Combining Orthogonal Range Search and Line Simplification Using Priority Search Trees

Master Thesis in Computer Science

Linda Cecilie Kjeldsen

June 25, 2005
Halden, Norway

Høgskolen i Østfold
Avdeling for Informasjonsteknologi

Abstract

Keywords: Computational Geometry, Spatial Data Structures, Orthogonal Range Search, Line Simplification, Priority Search Tree

This thesis presents a method for combining orthogonal range queries and line simplification in 2D geographical data. Asking for data representing a specific area and have it displayed in an appropriate resolution is one of the basic operations of a geographic information system. In most applications today, the data is stored in several layers of different resolutions, and when a query is performed the approximation of the resulting area is created by querying in the correct resolution layer. The combined solution proposed here is based on the priority search tree (PST) developed by Edvard T. McCreight in the mid eighties. The PST is a data structure used for performing semi-infinite range queries. It was realized that its properties made it suitable for answering stabbing queries as well. This ability to answer stabbing queries is what has been utilized and developed further in the data structure presented here. To solve the line simplification part of the problem, an algorithm presented by Zhilin Li and Stan Openshaw in 1992 was used. This algorithm creates approximations by placing a grid over the map, and selecting the points where the original line intersects the grid lines. We realized that this problem could be solved as multiple stabbing queries. In the combined solution presented in this thesis, the PST is incorporated in another, two-level, data structure. A query in this new data structure returns an orthogonal range which is an approximation of the same range from the original data set. This approximation is created by performing an adjusted query in the PST, which answers multiple stabbing queries. The test results presented show that this new adjusted query in the PST does not have exactly the same complexity as the traditional query, but it still performs surprisingly well.

Acknowledgements

There are some people I would like to thank for their help and support during the work with this thesis. First of all, I would like to thank my teaching supervisor, Gunnar Misund. He suggested the line simplification method that was chosen for this project, that this could be formulated as several stabbing queries, and that these stabbing queries could be answered by adjusting the search algorithm for the PST. He has given excellent guidance and has been a great support throughout this project.

I would also like to thank my class mates, Bjørn Håkon, Kristian, Hilde and Mats for the wonderful time we have had during the work with our theses. They have given indispensable assistance in everything from determining the layout of the thesis to proof-reading the final text. This semester could have been tough and no fun at all, but we have supported each other and have had some great laughs even when things looked the worst.

Finally, I would like to thank my boyfriend for bearing with me, though I have been absent both in spirit and in flesh for the most part of this semester.

Prerequisites

Because the problems that this thesis deals with cover so many different aspects of computer science, it is not possible to go into every little detail of all the subjects that are mentioned. Consequently, it is assumed that the reader already has knowledge at the same level as a third year student of computer science. The reader should have basic knowledge of the construction of and search algorithms for data structures like binary trees, heaps and arrays. He or she should also be familiar with basic analysis concepts like big-O notation, and the difference between worst case and average case complexity. However, in case there still are unclarities, a glossary can be found in Appendix A, introducing most of the important terms used in this thesis.

Table of Contents

Chapter 1

Introduction

The main purpose of this thesis is to perform empirical tests on a new data structure used for combining orthogonal range search and line simplification, and thereby determine whether the proposed approach to this problem is applicable. This new structure is based on the priority search tree (PST), and an adjusted variant of the search method usually associated with this structure.

Asking for data representing a specific area, and have it displayed in an appropriate resolution, is one of the basic operations of a geographic information system (GIS). Since computational geometry emerged as an independent field of research in the 1980s, several data structures and algorithms for efficiently solving these two problems have been presented. However, these structures and algorithms generally only solve one of the two problems. In most GIS applications today the data is stored in several layers of different resolutions, and when a query is performed the approximation of the resulting area is created by querying in the correct resolution layer. Figure 1.1 illustrates the file system of one of the applications that are based on the layer stored approach. This approach is efficient in that it gets the job done, but it is also very resource demanding. As one layer often contains half the amount of data that the preceding layer contains, the layers will converge at approximately double the storage space. Another, and more significant, problem with this approach has to do with consistency when updating the map. When one feature is added or deleted, all the layers must be updated accordingly.

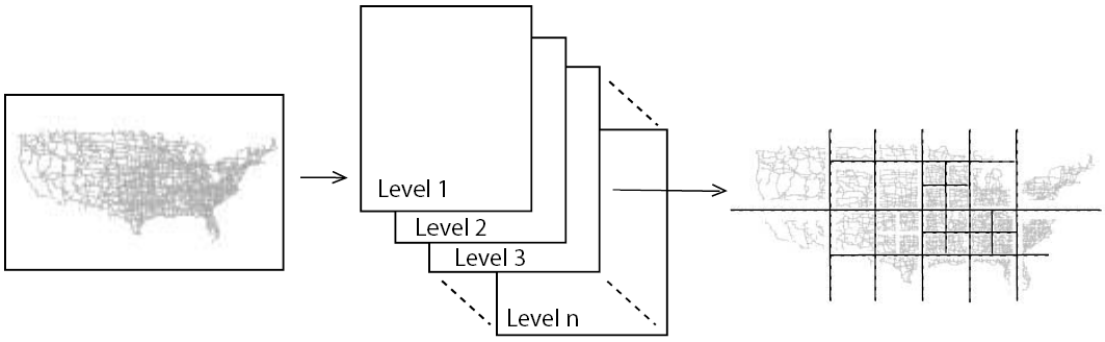


Figure 1.1: Approximated geographical data stored in layers (figure from [14]).

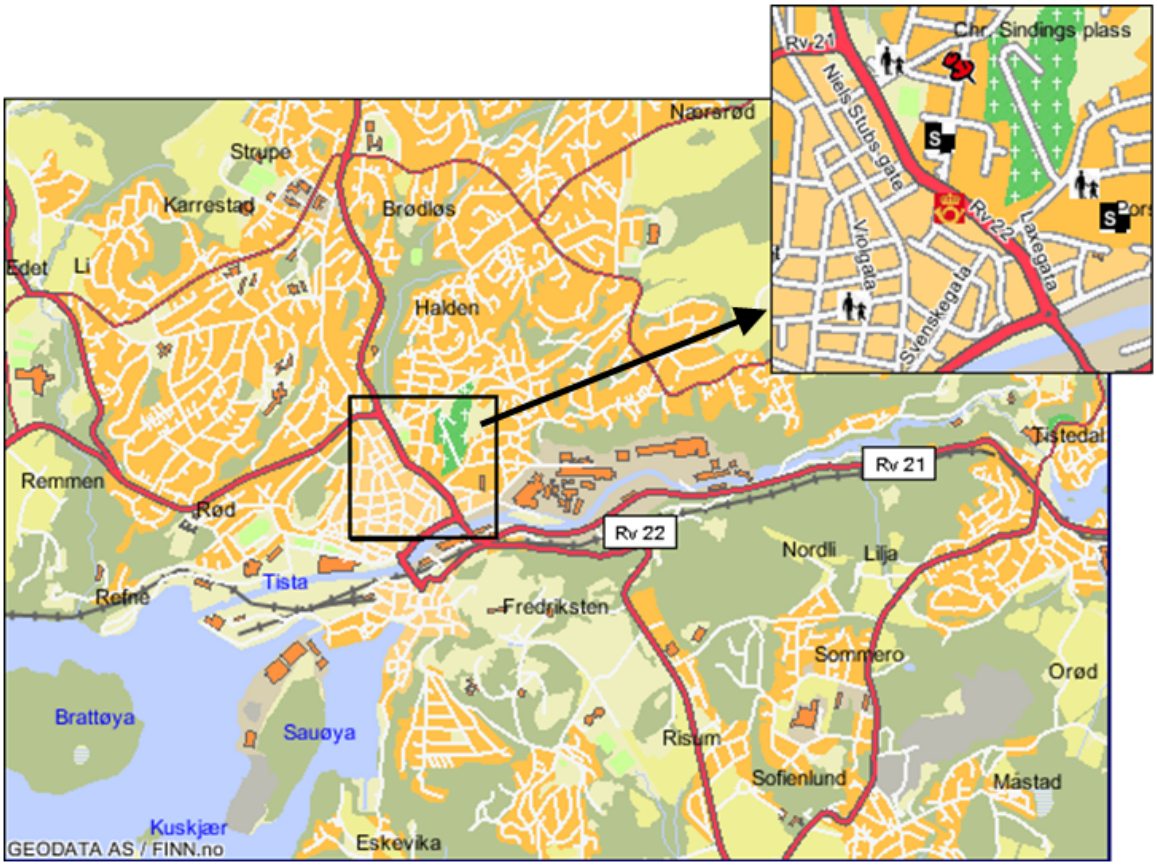
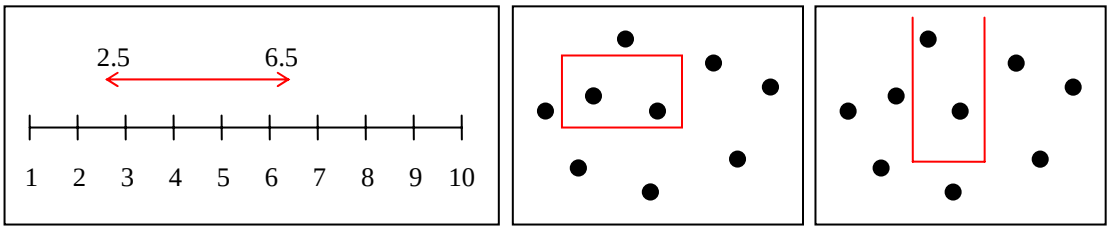


Figure 1.2: Example of a window query in a map (map from www.finn.no).

The process of zooming into a large complex object, like a digital map, and present the result with an appropriate resolution, is called a window query. These queries are not only interesting in digital maps. For instance they are used in the process of designing printed circuit boards where one may want to zoom in on a small portion to see more detail, or in a flight simulator where a landscape model is presented, but only a small area at the time. In addition to these concrete examples, window queries can be performed on all measurements that can be presented in 2D, like temperature or growth over time. Figure 1.2 gives an example of a window query in a map, where a small portion of the map is shown with more detail.

The data structure which has been developed, and is to be tested empirically in this thesis, aspires to solve both the range search and the line simplification problems in one operation. This means that the focus of this thesis must be on both these two separate problems.



(a) 1-dimensional (b) 2-dimensional (c) semi-infinite

Figure 1.3: Examples of range queries.

1.1 Range Search

A range query is a query where the search key is a range. In one dimension the range is defined by a start and a stop value, while in two dimensions the range can for instance be a square, a circle or a polygon. A range search can also be both finite and semi-infinite. A semi-infinite range search is a search where the query range is unbounded in at least one direction. Figure 1.3 shows examples of range queries in one and two dimensions.

In for instance databases, range queries in higher dimensions are useful. One query can be to list all countries that are republics, have an area over 300.000 km^2, a population under 10 million and a GNP (gross national product) over 30.000 PPP (purchasing power parities). This query will return a four-dimensional "box" containing entries that meet all four requirements.

Common for all range queries is that the complexity of the search is output sensitive. This means that the number of objects visited in the search depends on how many objects are reported, instead of only depending on the number of objects in the data set, as is the case in regular searches.

1.2 Line Simplification

Line simplification is the process of removing detail from a line feature without losing the perceptual characteristics of the line. Traditionally this process is done manually by cartographers, but over the years several solutions to automating the process have been developed. There are several advantages to automating this process, like saving time and effort and limiting the individual influence of the cartographer. However, it has proven difficult to develop an algorithm that produces a satisfactory result in all cases. For a more thorough, general overview of the different algorithms developed for line-simplification, see Weibel [32].

