traverse can be adapted to report all cells, arcs, and vertices of a subdivision with n curved arcs of constant description size without using mark bits or a stack. If the complexity of each cell is bounded by a constant, then the running time is $O(n)$; otherwise, the running time is $O(n^2)$.*

2.2.4 Related queries

Sometimes we don't want the whole subdivision $\mathcal{S}$ to be reported, but just some connected subset $\mathcal{S}' \subseteq \mathcal{S}$, such that all cells in $\mathcal{S}'$ have the same attribute as the starting cell c_{start} (Figure 2.12). For example, suppose that the starting cell lies in a forest; we then may ask to report all cells that lie in the same forest. We will show how to adapt Algorithm *Traverse* such that these queries can answered efficiently as well.

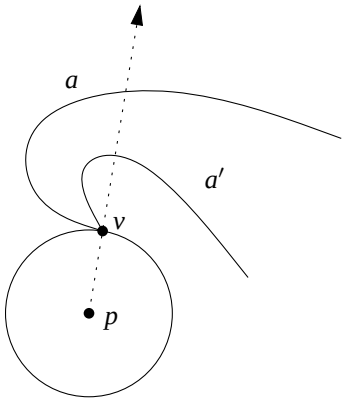


Figure 2.11: Arc a is exposed to p with respect to v, whereas a' is not.

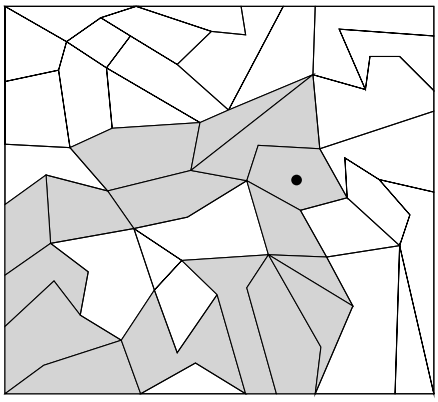


Figure 2.12: A connected set of cells with the same attribute as c_{start} (the cell containing the dot).

As Figure 2.12 indicates, the connected subset $\mathcal{S}'$ of $\mathcal{S}$ can contain holes consisting of cells that do not have the same attribute as the starting cell c_{start}. Some of the cells in these holes can be predecessors of cells in $\mathcal{S}'$. However, we don't want to visit cells or half-edges in the holes. Instead, we rather consider each hole as a single cell. To do this, we need a way to traverse the counterclockwise cycle of half-edges that bound each hole, without ever visiting half-edges $\vec{e}$ for which neither $\vec{e}$ nor $twin(\vec{e})$ bounds a cell of $\mathcal{S}'$. If we can achieve this, we can express the running time of the traversal algorithm in the combinatorial complexity of $\mathcal{S}'$, rather than in the complexity of the whole subdivision $\mathcal{S}$. To treat a hole as one single cell we do the following (Figure 2.13): suppose that

half-edge $\vec{e}$ with destination vertex v is a half-edge of the boundary of a hole. We find its successor in the counterclockwise cycle of half-edges that bound this hole by inspecting all outgoing half-edges of v in counterclockwise order, starting with $twin(\vec{e})$, until we find an edge $\vec{e}'$ that is incident to the hole. This half-edge $\vec{e}'$ is the successor of $\vec{e}$. In this way, we can treat the holes as ordinary cells of the subdivision, except that they are not reported. Notice that the cells of $\mathcal{S} \setminus \mathcal{S}'$, not enclosed by $\mathcal{S}'$, are handled correctly also. The algorithm won't even notice the difference between a hole and these outer cells.

What is the effect on the running time of Algorithm *Traverse*? Let k be the number of edges in $\mathcal{S}'$. Then the total number of edges on the boundary of all holes is at most $O(k)$. Each boundary edge of each hole is tested once for being the entry of the hole. Testing an edge of a hole involves traversing all edges that bound that hole. If we do this as described above, this takes time linear in the number of edges in $\mathcal{S}'$ of which the source vertex lies on the boundary of the hole; this number is $O(k)$ for all holes together. Testing each hole once takes $O(k)$ time; since each hole is tested at most $O(k)$ times, all these tests together take at most $O(k^2)$ time. If we combine this with the analysis in Section 2.2.2 we derive a running time of $O(k^2)$: the running time depends only on the complexity of the reported cells and not on the complexity of the whole subdivision. Unfortunately, the running time is also $O(k^2)$ if all cells are convex or if the complexity of each cell is bounded by a constant, because the holes can be non-convex.

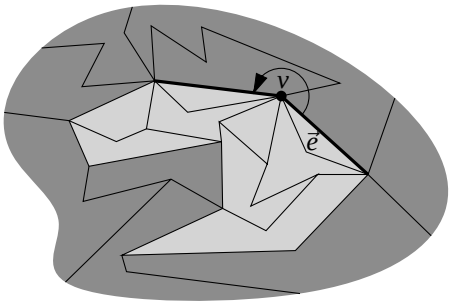


Figure 2.13: Finding the next edge of the (lightly shaded) hole.

Note that we cannot treat holes and outer cells consisting of cells in $\mathcal{S} \setminus \mathcal{S}'$ as single cells if $\mathcal{S}'$ is connected but not edge-connected, as depicted in Figure 2.14: we cannot go from cell c to cell c' by inspecting the outgoing half-edges of v in counterclockwise order.

Theorem 2 *Algorithm Traverse can be adapted to report all cells, vertices, and edges of an edge-connected subdivision $\mathcal{S}' \subseteq \mathcal{S}$ without using mark bits or a stack. The running time is $O(k^2)$, where k is the number of edges in $\mathcal{S}'$.*

Another query that arises often in practice is "given a subdivision S and a rectangular window W, report all cells in S that intersect W" (Figure 2.15).

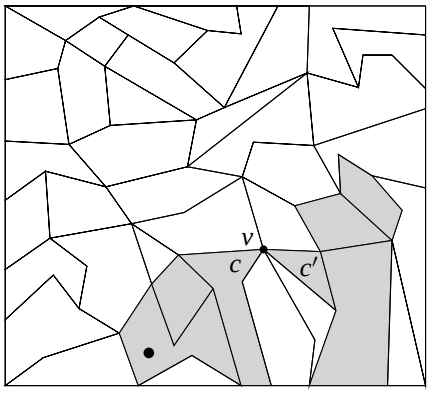


Figure 2.14: The shaded cells form a connected subdivision, but this subdivision is not edge-connected.

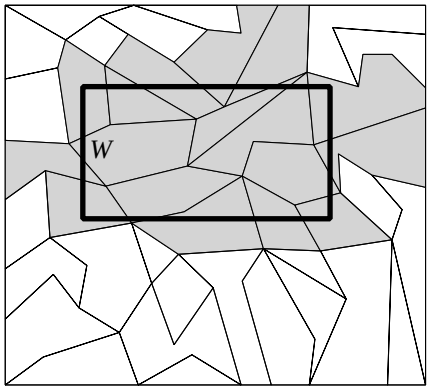


Figure 2.15: Reporting all cells that intersect a rectangular window.

We solve this query as follows. Normally when we test a half-edge e for being the entry, we traverse the cycle of half-edges around c, keeping track of the edge that has minimum distance to a predefined point p in the starting cell c (breaking ties as described in Section 2.2.1). We make an adaptation here: before calculating the distance of an edge to the point p, we clip the edge to the window W and perform our calculations on the clipped version of the edge (Figure 2.16). Edges that don't intersect W disappear; we consider their distance to p to be infinite. We also make a small adaptation in Algorithm *Traverse*: in line 5 we only test $twin[\vec{e}]$ for being the entry of its incident cell if $twin[\vec{e}]$ intersects the window W: if it doesn't it can't be the entry of its incident cell anyway, and omitting the test prohibits cells that don't intersect W to influence the running time of the algorithm.

It is straightforward to verify that with these adaptations our algorithm is still correct. Clipping the edges has the effect that cells that intersect the window W possibly fall apart into two or more pieces. Each of these pieces has a well-defined entry, and the piece of which the entry has the minimum distance to p determines the entry of the original (unclipped) cell.

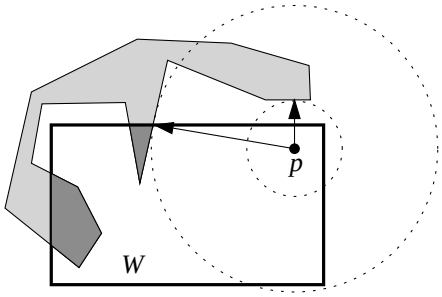


Figure 2.16: Clipping the edges of a cell to the window.

Clipping to the window W can be done in $O(1)$ time for each half-edge. Convex cells can have at most one piece inside a rectangular window; testing a half-edge of a convex cell for being the entry involves only clipping and comparing its distance to p with the distances of its predecessor and successor to p. Combining this with the analysis in Section 2.2.2, we derive the following result:

Theorem 3 *Algorithm Traverse can be adapted to report all cells, edges, and vertices of a connected planar subdivision $\mathcal{S}$ with n edges that lie in a rectangular query window W without using mark bits or a stack. If all cells of $\mathcal{S}$ are convex, or if the complexity of each cell is bounded by a constant, then the running time is $O(k)$, where k is the total complexity of the cells that intersect W; otherwise, if the cells are non-convex polygons of which the complexity is not bounded by a constant, then the running time is bounded by $O(k^2)$.*

2.3 Extension to three dimensions

Extending our algorithm such that it traverses convex subdivisions in three dimensions and reports all (three-dimensional) cells, (two-dimensional) faces, edges, and vertices is straightforward. Dobkin and Laszlo [38] have developed data structures and operations for handling three-dimensional subdivisions; these are comparable with the DCEL data structure and operations that we used in the two-dimensional setting.

Again, we need to determine for each cell in the subdivision which one of its neighbors is its predecessor in the traversal. The entry of a cell is that face of the cell that is incident to

the predecessor of the cell. Once we are able to determine for each cell which of its faces is its entry, we can apply the simple scheme of Algorithm *Traverse* again: we enter a cell c through its entry, and traverse all its faces (which we can do with the method described in Section 2.2.3, since the cells are convex). During this traversal of the faces, we test every face on being the entry of the other incident cell. If this is the case we continue in a depth-first manner on this cell, again without using a stack, as described in Section 2.2.2. After returning from this cell, we proceed with the next face of the cell c, until we are back at the entry of c again; then we return to the predecessor of c.

Determining the entry of a cell c of a three-dimensional subdivision is analogous to the two-dimensional case described in Section 2.2.1: we choose an arbitrary point p in the starting cell c_{start}, and define some face f that has minimum distance to p to be the entry of c, for any cell c except for the starting cell. Let p' be the point in the boundary of c that realizes the minimum distance. If p' is not uniquely determined, the candidates for being p' lie on a sphere centered at p; we choose the candidate with the highest z-coordinate. If this still doesn't uniquely determine p', the remaining candidates lie on a circle; as in the two-dimensional case, we choose the one that is encountered first if we rotate around the center of this circle, starting in some fixed direction θ.

If p' lies in the interior of a face f of c, then f is the entry of c. If p' lies on an edge or a vertex of c, we have to make a choice between the faces of c that are exposed to p. We choose the one that has the smallest angle with the plane that is tangent to the sphere C in the point p'. Ties can be broken in various ways, as long as it is done consistently.

2.4 Conclusions and further research

We have developed a simple, flexible, and efficient algorithm for traversing various kinds of planar subdivisions without using mark bits in the structure or a stack. Our algorithm reports each cell, edge, and vertex exactly once. The algorithm can handle subdivisions embedded in the Euclidean plane $\mathbb{E}^2$, as well as polyhedral terrains, and boundaries of convex three-dimensional polyhedra. It can easily be adapted to report a connected subset of the cells in the subdivision, or to answer windowing queries; both adaptations result in an output sensitive algorithm. Extending the algorithm to handle convex subdivisions in three dimensions is straightforward. An implementation of the algorithm for planar subdivisions with straight edges took about 100 lines of C-code.

A number of problems remains to be solved. We have looked at non-connected subdivisions, that is, subdivisions of which the edge and vertex set is unconnected (Figure 1.10), but at this moment it is unclear if these can be handled without making use of a stack or other memory resources to keep track of the components in the subdivision that have been visited.

Also unsolved is the problem of traversing the boundary of non-convex polyhedra; although these are topologically equivalent to a sphere, which means that they can be projected to $\mathbb{E}^2$, there is no way to determine the projection of a vertex of the boundary of a

non-convex polyhedron if we are only allowed to use local information: we need to know the geometry of the whole polyhedron.

Since traversing non-convex three-dimensional subdivisions involves traversing the boundaries of its cells which are non-convex polyhedra, there is no use in attacking this problem before the former problem has been solved.

In many practical situations such as those arising in GISs, the cells in a subdivision represent geographical entities like countries. It may well be that two adjacent cells have long chains of edges on their common boundaries. If one of the cells is the predecessor of the other, then only one edge in the chain is the entry. It would be interesting to find an elegant way to represent chains of edges by "pseudo-edges" between vertices of degree three and higher, in order to avoid the traversal of all the edges in the chain. This, of course, would involve modifying the data structures and adapting the definition of the *entry* of a cell. We regard this problem as an interesting subject for further research.

CHAPTER 3

Contour Trees and Small Seed Sets for Isosurface Generation

3.1 Introduction

One of the functionalities of a GIS is to display the stored data by generating tables, charts, and maps, either on paper or on a computer screen. As we have seen in Chapter 1, several kinds of maps are available for displaying the different types of data. Choropleth maps are used to display categorial data, such as different types of vegetation. Network maps, such as railroad maps, show connections (railways) between geographic objects (stations); the regions on a network map are meaningless. Finally, isoline maps are very effective means for displaying scalar data defined over the plane. Such data can be visualized after interpolation by showing one or more contours: the sets of points having a specified value. For example, scalar data over the plane is used to model elevation in the landscape, and a contour is just an isoline of elevation. Contours can be used for visualizing scalar data defined over the three-dimensional space as well. In that case, the contours are two-dimensional isosurfaces. For instance, in atmospheric pressure modeling, a contour is a surface in the atmosphere where the air pressure is constant, an isobar. The use of isolines or isosurfaces for displaying scalar data is not limited to the field of GIS. In medical imaging for example, isosurfaces are used to show reconstructed data from scans of the brain or parts of the body. The scalar data can be seen as a sample of some real-valued function, which is called a terrain or elevation model in GIS, and a scalar field in imaging.

A real-valued function over a two- or three-dimensional domain can be represented in a computer using a two- or three-dimensional mesh, which can be regular (all cells have the same size and shape) or irregular. A terrain (mountain landscape) in GIS is commonly

represented by a regular square grid or an irregular triangulation. The elements of the grid, or vertices of the triangulation, have a scalar function value associated to them. The function value of non-vertex points in the two-dimensional mesh can be obtained by interpolation. An easy form of interpolation for irregular triangulations is linear interpolation over each triangle. The resulting model is known as the TIN model for terrains (Triangulated Irregular Network) in GIS; see Section 1.2.2. In computational geometry, it is known as a polyhedral terrain. More on interpolation of spatial data and references to the literature can be found in the book by Watson [124].

One can expect that the combinatorial complexity of the contours with a single function value in a mesh with n elements is roughly proportional to $\sqrt{n}$ in the two-dimensional case and to $n^{2/3}$ in the three-dimensional case [70]. Therefore, it is worthwhile to have a search structure to find the mesh elements through which the contours pass. This will be more efficient than retrieving the contours of a single function value by inspecting all mesh elements.

There are basically two approaches to find the contours more efficiently. Firstly, one could store the two-dimensional or three-dimensional domain of the mesh in a hierarchical structure and associate the minimum and maximum occurring scalar values at the subdomains to prune the search. For example, octrees have been used this way for regular three-dimensional meshes [126].

The second approach is to store the *scalar range*, also called *span*, of each of the mesh elements in a search structure. Kd-trees [70], segment trees [14], and interval trees [24, 116] have been suggested as the search structure, leading to a contour retrieval time of $O(\sqrt{n}+k)$ or $O(\log n+k)$, where n is the number of mesh elements and k is the size of the output. A problem with this approach is that the search structure can be a serious storage overhead, even though an interval tree needs only linear storage. One doesn't want to store a tree with a few hundred million intervals that would arise from regular three-dimensional meshes. It is possible to reduce the storage requirements of the search structures by observing that a whole contour can be traced directly in the mesh if one mesh element through which the contour passes is known. Such a starting element of the mesh is also called a *seed*. Instead of storing the scalar range of all mesh elements, we need only store the scalar range of the seeds as intervals in the tree, and a pointer into the mesh, or an index, if a (two- or three-dimensional) array is used. Of course, the seed set must be such that every possible contour of the function passes through at least one seed. Otherwise, contours could be missed. There are a few papers that take this approach [14, 59, 116]. The tracing algorithms to extract a contour from a given seed have been developed before, and they require time linear in the size of the output [7, 57, 59].

The objective of this chapter is to present new methods for seed set computation. To construct a seed set of small size, we use a variation of the *contour tree*, a tree that captures the contour topology of the function represented by the mesh. It has been used before in image processing and GIS research [44, 46, 64, 104, 110]. Another name in use is the *topographic change tree*, and it is related to the *Reeb graph* used in Morse Theory [89, 101, 102, 110]. It can be computed in $O(n\log n)$ time for functions over a two-dimensional domain [30].

This chapter includes the following results:

- We present a new, simple algorithm that constructs the contour tree. For two-dimensional meshes with n elements, it runs in $O(n \log n)$ time like a previous algorithm [30], but the new method is much simpler and needs less additional storage. For meshes with n faces in d-dimensional space, it runs in $O(n^2)$ time. In typical cases, less than linear temporary storage is needed during the construction, which is important in practice. Also, the higher-dimensional algorithm requires subquadratic time in typical cases.
- We show that $\Omega(n \log n)$ is a lower bound for the construction of the contour tree.
- We show that the contour tree is the appropriate structure to use when selecting seed sets. We give an $O(n^2 \log n)$ time algorithm for seed sets of minimum size by using minimum cost flow in a directed acyclic graph [4].
- In practice one would like a close to linear time algorithm when computing seed sets. We give a simple algorithm that requires $O(n \log^2 n)$ time and linear storage after construction of the contour tree, and gives seed sets of small size.
- The approximation algorithm has been implemented, and we supply test results of various kinds.

Previous methods to find seed sets of small size didn't give any guarantee on their size [14, 59, 116]. Shortly after the results of this chapter were published, Tarasov and Vyalyi [111] extended our contour tree construction algorithm and obtained an $O(n \log n)$ time algorithm for the three-dimensional case. Their algorithm consists of a preprocessing step with two sweeps, after which our algorithm is used.

3.2 Preliminaries on scalar functions and the contour tree

In this section we provide background and definitions of terms used in the following sections. On a continuous function $\mathcal{F}$ from d-space to the reals, the *criticalities* can be identified. These are the local maxima, the local minima, and the saddles (or passes). If we consider all contours of a specified function value, we have a collection of lower-dimensional regions in d-space (typically, $(d-1)$-dimensional surfaces of arbitrary topology). If we let the function value take on the values from $+\infty$ to $-\infty$, a number of things may happen to the contours. Contour shapes deform continuously, with changes in topology only when a criticality is met (i.e., its function value is passed). A new contour component starts to form whenever the function value is equivalent to a locally maximal value of $\mathcal{F}$. An existing contour component disappears whenever the function value is equivalent to a locally minimal value.

At saddle points, various different things can happen. It may be that two (or more) contour components adjoin, or one contour component splits into two (or more) components, or

that a contour component gets a different topological structure (e.g., from a sphere to a torus in three dimensions). The changes that can occur have been documented in texts on Morse theory and differential topology [56, 76]. They can be described by a structure called the contour tree, which we describe below.

As an example, consider a function modeled by a two-dimensional triangular mesh with linear interpolation and consider how the contour tree relates to such meshes. For simplicity, we assume that all vertices have a different function value. If we draw the contours of all vertices of the mesh, then we get a subdivision of the two-dimensional domain into regions. All saddle points, local minima and maxima must be vertices of the mesh in our setting. The contour through a local minimum or maximum is simply the point itself. One can show that every region between contours is bounded by exactly two contours [30].

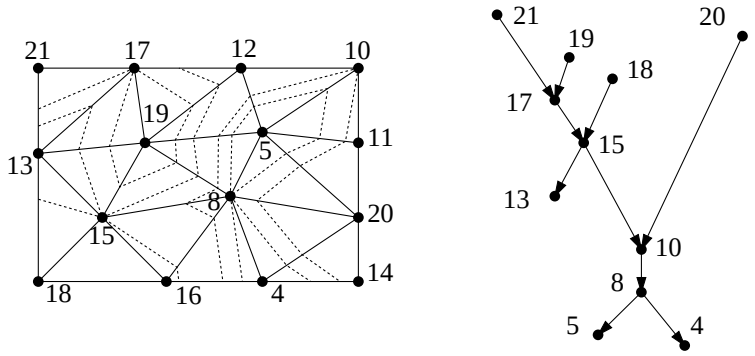


Figure 3.1: Two-dimensional triangular mesh with the contours of the saddles, and the contour tree.

We let every contour in this subdivision correspond to a node in a graph, and two nodes are connected if there is a region bounded by their corresponding contours. This graph is a tree, which is easy to show [30, 116], and it is called the contour tree. All nodes in the tree have degree 1 (corresponding to local extrema), degree 2 (normal vertices), or at least 3 (saddles). In other words, every contour of a saddle vertex splits the domain into at least three regions. For each vertex in the triangulation, one can test locally whether it is a saddle. This is the case if and only if it has neighboring vertices around it that are higher, lower, higher, and lower, in cyclic order around it. If one would take the approach outlined above to construct the contour tree, $\Omega(n^2)$ time may be necessary in the worst case, because the total combinatorial complexity of all contours through saddles may be quadratic. An $O(n \log n)$ time divide-and-conquer algorithm exists, however [30].

In a general framework, we define the contour tree with only few assumptions on the type of mesh, form of interpolation, and dimension of the space over which function $\mathcal{F}$ is defined. The input data is assumed to be:

- a mesh M of size n embedded in $\mathbb{R}^d$;
- a continuous real-valued function $\mathcal{F}$ defined over each cell of M.

A *contour* is defined to be a maximal connected piece of $\mathbb{R}^d$ where the function value is the same. Usually, a contour is a $(d-1)$-dimensional hypersurface, but it can also be lower dimensional or d-dimensional. We define the contour tree $\mathcal{T}$ as follows.

- Take each contour that contains a criticality.
- These contours correspond to the *supernodes* of $\mathcal{T}$ (the tree will be extended later with additional nodes, hence we use the term supernodes here). Each supernode is labeled with the function value of its contour.
- For each region bounded by two contours, we add a superarc between the corresponding supernodes in $\mathcal{T}$.

The contour tree is well defined, because each region is bounded by two and only two contours which correspond to supernodes. In fact, it is easy to see that the contour tree is a special case of the more general Reeb graph in the $(d+1)$-dimensional space obtained from the domain (the mesh) extended with the function image space [89, 101, 102, 110]. Furthermore, one can show that the contour tree is indeed a tree: the proof for the two-dimensional case given in [30] can easily be extended to d dimensions.

For two-dimensional meshes, all criticalities correspond to supernodes of degree 1, or degree 3 or higher. For higher-dimensional meshes there are also criticalities that correspond to a supernode of degree 2. This occurs for instance in three dimensions when the genus of a surface changes, for instance when the surface of a ball changes topologically to a torus (Figure 3.2(b)).

Superarcs are directed from higher scalar values to lower scalar values. Thus, supernodes corresponding to the local maxima are the sources and the supernodes corresponding to the local minima are the sinks.

To be able to compute the contour tree, we make the following assumptions:

- Inside any face of any dimension of M, all criticalities and their function values can be determined.
- Inside any face of any dimension of M, the range (min, max) of the function values taken inside the face can be determined.
- Inside any face of any dimension of M, the (piece of) contour of any value in that face can be determined.

We assume that in facets and edges of two-dimensional meshes, the items listed above can be computed in $O(1)$ time. For vertices, we assume that the first item takes time linear in its degree. Similarly, in three-dimensional meshes we assume that these items take $O(1)$ to compute in cells and on facets, and time linear in the degree on edges and at vertices.

In d-dimensional space (for $d > 2$), a saddle point p is a point such that for any sufficiently small hypersphere around p, the contour of p's value intersects the surface of the hypersphere in at least two separate connected components.

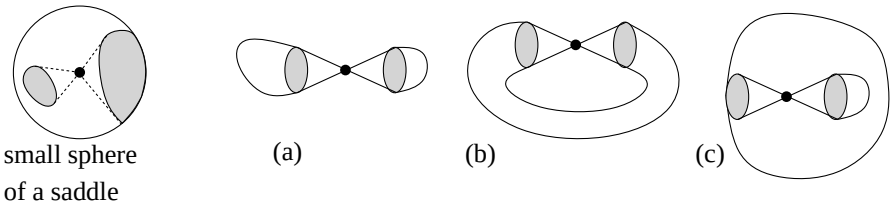


Figure 3.2: Criticalities in the three-dimensional case.

Possible criticalities in the three-dimensional case are shown in Figure 3.2. When sweeping the function value from ∞ to $-\infty$, they correspond to (a) two contours merging or splitting, but not containing the other, (b) an increment or decrement of the genus of one contour surface, and (c) two contours merging or splitting, and one containing the other. More cases can be distinguished when a criticality causes several of these changes to occur at once, or when the contour ends at the boundary of the mesh.

3.3 Contour tree algorithms

In this section we assume for ease of presentation that the mesh M is a simplicial decomposition with n cells, and that linear interpolation is used. As a consequence, all critical points are vertices of the mesh M. Instead of computing the contour tree as defined in the previous section, we compute an extension that includes nodes for the contours of all vertices of M, also the non-critical ones. So supernodes correspond to contours of critical vertices and regular nodes correspond to contours of other vertices. Each superarc is now a sequence of arcs and nodes, starting and ending at a supernode. The algorithm we'll describe next can easily be adapted to determine the contour tree with only the supernodes. But we'll need this extended contour tree for seed selection in the next section. From now on, we call the contour tree with nodes for the contours of all vertices the contour tree $\mathcal{T}$.

