Topological Data Modelling for Vector Map

Master's Thesis

Hyeyeon Park

30.6.2006

Department of computer science
University of Joensuu

Contents

List of Figures

List of Tables

Abstract

Vector map can be modelled with different data models. First, we build topological information from the map with spaghetti model. We will look through basic vector objects, describe the structure of topology of the map, and then how a non-topological map is transformed to a topological map is explained. Next, bounding containers are explained for more efficient accessing of objects. We do not need whole detailed information of each object in some cases. Different bounding containers are explained, and how minimal bounding rectangle (MBR) is implemented is described. Arc, one of vector objects in the map, can be represented by hierarchical structure. Well known tree structures, strip and arc trees, are reviewed, and smallest bounding area (SBA) tree is proposed. Hierarchical representation can be used in many areas. We can keep the topological information while doing polygonal approximation. Also, hierarchical structure can make windowing, clipping, and point inclusion more efficient. We compare different bounding containers, and different hierarchical structures in experiments.

Chapter 1

Introduction

This chapter describes background and motivation of this work. Also the outline of this thesis is presented.

1.1 Background

Maps have guided people for thousands of years. Traditionally maps were hand-made and for the last century they were printed. These paper maps could not be modified. Then computer revolution came, and these days most of the maps are digitized and stored in digital format, so they can be easily created and processed by a computer. Digital map processing allows many map technologies which are not able in paper maps such as storing a huge set of maps, compress maps using image compression technologies, comfortable interface for browsing maps and so on.

There exists two different formats for presenting digital maps. These are raster maps and vector maps. Raster maps store visual information as a raster image or a set of raster images. Raster image consists of pixels and each pixel can have one of the several colors, depending on the color depth of the image. Typical image formats for this map are PPM, GIF and PNG. Vector maps store visual and geographical information using vector graphics. This map is not a image but a set of graphical entities such as Point, Polyline, Polygon, Arc, Node, and so on. This work concentrates on the latter map format.

Vector maps can have topology which means relationships between entities in maps. Topology describes how map elements are connected each other. If the map has topological information, then each element is aware of its neighbors, therefore, editing or updating maps can be easier.

Data modelling for the vector map can be applied in many different areas. It can help to make the process more efficient and faster. One of most useful geometrical computations is finding intersections between objects. This can be in use for polygonal approximation, windowing, clipping, and etc.

1.2 Motivation

There are many areas in GIS where data modelling for the vector map can make processes more efficient. For example, polygonal approximation is a method for modifying complex vector map so that less important elements are removed. This operation is useful because maps with complex elemental structure are expensive to process and approximated maps can still be used in many applications. However, while map being approximated, errors can occur. This is because topology is not considered. For topologically consistent simplification, hierarchical representation of arcs can be helpful for faster and more efficient processing. For windowing, clipping, and point inclusion, it also can be helpful, because their basic computation is finding intersections between a line segment and an object.

Arc, line segment which has topological information, can be modelled by a hierarchical structure. Tree data structure is commonly used. The widely known techniques are Strip Tree [Bal81], Arc Tree [GW90], and Bezier Tree [Bez74]. In this paper, we first show how vector maps with different data models are embodied and how to build a topological structure, and then look through different kinds of bounding containers. Next, Strip Tree and Arc Tree will be compared with new designed tree. Finally, how these tree structures can be applied in many GIS areas. C++ library which has all functions was built for implementation.

1.3 Outline

The thesis begins with background knowledge about digital images and maps, and then explains the motivation for this study and defines the objective. Next, the structure of vector map is defined and a process of building a topological structure from a non-topological vector map is explained. Chapter 3 is about bounding containers which are used for an approximation of objects, and then two well-known hierarchical structures of arcs, using a minimal bounding rectangle as a bounding container, are explained. A new hierarchical structure is proposed and its performance is compared with others in chapter 4. Several applied areas are looked through and how the usage of a hierarchical structure of arcs affects to the performance is explained in chapter 5. Experiments for comparing performances with and without the hierarchical structure and with different bounding containers, and for comparing performances between different hierarchical structures are described in chapter 6. Finally, conclusions are presented and future works are discussed.

Chapter 2

Vector Data Model

This chapter describes about vector data representation, different vector data models and how they are implemented.

2.1 Vector Data Representation

Vector data represents the real world using discrete points, lines or polygons. In real world, most objects consist of curved lines and areas with soft boundaries. However, those real objects are substituted for discontinuous lines and points, so that their boundaries do not look soft and natural [KO03, HA03]. If the data are represented with smaller and more objects, they look more natural but size will be increased. Figure 2.1 shows how real world can be changed to vector data.

Here are more detailed explanation about typical primary objects which are used in vector data [HCC02, BV02, DeM05].

2.1.1 Point

Point is zero-dimensional abstraction of an object represented by a single set of x and y coordinates. It can be used to depict map features or symbols such as location of buildings on a small-scaled map.

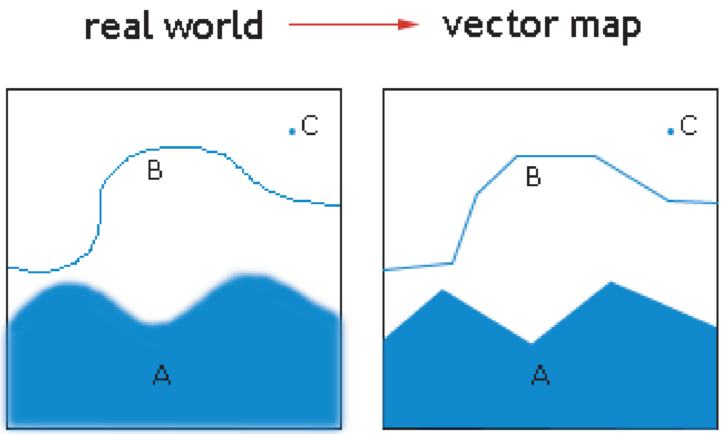


Figure 2.1: Transformation to vector data

2.1.2 Node

Node is same format with Point, but it has additionally topological information. Points where lines from different polygon or polyline intersect are chosen to nodes. Node is also the end point of arc which will be explained later, so it has arc information which has the node. Figure 2.2 shows an example of point and node objects.

2.1.3 Line

Line is a set of x and y coordinates that represent the shape of geographic features such as contours, street centerlines, or streams or linear features with no area such as country boundary lines. It is also called polyline.

2.1.4 Arc

Arc is same format with Line which starts and ends with nodes and has adjacent polygon information. Arc has start and end nodes, left and right polygon identifications, and points between nodes. Figure 2.3 shows an example of line and arc objects.

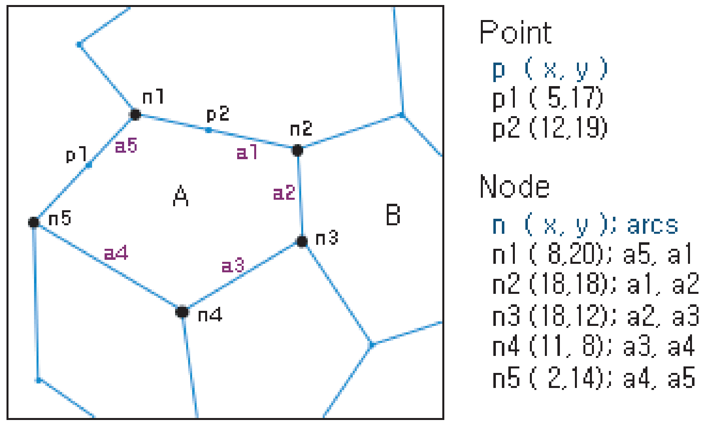


Figure 2.2: Point and node objects

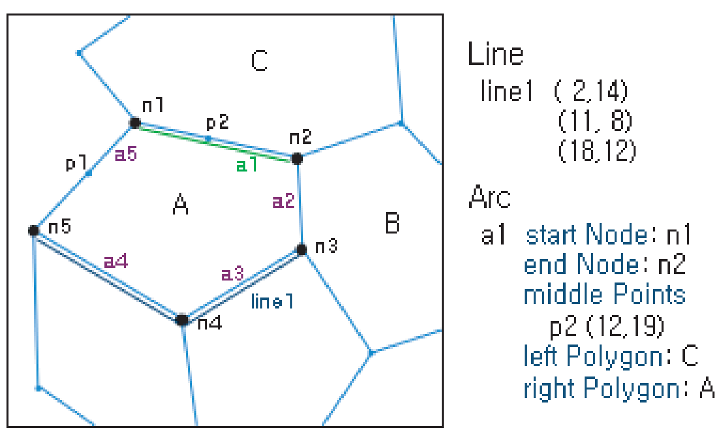


Figure 2.3: Line and arc objects

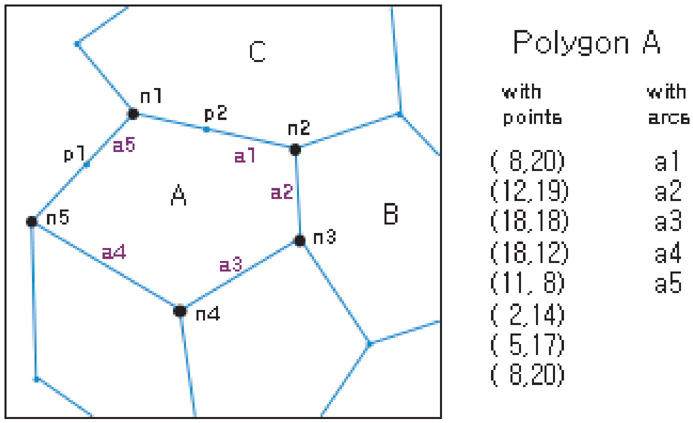


Figure 2.4: Polygon consisting of points and arcs

2.1.5 Polygon

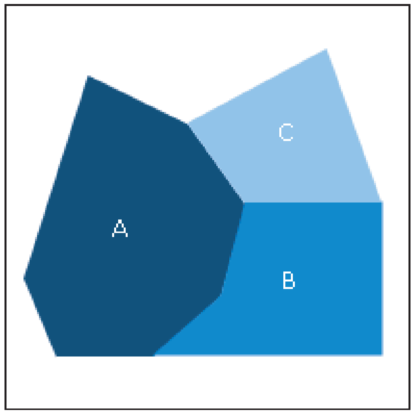
Polygon is a feature used to represent areas such as swamps or lakes. It can be a set of x and y coordinates as the same of a line, but start and end points should be same because polygon is a closed polyline. Lines of polygon should not intersect. Polygon also can consist of arcs. In this case, polygon does not have a set of points but a set of arcs which has adjacent polygon's information. Figure 2.4 shows and example of polygons with points and arcs.

2.2 Vector Data Models

Vector map can be based on several different data models. Common to all these models is that they contain one or more geographical objects. Some models contain also information about object relations. This following section introduces two different vector data models: Non-Topological Model and Topological Model, and shows how they are implemented in real data.

2.2.1 Non-Topological Model

This is the simplest vector data model that stores the data without establishing relationships among the geographic features. This is sometimes called the *spaghetti model*, because lines overlap but do not intersect, just like spaghetti on a plate. All



A	B	C
x,y	x,y	x,y
(2,2)	(6,2)	(8,11)
(6,2)	(16,2)	(10,8)
(9,4)	(16,8)	(16,8)
(10,8)	(10,8)	(14,15)
(8,11)	(9,4)	(8,11)
(3,13)	(6,2)	
(1,5)		
(2,2)		

Figure 2.5: Polygons with spaghetti model

objects in the map are stored as independent entities and each is represented as a set of x and y coordinates (See Fig. 2.5).

The best advantage of spaghetti model is simplicity. In addition, it is easy for end users to input new objects because all objects are independent. On the other hand, there are disadvantages of this model and they are mostly because of the lack of topological information such as adjacency. For example, if we want to know which boundaries are shared with other polygons, we need expensive process. Secondly, data is stored with some redundancy because lines between adjacent polygons must be represented twice. If data size is large then waste of memory will be noticeable. Thirdly, risk of inconsistency exists. If we use different sources of information or change or move some objects, there can be a gap or sliver between adjacent polygons.

2.2.2 Topological Models

There are two different topological models - Network Model and Topological Model. They are similar that they have nodes and arcs. In fact, network model does not have perfect topological structure. It is mainly for network (graph)-based data such as transportation services. Node is an intersection point between different lines and arc is a line which starts and ends with nodes. This model does not include relationship between 2D objects. Therefore, network model is useful for finding an optimal path using the connectivity. There are planar and non-planar networks. In a planar network, each line intersection is chosen as a node, even though that node is not a geographical object. In non-planar network, it is possible that lines may cross and intersection is not a node. An example of network model of planar and non-planar networks is shown in figure 2.6.

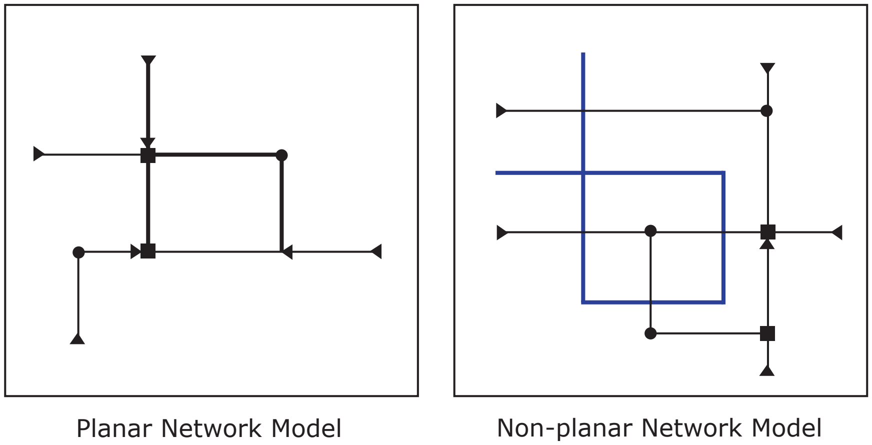


Figure 2.6: Network model - planar and non-planar

Topological model has relationship information between adjacent polygons. Node and arc are same with ones in network model except that arc has information which polygon is on left and right side. In addition, polygon consists of a series of arcs, not points. Nodes and arcs are not duplicated and they can be referenced to more than one polygon. Boundaries which are shared by two polygons will be stored only once, so redundancy problem in spaghetti model can be solved. This is one of advantages of topological model. Another benefit is efficiency to ask topological queries. For example, if you want to search a polygon adjacent to a given polygon P, then check the arcs of P. Each arc will give the information of adjacent polygons. In addition, it is easier to maintain consistency when the map data is updated or edited. In non-topological model, there may be errors when the map is edited. On the other hand, in topological model, there is no error, because the border arc is shared between two polygons (See Fig. 2.7).

There are also disadvantages in this model. Data structure is more complex than spaghetti model, so it may slow down some other operations. Another one is that topology should be established again after each updating.

Existing topological data formats

DXF (Drawing Interchange Format)
DXF files are defined to assist in interchanging drawings between AutoCAD and other programs. DXF files are standard ASCII text files. They can be easily translated to the formats of other CAD systems or other programs for specialized analysis.

DIGEST (Digital Geographic Information Exchange Standard)
DIGEST is developed by DGIWG (Digital Geographic Information Working Group)

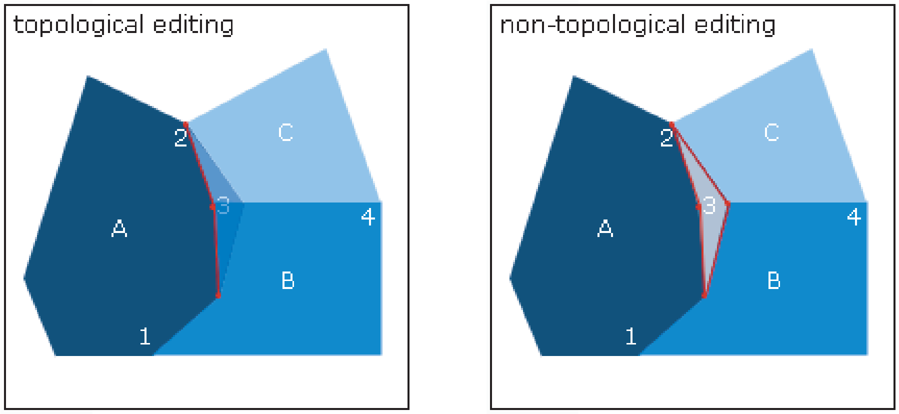
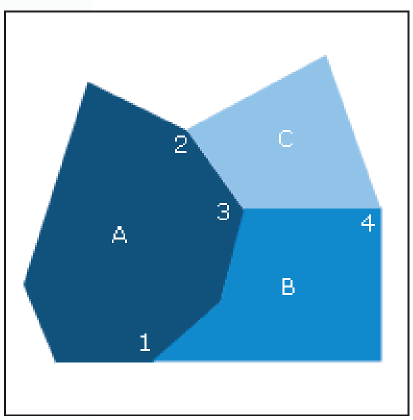


Figure 2.7: Editing in a vector map with topological and non-topological models



Arc

id	start node	end node	left poly	right poly	mid-points
1	1	2	A	out	(2,2)(1,5)(3,13)
2	2	3	A	C	0
3	3	1	A	B	(9,4)
4	3	4	B	C	0
5	4	1	B	out	(16,2)
6	2	4	C	out	(14,15)

Node

id	x,y
1	6,2
2	8,11
3	9,4
4	16,8

Polygon

id	arcs
A	1,2,3
B	3,4,5
C	2,4,6

Figure 2.8: Vector map with topological model

to support data exchange and co-production among NATO nations. It supports raster, vector, and matrix data exchange and the entire range of topological structures from no topology to full topology.

TIGER (Topologically Integrated Geographic Encoding and Referencing)
TIGER is digital database developed at the U.S. Census Bureau to support its mapping needs for the Decennial Census and other Bureau programs. TIGER/Line files are for geographic features like roads, rivers, lakes, legal boundaries, etc.

TIGER/Line data format consists of

- Node : topological junction of two or more links or chains, or end point of a chain
- Entity point : point for identifying the location of point features like towers, buildings, etc.
- Chain : simple polyline with start and end nodes and list of intermediate points. A complete chain has references to left and right polygons and a network chain doesn't have.
- GT-polygon : list of complete chains that form its boundary.

STDS (Spatial Data Transfer Standard)
U.S. Geological Survey (USGS) developed STDS for academic, industrial and federal, state, and local government users of computer mapping and GIS.

NTF (National Transfer Format)
NTF files are provided by the Ordnance Survey in the United Kingdom.