There are a lot of different considerations to take when producing an approximation of a line. A cartographer can see and make sound judgements based on what he or she sees, but a computer can not do that. The computer can only take the input and produce an output based on the rules it has been given. One example is if the approximation is based on snapping points to a grid and a river and a road run parallel; how can we make sure the computer "knows" that and keeps them parallel in the approximation? Nevertheless, despite all the obvious problems, many different approaches to this problem have been taken, and many different algorithms have been developed. Figure 1.4

(a) input (b) output

Figure 1.4: Input and result from running the Douglas-Peucker line simplification algorithm.

shows input and output from one of them, the Douglas-Peucker algorithm from 1973 [7].

1.3 Thesis Overview

The second chapter is a background chapter, providing background information on range search and line simplification. In the section concerning range search, several data structures and algorithms are presented. The second section describes some of the different techniques developed for performing completely automated line simplification procedures. Finally another proposed approach to combining the two problems is presented.

The third chapter gives a general description of the PST and what it can be used for. The interval stabbing problem is an important aspect of the work presented in this theses, and the third chapter explains how to solve this with a PST. Next, the interval stabbing problem is expanded to a "grid stabbing problem" which also can be solved using a PST, and the reason for this is described. The fourth chapter gives a detailed description of the new data structure and the search methods that have been developed. After this, theoretical analyses are provided. This chapter also explains how an external version of it has been implemented, along with empirical test results to support the theory.

The fifth chapter presents suggestions for further work. Some work on the suggestions that are made has already been conducted, and this work is also described in this chapter. Finally there is a chapter providing discussions and conclusions to whether or not the problem can be solved using the approach presented in this thesis.

Chapter 2

Background

This chapter provides background information for the two separate problems treated in this thesis. The first section deals with range search. It starts with an introduction to this type of search followed by an overview over already developed data structures and algorithms that solves this problem. The second section describes some of the different techniques developed for performing completely automated line simplification procedures. In the third section, another proposed approach to combining the two problems is presented.

2.1 Range Search

As already stated in the introduction, a range search is a query where the query key is a range. Searching for a range leads to more than one value being reported. In one dimension the data set can be seen as points on a line, and the query key will be an interval within which all points shall be reported. Such a query can be efficiently solved using a regular binary tree, or even a sorted array. Figure 2.1 illustrates how a 1-dimensional range search is carried out in a regular balanced binary tree. Using an array is not that interesting in this context, because it can not be extended to higher

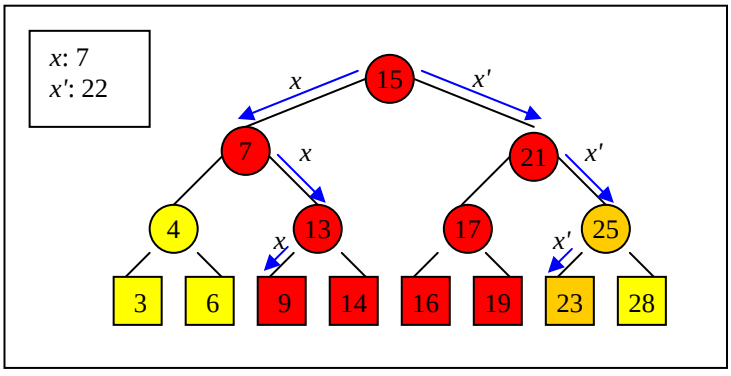


Figure 2.1: A range search in a regular balanced binary tree. The red nodes are the ones reported from the search, the orange are visited but not reported, and the yellow are not affected by the search at all.

dimensions.

The search is performed by iterating down the tree until one node where the paths to the two end points of the query interval split, in Figure 2.1 this node is the root. After this node is found, the search is continued both to the left and to the right, comparing the nodes in the tree to each of the end points. When comparing to the lower end point of the interval query and the search path goes left, all nodes in that node's right subtree are reported, and vice versa with the upper end point. This can be seen in the figure; When node 21 is reached the search path goes right, and then it is clear that the nodes in 21's left subtree (nodes 16, 17 and 19) must be reported.

As the data structure used to answer the query is a balanced binary tree, its complexity is for the most part already known. It uses $O(N)$ storage and can be built in $O(NlogN)$ time, where N is the number of points in the data set. Because the search key is a range, and therefore the result also is a range, the query time can not be determined only based on the number of points in the data set. If the query interval is larger than the range of the data set, all points must be reported, which leads to a query time of $O(N)$. This seen isolated is not a good query time. However, when all the points must be reported, it is naturally also necessary to visit all the points in the tree. If we refer to the number of reported points as k, the time needed to report the points from a query is $O(k)$. In addition to reporting the points, some nodes must be visited to find which nodes to report. The height of a balanced binary tree is $O(logN)$, and therefore, so is the maximal number of nodes visited that are not reported. If this is added to the reporting time, the total time used for performing a range search in a binary tree is $O(logN + k)$ [5](page 99).

Now that a one dimensional range query can be answered, it is time to expand to two dimensions. An orthogonal range query in two dimensions should return all points that are positioned within the boundaries of the given search area. As the search area of an orthogonal range query in two dimensions will be shaped like a rectangle, this search can be seen as two one dimensional range queries; one in each dimension. Several data structures and search algorithms have been developed to solve this type of query. The following subsections present some of them.

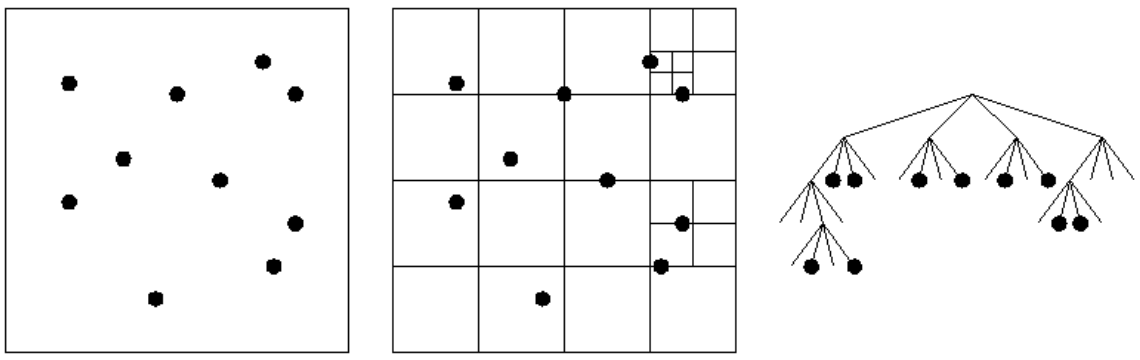


Figure 2.2: The construction of a quad-tree (figure from [3])

2.1.1 Quad Tree

Quadtrees recursively divide space into quadrants as long as more than one node falls into each quadrant, see Figure 2.2. Every non-leaf node in the tree has four children, each corresponding to a quadrant that is one of the squares into which the parent's quadrant has been divided. The leaves are either a node or $null$, depending on whether or not there is a node in the corresponding quadrant.

The depth of a quadtree is determined by the number of points in the data set and the spacing of these points: "The depth of a quadtree for a set P of points in the plane is at most $log(s/c) + \frac{3}{2}$ where c is the smallest distance between any two points in P and s is the side length of the initial square that contains P." [5] (page 295).

The worst case complexity of a range query in a quadtree is $O(k * N^{1-1/k})$ where N is the number of nodes in the tree and k is the number of nodes that are returned from the query [20]. The required storage for the tree is $O((d+1)N)$ where d is the depth of the tree [5] (page 296).

2.1.2 Kd-tree

The kd-tree was presented by Jon Louis Bentley in 1975 [2], and is used to organize data in k dimensions. In two dimensions, the kd-tree is a binary tree, representing coordinates in the plane. The plane is recursively split into two equal subsets by axis-orthogonal lines. On even levels in the tree the line is orthogonal to the x-axis, while on odd levels it is orthogonal to the y-axis. Each left subtree consists of the points that have a coordinate value lower than it's parent's splitting value, and each right subtree consists of those that have a value that is higher. All data points are stored in the leaves, which form a partition of the plane into disjoint rectangular regions containing one point each, see Figure 2.3. This tree will have size $O(N)$, depth $O(logN)$ and construction time $O(NlogN)$, where N is the number of points in the data set.

As can be seen from the figure, each node v in the tree corresponds to a region $Reg(v)$ on the plane where the points in the set are situated. Pseudo code of the algorithm for performing a range search in a kd-tree is provided below:

```
1. Recursive procedure, starting at v = root.
2. Search(v,R)
    a. If v is a leaf, then report the point stored in v if it lies in R
    b. Otherwise, if Reg(v) is contained in R, report all points in the
    subtree of v
    c. Otherwise:
        i. If Reg(left(v)) intersects R, then Search(left(v),R)
        ii. If Reg(right(v)) intersects R, then Search(right(v),R)
```

This search has a complexity of $O(\sqrt{N}+k)$, where N is the number of nodes in the kd-tree, and k is the number of nodes reported from the search. $O(k)$ is the time needed to report the points that fall within the range. The number of visited nodes in the tree is the same as the number of regions $(Reg(N))$ that intersect R but is not contained in R. To find this, find the number of $Reg(N)$ which are crossed by any of the four horizontal or vertical lines in R. The maximum number $Q(n)$ of regions in an n-point kd-tree that intersects a vertical line is $Q(n) = 1 + Q(n/2)$ if we split on x, and $Q(n) = 1+2*Q(n/2)$ if we split on y. Since we alternate, we can write $Q(n) = 2+2Q(n/4)$, which solves to $O(n^{1/2})$. Hence the complexity of $O(\sqrt{N}+k)$.

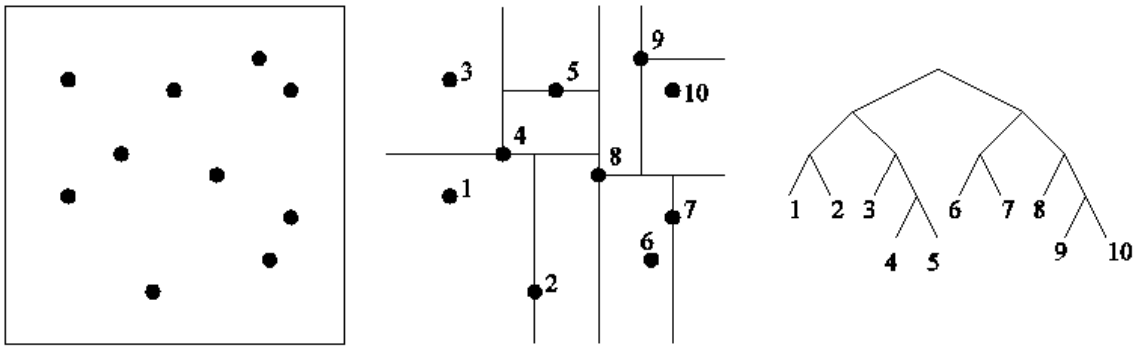


Figure 2.3: The construction of a Kd-tree (figure from [3])

Bkd-tree

A Bkd-tree is a dynamic scalable kd-tree organized as a B+ tree presented by Procopiuc et.al. in 2002 [19]. It is described as "the first theoretically and practically efficient dynamic adaptation of the kd-tree to external memory" [19]. This tree is actually a set of $log_2(N/M)$ static trees (N is the number of items in the data set, and M is the number of items that can be stored in main memory), and updates are performed by rebuilding a carefully chosen subset of these static structures. The Bkd-tree is efficient in terms of reducing the number of I/O-operations both in queries and in updates, and it has high storage utilization (uses close to N/B disk blocks, where B is the number of items in a block).