The supernodes of $\mathcal{T}$ that have in-degree 1 and out-degree greater than 1 are called *bifurcations*, and the supernodes with in-degree greater than 1 and out-degree 1 are called *junctions*. All normal nodes have in-degree 1 and out-degree 1. We'll assume that all bifurcations and junctions have degree exactly 3, that is, out-degree 2 for bifurcations and in-degree 2 for junctions. This assumption can be removed; one can represent all supernodes with higher degrees as clusters of supernodes with degree 3. For example, a supernode with in-degree 2 and out-degree 2 can be treated as a junction and a bifurcation, with a directed arc from the junction to the bifurcation. The assumption that all junctions and bifurcations have degree 3 facilitates the following descriptions considerably.

3.3.1 The general approach

To construct the contour tree $\mathcal{T}$ for a given mesh in d-space, we let the function value take on the values from $+\infty$ to $-\infty$ and we keep track of the contours for these values. In other words, we sweep the scalar value (see Section 1.3.2). For two-dimensional meshes, one can image sweeping a polyhedral terrain embedded in a three-dimensional space and moving downward a horizontal plane. The sweep stops at certain event points: the vertices of the mesh. During the sweep, we keep track of the contours in the mesh at the value of the sweep function, and the set of cells of the mesh that cross these contours. The cells that contain a point with value equivalent to the present function value are called *active*. The tree $\mathcal{T}$ under construction during the sweep will be growing at the bottom at several places simultaneously, see Figure 3.3.

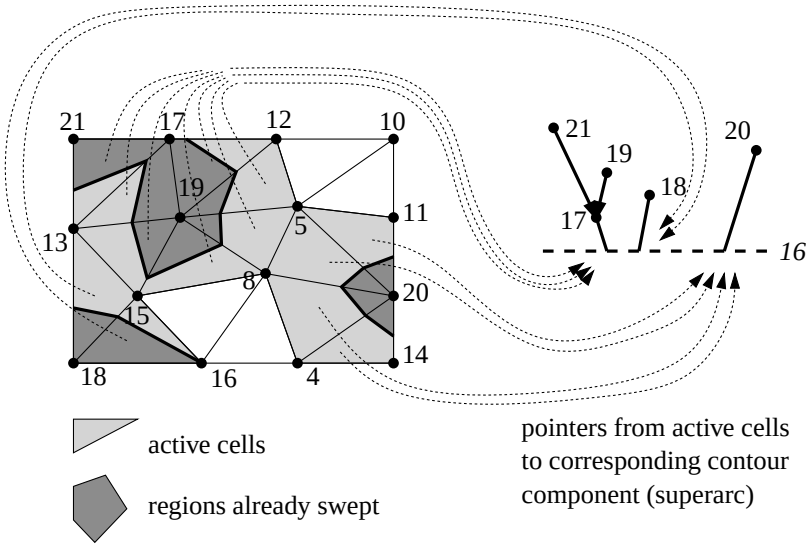


Figure 3.3: Situation of the sweep of a two-dimensional mesh when the function value is 16.

Each part of $\mathcal{T}$ that is still growing corresponds to a unique contour at the current sweep value. We group the cells into contours by storing a pointer at each active cell in the mesh to the corresponding superarc in $\mathcal{T}$. The contours can only change structurally at the event points, and the possible changes are the following:

- At a local maximum of the mesh (more precisely: of the function), a new contour appears. This is reflected in $\mathcal{T}$ by creating a new supernode and a new arc incident to it. This arc is also the start of a new superarc, which will be represented. Each cell incident to the maximum becomes active, and we set their pointer to the new superarc of $\mathcal{T}$. At this stage of the algorithm, the new superarc has no lower node attached to it yet.

- At a local minimum of the mesh, a contour disappears; a new supernode of $\mathcal{T}$ is created, and the arc corresponding to the disappearing contour at the current value of the sweep is attached to the new supernode. It is also the end of a superarc. The cells of the mesh incident to the local minimum are no longer active.

- At a non-critical vertex of the mesh, a new node of $\mathcal{T}$ is created, the arc corresponding to the contour containing the vertex is made incident to the node, and a new arc incident to the node is created. There is no new superarc. Some cells incident to the vertex stop being active, while others start being active. The pointers of the latter cells are set to the current superarc of the contour. For the cells that remain active, nothing changes: their pointer keeps pointing to the same superarc.

- At a saddle of the mesh, there is some change in topology in the collection of contours. It may be that two or more contours merge into one, one contour splits into two or more, or one contour changes its topological structure. A combination of these is also possible in general. The first thing to do is to determine what type of saddle we are dealing with. This can be decided by traversing the whole contour on which the saddle lies.

  If two contours merge, a new supernode (junction) is created in $\mathcal{T}$ for the saddle, and the superarcs corresponding to the two merging contours are made incident to this supernode. Furthermore, a new arc and superarc are created for the contour that results from the merge. The new arc is attached to the new supernode. All cells that are active in the contour after the merge set their pointer to the new superarc in $\mathcal{T}$. If a contour splits, then similar actions are taken.

  If the saddle is because of a change in topology of one single contour (i.e., an increase or decrease of its genus by one), a new supernode is made for one existing superarc, and a new arc and superarc are created in $\mathcal{T}$. All active cells of the contour set their pointers to the new superarc.

For the sweep algorithm, we need an event queue and a status structure. The event queue is implemented with a standard heap structure so insertions and extractions take logarithmic time per operation. The status structure is implicitly present in the mesh with the additional pointers from the cells to the superarcs in the contour tree.

Theorem 4 *Let M be a mesh in d-space with n faces in total, representing a continuous, piecewise linear function over the mesh elements. The contour tree of M can be constructed in $O(n^2)$ time and $O(n)$ storage.*

Proof: The algorithm clearly takes time $O(n \log n)$ for all heap operations. If the mesh is given in an adjacency structure, then the traversal of any contour takes time linear in the combinatorial complexity of the contour. Any saddle of the function is a vertex, and any contour can pass through any mesh cell only once. Therefore, the total time for traversal is $O(n^2)$ in the worst case, and the same amount of time is needed for setting the pointers of the active cells. □

The quadratic running time shown above is somewhat pessimistic, since it applies only when there is a linear number of saddles for which the contour through them has linear complexity. We can also state that the running time is $O(n \log n + \sum_{i=1}^{m} |C_i|)$, where the m saddles lie on contours $C_1, \ldots, C_m$ with complexities $|C_1|, \ldots, |C_m|$.

We claimed that the additional storage of the algorithm could be made sublinear in practice. With additional storage we mean the storage besides the mesh (input) and the contour tree (output). We will show that $O([\text{no. maxima}] + \max_{1 \leq i \leq m} |C_i|)$ extra storage suffices. We must reduce the storage requirements of both the event queue and the status structure.

Regarding the event queue, we initialize it with the values of the local maxima only. During the sweep, we'll insert all vertices incident to active cells, as soon as the cell becomes active. This guarantees that the event queue uses no more additional storage than claimed above. Considering the status structure, we cannot afford using additional pointers with every cell of the mesh to superarcs any more. However, we need these pointers only when the cell is active. We'll make a copy of the active part of the mesh, and with the cells in this copy, we may use additional pointers to superarcs in $\mathcal{T}$ and to the corresponding cells in the original mesh. When a cell becomes inactive again, we delete it from the copy. With these modifications, the additional storage required is linear in the maximum number of active cells and the number of local maxima. This can be linear in theory, but will be sublinear for most real-world meshes. The asymptotic running time of the algorithm is not influenced by these changes.

3.3.2 The two-dimensional case

In the two-dimensional case, the time bound can be improved to $O(n \log n)$ time in the worst case by a few simple adaptations. First, a crucial observation: for two-dimensional meshes representing continuous functions, all saddles correspond to nodes of degree at least 3 in $\mathcal{T}$. Hence, at any saddle two or more contours merge, or one contour splits into at least two contours, or both. This is different from the situation in three dimensions, where a saddle can cause a change in genus of a contour, without causing a change in connectedness. The main idea is to implement a merge in time linear in the complexity of the *smaller* of the two contours, and similarly, to implement a split in time linear in the complexity of the *smaller resulting contour*.

In the structure, each active cell has a pointer to a *name* of a contour, and the name has a pointer to the corresponding superarc in $\mathcal{T}$. We consider the active cells and names as a union-find like structure [75, 112, 115] that allows the following operations:

- *Merge*: given two contours about to merge, combine them into a single one by renaming the active cells to have a common name.
- *Split*: given one contour about to split, split it into two separate contours by renaming the active cells for one of the contours in creation to a new name.
- *Find*: given one active cell, report the name of the contour it is in.

Like in the simplest union-find structure, a *Find* takes $O(1)$ time since we have a pointer to the name explicitly. A *Merge* is best implemented by changing the name of the cells in smaller contour to the name of the larger contour. Let's say that contours C_i and C_j are about to merge. Determining which of them is the smallest takes $O(\min(|C_i|,|C_j|))$ time if we traverse both contours simultaneously. We alternatingly take one "step" in C_i and one "step" in C_j. After a number of steps twice the combinatorial complexity of the smaller contour, we have traversed the whole smaller contour. This technique is sometimes called *tandem search*. To rename for a *Merge*, we traverse this smaller contour again and rename the cells in it, again taking $O(\min(|C_i|,|C_j|))$ time.

The *Split* operation is analogous: if a contour C_k splits into C_i and C_j, the name of C_k is preserved for the larger of C_i and C_j, and by tandem search starting at the saddle in two opposite directions we find out which of C_i and C_j will be the smaller one. This will take $O(\min(|C_i|,|C_j|))$ time. (Note that we cannot keep track of the size in an integer for each contour instead of doing tandem search, because then a *Split* cannot be supported efficiently.)

Theorem 5 *Let M be a two-dimensional mesh with n faces in total, representing a continuous, piecewise linear scalar function. The contour tree of this function can be computed in $O(n \log n)$ time and linear storage.*

Proof: We can distinguish the following operations and their costs involved:

- Determining for each vertex of what type it is (min, max, saddle, normal) takes $O(n)$ in total.
- The operations on the event queue take $O(n \log n)$ in total.
- Creating the nodes and arcs of $\mathcal{T}$, and setting the incidence relationships takes $O(n)$ time in total.
- When a cell becomes active, the name of the contour it belongs to is stored with it; this can be done in $O(1)$ time, and since there are $O(n)$ such events, it takes $O(n)$ time in total.
- At the saddles of the mesh, contours merge or split. Updating the names of the contours stored with the cells takes $O(\min(|C_i|,|C_j|))$ time, where C_i and C_j are the contours merging into one, or resulting from a split, respectively. It remains to show that summing these costs over all saddles yields a total of $O(n \log n)$ time.

We prove the bound on the summed cost for renaming by transforming $\mathcal{T}$ in two steps into another tree $\mathcal{T}'$ for which the construction is at least as time-expensive as for $\mathcal{T}$, and showing that the cost at the saddles in $\mathcal{T}'$ are $O(n \log n)$ in total.

Consider the cells to correspond to additional *segments* in $\mathcal{T}$ as follows. Any cell becomes active when the sweep plane reaches its highest vertex, and stops being active when the sweep plane reaches its lowest vertex. These vertices correspond nodes in $\mathcal{T}$, and the cell is represented by a segment connecting these nodes. Note that any segment connects two

nodes one of which is an ancestor of the other. A segment can be seen as a shortcut of a directed path in $\mathcal{T}$, where it may pass over several nodes and supernodes.

The number of cells involved in a merge or split at a saddle is equivalent to the number segments that pass over the corresponding supernode v in $\mathcal{T}$. The set of segments passing v can subdivided into two subsets as follows: segments corresponding to cells that are intersected by the same contour before the merge or after the split at the saddle corresponding to v are in in same subset. The size of the smallest subset of segments passing v determines the costs for processing the saddle (since we do tandem search).

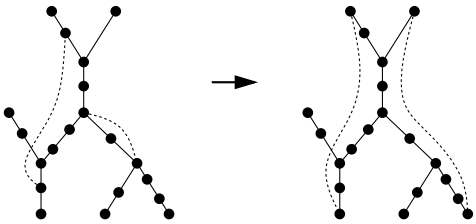


Figure 3.4: Stretching two segments (dotted) in $\mathcal{T}$.

The first transformation step is to *stretch* all segments (see Figure 3.4); we simply assume that a segment starts at some source node that is an ancestor of the original start node, and ends at a sink that is a descendant of the original end node. It is easy to see that the number of segments passing any saddle can only increase by the stretch.

The second transformation step is to repeatedly *swap* superarcs, until no supernode arising from a split (bifurcation) is an ancestor of a supernode arising from a merge (junction). Swapping a superarc s from a bifurcation v to a junction u is defined as follows (see Figure 3.5): let $s' \neq s$ be the superarc that has u as its lower supernode, and let $s'' \neq s$ be the superarc that has v as its upper supernode. The number of segments passing the superarcs s', s, and s'' is denoted by a, b, and c, respectively, as is illustrated in Figure 3.5.

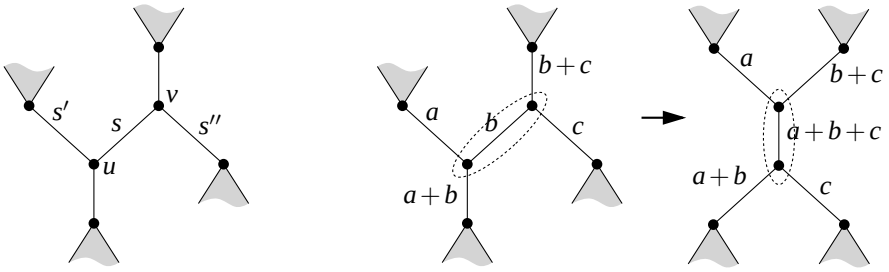


Figure 3.5: Swapping a superarc.

These numbers are well-defined, since after stretching, any segment passes a superarc either completely or not at all. Now shift s' upward along s, such that v becomes its new lower supernode, and shift s'' downward along s, such that u becomes its new upper supernode. Note that all edges passing s' and all edges passing s'' before the swap now also pass s.

Before the swap, the time spent in the merge at u and the split at v, is $O(\min(a,b)+\min(b,c))$ where a,b,c denote the number of segments passing these superarcs. After the swap, this becomes $O(\min(a,b+c)+\min(a+b,c))$, which is at least as much. No segment ends, because all of them were stretched.

It can easily be verified that a tree $\mathcal{T}'$, with no bifurcation as an ancestor of a junction, can be derived from any tree $\mathcal{T}$ by swaps of this type only. Any segment in $\mathcal{T}'$ first passes a sequence of at most $O(n)$ junctions, and then a sequence of at most $O(n)$ bifurcations.

We charge the costs of the merge and split operations to the segments that are in the smallest set just before a merge and just after a split. Now, every segment can pass $O(n)$ junctions and bifurcations, but no segment can be more than $O(\log n)$ times in the smaller set. Each time it is in the smaller set at a junction, it will be in a set of at least twice the size just after the junction. Similarly, each time it is in the smallest set just after a bifurcation, it came from a set of at least twice the size just before the bifurcation. It follows that any segment is charged at most $O(\log n)$ times. Summing over the $O(n)$ segments, this results in a total of $O(n\log n)$ time for all renamings of cells. This argument is standard in the analysis of union-find structures, for instance [27]. □

As we noted before, Tarasov and Vyalyi [111] succeeded in extending the ideas above and obtain an $O(n\log n)$ time algorithm to construct the contour tree for three-dimensional meshes.

The $O(n\log n)$ time bounds for the contour tree construction are tight: Given a set S of n numbers $s_1,\ldots,s_n$, we can construct in $O(n)$ time a triangular mesh with n saddles at heights $s_1,\ldots,s_n$, such that in the corresponding contour tree all the saddles lie in sorted order along the path from the global minimum to the global maximum.

The mesh is constructed as follows (see Figure 3.6): We place n vertices $v_1,\ldots,v_n$ equally spaced on a circle C in the (x,y)-plane with radius 2 and center at a point c. Now we elevate each v_i such that its z-coordinate is s_i. These vertices will be the saddles in the terrain. Next, we place n vertices $w_1,\ldots,w_n$ at a circle C' with radius 3 and also centered at c, such that each w_i is collinear with c and v_i. We elevate each w_i to height $\max(S)+1$; these vertices will be the local maxima. At the center c of C and C', we place one vertex at height $\max(S)+2$: the global maximum. Finally, we place a third set of vertices $u_1,\ldots,u_n$ at a circle C'' with radius 4 and centered at c, such that each vertex u_i is radially interleaved with the vertices v_i and v_{i+1}. The height of all these vertices u_i is $\min(S)-1$; all these vertices lie on the global minimum. Edges in the terrain are as shown in Figure 3.6, and the corresponding contour tree is shown in the same figure.

If we could construct the contour tree in $o(n\log n)$ time, then we could effectively sort any set S of n reals in $o(n\log n)$ time by constructing a mesh for it in $O(n)$ time as described

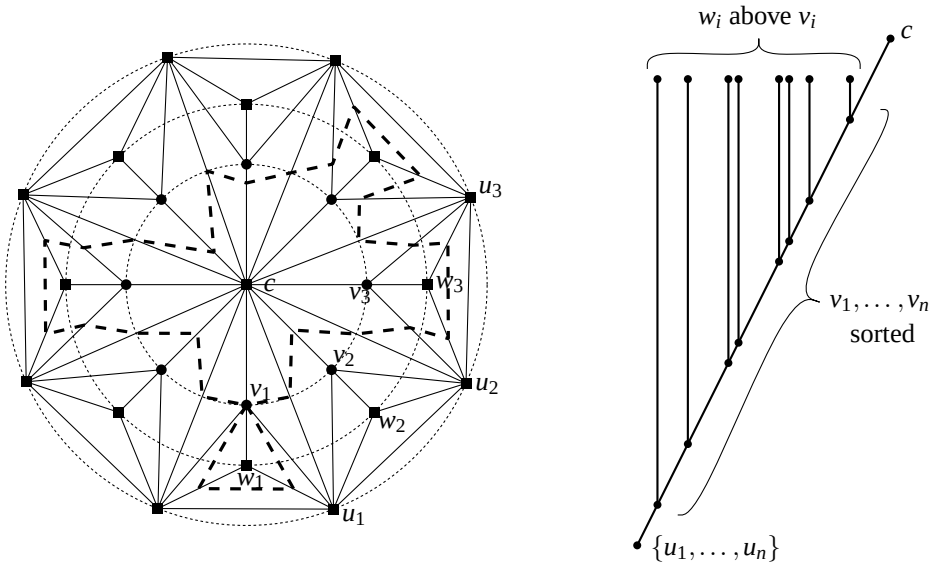


Figure 3.6: Unstructured mesh of which the contour tree contains the sorted sequence of the input values. The contour through v_1 is shown; we must have $v_2 < v_1 < v_3$.

above, constructing a contour tree for the mesh in $o(n \log n)$ time, and reporting the height values of the vertices corresponding to the nodes on the path from the global minimum to the global maximum in the contour tree in linear time.

3.4 Seed set selection

A seed set is a subset of the cells of the mesh. Such a set serves as a collection of starting points from which contours can be traced, for instance for visualization. A seed set is *complete* if every possible contour passes through at least one seed. From now on, we understand seed sets to be complete, unless stated otherwise. Since we assume linear interpolation over the cells, the function values occurring in one cell form exactly the range between the lowest and the highest valued vertices. Any cell is represented as a *segment* between two nodes of the contour tree $\mathcal{T}$, as in the proof of Theorem 5. Segments can only connect two nodes of which one is an ancestor of the other. Like the arcs of $\mathcal{T}$, the segments are directed from the higher to the lower value. So each segment is in fact a shortcut of a directed path in $\mathcal{T}$. We say that the segment *passes*, or *covers*, these arcs of $\mathcal{T}$. Let $\mathcal{G}$ denote the directed acyclic graph consisting of the contour tree extended with the segments of all mesh elements. The small seed set problem now is the following graph problem: find a small subset of the segments such that every arc of $\mathcal{T}$ is passed by some segment of the subset.

In this section we give two methods to obtain complete and small seed sets. The first gives a seed set of minimum size, but it requires $O(n^2 \log n)$ time for its computations. The second method requires $O(n \log^2 n)$ time and linear storage (given the contour tree and the segments), and gives a seed set that can be expected to be small, which is evidenced by test results.

3.4.1 Seed sets of minimum size in polynomial time

We can find a seed set of minimum size in polynomial time by reducing the seed set selection problem to a minimum cost flow problem. The flow network $\mathcal{G}'$ derived from $\mathcal{G}$ is defined as follows: we augment $\mathcal{G}$ with two additional nodes, a *source* σ and a *sink* σ'. The source σ is connected to all maxima and bifurcations by additional segments, and the sink is connected to all minima and junctions with additional segments. This is illustrated in Figure 3.7, left. In the same figure (right) a shorthand for the same flow network has been drawn: for readability, σ and σ' have been left out, and the additional segments incident to σ and σ' have been replaced by "$+$" and "$-$" signs, respectively. From now on we will use this shorthand notation in the figures.

Costs and capacities for the segments and arcs are assigned as follows: nodes in $\mathcal{G}$ are ordered by the height of the corresponding vertices in the mesh, and segments and arcs are considered to be directed: segments (dotted) go downward from higher to lower nodes, arcs (solid) go upward from lower to higher nodes. The source σ is considered to be the highest node, and σ' the lowest. Segments in $\mathcal{G}$ have capacity 1 and cost 1, and arcs have capacity ∞ and cost 0. The additional segments in $\mathcal{G}'$ incident to σ and σ' also have capacity 1, but zero cost.

From graph theory we have the following lemma:

Lemma 2 *For any tree, the number of maxima plus the number of bifurcations equals the number of minima plus the number of junctions.*

Hence, the number of pluses in $\mathcal{G}$ balances the number of minuses. Let this number be f.

Consider the following two related problems, the *flow problem* (given the network $\mathcal{G}'$ as defined above and a value f, find a flow of size f from σ to σ'), and the *minimum cost flow* problem (find such a flow f with minimum cost). For both problems, a solution consists of an assignment of flow for each segment and arc in $\mathcal{G}'$. For such a solution, let the *corresponding segment set* $\mathcal{S}$ be the set of segments in $\mathcal{G}$ that have a non-zero flow assigned to them (the additional segments in $\mathcal{G}'$ from σ to the maxima and bifurcations and from the minima and junctions to σ' are not in $\mathcal{S}$). Hence, the cost of an *integral solution*, where all flow values are integer, equals the number of segments in $\mathcal{S}$. We will show that for any integral solution to the minimum cost flow problem on $\mathcal{G}'$, the corresponding segment set $\mathcal{S}$ is a minimum size seed set for $\mathcal{G}$.

Lemma 3 *For any integral solution to the flow problem on $\mathcal{G}'$, the corresponding segment set $\mathcal{S}$ is a seed set for $\mathcal{G}$.*

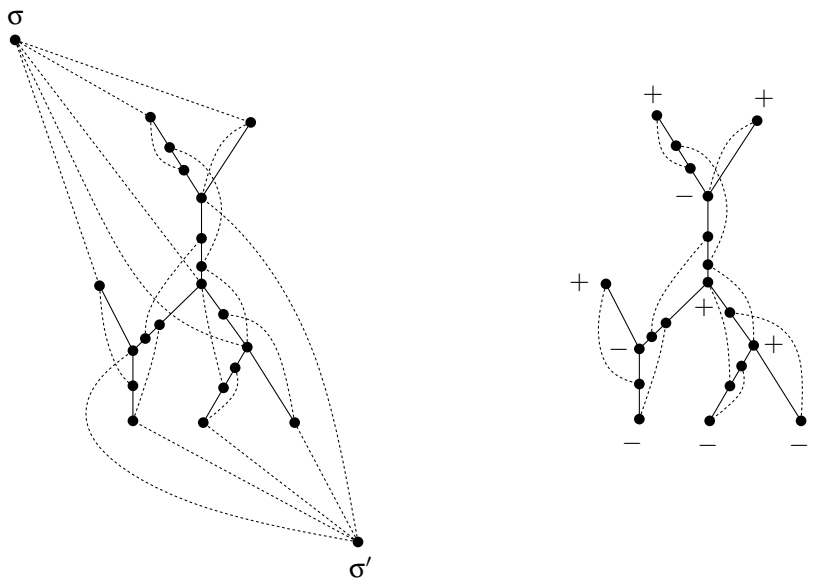


Figure 3.7: Flow network $\mathcal{G}'$ derived from $\mathcal{G}$, and shorthand for $\mathcal{G}'$.

Proof: Suppose that there is a flow of size f from σ to σ' in $\mathcal{G}'$ such that the corresponding segment set $\mathcal{S}$ is not a seed set for $\mathcal{G}$. In other words: there is an arc a in $\mathcal{G}'$ such that none of the segments in $\mathcal{G}'$ covering a has a non-zero flow assigned to it. We claim that the number of pluses in the subtree incident to and 'above' the highest incident node of a exceeds the number of minuses by one, and, analogously, that the number of minuses in the subtree incident to and 'below' the lowest incident node of a exceeds the number of pluses by one. This can be seen as follows: split $\mathcal{G}'$ into two subgraphs $\mathcal{G}'_1$ and $\mathcal{G}'_2$ by cutting a into a_1 and a_2 (see Figure 3.8), and creating a new minimum and maximum v_1 and v_2 incident to a_1 and a_2, respectively. Since by Lemma 2 the number of maxima and bifurcations in $\mathcal{G}'_1$ balances the number of minima and junctions in $\mathcal{G}'_1$, and there is no node in $\mathcal{G}$ that corresponds to the minimum v_1 in $\mathcal{G}'_1$, the claim holds for the subtree incident to and 'above' the highest incident node of a. Similarly, the claim holds for the subtree incident to and 'below' the lowest incident node of a. Since there is no downward flow via a or any of its covering segments, the flow from σ to σ' can be at most $f-1$. But this contradicts the assumptions that there is a flow of size f from σ to σ' in $\mathcal{G}'$ such that the corresponding segment set $\mathcal{S}$ is not a seed set for $\mathcal{G}$, and we conclude that the lemma holds. □

A seed set is *minimal* if the removal of any segment yields a set that is not a seed set. A *minimum* seed set is a seed set of smallest size.