2.3 Building A Topological Structure

Why topology is necessary? Topology is a mathematical approach that allows us to structure data based on the relationships between objects. These relationships are connectivity, contiguity and containment. *Connectivity* refers to the interconnected pathways or networks, such as streets, electrical power lines, streams and transportation networks. Connectivity functions are useful to find optimal routes through the network. *Contiguity* is the spatial relationship between objects that touch each other. Adjacency has same meaning with contiguity. *Containment* refers to the intersection between objects, for example, by boolean relationships such as "and" "or" "inside" "outside" "intersecting" "non-intersecting" etc. Therefore, topological data model can quickly answer these queries:

- Which roads are connected to the center?
- How many people have a car in the neighboring region?
- Where the factory can be built in? - not in the forest "and" not close to the center

Library for building a topological structure from simple spaghetti vector map are built for this paper. First we will look into the vector map with spaghetti model, and how to find nodes and arcs, then lastly, topological vector map and XML output files. Vector map files are ASCII files for easy input and editing.

2.3.1 Non-topological Vector Map

This file is simple. It has label, number of points, and a list of points. Point has X and Y coordinates and one number (0 or 2) for separating polygons. Figure 2.9 shows an example file.

This file has two objects - Polyline and Point. Polyline class is for polygon, which is closed polyline, or not closed polyline features. It contains 1 to N point objects. In addition, it contains a bounding box for reducing comparing time when finding a neighbor polygon. It has functions for finding nodes and arcs. Point class is for point object. It contains the number of neighbor polygons and their id numbers as well as x and y coordinates. Figure 2.10 shows object diagram of non-topological vector map.

2.3.2 Finding Nodes

Next step is finding nodes. First shared points with adjacent polygons should be found and then count how many neighbor polygons each point has. All points in each polygon should be compared with all points in all other polygons. However, points actually can be shared with close polygons, so not all points need to be checked.

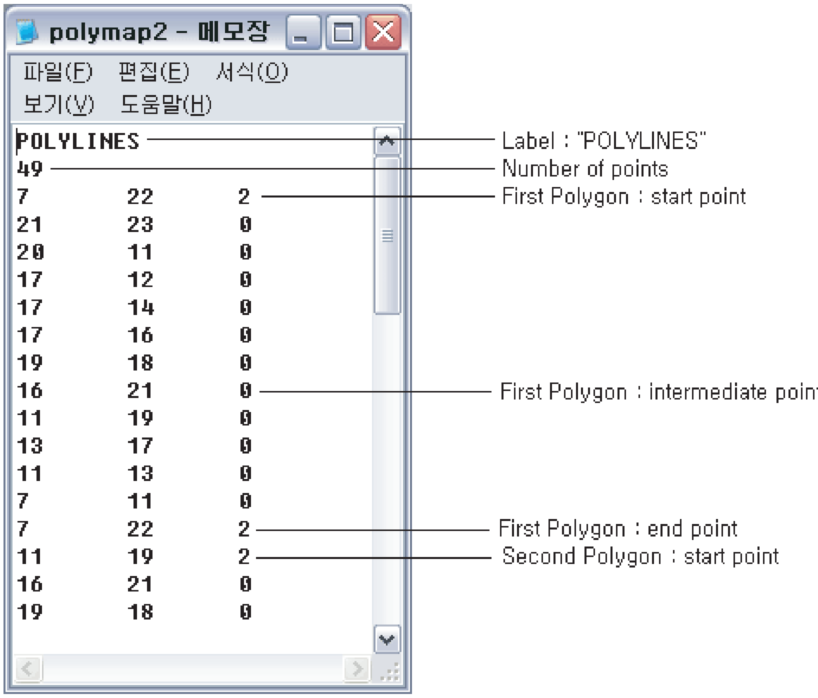


Figure 2.9: Vector map data file with spaghetti model

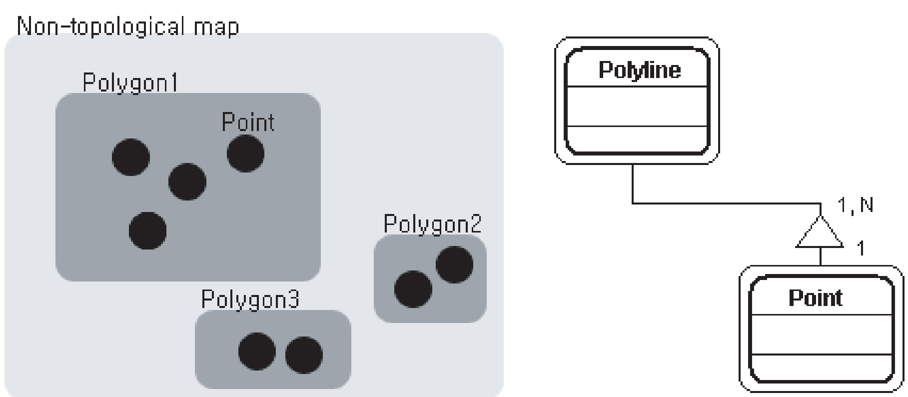


Figure 2.10: Object diagram of non-topological model

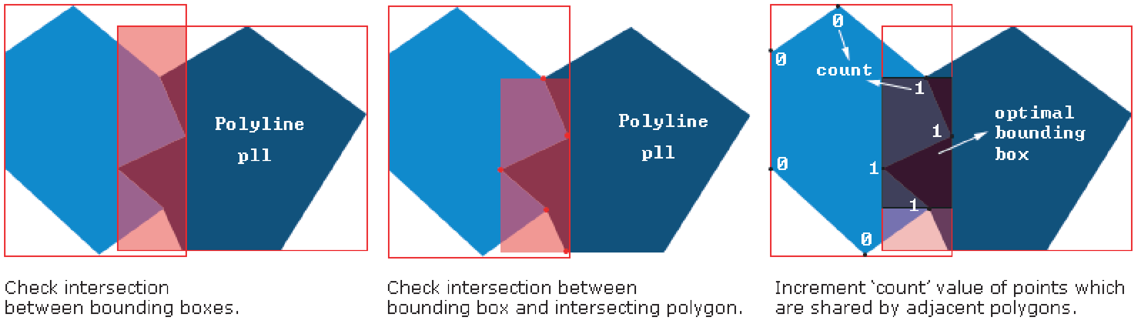


Figure 2.11: Counting adjacent polygons

For this, bounding box of each polygon is used. First check if bounding boxes are intersecting between two polygons, then check only points inside intersection between two bounding boxes. Bounding box is easy to calculate and operations such as including or intersection are cheap. Figure 2.11 shows the process of counting neighbor polygons.

Count values in figure 2.11 will decide which point is a node and which is not. There are several cases that show the point is a node.

1. Count value is more than 2 : Node
2. Count value changes 0 to 1 or 1 to 0 : Node
3. Count values are same with 1 in a row : should check their neighbors. If neighbor polyline id numbers are same, then the point is not a node. If they are different, then it is a node (see Fig. 2.12).
4. Current count value is 1 and previous or next is more than 2 : Node
5. If the polyline is not closed : start and end points are nodes.

In figure 2.13, you can see that shared points have a list of neighbor polygons' id numbers. They will be used for finding arcs.

2.3.3 Finding Arcs

Arc starts and ends with nodes. For each polygon, all points are looked up and if first one node is found, then arc saving starts and middle points will be stored until another node appears. In addition, arc should have neighbor information -

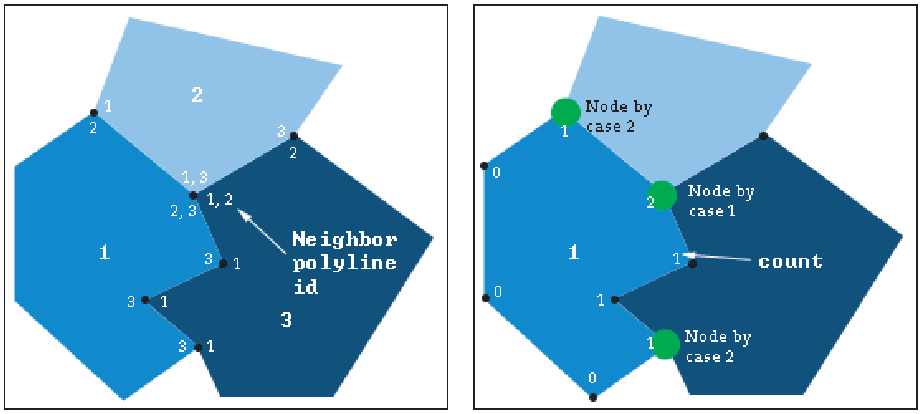


Figure 2.12: Finding neighbor polygons and deciding whether a point is a node or not

left and right polygons' id numbers. If there are any points between start and end nodes, then it is straightforward - checking the neighbor polygon id from the middle points. However, if there is no middle point, then neighbor polygons of start and end nodes are checked. If they have same neighbor polygons, arcs in not-sure array are checked for finding the same arc which has same neighbor polygons. If there is same arc, the arc will be removed from not-sure array and be stored as a normal arc. If there is no, the arc will be new not-sure arc. During the whole process, same arc should not be saved twice.

In addition, while finding arcs, polygon should be saved with new form - referencing arcs but not points.

2.3.4 Topological Vector Map

Finally after building a topological structure from spaghetti vector map, topological vector map will be stored as a file. There are two functions for generating ASCII file and XML file. XML file is easy to see the structure. For loading XML files, existing library - Xerces C++ Parser is used [Apa]. Table 2.1 shows the structure of node, arc and polygon and tags for XML file, and following figure shows ASCII and XML files.

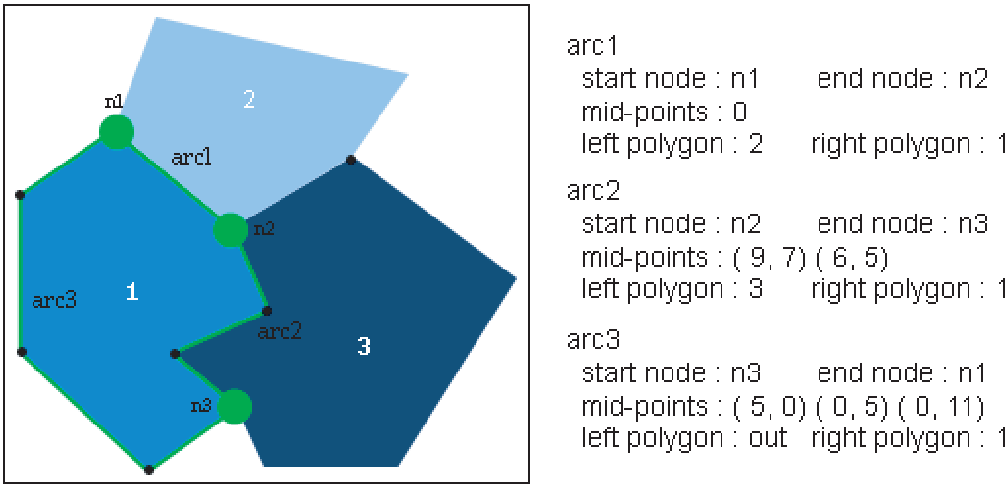


Figure 2.13: Finding arcs

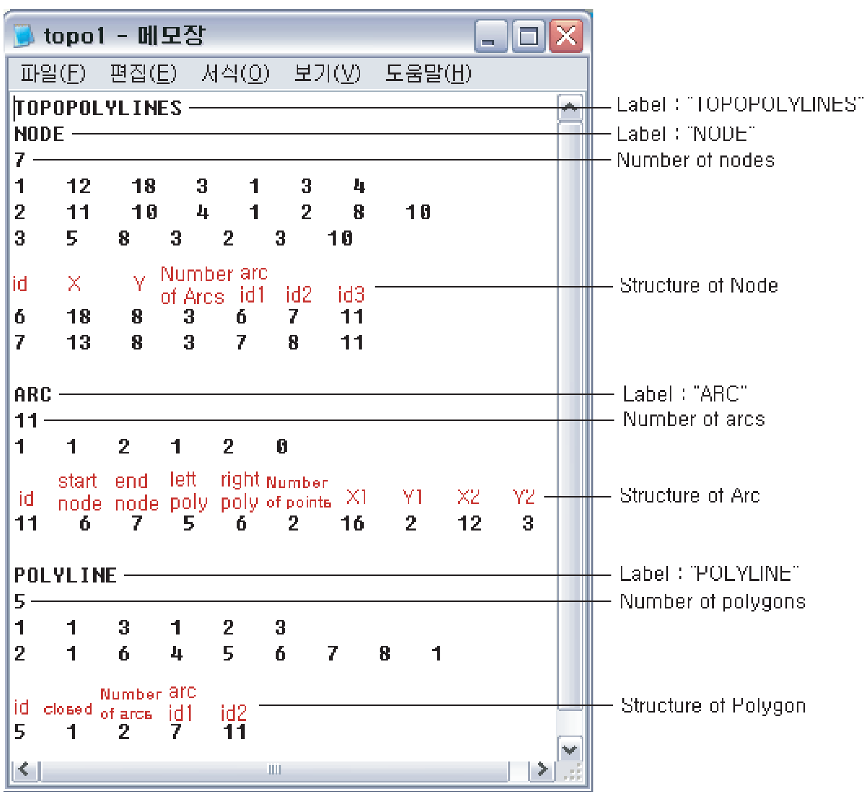


Figure 2.14: Topological vector map data file - ASCII

```
- <topoMap>
  - <nodes>
      <nodeNum>11</nodeNum>
    - <node>
        <nodeId>1</nodeId>
        <nodeX>17</nodeX>
        <nodeY>14</nodeY>
        <belongsArcNum>6</belongsArcNum>
        <belongsArcId>1</belongsArcId>
        <belongsArcId>7</belongsArcId>
        <belongsArcId>9</belongsArcId>
        <belongsArcId>10</belongsArcId>
        <belongsArcId>16</belongsArcId>
        <belongsArcId>17</belongsArcId>
      </node>
```


Figure 2.15: Topological vector map data file - XML

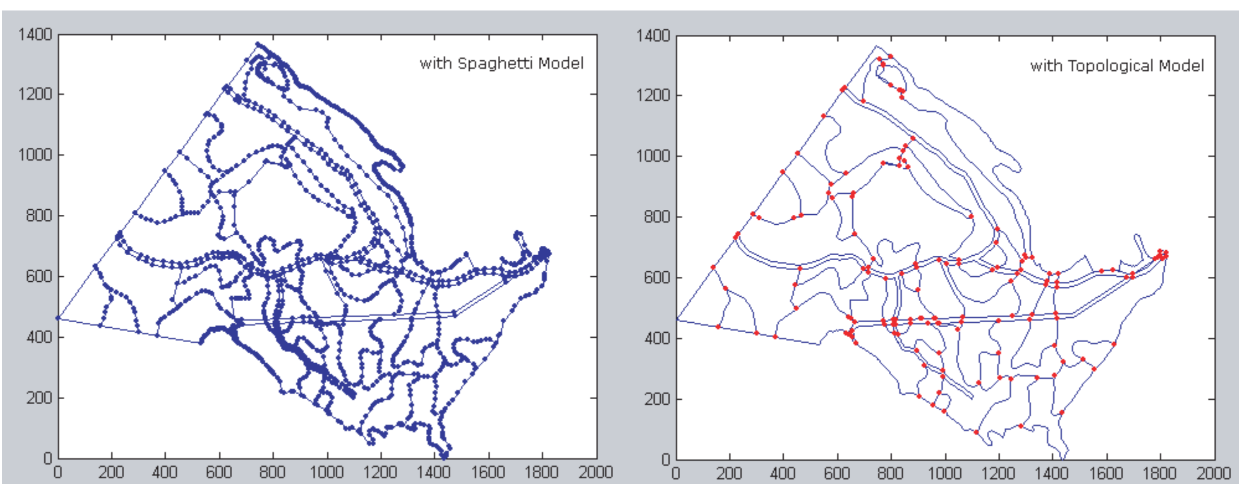


Figure 2.16: Building a topological structure

Node		<nodes>
id	Node id number	<nodeId>
X	X coordinate	<nodeX>
Y	Y coordinate	<nodeY>
Number of Arcs	How many arcs have this node	<belongsArcNum>
Arc id numbers	List of arc id numbers	<belongsArcId>
Arc		<arcs>
id	Arc id number	<arcId>
start node	Start node id number	<startNodeId>
end node	End node id number	<endNodeId>
Left Poly	Left polygon id number	<leftPolyId>
Right Poly	Right polygon id number	<rightPolyId>
Number of points	Number of middle points	<midPointsNum>
X	X coordinate of middle point	<midX>
Y	Y coordinate of middle point	<midY>
TopoPoly		<topoPolys>
id	Polygon id number	<polyId>
closed	Boolean value for checking closed or not	<closed>
Number of arcs	Number of arcs	<ownArcNum>
Arc id numbers	Arc id number	<ownArcId>

Table 2.1: Structure of topological objects and XML tags

Chapter 3

Bounding Containers

This chapter describes bounding containers as a finite geometric object and how minimum rectangle area, which is one of linear bounding containers, is implemented.

3.1 What Is Bounding Container?

Bounding container is a simple geometric object for bounding a complicated object. It is useful for computational geometry application such as ray tracing, collision avoidance, hidden object detection, etc [Suna]. Before doing expensive intersection or containment process of a complicated object, simple process of a bounding container can reduce the possibility of intersection and containment, and no more process is needed. For example, when two complicated objects are far from each other and should be checked for intersection, checking two objects perfectly is not necessary if simple comparing with bounding containers of two objects is done and shows that there is no intersection between them. For this usefulness, bounding containers should satisfy some important requirements [Suna].

- If the bounding container include all points of an object, then it also should include the whole object. For example, if two vertices are inside the bounding container, then the line joining them will be included in it.
- The test for containment and intersection, such as checking one point is inside or outside the container, two bounding containers are disjoint, and a line

intersects the container, should be easy. Therefore, container should have a small number of inequalities to test inclusion of a point.

- The bounding container should be efficient to build and store. Linear time - $O(n)$ and small space for storing are aimed. However, there is trade-off. More efficient container needs more time for processing.
- The container can approximate the object. Smaller area of the container will be more accurate.

There are two basic types of bounding containers - linear and quadratic containers. In this paper, linear containers will be focused. In the following sections, different linear bounding containers will be introduced and then how one of linear containers, minimal bounding rectangle, is implemented will be explained.

3.2 Linear Bounding Containers

A linear container is a convex polygon which is bounded by finite inequalities. In 2D, a container can have k inequalities : $f_i(x, y) = a_i x + b_i y + c_i \leq 0 (i = 1, k)$ [Suna]. If a point(x, y) is true to all inequalities, then it is inside the container. If any inequalities fails, then the point is outside the container. Each inequality decides a half-space $\mathbf{H}_i$ bounded by the line $\mathbf{L}_i : f_i(x, y) = 0$. The intersection of these half-spaces is the region of the container (See Fig. 3.1).