To query a Bkd-tree, all the $log_2(N/M)$ trees must be queried, but theoretically holds the worst case optimal query bound. In addition the average case update time is substantially lower than that for other attempts of developing a dynamic kd-tree. The periodical updates leads to the update time varying, hence, the average update time is measured.

Extensive testing of the Bkd-tree shows that the theoretical properties makes this implementation of a external and dynamic kd-tree substantially more efficient than other versions. The updates are performed much more efficiently, while the space utilization is maintained. In addition, the fact that the data set is stored in $log_2(N/M)$ static trees, does not appear to affect the query time.

2.1.3 Range Tree

A 1D range tree is a binary tree structure used for answering 1 dimensional range queries. This structure and the search method is very similar to the regular binary tree described above, but there is one difference. In a 1D range tree, all the information is stored at the leaves of the tree.

The search in a 1D range tree is carried out by comparing the x-values of the nodes in the tree to the x and the x' values. When a node v_s is reached, whose x-value lies between x and x', the search is split. The search is continued in the left subtree by comparing the x-values to x. If x is smaller than the x-value, all values in the right subtree of the node is reported. Similarly the right subtree of v_s is searched using x'. When x' is higher than the x-value of a node, all values in this node's left subtree are reported. Figure 2.4 illustrates a search in a 1D range tree.

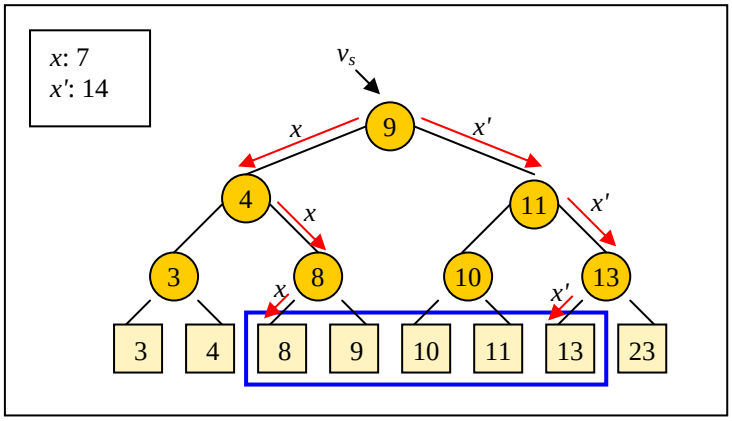


Figure 2.4: A search in a 1D range tree.

A 2D range tree is a two-level tree structure where the first level is a 1D range tree, and all the nodes in this tree are linked to another 1D range tree. The 2D range tree is used for, for instance, organizing geographical points both according to their x-coordinates and their y-coordinates, and for performing range queries on these points. In this case, the x- and y-coordinates are each given their own level. The first level of the tree is a 1D range tree organizing the points according to their x-coordinates. Each node on this level has an associated 1D range tree which organizes the points according to their y-coordinates. Figure 2.5 shows an example of a 2D-range tree.

In a 2D range tree, the range query is carried out in a very similar way as in 1D. The difference is that instead of reporting all values in the subtrees of v_s, a second search is carried out in v_s's associated tree structure. The coordinates reported are those from this associated tree structure that have a y-value that lies within the y-limits of the query rectangle. The search in the associated tree structure is parallel to that of the x-values in the 1D structure.

A 2D range tree is built in $O(NlogN)$ time, and uses $NlogN$ storage, in both cases N equals the number of coordinates in the data set. Because the search is performed in two levels, the time used for searching this structure is $O(log^2N + k)$, where k is the number of coordinates returned from the search [5] (page 109).

Yi-Jen Chiang and Roberto Tamassia have described a method for making the 2D range tree dynamic [4]. According to this method, inserting a point in a 2D range tree is done as follows:

```
1. Create a new leaf, n, for the new point p.
2. Insert p in the associated datastructure of every ancestor of n.
3. Rebalance the tree by rotating.
```

As the associated datastructures depend on what points belong to which subtree, the rotations become more complex than in a regular balanced binary tree. When performing a rotation at a node v, the associated datastructures of the nodes involved in the rotation must be rebuilt from scratch. The nodes involved are all the nodes in the subtrees of v, hence, the rotation takes time proportional to the number of nodes in these subtrees. This is done in $O(logN)$ time amortized. Inserting p in the associated datastructures takes $O(log^2N)$ time. This leads to a total update time of $O(log^2N)$ amortized.

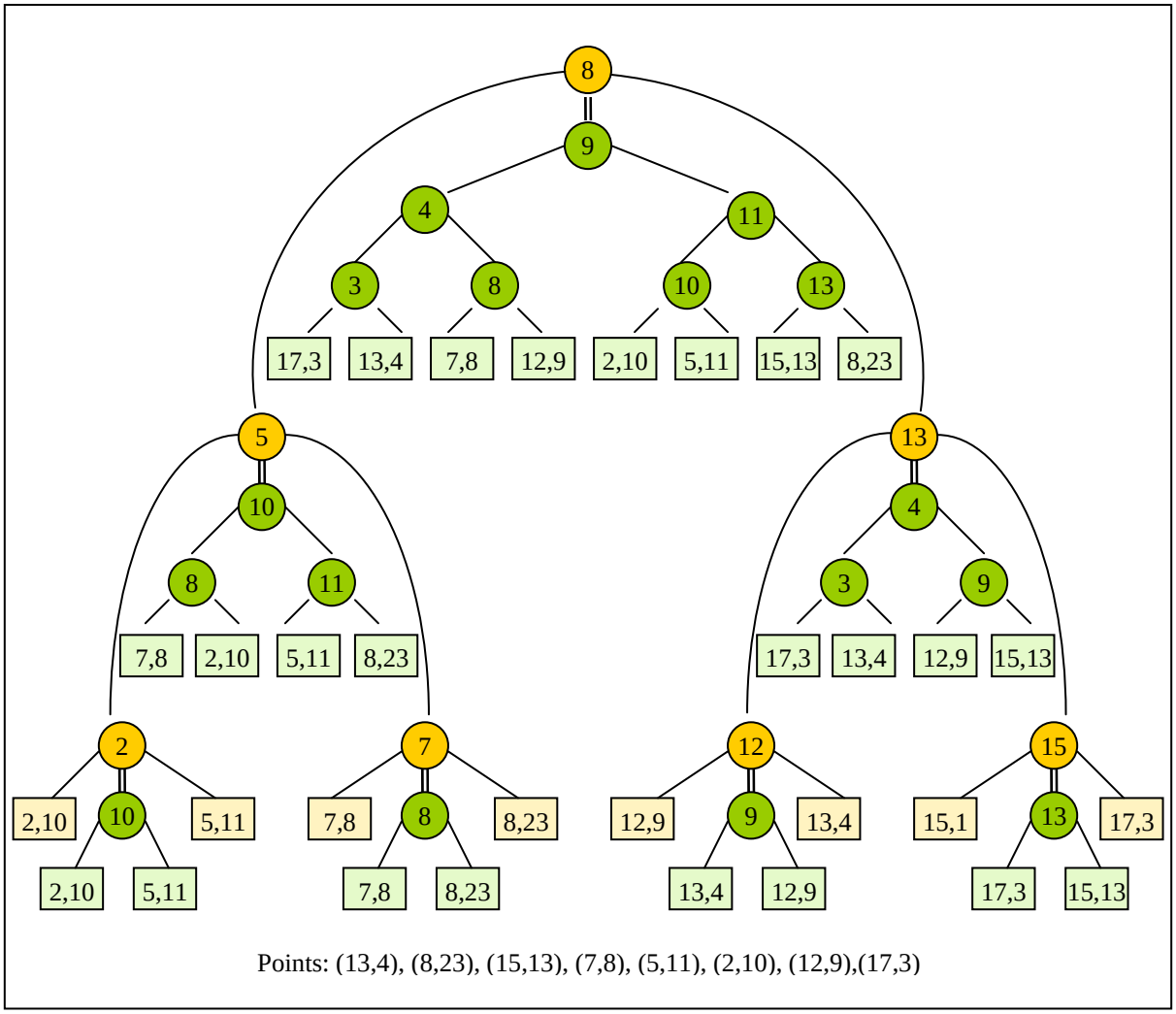


Figure 2.5: An example of a 2D-range tree. The orange nodes belong to the first level, and the associated structures have green nodes.

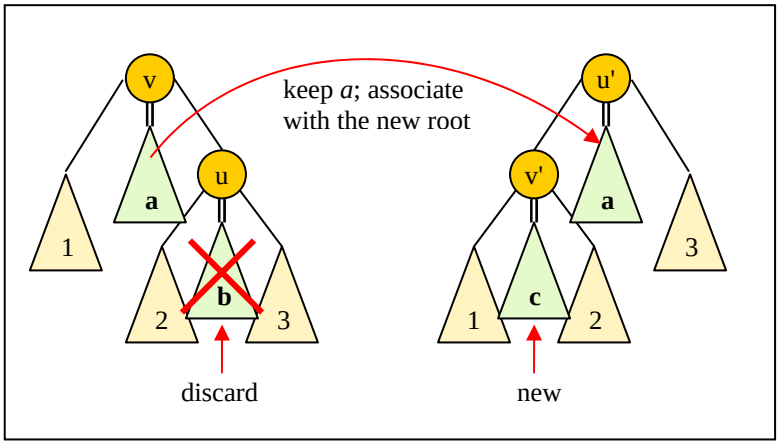


Figure 2.6: A rotation in a 2D range tree.

However, in a lecture from 1991 [24], Tamassia says that the only associated structure that needs to be rebuilt from scratch is that of the node that becomes the child after the rotation. Figure 2.6 shows how this rotation is carried out, and v' in the figure is the node whose associated structure must be rebuilt from scratch. As the update time is dominated by the step where the new node is inserted in the associated structures, this procedure also has an update time of $O(log^2 N)$ amortized.

2.1.4 Interval Stabbing-max Structure

The "Interval Stabbing-Max Data Structure" is developed by Agarwal et.al. [1], and is used for efficiently answering stabbing-max queries on orthogonal ranges in all dimensions. This is interesting because the adjusted search method for the PST, which is presented in chapter 4, is developed for answering stabbing queries with multiple stabbing values in two dimensions.

Agarwal et.al. present a new data structure for solving the problem of dynamically maintaining a set of axis-parallel hyper-rectangles in $\mathbb{R}^d$. This new structure is based on the interval tree which is a binary tree for storing intervals. Every node in the interval tree stores a value that splits the endpoints of the intervals in its subtrees in half. The intervals completely to the left and to the right of this value are also stored in the node, while the intervals that intersect the split value are stored in two separate structures (lists or binary trees or similar); one sorted on the left endpoints and one sorted on the right. The interval tree is used for efficiently finding all intervals that intersect a given value. It uses $O(N)$ storage, has $O(logN)$ depth, and answers queries in $O(logN + k)$ time, where N is the number of intervals in the data set and k is the number of intervals reported from the query [5] (page 213).

The idea of the new structure is to increase the fan-out of the base tree τ to $log^c N$, and thereby decreasing the hight of the tree to $O(logN/loglogN)$ which allows spending $O(loglogN)$ time at every node in the search path and still obtain $O(logN)$ query time. The new structure consists of a balanced base tree τ with $N/logN$ leaves with a fan-out of $f = \sqrt{logN}$. The f children of a node v are organized in a binary search tree, which can be searched in $O(logf) = O(loglogN)$ time.