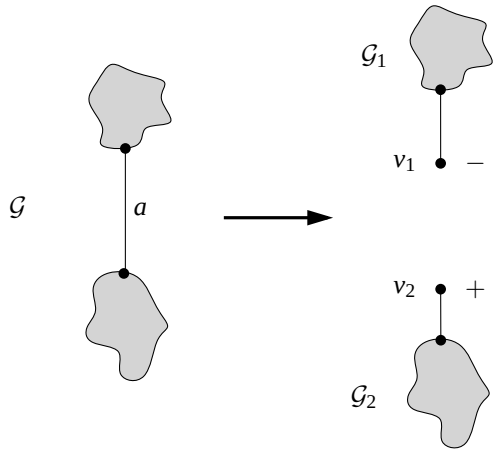


Figure 3.8: The number of pluses is not equal to the number of minuses within both subgraphs incident to a.

Lemma 4 *For any minimal seed set S for $\mathcal{G}$, there is a solution to the flow problem on $\mathcal{G}'$ such that the corresponding segment set $\mathcal{S}$ for that solution equals S.*

Proof: We show this by induction on n, the number of nodes of $\mathcal{G}$. It is straightforward to verify that the lemma holds for $n \leq 3$. For $n > 3$, we observe that for any minimal seed set S there is at least one arc in $\mathcal{G}$ that is covered by precisely one segment $s \in S$ (and possibly by some segments in $\mathcal{G}$ that are not in S); otherwise, removing an arbitrary segment from S would yield a smaller seed set, contradicting the minimality of S. Let a be such an arc in $\mathcal{G}$ that is covered by precisely one segment $s \in S$. If a is not incident to a leaf node of $\mathcal{G}$, then we can split $\mathcal{G}$ into two subgraphs $\mathcal{G}_1$ and $\mathcal{G}_2$ by cutting a and the segment s covering it. This introduces a new minimum for one of the subgraphs, and a new maximum for the other (Figure 3.9). Let S_1 be the set of segments from S that cover $\mathcal{G}_1$, with $s \in S$ replaced by the appropriate segment resulting from cutting s into two parts, and define S_2 in a similar way. S_1 and S_2 are minimal seeds sets for $\mathcal{G}_1$ and $\mathcal{G}_2$, respectively, and both subgraphs have fewer than n nodes. Hence, by induction, there is a solution for the flow problem on $\mathcal{G}'_1$ such that the corresponding segment set $\mathcal{S}_1$ for that solution equals S_1, and there is a solution for the flow problem on $\mathcal{G}'_2$ such that the corresponding segment set $\mathcal{S}_2$ for that solution equals S_2. Note that the sum of the sizes of the flows for both subgraphs is $f+1$, since we added a plus and a minus in the splitting process. Given the solutions to the flow problems for both subgraphs, it is straightforward to construct a flow that solves the flow problem for $\mathcal{G}$: simply remove the plus and minus that were added in the split, and undo the cutting of a and s.

This only works if a is not incident to a leaf node of $\mathcal{G}$, otherwise the split operation results in two subtrees, one with 2 nodes and one with n nodes, and we cannot apply induction. As noted before, there is at least one arc a that is covered by precisely one segment $s \in S$,

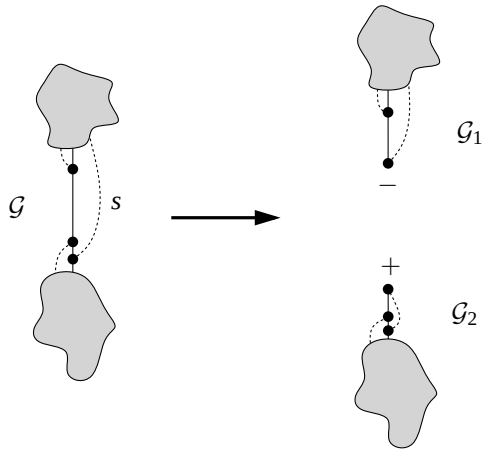


Figure 3.9: Splitting $\mathcal{G}$ at an internal arc.

and by zero or more segments that are in $\mathcal{G}$ but not in S. So suppose that all arcs covered by only one segment are incident to a minimum or maximum. Let a be one of those arcs, and assume that a is incident to a maximum v of $\mathcal{G}$ (the case that a is incident to a minimum is analogous). Let the other endpoint of s be μ. As stated before, we cannot apply induction directly. Instead, we transform $\mathcal{G}$ by shortening some of its edges, such that S remains a minimal seed set and $\mathcal{G}$ can be split into two subgraphs $\mathcal{G}_1$ and $\mathcal{G}_2$ with fewer than n nodes and with corresponding minimal seed sets S_1 and S_2. By the induction hypothesis, there is a flow for $\mathcal{G}'_1$ for which S_1 is the corresponding segment set, and a flow for $\mathcal{G}'_2$ for which S_2 is the corresponding segment set. Because of the simplicity of the reduction from $\mathcal{G}$ to $\mathcal{G}_1$ and $\mathcal{G}_2$, it is straightforward to construct a flow for $\mathcal{G}'$ corresponding to S, given the flows for $\mathcal{G}'_1$ and $\mathcal{G}'_2$.

We distinguish four cases:

- v and μ lie on the same superarc of $\mathcal{G}$ (see Figure 3.10). In that case, we transform $\mathcal{G}$ into $\mathcal{G}_1$ by retracting all segments that pass μ (i.e., μ is made their highest incident node), and by removing v, s and all arcs and segments between μ and v (the latter segments are the ones that have their lower endpoint inbetween v and μ, or incident to μ). Now $S \setminus \{s\}$ is a minimal seed set for the resulting graph $\mathcal{G}_1$, which has fewer than n nodes, and by the induction hypothesis, there is a solution to the flow problem on $\mathcal{G}'_1$ for which the corresponding segment set equals $S \setminus \{s\}$. In this case, $\mathcal{G}_2$ is the empty graph. It is straightforward to construct a solution to the flow problem for $\mathcal{G}'$, given a solution to the flow problem for $\mathcal{G}'_1$.
- The first supernode on the path from v to μ is a bifurcation (see Figure 3.11). The segments passing that bifurcation that go into the same subtree as s remain as they

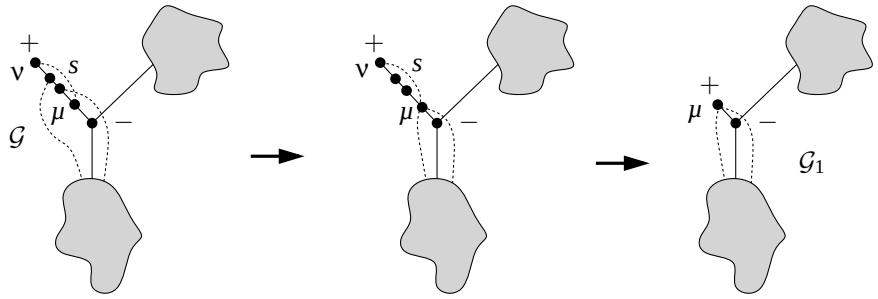


Figure 3.10: Retracting segments to the lower endpoint μ of s.

are; the ones that go into the other subtree are retracted to the bifurcation. $\mathcal{G}$ is split into two subgraphs $\mathcal{G}_1$ and $\mathcal{G}_2$ by cutting off the subtree below the bifurcation and not containing μ. Let S_1 and S_2 be the subsets of S as defined in the first paragraph of the proof. S_1 and S_2 are minimal seed sets for the two resulting subgraphs, both of which have fewer than n nodes. By the induction hypothesis, there is a solution to the flow problem on $\mathcal{G}_1'$ for which the corresponding segment set equals S_1, and there is a solution to the flow problem on $\mathcal{G}_2'$ for which the corresponding segment set equals S_2. From these solutions, a solution for the flow problem for $\mathcal{G}'$ for which the corresponding segment set is S can easily be derived.

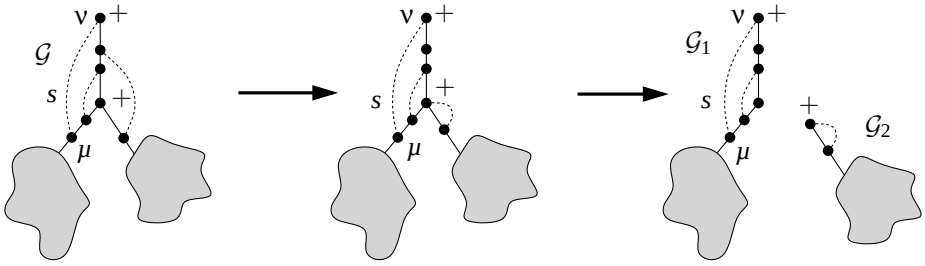


Figure 3.11: Retracting segments to a bifurcation.

- The first supernode on the path from ν to μ is a junction, and μ is not a minimum of $\mathcal{G}$ for which s is the only segment in S incident to it (see Figure 3.12). $\mathcal{G}$ is split into $\mathcal{G}_1$ and $\mathcal{G}_2$ by separating the superarc from ν to the junction. The subgraph containing ν is $\mathcal{G}_1$, and the other subgraph is $\mathcal{G}_2$. Segment s is retracted to end at the junction and is only used in $\mathcal{G}_1$. The segments originating in $\mathcal{G}_1$ and ending in $\mathcal{G}_2$ are retracted to start at the junction. S_1 and S_2 are defined as before. Note

that s is the only segment in S_1; the only other segments in $\mathcal{G}_1$ are the ones that both started and ended on the superarc incident to ν. These segments were not in S before the modification of $\mathcal{G}$, because they were completely covered by s. Hence, S_1 is a minimal seed set for $\mathcal{G}_1$. All segments in $S \setminus \{s\}$ cover only arcs in $\mathcal{G}_2$, since the segments in $S \setminus \{s\}$ that originated in $\mathcal{G}_1$ were retracted to the junction. Furthermore, $S \setminus \{s\}$ is a seed set for $\mathcal{G}_2$, since any arc in $\mathcal{G}_2$ that was covered by s before the split was covered by some other segment of S as well, and these segments still cover the same arcs in $\mathcal{G}_2$ after the split. Finally, $S \setminus \{s\}$ is a minimal seed set for $\mathcal{G}_2$: if any segment s' could be removed from $S \setminus \{s\}$, such that $S \setminus \{s, s'\}$ would be a seed set for $\mathcal{G}_2$, then $S \setminus \{s'\}$ would be a seed set for $\mathcal{G}$, contradicting the minimality of S.

$\mathcal{G}_1$ and $\mathcal{G}_2$ both have fewer nodes than $\mathcal{G}$. Hence, by the induction hypothesis, there is a solution to the flow problem on $\mathcal{G}'_1$ for which the corresponding segment set equals S_1, and there is a solution to the flow problem on $\mathcal{G}'_2$ for which the corresponding segment set equals S_2. From these solutions, a solution for the flow problem for $\mathcal{G}'$ for which the corresponding segment set is S can easily be derived.

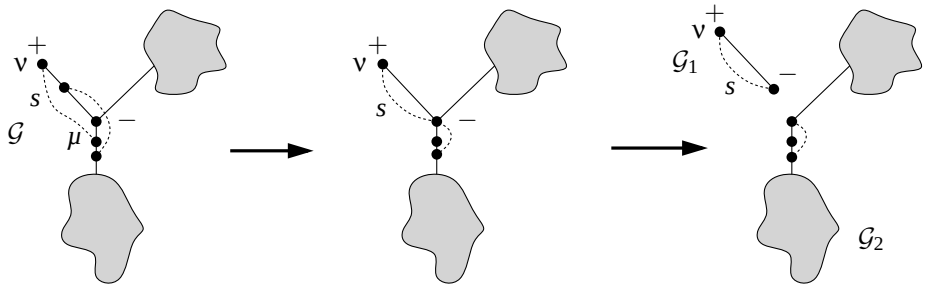


Figure 3.12: Retracting segments to a junction.

- The first supernode on the path from ν to μ is a junction, and μ is a minimum of $\mathcal{G}$ for which s is the only segment in S incident to it (see Figure 3.13). In this case, we cannot retract s to the junction as in the previous case, since that would leave the arc incident to μ uncovered. If the last supernode on the path from μ to ν is a junction, the situation is symmetric to the second case; this can be seen by turning $\mathcal{G}$ 'upside-down' and reversing the roles of ν and μ, and of the pluses and the minuses. Otherwise, if the last supernode on the path from μ to ν is a bifurcation, we transform $\mathcal{G}$ into $\mathcal{G}_1$ by removing s, ν, the superarc incident to ν, and all segments that start and end on this superarc. The segments that start in the removed superarc and pass the first junction on the path from ν to μ are retracted to start at the junction. Symmetrically, we remove μ, the superarc incident to μ, and all segments that start and end on this superarc. Segments that pass the last bifurcation on the path from ν to μ are retracted to end at the bifurcation. In this case, $\mathcal{G}_2$ is the empty graph. Note that s is the only segment that we removed from S; all other

segments that we removed are the ones that both started and ended in the superarc incident to v, or to μ. These segments were not in S before the modification of $\mathcal{G}$, because they were completely covered by s. All segments in $S \setminus \{s\}$ cover only arcs in $\mathcal{G}_1$, since the segments in $S \setminus \{s\}$ that originated in the superarc incident to v were retracted to start at the first junction on the path from v to μ, and the segments in $S \setminus \{s\}$ that ended in the superarc incident to μ were retracted to end at the last bifurcation on the path from v to μ. Furthermore, $S \setminus \{s\}$ is a seed set for $\mathcal{G}_1$, since any arc in $\mathcal{G}_1$ that was covered by s before the split was covered by some other segment of S as well, and these segments still cover the same arcs in $\mathcal{G}_1$ after the split. Finally, $S \setminus \{s\}$ is a minimal seed set for $\mathcal{G}_1$: if any segment s' could be removed from $S \setminus \{s\}$, such that $S \setminus \{s, s'\}$ would be a seed set for $\mathcal{G}_1$, then $S \setminus \{s'\}$ would be a seed set for $\mathcal{G}$, contradicting the minimality of S.

$\mathcal{G}_1$ has fewer nodes than $\mathcal{G}$, and by the induction hypothesis, there is a solution to the flow problem on $\mathcal{G}'_1$ for which the corresponding segment set equals S_1. From this solution, a solution for the flow problem for $\mathcal{G}'$ for which the corresponding segment set is S can easily be derived.

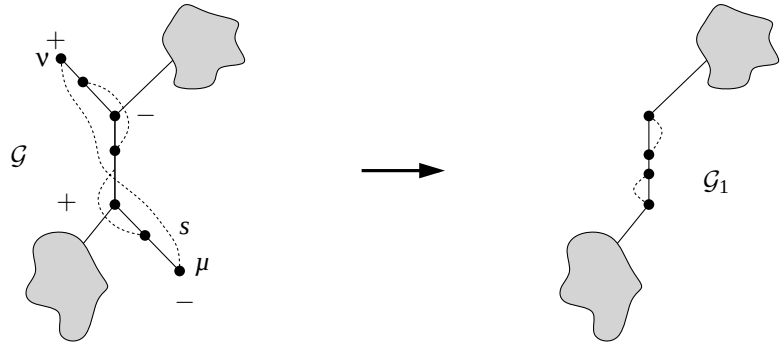


Figure 3.13: Removing a segment between a local maximum and a local minimum.

□

Combining Lemmas 3 and 4 gives the following result:

Theorem 6 *The minimum seed set selection problem for $\mathcal{G}$ can be solved by applying a minimum cost flow algorithm to $\mathcal{G}'$ that gives an integral solution. Such a solution is guaranteed to exist, and the corresponding segment set for that solution is an optimal seed set for $\mathcal{G}$.*

The minimum cost flow problem can be solved with a successive shortest path algorithm[4, pp. 320–324]. Starting with a zero flow, this algorithm determines at every step

the shortest path π from σ to σ', where the length of an arc or segment is derived from its cost. The arc or segment with the lowest capacity c on this shortest path π determines the flow that is sent from σ to σ' along π. Then the *residual network* is calculated (costs and capacities along π are updated), and the algorithm iterates until the desired flow from σ to σ' is reached, or no additional flow can be sent from σ to σ' along any path.

In our case, c is always 1 and the algorithm terminates after f iterations. If we use Dijkstra's algorithm to find the shortest path in each iteration, the algorithm runs in $O(n^2 \log n)$ time on our graph $\mathcal{G}'$, and uses $O(n)$ memory.

Theorem 7 *An optimal seed set for $\mathcal{G}$ can be found in $O(n^2 \log n)$ time, using $O(n)$ memory.*

3.4.2 Efficient computation of small seed sets

The roughly quadratic time requirements for computing optimal seed sets makes it rather time consuming in practical applications. We therefore developed an algorithm to compute a seed set that, after constructing the contour tree $\mathcal{T}$, uses linear storage and $O(n \log^2 n)$ time in any dimension. The seed sets resulting from this algorithm can be expected to be small, which is supported by test results.

As before we will describe the algorithm in the simplified situation that each critical vertex of the mesh is either a minimum, maximum, junction, or bifurcation. In the case of a junction or bifurcation, we assume that the degree is exactly three. These simplifying assumptions make it easier the explain the algorithm, but they can be removed as before.

Our algorithm is a simple greedy method that operates quite similar to the contour tree construction algorithm. We first construct the contour tree $\mathcal{T}$ as before. We store with each node of $\mathcal{T}$ two integers that will help determine fast whether any two nodes of $\mathcal{T}$ have an ancestor/descendant relation. The two integers are assigned as follows. Give $\mathcal{T}$ some fixed, left-to-right order of the children and parents of each supernode. Then perform a left-to-right topological sort to number all nodes. Then perform a right-to-left topological sort to give each node a second number. The numbers are such that one node u is an ancestor of another node v if and only if the first number and the second number of u are smaller than the corresponding numbers of v (see Figure 3.14).

This preprocessing of the contour tree takes $O(n)$ time, and afterwards, we can determine in $O(1)$ time for any two nodes whether one is a descendant or ancestor of the other.

Next we add the segments, one for each cell of the mesh, to the contour tree $\mathcal{T}$ to form the graph $\mathcal{G}$. Then we sweep again, now in the mesh and in the graph $\mathcal{G}$ simultaneously. During this sweep the seeds are selected. At each event point of the sweep algorithm (the nodes of $\mathcal{T}$), we test whether the arc incident to and below the current node is covered by at least one of the already selected seeds. If this is not the case, we select a new seed. The new seed will always be the greedy choice, that is, the segment (or cell) for which the function value of the lower endpoint is minimal. To determine if a new seed must be chosen, and to be able to make a greedy choice, a few data structures are needed that

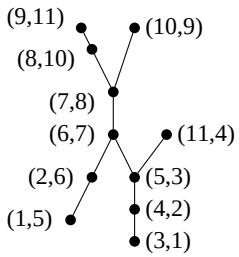


Figure 3.14: The numbering of $\mathcal{T}$.

maintain the currently chosen seed set and the candidate seeds that may be chosen next. As before, we call the cells that contain the current sweep value *active*. The segments and seeds of currently active cells are also called active. Similar, the superarcs for which the higher supernode has been passed, but the lower one not yet, are called active. We maintain the following sets during the sweep:

- A set S of currently chosen seeds. Initially, this set is empty; at the end of the algorithm, S contains a complete set of seeds.
- For an active superarc a, let $\hat{S}_a$ be the set of active seeds (already chosen) that cover a or part of it. We store a subset $S_a \subseteq \hat{S}_a$ that only contains the "deepest going" seeds of $\hat{S}_a$. More precisely, $s \in S_a$ if and only if for all $s' \in \hat{S}_a, s' \neq s$, we have that the lower endpoint of s is not an ancestor of the lower endpoint of s'.
- For each active superarc a, let $\hat{C}_a$ be a set of of active candidate seeds that cover a or part of it. We store a subset $C_a \subseteq \hat{C}_a$ that only contains the deepest going candidates of $\hat{C}_a$, and only if they go deeper than seeds of S_a. More precisely, $c \in C_a$ if and only if for all $c' \in \hat{C}_a, c' \neq c$, and for all $s \in S_a$, we have that the lower endpoint of c is not an ancestor of the lower endpoint of c' or s.

The algorithm must be able to determine if the next arc to be swept of superarc a is covered by some chosen seed. The subset S_a is exactly the subset of non-redundant seeds of $\hat{S}_a$. Similarly, the algorithm needs to maintain candidates that can be chosen if the next arc to be swept is not covered. The set $\hat{C}_a$ contains the active candidates, but the subset C_a contains only those candidates that could possibly be chosen. We'll show next that S_a and C_a can simply be stored in balanced binary trees.

The sets S_a and C_a correspond to a set of points in the plane whose coordinates are the two numbers assigned to the lower endpoints of the segments in S_a and C_a, see Figure 3.15. Since there are no ancestor/predecessor relationships between the endpoints of the segments in one set, none of the corresponding points lies to the right and above (or to the left and below) any other point in the same set; the points form a so-called *staircase*. This

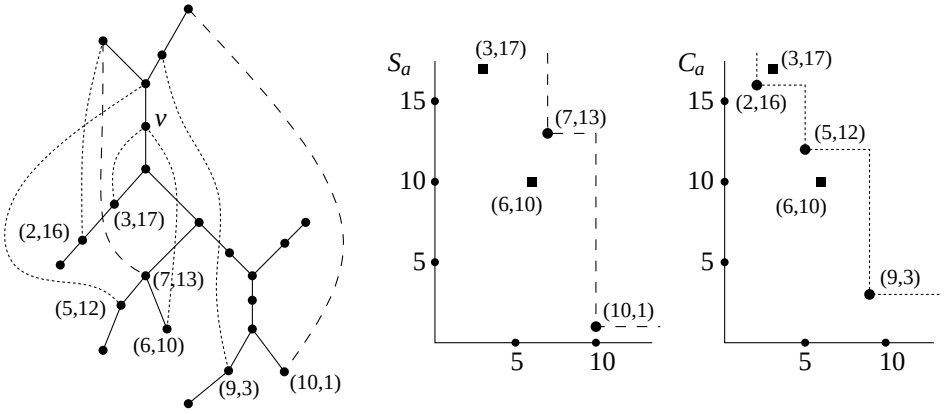


Figure 3.15: Just before the sweep reaches node v, the staircases of the active, chosen seeds in S_a (dashed) and of the active candidates in C_a (dotted).

means that S_a and C_a can each be maintained as a balanced binary search tree, ordered on the first number assigned to the lower endpoints of the segments alone. An *ancestor query* with a point (x,y) asks if the set contains a point (i,j) for which $x \geq i$ and $y \geq j$. Answering such a query is done by finding the point with maximum first number $\leq x$, and testing if this point has its second number $\leq y$. Similarly, we can determine easily whether a query point (x,y) has both numbers smaller than some point in the tree—a *descendant query*. Since the sorted order on the first number suffices, queries on, insertions in, and deletions from S_a and C_a can be done in time logarithmic in the size of the set.

We also maintain a heap on the set C_a, or rather, on the lower endpoints of the candidates in C_a, with cross-pointers between corresponding nodes in the heap and the binary tree for C_a. The heap allows us to extract efficiently the candidate segment with the lowest lower endpoint from C_a.

We will now describe the sweep algorithm that computes a small seed set, and analyze the total running time. We initialize the global set S of seeds to be empty. Then we sweep the nodes in $\mathcal{T}$ from high to low values. The following events can occur:

- Source: Initialize empty sets S_a and C_a for the superarc a starting at this node. This takes $O(1)$ time. Next, proceed as if the current node were a normal node.

- Normal node v: First, update the set C_a of candidate seeds for the superarc a on which the current node v lies. For each segment s that starts at v, we determine how it affects the set C_a. Let u be the lower endpoint of segment s. Perform an ancestor query on the tree storing S_a; if u is ancestor of the lower endpoint of any seed in S_a, we don't need the segment s as a candidate seed. In Figure 3.15, the queries are performed with the segments that have lower endpoints at $(3,17)$ and $(6,10)$. If u is not an ancestor for any lower endpoint from S_a, perform an ancestor query with

u on C_a. If u is an ancestor of the lower endpoint of any of the candidates in C_a, we also don't need the segment as a candidate. Otherwise, perform a descendant query with u. If u is the descendant of the lower endpoint of some candidate in C_a (there is at most one such candidate), then replace this candidate seed with the segment s. If u has no ancestor or descendant relation, then the query segment becomes a candidate seed; it is inserted in the binary tree and the heap for C_a. Note that we never have to worry about candidate seeds no longer being active; they will be replaced by newer candidates before this happens.

Next, test whether the arc of $\mathcal{T}$ starting at v is covered by any of the active seeds in S_a. This is surprisingly easy: if $|S_a| > 1$, the lower endpoints of the segments in S_a lie in different subtrees rooted at one or more bifurcations below the current node, since there are no ancestor/descendant relations between the endpoints of the segments in S_a. This means that the arc incident to and and below the current node is surely covered. On the other hand, if $|S_a| = 1$, we check in constant time whether the segment in S_a ends at the current node. If that is the case, we have to remove the only segment from S_a and choose a new seed, otherwise we are done. Choosing a new seed is also easy: Extract the candidate with the lowest lower endpoint using the heap on C_a, and remove this candidate from the binary tree on C_a as well, using the cross-pointers between the nodes in the heap and the binary tree. Next, insert this candidate as a seed in S_a and in the global set of seeds S.

The total time needed for all queries, replacements, and insertions at node v is $O(d \log n)$, where d is the degree of v in $\mathcal{G}$.

- Sink: Remove the sets S_a and C_a.
- Junction: First, for the two incoming arcs a and b at the junction, we determine which of the two values is smaller: $|S_a| + |C_a|$ or $|S_b| + |C_b|$. This takes $O(1)$ time if we keep track of the size of the sets. Suppose without loss of generality that the first of the two sums is the smallest. Then, for each seed s in S_a, we do the following. Let u be the lower endpoint of s. Perform an ancestor and descendant query on S_b with u. If u is ancestor, we do nothing; if u is descendant of the lower endpoint of some $s' \in S_b$, we replace s' by s in the tree on S_b. Otherwise, there are no ancestor/descendant relations and we insert s in the tree on S_b. If s is stored in S_b, it may be that s renders at most one of the candidates in C_b redundant: we perform a descendant query with u on C_b to discover this, and if there is a candidate whose lower endpoint is ancestor of s, we remove this candidate from C_b. The time needed for this step of the merge is $O(k \log n)$, where $k = \min(|S_a| + |C_a|, |S_b| + |C_b|)$. The merged set of active seeds is denoted $S_{a,b}$.