3.2.1 Orthogonal Bounding Rectangle

The orthogonal bounding rectangle is defined by two extreme points (x_{min}, y_{min}) and (x_{max}, y_{max}) and four edges are parallel to the coordinate axes. It has four inequalities, so if all inequalities are true with the point, then the point is inside the box. If any one of inequalities fails, then the point is outside the box. Even though there are four inequalities, on the average, the point will be decided inside or outside after two tests. The test for disjoint of two rectangles is similar to the test for the point. It is done by comparing their minimum and maximum extents of two boxes. For example, if $x_{max1} < x_{min2}$ or $x_{max2} < x_{min1}$, then box1 and box2 are disjoint.

The orthogonal bounding rectangle is the simplest container so that it is used most frequently in many applications. It is simple because minimum and maximum

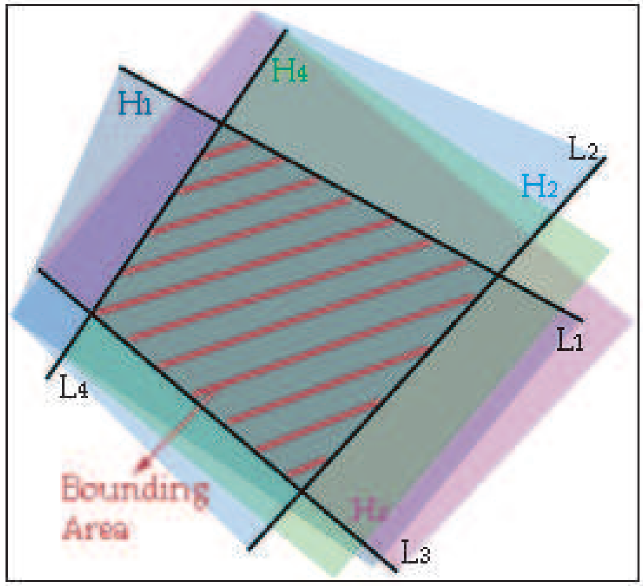


Figure 3.1: $\mathbf{L}_i$, half-space $\mathbf{H}_i$ by $\mathbf{L}_i$, and bounding area

coordinate values can be found easily in linear time $O(n)$ with one scan of all points in the object. In addition, comparing test does not have any arithmetic computing, but only comparing x and y coordinate values with extent values (See Fig. 3.2).

3.2.2 Bounding Diamond

The bounding diamond is a rectangle rotated by 45°, so it looks like a diamond. It has four inequalities and they are computed by the simplest arithmetic expressions, adding and subtracting. They are $p = (x + y)$ and $q = (x - y)$ which are lines with slopes of -1 and 1. All points will be scanned, p and q computed, and then $(p_{min}, p_{max}, q_{min}, q_{max})$ will be found. For the test of point inclusion, it needs a bit more computation than the bounding box, but it still can be done in $O(n)$ time with single scan of all points in the object. Also disjoint test of two objects is easy because only parallel edges will be compared. Figure 3.2 shows the example of the bounding diamond.

3.2.3 Bounding Octagon

The bounding octagon is the combined geometric object of an orthogonal bounding rectangle and bounding diamond. It thus is defined by eight inequalities. The

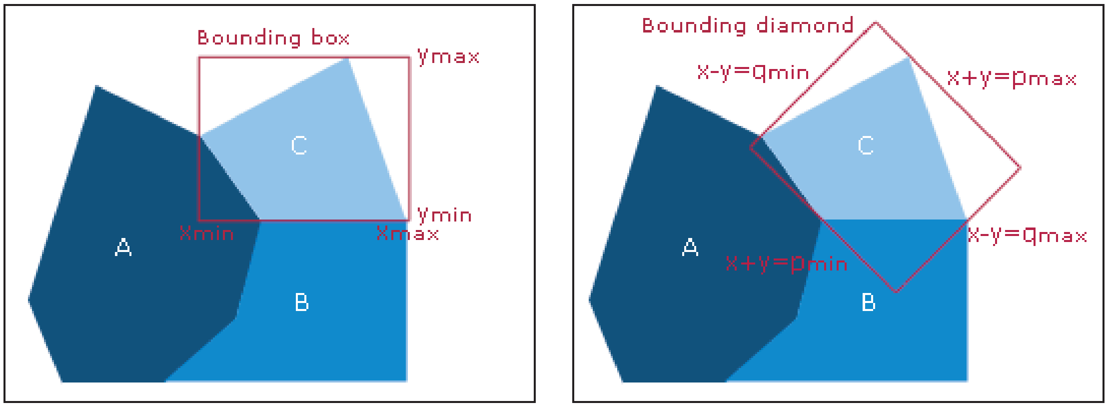


Figure 3.2: Orthogonal bounding rectangle and bounding diamond

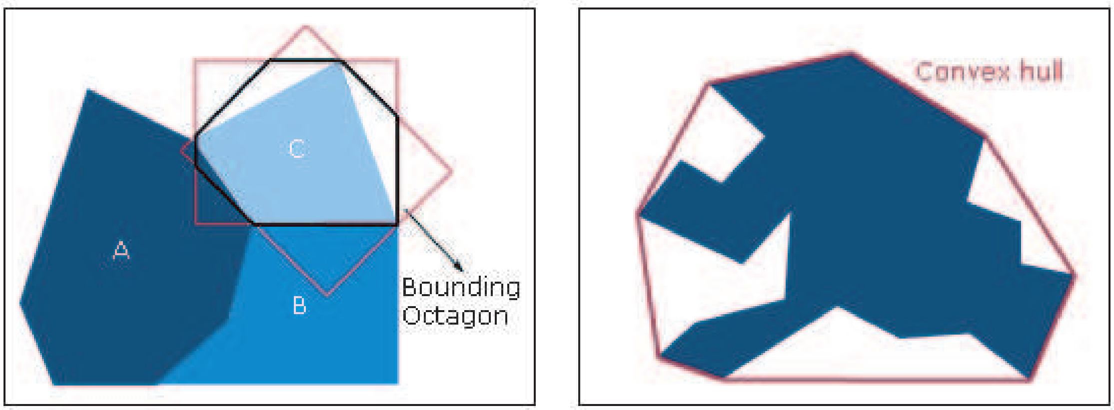


Figure 3.3: Bounding octagon and convex hull

bounding octagon is used frequently because it is smaller area then the orthogonal bounding rectangle and bounding diamond and still can be computed in linear time. For example, first, the point inclusion test can be checked by extents of the orthogonal bounding rectangle. If the point is inside, secondly, $(x + y)$ and $(x - y)$ will be calculated and the point inclusion is decided by the bounding diamond. The test for disjoint of two octagons is processed similarly with the point inclusion test. Figure 3.3 shows the example of the bounding octagon.

3.2.4 Convex Hull

Convex hull is the smallest convex set of points of an object. It is easy to understand if you imagine surrounding the set of points by a large, stretched rubber band

[PS85, dBvKOS00]. Because it is the smallest region, it approximates an object most closely and it has the least area among all bounding containers. Each boundary can be defined by a linear equation ($ax+by+c=0$). Therefore, the point inclusion test can be done with an inequality : $(ax+by+c) \leq 0$. An example of a convex hull is in Figure 3.3. In spite of the most accurate approximation of an object, convex hull is not used practically as a bounding container, because it may have a lot of boundaries and then it needs much computation for checking independent inequalities. Moreover, the test for disjoint of two convex hulls is more complicated, because two hulls can not have always opposed parallel edges. There are many existing algorithms for computing the convex hull - Grahamhull, Gift-wrapping approach, Quikhull, Mergehull, etc [PS85].

3.2.5 Minimal Bounding Rectangle

Minimal bounding rectangle is the result of combining two features which are minimizing area of the container and reducing inequalities for point inclusion test. Therefore, it approximates an object more precisely and, at the same time, it has only four inequalities, so easy and fast to decide the point is inside or outside the container. It has two pairs of parallel lines, $f1 = (a_1x + b_1y)$ and $f2 = (a_2x + b_2y)$, and each pair has minimum and maximum extents. If a point $P(x, y)$ fulfills

$$f1_{min} \leq a_1x + b_1y \leq f1_{max}$$
$$f2_{min} \leq a_2x + b_2y \leq f2_{max}$$

then P is inside the rectangle [Suna]. For the algorithm finding a minimal bounding rectangle, 'Rotating Calipers' [Tou83] can be used because it can compute the minimal bounding rectangle in $O(n)$ time if an object is convex. If an object is not convex, then first, a convex hull should be found. More details about how 'Rotating Calipers' is used will be explained in following section.

3.3 Implementation of Minimal Bounding Rectangle

In this chapter, how to implement minimal bounding rectangle will be described. If we use 'Rotating Calipers', time complexity can be $O(n)$, but object should be convex. We will compute minimal bounding rectangles mainly for arcs in this paper, therefore, first should make a convex hull for each arc before minimal bounding rectangle. Algorithms for convex hull and rotating calipers will be explained.

3.3.1 Algorithm for convex hull

For finding a minimal bounding rectangle for an arc, convex hull for each arc should be computed. There are existing algorithms for convex hull and general computing time is $O(nlogn)$. This is because all points should be sorted before finding a hull and sorting algorithm generally takes $O(nlogn)$. After sorting, computing a hull takes $O(n)$ time. However, there is more efficient algorithm for connected simple polyline by (Melkman,1987). Arc is a connected simple polyline because it is a series of ordered points and there is no self-intersection. Therefore, Melkman's algorithm can be applied for an arc. Important features of his algorithm are

1. It works for a simple polyline.
2. It does not need preprocessing for sorting. All points will be processed sequentially once.
3. It uses a double-ended queue (a deque) to store processed points which indicates an increasing hull [Sunb].

The deque (double-ended queue) has both top and bottom. It allows one to push or pop on the *top* of deque and to insert or remove from the *bottom* of the deque. Melkman's algorithm is straightforward. It processes each point of the polyline at each stage. Let the simple polyline be PL = $P_0, P_1, ..., P_n$. Initial convex hull is made with first three points, and then the next point P_k is considered in each stage. If point P_k is inside the current convex hull, then it can be ignored. Therefore, convex hull CH_k will be same with CH_{k-1}. If it is outside the current convex hull, then new convex hull should be built. The new point simply can be added at the bottom and top of the deque. However, points which will be inside the new convex hull should be removed before adding new point for new increased convex hull into the deque. Figure 3.4 shows how his algorithm works.

Melkman Algorithm

---

1. Make a convex triangle with first three points.
2. Test that next point is inside the convex hull. If it is inside, then skip this point and continue to next point.
3. Remove points which will be inside new convex hull from the bottom of the deque, then insert this point.

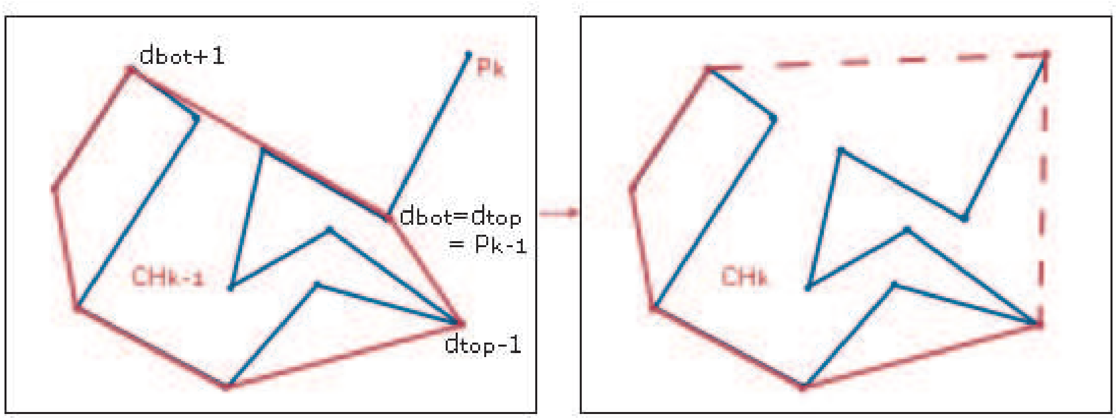


Figure 3.4: Convex hull by Melkman's algorithm

4. Remove points which will be inside new convex hull from the top of the deque, then push this point.

5. Repeat steps 2 to 4 until all points in the polygon are tested.

3.3.2 Rotating Calipers

If convex hull of an arc object is ready, the process to find a minimal bounding rectangle can be computed in linear time using rotating calipers [Pir99]. 'Calipers' are two pairs of parallel lines around the convex hull and these pairs are orthogonal to each other. They are initialized with extreme points and rotated until calipers meet the edges of convex hull. This process can find a minimal area rectangle because the rectangle of minimum area enclosing a convex polygon has a side collinear with one of the edges of the polygon [Tou83].

We can define a minimal bounding rectangle R with a given convex polygon P such that $\forall p \in P, p \in R$. If area$(R) \leq$ area(R') for all rectangles R', then R is a minimal bounding rectangle for P. In order to minimize the area, we can intuitively think that the rectangle's edges would have to touch the convex polygon. Here is this theorem and proof of it [Pir99, HR75].

Theorem: *The rectangle of minimum area enclosing a convex polygon has a side collinear with one of the edges of the polygon.*

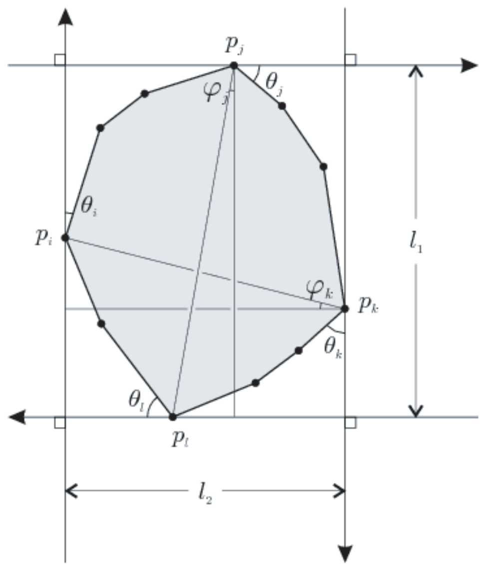


Figure 3.5: An example of enclosing rectangle P

Proof: We have a given convex polygon P, and let us assume that the smallest box is given and it does not have one side collinear with one of P's edges. In figure 3.5, the rectangle only touches P at four points p_i, p_j, p_k, p_l. We can prove that it is always possible to find a smaller enclosing rectangle.

A, the area of the enclosing rectangle is $l_1 l_2$ (See Fig. 3.5). Let $d_{ik} = dist(p_i, p_k)$, and $d_{jl} = dist(p_j, p_l)$. Therefore we get

$$l_1 = d_{jl} \cos(\varphi_j)$$

$$l_2 = d_{ik} \cos(\varphi_k)$$

Both l_1 and l_2 can be reduced by rotating their corresponding lines in their preferred direction of rotation. Therefore there are two cases - case 1, where l_1 and l_2 can be decreased by rotating all lines in the same direction, and case 2, where rotating in a given direction decreases one length but increases the other.

Case 1: By rotating all lines counterclockwise by some angle η, both l_1 and l_2 are decreased. A', the area of new box is determined by edges of length l_1' and l_2' where

$$l_1' = d_{jl}\cos(\varphi_j + \eta) \Rightarrow l_1' < l_1$$
$$l_2' = d_{ik}\cos(\varphi_k + \eta) \Rightarrow l_2' < l_2$$
$$\Rightarrow A' = l_1' l_2' < A$$

In this case it is always possible to find a smaller enclosing rectangle.

Case 2: The preferred directions of rotation are different. Let us define δ_j as the maximum angle we can rotate the lines in l_1's preferred direction of rotation before we hit the edge, and in a same way we define δ_k for l_2. Let $\delta = min(|\delta_j|, |\delta_k|)$. Assume that the preferred direction of rotation for l_1 is clockwise and the preferred direction of rotation for l_2 is counterclockwise. If we rotate clockwise, we get new lengths l_1', l_2' and a new area A_C:

$$l_1' = d_{jl}\cos(\varphi_j + \delta)$$
$$l_2' = d_{ik}\cos(\varphi_k - \delta)$$
$$\Rightarrow A_C = l_1' l_2'$$

If we rotate counterclockwise, we get:

$$l_1'' = d_{jl}\cos(\varphi_j + \delta)$$
$$l_2'' = d_{ik}\cos(\varphi_k - \delta)$$
$$\Rightarrow A_{CC} = l_1'' l_2''$$

If $A_C/A < 1$ then we rotate clockwise and we can get a smaller enclosing rectangle. However, if $A_C/A \geq 1$, then we have:

$$\frac{A_C}{A} = \frac{\cos(\varphi_j + \delta)\cos(\varphi_k - \delta)}{\cos\varphi_j \cos\varphi_k} \geq 1$$

$$\Leftrightarrow \cos^2\delta + (\tan\varphi_k - \tan\varphi_j)\cos\delta\sin\delta - \tan\varphi_j\tan\varphi_k\sin^2\delta \geq 1$$

$$\Leftrightarrow (\tan\varphi_k - \tan\varphi_j)\cos\delta\sin\delta \geq \cos^2\delta - \tan\varphi_j\tan\varphi_k\sin^2\delta - 1$$

$$\begin{aligned}\Rightarrow \frac{A_{CC}}{A} &\leq 2(\cos^2\delta - \tan\varphi_j \tan\varphi_k \sin^2\delta) - 1 \\ &\leq 2(1 - \sin^2\delta - \tan\varphi_j \tan\varphi_k \sin^2\delta) - 1 \\ &\leq 1 - 2(1 + \tan\varphi_j \tan\varphi_k)\sin^2\delta \\ &< 1\end{aligned}$$

Hence we get $A_{CC}/A < 1$, and it means that we can obtain a smaller enclosing rectangle by rotating counterclockwise.

Therefore, for both of cases, it is possible to have a smaller enclosing box.

Rotating Calipers Algorithm

---

1. Find four points with minimum and maximum x and y-coordinates for the polygon - $P_{Xmin}, P_{Xmax}, P_{Ymin}, P_{Ymax}$.
2. Construct two sets of "calipers", parallel to x and y axes, thus forming a rectangle enclosing the polygon.
3. Let $\theta = min(\theta_i, \theta_j, \theta_k, \theta_l)$.
4. Rotate the lines by θ, thus until any of them meets the edge of the polygon.
5. Calculate the area of a rectangle built by four lines and compare with minimum area. If it is smaller, then keep the new rectangle as our new "minimum".
6. Recompute $\theta_i, \theta_j, \theta_k$, and θ_l.
7. Repeat steps 3 and 6, until the lines are rotated an angle more than 90°.