Each node in the tree is associated with a range σ_v which is the union of all ranges associated

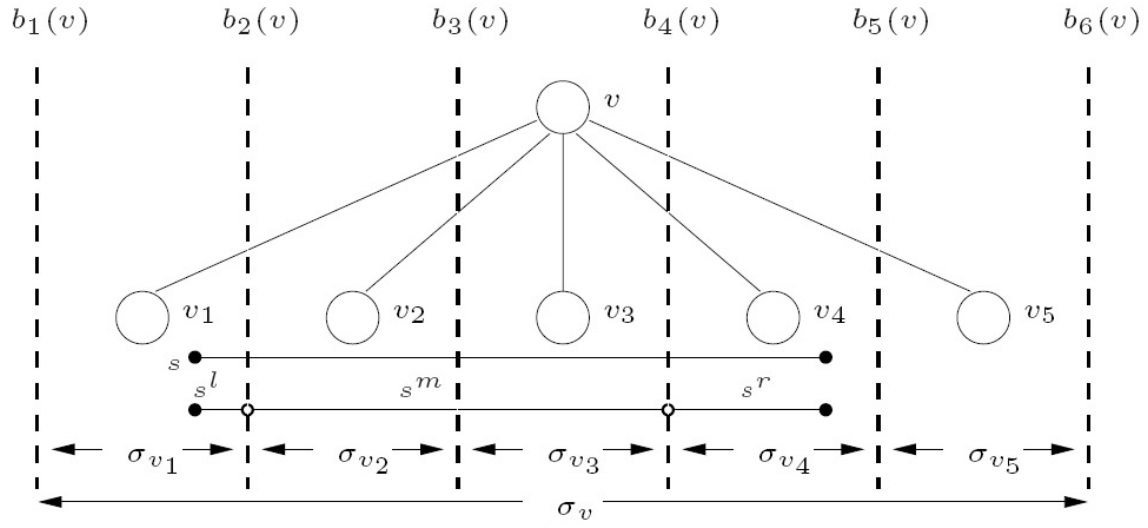


Figure 2.7: An internal node v in the base tree τ. (Figure from [1])

with its children. σ_v is divided into f subranges according to the children's ranges, these subranges are referred to as *slabs*. The boundaries of the subranges are referred to as *slab boundaries*, and $b_i(v)$ is used to denote the slab boundary between $\sigma_{v_{i-1}}$ and σ_{v_i}. *Multislabs* are continuous ranges of slabs; $\sigma_v[i:j]$ is the multislab consisting of slabs σ_{v_i} through σ_{v_j} for $1 \leq i \leq j \leq f$. Figure 2.7 shows a node v with $f = 5$ children.

An interval s is associated with a node v if it crosses at least one of v's slab boundaries, but none of v's parent's. If an interval has both endpoints in a leaf z, it crosses no boundaries, and is stored in z. If an interval has both endpoints in an internal node, it is stored in this node's associated structure.

A stabbing-max query, q, in this structure is carried out by querying the associated structures of all the internal nodes on the path from the root of the tree to the leaf containing q, and then return the maximum of the reported intervals. Since the height of the tree is $O(logN/loglogN)$ and $O(loglogN)$ time is spent at each node in the path, the total query time will be $O(logN)$.

2.2 Line Simplification

Line simplification is one of the key elements to cartographic generalization. In [12] generalization is defined as the process of reducing the size and complexity of a spatial dataset with visual quality preserved or enhanced. In a cartographic context this means that, for instance, in a GIS, if a map of a town is displayed, and then the map is zoomed out to display the entire country, the amount of detail presented in the map of the town must be reduced in order to maintain the clarity of the map. To reduce the amount of detail, some of the data presented must be removed; the lines must be simplified, some features must be eliminated, others moved, enlarged or combined. Introductions to and analyses of many tools and data structures developed for performing automated generalization

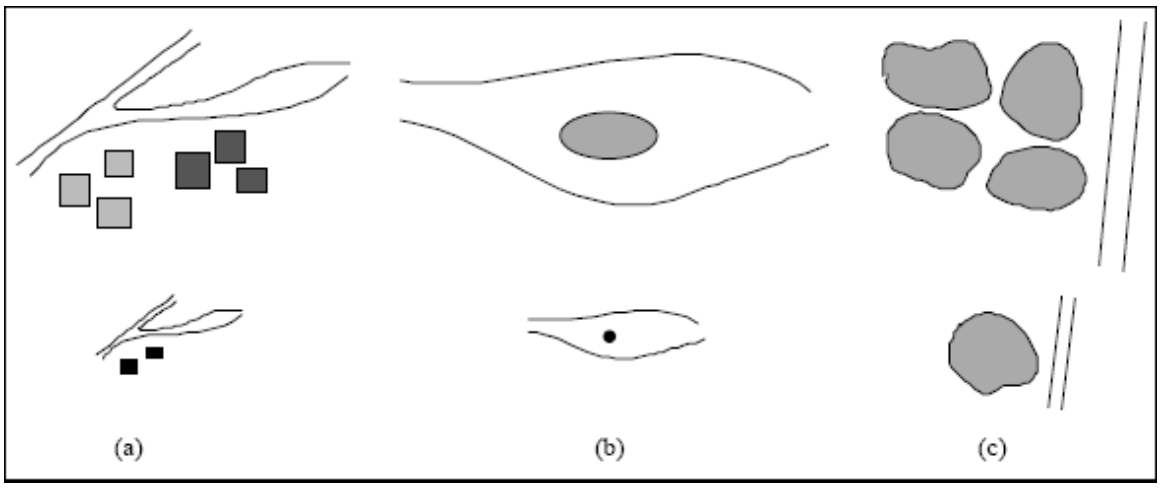


Figure 2.8: Some generalization techniques: (a) aggregation - join a group of different features into a higher-order feature; (b) collapse - symbolize a feature as a lower-order feature; (c) amalgamation - join features of the same class into a larger feature of the this class. (Figure from [6])

of geospatial data can be found in [30] and in [13].

As geographic information systems and spatial databases increase in number and size, developing techniques for automated generalization is one of the issues must be addressed. Both in areas like the creation and maintenance of spatial databases at multiple scales, cartographic visualization at variable scales and data reduction, these techniques are needed. Generalization is a composite and complex process and can be split into categories [27]:

- Simplification - e.g. line simplification
- Combination - combine features theoretically or geometrically
- Symbolization - e.g. reduce a polygon to a polyline or a point
- Selection - eliminate features that are too detailed
- Exaggeration - enlarge features that are important for a specific type of map
- Displacement - move features

Figure 2.8 gives examples of techniques within the combination and symbolization categories.

As already stated, line simplification is one of the key elements to cartographic generalization. The line simplification algorithms can be divided into the following categories [32]:

- ***Independent point algorithms*** are outdated algorithms where the points in the simplified line are picked by random.
- ***Local processing algorithms*** are algorithms where the characteristics of immediate neighboring points determine which points to include in the simplified line. The local processing algorithms can be divided into two sub categories:

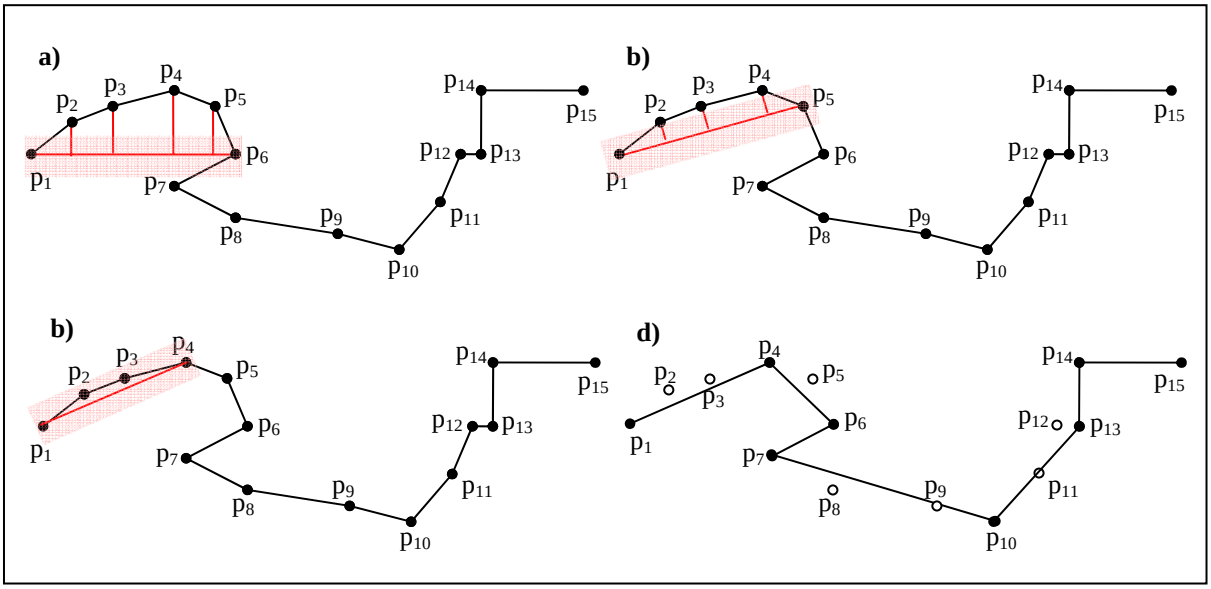


Figure 2.9: Lang algorithm. Figures a-c show the first iteration for a complete look-ahead. Figure d depicts the resulting line segments, with the eliminated points shown in white. (Figure reproduced from [32])

  - ***Constrained extended local processing algorithms*** operate on sections of the line to decide which points to include and which to exclude.
  - ***Unconstrained extended local processing algorithms*** evaluate sections of the line in the simplification process, but in this class the extent of the search is determined by the shape complexity of the line.

- ***Global algorithms*** consider the entire line and iteratively select critical points, while excluding others based on criteria of tolerance.

The following sections describe some of the many different algorithms and data structures for automated line simplification developed over the years. Many approaches have been taken, each with the goal of automatically creating an approximation of a polyline as efficient as possible and with the best possible result.

2.2.1 Lang

The Lang algorithm was published in 1969, and is one of the earliest line simplification algorithms [9]. It belongs to the constrained extended local processing algorithms, and the extent of the local search is controlled by the so-called 'look-ahead' parameter. If, for instance, this parameter is five, then segments of five points are controlled to see if any of the points fall outside a tolerated distance.

Figure 2.9 shows how the Lang algorithm works on a line with a look-ahead of 5 points. Start by drawing a base line between point p_1 and a floating end point starting at $(1 + 5 = 6)$. Then

calculate the distance between the base line and the points that are between the starting point and the end point. If any of these points have a distance to the base line that is larger than the tolerated distance, the point previous to the former end point becomes a new floating end point, and the new distances are calculated and checked. This is repeated until all of the points fall within the tolerated distance to the base line. The points that are within the tolerated distance are then eliminated, and the current floating end point becomes the new starting point of the next base line.

The worst case running time for this simplification is .... This worst case scenario occurs when no point is within the tolerated distance at any point, so all floating end points becomes the point immediately following the starting point at every look-ahead sequence.