  Next, we do something similar for the two sets of candidate seeds. For each candidate c in C_a, let u be the lower endpoint of c. Perform an ancestor query with u on the set $S_{a,b}$ to test if c still is a valid candidate. If u is ancestor of the lower endpoint of some seed, then we discard c. Otherwise, we query C_b to see if u is an ancestor or descendant of the lower endpoint of a candidate c' in C_b. If u is the ancestor, we discard c; if u is the descendant, we replace c' by c. Otherwise, there are no ancestor/descendant relations and we insert c in C_b.

  Finally, we proceed as if the current node were a normal node.

Note that we cannot independently insert the seeds of the smaller set of S_a and S_b in the larger, and the candidates of the smaller set of C_a and C_b in the larger; we have to compare the seeds in S_a with the candidates in C_b, and the seeds in S_b with the candidates in C_a.

- Bifurcation: We have to split the set of active seeds S_a in two sets S_b and $S_{b'}$, the sets of active seeds for the left arc b and the right arc b' below the bifurcation, respectively. Note that the lower endpoint of any segment in S_b has a smaller first number assigned in the left-to-right topological sort of $\mathcal{T}$ than the lower endpoint of any segment in $S_{b'}$. Also note that we can test in $O(1)$ time for a lower endpoint u of a segment in S_a in which of the two subtrees of $\mathcal{T}$ rooted at the bifurcation u lies, by comparing u with the highest nodes in the two subtrees. So, if we test the segments in S_a simultaneously from low to high and from high to low values of their lower endpoints, we can determine in $O(\min(|S_b|, |S_{b'}|))$ time which of the two resulting sets will be the smaller one. Once we know this, we can split the tree for S_a into trees for S_b and $S_{b'}$ in $O(\min(|S_b|, |S_{b'}|) \log n)$ time, by extracting the seeds that will go in the smaller set from S_a and inserting them in a new set. For the set C_a of candidates, we do exactly the same to obtain the sets C_b and $C_{b'}$.

  Next, we process the bifurcation twice as a normal node, once for the arc and the segments that go into the left subtree of the bifurcation, and once for the arc and the segments that go into the right subtree.

It is easy to verify that the total time needed to process all sources, sinks and normal nodes is $O(n \log n)$; the time needed for the junctions and bifurcations can be analyzed much the same way as in Section 3.3.2, where we analyzed the running time for the construction of $\mathcal{T}$ for a two-dimensional mesh. In this case we get a bound of $O(n \log^2 n)$, which is also a bound on the overall running time.

At any stage of the sweep, the memory requirements are proportional to the size of the binary trees of all active superarcs, which is $O(n)$ worst-case.

Test results, presented in the next section, show that the seed sets resulting from the greedy algorithm are small. This doesn't surprise us; in fact, for a contour tree that only has maxima, junctions, and one single minimum, the greedy strategy is optimal. This is easy to show by a standard replacement argument: Let S_{opt} be an optimal seed set, let S be the chosen seed set, and assume $S \neq S_{\text{opt}}$. Let $s \in S_{\text{opt}} - S$ be the segment that ends highest. Let a be the highest arc of $\mathcal{T}$ such that s covers a but no other seed of S_{opt} does so. Such an a must exist otherwise $S_{\text{opt}} \setminus \{s\}$ also covers all of $\mathcal{T}$, contradicting the optimality of S_{opt}. Since S and S_{opt} are the same for the subtree above a, the greedy algorithm makes its choice only when it reaches the upper node of a. Suppose the greedy choice is $s' \neq s$. Then s' has its lower node on or below the lower node of s, since $\mathcal{T}$ doesn't have bifurcations. It follows that $\{s'\} \cup S_{\text{opt}} \setminus \{s\}$ is also an optimal seed set, but one which has one more segment in common with S.

For a contour tree with bifurcations, the greedy algorithm may choose segments that are not in an optimal seed set. This happens when the algorithm must choose a seed from a set of candidates that all pass some bifurcation. In that case, it is not guaranteed that the

candidate with the lowest lower endpoint is the optimal one. However, we suspect that the non-optimal choices that the algorithm may make in these cases do not increase the size of the resulting seed set by a large factor, compared to the size of an optimum seed set.

3.5 Test results

In this section we present empirical results for generation of seed sets using the method of Section 3.4.2 (the method of Section 3.4.1 has not been implemented). In Table 3.5 results are given for seven data sets from various domains, both two-dimensional and three-dimensional. The data used for testing include:

- Heart: a two-dimensional regular grid of MRI data from a human chest;
- Function: a smooth synthetic function sampled over a two-dimensional domain;
- Bullet: a three-dimensional regular grid from a structural dynamics simulation;
- HIPIP: a three-dimensional regular grid of the wave function for the high potential iron protein;
- LAMP: a three-dimensional regular grid of pressure from a climate simulation;
- LAMP 2d: a two-dimensional slice of the three-dimensional data which has been coarsened by an adaptive triangulation method;
- Terrain: a two-dimensional triangle mesh of a height field.

The tests were performed on a Silicon Graphics Indigo^2 IMPACT with 128Mb memory and a single 250MHz R4400 processor. Presented are the total number of cells in the mesh, in addition to seed extraction statistics and comparisons to a previously known efficient seed set generation method. The method presented in Section 3.4.2 represents an improvement of 2 to 6 times over the method of [14]. The presented storage statistics account only for the number of items, and not the size of each storage item (a constant). Note that the seed set method presented here has, in general, greater storage demands, though storage remains sublinear.

3.6 Conclusions and further research

This chapter presented the first method to obtain seed sets for contour retrieval that are provably small in size. We gave an $O(n^2 \log n)$ time algorithm to determine the smallest seed set, and we also gave an algorithm that yields small seed sets and takes $O(n \log^2 n)$ time for functions over a two-dimensional domain and $O(n^2)$ time for functions over

Data	total cells	# seeds	storage	time (s)	# seeds of [14]	storage of [14]	time (s)
Structured data sets							
Heart	256x256	5631	30651	32.68	12214	255	0.87
Function	64x64	80	664	1.23	230	63	0.15
Bullet	21x21x51	8	964	2.74	47	1000	0.30
HIPIP	64x64x64	529	8729	121.58	2212	3969	3.24
LAMP 3d	35x40x15	172	9267	6.82	576	1360	0.33
Simplicial data sets							
LAMP 2d	2720	73	473	0.69	–	–	–
Terrain	95911	188	2078	13.67	–	–	–

Table 3.1: Test results and comparison with previous techniques.

higher-dimensional domains. In typical cases, the worst case quadratic time bound seems too pessimistic. The algorithms make use of new methods to compute the so-called contour tree.

Test results indicate that seed sets resulting from the methods described here improve on previous methods by a significant factor. Storage requirements in the seed set computation remain sublinear, as follows from the test results.

Our work may be extended in the following directions. Firstly, it may be possible to give worst case subquadratic time algorithms for four and higher-dimensional meshes; the three-dimensional case was solved recently [111]. Secondly, it is important to study what properties an interpolation scheme on the mesh should have to allow for efficient contour tree construction and seed set selection. Finally, we strongly suspect that the algorithm presented in Section 3.4.2 yields a seed set of at most twice the set of an optimum seed set, but the problem of finding a proof is still open.

CHAPTER 4

Efficient Settlement Selection for Interactive Display

4.1 Introduction

The generation of a digital map on a computer screen is a process involving many steps. Decisions have to be made on, for instance, the target scale of the map, the kind of map (choropleth, isoline or network map; see Section 1.1.3), which thematic layers to display, which features of a specific layer to include or exclude, the use of colors, and the position of the labels. Some of these steps need to be taken only once or a limited number of times, and the decisions involved in these steps may be made by the GIS user. Other steps may be taken repeatedly and may involve a lot of computation. It is desirable that those steps can be performed automatically and efficiently by the GIS. An example of such a step is the generalization process. Even when only a few thematic map layers are selected, displaying all data in full detail is often unwanted, because it would result in a cluttered and unreadable map. Important cartographic generalization operations are aggregation and simplification. In aggregation, individual map features are replaced by one or more regions representing groups of those features; for instance, individual houses by may be replaced by a single region depicting urban area. Simplification is the reduction of detail; for example, a polyline representing a river may be displayed with fewer line segments, such that only the global shape of the river is maintained. More information on generalization can be found in Robinson et al. [90] and other textbooks on cartography, e.g. [19, 28, 63].

When cities and towns have to be displayed, choices have to be made which of them to include and which to omit; this is called settlement or place selection (Flewelling and

Egenhofer[42]; Kadmon[61]; Langran and Poiker[66]; Töpfer and Pillewizer[114]). It is intuitively clear that a large city should take precedence over a smaller one when the two have to compete for space on the computer screen. However, it is not necessarily true that each of the selected cities is larger than any of the cities that are not selected for display. A large city close to a yet larger city may be excluded, and a smaller city not in the neighborhood of any other larger city may be included because of its *relative importance*.

Settlement selection is performed just prior to generalization, although it can be considered as part of the generalization procedure as well. It has to be performed when a cartographer is in the process of interactively designing a map from a geographic database, or when a GIS user is panning and zooming in or out on a small scale map. Especially in the latter case it is necessary, or at least strongly desirable, that the selection process can be performed automatically by the GIS. To minimize the delay in screen updates, efficient methods for settlement selection are needed.

This chapter discusses models that have been described for settlement selection. We also propose a new model and three variations and discuss the advantages of our models over the existing ones. We implemented several of the models for comparison, and we provide figures and statistics on the output of several test runs on two different data sets. In the process of interactive map design, it is useful if the cartographer has control over things like number of cities selected, and the degree in which clustering is allowed. We have included these controls in the interface of the implementation.

4.1.1 Previous work

Several decades ago, Töpfer and Pillewizer[114] formalized a means to determine how many features should be retained on a map when the scale is reduced and generalization is performed. Settlement selection itself starts by assigning an importance value to all settlements. The importance can simply be the population, but also a combination of population, industrial activities, presence of educational institutions, and so on.

Langran and Poiker[66] report five different methods for the selection of settlements. Most of them are incremental: cities are considered for display from most important to least important, and the addition to the map is performed only if some spatial condition is not violated. In two of the models, *settlement-spacing ratio* and *distribution-coefficient control*, the selection of a settlement is determined by only one, more important settlement close by. In the *gravity-modeling* method, selection is dependent on several settlements in the neighborhood. The *set-segmentation* and *quadrat-reduction* methods use recursive subdivision of the plane, and a direct application of the radical law by Töpfer and Pillewizer[114].

Flewelling and Egenhofer[42] discuss a number of factors that influence the selection of settlements. Following Mark[72], they assume that an importance attribute is assigned to the map features to allow for intelligent selection. Then they give a global discussion of ranking of settlements on non-spatial properties.

4.2 Existing and new models

Before describing the four models for settlement selection that we developed, we first discuss in Section 4.2.1 below three existing models, reported by Langran and Poiker[66]: *settlement-spacing ratio*, *gravity modeling* and *distribution-coefficient control*. The other two methods that Langran and Poiker[66] describe, *set-segmentation* and *quadrat-reduction*, require too much human intervention to be suitable for automated, or interactive, map design.

A disadvantage of the three existing models is that they don't directly give a *ranking* of the base set of settlements. A ranking is a display order; after computing a ranking of the base set beforehand, selecting any number of cities is simply a matter of choosing them in order of rank. For methods that don't determine a ranking, changing the number of selected settlements involves recomputation.

Adaptations can be made to the existing models to control the number of selected settlements from the base set, but this may have strange effects. For example, when selecting more settlements, it can happen that one of the chosen settlements is no longer selected, but instead a couple of others are. When selecting even more settlements, these discarded settlements can reappear. We say that a settlement-selection model has the *monotonicity property* if any selection of settlements necessarily includes the settlements of any smaller selection. Since our new selection models are based on a complete ranking of the settlements, they have the monotonicity property.

Although ranking facilitates the selection process, a model that produces a complete ranking is not necessarily better than a model that doesn't. The quality of a final selection depends on the data set used and the purpose of the resulting map. The quality of the existing models and our new ones can be assessed by comparing figures and statistics of selections (Section 4.4) and by visual inspection based on our implementation available on the World Wide Web.

In the models to be described next, we assume that an importance value is known for each settlement. The model defines which settlements are selected when their geographic location is incorporated as well.

4.2.1 Existing models

Settlement-spacing ratio

In the settlement-spacing ratio model, circles are placed centered at the settlements, and the size of each circle is inversely proportional to the importance of its corresponding settlement. More precisely, the radius is c/i where i is the importance and $c > 0$ is some constant (the same for all settlements). Settlements are tested in order of importance, starting with the most important one. A settlement is only accepted if its circle contains none of the previously accepted settlements. In other words: small settlements will only be accepted if they are isolated.

Langran and Poiker[66] don't discuss how to influence the number of selected settlements. Since on small scale maps distances between settlements are smaller than on large scale maps of the same area, one can generally expect the number of displayed settlements to increase when the scale is increased and the constant of proportionality is kept the same. However, when a cartographer is not satisfied with the number of displayed settlements at a given scale, there should be a means to increase or decrease this number. Selecting fewer settlements can be as simple as stopping when the desired number is reached, but it is impossible to increase the number of settlements this way. Alternatively, one can tune the constant of proportionality *c*. This constant determines how many settlements are accepted; smaller values for *c* mean smaller circles and this generally leads to more settlements being selected for display. This is, however, not always the case, as is illustrated in Figure 4.1: settlements A is accepted, B is rejected, and C and D are accepted. But if *c* were slightly smaller, the circle of B would not contain settlement A anymore. So settlements A and B are accepted, but C and D are rejected, since their circles contain settlement B. If we continue to decrease *c*, settlements C and D will reappear. Note that decreasing *c* while keeping the scale fixed is equivalent to keeping *c* fixed and increasing the scale.

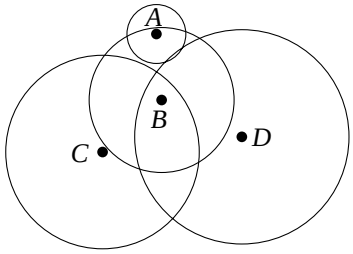


Figure 4.1: The settlement-spacing-ratio method doesn't satisfy the monotonicity property.

The example above shows that this method doesn't have the monotonicity property and that a complete ranking of the cities cannot be calculated. In fact, it can be that no value of *c* gives a desired number of settlements. It is also possible that two different selections have the same size. This is all caused by the fact the the monotonicity property is not respected by the model.

Gravity modeling

In the *gravity-modeling* method, a notion of *influence* is introduced: the influence of one settlement on another one is computed by dividing the importance of the first one (the selected one) by the distance to the other. Settlements are tested in decreasing order of importance, and a settlement *s* is only accepted if its importance is greater than the summed influence of all already selected settlements on *s*.

As in the previous model, the number of selected settlements is fixed for a given scale. This can be overcome by adapting the original model as follows: in the selection process, the next settlement under consideration is accepted if the summed influence of the already

accepted settlements on the candidate is less than c times the importance of the candidate. By controlling the tuning factor c, the number of selected settlements can be adjusted. However, if we adapt the model in this way, it doesn't respect the monotonicity property and doesn't give a complete ranking. Consider the following example (see Figure 4.2):

- The set of settlements consists of three settlements A, B, and C, with importance values of 4, 3, and 1, respectively;
- The three settlements are collinear, and B lies in between A and C;
- The distance between A and B is 1, and the distance between B and C is 3.

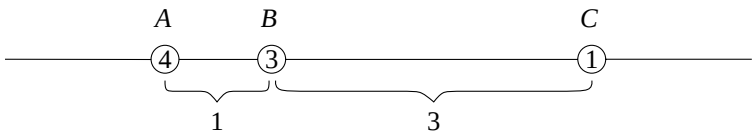


Figure 4.2: The gravity-modeling method doesn't satisfy the monotonicity property.

In this case, the influence from A on B is 4, the influence from A on C is 1, and the summed influence from A and B on C is 2. Hence, if $0 < c \leq 1$ then only A is selected; if $1 < c \leq 4/3$ then A and C are selected; if $4/3 < c \leq 2$ then A and B are selected; if $c > 2$ then A, B, and C are all selected.

Distribution-coefficient control

The third method, *distribution-coefficient control*, uses the *nearest-neighbor index* for the selection process. The nearest-neighbor index is the ratio of the actual mean distance to the nearest neighbor and the expected mean distance to the nearest neighbor for a uniformly distributed set of settlements in a square region. The expected mean distance is $1/(\sqrt{2p})$, where p is the number of settlements per unit area. Again, settlements are processed in decreasing order of importance. Starting with a small set of largest ones, settlements are only accepted if their addition to the already accepted set doesn't decrease the nearest neighbor index. The number of settlements in the final selection is fixed and is even independent of scale, but can be controlled again by introducing a tuning factor: a settlement is accepted if the nearest neighbor index after its addition is at least c times the nearest neighbor index before its addition. As c is lowered, more and more settlements are selected. However, there can be values of c for which a jump in the number of selected settlements occurs: if c is lowered below one of these thresholds, the acceptance of the next settlement causes the nearest neighbor index to drop, which in turn causes the next settlement to be selected also, and so on. Test results of our implementation show that this cascading effect can occur in practice. This model also does not respect the monotonicity property when a tuning factor c is introduced. Consider the following example (see Figure 4.3):

- The set of settlements consists of four settlements *A*, *B*, *C*, and *D*, with importance values 4, 3, 2, and 1, respectively;
- The four settlements are collinear, *B*, and *C* lie in between *A* and *D*, and *C* lies inbetween *B* and *D*;
- The distance between *A* and *B* is $\sqrt{3}$, the distance between *B* and *C* is $\frac{3\sqrt{2}-\sqrt{3}}{2} - \varepsilon$, and the distance between *C* and *D* is ε, for $\varepsilon > 0$ sufficiently small.

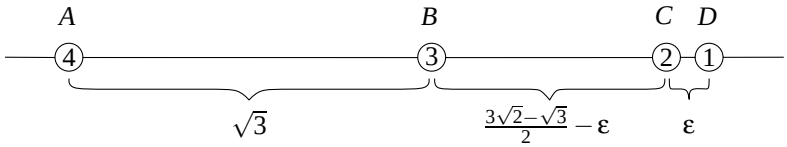


Figure 4.3: The distribution-coefficient control method doesn't satisfy the monotonicity property.

For the expected distance to the nearest neighbor we have $1/\sqrt{2p} = 1/(c'\sqrt{k})$, where k is the number of selected settlements and c' a constant depending on the area of interest.

The nearest-neighbor index for the initial set of settlements $\{A,B\}$ is $c'\sqrt{2}\sqrt{3}$. If c is set to 1, then C cannot be added to the set of selected settlements, since the average distance to the nearest neighbor for $\{A,B,C\}$ is $(\sqrt{3}+3\sqrt{2}-\sqrt{3}-2\varepsilon)/3 = \sqrt{2}-2\varepsilon/3$, and the nearest-neighbor index for $\{A,B,C\}$ is $c'(\sqrt{2}\sqrt{3}-2\varepsilon/\sqrt{3}) < c'\sqrt{2}\sqrt{3}$. Similarly, the nearest-neighbor index for $\{A,B,D\}$ is $c'\sqrt{2}\sqrt{3}$, so if the tuning factor c is set to 1, then *A*, *B*, and *D* will be selected. If we make another selection, starting with $\{A,B\}$, and with c lowered to $1-2\varepsilon/(3\sqrt{2})$, then C can be added to the selection, but for sufficiently small values of ε, D cannot be added. Finally, for very small values of c, all four settlements are selected. It follows that in this example the monotonicity property is not respected.

Another disadvantage of the model is that the actual importance of a settlement is only used in the order of processing, and not in the test whether the settlement should be selected or not.

4.2.2 New models

Circle growth

In our first model, the *circle-growth* method, a ranking of the settlements is determined as follows: for each settlement a circle is drawn with a radius that is proportional to the importance of the settlement. The initial constant of proportionality c is such that no two circles overlap. The next step is to increase c, causing all circles to grow, until the circle of some settlement fully covers the circle of some other one. The former is said to *dominate* the latter; the latter has the lowest rank of all settlements and is removed. This process is repeated while assigning higher and higher ranks, until only the most important

settlement remains. For two settlements s_1 and s_2 with importance values I_1 and I_2, $I_1 > I_2$, and distance $d(s_1, s_2)$, the circle of s_1 covers that of s_2 for $c \geq d(s_1, s_2)/(I_1 - I_2)$.

This method satisfies two important conditions:

- When two settlements compete for space on the map, the most important one of the two will survive.
- Settlements of low importance will be displayed on the map if there are no settlements of higher importance in their proximity.

The drawback of this method is that a settlement with very high importance can have a global effect on the map: its neighborhood is a large part of the map, and too many settlements near to it are suppressed. At the same time, in large regions with no settlement of high importance several settlements are selected. One way of resolving this is instead of giving each settlement a circle with a radius proportional to its importance i, letting the radius of the circle be proportional to i^α, with $0 \leq \alpha \leq 1$. By tuning α the influence of the importance of the settlements on the selection can be reduced.

Circle-growth variation I

The drawback of the (unmodified) circle-growth model led to the observation that settlements with a very high importance have too much influence on the selection, and this resulted in the opposite of preserving density locally. Our second method, a variation on the circle-growth method, doesn't have this problem. We'll rank from first to last this time, and as soon as a settlement is ranked, it receives a circle of the same size as the other settlements that are already ranked. All settlements that are not ranked yet have a circle of a radius proportional to their importance.

The complete ranking is calculated as follows: the settlement with the highest importance is assigned the highest rank. Next, the settlement that is second in rank is determined by applying the circle-growth idea. We choose the settlement whose circle is covered last by the circle of the settlement with the highest rank, and set its importance to that of its dominating settlement. This process is iterated, ranking a next settlement when its circle is covered last by any one of the ranked settlements.

With this method the distribution of the selected settlements can be expected to be more even than the distribution of the selection resulting from the circle-growth method, since in our second method the size of the circles is the same for all selected settlements. Indeed, our implementation verifies this; an evenly distributed selection is the result.

Although this method may seem far more time-consuming than the original circle-growth method, it can actually be implemented more efficiently than the original method. We will discuss this in Section 4.3.2.

Circle-growth variation II

In the previous two methods, all calculations are done with absolute importance values of the settlements. Our third method makes qualitative rather than quantitative comparisons.

First, the settlements are sorted by importance from low to high. Each settlement receives as a number the position in the sorted order. This number replaces the importance value, after which the ranking is computed as in variation I. Circles of selected settlements have equal size, and the size of the circles of the not-yet-selected settlements is proportional to their position in the list sorted on importance. In all our models, the domination of one settlement over another one is determined by the importance of the settlements and the distance between them. Compared to the previous model, the influence of the importance values on the selection of a settlement is reduced in favor of an isolated location, and therefore one can expect an even more evenly distributed selection than in the previous model, but with a lower total importance. Our experiments show that this is indeed the case.

Circle-growth variation III

In circle-growth variation I, we gave all selected settlements a circle of the same size, proportional to the importance of the most important settlement. In this model, circle-growth variation III, the selected settlements also get equal-sized circles, but this time the size of the circles is c times the size of the circle of the most-important not-yet-selected settlement for some $c > 1$. For the remainder, the circle-growth principles are applied in the same way as in circle-growth variation I. This means that each time that a settlement is selected, the size of the circles of the selected settlements decreases. The outcome of this model depends on the value of c. For large values of c, we expect evenly distributed selections with relatively low total importance values; for values of c very close to one, the settlements will be ranked in order of importance, which results in unevenly distributed selections with high total importance values.

4.3 Implementation

To be able to test our models, to collect some statistical data on the different selections, and to compare our new models with the existing ones, we implemented the discussed models in Java, and made our implementation accessible through the World Wide Web (see http://www.cs.uu.nl/~rene/settlement/). This enables others to experiment with the models and to validate our results.

4.3.1 User interface

The user interface is depicted in Figure 4.4: a large portion of the screen is reserved for displaying the settlements. Below the display area are the controls: buttons for selecting which of the seven implemented methods to use, or simply ranked by importance; buttons for displaying names and importance values with the settlements; and buttons to set the number of displayed settlements for the four new models. When the user selects one of the existing selection methods, this latter group of buttons is replaced by a slider for adjusting the tuning factor used in the three existing models (see Section 4.2.1). Statistics that are

displayed are the total number of selected settlements, the total population of the selected settlements (in thousands), and the average distance to the nearest neighbor (in pixels). In our experiments, we used the population of the settlements as importance values.

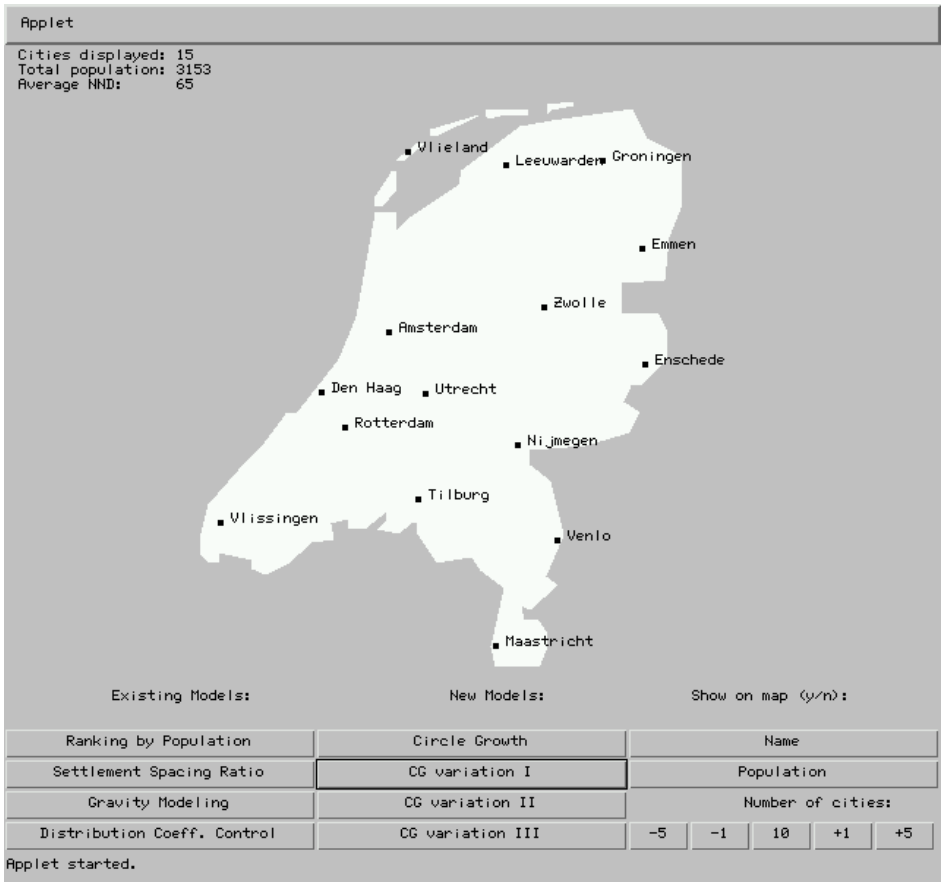


Figure 4.4: The user interface

4.3.2 Algorithms and data structures

In the implementation of the models, we didn't pay much attention to the efficiency of the algorithms, since our focus was on the outcome of the models rather than on speed. All models were implemented with a straightforward $O(n^2)$ time algorithm, or worse. Recall that for the new models, computing a ranking is preprocessing; once a ranking has been computed, making a selection with k settlements is simply a matter of outputting the first

k settlements in order of rank. For completeness, we discuss some ideas to improve on the running time of a straightforward implementation for some of the models.