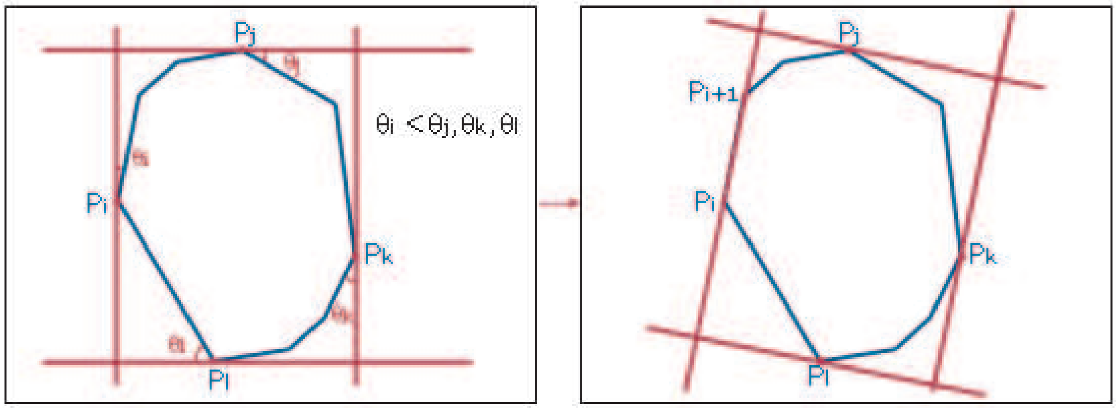


Figure 3.6: Rotating calipers

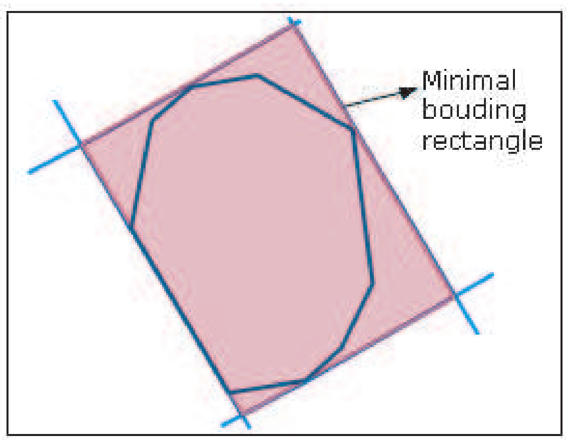


Figure 3.7: Minimal bounding rectangle by using rotating calipers

Algorithm 1 Arc.calcSmallBBox()

```
CH ← calculate_convexhull();
for (all points of convex hull) do
   p ← initial points by x_min, x_max, y_min, y_max
end for
calculate_calipers();
box ← calculate_MBR();
while (sumθ < 90°) do
   for (k=0;k<4;k++) do
      θ ← angle between the caliper p[k] and new caliper with next point
      if θ < minθ then
         minθ ← θ
         minP ← k
      end if
   end for
   rotate_caliper(k);
   calculate_calipers();
   sumθ ← sumθ + minθ
   tempBox ← calculate_MBR();
   area ← area(tempBox);
   if area < minArea then
      box ← tempBox
      minArea ← area
   end if
end while
```

Chapter 4

Hierarchical Representation of Arcs

This chapter describes hierarchical representation schemes for arcs and different methods of them. Two commonly used tree structures, strip and arc tree, will be explained and new approach with a splitting point decided by the minimum area of the bounding container will be introduced.

4.1 Hierarchical Representation

Curves are important two-dimensional structures in many areas. For example, curves are used to represent map features such as contour lines, roads, and rivers in geography. If a map is huge and very large amount of data is involved, efficiency to perform operations, such as finding an intersection of road and river or checking some point features are inside or outside of some areas, on this data is crucially needed. Hierarchical tree structure for representation of curves is one of methods to do these operations more efficiently because the operations are performed faster at lower resolutions than the ultimate resolution [Bal81]. lt is built recursively and added more detailed features of the curve. Every next level has more points of the curve, so the curve can be represented more precisely. These points that are chosen for hierarchical structure are not independent each other [SRS03]. This is because a new point for next level should be chosen between start and end points of preceding level representation. Hence, as building more levels, the curve will be subdivided recursively into shorter sub-curves. Each tree node is this sub-curve and it is approximated by bounding containers. If the curve is well-behaved, intersection and point inclusion calculations can be solved in $O(logn)$ where n is the number of points of the curve.

There are various well-known schemes for hierarchical representation of curves. They are Strip Tree [Bal81], Arc Tree [GW90] and Bezier Tree [Bez74]. These schemes are mainly different with what kind of bounding container is chosen, how dividing point is decided, and how much information is stored in each level. In following sections, Strip Tree and Arc Tree will be explained and additionally, a new tree by different approach to how to decide a dominant splitting point will be described. This paper is focused on how different method of decomposition - it means which point is decided as a splitting point - is performed and compared. Therefore, all trees use minimal bounding rectangle as a bounding container in common.

In this paper, arcs in a topological map are similar with curves, so hierarchical representation method is used for arcs.

4.2 Strip Tree

4.2.1 Strip Tree definition

Strip tree was proposed by Dana H. Ballard in 1981. It has a binary tree as a hierarchical structure, and a node of the tree has a strip which bounds a curve and pointers to left and right children nodes. A strip is defined by six values $(P_s(x_s, y_s), P_e(x_e, y_e), w_r, w_l)$ where (x_s, y_s) is starting point of the strip, (x_e, y_e) is ending point, and w_r and w_l are right and left distances from the directed line between the starting and ending points of the strip to the strip borders [Bal81]. Figure 4.1 is a strip segment defined.

Root of the strip tree has a bounding rectangle for the entire curve, and the curve is divided to two sub-curves by a splitting point. This splitting point is decided by the farthest distance between the point and the directed line $\overline{P_s P_e}$. This process is recursively done to the two children until every strip has a width $w = w_r + w_l$ which is less than predetermined limit value.

Figure 4.2 shows the process of building a strip tree for a curve C. Root strip S_1 is divided to two strips S_2 and S_3 first, and then strip S_3 is divided again to two parts because the width of the strip is longer than the limit length. S_3 is divided to S_4 and S_5, then the process is finished.

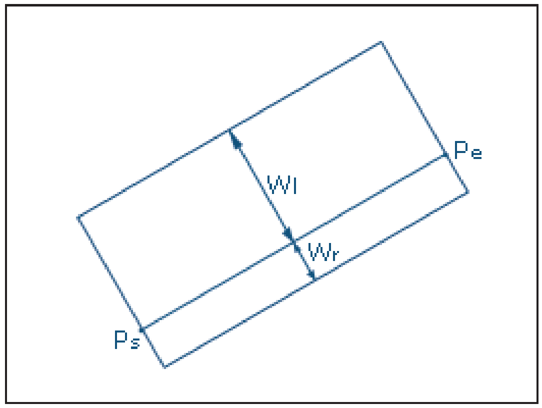


Figure 4.1: Definition of a strip segment

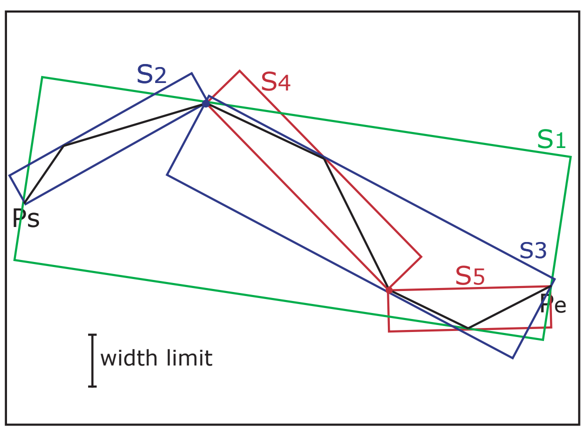


Figure 4.2: Building a strip tree by top-down method

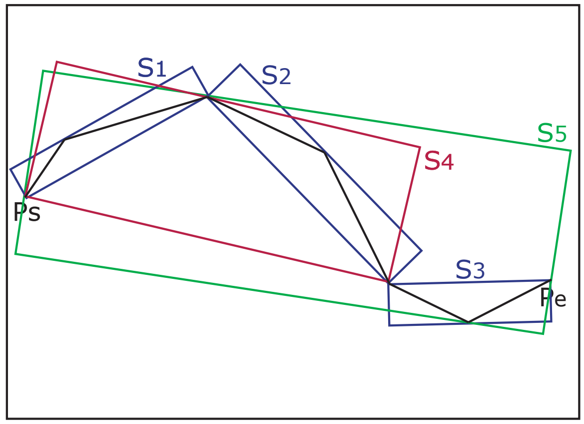


Figure 4.3: Building a strip tree by bottom-up method

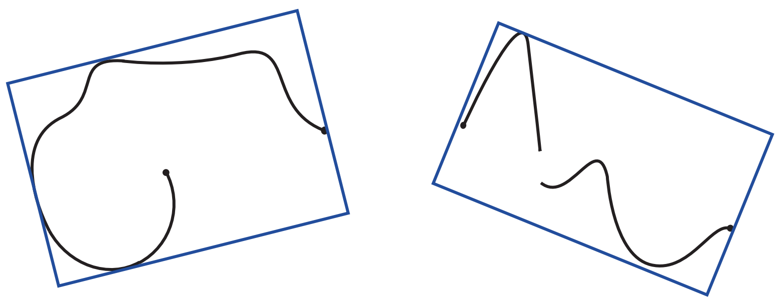


Figure 4.4: Non-regular strips

This process is a top-down method. This method needs a search to find the splitting points in each node. Each point is checked at each of the $log_2 n$ levels, thus it takes $O(nlog_2 n)$ time. There is the second method in bottom-up style. First make strips $S_0, S_1 \ldots S_{n-1}$ for each successive pair of points $(P_0, P_1)(P_1, P_2) \ldots (P_{n-1}, P_n)$, then make pairs with strips, that is, $(S_0, S_1)(S_2, S_3) \ldots$, and cover them with larger strips. Continue until there is a single strip as a root. It takes $O(n)$ time, but approximation result is not better than the first method. Figure 4.3 shows the bottom-up method.

The example of the curve above is *regular* which means that the curve is connected and its end points are on both end edges of strips [Bal81]. There are more complex curves such as closed one, curve which extend its end points, or curve which consists of disconnected segments. These curves need more complex calculation for finding a bounding strip. Examples are in Fig. 4.4.

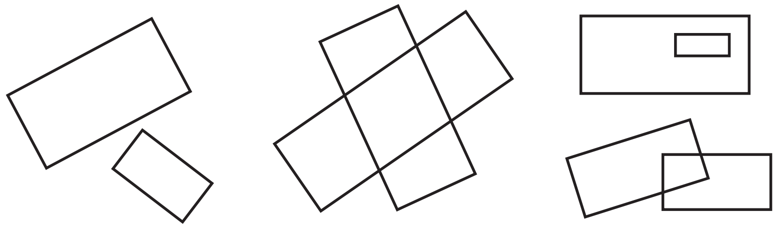


Figure 4.5: Three possible results of intersecting two strips

```
struct stripTree
{
    unsigned long int start, end;
    double wL, wR;
    smallBbox * bbox;
    stripTree * leftchild;
    stripTree * rightchild;
};
```

Figure 4.6: Data structure of strip tree

Strip tree is useful to find intersection between curves such as finding in which area river and road crosses. For solving this query, first intersection between strip trees should be checked. There are three different cases - null, clear, and possible (See Fig. 4.5).

If the result is null, then it means that there is no intersection. If the result is clear, then two strips are clearly intersecting. If the result is possible, then they may be intersecting, so more specific process is necessary. Thus, their children nodes should be checked. The process is going on in this way until the result is determined null or clear. If strips are more precise, so if the answer - null or clear - is determined faster, then execution time will be saved a lot. That is why decomposition of strips is important. For well-behaving curves, execution time is expected to $O(log_2 n)$ where n is the number of points constructing the curve.

4.2.2 Implementation of Strip Tree

Strip tree which is implemented in this paper is a little different with definition of strip tree. Minimal bounding rectangle is used as a bounding container instead of a strip. Figure 4.6 shows data structure of a strip tree.

Node of strip tree has start and end point information, wL and wR values, smallBbox which is minimal bounding rectangle, and pointers to left and right children nodes. Width of a strip, wL and wR, is used for deciding a splitting point. The farthest point from a line connected between start and end points is the one which divides the curve to two strips on next level.

Algorithm 2 buildStripTree()

```
if there are only two points then
  Finish building the strip tree
else
  calculate_MBR(box,start,end);
  division ← Find the farthest point from the line connected with two end points
  buildStripTree(start, division)
  buildStripTree(division, end)
end if
```

Strip tree is recursively built until node has only two points, that is one line segment. This is because the exact intersecting segment should be found. Figure 4.7 shows the example of building a strip tree and finding an intersection with a random segment.

4.3 Arc Tree

4.3.1 Arc Tree definition

Arc tree was proposed by Günther in 1987. It is close with strip tree but the rule of decomposition of the curve is different. The curve is divided based on its length to several sub-polylines. All sub-polylines should have same length. Thus, the curve is approximated to the connected line between two endpoints in the first level of the arc tree, then the curve is divided to two sub-polylines of same length by a midpoint recursively as the tree is built deeper. Figure 4.8 shows how the arc tree is built.

If the curve C has kth arc tree and its length is l, then it means that C is approximated with 2^k line segments and the length of each line segment is $l/2^k (k \geq 0)$. A function *C(t)* is defined in interval [0:1] with 2D Euclidean space. Thus, the k^{th} approximation of *C(t)* is a sequence of line segments consisting of points $C(i/2^k)$ and $C((i+1)/2^k)$, $0 \leq i < 2^k$. The approximation process is done recursively until the error is less than a given limit.

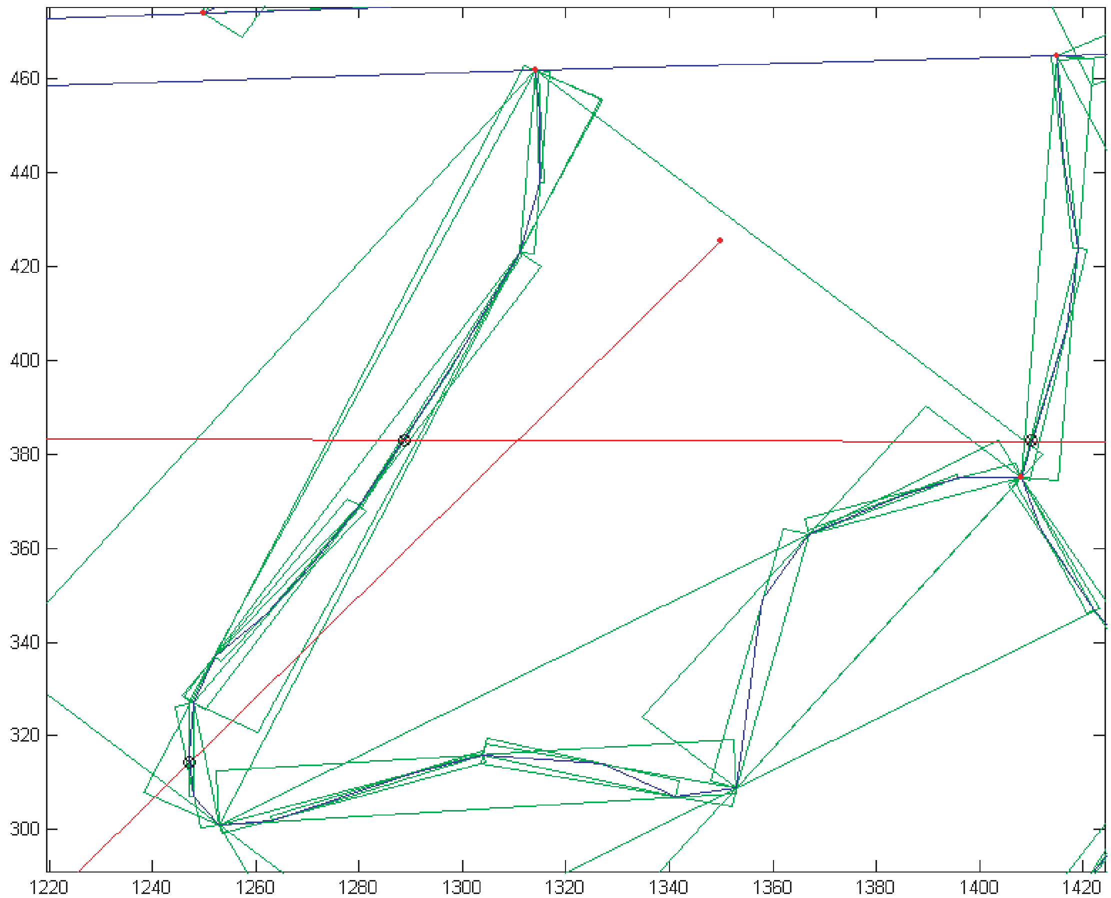


Figure 4.7: Strip tree with minimal bounding rectangle and finding intersections with random line segments

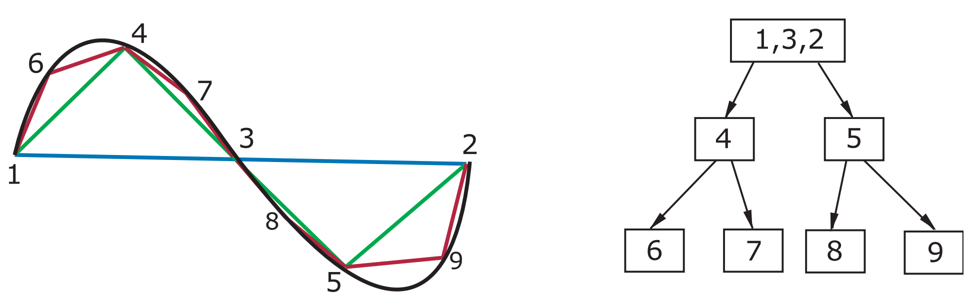


Figure 4.8: Building an arc tree

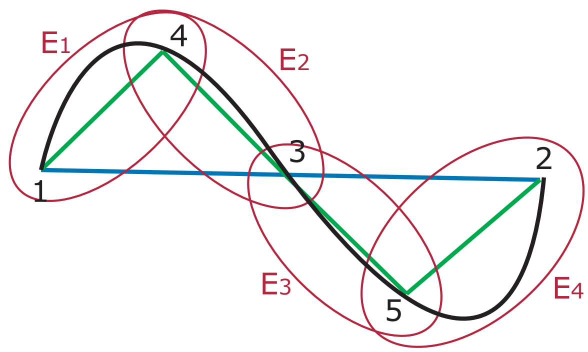


Figure 4.9: Arc tree with ellipses

The construction of an arc tree includes two processes. One is dividing the polyline by the length and the other is calculating a bounding container. For an arc tree, an ellipse is used for a bounding container. This ellipse is defined by a major axis whose length is $l/2^k$ and two focal points which are at $C(i/2^k)$ and $C((i+1)/2^k)$ (See Fig. 4.9).