2.2.2 Douglas-Peucker

The Douglas-Peucker algorithm is a global line simplification algorithm, and was presented in 1973 [7]. Very early a Fortran implementation was developed [33], and this lead to the algorithm being included in practically every GIS package on the market. Because of this the Douglas-Peucker algorithm probably is the most used line simplification algorithm today, and many have suggested and developed improved and modified versions of this algorithm.

The algorithm creates approximations by excluding all points that are within a given tolerated distance, similar to the Lang algorithm described in the previous section. A point's distance is established by drawing a line between the end points of the polyline, and calculating its distance to this line. If there are any points that have a distance larger than what is tolerated, the line is split at the one with the largest distance, and so the process is recursively repeated. Figure 2.10 illustrates this process. More on the Douglas-Peucker algorithm, along with some test results, can be read in [18] (chapter 3).

The worst-case complexity of the Douglas-Peucker line simplification is $O(N^2)$, and the best-case is $O(N)$. The best-case is when the polyline can be simplified to a single line drawn between its end points. The worst-case is when no simplification is performed, that is, all the original points are included in the simplified result [8].

The Binary Line Generalization Tree

In a Binary Line Generalization tree (BLG) is a binary tree used for storing the result of applying the Douglas-Peucker algorithm to a polyline [28]. The points of the polyline are stored as nodes in the tree, and the root of the BLG is the point with the largest distance to the line between the starting and the ending points of the polyline. Figure 2.11 gives an example of a polyline and its BLG, and, as can be seen in Figure 2.11(a), the point p_8 is the one with the largest distance to the line, and is therefore the root of the BLG in Figure 2.11(b).

When the algorithm is completed, all the points, except for the starting point and the ending point which are implicit, are stored in the tree along with their error value, ϵ. ϵ is the distance between the point and the line between the two current end points. Figure 2.11(b) gives the complete BLG that is the result of applying the Douglas-Peucker algorithm to the polyline in Figure 2.11(a).

The most coarse approximation of a polyline is the line between the starting point and the ending point of the polyline. The error of this approximation is determined by the ϵ of the root of the BLG,

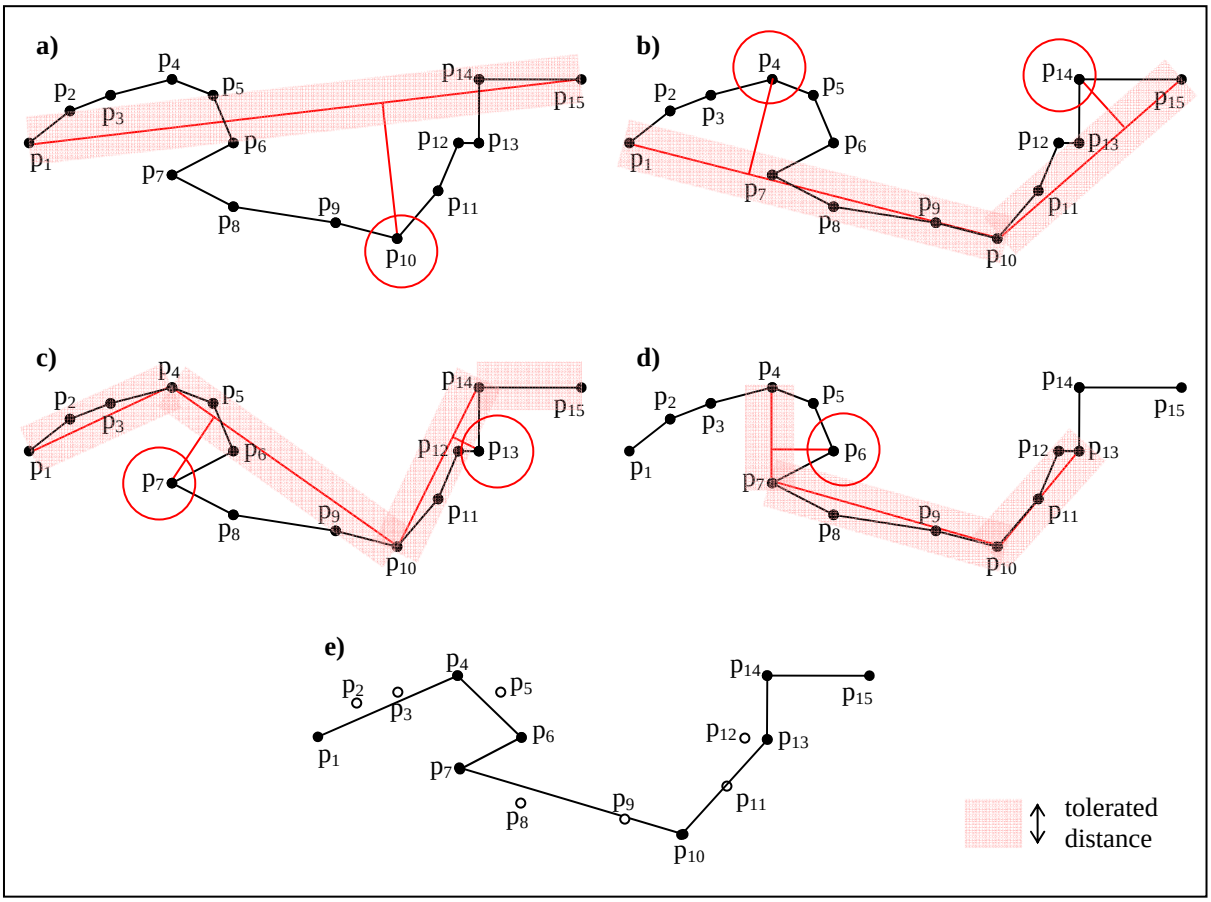
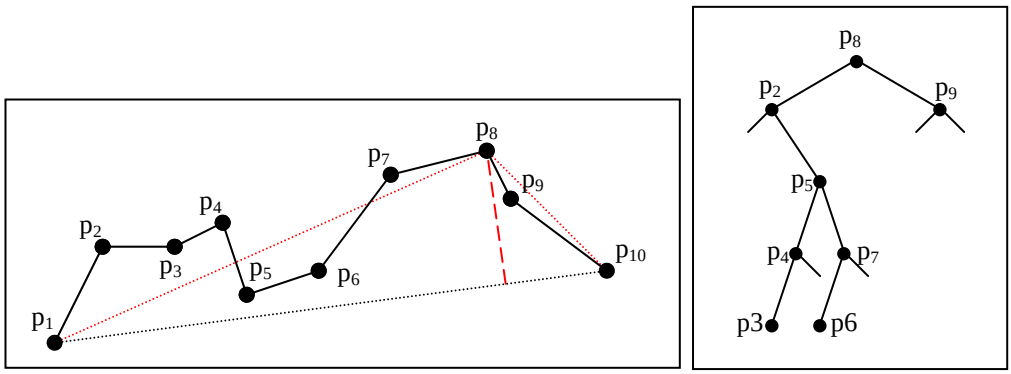
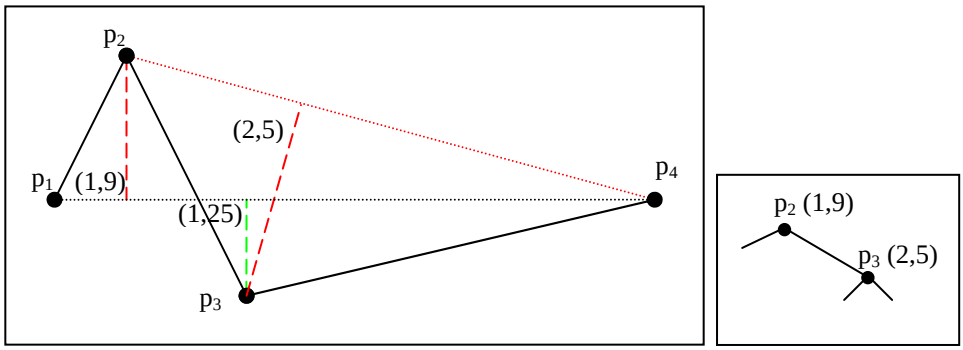


Figure 2.10: The Douglas-Peucker algorithm. a) Initial base line with furthest vertex (p_{10}). b) First split into two parts, again with furthest vertices (p_4, p_{14}) shown. c) Second split. Vertices p_2 and p_3 are now within the tolerated distance, while the second part must be split further at vertex p_7. Also need one more split at p_{13} on the other side. d) One more split is required at p_6. e) The result of the algorithm. Excluded points are colored white. (Figure reproduced from [32])



(a) The input line (b) The output tree

Figure 2.11: Example of a polyline and its BLG.



(a) The input line (b) The output tree

Figure 2.12: Example of a polyline leading to increasing error values in a BLG.

which is the point farthest away from the line and in Figure 2.11(a) this is the point p_8. The next approximation is formed by two lines; one from the starting point to the the root of the BLG, and one from this point to the ending point. This will generally be a more accurate representation of the polyline.

When all the nodes and their ϵs are stored in the BLG, an approximation is generated by performing a search in the tree. the search is continued until the ϵ of a node is so small that the line segment it represents not will be included in the result, and then the search is stopped. In a "normal" situation the ϵs of the nodes in the tree are descending. This means that all the nodes in tree have ϵs that are smaller than their parent's. However, this is not always the case. Figure 2.12 illustrates an example of a polyline leading to increasing error in the BLG. As the search is stopped when one node is reached that has an ϵ that is too small, this type of error can lead to points being excluded that should have been included in the result.

The worst-case complexity of building a BLG is $O(N^2)$, where N is the number of points in

the polyline. The worst-case occurs when the points with the largest distance to the lines always is at one end of the segment of the polyline, so that all the points end up in one subtree at every level of the BLG. The best-case complexity is $O(N)$, and this occurs when the point with the largest ϵ always is the mid point, and, hence, the points are evenly split in half on every level. The complexity of creating an approximation of a BLG is $O(logN + k)$, where N is the number of points in the polyline and k is the number of points returned.

2.2.3 Li/Openshaw

In an article from 1992, Zhilin Li and Stan Openshaw describe a set of algorithms for locally-adaptive line generalization based on the so-called natural principle of objective generalization [10]. Prior to this, many of the existing methods where based on the concept of critical points, like for instance the Douglas-Peucker data reduction method described in the previous section. The article gives examples of, and compares, traditional manual generalization, generalization by the Douglas-Peucker algorithm, and generalization by the set of new algorithms presented in the article.

Li and Openshaw present three new algorithms for map generalization: the vector mode algorithm, the raster mode algorithm and the raster-vector algorithm. A critical parameter of their approach is the size of the smallest visible object, referred to as SVO, which can be a circle or a raster. The following equation is used to calculate the size of the SVO to generalize a map to a given scale:

$$F_c = S_t \times D \times (1 - S_f/S_t) \tag{2.1}$$

In the equation S_t is the scale factor of the map to which the generalization is carried out; D is the *ideal* length of the sides of the SVO at map scale S_t, within which all information can be neglected; S_f is the scale of the original map before the generalization; F_c is the corresponding length of the sides of the SVO in terms of ground distance. To be able to use this formula, a suitable value for D must be found. Li and Openshaw have used $0.4mm$ because that was suggested as the minimum needed to ensure visual separability by Muller [16].