Both circle-growth variation I and II can be implemented by incrementally constructing the Voronoi diagram of the ranked settlements (see Figure 4.5). We start with one ranked settlement, the most important one; its Voronoi cell is the whole plane. Since in both models the circles of all ranked settlements have the same size, all non-ranked settlements are dominated by their nearest ranked neighbor. That is, their circle will be covered first by the circle of the nearest ranked settlement. We exploit this by maintaining for each Voronoi cell a list of non-ranked settlements that lie in that cell. One of the settlements in each list is the last to be dominated, and it is a candidate for the next settlement to be chosen. Each Voronoi cell gives one candidate, unless there are no non-ranked settlements in the cell. We maintain all these candidate settlements in a heap, which makes it possible to determine in $O(1)$ time the next settlement to be added to the set of ranked settlements. The algorithm repeatedly takes the following steps until all settlements have been ranked:

1. Determine the next settlement s to be ranked by extracting it from the heap.
2. Determine in which Voronoi cell s lies. This can be done efficiently if we store a pointer with each candidates to the cell in which it lies.
3. Create a new cell for s in the Voronoi diagram. This involves modifying a number of existing cells, i.e., the neighbors of the new cell of s.
4. Inspect the lists of non-ranked settlements of all modified cells; some of the non-ranked settlements lie in the Voronoi cell of s. These settlements are removed from their original list and inserted in the (new) list of s, and the pointers of these settlements are updated. If any of the moved settlements was a candidate for its original cell, it is removed from the heap. A new candidate is determined for that cell, and inserted in the heap.
5. Determine which one of the non-ranked ranked in the list of s is the last that is covered by s. This is a new candidate, and it is inserted in the heap.

This algorithm resembles the randomized incremental construction algorithm of Guibas *et al.* [49] for computing the Delaunay triangulation of n points in $O(n \log n)$ expected time. Given the Delaunay triangulation of a point set, the Voronoi diagram can be constructed in $O(n)$ time. The main differences of our algorithm with the one from of Guibas *et al.* [49] are:

- The order in which the settlements are ranked is not random in our algorithm. In the analysis of the algorithm of Guibas *et al.* [49], it is essential that the order in which the points are inserted into the Delaunay triangulation is random. If the order of insertion is not random, one can only prove a $\theta(n^2)$ upper bound on the worst-case running time.

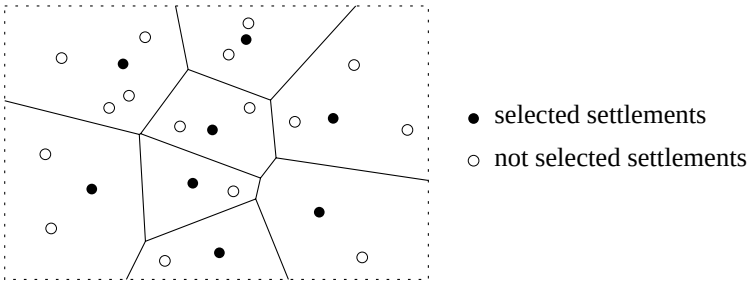


Figure 4.5: Maintaining the Voronoi diagram of the selected settlements

- At each iteration of our algorithm, the lists of non-ranked settlements are inspected. The total length of these lists can be linear during a linear number of iterations, even if the settlements were ranked in random order. This also leads to an $\Theta(n^2)$ running time in the worst case.

Because of these differences, we can only prove a worst-case running time of $\Theta(n^2)$ for our algorithm. However, under the condition that the number of Voronoi neighbors of the settlement that is ranked in the k-th iteration is $O(1)$ in most iterations, and that after a constant number of iterations the non-ranked settlements are distributed more or less evenly among the cells of the ranked settlements, the running time will probably be closer to $O(n \log n)$ than to $O(n^2)$. These conditions seem realistic to us, and comparable experiments with real-world data [41, 53, 54], in which points are inserted in non-random order, showed running times that are considerably better than $\theta(n^2)$.

Using Voronoi diagrams isn't helpful in our first method, the unmodified circle-growth model. The ranked settlements don't have equal-sized circles, and this means that we would have to use multiplicatively weighted Voronoi diagrams [10, 82], a variation of the standard Voronoi diagram that uses a different metric. Unfortunately, no $o(n^2)$ algorithm for the construction of such Voronoi diagrams is known.

In our last model, circle-growth variation III, Voronoi diagrams aren't helpful either. Since the size of the circles of the ranked settlements may change in a linear number of iterations, the order in which the non-ranked settlements are dominated may also change a linear number of times, and we would have to inspect all lists of non-ranked settlements to test for new candidates. This would lead to an $\Omega(n^2)$ running time.

Of the existing methods, the settlement-spacing ratio method can also be implemented by incrementally constructing the Voronoi diagram of the selected settlements; a settlement is only accepted if its circle does not contain its nearest neighbor, which we find by doing point location in the Voronoi diagram of the selected settlements. Since settlements are added in order of importance, we don't need to maintain lists of non-selected settlements for each Voronoi cell. As before, we can only guarantee a $\Theta(n^2)$ worst-case running

time, but under the assumption that the number of Voronoi neighbors of the settlement that is ranked in the k-th iteration is $O(1)$ in most iterations, the running time will be closer to $O(n \log n)$ than to quadratic. Note however that only one complete selection is computed in $O(n \log n)$ time, not a complete ranking. So if more settlements are needed, the algorithm has to be started all over with a different constant of proportionality.

For the gravity-modeling method, computing even one selection takes $O(n^2)$ time. It is not clear how to improve the performance of this model.

In the distribution-coefficient control method, testing each settlement involves determining its nearest neighbor, and determining for which of the already selected settlements the new settlement becomes the new nearest neighbor. With a straightforward algorithm this will take $O(n^2)$ time in total, but this can be improved to $O(n \log n)$ time under the same assumptions as before by incrementally constructing the Voronoi diagram of the selected settlements. Again, this is the time needed for computing a single selection of settlements.

4.4 Test results

We tested the three existing and the four new models on two data sets that we retrieved from the World Wide Web, one consisting of 156 cities of the USA with population figures of 1990, and the other consisting of 139 municipalities in the Netherlands with more recent population figures. The population of the cities was used as the importance, and in each of the new models we made selections of sizes 10 to 60 in steps of 10. Circle-growth variation III involves a tuning factor, and we ran that model with four different values for the factor. For the existing models, making a selection of a specified size is not always possible. To allow for comparison, we made selections of sizes as close as possible to the sizes of the selections made with the new models. The results are listed in Tables 4.1 and 4.2. Note that for the US data set, it was impossible to make selections of any size between 10 and 55 with the distribution coefficient control method. The visual appearance of selections of about 15 cities made with the different models is shown in Figures 4.6–4.9.

In general, it is difficult to define what constitutes a good selection. The degree of clustering can be one of the criteria, but a selection that is considered too clustered in one situation may be perfectly acceptable in another situation. On maps where cities fulfill a reference function, like on weather charts, clustering is undesirable, but on maps where for instance state boundaries have a reference function, clustering need not be avoided. The total importance of the selected settlements can be another criterion. For two selections with about the same degree of clustering, the one with the larger total importance can be considered the better one.

To compare the different models, we included figures on the total population size of the selected cities and the average distance to the nearest neighbor in Tables 4.1 and 4.2. From these tables we can conclude that circle growth variation I generally gives better results than the (standard) circle growth method; in most cases, variation I combines a larger

Model		Number of selected settlements					
		10	20	30	40	50	60
Ranking by Population	Population	20886	27304	31899	35488	38518	40858
	Avg. NND	32	25	36	27	22	19
CG	Population	17491	21562	24942	28072	30663	32972
	Avg. NND	80	54	41	32	27	24
CG Var. I	Population	17182	21803	25617	28014	31512	35002
	Avg. NND	119	67	50	39	35	30
CG Var. II	Population	10751	13774	16937	21811	24011	26811
	Avg. NND	121	70	51	42	37	31
CG Var. III factor: 5.0	Population	13701	16249	19329	23183	25468	29757
	Avg. NND	116	69	51	41	36	30
CG Var. III factor: 2.0	Population	17182	22064	26247	29239	33892	36805
	Avg. NND	119	70	50	39	34	29
CG Var. III factor: 1.5	Population	17182	23188	26247	32099	34492	37172
	Avg. NND	119	64	50	39	34	29
CG Var. III factor: 1.05	Population	20885	26867	31885	35156	38178	40134
	Avg. NND	34	49	38	31	25	25
SSR	No. settlements	10	19	30	40	50	59
	Population	18280	23747	28983	32982	35999	38029
	Avg. NND	99	65	48	38	32	29
GM	No. settlements	12	22	31	40	50	59
	Population	20910	27212	31635	34890	37853	39922
	Avg. NND	61	43	33	28	25	24
DCC	No. settlements	10	–	–	–	55	68
	Population	18185	–	–	–	24512	26657
	Avg. NND	95	–	–	–	23	18

Table 4.1: Results of test runs with a data set of 156 US cities. Population is in thousands, summed over all selected settlements. Average distance to the nearest neighbor is in pixels. For the existing models, it is not always possible to make a selection of a specified size.

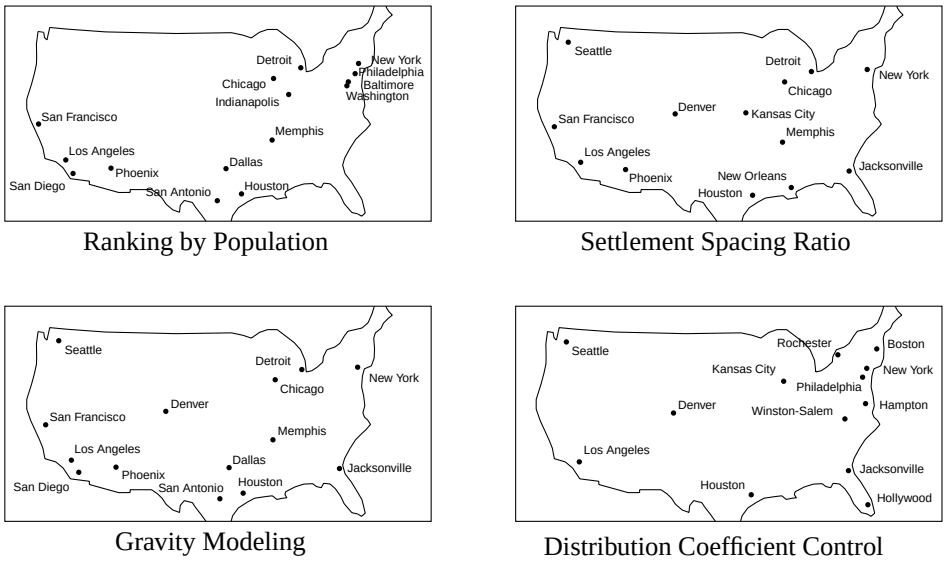
total population size with a greater average distance to the nearest neighbor. Variation II gives the lowest total importance values and large distances to the nearest neighbor, and is outperformed by variation III with factor 5.0, that gives about the same amount of clustering but higher population sizes. Variation III with factor 2.0 is comparable with variation I. Lowering the factor increases the total population sizes and increases the amount of clustering.

Of the existing models, the settlement spacing ratio method combines higher total population values with smaller average nearest neighbor distances, compared to our circle growth variation I. Gravity modeling favors high importance values even more, at the cost

Model		Number of selected settlements					
		10	20	30	40	50	60
Ranking by Population	Population	2986	4267	5246	5992	6634	7181
	Avg. NND	54	35	27	22	21	18
CG	Population	2647	3464	4037	4656	5080	5631
	Avg. NND	50	33	31	24	23	22
CG Var. I	Population	2558	3687	4278	4803	5493	5999
	Avg. NND	84	53	42	36	31	27
CG Var. II	Population	1414	1963	2370	2972	3324	3976
	Avg. NND	100	58	43	37	31	26
CG Var. III factor: 5.0	Population	1802	2333	2803	3358	3877	4889
	Avg. NND	94	59	44	36	32	28
CG Var. III factor: 2.0	Population	2014	3385	4283	4855	5470	6096
	Avg. NND	95	57	42	35	31	26
CG Var. III factor: 1.5	Population	2148	3772	4538	5262	5811	6266
	Avg. NND	93	50	40	33	29	26
CG Var. III factor: 1.05	Population	2977	4256	5245	5946	6613	7172
	Avg. NND	57	34	28	27	23	19
SSR	No. settlements	7	19	29	40	50	59
	Population	2090	3695	4739	5501	5973	6575
	Avg. NND	87	57	36	30	28	24
GM	No. settlements	12	20	31	41	49	61
	Population	3202	4143	5103	5784	6328	6938
	Avg. NND	53	37	32	23	22	20
DCC	No. settlements	13	20	28	40	47	70
	Population	2859	3238	3487	3800	4191	4832
	Avg. NND	67	51	43	34	30	20

Table 4.2: Results of test runs with a data set of 139 municipalities in the Netherlands. Population is in thousands, summed over all selected settlements. Average distance to the nearest neighbor is in pixels. For the existing models, it is not always possible to make a selection of a specified size.

of a higher degree of clustering. But despite the fact that both of these models give quite good individual selections, there is little coherence between selections of different sizes, and some selection sizes are even impossible to obtain. Our methods, which are based on ranking of the settlements, do not suffer from these drawbacks. Finally, the distribution coefficient control method does not seem to give very good results: in most cases, it combines a low total importance value with a small average distance to the nearest neighbor.



Ranking by Population | Settlement Spacing Ratio

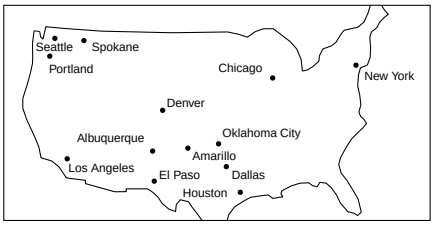
Gravity Modeling | Distribution Coefficient Control

Figure 4.6: Sample output of the existing methods for the dataset of US cities.

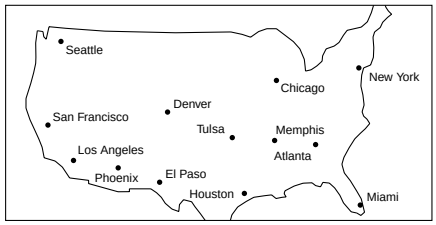
4.5 Conclusions and further research

We developed four new models for the settlement-selection problem and compared them with three existing models. While the existing models compute a single selection, the new models determine a complete ranking of the settlements. After ranking, selecting any number of settlements is easy. Moreover, when selecting more settlements, all previously selected settlements remain selected, which is not the case in the existing models. Using our models, a cartographer can balance between a large total importance and an even distribution of the selected settlements.

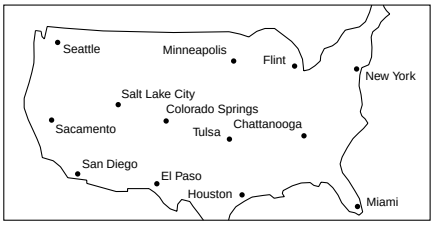
One of the topics for further research is the effects of panning and zooming on the selection. It would also be interesting to develop methods for selection of map features that are not represented by points, such as roads, lakes, and rivers (see also [71, 84, 113, 119, 120]).



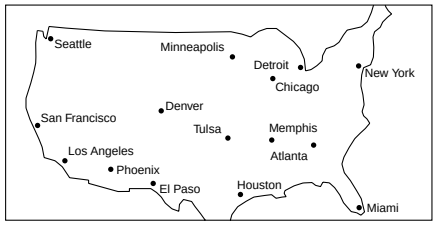
Circle Growth



Circle Growth Var. I



Circle Growth Var. II



Circle Growth Var. III

Figure 4.7: Sample output of the new methods for the dataset of US cities.

Ranking by Population

Settlement Spacing Ratio

Gravity Modeling

Distribution Coefficient Control

Figure 4.8: Sample output of the existing methods for the dataset of Dutch municipalities.

Circle Growth

Circle Growth Var. I

Circle Growth Var. II

Circle Growth Var. III

Figure 4.9: Sample output of the new methods for the dataset of Dutch municipalities.

CHAPTER 5

Facility Location on Terrains

5.1 Introduction

The main advantage of a GIS over a collection of paper maps is the ability to combine the data stored in different layers, and to do all sorts of automated analysis. Geographic analysis is a broad subject, ranging from computing simple numbers such as population densities of the regions of a choropleth map, to complex optimization procedures to determine one or more suitable geographic locations for some specific purpose. Suppose for example that a group of physical geographers are planning to do field studies in mountainous area: where will they set up a base camp and several satellite camps? The actual research is done in day trips to the satellite camps, and in the evening they return to the base camp to eat and sleep. They want to find a location for the base camp that minimizes the maximum distance from the base camp to any of the satellite camps. This is an example of the *facility location problem*, and in this chapter we address this problem in its abstract form. We model the mountainous area as a polyhedral terrain, or a TIN, defined by n triangles (see Section 1.2.2), and assume that a set of m point *sites* on the TIN is given. We assume throughout this chapter that $m \leq n$. The distance between two points on the terrain is the minimum length of any path between those points that lies on the TIN. The *facility center* of the sites is the point on the TIN that minimizes the maximum distance to a site. To avoid problems involving the boundary of the TIN, we show how to extend the TIN to a polyhedron, such that for any two points p and q on the original TIN, any path between p and q that leaves the original TIN cannot be a shortest path. This also enables us to use the results of others on shortest paths on polyhedra.

5.1.1 Previous work and new results

In the Euclidean plane, the facility center, or the center of the *smallest enclosing disc* of a set of m point sites, can be determined in $O(m)$ time. Several algorithms attain this bound. Megiddo [74] gave the first deterministic linear-time algorithm, and a much simpler, linear expected time algorithm was found by Welzl [125].

There is a close connection between the facility center and the furthest-site Voronoi diagram of the sites. Namely, the facility center must lie at a vertex or on an edge of this diagram. In the plane, with Euclidean distance, the furthest-site Voronoi diagram has cells only for the sites on the convex hull of the set of sites, and all cells are unbounded.

It appears that on a polyhedron, some of the properties of furthest-site Voronoi diagrams in the plane no longer hold. For instance, a bisector on the polyhedron is generically a closed curve consisting of as many as $\Theta(n^2)$ straight-line segments and/or hyperbolic arcs, in the worst case. In general, it may also contain two-dimensional portions of the surface of the polyhedron.

Mount [78] showed that the *nearest-neighbor* Voronoi diagram of m sites on (the surface of) a polyhedron with n faces with $m \leq n$ has complexity $\Theta(n^2)$ in the worst case; he also gave an algorithm that computes the diagram in $O(n^2 \log n)$ time. We do not know of any previous work on furthest-site Voronoi diagrams on a polyhedron.

The problem of computing the shortest path between two points along the surface of a polyhedron has received considerable attention; see the papers by Sharir and Schorr [100], Mitchell, Mount and Papadimitriou [77], and Chen and Han [22]. The best known algorithms [22, 77] compute the shortest path between two given points, the source s and destination t, in roughly $O(n^2)$ time. In fact, these algorithms compute a data structure that allows us to compute the shortest path distance between the source s to any query point p in $O(\log n)$ time. The algorithm of Mitchell et al [77] is a continuous version of Dijkstra's algorithm for finding shortest paths in a graph [35], while Chen and Han [22] solve the problem by determining shortest paths in an *unfolding* of the polyhedron; see also [5].

In his master's thesis, van Trigt [121] gave an algorithm that solves the facility location problem on a polyhedral terrain in $O(m^4 n^3 \log n)$ time, using $O(n^2(m^2+n))$ space.

This chapter gives an $O(mn^2 \log^2 m \log n)$ time algorithm to compute the furthest-site Voronoi diagram and find the facility center for a set S of m sites on the surface of a polyhedron with n faces. Given the linear-time algorithm for finding the facility center in the plane, this bound may seem disappointing. However, the algorithm for computing the furthest-site Voronoi diagram is near-optimal, as the combinatorial complexity of the diagram is $\Theta(mn^2)$.

5.2 Extending a TIN to a polyhedron

In many practical situations, a TIN is defined over a rectangle. More precisely, it is the graph of a piecewise linear function defined over $[x_{\text{left}}, x_{\text{right}}] \times [y_{\text{bottom}}, y_{\text{top}}]$. To avoid complications involving the boundary of the TIN, and to be able to use results of others on shortest paths and Voronoi diagrams on polyhedra, we extend the terrain to the surface of a polyhedron.

Any TIN consisting of n triangles can be extended with $O(n)$ additional triangles to the surface of a polyhedron that is homeomorphic to a sphere, such that for any two points p, q on the triangles of the original TIN, any path from p to q that leaves the original TIN cannot be a shortest path on the polyhedron. The construction is as follows:

The polyhedron will be shaped somewhat like a box, with the original TIN 'on top' (see Figure 5.1). Let d be an upper bound on the length of the shortest path between two points on the original TIN. First, we extend the domain of the TIN to $[x_{\text{left}} - d, x_{\text{right}} + d] \times [y_{\text{bottom}} - d, y_{\text{top}} + d]$. From each vertex on the original boundary of the TIN, we start a new edge, perpendicular to this boundary, and ending at the new boundary. We let all these new edges be horizontal, in other words, normal to the z-axis. Next, the resulting new rectangles are triangulated by adding a diagonal edge. So far, we have constructed the top of the polyhedron. (Note that the edges on the new boundary are not all horizontal, unless the edges on the original boundary are horizontal.)

Let z_{low} be the z-value of the lowest vertex in the terrain. The 'bottom' of the polyhedron is the rectangle $([x_{\text{left}} - d, x_{\text{right}} + d] \times [y_{\text{bottom}} - d, y_{\text{top}} + d], z_{\text{low}} - 1)$.

From the vertices on the boundary of the top of the polyhedron, we start edges parallel to the z-axis, and ending at the boundary of the bottom rectangle. The resulting vertical rectangles are triangulated by adding diagonal edges. Finally, we place a vertex in the interior of the bottom rectangle, and connect it with edges to all vertices on the boundary of the bottom rectangle. The resulting polyhedron is highly degenerate, but our algorithm is not influenced by these degeneracies.

Because of the dimensions of the top of the polyhedron, no shortest path from p to q, both on the original TIN, can cross a triangle on the sides or the bottom of the polyhedron. For any path from p to q that only crosses triangles on the top of the polyhedron, the maximal sub-paths that cross only new triangles on top of the polyhedron can be replaced by shorter paths along the boundary of the original TIN.

5.3 The complexity of the furthest-site Voronoi diagram on a polyhedron

Previous papers on shortest paths on polyhedra [22, 77, 100, 121] use a number of important concepts that we'll need as well. We review them briefly after giving the relevant definitions.

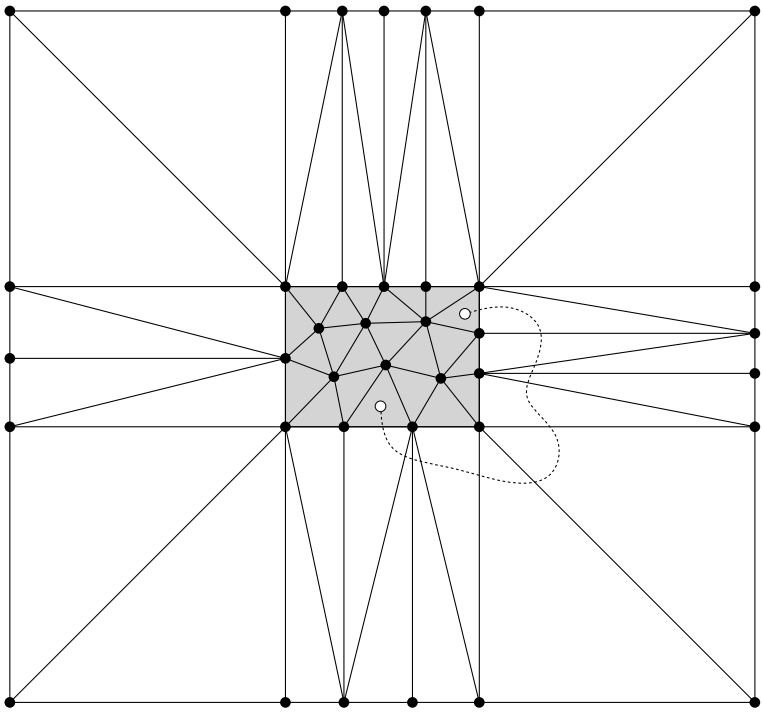


Figure 5.1: Extending a TIN (shaded) to the top of a 'box-like' polyhedron.

In the remainder of this chapter, P is (the surface of) a polyhedron. As stated before, we only allow polyhedra homeomorphic to a ball, so that their surfaces are homeomorphic to a sphere. For two points p and p' on the surface of P, we define the distance $\mathrm{d}(p, p')$ to be the length of the shortest path from p to p' along the surface of P. Let S be a set of m point sites on P. Consider first a single site $s \in P$. For any point p on P we consider a shortest path from p to s; note that in general such a path need not to be unique. Such a shortest path has a number of properties. First, if it crosses an edge of P properly, then a principle of refraction holds. This means that if the two incident triangles were pivoted about their common edge to become co-planar, then the shortest path would cross the edge as a straight-line segment. This principle is called *unfolding*. For any vertex on the polyhedron, we define its *total angle* as the sum of the angles at that vertex in each of the triangles incident to it. The shortest path cannot contain any vertex for which the total angle is less than 2π, except possibly at the source p and the target s.