Using ellipses as a bounding container has an advantage over using a strip in a strip tree, such as no need to worry about closed curves or curves that extend their two endpoints. However, ellipses are not easy to use. For example, when two polylines are intersecting, the intersection of ellipses should be tested first. This is not a simple operation. Therefore, bounding box or bounding circle is used more often instead of an ellipse.

4.3.2 Implementation of Arc Tree

The curves used in this paper consist of straight line segments. Therefore, we do not need artificial points $C(i/2^k)$ but use the median point. For example, if the curve has $n+1$ points labeled $p_1, p_2, \cdots, p_{n+1}$, it will be decomposed at $p_{\ulcorner n/2 \urcorner}$. Thus, the depth of the tree will be $log_2 n$ in maximum. This is called *polygon arc tree* [GW90].

In the definition of the arc tree, an ellipse was a bounding container. However, because of a complex operation, minimal bounding rectangle is used instead of an ellipse in this paper. You can see the data structure of the arc tree in figure 4.10. It is similar with the strip tree.

Building an arc tree is faster than the strip tree because it does not need much

```
struct boxTree
{
    unsigned long int start, end;
    smallBbox * bbox;
    boxTree * leftchild;
    boxTree * rightchild;
};
```

Figure 4.10: Data structure of arc tree

processing time to choose the splitting point. The polyline is divided by the median point until only two points are left so that there is no approximation error.

Algorithm 3 buildArcTree()

```
if there are only two points then
   Finish building the arc tree
else
   calculate_MBR(box,start,end);
   division ← p⌈n/2⌉
   buildArcTree(start, division)
   buildArcTree(division, end)
end if
```

Figure 4.11 shows the example of an arc tree which is applied to real data.

4.4 Smallest Bounding Area Tree

4.4.1 Smallest Bounding Area Tree definition

Two well-known hierarchical representations, strip tree and arc tree, are mainly differentiated by how to choose the splitting point for building a next level of the tree. New idea was from here: how the tree can be more efficient by different splitting points? If the decomposition of the polyline is optimized, will the tree also be optimized? More optimized decomposition means that a bounding container of a tree structure approximates the polyline more precisely, therefore, it does not have much vacant space. Figure 4.12 shows two cases with different splitting points.

There are same polyline and line segment l in both examples in figure 4.12, but they have differently decomposed sub-polylines. When the operation to find the

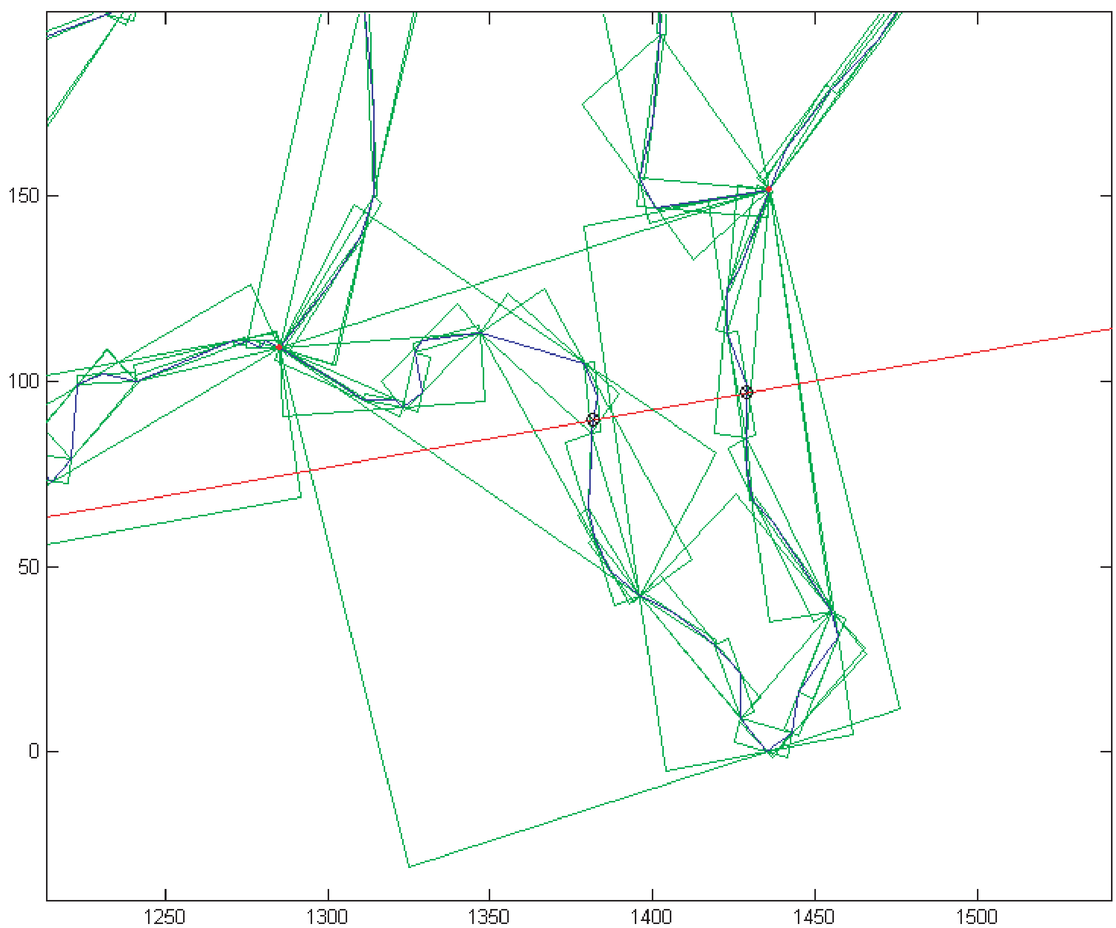


Figure 4.11: Arc tree with minimal bounding rectangle and finding intersections with random line segments

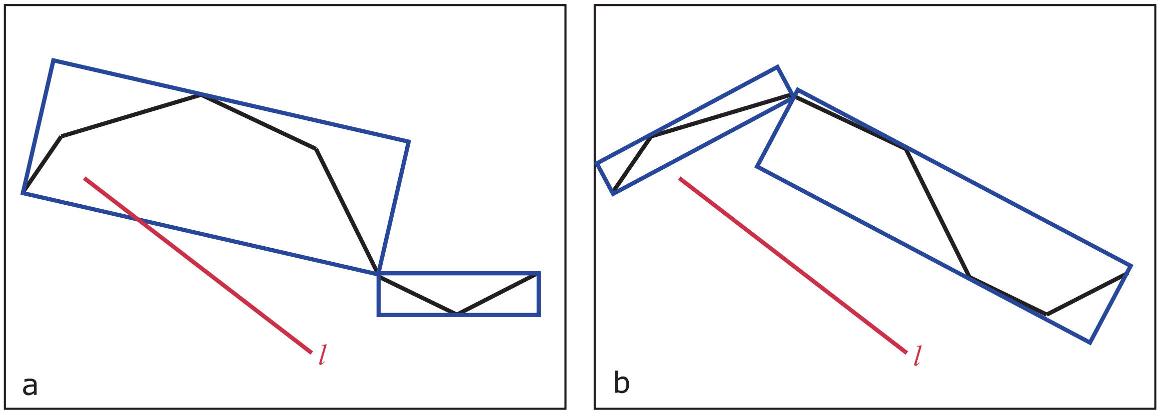


Figure 4.12: Comparing two trees by different splitting points

intersection between the polyline and a line segment l is executed, more processes are needed for the case in figure 4.12a. This is because the bounding container is intersecting with l, although l is actually not intersecting the polyline. Intersection is *possible* in this case. However, the case in figure 4.12b is *null*, which means that there is no intersection clearly. Therefore, we can know whether there is intersection or not faster so that we do not need extra operations.

More optimized decomposition can be achieved when the area of bounding containers is the smallest so that there is less vacant space. You can easily see that the area of bounding containers in figure 4.12b is smaller than in figure 4.12a. Thus, when we decide the splitting point in the process for building the tree, all possible points between two end points are checked, and then the one which has the smallest area of bounding containers will be a splitting point.

4.4.2 Implementation of Smallest Bounding Area Tree

There are two approaches: by greedy algorithm and by dynamic programming.

Greedy Algorithm

It is easy to understand by greedy approach. Splitting point is the point which makes the sum of divided bounding areas minimum in each level of resolution (see Algorithm 6). In each level of the tree, all points between starting and ending points are checked: if the curve is divided by each point, how big is the sum of areas of minimal bounding rectangles of sub-curves? Then choose the one which makes the sum of areas smallest (See Fig. 4.13). Therefore, we can make a cost function C with S which is the function calculating the area of a minimal bounding rectangle of a sub-curve with starting and ending points.

$$C(i,k) = \min_j \{S(i,j) + S(j,k)\}$$

We can calculate all values of function S between all points and make a matrix. It takes $O(n^2)$ time and space, and it takes $O(nlogn)$ for calculating a minimal bounding rectangle. Hence it takes $O(n^3 logn)$ for the matrix. In addition, $O(logn)$ is necessary for building a tree structure.

Algorithm 4 buildGreedyTree(start,end,tree)

```
calculate_MBR(box,start,end);
for k = start+1 TO end do
   S1 ← calculate_MBR(box1,start,k);
   S2 ← calculate_MBR(box2,k,end);
   if (S1 + S2 ≤ minArea) then
      minArea ← S1 + S2;
      division ← k;
   end if
end for
left ← initiate_new_node();
right ← initiate_new_node();
tree.bbox ← box;
tree.leftchild ← left;
tree.rightchild ← right;
if there are more than two points then
   buildGreedyTree(start,division,left);
end if
if there are more than two points then
   buildGreedyTree(division,end,right);
end if
```

Dynamic Programming

The process of building smallest bounding area (SBA) tree by dynamic programming has two steps. First, calculate the area of minimal bounding rectangles of all possible parts of an arc and make a matrix of smallest bounding area by dynamic programming (see Algorithm 4). Then, by using this matrix, find an optimal splitting point and build SBA tree (see Algorithm 5). Figure 4.13 shows the example of the process.

We have a cost function C, which is the area of all minimal bounding rectangles at the tree constructed for a piece of P from a vertex i to vertex k.

$$C^r(i,k) = \min_j \{C^{r-1}(i,j) + C^{r-1}(j,k)\} \text{ where } r \text{ is the depth of the tree}$$

For a leaf node in level 1: $(k-i) \leq 3$, C is calculated by a function S which is the sum of areas of minimal bounding rectangles.

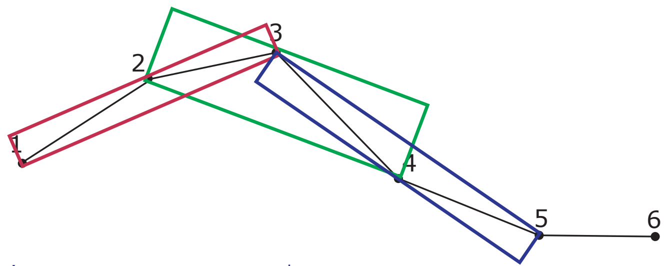
$$C^1(i,k) = \min_j \{S(i,j) + S(j,k)\}$$

If all possible points are checked and each area of minimal bounding rectangles is calculated, then processing time is not short. Time complexity of calculating a minimal bounding rectangle is $O(nlogn)$, thus, time complexity for making S matrix of smallest bounding area is $O(n^3 logn)$. It takes additionally $O(n^2)$ time and space for C matrix.

In this paper, the focus is on how different tree structures work efficiently, not on how fast tree structures can be built. This is because we can use the tree structure many times after building it once.

Algorithm 5 calcBoxArea(r,n1,n2)

```
for n1 = 1 TO N do
  for n2 = 1 TO N do
    calculate_MBR(box,n1,n2);
  end for
end for
```



Greedy Algorithm

M : area of MBR

	1	2	3	4	5	6
1			4.5	145.5	230	279.1
2				64.5	101	127.1
3					31	80
4						21
5						
6						

P : splitting point

	1	2	3	4	5	6
1			2	3	3	3
2				3	3	4
3					4	4
4						5
5						
6						

Dynamic Programming

A : smallest bounding area

	1	2	3	4	5	6
1			4.5	150	265.5	384.6
2				64.5	132	212.6
3					31	101
4						21
5						
6						

P : splitting point

	1	2	3	4	5	6
1			2	3	3	3
2				3	3	4
3					4	4
4						5
5						
6						

area[1-3] < area[2-4]

P[1][4] = 3

area[2-4] > area[3-5]

P[2][5] = 3

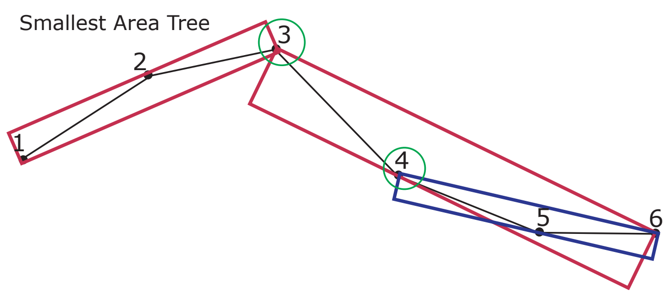


Figure 4.13: Calculating a matrix of splitting points and building SBA tree by greedy algorithm and dynamic programming

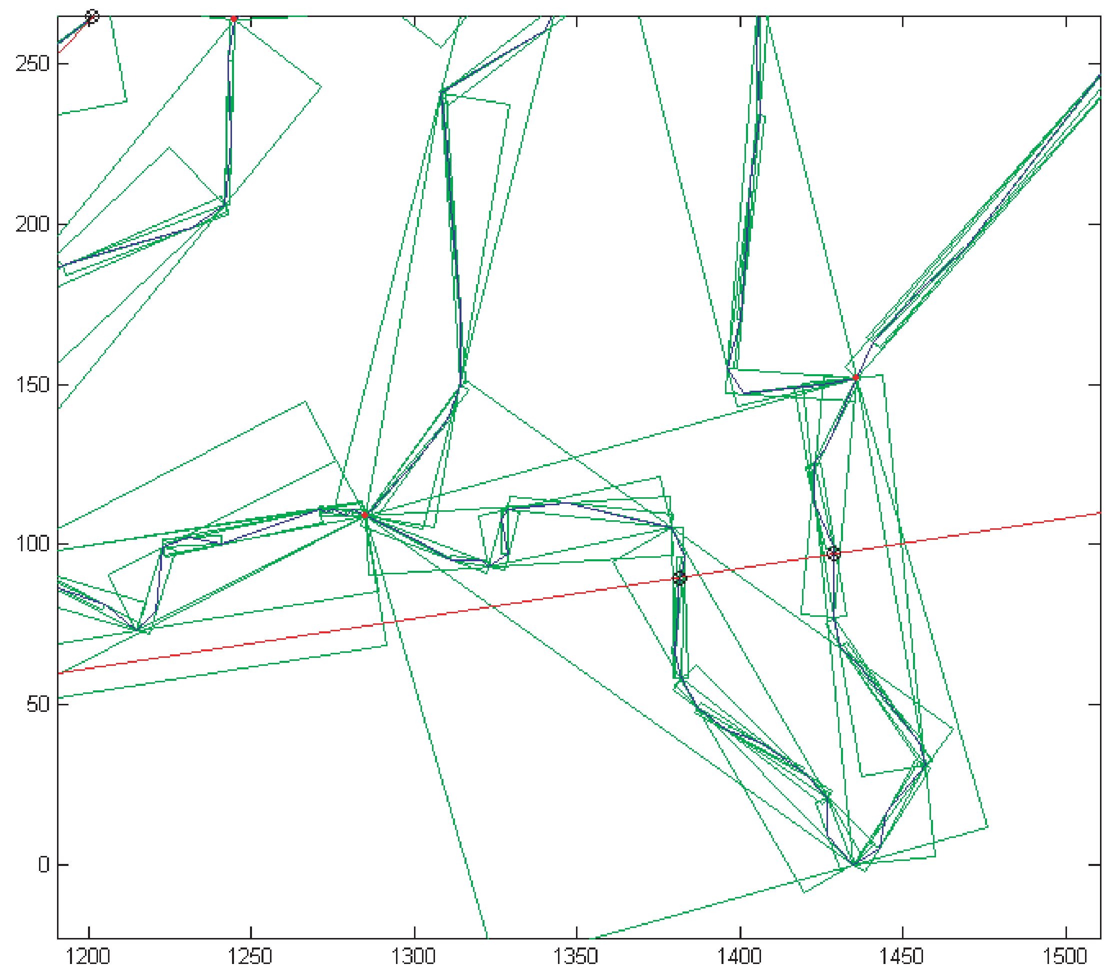


Figure 4.14: SBA tree with minimal bounding rectangle and finding intersections with random line segments

Algorithm 6 area = constructTree(start,end)

```
S_0 ← calcBoxArea(start,end);
r ← ⌈log_2(end − start + 1)⌉;
if r > 1 then
  for j=start+1 TO end-1 do
    S_1 ← constructTree(start,j);
    S_2 ← constructTree(j,end);
    if S_0 + S_1 + S_2 < S_min then
      S_min ← S_0 + S_1 + S_2;
      division ← j;
    end if
  end for
  return S_min;
else
  return S_0;
end if
```

Chapter 5

Applied Areas

This chapter describes areas hierarchical representation of arcs can be applied to. How the hierarchical data modelling can help to solve the problems is explained.

5.1 Using a Hierarchical Structure for Reporting Intersections

A hierarchical structure can be applied in many areas of computational geometry. Line segment intersection (LSI) is one of most important and basic problems, because computational problems such as polygon intersection or point inclusion can be based on LSI problem. Algorithms for LSI are reviewed and compared to an algorithm with a hierarchical structure.