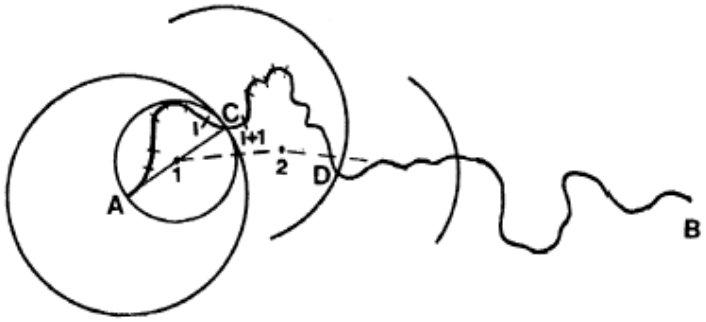


Figure 2.13: Line generalization in vector mode. (Figure from [10])

Vector Mode

The vector mode algorithm uses a circular SVO to generalize the lines, and the size of the SVO is calculated using the equation 2.1 above. To find the location of the first SVO, a circle with a diameter twice as large as the SVO's is positioned with the start point of the line as its center point. This point is marked A in Figure 2.13, and this large circle intersects the line at point C. The line AC will be the radius of the large circle and the diameter of the SVO. The area covered by the SVO can now be disregarded and will be represented by the SVO's center point, 1. This process is then repeated, now with C as the center point of the large circle. When the entire line has been processed, that is, the end point of the line is covered by the SVO, the generalization is complete and the resulting approximated line can be presented.

Raster Mode

The circular SVO used in the vector mode algorithm can be computationally expensive. This leads to another proposed algorithm working in raster mode which uses a raster SVO. Using rasters makes the algorithm faster and easier to implement. The equation used to determine the diameter of the SVO in vector mode, is in raster mode used to determine the length of the sides of the raster SVO. The first raster is positioned with its center point at the start point of the line that is to be generalized. The area covered by the raster will be represented by this center point. The next raster is positioned directly above, beside or below the first raster, depending on where the line string "moves". This process is then repeated until the end point of the line is reached. The rasters will create a grid over the map, and the approximated line is created by selecting the center points of the rasters affected by the original line.

Raster-vector Algorithm

The raster-vector algorithm is, as the name implies, a combination of the vector and the raster mode algorithms. The raster mode algorithm is fast and easy to implement, while the vector mode

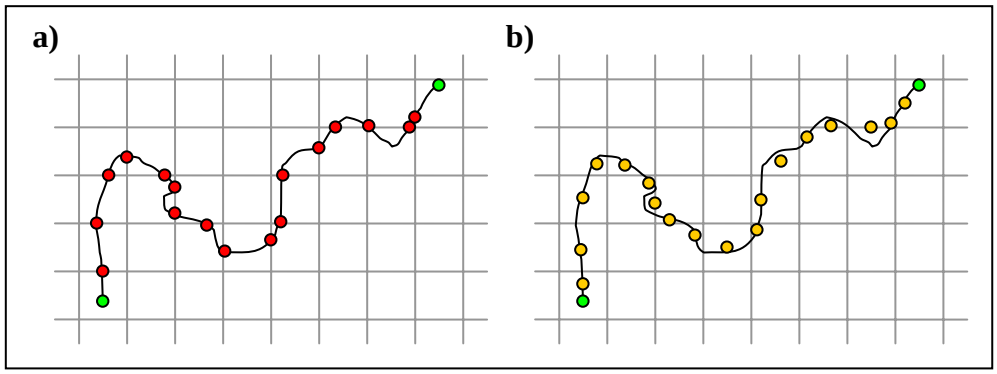


Figure 2.14: (a) The points where a polyline intersects the lines of a grid placed over it; (b) The midpoints between the intersecting points from (a).

Figure 2.15: Generalization examples a) original curve,b) c) and d) are generalized by the raster-vector algorithm, b) with grid size 0.25, c) grid size 1.0 and d) grid size 5.0.

algorithm produces a smoother result. In the raster-vector mode the raster mode's raster SVOs are chosen, combined with the vector mode's method for selecting the points that the generalized result will consist of. Figure 2.14 illustrates how in (a) the points where the line intersects the grid lines are chosen. In (b) the mid points of these intersecting points are calculated, and these are the points that the generalized line will consist of.

Figure 2.15 shows the result of running an implementation of the raster-vector algorithm with different size SVOs on a map of the coastlines of Australia. In Figure 2.16 Li and Openshaw's implementation of the raster-vector method is compared to corresponding generalizations carried out by a manual cartographer and by the Douglas-Peucker data reduction algorithm. The results show that the raster-vector algorithm produces a generalization much closer to that produced by traditional manual generalization, than what the Douglas-Peucker algorithm does.

2.2.4 Visvalingam-Whyatt

This algorithm is a local processing algorithm that was presented in 1993 in an attempt to preserve salient shapes and entire features rather than selecting specific points [31]. Visvalingam and Whyatt argue that the selection of the furthest point outside the tolerance band as a critical point is insufficient as these points can be erroneous or located on minor features.

The algorithm eliminates points on a line based on the effective area ϵ, which is the triangle formed by each point and their immediate neighbors, see Figure 2.17. During execution the original line is processed multiple times, and each time the point with the smallest ϵ is considered least significant, and removed. Then the areas of the neighboring points are recalculated, and the procedure is repeated until all the points are sorted in a sequence according to the size of their ϵ. Figure 2.18 illustrates the last step of generalizing the polyline from Figure 2.17, and the recomputing of the two areas that are left after the generalization is complete.

Below is the pseudo code of the algorithm, cited from [31].

```
Compute the effective area of each point

Delete all points with zero area and store them in a separate list
with this area

REPEAT
```

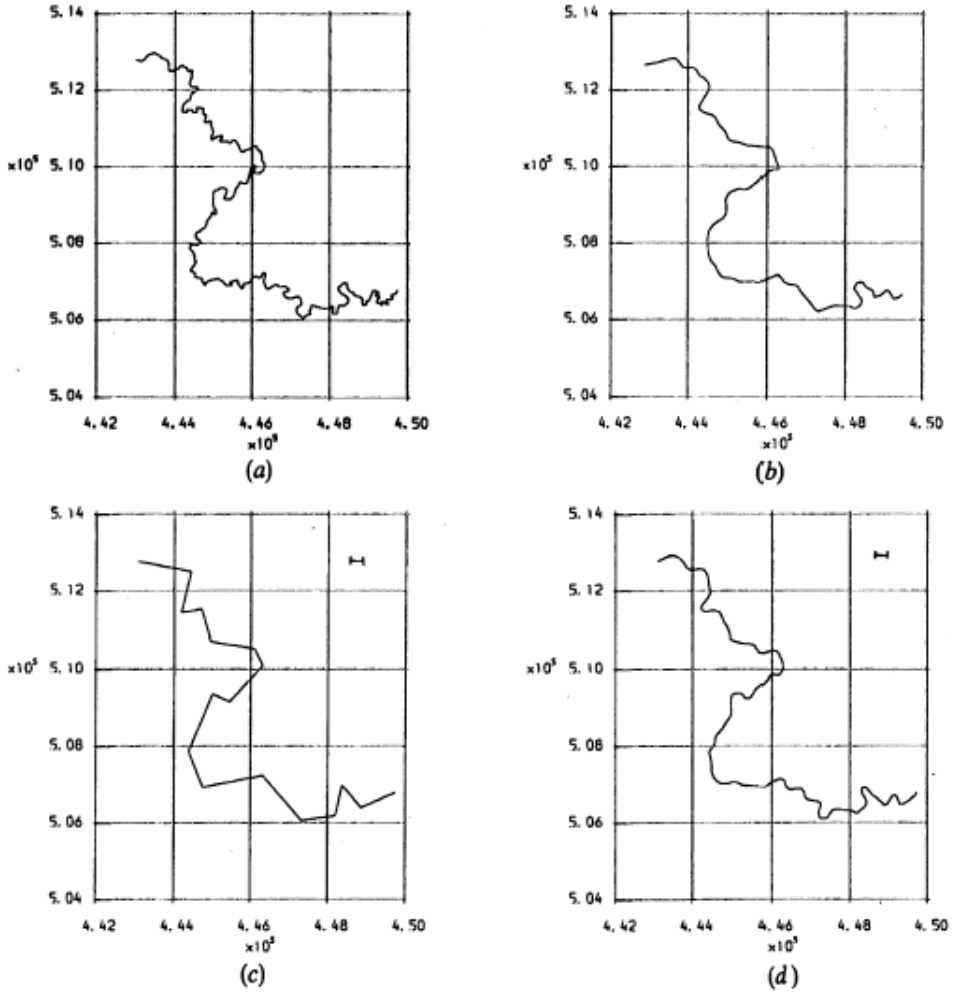


Figure 2.16: Generalization of a river segment by various methods. (a) A river segment digitized from 1 : 10.000 scale map; (b) The river digitized from 1 : 165.000 scale map (which is manually generalized); (c) The river generalized from 1 : 10.000 to 1 : 165.000 scale by Douglas data reduction algorithm; (d) The river generalized from 1 : 10.000 to 1 : 165.000 scale by the raster-vector algorithm. (Figure from [10])

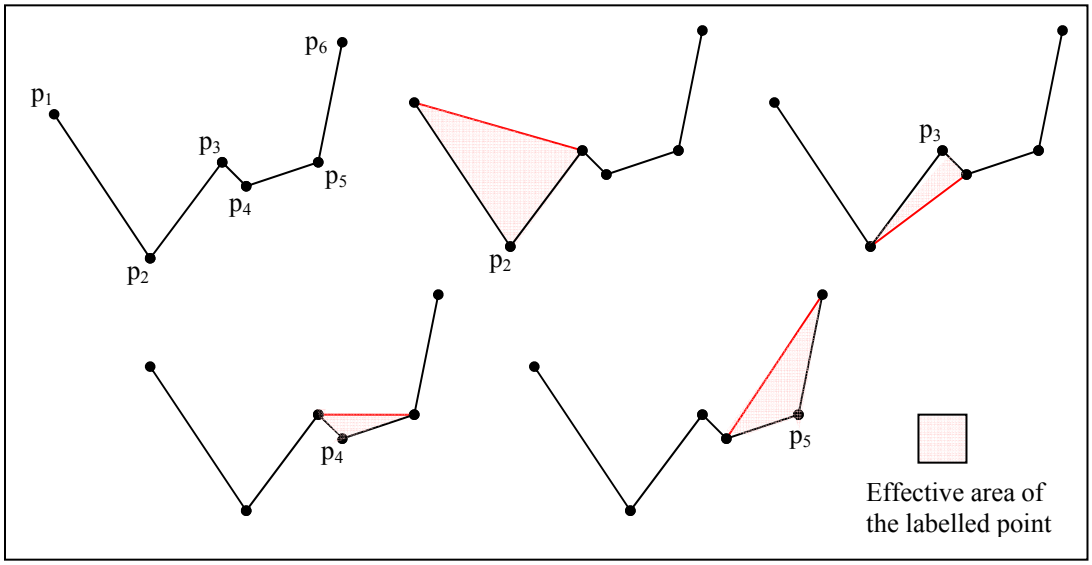


Figure 2.17: A polyline with the effective areas of the points marked. (Figure reproduced from [31])

```
– Find the point with the least effective area and call it
the current point. If its calculated area is less than that of the
last point to be eliminated, use the latter's area instead. (This
ensures that the current point cannot be eliminated without
eliminating previously eliminated points.)

– Delete the current point from the original list and add this to
the new list together with its associated area so that the line may
be filtered at run time.

– Recompute the effective area of the two adjoining points.

 UNTIL
 – The original line consists of only 2 points; the start and end points.
```

For a polygon with N points, $N-1$ ϵs will be generated, and for a line with N points, $N-2$ ϵs. The start and end point of the line are defined as immovable points. After the immovable points have been identified, the remaining points are ranked by the size of their ϵ. The point with the smallest ϵ will be removed first when an approximation is to be made [34].