Any shortest path crosses a sequence of triangles, edges, and possibly, vertices. If two shortest paths on the polyhedron cross the same sequence (in the same order), we say that these paths have the same *edge sequence*. If a shortest path from p to s contains a vertex

of the polyhedron, the vertex reached first from p is called the *pseudoroot* of p. If the path does not contain any vertex, then site s is called the pseudoroot of p.

The *shortest path map (SPM) of s* is defined as the subdivision of P into connected regions where the shortest path to s is unique and has a fixed edge sequence. For non-degenerate placements of s, the closures of the regions cover P, so the portion of P outside any region, where more than one shortest path to s exists, consists of one-dimensional pieces. When two pseudoroots have the same distance to s, the exterior of the regions of the SPM may have two-dimensional parts.

It is known that the shortest path map of a site has complexity $O(n^2)$; this bound is tight in the worst case. The SPM restricted to a triangle is actually the planar Euclidean Voronoi diagram for a set of pseudo-sites with additive weights (see Figure 5.2). The pseudo-sites are obtained from the pseudoroots. The coordinates of the pseudo-sites are obtained by unfolding the triangles in the edge sequence to the pseudoroot so that they are all co-planar. The weight of a pseudo-site is the shortest-path distance from the corresponding pseudoroot to the site s. It follows that the boundaries of regions in the SPM within a triangle consist of straight-line segments and/or hyperbolic arcs. For any point on a hyperbolic arc or a segment there are two shortest paths to s with different pseudoroots.

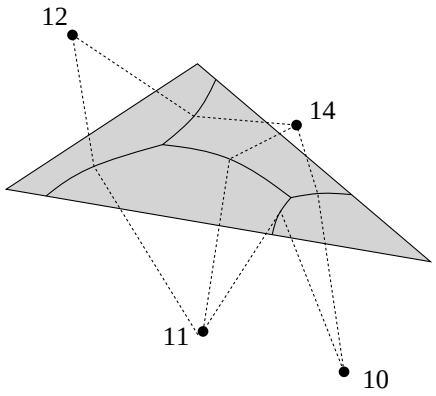


Figure 5.2: The SPM restricted to a triangle is the Euclidean Voronoi diagram for a set of sites with additive weights.

Given two sites s and t on the polyhedron, the *bisector* $\beta(s,t)$ is the set of those points p on the polyhedron whose shortest path to s has length equal to the shortest path to t. The bisector consists of straight-line segments, hyperbolic arcs, and may even contain two-dimensional regions. Such regions occur only when two sites have exactly the same distance to some vertex of P. For simplicity, we assume that these degeneracies don't occur.

The *closest-site Voronoi diagram* of a set S of m sites on P, denoted by $\mathrm{VD}(S)$, is a planar graph embedded in P that subdivides P into maximal open regions associated with the sites in S, with the property that a point $p \in P$ lies in the region of a site $s \in S$ if and only

if $d(p,s) < d(p,s')$ for each $s' \in S$ with $s' \neq s$. The interior of the boundary between two adjacent regions is an *edge* of the Voronoi diagram; it is easy to see that each edge lies on a bisector of two sites in S. The non-empty intersections of the closures of three or more regions of the Voronoi diagram are its *vertices*. We assume that all vertices have degree three; otherwise, a degeneracy is present.

The *furthest-site Voronoi diagram* of a set S of m sites on P is a similar subdivision of P into maximal open regions. The difference is that a point $p \in P$ lies in the region of a site $s \in S$ if and only if $d(p,s) > d(p,s')$ for each $s' \in S$ with $s' \neq s$. In this chapter, we give a new algorithm for computing the furthest-site Voronoi diagram of a set S of sites on a polyhedron. We denote it by $FVD(S)$, and refer to it more loosely as *the diagram*. The region of a site $s \in S$, either $VD(S)$ or $FVD(S)$, is denoted by $\mathcal{R}(s)$.

The following facts are crucial for the algorithm below to work and for the analysis to hold. Lemmas 5, 6, and 7 are similar to the lemmas in Leven and Sharir [69]; they are general statements about a large class of metrics and hold under very general conditions.

Lemma 5 *In the closest-site Voronoi diagram of a set S of sites on P, the region $\mathcal{R}(s)$ of a site $s \in S$ is connected.*

Proof: Let p be a point in $\mathcal{R}(s)$, let $\pi(p,s)$ be a shortest path from p to s, and let p' be an arbitrary point on $\pi(p,s)$. The sub-paths $\pi(p,p'), \pi(p',s) \subset \pi(p,s)$ are also shortest paths, and $d(p,s) = d(p,p') + d(p',s)$. It follows that $d(p',s) < d(p',t)$ for any $t \in S, t \neq s$; otherwise, there would be a path from p to t via p' that is shorter than $d(p,s)$, contradicting the fact that p is closer to s than to t. Hence, any point p' on $\pi(p,s)$ lies in $\mathcal{R}(s)$, and any two points p and q in $\mathcal{R}(s)$ are connected via s by a path that lies completely in $\mathcal{R}(s)$. □

Lemma 6 *Bisector $\beta(s,t)$ is connected and homeomorphic to a circle.*

Proof: Consider the closest-site Voronoi diagram of $\{s,t\}$. The closures of $\mathcal{R}(s)$ and $\mathcal{R}(t)$ in this Voronoi diagram cover the whole surface of the polyhedron, and, by the previous lemma, both $\mathcal{R}(s)$ and $\mathcal{R}(t)$ are connected. Since P is homeomorphic to a sphere, $\beta(s,t)$, which is the boundary between $\mathcal{R}(s)$ and $\mathcal{R}(t)$, must be connected and homeomorphic to a circle. □

Lemma 7 *For any three distinct sites s,t, and u, bisectors $\beta(s,t)$, and $\beta(s,u)$ intersect at most twice.*

Proof:

Consider the closest-site Voronoi diagram of $\{s,t,u\}$. At an intersection χ of $\beta(s,t)$ and $\beta(s,u)$, we have that $d(\chi,s) = d(\chi,t) = d(\chi,u)$. Therefore, χ also lies on the third bisector $\beta(t,u)$, and χ is a vertex of the Voronoi diagram of $\{s,t,u\}$, incident to $\mathcal{R}(s)$, $\mathcal{R}(t)$ and $\mathcal{R}(u)$.

Now suppose that the bisectors $\beta(s,t)$ and $\beta(s,u)$ (and consequently $\beta(t,u)$) intersect at least three times. Let these intersections be χ_1, χ_2, and χ_3. Look at the ε-neighborhoods of the three intersections. We can choose ε small enough to assert that none of s, t or u lies in any ε-neighborhood. Each of the intersections is incident to $\mathcal{R}(s)$, $\mathcal{R}(t)$ and $\mathcal{R}(u)$, and, by Lemma 5, these regions are connected. Therefore, there is a shortest path from s to each of χ_1, χ_2, and χ_3, and the interior of each of those shortest paths lies completely in $\mathcal{R}(s)$. The same holds for t and u. But no two of the nine shortest paths may cross, which is impossible, since a $K_{3,3}$ is not realizable as a planar graph (see Figure 5.3). Hence, it follows that bisectors $\beta(s,t)$, $\beta(s,u)$ intersect at most twice.

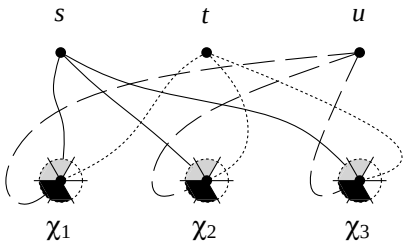


Figure 5.3: Shortest paths from s, t, and u to χ_1, χ_2, and χ_3.

□

Any family of simple closed curves (in this case, on a topological sphere) of which every pair crosses at most twice is called a *family of pseudocircles.* Thus for every fixed $s \in S$, the bisectors $\{\beta(s,t) : t \neq s\}$ form a set of pseudocircles. Every pseudocircle in such a set partitions the surface of the polyhedron into two connected two-dimensional regions, or pseudodisks. We call the region that contains s the *interior* (with respect to s) of a pseudocircle; the region not containing s is called the *exterior*.

Lemma 8 *Let $\mathcal{B}$ be a set of pseudocircles on the surface of a simple polyhedron P. If the common interior of the pseudocircles in $\mathcal{B}$ is non-empty, then the common exterior of the pseudocircles in $\mathcal{B}$ is connected.*

Proof: Suppose for the sake of contradiction that the common interior of the pseudocircles in $\mathcal{B}$ is non-empty, and that the common exterior of the pseudocircles in $\mathcal{B}$ is not connected. Let $\mathcal{B}' \subseteq \mathcal{B}$ be a minimal subset of pseudocircles such that the common exterior of the pseudocircles in $\mathcal{B}'$ consists of at least two connected regions R_1, R_2. Observe that R_1 must be incident to all pseudocircles in $\mathcal{B}'$. Otherwise, the removal of a pseudocircle not incident to R_1 would leave R_1 unchanged and can only enlarge R_2, but it cannot join the two regions, and this contradicts the minimality of $\mathcal{B}'$. Analogously, R_2 must be incident to all pseudocircles in $\mathcal{B}'$. Also observe that all pseudocircles in $\mathcal{B}'$ must intersect: a pseudocircle that lies completely in the interior of another one is not incident to R_1

and R_2, and can be removed without affecting the two regions. Again, this contradicts the minimality of $\mathcal{B}'$.

Let p_1 and p_2 be two arbitrary points in the interiors of R_1 and R_2, respectively. The fact that p_1 and p_2 lie in different components of the common exterior of the pseudocircles in $\mathcal{B}'$ means that there exists a closed path in the union of the interiors of the pseudocircles in $\mathcal{B}'$ that separates p_1 and p_2. Indeed, the situation must be as depicted in Figure 5.4: each pseudocircle can intersect at most two other pseudocircles. Otherwise, if the incidence graph of the pseudocircles would contain a chord, we could drop at least one of the pseudocircles and still find a closed path in the union of the interiors of the remaining pseudocircles that separates p_1 and p_2. On the other hand, the incidence graph of the pseudocircles in $\mathcal{B}'$ must be a complete graph, since their common intersection is non-empty and no pseudocircle in $\mathcal{B}'$ lies in in the interior of another one.

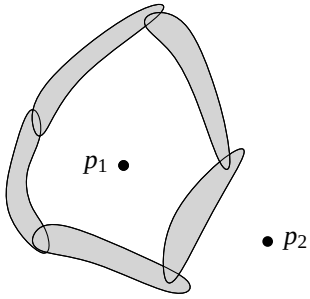


Figure 5.4: Pseudocircles separating p_1 from p_2.

It follows that the number of pseudocircles in $\mathcal{B}'$ is at most three, and inspection of all topologically different arrangements of at most three pseudocircles shows that it is not possible for the pseudocircles to attain a common exterior of two or more connected regions if their common interior is non-empty. □

Lemma 9 *Bisector $\beta(s,t)$ consists of $O(n^2)$ straight-line segments and hyperbolic arcs.*

Proof: The claim follows directly from the fact that the Voronoi diagram of m sites on a polyhedron with n faces with $m \leq n$ has complexity $\theta(n^2)$ in the worst case; see the paper by Mount [78]. □

Since the edges of the closest- and furthest-site Voronoi diagram lie on the bisectors of two sites in S, each edge also consists of $O(n^2)$ line segments and hyperbolic arcs. To simplify our exposition, the intersections between two adjacent segments or arcs on the edges are referred to as *breakpoints*, as opposed to the *vertices* of the diagram that we defined before. We consider the point where a bisector crosses an edge of P also to be a breakpoint.

Lemma 10 *The furthest-site Voronoi diagram FVD(S) of a set S of m sites on a polyhedron has $O(m)$ cells, vertices, and edges.*

Proof: Let $\mathcal{R}_{s>t}$ be the region of points that are further away from s than from t, for $s,t \in S$. In this notation $\mathcal{R}(s) = \bigcap_{t\in S, t\neq s} \mathcal{R}_{s>t}$. From Lemma 7 it follows that this intersection is the common exterior of set of pseudo-disks that all contain s. From Lemma 8 it follows that this common exterior is connected. So we have at most one cell (region) for each site in S, and each vertex of the diagram has degree at least three. By Euler's relation for planar graphs, the number of vertices and edges of FVD(S) is also $O(m)$. □

We define the *total complexity* of FVD(S) to be the sum of the number of vertices and breakpoints in FVD(S).

Lemma 11 *The maximum total complexity of FVD(S) is $\Theta(mn^2)$.*

Proof: Each edge of FVD(S) is part of some bisector $\beta(s,t)$ for two sites $s,t \in S$. Consequently, the upper bound follows from Lemmas 10 and 9.

As for the lower bound, we describe a construction that shows that FVD(S) for a set S of m point sites on a non-convex polyhedron P with $O(n)$ edges can have total complexity $\Omega(mn^2)$. The construction will focus on proving an $\Omega(mn)$-bound for a single edge of P. It is described for point sites in the plane with obstacles. This can then be "lifted" to a non-convex polyhedron.

First we will describe the location of the sites, then the obstacles. Assume that $|S|$ is even; we split S into S_1 and S_2 with $k = m/2$ points each. Figure 5.5 shows the configuration of the sites $S_1 = \{s_1, \ldots, s_k\}$ (in the figure, $k = 5$). For ease of description, we also specify two additional points s_0 and s_{k+1}; these are *not* sites. The sites $s_1, \ldots, s_k \in S_1$ and the points s_0 and s_{k+1} are placed equally spaced on the lower semi-circle of a circle $\mathcal{C}_1$. For $1 \leq i \leq k+1$, let b_{i-1} be the point where the bisector $\beta(s_{i-1}, s_i)$ meets the upper semi-circle of $\mathcal{C}_1$. Note that any point on the arc of the upper semi-circle $\mathcal{C}_1$ between b_{i-1} and b_i is further away from s_i than from any other site in S_1. Let γ_i denote the cone originating at site s_i that is bounded by the rays ray(s_i, b_{i-1}) and ray(s_i, b_i). The portion of the cone γ_i that lies outside $\mathcal{C}_1$ is further away from s_i than from any other site in S_1. Figure 5.5 only shows the cones γ_2, γ_3 and γ_4.

Let ℓ be a horizontal line lying some distance above the circle $\mathcal{C}_1$. The second set of sites $S_2 = \{s'_1, \ldots, s'_k\}$ is obtained by reflecting the set S_1 through ℓ. That is, s'_i is such that ℓ is the bisector of s_i and s'_i. The points in S_2 lie on a circle $\mathcal{C}'_1$ which is the reflection of $\mathcal{C}_1$. The cone γ'_i is defined analogously and is the reflection of γ_i. Let ℓ_i be the intersection of cone γ_i and ℓ. Note that ℓ_i is also the intersection of γ'_i and ℓ.

We have specified the point sites. Now we will specify the location of the obstacles. The important fact is that the cones $\gamma_i, \ldots, \gamma_k$ have a common intersection around the center of circle $\mathcal{C}_1$. Let $\mathcal{C}_2$ be a small circle lying within this common intersection, and let the segment $\overline{ab}$ be the horizontal diameter of $\mathcal{C}_2$. Figure 5.5 (detail) shows the circle $\mathcal{C}_2$ and

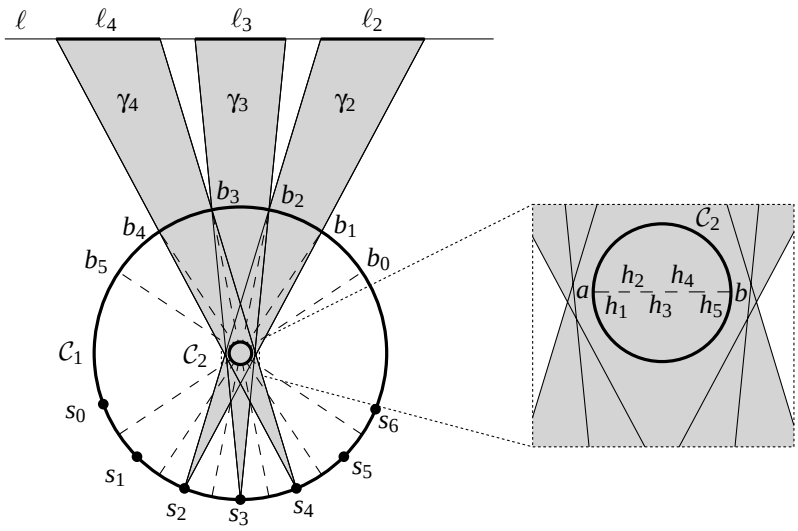


Figure 5.5: The configuration of S_1 and the obstacles in $\mathcal{C}_2$ (detail).

the segment $\overline{ab}$. Let $\overline{a'b'}$ be the reflection of $\overline{ab}$ through ℓ. Our obstacle set will be the segments $\overline{ab}$ and $\overline{a'b'}$ minus a number of narrow holes (through which a path can pass). The segment $\overline{ab}$ has an evenly spaced set $h_1,\ldots,h_n$ of narrow holes. The segment $\overline{a'b'}$ also has an evenly spaced set $h'_1,\ldots,h'_n$ of narrow holes; the only difference is that these holes are slightly shifted to the left.

We specified all the points and obstacles. Now, we will argue that the line ℓ is intersected by $k = m/2$ edges of FVD(S), each of which crosses ℓ $\Omega(n)$ times. Let us focus on the portion ℓ_i of the line ℓ. Since any point in ℓ_i is further away from s_i (resp. s'_i) than from any other site in S_1 (resp. S_2), s_i and s'_i are the only relevant sites for FVD(S) near ℓ_i. We will now argue that $\beta(s_i,s'_i)$ crosses ℓ $\Omega(n)$ times. For $1 \le j \le n$, let $p_{i,j}$ (resp. $p'_{i,j}$) be the point of intersection of the line through s_i (resp. s'_i) and h_j (resp. h'_j) and the line ℓ. Because of the horizontal shift of the holes in $\overline{a'b'}$, the points occur interleaved on ℓ_i as the sequence $p'_{i,1}, p_{i,1}, p'_{i,2}, p_{i,2}, \ldots, p'_{i,n}, p_{i,n}$. This is illustrated in Figure 5.6 for ℓ_2. For $1 \le j \le n$, since s_i can "see" $p_{i,j}$ whereas s'_i cannot, there is a neighborhood around $p_{i,j}$ that is closer to s_i than to s'_i. By symmetric reasoning, there is a neighborhood around $p'_{i,j}$ that is closer to s'_i than to s_i. It follows that the bisector $\beta(s_i,s'_i)$ must cross ℓ_i between $p'_{i,j}$ and $p_{i,j}$, and also between $p_{i,j}$ and $p'_{i,j+1}$. Thus, $\beta(s_i,s'_i)$ crosses ℓ_i $\Omega(n)$ times, as illustrated in Figure 5.6.

One gets $\Omega(kn) = \Omega(mn)$ crossings for line ℓ, since one gets $\Omega(n)$ crossing for each ℓ_i. The pattern can be repeated on n lines parallel to ℓ and sufficiently close to ℓ. This gives $\Omega(mn)$ crossings for each of the n lines. The sites and the obstacles can be perturbed to

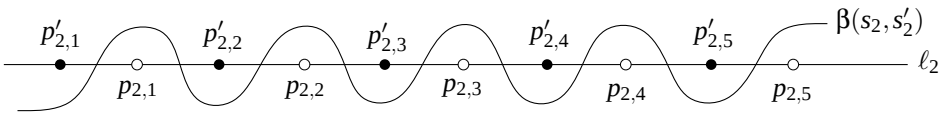


Figure 5.6: Detail of $\beta(s_2, s'_2)$.

a general position without affecting the lower bound complexity. By treating the lines as edges on a polyhedron, and 'raising vertical cylinders' with the obstacles as bases, we can get the $\Omega(mn^2)$ bound for the total complexity of FVD(S) on a polyhedron. □

The facility center of S can be found by traversing the edges of FVD(S), and determining for each elementary arc or line segment of each edge the maximum distance to the two sites of the regions on both sides of the edge. These distances can be computed in $O(1)$ time, and the maximum of all these distances determines the location of the facility center. Since FVD(S) has maximum total complexity $O(mn^2)$, we obtain the following.

Corollary 4 *Given FVD(S), the facility center of S can be computed in $O(mn^2)$ time.*

5.4 Computing the furthest-site Voronoi diagram

In this section, we describe our algorithm for computing the furthest-site Voronoi diagram of the given set S of m sites on the surface of polyhedron P, consisting of n triangles. Our algorithm uses ideas from the algorithm of Ramos [87] for computing the intersection of unit spheres in three dimensions. We first give an outline of the algorithm, and get into the details in the subsequent subsections.

The algorithm for computing FVD(S) works as follows:

- If $|S| = 1$, then FVD(S) is the whole surface of the polyhedron. If $|S| = 2$, compute the closest-site Voronoi diagram (which is equivalent to the furthest-site Voronoi diagram) with the algorithm of Mount [78] in $O(n^2 \log n)$ time.
- Otherwise, if $|S| \geq 3$, subdivide S into two subsets R (the *red* sites) and B (the *blue* sites) of about equal size, i.e., $|R| = \lfloor |S|/2 \rfloor$, and $|B| = \lceil |S|/2 \rceil$.
- Recursively compute FVD(R) and FVD(B).
- Merge FVD(R) and FVD(B) into FVD$(R \cup B) =$ FVD(S) as follows:
  - Determine the set of sites $R_0 \subseteq R$ that have a non-empty region in FVD(R), i.e., FVD$(R) =$ FVD(R_0). Observe that the remaining sites in $R \setminus R_0$ don't influence the final diagram. Similarly, compute $B_0 \subseteq B$.

- Determine an *low-degree independent set* $R_0' \subset R_0$, which is a subset with the property that the region of a site $s \in R_0'$ has at most 9 neighbors in $\mathrm{FVD}(R_0)$, and no two sites $s, s' \in R_0'$ are neighbors in $\mathrm{FVD}(R_0)$. (Two sites are said to be *neighbors* if their regions share an edge of the diagram.) Compute $R_1 = R_0 \setminus R_0'$ and $\mathrm{FVD}(R_1)$, and repeat this step to generate a Dobkin-Kirkpatrick hierarchy [37] $R_0 \supset R_1 \supset \ldots \supset R_k$ and their furthest-site Voronoi diagrams, such that R_k has only a constant number of sites. Do the same for the blue sites to achieve $B_0 \supset B_1 \supset \ldots \supset B_l$ and their furthest-site Voronoi diagrams. See Section 5.4.2 for details.
- Compute $\mathrm{FVD}(R_i \cup B_l)$ for $0 \leq i \leq k$, exploiting the fact that B_l has only a constant number of sites. Similarly, compute $\mathrm{FVD}(R_k \cup B_j)$ for $0 \leq j \leq l$. This is the *basic merge step*. See Section 5.4.3 for details.
- Compute $\mathrm{FVD}(R_i \cup B_j)$ from $\mathrm{FVD}(R_i \cup B_{j+1})$ and $\mathrm{FVD}(R_{i+1} \cup B_j)$. This is the *generic merge step*, which when repeated gives $\mathrm{FVD}(R_0 \cup B_0) = \mathrm{FVD}(S)$. See Section 5.4.4 for details.

During the construction of $\mathrm{FVD}(S)$, we create standard and furthest-site Voronoi diagrams of subsets of S as intermediate structures. We maintain $\mathrm{FVD}(S)$ and these intermediate structures as doubly connected edge lists (see Section 1.3.1), to be able to efficiently determine and preserve topological relations between Voronoi regions, edges, and vertices.

5.4.1 Edge tracing

Several stages of the algorithm for constructing $\mathrm{FVD}(S)$ involve the computation of new Voronoi cells of $\mathrm{FVD}(S')$ for $S' \subseteq S$, or the modification of existing Voronoi cells. A basic step is the generation of Voronoi edges or parts of Voronoi edges. Recall that the edges of $\mathrm{FVD}(S')$ lie on the bisectors of sites in S', and consist of $O(n^2)$ hyperbolic arcs and/or straight line segments. To generate an edge e that is incident to the regions of $s_i, s_j \in S'$, we need to know a starting point of the edge (i.e., the location of one of the vertices of $\mathrm{FVD}(S')$ incident to e), and an endpoint (i.e., the location of the other vertex incident to e). We calculate the bisector $\beta(s_i, s_j)$ in $O(n^2 \log n)$ time using the algorithm of Mitchell et al. [77]. We store it as a doubly linked list of hyperbolic arcs and straight line segments, such that we can traverse it in two directions. Next, we traverse $\beta(s_i, s_j)$, until we reach the starting point of e. From that point on, we output the hyperbolic arcs and straight line segments of which e consists, until we reach the endpoint of e. Traversing $\beta(s_i, s_j)$ takes $O(n^2)$ time, and testing whether we have reached the starting point or the endpoint of e can be done in $O(1)$ time for each elementary hyperbolic arc or straight line segment. Hence, the total time needed to generate an edge of $\mathrm{FVD}(S')$ is $O(n^2 \log n)$. The amount of memory needed is bound by the size of the shortest path maps of s_i and s_j, and of $\beta(s_i, s_j)$, which is $O(n^2)$. These results are summarized in the following lemma:

Lemma 12 *Given a set of sites S on a polyhedron P with n triangles and the two vertices incident to an edge e of FVD(S') for $S' \subseteq S$, e itself can be computed in $O(n^2 \log n)$ time using $O(n^2)$ memory.*

Suppose that we have generated all the edges of the region of $s_i \in S'$, including e, the common edge of the regions of s_i and s_j. Later on in the algorithm, we may have to generate the edges of the region of s_j, including e. This poses two problems. First, it means that we have to compute the bisector of s_i and s_j again. However, since each edge is incident to two regions, this doesn't influence the asymptotic running time and memory requirements of the algorithm. A more serious problem is that e would be generated twice. We avoid that by calculating the bisector of two sites s_i and s_j only once, and storing a pointer to it in a two-level binary search tree; the first level is a binary search tree on the smallest index of the two sites that constitute the bisector, and the second level is a binary search tree on the largest index of the two sites. Now, if we have to generate an edge e that lies on the bisector of s_i and s_j, with $i < j$, we first search for i in the highest level of the search tree; if there is an entry for i, we search for j in the next level. If there is an entry for j in the second level, this gives us a pointer to $\beta(s_i, s_j)$. If we don't find the bisector, we calculate it as described above, and insert a pointer to it in the search structure. The total size of the search structure is linear in the number of bisectors that we compute; we will show this to be $O(m \log m)$. Hence, querying the search structure and inserting new pointers can be done in $O(\log m)$ time per operation. Since we assume that $m < n$, generating an edge still takes $O(n^2 \log n)$ time using $O(n^2)$ memory, regardless of whether we have to calculate its supporting bisector, or simply look it up in the search structure.