5.1.1 Line Segment Intersection(LSI)

Line segment intersection problem is defined as follows:

- A set $S = s_1, s_2, \ldots s_n$ of n line segments(see Fig. 4.15)
- Find all pairs $(s_i, s_j) \in S^2$ such that $i \neq j$ and $s_i \cap s_j \neq \phi$

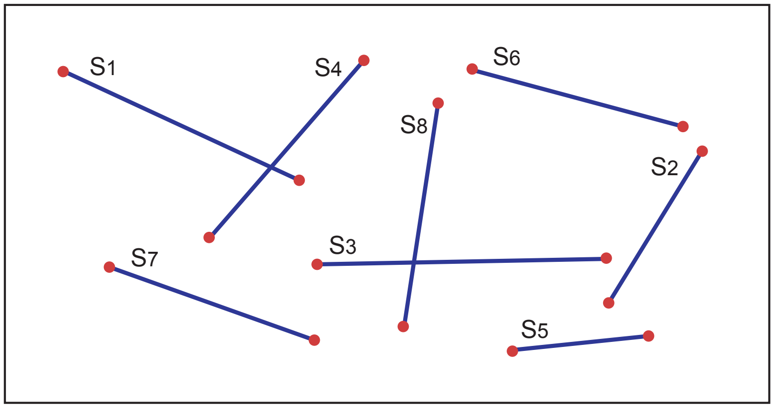


Figure 5.1: A set S of n line segments

There are different algorithms for reporting all intersections between line segments. Each has different time complexity. Brute force algorithm takes $O(n^2)$, simply finds intersections between all possible groups of two line segments. LSI with plane-sweep technique [PS85, dBvKOS00] can be solved in $O(nlogn)$.

LSI problem can be applied to find intersections between arcs and a line segment, because arcs consist of several line segments. One simple method is first sorting all line segments of arcs then finding intersections. This takes $O(nlogn)$ for sorting (in case of merge sorting) plus $O(logn)$ for searching intersections.

5.1.2 Hierarchical Structure and LSI

A hierarchical structure can be used for finding intersections between arcs and a line segment. This takes $O(Mlogk)$ such that M is a number of arcs and k is a number of line segments of each arc. Therefore, using a hierarchical structure may be faster than using LSI algorithm or quite same - it depends on M and k.

Using a hierarchical structure can have benefits($+$) and losses($-$) against LSI as follows:

\+ More understandable and more heuristic

\+ Each arc, not individual line segment, has topological information - saving space.

– More computationally complicated

– It takes time and needs space to construct a hierarchical structure.

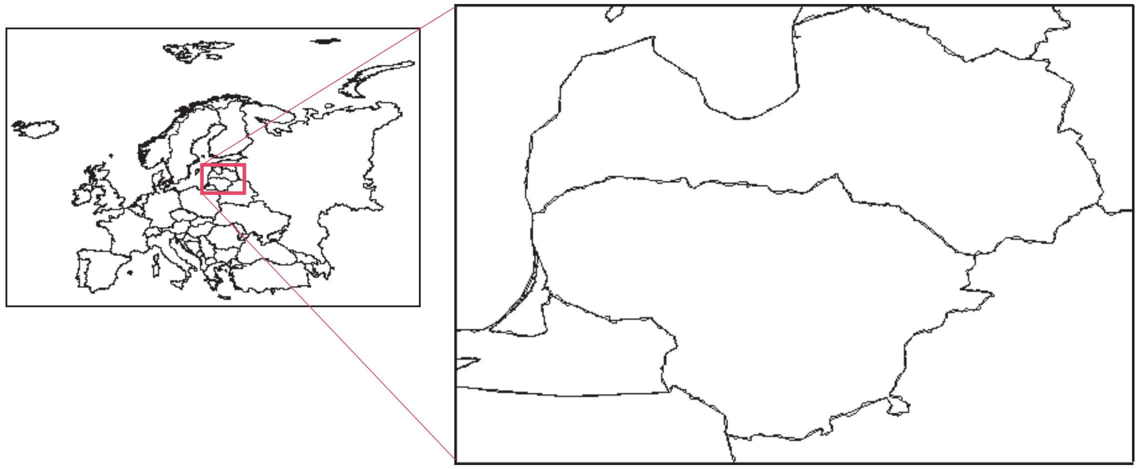


Figure 5.2: Not big difference between original and simplified maps at small scale

5.2 Polygonal Approximation

5.2.1 Definition of Polygonal Approximation

Polygonal approximation is a process of elimination of points which produce the least errors. This process is necessary because a size of data can be reduced much so that data retrieval and management can be faster. Also, it takes less time to show the map data. At small scale map, not many points are necessary because visual difference is not noticeable with human bare eyes (See Fig. 5.2). Vector processing such as point inclusion or polygon intersection can be faster because a simplified polygon has less boundaries to be checked [Tay].

Line segment L in 2-dimensional space is represented by ordered point set P which has N points: $P = p_1, \ldots, p_N = (x_1, y_1), \ldots, (x_N, y_N)$. After polygonal approximation process, L has a new ordered point set Q which is represented by M points: $Q = q_1, \ldots, q_M$. The point set of Q is a subset of P and $M \leq N$. The end points of Q are same with the end points of P: $q_1 = p_1, q_M = p_N$ [Kol03].

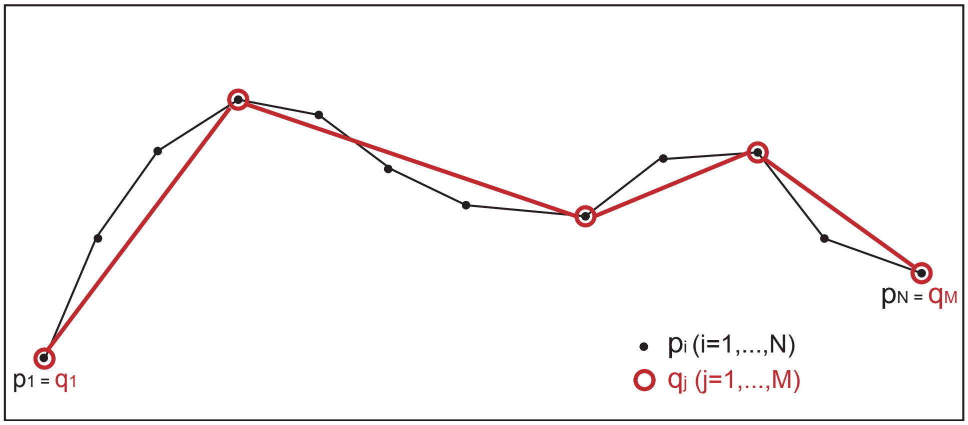


Figure 5.3: P and Q sets

5.2.2 Algorithms

Heuristic Algorithms

Many algorithms for polygonal approximation are developed with different techniques. Heuristic algorithms are not always optimal but the process is easy to understand and can be done quite fast. Heuristic algorithms can be grouped by two strategies, *decimation and refinement* [KDE05].

Most of algorithms are decimation methods in which removable points by a given error tolerance are chosen and removed. This process starts with all points describing a line, and the result is simplified line with less points. On contrast to decimation algorithms, Douglas-Peucker algorithm(1973) is by a refinement strategy. It starts with two endpoints of a line, and points are getting inserted according to a given error criterion.

Polygonal Boundary Reduction is a simple decimation algorithm proposed by Leu and Chen [GL98]. This algorithm considers boundary arcs of two and three edges. It calculates the maximum distance between the arc and the directed line of two endpoints. If the distance is less than a given threshold, then it replaces the arc to the directed line of two endpoints (See Fig. 5.4).

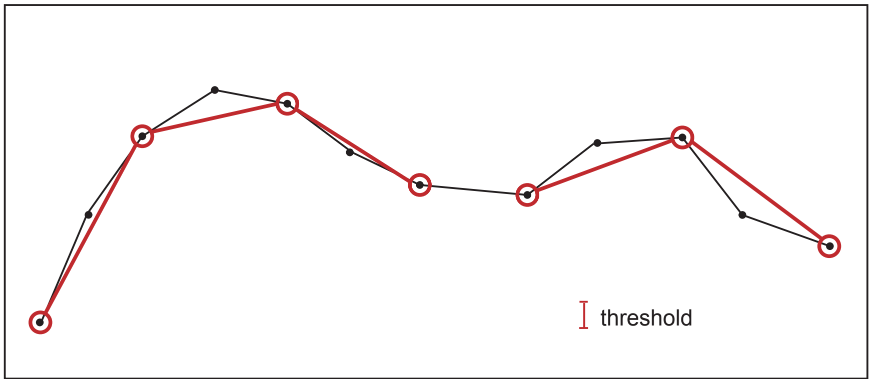


Figure 5.4: Polygonal boundary reduction

Optimal Approximations

There are two different types by error bounds(Imai and Iri, 1998).

- min-ε : minimizing the approximation error for a certain number of points
- min-$\sharp$: minimizing the number of points for a given error bound ε

5.2.3 Topologically Consistent Simplification Using Hierarchical Structure

Polygonal approximation algorithms do not always guarantee topological consistency. There may be some inconsistencies such as an intersection with neighbor objects or a self-intersection [EM01]. Figure 5.5 and 5.6 shows the examples of inconsistency of topology.

Self-intersection can occur in an approximation of severely bent curves [HK01]. In figure 5.5, self-intersection is generated by using Douglas-Peucker algorithm [JSG99]. These intersections make wrong topological information. Therefore, they should be found before or after approximation and be fixed.

For an efficient process to find intersections, a hierarchical structure of curves described in chapter 4 is used. Checking all curves in the map for intersection with new simplified line segment is not efficient because it is obvious for curves far away from the corresponding line segment not to intersect each other. Irrelevant curves are excluded by checking an intersection with a bounding container which bounds

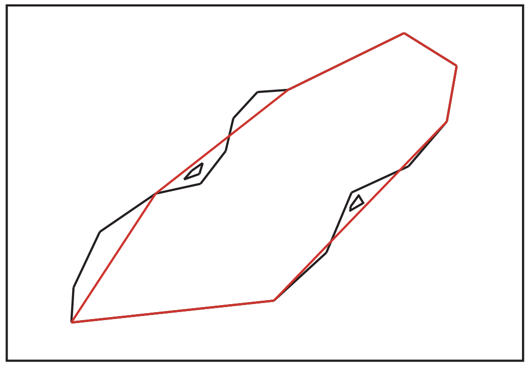


Figure 5.5: Islands disappeared after polygonal approximation

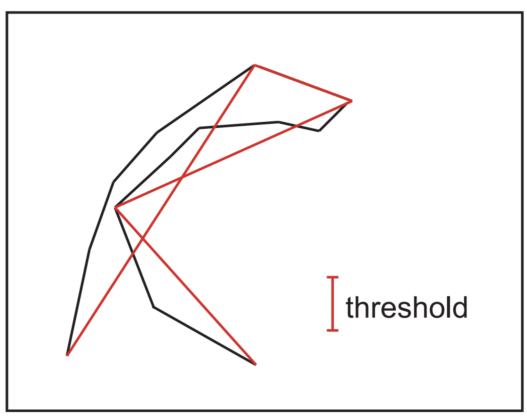


Figure 5.6: Self-intersection after polygonal approximation by Douglas-Peucker algorithm

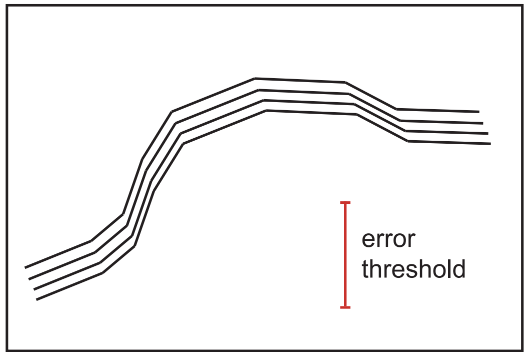


Figure 5.7: Nested curves to which simplification by defining safe sets can not be applied

the whole curve. Hence, it is computationally faster than without the hierarchical structure.

There are two methods for fixing errors. First method is fixing errors after approximation process. As an example, Estkowski and Mitchell proposed Simple Detours (SD) heuristic idea in 2001 [EM01]. First, a standard polygonal approximation is applied, then intersections are found. One of intersecting segments is declared as a *detour segment*, and *detour graph* $G(s)$ is constructed. In $G(s)$, two points can be joined if and only if the corresponding line segment is error-tolerant and does not intersect with another line segment.

Second method is applying approximation process only when a new simplified line segment does not make any intersections with neighbor objects, that is, when there are no topological errors. There is an actual work of preventing topological changes by defining "safe sets" using a Vornoi diagram [MS00]. This method is working better for maintaining an original shape than first method because a simplification can occur only in a safety zone. However, this safety can be a weak point in some cases. Figure 5.7 shows the example of nested curved lines and an error bound for approximation [EM01]. In this case, approximation may not be applied.

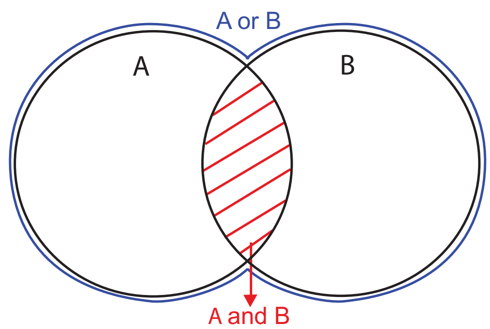


Figure 5.8: Intersection and union of sets A and B

5.3 Windowing and Clipping

5.3.1 Polygon Overlay

Map overlay operations are often necessary in GIS. For example, when making land use decision, there can be many layers of geographical data such as environmental or social factors. Topological map overlay creates new objects and attribute relations by overlaying objects from many input map layers. A polygon can be thought of as representing a set. When two sets (polygons) A and B are overlaid, we can have set concepts *intersection* and *union* (see Fig. 5.8). There are 16 possible combinations of boolean expression, but *intersection* is of most interest in polygon overlay operations.

In following sections, we will look through windowing and clipping which are intersection between the window rectangle and polygon objects of the map data.

5.3.2 Windowing

There is a given rectangle R, which is the window, and whether a shape S intersects the rectangle R or not is tested [RSV02]. In a simple method, we can basically look through all segments of all arcs and find intersections with the rectangle R. If the arc is intersecting R or inside R, then the arc and the polygon which has the arc is visualized. This can have many redundant operations, for example, when if the window rectangle R is very small and the map is big so that there are many polygons far away from the R. Therefore, if we use hierarchical structure for arcs, we can reduce these operations. If some arcs are inside the R or intersecting the R, then polygons related to those arcs intersect the rectangle R. From the information

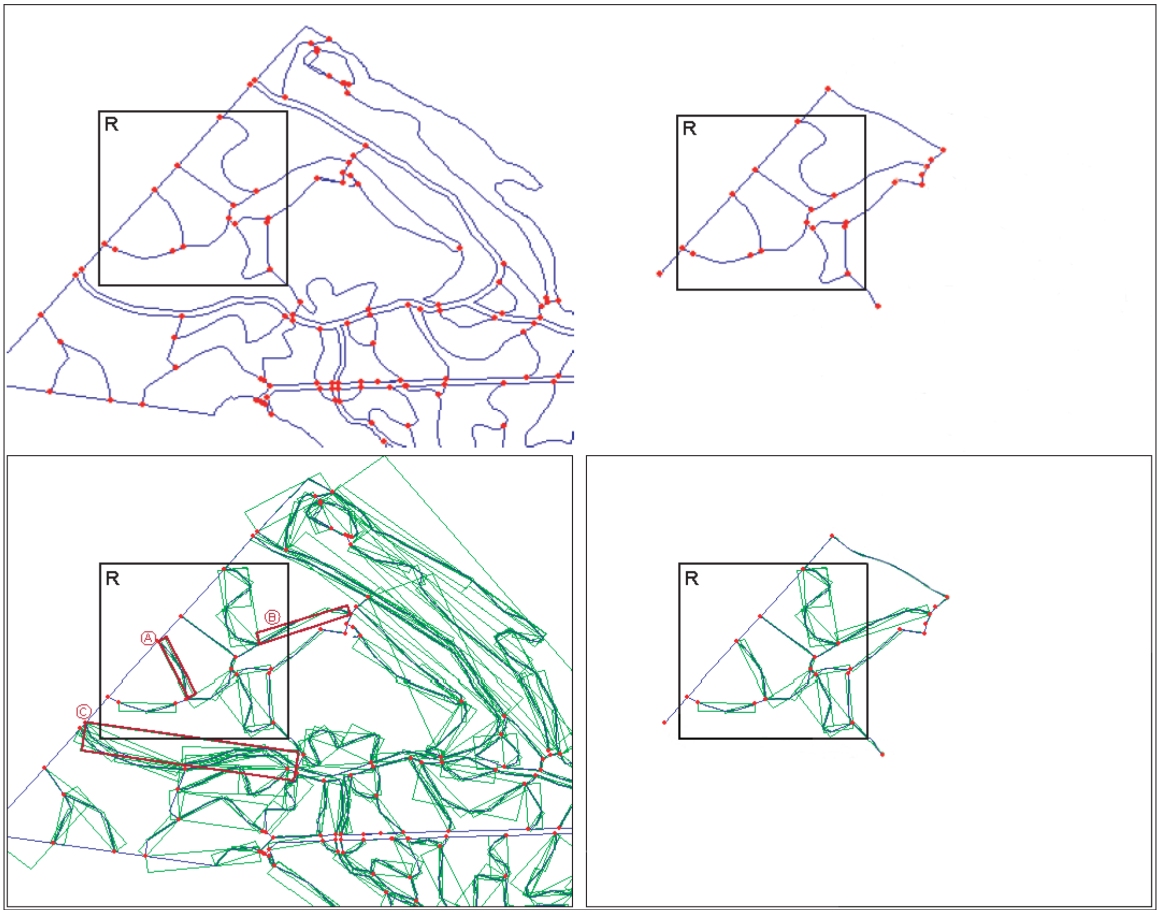


Figure 5.9: Example of windowing

of left and right polygons of the arc, we know which polygons are related.

In figure 5.9, there are three possible cases. In case A, the bounding box of the arc is included in the R, so polygons which has this arc are intersecting the R. In case B, the arc and bounding box of the arc are intersecting the R, so polygons related with this arc are intersecting. In case C, the arc is not intersecting the R, but the bounding box of the arc is intersecting the R. In this case, more detailed levels of the tree structure of the arc are checked and whether the arc is intersecting the R or not is confirmed.