2.3 Combining Range Search and Generalization

As stated in the introduction, most GIS packages today have the data stored in layers of different levels of resolutions. This leads to a lot of redundant storing and difficulties when updating. The solution presented in this thesis is not the first proposed combined solution to this problem. In the early 1990s Peter van Oosterom, a Dutch computer scientist, described how a combination of three

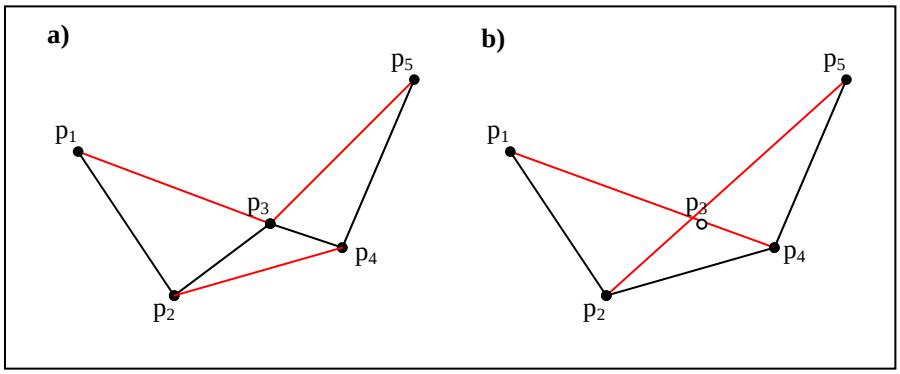


Figure 2.18: The last step in generalizing the line from Figure 2.17 by repeatedly deleting the smallest E. (Figure reproduced from [31])

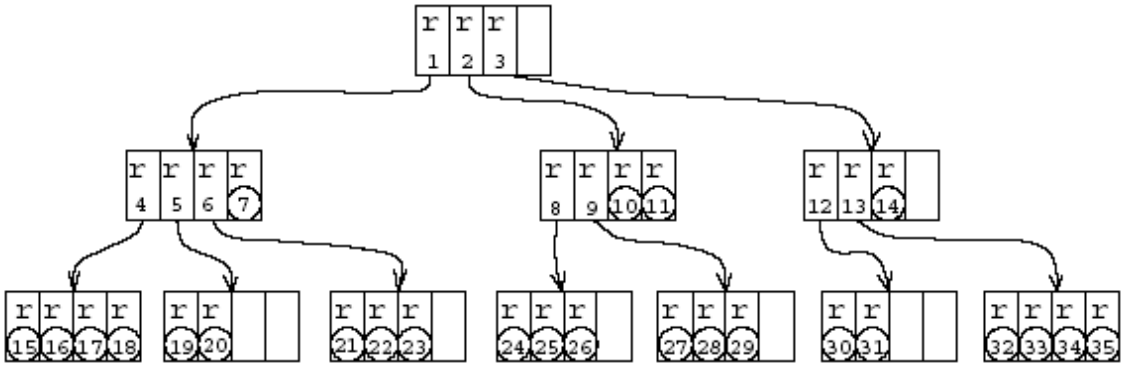


Figure 2.19: An example of a reactive tree. (Figure from [27])

data structures called the *GAP-tree*, the *reactive tree* and the *BLG tree* (see section 2.2.2) can be used to perform a complete generalization of a chosen section of a map [29]. In this approach the BLG tree is used for line simplification, and the reactive tree for selecting or ignoring entire features based on their importance. When performing these two tasks, two problems can occur: Simplifying two lines can lead to overlaps or gaps between neighbouring features, and ignoring entire features will lead to a map full of holes. The GAP-tree is presented as a possible solution to these two problems.

2.3.1 Reactive Tree

The reactive tree is a data structure used for organizing map features according to the result of a generalization evaluation process, where the features are given importance values based on their size and type [27]. The importance values of a feature will vary depending on the function of the map, and this evaluation process is performed by a specialist. The reactive tree is based on the R-tree, so each node can have several children. Figure 2.19 shows an example of a reactive tree.

The reactive tree has two types of entries: object-entries and tree-entries. Both entries have an MBR which is the minimal bounding rectangle and an importance value. In addition the object-entry has an $objectID$ and the tree-entry has a child-pointer which contains the reference to a sub tree. The reactive tree also has a number of properties that must be maintained:

1. For each object-entry, MBR is the smallest axes-parallel rectangle that geometrically contains the represented object of importance $impVal$.

2. For each tree-entry, MBR is the smallest axes-parallel rectangle that geometrically contains all rectangles in the child node, and $impVal$ is the importance of the child node incremented by 1.

3. All the entries contained in nodes on the same level are of equal importance, and more important entries are stored at higher levels.

4. Every node contains between m and M object-entries and/or tree-entries unless it has no brothers (a *pseudo-root*).

5. The root contains at least 2 entries unless it is a leaf.

A search in a reactive tree is carried out by reporting all objects that have an importance value greater or equal to the limit given in the search. The importance limit should be selected so that an equal number of objects are returned in every search; give a high importance limit for large areas and a low limit for small areas. The search algorithm is as follows, starting with the current node being the root of the tree:

```
1. If the importance of the current node, N, is less than the
importance limit, imp, there are no qualifying records in this node
or in any of its subtrees.

2. If the importance of N is greater than or equal to the imp,
report all object-entries in this node that overlap the search
region, S.

3. If the importance of the N is greater than imp, also invoke the
search algorithm for the sub trees that correspond to tree-entries
that overlap S.
```

2.3.2 GAP-tree

The GAP-tree (Generalized Area Partitioning) is used to store an area partitioning hierarchy determining which features will be removed and which will be used to fill the gap of the removed feature. The area partitioning is usually stored in a topological data structure with *nodes*, *edges* and *faces*. The nodes contain their point and a list of references to edges that are connected to the point. The edges contain their polyline, length and references to the left and right face, and the faces contain their weight factor, area and a list of references to edges forming the face's boundary. From this topological data structure, the hierarchy can be constructed as follows:

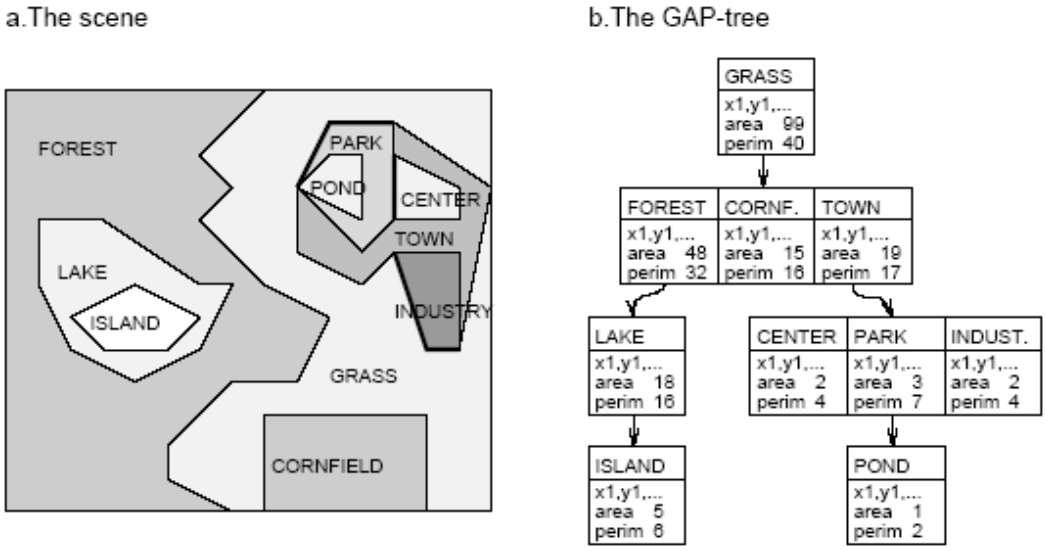


Figure 2.20: The scene and the associated GAP-tree. (Figure from [27])

1. For each face in the topological data structure an "unconnected empty node in the GAP–tree" is created.

2. Remove the least important area feature a from the topological data structure.

3. Use a topological data structure to find the neighbors of a, and determine for every neighbor b the length of their common boundary L(a,b).

4. Fill the gap by selecting the neighbor b with the highest value of the collapse function:

Collapse(a,b) = f(L(a,b), CompatibleTypes(a,b), weight_factor(b))

The function CompatibleTypes(a,b) determines how close the two feature types of a and b are in the feature classification hierarchy associated with the data set. For example, the feature types "tundra" and "trees" are closer together than feature types "tundra" and "industry".

5. Store the polygon and other attributes of face a in its node in the GAP–tree and make a link in the tree from parent b to child a.

6. Adjust the topological data structure, the importance value of b, and the length of common boundaries L(b,c) for every neighbor c of the adjusted face b to the new collapsed situation.

Repeat the steps 2-6 until only one huge area feature is left. This last area feature will become the root of the GAP-tree. Figure 2.20 shows an example of a scene and its associated GAP-tree.

2.3.3 Combining the Three Data Structures

Oosterom's approach does not only concern line simplification, but also other parts of the generalization process like selection, symbolization and combination. The generalization has been developed in the Postgres DPMS environment. Carrying out the generalization process is done by a Postquel query with the proper values for the BLG and the reactive tree depending on the current scale:

```
retreive (blg_pgn2= Blg2Pgn(AreaFeature.shape,"0.01"::float4))
    where AreaFeature.reactive && "(13,40,23,47,2)"::REACTIVE2
    sort by AreaFeature.oid
```

The function $Blg2Pgn$ uses the BLG-tree, and the GAP-tree is reflected by the $sort\ by$ statement in the query.

One problem with using the BLG is that when the map consists of several polylines, one BLG must be created for each of them. This means that when an approximation of only a small area is to be made, still many BLGs may have to be searched, and this will affect the efficiency of using the BLG for line simplification. Another problem with this approach is that the empty areas that may occur only are filled by features higher in the GAP-hierarchy. With this is mind, the question is: What will happen when one wants to extract only a small piece of a large object? Can this approach handle that kind of query? An example of such a situation could be that one wants to display a beach extracted from a global coastline. In the rest of this thesis, one other approach, that can handle this type of query, is presented.

Chapter 3

Using the Priority Search Tree to Solve Computational Geometry Problems

This chapter provides information about the priority search tree (PST) which is used as a basis in the method for combining orthogonal range search and line simplification which is presented in this thesis. The first section describes the PST and the search method normally used with this tree structure; the bucket search. The second section presents one interesting application of the PST and the bucket search. The third section gives an introduction to the adjusted search method described in the following chapter, along with an explanation of why this adjustment is interesting.

3.1 The Priority Search Tree

The PST was discovered in the mid eighties by Edvard T. McCreight [11]. It is a hybrid of a heap and a balanced search tree, suitable for organizing geographical points because it makes it possible to organize the points in two dimensions. It is used for efficiently answering semi-infinite range queries, that is; range queries where at least one of the sides of the range is unbounded.

Heaps are stored in binary trees and are normally used for finding the largest (or smallest) value in a set. However, they can also be used for answering range queries of the type $[q : \infty)$. The query time for this type of query is $O(1 + k)$ where k is the number of values reported. The query is performed by entering the tree and checking whether the root is within the query range (the value is greater than q). If it is, report the point and proceed in both sub trees. As soon as a node is reached that has a value smaller than q, the search is aborted. The search can be aborted at this stage because every node in this node's sub tree will have an even smaller value. Hence, the $O(1 + k)$ query time.