Keeping the bisectors serves a second purpose: when we output a hyperbolic arc or straight line segment during the edge generation, we set pointers from the arc or segment on the edge to the corresponding arc or segment on the bisector, and vice versa. This means that we can access the neighbors of a region in $\mathrm{FVD}(S')$ in $O(1)$ time via the supporting bisectors of their common edges, which facilitates maintaining the proper topological relations between regions during the construction of the Voronoi diagrams.

In some cases we need a variation on the edge tracing procedure. As before, we compute a bisector of two sites or look it up, traverse it until we find the starting point of the edge that is to be generated, and output hyperbolic arcs and straight line segments from that moment on. The difference is that we don't have a single endpoint at which we stop the tracing, but a constant number of candidate endpoints, and we stop when we have reached the first of these. Testing whether we have reached any of the candidate endpoints takes $O(1)$ time per hyperbolic arc or straight line segment, and this edge tracing variation also requires $O(n^2 \log n)$ time and $O(n^2)$ memory per Voronoi edge.

5.4.2 Constructing the hierarchy for R_0 and B_0.

We describe how to compute the hierarchy $R_0 \supset R_1 \supset \ldots \supset R_k$ and their furthest-site Voronoi diagrams; for B_0, this is done analogously. The computation is similar to the Dobkin-Kirkpatrick hierarchy construction [37].

Let G_0 be the dual graph of $\mathrm{FVD}(R_0)$, i.e., $G_0 = (R_0, E_0)$, with $(s_i, s_j) \in E_0$ for $s_i, s_j \in R_0$ if the regions of s_i and s_j share an edge in $\mathrm{FVD}(R_0)$. Note that $\mathrm{FVD}(R_0)$ and G_0 are planar graphs. An *independent set* of vertices in a graph $G = (V, E)$ is a set $V' \subset V$ such that

there is no edge (v_i, v_j) in E for any $v_i, v_j \in V'$. Snoeyink en van Kreveld [106] showed that for any planar graph $G = (V, E)$ with n vertices, an independent set V' of vertices of degree at most 9 with $|V'| \geq |V|/6$ can be found in $O(n)$ time. We apply this to G_0 to find an independent set $R'_0 \subset R_0$ in $O(|R_0|)$ time. A site $s \in R'_0$ has at most 9 neighbors in $\mathrm{FVD}(R_0)$, no two sites $s, s' \in R'_0$ are neighbors in $\mathrm{FVD}(R_0)$, and $|R'_0| \geq |R_0|/6$.

To compute $\mathrm{FVD}(R_1) = \mathrm{FVD}(R_0 \setminus R'_0)$, we remove the sites in R'_0 one at a time from R_0 and update the diagram after the removal of each site. Let s_0 be such a site in R'_0, and let p be a point that lies in the region in $\mathrm{FVD}(R_0)$ of s_0. After updating the diagram, p must lie in the region of a site s' that is a neighbor of s_0 in $\mathrm{FVD}(R_0)$. So the region of s_0 is divided among its neighbors, of which there are only a constant number, and all diagram edges in that region lie on the bisectors of those neighbors (see Figure 5.7).

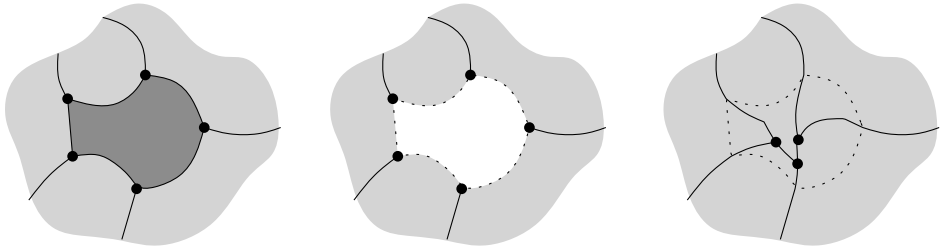


Figure 5.7: Removing a site s_0, and dividing its region among its neighbors.

Let $v_1, \ldots, v_k$ be the at most 9 vertices of the region of s_0 in clockwise order. Let $e_1, \ldots, e_k$ be the edges of $\mathcal{R}(s_0)$, with e_i incident to v_i and v_{i+1} for $1 \leq i < k$, and with e_k incident to v_k and v_1. Finally, let s_i be the neighbor of s_0 whose region is incident to e_i for $1 \leq i \leq k$. See Figure 5.8.

We will describe how to reconstruct $\mathcal{R}(s_1)$ after the removal of s_0; for $\mathcal{R}(s_2), \ldots, \mathcal{R}(s_k)$ this is similar. Edge e_1 is no longer an edge of $\mathcal{R}(s_1)$, and it is removed. Vertices v_1 and v_2 are also removed; the edges of $\mathcal{R}(s_1)$ incident to these vertices (e'_1 and e'_2) have to be extended into the region of s_0. Recall that e'_1 lies on the bisector $\beta(s_1, s_k)$ of s_1 and s_k, and e'_2 lies on $\beta(s_1, s_2)$. We extend e'_1 by tracing $\beta(s_1, s_k)$ as described in Section 5.4.1, starting at the location of v_1. At some point p on the bisector, the distance between p and s_1 and the distance between p and s_k equals the distance between p and some other neighbor s_i of s_0. At this point p we reach a new vertex of $\mathcal{R}(s_1)$. Note that this vertex is also a vertex of the standard Voronoi diagram of s_1, s_k and s_i. We have finished the reconstruction of e'_1 at this point, record the new vertex of $\mathcal{R}(s_1)$, and proceed with the next edge of $\mathcal{R}(s_1)$ by tracing $\beta(s_1, s_i)$. This is repeated until we finally trace $\beta(s_1, s_2)$ and end up at the location of v_2, which concludes the reconstruction of $\mathcal{R}(s_1)$.

To determine whether we have reached a vertex during the edge tracing, we precompute

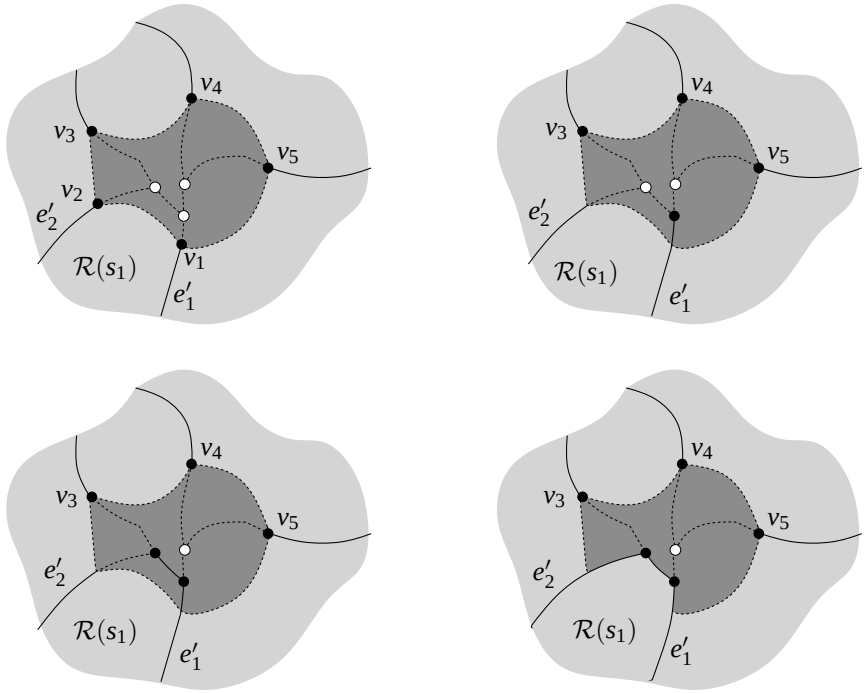


Figure 5.8: The reconstruction of $\mathcal{R}(s_1)$ after the removal of s_0.

all the points p that are equidistant to three neighbors of s_0. For a fixed triple of neighbors of s_0, these points p are the $O(1)$ vertices of the standard Voronoi diagram of the triple, which can be computed in $O(n^2 \log n)$ time again with the techniques from Mitchell et al. [77] and Mount [78]. Computing the Voronoi diagrams for all triples of the at most 9 neighbors of s_0 takes asymptotically the same amount of time. We trace a constant number of edges, and for each hyperbolic arc or straight edge on these edges we determine in constant time (by testing the $O(1)$ precomputed vertices of the Voronoi diagrams of triples of neighbors of s_0) whether we have reached the endpoint of the edge that we are generating. It follows that the removal of a single site and the reconstruction involved takes $O(n^2 \log n)$ time.

After all the sites in R_0' have been removed from R_0 and $\mathrm{FVD}(R_1)$ has been constructed, we recursively repeat the procedure of removing an independent set of sites to create $\mathrm{FVD}(R_2), \ldots, \mathrm{FVD}(R_k)$. The total number of diagrams we construct this way is $O(\log m)$.

Since $\sum_{i=0}^{k} |R_i|$ is a geometric series, the total time for computing all independent sets is $O(m)$. The the reconstruction of the diagram after the removal of a single site from takes $O(n^2 \log n)$ time, and the total number of sites removed is less than m. It follows that the

construction of the the hierarchy $R_0 \supset R_1 \supset \ldots \supset R_k$ and their furthest-site Voronoi diagrams takes $O(mn^2 \log n)$ time in total. By Lemma 11, the size of $\mathrm{FVD}(R_i) = O(|R_i|n^2)$. Therefore, the total size of all bisectors and diagrams constructed is $O(mn^2)$.

5.4.3 The basic merge step

In the basic merge step, we compute $\mathrm{FVD}(R_i \cup B_l)$ for $0 \leq i \leq k$, and $\mathrm{FVD}(R_k \cup B_j)$ for $0 \leq j \leq l$. We will exploit the fact that B_l and R_k contain only a constant number of sites. We will only describe the computation of $\mathrm{FVD}(R_i \cup B_l)$ for a fixed i; all other diagrams are computed similarly.

- For each site $r \in R_i$ and $b \in B_l$, we compute the region of r in $\mathrm{FVD}(\{r,b\})$. To do this, we compute the *closest-site* Voronoi diagram for sites r and b using the $O(n^2 \log n)$ algorithm of Mitchell et al. [77, 78]. The region of r in $\mathrm{FVD}(\{r,b\})$ is clearly the region of b in the closest-site diagram. The total time for all pairs r and b is $O(|R_i|n^2 \log n)$, since there are only $O(|R_i|)$ pairs.

- Next, we compute the region of each site $r \in R_i$ in $\mathrm{FVD}(\{r\} \cup B_l)$ by successively intersecting the regions of r in $\mathrm{FVD}(\{r,b\})$ over all $b \in B_l$ with a line-sweep. We do this separately for each triangle of P that is intersected by an edge of either of the two regions that are to be intersected. The total number of intersection computations for a single site $r \in R_i$ is $|B_l| - 1$, which is bounded by a constant. Since B_l has only a constant number of sites, r has a constant number of neighbors in $\mathrm{FVD}(\{r\} \cup B')$ for any $B' \subseteq B_l$, and the complexity of the region of r in any of these diagrams is $O(n^2)$. This means that the intersections can be computed in $O(n^2 \log n)$ time for a single red site $r \in R_i$. The time taken for all the sites in R_i is $O(|R_i|n^2 \log n)$.

- Next, we compute the region of each site $r \in R_i$ in $\mathrm{FVD}(R_i \cup B_l)$ by intersecting its regions in $\mathrm{FVD}(\{r\} \cup B_l)$ and $\mathrm{FVD}(R_i)$ with a line-sweep in a similar way as in the previous step. Since the complexity of the region of r in $\mathrm{FVD}(R_i \cup B_l)$ is $O(n^2)$ and the complexity of the region of r in $\mathrm{FVD}(R_i)$ is $O(N_r \cdot n^2)$, where N_r is the number of neighbors of r in $\mathrm{FVD}(R_i)$, the time needed to compute the region of a single site r in $\mathrm{FVD}(R_i \cup B_l)$ is $O(n^2 \log n \cdot N_r \log N_r)$. Summing this over all $r \in R_i$ gives

$$O(n^2 \log n \sum_{r \in R_i} N_r log N_r)$$

  We have that $N_r \leq m$ for all $r \in R_i$, and $\sum_{r \in R_i} N_r = |R_i|$. The time needed to compute the region of each site $r \in R_i$ in $\mathrm{FVD}(R_i \cup B_l)$ is therefore bounded by $O(|R_i|n^2 \log m \log n)$.

- To complete the computation of $\mathrm{FVD}(R_i \cup B_l)$, it remains to compute the regions of the blue sites. We have computed the regions of all red sites, and carefully

maintained topological relations between each region of a red site and its neighbors. Using the topological information, we do a depth-first traversal of the regions in $\mathrm{FVD}(R_i \cup B_l)$ (that is partially constructed), starting in the region of a red site. When a region of a blue site is visited for the first time, it has to be constructed. At least one of its edges has already been computed, namely, the edge incident to its predecessor in the traversal. Zero or more of the other edges of the region are incident to red regions; these have also been computed already. We still have to trace these edges, namely to find the starting points for the edges that have not been computed yet. These are *blue edges*, the edges between two blue regions. All blue edges are sub-edges of the edges of $\mathrm{FVD}(B_l)$, of which there are only a constant number. The endpoints of these edges are either vertices of $\mathrm{FVD}(B_l)$, or they are vertices of a red region. The latter vertices can be determined in $O(|R_i|n^2)$ time by traversing the edges of all regions of red sites, and reporting the vertices that are incident to blue edges. The total number of endpoints of blue edges is bounded by $O(|B_j|)$, which is a constant, so we can exploit the techniques described in Section 5.4.1 to trace the blue edges in $O(n^2 \log n)$ time per edge. The total time for computing the regions of the blue sites is $O(|R_i|n^2 \log n)$.

Putting everything together, computing $\mathrm{FVD}(R_i \cup B_l)$ takes $O(|R_i|n^2 \log m \log n)$ time, and computing $\mathrm{FVD}(R_k \cup B_j)$ takes $O(|B_j|n^2 \log m \log n)$ time. Since both $\sum_{i=0}^{k} |R_i|$ and $\sum_{i=0}^{l} |B_j|$ are bounded by $O(m)$, the time needed for computing all the diagrams in the basic merge step is $O(mn^2 \log m \log n)$. The amount of memory needed in the basic merge step is linear in the complexity of all bisectors and diagrams that we computed, which is $O(mn^2)$.

5.4.4 The generic merge step

The generic merge step is the computation of $\mathrm{FVD}(R_i \cup B_j)$ from $\mathrm{FVD}(R_i \cup B_{j+1})$ and $\mathrm{FVD}(R_{i+1} \cup B_j)$, which eventually gives the required $\mathrm{FVD}(R_0 \cup B_0) = \mathrm{FVD}(S)$. First some terminology: we call the sites in R_{i+1} the *old* red sites, and the sites in $R_i \setminus R_{i+1}$ the *new* red sites. Similarly, the sites in B_{j+1} are the *old* blue sites, and the sites in $B_j \setminus B_{j+1}$ are the *new* blue sites. Now consider any vertex v of $\mathrm{FVD}(R_i \cup B_j)$. The important fact is that not all three Voronoi regions incident to that vertex correspond to new sites; there must be at least one old red or blue site whose face is incident to v, because new red (blue) regions form an independent set in $\mathrm{FVD}(R_i)$ (resp. $\mathrm{FVD}(B_j)$). So to determine all the vertices of $\mathrm{FVD}(R_i \cup B_j)$, it suffices to compute the regions in $\mathrm{FVD}(R_i \cup B_j)$ of all old red and blue sites.

Consider an old red site r. The region of r in $\mathrm{FVD}(R_i \cup B_{j+1})$ contains all points that are further from r than from any other site in $R_i \cup B_{j+1}$, and the region of r in $\mathrm{FVD}(R_{i+1} \cup B_j)$ contains all points that are further from r than from any other site in $R_{i+1} \cup B_j$. The region of r in $\mathrm{FVD}(R_i \cup B_j)$ is therefore the intersection of its regions in $\mathrm{FVD}(R_i \cup B_{j+1})$ and $\mathrm{FVD}(R_{i+1} \cup B_j)$. We can compute this intersection for each face of the polyhedron separately by a line-sweep of the regions of r in $\mathrm{FVD}(R_i \cup B_{j+1})$ and $\mathrm{FVD}(R_{i+1} \cup B_j)$. The time needed for computing the vertices of $\mathrm{FVD}(R_i \cup B_j)$ is therefore bounded by

$O(C\log C)$, where $C = \max(n^2|R_i \cup B_{j+1}|, n^2|R_{i+1} \cup B_j|, n^2|R_i \cup B_j|)$, which in turn is at most $n^2(|R_i| + |B_j|)$. Hence, computing the vertices of $\mathrm{FVD}(R_i \cup B_j)$ takes $O(n^2(|R_i| + |B_j|)\log(n^2(|R_i| + |B_j|))) = O(n^2(|R_i| + |B_j|)\log n)$ (recall that $m < n$).

The edges of $\mathrm{FVD}(R_i \cup B_j)$ are either edges incident to the faces of old red or blue sites (which we already computed), or edges between the faces of two new sites of the same color (these edges are sub-edges of edges in $\mathrm{FVD}(R_i)$ or $\mathrm{FVD}(B_j)$, and can easily be traced), or they are edges between the faces of a new red and a new blue site. For the latter category of edges we already have the incident vertices computed, and we can trace the edges after computing the bisector of the new red and new blue site. The total number of bisectors we have to compute and trace is bounded by $|R_i \cup B_j|$, so this takes $O(n^2 \log n(|R_i| + |B_j|))$ time. We conclude that computing $\mathrm{FVD}(R_i \cup B_j)$ from $\mathrm{FVD}(R_i \cup B_{j+1})$ and $\mathrm{FVD}(R_{i+1} \cup B_j)$ takes $O(n^2 \log n)(|R_i| + |B_j|))$ time.

Summing this over all $0 \le i \le k$, $0 \le j \le l$ gives time

$$O(n^2 \log n \sum_{i=0}^{k} \sum_{j=0}^{l} (|R_i| + |B_j|))$$

We have

$$\sum_{i=0}^{k} \sum_{j=0}^{l} |B_j| = O(\sum_{i=0}^{k} |B_0|) = O(k|B_0|) = O(m \log m),$$

and similarly $\sum_{i=0}^{k} \sum_{j=0}^{l} (|R_i|) = O(m \log m)$. It follows that the total time spent in all the iterations of the generic merge step is $O(mn^2 \log m \log n)$.

5.4.5 Total running time and memory requirements

The time for merging $\mathrm{FVD}(R)$ and $\mathrm{FVD}(B)$ into $\mathrm{FVD}(R \cup B)$ is dominated by the generic merge step, which requires $O(mn^2 \log m \log n)$ time; the total running time satisfies the recurrence

$$\begin{aligned} T(1) &= O(1) \\ T(2) &= O(n^2 \log n) \\ T(m) &= T(\lfloor m/2 \rfloor) + T(\lceil m/2 \rceil) + O(mn^2 \log m \log n) \end{aligned}$$

which solves to $T(m) = O(mn^2 \log^2 m \log n)$.

The memory requirements of the algorithm are linear in the size of all diagrams that are constructed in the process, which is $O(mn^2 \log m)$.

Theorem 8 *The complexity of the furthest-site Voronoi diagram of m sites on the surface of a polyhedron with n triangles has complexity $\Theta(mn^2)$. The diagram can be computed in $O(mn^2 \log^2 m \log n)$ time, using $O(mn^2 \log m)$ memory.*

5.5 Conclusions and further research

We have shown that the furthest-site Voronoi diagram of a set S of m sites on the surface of a polyhedron P with n triangles has complexity $\Theta(mn^2)$, and we have given an algorithm for computing the diagram in $O(mn^2 \log^2 m \log n)$ time. Once the diagram has been computed, the facility center, which is the point on P that minimizes the maximum distance to a site in S, can be found in $O(mn^2)$ time by traversing the edges of the diagram.

The merge step in our divide-and-conquer approach for the computation of $\mathrm{FVD}(S)$ is quite complicated, and it would be pleasant to find a simpler method. Merging the recursively computed diagrams by sweeping seems natural, but the number of intersections of edges of both diagrams can be superlinear (in m), while only a linear number of them can end up as a vertex of the resulting diagram.

It would be a challenge to find an output-sensitive algorithm, i.e., an algorithm that takes time proportional to the number edges/vertices in the diagram plus the number of their intersections with the edges of P. Even more ambitious would be the computation of the diagram without explicitly representing all intersections of furthest-site Voronoi edges and edges of the polyhedron.

Another interesting issue is approximation: find (in $o(mn^2)$ time) a point with the property that the distance to the furthest site is at most $(1+\varepsilon)$ times the radius of the smallest enclosing circle.

Finally, it is worth investigating whether the facility location problem can be solved without constructing the furthest-site Voronoi diagram. Recall that the facility location problem in the plane can be solved using techniques related to fixed-dimensional linear programming [74, 125].

Bibliography

[1] ARC/INFO: an example of contemporary geographic information system. In D. J. Peuquet and D. F. Marble, editors, *Introductory Readings in Geographic Information Systems*, pages 90–99. Taylor & Francis, London, 1990.

[2] Technical description of the DIME system. In D. J. Peuquet and D. F. Marble, editors, *Introductory Readings in Geographic Information Systems*, pages 100–111. Taylor & Francis, London, 1990.

[3] P. K. Agarwal, M. de Berg, J. Matoušek, and O. Schwarzkopf. Constructing levels in arrangements and higher order Voronoi diagrams. *SIAM J. Comput.*, 27:654–667, 1998.

[4] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[5] B. Aronov and J. O'Rourke. Nonoverlap of the star unfolding. *Discrete Comput. Geom.*, 8:219–250, 1992.

[6] B. Aronov, M. van Kreveld, R. van Oostrum, and K. Varadarajan. Facility location on terrains. In *Algorithms and Computation, Proc. 9th Int. Symp. (ISAAC'98)*, volume 1533 of *Lecture Notes Comput. Sci.*, pages 19–28. Springer-Verlag, 1998.

[7] E. Artzy, G. Frieder, and G. T. Herman. The theory, design, implementation, and evaluation of 3-d surface detection algorithms. *Comput. Graph. Image Process.*, 15:1–24, 1981.

[8] T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmayer. Space-filling curves and their use in the design of geometric data structures. *Theoret. Comput. Sci.*, 181(1):3–15, July 1997.

[9] F. Aurenhammer. Voronoi diagrams: A survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405, Sept. 1991.

[10] F. Aurenhammer and H. Edelsbrunner. An optimal algorithm for constructing the weighted Voronoi diagram in the plane. *Pattern Recogn.*, 17:251–257, 1984.

[11] F. Aurenhammer and O. Schwarzkopf. A simple on-line randomized incremental algorithm for computing higher order Voronoi diagrams. *Internat. J. Comput. Geom. Appl.*, 2:363–381, 1992.

[12] D. Avis and K. Fukuda. A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra. *Discrete Comput. Geom.*, 8:295–313, 1992.

[13] D. Avis and K. Fukuda. Reverse search for enumeration. *Discrete Appl. Math.*, 65:21–46, 1996.

[14] C. L. Bajaj, V. Pascucci, and D. R. Schikore. Fast isocontouring for improved interactivity. In *Proc. 1996 ACM/IEEE Symposium on Volume Visualization*, pages 39–46, Oct. 1996.

[15] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 322–331, 1990.

[16] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28(9):643–647, Sept. 1979.

[17] K. Q. Brown. Comments on "Algorithms for reporting and counting geometric intersections". *IEEE Trans. Comput.*, C-30:147–148, 1981.

[18] P. A. Burrough. *Principles of Geographical Information Systems for Land Resourses Assessment*. Oxford University Press, New York, 1986.

[19] B. P. Buttenfield and R. B. McMaster, editors. *Map Generalization: Making Rules for Knowledge Representation*. Longman, Harlow, 1991.

[20] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(3):133–162, 1986.

[21] B. Chazelle and L. J. Guibas. Fractional cascading: II. Applications. *Algorithmica*, 1:163–191, 1986.

[22] J. Chen and Y. Han. Shortest paths on a polyhedron. *Internat. J. Comput. Geom. Appl.*, 6:127–144, 1996.

[23] J. Christensen, J. Marks, and S. Shieber. An empirical study of algorithms for point-feature label placement. *ACM Trans. Graph.*, 14:202–232, 1995.

[24] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno. Optimal isosurface extraction from irregular volume data. In *Proc. 1996 ACM/IEEE Symposium on Volume Visualization*, pages 31–38, 1996.

[25] K. C. Clarke. *Analytical and Computer Cartography*. Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 1995.

[26] R. Cole. Searching and storing similar lists. *J. Algorithms*, 7:202–220, 1986.

[27] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[28] R. G. Cromley. *Digital Cartography*. Prentice Hall, Englewood Cliffs, 1992.

[29] J. Dangermond. A review of digital data commonly available and some of the practical problems of entering them into a GIS. In D. J. Peuquet and D. F. Marble, editors, *Introductory Readings in Geographic Information Systems*, pages 222–232. Tayler & Francis, London, 1990.

[30] M. de Berg and M. van Kreveld. Trekking in the alps without freezing or getting tired. *Algorithmica*, 18:306–323, 1997.

[31] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[32] M. de Berg, M. van Kreveld, R. van Oostrum, and M. Overmars. Simple traversal of a subdivision without extra storage. *International Journal on Geographical Information Science*, 11:359–373, 1997.

[33] L. De Floriani, B. Falcidieno, G. Nagy, and C. Pienovi. On sorting triangles in a Delaunay tessellation. *Algorithmica*, 6:522–532, 1991.

[34] B. D. Dent. *Cartography – Thematic Map Design*. WCB/McGraw-Hill, Boston, fifth edition, 1999.