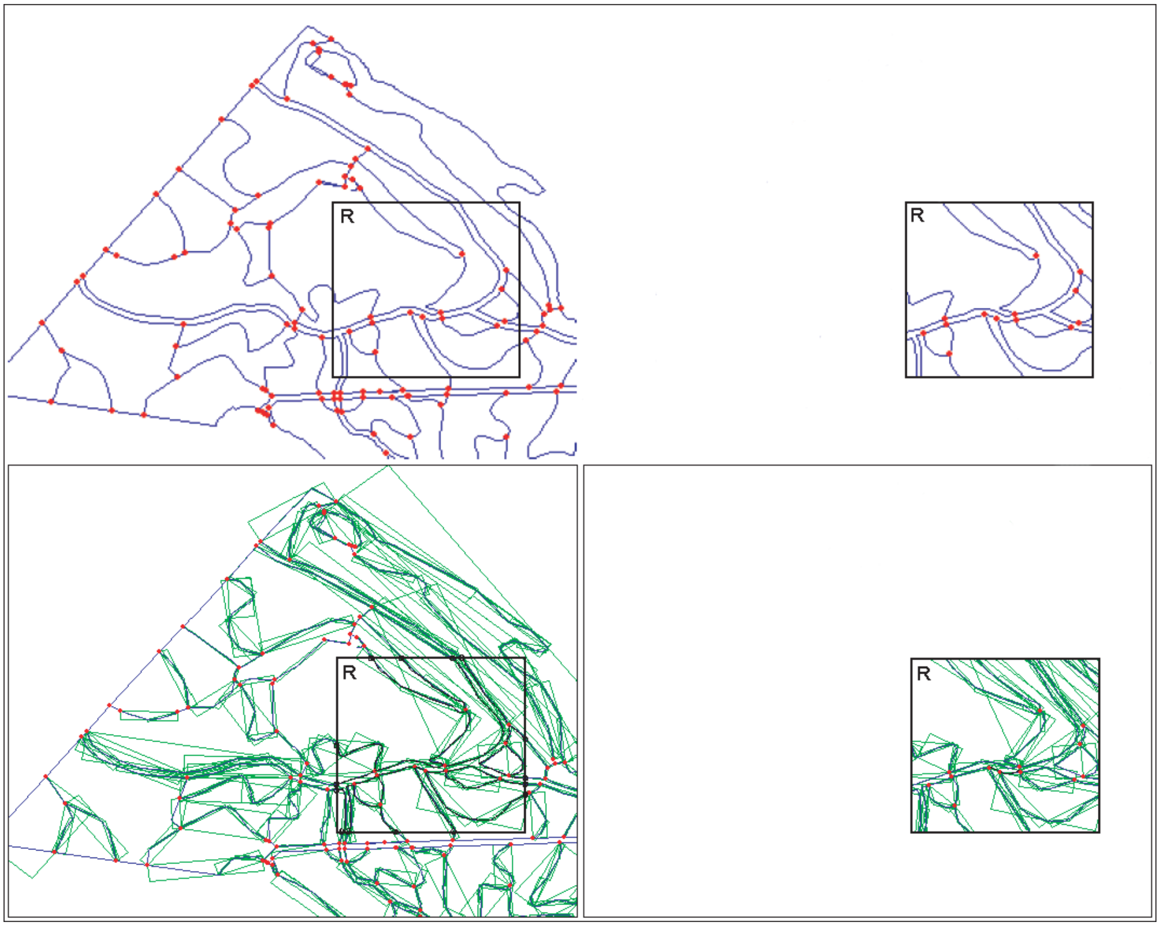


Figure 5.10: Example of clipping

5.3.3 Clipping

Clipping is similar with windowing, however it needs more complicated operations. There is a given rectangle R, and we clip the polygons which are inside the rectangle R. After clipping, new objects are created, because the segment of arcs which is intersecting the rectangle R will be cut by the edge of the R.

The usage of hierarchical structure of arcs is basically same with windowing. If the bounding container of the arc is inside the R, then the whole arc is included. If the bounding container and the arc are intersecting the R, then we should find the intersecting point between the edge of the rectangle R and the arc. By using this point, the line segment intersecting the edge of R can be cut (See Fig. 5.10).

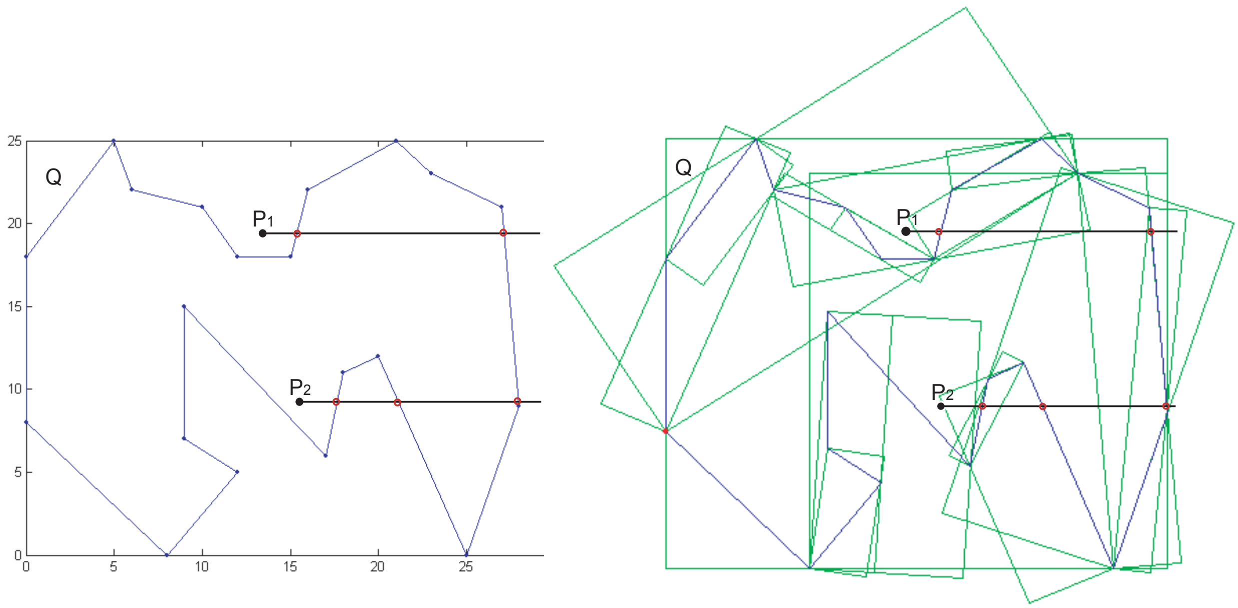


Figure 5.11: Inclusion of the point P in the polygon Q

5.4 Point Inclusion

Point inclusion is one of basic operations in GIS. Hierarchical structure of an arc also can be useful to check the point inclusion. If we want to know that the point P is inside the polygon Q, we have to find out how many times a ray from the point P intersects edges of the polygon Q (See Fig. 5.11). When finding intersections, hierarchical structure can make it more efficient. If the ray from the point P intersects times of an even number, P is outside Q. If the ray intersects times of an odd number, P is inside Q.

Chapter 6

Experiments

Smallest Bounding Area Tree which is proposed in this thesis have implemented and tested with real data for its efficiency and effectiveness. These tests are done with one 1,400MHz Intel Pentium M processor and 512MB of memory.

Test data are a digital map which has 1,941 points and a map which has 10,925 points (See Fig. 6.1).

Tests are for checking how hierarchical structures make intersection checking efficient. Therefore, map data is tested with hierarchical structures or without, and how much time was taken in each case is calculated and compared. Figure 6.2 shows the example of intersections between random line segments and Map1. The map is transformed to a map with topological structure - Node, Arc, and Polygon before the test.

6.1 Comparison 1: With Different Bounding Containers and Without

First experiment is finding intersections between random line segments and the map with a hierarchical structure and without. For the test, 1000 line segments for Map1 and 500 line segments for Map2 are randomly created. Smallest Bounding Area Tree (SBA Tree) is used as a hierarchical structure and orthogonal box and

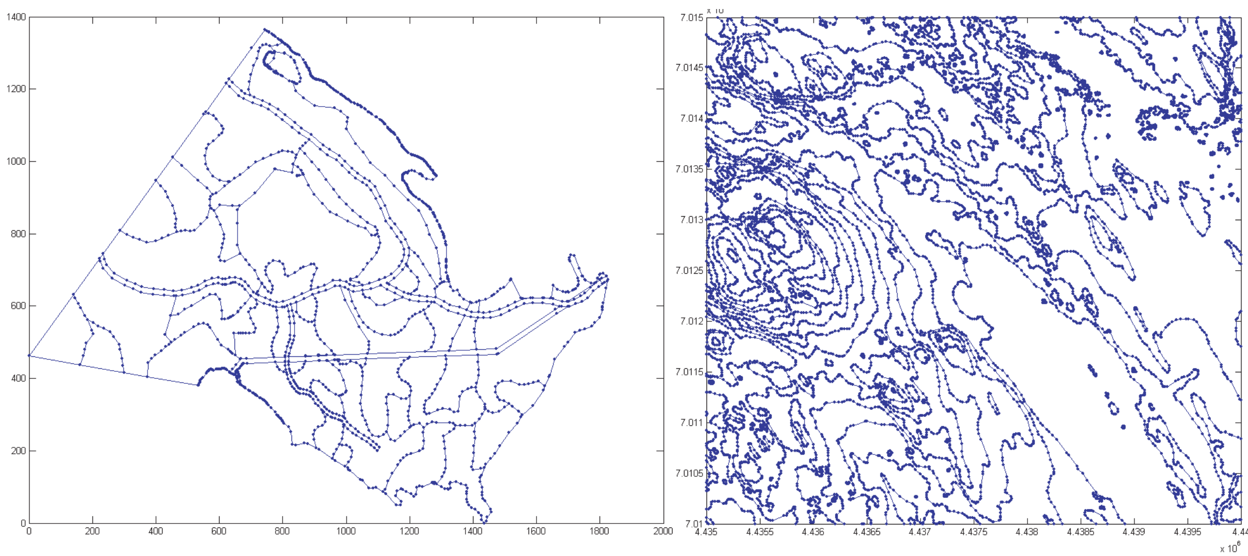


Figure 6.1: Map1 and Map2 for testing

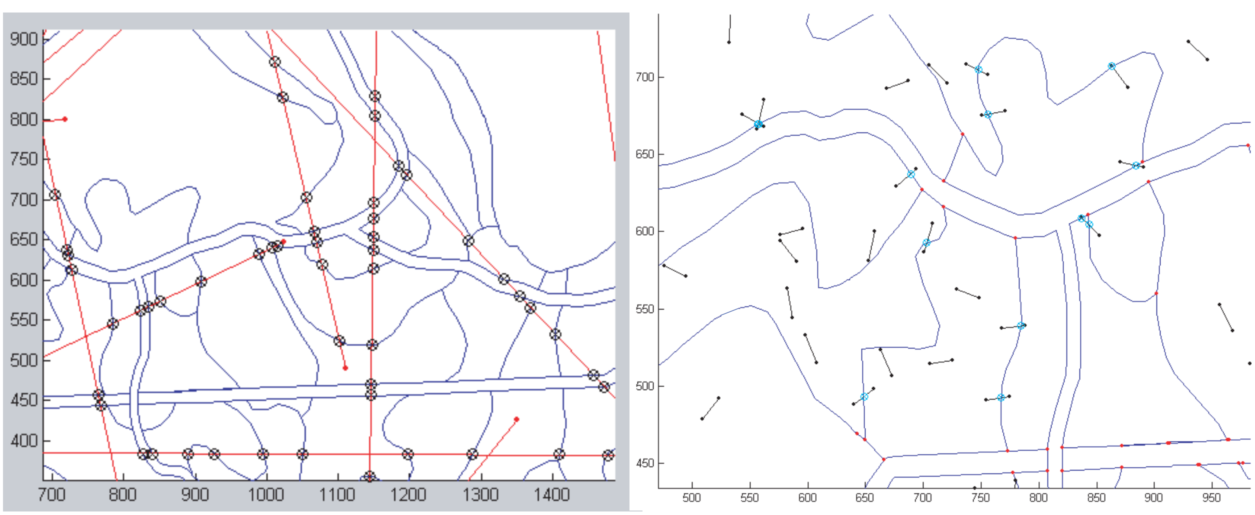


Figure 6.2: Map1, long and short random line segments, and intersections

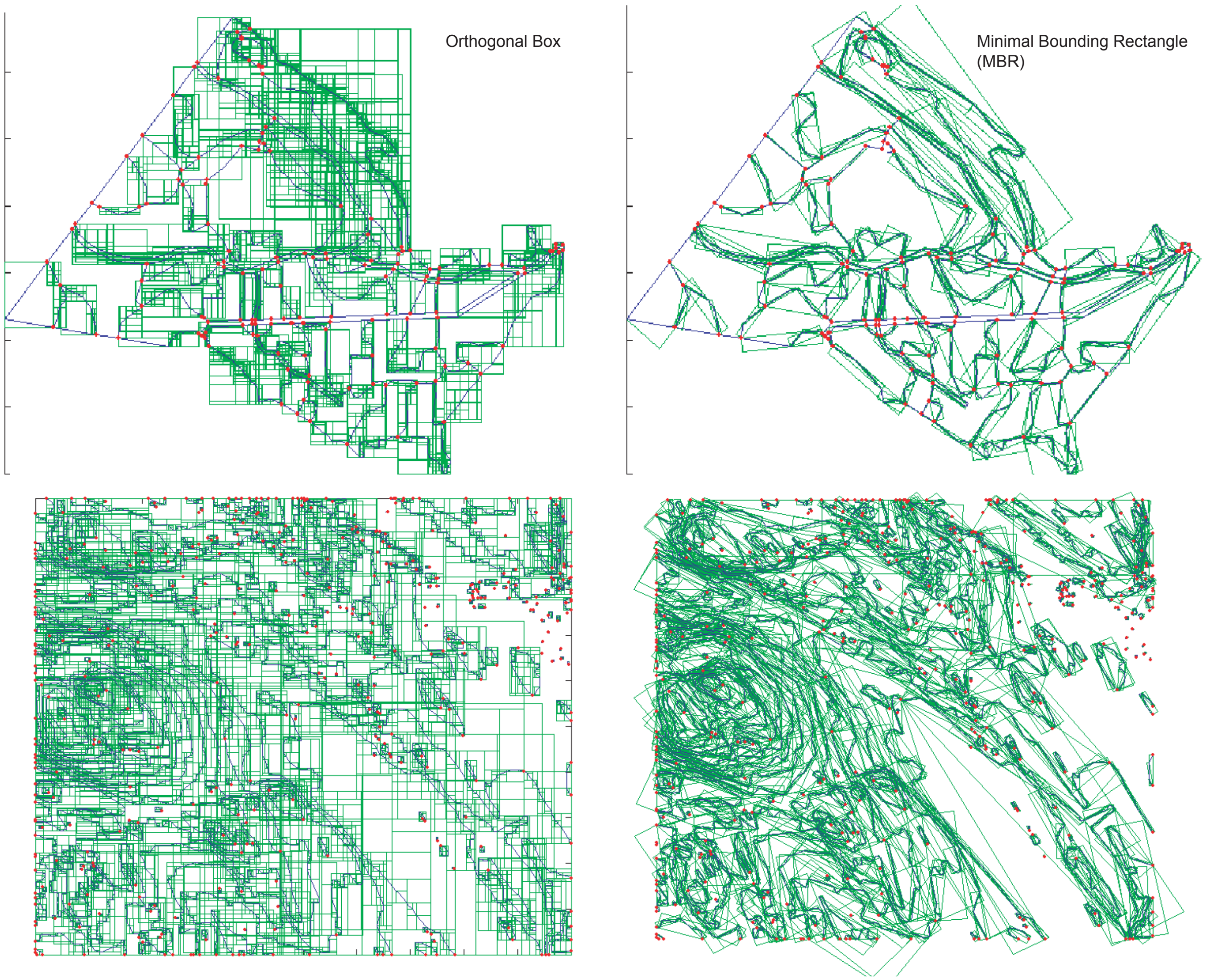


Figure 6.3: Map1 and Map2 with orthogonal boxes and with minimal bounding rectangles

minimal bounding rectangle (MBR) are used as a bounding container (See Fig. 6.3).

Without hierarchical structures, all segments of all arcs should be checked for each line segment. If the line segment is far away from some polygons, then it is not necessary to do a checking process with them, hence, it is not efficient.

Using MBR as a bounding container is slightly faster than using orthogonal boxes in average time, but there is not big difference between them. Checking intersections with an orthogonal box is faster than with a MBR. Therefore, even though MBR approximates more precisely than orthogonal box, using orthogonal boxes can be faster in some cases (See Table 6.1 and 6.2).

6.2 Comparison 2: Different Hierarchical Structures

Second experiment is comparing efficiency of three different hierarchical structures - Arc Tree, Strip Tree, and Smallest Bounding Area Tree (SBA tree). Figure 6.5 shows the process of building each tree structure for the map.

You can see that boxes by Arc tree are bigger than Strip and SBA tree. Boxes by Strip tree look also well-behaving, however, if a line is complicated and distorted, boxes by SBA tree is more efficient. Figure 6.4 shows an example of a complicated line and boxes by Strip and SBA trees.

Table 6.3 and 6.4 show how much time is taken to find intersections between random lines and all objects of the map using Strip, Arc, and SBA trees. SBA tree works better than Strip and Arc trees, not always but generally according to the tests. Arc tree works generally worst among three of them, because the area of its bounding boxes is bigger so that its approximation of objects is not better than others.

We can decide which tree we can use by considering what kind of map is. Also, how many times the tree is used can be considered. If arcs of the map are simple, and the tree structure is not used much, then we can use an arc tree because building time is short. If arcs of the map are complicated, and the tree structure is used many times again, then strip tree and SBA tree are better than arc tree, though it takes more time to build them.

Map (1,941)	Without Tree	With SBA Tree + Orthogonal Box	With SBA Tree (DP) + MBR
	0.620	0.421	0.430
	0.591	0.441	0.341
	0.671	0.401	0.311
	0.632	0.260	0.351
	0.600	0.330	0.371
	0.571	0.341	0.350
	0.570	0.401	0.340
	0.625	0.300	0.391
	0.561	0.441	0.370
	0.630	0.190	0.400
	0.611	0.341	0.421
	0.586	0.320	0.300
	0.580	0.360	0.271
	0.627	0.421	0.330
	0.590	0.360	0.231
	0.600	0.291	0.360
	0.592	0.351	0.391
	0.610	0.431	0.361
	0.561	0.331	0.351
	0.630	0.340	0.351
Average time	0.6029	0.3536	0.3511

Table 6.1: Running time (seconds) for finding intersections between 1,000 random line segments and all features of Map1 (1,941 points). Tests are done 20 times and average time is calculated.

Map (10,925)	Without Tree	With SBA Tree + Orthogonal Box	With SBA Tree (DP) + MBR
	1.552	0.441	0.461
	1.563	0.420	0.462
	1.532	0.441	0.391
	1.532	0.431	0.381
	1.532	0.440	0.440
	1.543	0.431	0.460
	1.532	0.421	0.440
	1.512	0.420	0.441
	1.512	0.421	0.501
	1.512	0.420	0.481
	1.502	0.411	0.440
	1.513	0.421	0.382
	1.502	0.440	0.450
	1.522	0.421	0.411
	1.512	0.430	0.371
	1.512	0.431	0.450
	1.502	0.441	0.530
	1.533	0.440	0.441
	1.512	0.411	0.431
	1.502	0.421	0.412
Average time	1.5217	0.42765	0.4388

Table 6.2: Running time (seconds) for finding intersections between 500 random line segments and all features of Map2 (10,925 points). Tests are done 20 times and average time is calculated.

Map (1,941)	With Arc Tree	With Strip Tree	With SBA Tree DP	With SBA Tree Greedy Alg.
	0.441	0.390	0.310	0.401
	0.451	0.342	0.420	0.330
	0.440	0.381	0.410	0.331
	0.410	0.400	0.321	0.411
	0.350	0.511	0.341	0.380
	0.432	0.330	0.401	0.380
	0.420	0.331	0.361	0.430
	0.360	0.402	0.390	0.390
	0.502	0.360	0.390	0.280
	0.380	0.341	0.381	0.420
	0.431	0.320	0.412	0.380
	0.300	0.401	0.390	0.451
	0.330	0.420	0.421	0.371
	0.340	0.440	0.361	0.401
	0.410	0.421	0.310	0.381
	0.350	0.421	0.442	0.330
	0.420	0.351	0.441	0.320
	0.370	0.401	0.451	0.350
	0.371	0.420	0.350	0.391
	0.380	0.431	0.270	0.452
Average time	0.3944	0.3907	0.37865	0.379

Table 6.3: Running time (seconds) for finding intersections with three different tree structures between 1,000 random line segments and all features of Map1 (1,941 points). Tests are done 20 times and average time is calculated.

Map (10,925)	With Arc Tree	With Strip Tree	With SBA Tree DP	With SBA Tree Greedy Alg.
	0.540	0.471	0.381	0.441
	0.380	0.490	0.431	0.531
	0.460	0.511	0.412	0.440
	0.330	0.442	0.530	0.511
	0.440	0.431	0.440	0.521
	0.471	0.491	0.481	0.400
	0.520	0.472	0.490	0.320
	0.430	0.583	0.430	0.390
	0.562	0.460	0.420	0.401
	0.421	0.450	0.401	0.540
	0.450	0.410	0.512	0.441
	0.481	0.420	0.380	0.552
	0.480	0.451	0.411	0.490
	0.441	0.380	0.511	0.521
	0.421	0.581	0.380	0.471
	0.581	0.511	0.440	0.290
	0.431	0.430	0.390	0.572
	0.480	0.490	0.361	0.491
	0.481	0.542	0.400	0.370
	0.440	0.371	0.502	0.500
Average time	0.4620	0.46935	0.43515	0.45965

Table 6.4: Running time (seconds) for finding intersections with three different tree structures between 500 random line segments and all features of Map2 (10,925 points). Tests are done 20 times and average time is calculated.

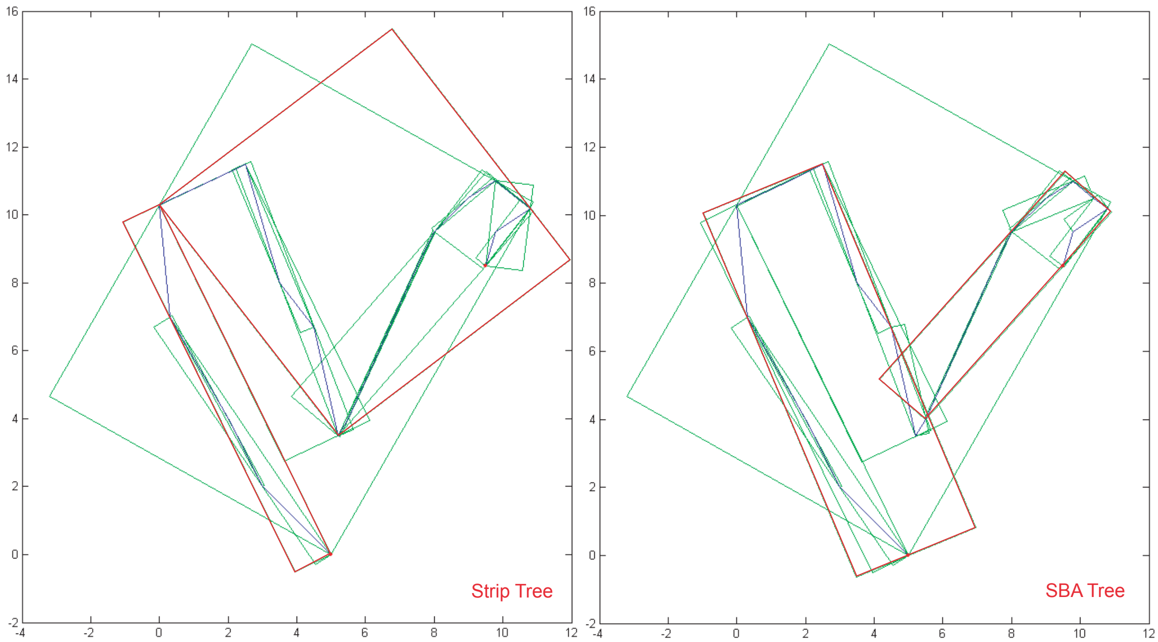


Figure 6.4: Comparing strip tree and SBA tree

Starting and ending points of random lines used for the tests are chosen randomly so that the length of most lines are long. Hence there are many intersections between the line and map objects. One more test with short random lines is processed, because there are also operations for intersections with mostly short lines. For example, for polygonal approximation, most of operations may be with short lines. The part of approximated lines is short, because new approximated line segment is checked for intersections not with other approximated line segments but with other original line segments. This means that the approximation is more strict and not much shape-changed (See Fig.6.6).

Approximated line segment Q_1 is illegal if we find intersections between Q_1 and other polyline P_2, however, Q_1 approximation is possible if we find intersections between Q_1 and Q_2, new approximated line segment of the part of P_2. Table 6.5 is the result of finding intersections between 1000 random short lines and Map1.

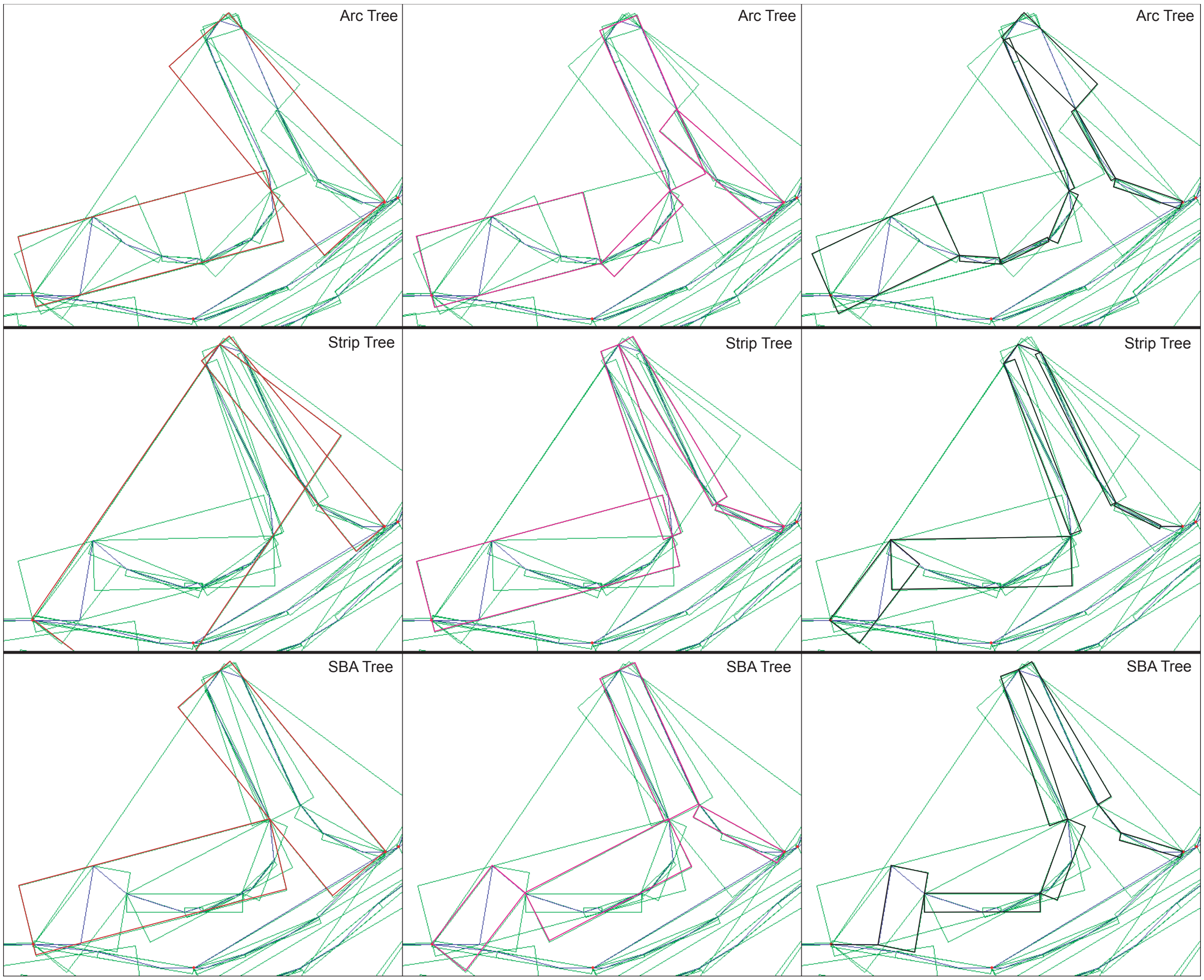


Figure 6.5: Building arc tree, strip tree, SBA tree for Map1

Map (1,941)	With Arc Tree	With Strip Tree	With SBA Tree DP	With SBA Tree Greedy Alg.
	0.300	0.380	0.290	0.332
	0.330	0.300	0.361	0.310
	0.290	0.402	0.320	0.300
	0.380	0.311	0.350	0.271
	0.350	0.330	0.321	0.310
	0.350	0.330	0.351	0.300
	0.341	0.320	0.381	0.290
	0.320	0.341	0.321	0.330
	0.380	0.281	0.350	0.301
	0.280	0.371	0.340	0.331
	0.271	0.351	0.310	0.350
	0.310	0.391	0.350	0.251
	0.340	0.340	0.291	0.351
	0.271	0.371	0.320	0.360
	0.290	0.410	0.311	0.331
	0.331	0.320	0.310	0.310
	0.351	0.370	0.271	0.320
	0.290	0.351	0.310	0.341
	0.361	0.350	0.340	0.270
	0.320	0.332	0.330	0.340
Average time	0.3228	0.3476	0.3264	0.31495

Table 6.5: Running time (seconds) for finding intersections with three different tree structures between 1,000 random short line segments and all features of Map1 (1,941 points). Tests are done 20 times and average time is calculated.

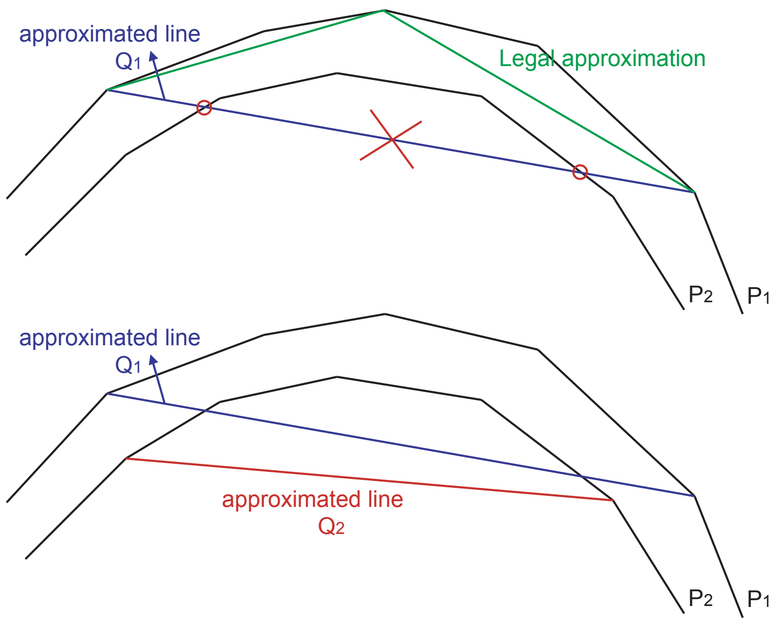


Figure 6.6: Illegal and legal approximations

Chapter 7

Conclusion and Future Work

In this paper, data modelling for a vector map is studied. Vector data model can be divided to non-topological and topological models. Spaghetti model is non-topological, and it is the simplest vector map type. The map with spaghetti model is transformed to a topological vector map which has the information of neighbors. The topological structure built in this paper has node, arc, and polygon objects. Arc is similar with a line object but it has left and right neighbors' information.

For more efficient representing arcs, hierarchical structures are in use. First, several bounding containers are explained, and minimal bounding rectangle (MBR) is described in detail and implemented using rotating calipers. With these bounding containers, strip and arc trees which are widely used are explained and implemented. Smallest bounding area (SBA) tree is newly suggested in this paper. This tree is built by the splitting point which is optimized by bounding area. Splitting point is the point which has the smallest bounding area. This is accomplished by greedy algorithm and by dynamic programming. The bounding area is optimized in current level by greedy approach, and the bounding area is optimized in whole levels of the tree by dynamic programming.

SBA tree makes finding intersections with random lines faster sometimes, but not always in experiments. Each tree structure has good and bad sides. It is fast to build an arc tree, because it does not have complicated calculation for deciding the splitting point. However, bounding area made by arc tree can not approximate the real object well in some cases. Strip tree works quite good, but if the arc is complicated and distorted much, approximation by strip tree can be not that good. SBA tree takes more time to be built than other trees, but it approximates the real object more tightly. These tree structures can be used in many operations for managing a vector map. Polygonal approximation is one of important operations for many reasons such as simpler visualization and faster transmission. When the map is approximated, topological information can be changed. Hence, we should avoid wrong topological changes and keep the original one. This can be done by approximating only if the topology is same, and fixing errors after approximation. For both cases, the most important and often used operation is finding intersections with other arcs or line segments. Therefore, hierarchical structures can be used for topologically consistent simplification. In addition, we can also apply the structures to windowing, clipping, and point inclusion test. For windowing and clipping, we can use the hierarchical structure when we find which arc is intersecting the rectangle R, then get the polygon information from the arc and find intersection between the rectangle R and the line segment of the arc. For point inclusion, we should find out how many times the ray from the point is intersecting the polygon. Using hierarchical structure also can help the process. More applied areas can be studied in the future.

Structures for hierarchical representation are focused on in this paper, so implementations of some parts are not efficient. For example, the algorithm for finding intersections between MBRs or between MBR and line segment is not efficient. Therefore, this can be improved more in the future. More various bounding containers can be implemented with the SBA tree, so we can decide which bounding container works better with the SBA tree. Also, if we find not perfectly optimized splitting point, then time for building the tree can be shorter. It may be achieved by combining optimal and heuristic algorithms. This issue also can be improved in the future.

Bibliography

[Apa] Apache software foundation, http://xml.apache.org/xerces-c/index.html. *Xerces-C++ Parser*, version 2.7.0 edition.

[Bal81] D. H. Ballard. Strip trees, a hierarchical representation for curves. *Communications of the ACM*, 24(5):310–321, 1981.

[Bez74] P.E. Bezier. Mathematical and practical possibilities of unisurf. *Computer-Aided Geometric Design*, pages 127–152, 1974.

[BV02] T. Bernardsen and A.A. Viak. *Geographic Information Systems: An Introduction*. Wiley, 2002.

[dBvKOS00] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzlopf. *Computational Geometry: Algorithms and Applications*. Springer, 2000.

[DeM05] M.N. DeMers. *Fundamentals of Geographic Information Systems*. Wiley, 2005.

[EM01] Regina Estkowski and Joseph S.B. Mitchell. Simplifying a polygonal subdivision while keeping it simple. *Proceedings of the seventeenth annual symposium on Computational geometry*, pages 40–49, 2001.

[GL98] Leu G. and Chen L. Polygonal approximation of 2-d shapes through boundary merging. *Pattern Recognition Letters*, 7(4):231–238, April 1998.

[GW90] O. Günther and E. Wong. The arc tree: an approximation scheme to represent arbitrarily curved shapes. *Computer Vision, Graphics, and Image Processing*, 51(3):313–337, 1990.

[HA03] J.E. Harmon and S.J. Anderson. *The design and implementation of Geographic Information Systems*. Wiley, 2003.

[HCC02] Ian Heywood, Sarah Cornelius, and Steve Carver. *An Introduction to Geographical Information Systems*. Prentice Hall, 2002.

[HK01] Pong-Sik Ho and Min-Hwan Kim. A hierarchical scheme for representing curves without self-intersections. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition(CVPR'01)*, 2:498–503, 2001.

[HR75] Freeman H. and Shapira R. Determining the minimum-area encasing rectangle for an arbitrary closed curve. *Communications of the ACM*, 18(7):409–413, July 1975.

[JSG99] Mark R. Johnston, Christine D. Scott, and Robert G. Gibb. Problems arising from a simple gis generalisation algorithm, 1999.

[KDE05] Lars Kulik, Matt Duckham, and Max J. Egenhofer. Ontology-driven map generalization. *Jounal of Visual Languages and Computing*, 2005.

[KO03] M.J. Kraak and F. Ormeling. *Cartography: Visualization of Geospatial Data*. Prentice Hall, 2003.

[Kol03] Alexander Kolesnikov. *Efficient Algorithms for Vectorization and Polygonal Approximation*. PhD thesis, University of Joensuu, 2003.

[MS00] Andrea Mantler and Jack Snoeyink. Safe sets for line simplification. *Abstracts of the Tenth Annual Fall Workshop on Computational Geometry*, October 2000.

[Pir99] Hormoz Pirzadeh. Computational geometry with the rotating calipers. Master's thesis, School of Computer Science, McGill University, November 1999.

[PS85] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry an introduction*. Springer-Verlag, 1985.

[RSV02] P. Regaux, M. Schnoll, and A. Voisard. *Spatial Databases with applications to GIS*. Academic Press, 2002.

[SRS03] Biswajit Sarkar, Sanghamitra Roy, and Debranjan Sarkar. Hierarchical representation of digitized curves through dominant point detection. *Pattern Recognition Letters*, 24, 2003.

[Suna] Dan Sunday. Bounding containers for polygons, polyhedra, and point sets(2d & 3d). `http://geometryalgorithms.com/Archive/algorithm_0107/algorithm_0107.htm`.

[Sunb] Dan Sunday. Convex hull of a 2d simple polyline. http://geometryalgorithms.com/Archive/algorithm_0203/algorithm_0203.htm.

[Tay] Dr.George Taylor. Line simplification algorithms. Technical report.

[Tou83] Godfried Toussaint. Solving geometric problems with the rotating calipers. *IEEE MELECON'83*, May 1983.