[35] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[36] G. L. Dirichlet. Über die reduktion der positiven quadratischen formen mit drei unbestimmten ganzen zahlen. *J. Reine Angew. Math.*, 40:209–227, 1850.

[37] D. P. Dobkin and D. G. Kirkpatrick. A linear algorithm for determining the separation of convex polyhedra. *J. Algorithms*, 6:381–392, 1985.

[38] D. P. Dobkin and M. J. Laszlo. Primitives for the manipulation of three-dimensional subdivisions. *Algorithmica*, 4:3–32, 1989.

[39] J. S. Doerschler and H. Freeman. A rule-based system for dense-map name placement. *Commun. ACM*, 35:68–79, 1992.

[40] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Comput.*, 15(2):317–340, 1986.

[41] P.-O. Fjällström. Polyhedral approximation of bivariate functions. In *Proc. 3rd Canad. Conf. Comput. Geom.*, pages 187–190, Aug. 1991.

[42] D. M. Flewelling and M. Egenhofer. Formalizing importance: parameters for settlement selection in a geographic database. In *Proc. Auto-Carto 11*, pages 167–175, 1993.

[43] S. J. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[44] H. Freeman and S. P. Morse. On searching a contour map for a given terrain profile. *J. of the Franklin Institute*, 248:1–25, 1967.

[45] K. Fukuda and V. Rosta. Combinatorial face enumeration in convex polytopes. *Comput. Geom. Theory Appl.*, 4:191–198, 1994.

[46] C. Gold and S. Cormack. Spatially ordered networks and topographic reconstructions. In *Proc. 2nd Internat. Sympos. Spatial Data Handling*, pages 74–85, 1986.

[47] 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. *Comput. Graph.*, 11(2):170–175, 1977.

[48] C. M. Gold and U. Maydell. Triangulation and spatial ordering in computer cartography. In *Proc. Canad. Cardographic Association Annu. Meeting*, pages 69–81, 1978.

[49] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7:381–413, 1992.

[50] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graph.*, 4(2):74–123, Apr. 1985.

[51] L. J. Guibas and J. Stolfi. Ruler, compass and computer: the design and analysis of geometric algorithms. In R. A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, volume 40 of *NATO ASI Series F*, pages 111–165. Springer-Verlag, 1988.

[52] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Conf. Principles Database Systems*, pages 47–57, 1984.

[53] P. S. Heckbert and M. Garland. Fast polygonal approximation of terrains and height fields. Report CMU-CS-95-181, Carnegie Mellon University, 1995.

[54] M. Heller. Triangulation algorithms for adaptive terrain modeling. In *Proc. 4th Internat. Sympos. Spatial Data Handling*, pages 163–174, 1990.

[55] I. Heywood, S. Cornelius, and S. Carver. *An Introduction to Geographical Information Systems*. Addison Wesley Longman, Harlow, 1998.

[56] M. W. Hirsch. *Differential Topology*. Springer-Verlag, New York, NY, 1976.

[57] C. T. Howie and E. H. Blake. The mesh propagation algorithm for isosurface construction. *Comput. Graph. Forum*, 13:65–74, 1994.

[58] E. H. Isaaks and R. M. Srivastava. *An Introduction to Applied Geostatistics*. Oxford University Press, New York, 1989.

[59] T. Itoh and K. Koyamada. Automatic isosurface propagation using an extrema graph and sorted boundary cell lists. *IEEE Trans. Visualizat. Comput. Graph.*, 1:319–327, 1995.

[60] R. J. Johnston, D. Gregory, and D. M. Smith, editors. *The Dictionary of Human Geography*. Blackwell, Oxford, 2nd edition, 1986.

[61] N. Kadmon. Automated selection of settlements in map generation. *The Cartographic Journal*, 9:93–98, 1972.

[62] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12(1):28–35, 1983.

[63] M.-J. Kraak and F. J. Ormeling. *Cartography – visualization of spatial data*. Longman, Harlow, 1996.

[64] I. S. Kweon and T. Kanade. Extracting topographic terrain features from elevation maps. *CVGIP: Image Understanding*, 59:171–182, 1994.

[65] G. Langran. *Time in Geographic Information Systems*. Taylor & Francis, London, 1992.

[66] G. E. Langran and T. K. Poiker. Integration of name selection and name placement. In *Proc. 2nd Int. Symp. on Spatial Data Handling*, pages 50–64, 1986.

[67] R. Laurini and D. Thompson. *Fundamentals of Spatial Information Systems*. Academic Press, Boston, MA, 1992.

[68] D. T. Lee and F. P. Preparata. Location of a point in a planar subdivision and its applications. *SIAM J. Comput.*, 6(3):594–606, 1977.

[69] D. Leven and M. Sharir. Intersection and proximity problems and Voronoi diagrams. In J. T. Schwartz and C.-K. Yap, editors, *Advances in Robotics 1: Algorithmic and Geometric Aspects of Robotics*, pages 187–228. Lawrence Erlbaum Associates, Hillsdale, NJ, 1987.

[70] Y. Livnat, H.-W. Shen, and C. R. Johnson. A near optimal isosurface extraction algorithm using the span space. *IEEE Trans. Visualizat. Comput. Graph.*, 2:73–84, 1996.

[71] W. A. Mackaness and M. K. Beard. Use of graph theory to support map generalization. *Cartography and Geographic Information Systems*, 20(4):210–221, 1993.

[72] D. M. Mark. Competition for map space as a paradigm for automated map design. In *Proc. GIS/LIS*, pages 97–106, 1990.

[73] H. Matthews and I. Foster. *Geographical Data – Sources, Presentation, and Analysis*. Oxford University Press, London, 1989.

[74] N. Megiddo. Linear-time algorithms for linear programming in R^3 and related problems. *SIAM J. Comput.*, 12:759–776, 1983.

[75] K. Mehlhorn and S. Näher. Dynamic fractional cascading. *Algorithmica*, 5:215–241, 1990.

[76] J. W. Milnor. *Morse Theory*. Princeton University Press, Princeton, NJ, 1963.

[77] J. S. B. Mitchell, D. M. Mount, and C. H. Papadimitriou. The discrete geodesic problem. *SIAM J. Comput.*, 16:647–668, 1987.

[78] D. M. Mount. Voronoi diagrams on the surface of a polyhedron. Technical Report 1496, Department of Computer Science, University of Maryland, 1985.

[79] D. E. Muller and F. P. Preparata. Finding the intersection of two convex polyhedra. *Theoret. Comput. Sci.*, 7:217–236, 1978.

[80] K. Mulmuley. A fast planar partition algorithm, I. *J. Symbolic Comput.*, 10(3-4):253–280, 1990.

[81] W. Norton. *Human Geography*. Oxford University Press, 1992.

[82] A. Okabe, B. Boots, and K. Sugihara. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. John Wiley & Sons, Chichester, UK, 1992.

[83] J. Pach and M. Sharir. On vertical visibility in arrangements of segments and the queue size in the Bentley-Ottmann line sweeping algorithm. *SIAM J. Comput.*, 20:460–470, 1991.

[84] J. Peschier. Computer aided generalization of road network maps. Master's thesis, Dept. of Computer Science, Utrecht University, 1997. INF/SCR-97-15.

[85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.

[86] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 3rd edition, Oct. 1990.

[87] E. Ramos. Intersection of unit-balls and diameter of a point set in R^3. *Computat. Geom. Theory Appl.*, 8:57–65, 1997.

[88] J. F. Raper, editor. *Three Dimensional Applications in Geographical Information Systems*. Taylor & Francis, London, 1989.

[89] G. Reeb. Sur les points singuliers d'une forme de pfaff complement integrable ou d'une fonction numerique. *Comptes Rendus Acad. Sciences Paris*, 222:847–849, 1946.

[90] A. A. Robinson, J. L. Morrison, P. C. Muehrcke, A. J. Kimerling, and S. C. Guptill. *Elements of Cartography*. John Wiley and Sons, New York, sixth edition, 1995.

[91] A. Rogers, H. Viles, and A. Goudie, editors. *The Student's Companion to Geography*. Blackwell, Oxford, 1992.

[92] H. Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2), June 1984.

[93] H. Samet. An overview of quadtrees, octrees, and related hierarchical data structures. In R. A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, volume 40 of *NATO ASI Series F*, pages 51–68. Springer-Verlag, Berlin, West Germany, 1988.

[94] H. Samet. *Spatial Data Structures: Quadtrees, Octrees, and Other Hierarchical Methods*. Addison-Wesley, Reading, MA, 1989.

[95] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, MA, 1990.

[96] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[97] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, July 1986.

[98] R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Comput. Geom. Theory Appl.*, 1(1):51–64, 1991.

[99] M. I. Shamos and D. Hoey. Geometric intersection problems. In *Proc. 17th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 208–215, 1976.

[100] M. Sharir and A. Schorr. On shortest paths in polyhedral spaces. *SIAM J. Comput.*, 15:193–215, 1986.

[101] Y. Shinagawa and T. L. Kunii. Constructing a Reeb graph automatically from cross sections. *IEEE Comput. Graph. Appl.*, 11:44–51, Nov. 1991.

[102] Y. Shinagawa, T. L. Kunii, and Y. L. Kergosien. Surface coding based on morse theory. *IEEE Comput. Graph. Appl.*, 11:66–78, Sept. 1991.

[103] R. Sibson. Locally equiangular triangulations. *Comput. J.*, 21(3):243–245, 1978.

[104] J. K. Sircar and J. A. Cerbrian. Application of image processing techniques to the automated labelling of raster digitized contours. In *Proc. 2nd Internat. Sympos. Spatial Data Handling*, pages 171–184, 1986.

[105] J. Snoeyink, 1995. Personal Communication.

[106] J. Snoeyink and M. van Kreveld. Linear-time reconstruction of Delaunay triangulations with applications. In *Algorithms – ESA'97, Proc. 5th Ann. European Symp.*, volume 1284 of *Lecture Notes Comput. Sci.*, pages 459–471. Springer-Verlag, 1997.

[107] J. Star and J. Estes. *Geographic Information Systems: an Introduction*. Prentice Hall, Englewood Cliffs, 1990.

[108] M. Stonebraker, T. Sellis, and E. Hanson. An analysis of rule indexing implementations in data base systems. In L. Kerschberg, editor, *Expert Database Systems (EDS), Proc. 1st Int. Conf*, pages 465–476. Benjamin/Cummings, Menlo Park, California, 1986.

[109] A. H. Strahler and A. N. Strahler. *Physical Geography, Science and Systems of the Human Environment*. Wiley, New York, 1997.

[110] S. Takahashi, T. Ikeda, Y. Shinagawa, T. L. Kunii, and M. Ueda. Algorithms for extracting correct critical points and constructing topological graphs from discrete geographical elevation data. In *Eurographics'95*, volume 14, pages C–181–C–192, 1995.

[111] S. P. Tarasov and M. N. Vyalyi. Contour tree construction in $o(n \log n)$ time. In *Proc. 14th Annu. ACM Symp. Comp. Geometry*, pages 68–75, 1998.

[112] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22:215–225, 1975.

[113] R. C. Thomson and D. E. Richardson. A graph theory approach to road network generalization. In *Proc. 17th Int. Cartographic Conference*, pages 1871–1880, Barcelona, 1995.

[114] F. T. Töpfer and W. Pillewizer. The principles of selection. *The Cartographic Journal*, 3:10–16, 1966.

[115] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Syst. Theory*, 10:99–127, 1977.

[116] M. van Kreveld. Efficient methods for isoline extraction from a TIN. *Internat. J. of GIS*, 10:523–540, 1996.

[117] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. Schikore. Contour trees and small seed sets for isosurface traversal. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 212–220, 1997.

[118] M. van Kreveld, R. van Oostrum, and J. Snoeyink. Efficient settlement selection for interactive display. In *Proc. Auto-Carto 13: ACSM/ASPRS Annual Convention Technical Papers*, pages 287–296, 1997.

[119] P. van Oosterom. A reactive data structure for geographic information systems. In *Proc. Auto-Carto 9*, pages 665–674, 1989.

[120] J. van Putten. Experiences with the gap-tree. Master's thesis, Dept. of Computer Science, Utrecht University, 1997. INF/SCR-97-30.

[121] M. J. van Trigt. Proximity problems on polyhedral terrains. Master's thesis, Dept. of Computer Science, Utrecht University, 1995. INF/SCR-95-18.

[122] G. M. Voronoi. Nouvelles applications des paramètres continus à la théorie des formes quadratiques. premier Mémoire: Sur quelques propriétés des formes quadratiques positives parfaites. *J. Reine Angew. Math.*, 133:97–178, 1907.

[123] G. M. Voronoi. Nouvelles applications des paramètres continus à la théorie des formes quadratiques. deuxième Mémoire: Recherches sur les paralléllоèdres primitifs. *J. Reine Angew. Math.*, 134:198–287, 1908.

[124] D. F. Watson. *Contouring: A Guide to the Analysis and Display of Spatial Data*. Pergamon, 1992.

[125] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer, editor, *New Results and New Trends in Computer Science*, volume 555 of *Lecture Notes Comput. Sci.*, pages 359–370. Springer-Verlag, 1991.

[126] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. *ACM Trans. Graph.*, 11:201–227, 1992.

[127] M. F. Worboys. *GIS: A Computing Perspective*. Taylor & Francis, London, 1995.

Acknowledgements

Many people contributed to this thesis in one way or another, and I wish to express my gratitude towards them.

First of all, I would like to thank my promotor, Mark Overmars, and my co-promotor, Marc van Kreveld. The way they supervised has been a perfect fusion of giving me the freedom to do what I wanted to do, stimulating and helping me, and steering me in the right direction. On occasions, other Ph.D. students referred to me as “the one that gets actually supervised”, and I think this says a lot. During our regular meetings, Mark Overmars’ bright observations always helped me to see “the big picture”, and the many research sessions with Marc van Kreveld have been the icing on the cake of my Ph.D.

I should also thank the co-authors of the papers of which this thesis is assembled: apart from Marc van Kreveld and Mark Overmars, these are Boris Aronov, Chandrajit Bajaj, Mark de Berg, Valerio Pascucci, Dan Schikore, and Jack Snoeyink.

I would also like to thank Peter van Emde Boas, Joost Kok, Jan van Leeuwen, Jack Snoeyink, and Peter Widmayer, for taking place in the reviewing committee. Jan van Leeuwen and Jack Snoeyink provided many corrections and suggestions for improvement of the manuscript, for which I’m grateful.

I want to thank Otfried Cheong for a lot of reasons. First of all, he supervised me during the writing of my M.Sc. thesis, and it was during that time that I became convinced that I wanted to do a Ph.D. in Computational Geometry. A few years later, he invited me to come to South Korea and Hong Kong to do research, and I had a great time on both occasions. Finally, I’d like to thank him for introducing me to Kim-Chi (although it was actually Mark de Berg who wetted my appetite with his horror stories about these “disgusting rotten vegetables”).

I won’t list the many friends and beloved ones that indirectly contributed to this thesis by making my life more pleasurable. The list would be too long . . . Thank you all!

Samenvatting

Een *Geografisch Informatiesysteem*, of GIS, is een systeem van hard- en software voor het opslaan, verwerken en analyseren van ruimtelijke gegevens. De hoeveelheid opgeslagen informatie is dikwijls groot, en de aard van de informatie zeer divers. Een GIS wordt onder andere gebruikt door de overheid, bijvoorbeeld voor het plannen van nieuwe infrastructuur zoals de Betuwelijn. Om het meest gunstige traject te bepalen moet er informatie voorhanden zijn over de verschillende bodemsoorten in het gebied waarbinnen het traject moet komen te liggen; verschillende bodemsoorten brengen verschillende aanlegkosten met zich mee. Ook zal er bij de planning rekening gehouden moeten worden met geluidsoverlast in bewoonde gebieden, met schade aan de natuur, met de kosten van schadeloosstelling bij onteigening van grond en onroerende goederen, en met vele andere criteria. Waar het handmatig doorrekenen van alle alternatieven een ondoenlijke zaak is, kan men met behulp van een GIS relatief eenvoudig cijfermateriaal, kaartjes en tabellen genereren die de kosten en gevolgen van de alternatieven overzichtelijk maken. Andere gebieden waarin een GIS wordt toegepast zijn bijvoorbeeld geologie, meteorologie en cartografie.

Veel basisvraagstukken die een GIS moet oplossen zijn meetkundig van aard, zoals het rekenen aan afstanden. Het alledaagse afstandsbegrip “de kortste weg tussen twee punten is een recht lijnstuk” is niet in alle situaties bruikbaar. Bij routeplanning over het wegennetwerk is er zelden tot nooit een rechtlijnige verbinding tussen het vertrekpunt en de bestemming, maar moeten er verschillende mogelijke routes worden doorgerekend. Computers zijn daar goed in, getuige de vele routeplannings-programma’s die voor personal computers verkrijgbaar zijn. Wanneer we de wegen mogen verlaten en ons ook door weiland, bos of zandvlakten gaan verplaatsen wordt het berekenen van de kortste weg van A naar B (waarbij afstand wordt uitgedrukt in de tijd die het kost om van A naar B te komen) ineens een stuk lastiger. Terwijl we op een kruising van wegen slechts een handvol keuzen hebben, is er op de grens van twee bodemsoorten van verschillende begaanbaarheid een oneindig aantal richtingen waarin we verder kunnen trekken. Ook in bergachtig gebied is het rekenen aan afstanden vaak lastig, zelfs wanneer we de zaak vereenvoudigen door te stellen dat we ons overal met dezelfde snelheid kunnen voortbewegen, ongeacht bodemsoort of de helling van het terrein. Wanneer we dergelijke vereenvoudigende aannamen maken in vlak terrein komen we weer uit bij het rechte lijnstuk als kortste weg tussen begin- en eindpunt. In de bergen echter hebben we onder andere de keuze om over

de berg te gaan, of eromheen, zowel linksom als rechtsom. Afhankelijk van het aantal pieken en dalen in het gebied kan het aantal mogelijke routes fors toenemen.

Binnen het onderzoeksgebied van de *Geometrische Algoritmen* houdt men zich bezig met onder andere dit soort vraagstukken. In plaats van oplossingen te forceren door middel van brute kracht (lees: grote, snelle computers), probeert men slimme rekenmethoden (algoritmen) te ontwerpen. Zo is het vaak mogelijk om, in plaats van alle mogelijke oplossingen na te gaan, al vroegtijdig een ruwe selectie te maken tussen kansloze en kanshebbende oplossingen. Door de selectie telkens te verfijnen en alleen verder te rekenen met de kanshebbende oplossingen, komt men uiteindelijk uit bij "het goede antwoord". Het gebied van de Geometrische Algoritmen heeft vele praktische toepassingen, onder andere in de Robotica, Computer Graphics, CAD/CAM, en GIS.

Dit proefschrift is een bundeling van een viertal artikelen, in elk waarvan een specifiek GIS-vraagstuk wordt opgelost met behulp van bestaande en nieuwe technieken uit de Geometrische Algoritmen.

Het eerste hoofdstuk is een inleiding op de vier artikelen, en voorziet de lezer van de achtergrondinformatie, globale kennis en literatuurverwijzingingen die nodig zijn om de vier volgende hoofdstukken te kunnen waarderen.

Het tweede hoofdstuk geeft een nieuwe methode om een zogenaamde *subdivisie* af te lopen. Men kan hierbij bijvoorbeeld denken aan bepaalde typen landkaarten, waarbij een gebied is onderverdeeld in vlakjes, en waarbij bij ieder vlakje extra informatie is opgeslagen, bijvoorbeeld het aantal olifanten per vierkante kilometer. Een GIS kan verschillende bewerkingen op een dergelijke kaart uitvoeren, zoals het afbeelden van (een gedeelte van) de kaart, of het berekenen van het totale aantal olifanten in een gedeelte van het beschreven gebied. Bij dergelijke bewerkingen moeten de vlakjes van de kaart afgelopen worden, om (bij het laatste voorbeeld) de gegevens over de olifantenpopulatie te verzamelen. Daarbij is het van belang dat elk relevant vlakje precies éénmaal wordt behandeld; het overslaan of het meerdere malen behandelen van vlakjes zou resulteren in foute uitkomsten. Een veelgebruikte oplossing hiervoor is om expliciet bij te houden welke vlakjes al behandeld zijn. Dit heeft echter het nadeel dat het computergeheugen kost en, afhankelijk van de methode van bijhouden, verhindert dat meerdere gebruikers gelijktijdig met dezelfde gegevens kunnen werken. De methode die we in hoofdstuk twee presenteren heeft die nadelen niet. Hoewel we niet expliciet bijhouden welke vlakjes al behandeld zijn, kunnen we toch garanderen dat onder alle omstandigheden elk vlakje precies éénmaal behandeld wordt.

Het derde hoofdstuk is gericht op het efficiënt visualiseren van twee- en drie-dimensionale gegevens met behulp van zogenaamde *iso-contouren*. In het twee-dimensionale geval kan men denken aan de welbekende contourlijnen op landkaarten, die naburige locaties van gelijke hoogte met elkaar verbinden. Dicht op elkaar liggende contourlijnen geven een steile helling aan, terwijl in relatief vlak gebied de contourlijnen ver uit elkaar liggen. Voor het drie-dimensionale geval kan men denken aan medische beelden verkregen uit CT-scans. De iso-contouren zijn hier aaneengesloten twee-dimensionale oppervlakten in

een drie-dimensionale ruimte. Om de iso-contouren van een bepaalde waarde (hoogtewaarde, of een waarde die een bepaald type weefsel aangeeft) af te beelden, zou men het hele twee- of drie-dimensionale plaatje kunnen aflopen, om alleen die locaties te rapporteren die de gevraagde waarde hebben. Dat is echter niet efficiënt, omdat de gezochte iso-contouren doorgaans slechts een klein deel van het plaatje omvatten. Efficiënter is het om beginpunten van alle mogelijke iso-contouren op te slaan, in die verzameling beginpunten te zoeken naar de gevraagde waarden, en dan de contouren in het plaatje af te lopen. Die beginpunten worden *seeds* genoemd. In een ideale situatie is het aantal seeds zo klein mogelijk, maar wel zodanig dat alle mogelijke contouren gevonden kunnen worden. In dit hoofdstuk geven we een nieuwe methode voor het bepalen van een zo klein mogelijke verzameling seeds; de structuur die we hiervoor gebruiken is de *contour tree*, en de methode voor het bouwen van deze structuur is eveneens nieuw. De looptijd van de methode voor het bepalen van de kleinste verzameling seeds is echter in de praktijk niet efficiënt genoeg. Daarom geven we daarnaast een tweede, efficiëntere methode, waarbij de opgeleverde verzameling seeds niet de kleinst mogelijke is, maar wel klein genoeg. Deze tweede methode maakt eveneens gebruik van de contour-tree.

Het vierde hoofdstuk houdt zich bezig met het afbeelden van steden op kaarten (al dan niet op een computerscherm). Het aantal steden dat kan worden afgebeeld is onder meer afhankelijk van de gekozen schaal van de kaart. Het afbeelden van alle steden die in de GIS zijn opgeslagen op een kleinschalige kaart leidt doorgaans tot een onoverzichtelijke brij van punten. Er zal in dergelijke gevallen een selectie gemaakt moeten worden. Intuïtief zou men misschien denken dat het kiezen van de top-zoveel van grootste steden een logische en goede keuze is. Dat hoeft echter niet zo te zijn: een grote stad kan bij een bepaalde schaal te dicht bij een nog grotere stad liggen, terwijl een klein dorpje dat tamelijk geïsoleerd in landelijk gebied ligt wel probleemloos afgebeeld kan worden. Men zou kunnen zeggen dat het kleine dorpje *relatief belangrijk* is ten opzichte van de grote stad die niet afgebeeld kan worden. In dit hoofdstuk geven we een viertal nieuwe modellen om relatieve belangrijkheid uit te drukken, en we vergelijken onze modellen met drie bestaande modellen. Onze modellen hebben het voordeel dat ze een zogenaamde *ranking* van de steden bepalen. Dat heeft als voordeel dat er slechts eenmaal gerekend hoeft te worden, waarna het afbeelden van de steden eenvoudigweg in volgorde van ranking plaatsvindt totdat het gewenste aantal steden is bereikt. Bij de bestaande modellen moet bij verandering van het gewenste aantal steden telkens opnieuw gerekend worden, waarbij er bovendien vreemde “sprongen” kunnen optreden. Zo is het bij de bestaande methoden niet noodzakelijk zo dat een selectie van bijvoorbeeld vijfentwintig steden alle steden bevat die in een selectie van vierentwintig steden zitten. Het kan bijvoorbeeld gebeuren dat er, wanneer we van vierentwintig naar vijfentwintig steden gaan, drie steden verdwijnen en vier nieuwe steden bijkomen. Bij onze methode treedt dat verschijnsel niet op, omdat er wordt geselecteerd in volgorde van de (van te voren berekende) ranking.

Het vijfde hoofdstuk is gericht op het vinden van een optimale locatie voor een servicecentrum of dienstverlenende instantie, bijvoorbeeld een voedsel-distributiecentrum. We gaan er daarbij vanuit dat het aantal “klanten” van het service-centrum (bijvoorbeeld de supermarkten die bevoorraad worden vanuit het distributiecentrum) vastligt, evenals hun

locatie, en dat één en ander gesitueerd is in bergachtig gebied. De optimale locatie voor het service-centrum is in ons model de locatie waarvandaan de maximale afstand tot de klanten zo klein mogelijk is. Om die locatie te kunnen bepalen maken we gebruik van een structuur die in de Geometrische Algoritmen al langer bekend is, namelijk het zogenaamde *furthest-site Voronoi diagram*. Het nieuwe aan dit hoofdstuk is dat we het eerste algoritme geven dat deze structuur efficiënt berekent voor modellen voor bergachtig gebied.

Curriculum Vitae

René Willibrordus van Oostrum

7 november 1965

Geboren te Utrecht.

augustus 1978 – juni 1984

Gymnasium aan het Bonifatiuscollege te Utrecht.

september 1984 – juni 1989

Hoofdvak orgel aan de Hogeschool voor de Kunsten Utrecht.

september 1989 – januari 1995

Studie Informatica aan de Universiteit Utrecht.

maart 1995 – februari 1999

Assistent in Opleiding aan het Informatica-instituut van de Universiteit Utrecht.

maart 1999 – heden

Postdoc aan de Hong Kong University of Science and Technology.