The heap is constructed by selecting the highest value, and storing this in the root of the heap. The rest of the values are divided into two subsets of approximately equal size, and the process is repeated. See an example of a heap in Figure 3.1. When constructing a heap, there are no restrictions as to how the rest of the values are split in half, and this is where McCreight came up with a good idea: Instead of splitting the remaining values into two arbitrary subsets, use the other dimension as a splitting criterion. This way, if the search is unbounded in the upper y-direction, the points in the subset that will make out the left sub tree of the heap all have smaller x-coordinates than the ones in

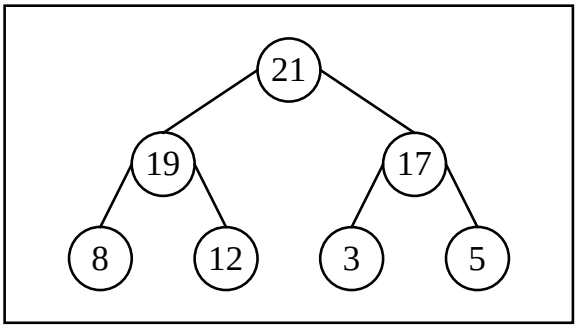


Figure 3.1: Example of a heap where the root of every sub tree has a higher value than all the nodes in the subtree.

the right sub tree, which is the how the values are distributed in a balanced search tree. For a given set of points S, a PST is created as follows [25]:

```
1. If S is empty, return a null pointer.
2. The point P in S with the greatest y-coordinate becomes the root
   R of the PST.
3. If (S-P) is empty, return R.
4. Let Split(P) be a value such that half of the points in (S-P)
   have x-coordinates lower than Split(P), and half higher.
5. Recursively create a PST on the lower half of (S-P), let its root
   be the left child of R.
6. Recursively create a PST on the upper half of (S-P), let its root
   be the right child of R.
```

Figure 3.2 illustrates the construction of a PST, from a set consisting of 11 points. The points that are included in the tree are colored green, the points being chosen next are red, and the points yet to be processed are blue. The black squares indicate $null$ pointers.

The semi-infinite search that is unbounded in the upper y-direction is called a bucket search. In Figure 3.3 one such search is illustrated. The algorithm for answering a bucket search in a PST is as follows [25]:

```
1. If the tree is null, return without finding any points.
2. Let R be the root of the tree, X its x-coordinate, Y its
   y-coordinate and Split(R) the value splitting the x-ranges of the
   nodes in the subtrees of R. Also let the search area be defined by
   X', X'' and Y'.
3. Compare Y to Y'. If (Y < Y'), return without finding any points
   (the root of the tree has the highest y-coordinate).
4. If (X' <= X <= X''), report R.
5. If (X' < Split(R)), the x-range of the left subtree overlap the
   x-range of the query, recursively search the left subtree of R.
6. If (Split(R) < X''), the x-range of the right subtree overlap the
   x-range of the query, recursively search the right subtree of R.
```

In the example of such a bucket search in Figure 3.3, the range of the query is indicated by the red lines forming a “bucket”. The nodes reported are colored red, the nodes visited are blue, and the

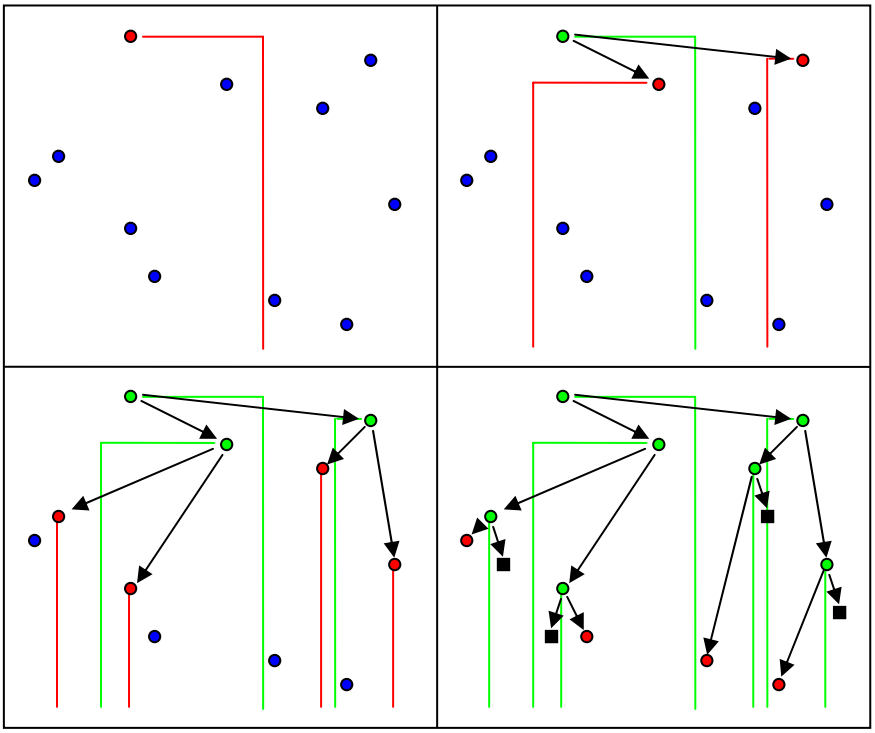


Figure 3.2: Building a PST.

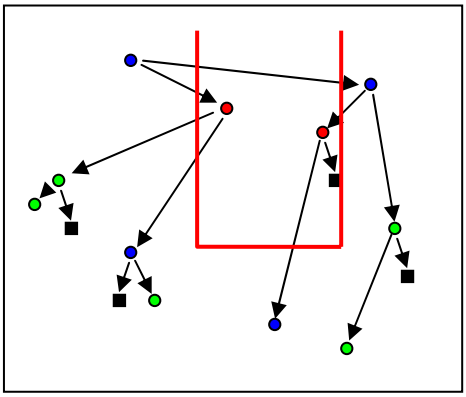


Figure 3.3: Illustrating the concept of a bucket search in a PST.

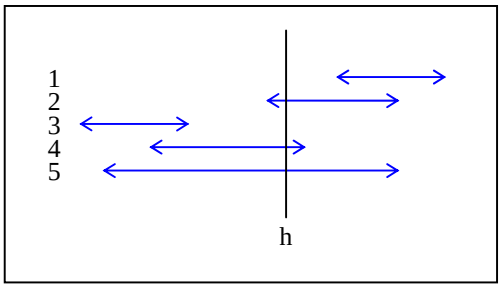


Figure 3.4: An interval stabbing query. The intervals 2, 4 and 5 will be returned from this query, as they are the intervals that contain the stabbing value h.

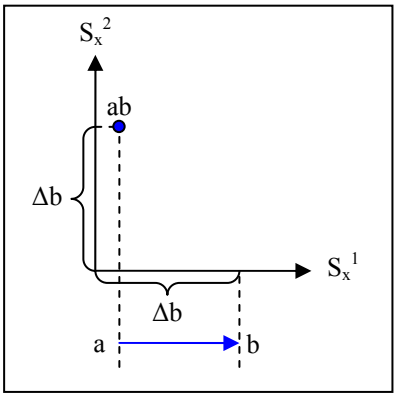


Figure 3.5: How to map an interval with two end points to a point with two coordinates.

nodes not touched by the query are green.

3.2 Interval Stabbing

"Interval stabbing" is the term for maintaining a set of intervals so that for any given value h, all intervals that contain h can be reported efficiently. Figure 3.4 illustrates an interval stabbing query. These queries have a number of applications, and the following paragraph gives a couple of examples.

In 3D computing the interval stabbing query can be used for determining which isosurface is active. Every cell in the scene is given an interval $I = [min : max]$ where min and max are the minimum and maximum among the scalar values of the vertices of the cell. When this is done, a cell is active if and only if its interval contains the isovalue h [22]. In the design of printed circuit boards window queries are performed to inspect small parts in more detail. These window queries can be reduced to interval stabbing queries because the objects on such a board usually consist of line segments parallel to one of the edges of the board or make $45°$ angles with the sides. The line segments that are to be included in the window query will then intersect one or two of the sides of the query range, or they can partially overlap one of the sides [5] (page 212).

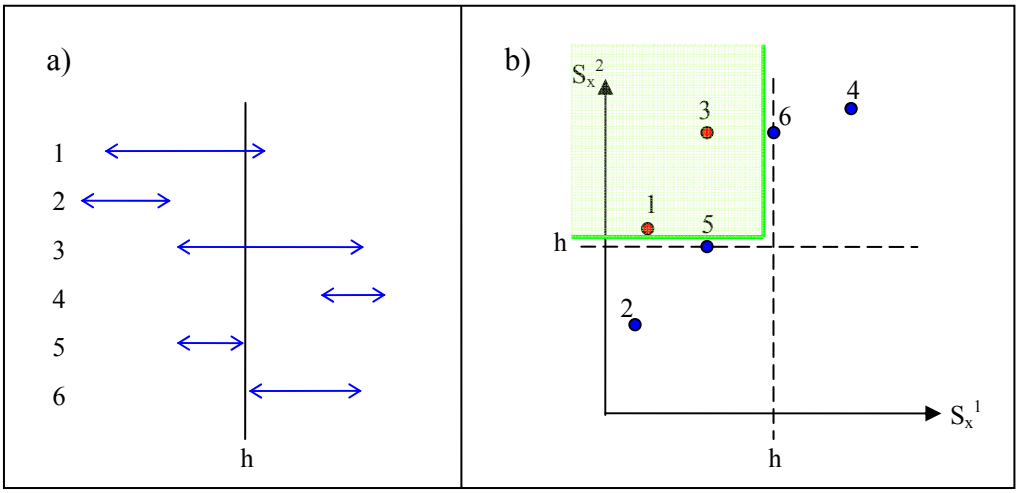


Figure 3.6: Mapping intervals to points, and answering an interval stabbing query using a PST.

McCreight (the designer of the PST) observed that the PST can be used for answering stabbing queries as well as the semi-infinite bucket queries described above. The transformation is simple: map each interval $[a : b]$ to the point (a, b) in the plane. Once this is done, an interval stabbing query with a value h can be performed by formulating a query with the range $(-\infty : hx] \times [hx : +\infty)$, which is a special case of the bucket query already supported by the PST [5] (page 229). Figure 3.5 illustrates how the intervals can be mapped to a point in the 2D plane, and figure 3.6 shows how the transformed bucket query is performed on these points.

3.3 Multiple Interval Stabbing

We discovered that interval stabbing problems also can be useful when performing grid-based line simplification. In these simplification methods the approximation of the line is created on the basis of placing a grid over the map and selecting the points where the original polyline intersect the grid lines. In this case, the grid can be regarded as multiple intervals stabbing segments of the polyline both horizontally and vertically. In Figure 3.7, (a) and (b) illustrate the vertical and the horizontal stabbing values, respectively.

The raster-vector line simplification algorithm developed by Li and Openshaw, described in section 2.2.3, is based on selecting the points where the polyline intersects the boundaries of rasters laid out across the plane to form a grid over the polyline. We realized that if this raster-intersecting problem is split into two separate intersecting problems, one horizontally and one vertically, the problem can actually be solved using a PST.

In the previous section, the single interval stabbing problem was described, and a solution using a transformed bucket query in the PST was presented. In this multiple interval stabbing problem, several such bucket queries must be performed in order to retrieve all the segments of the polyline that are intersected by any of the grid values. If one bucket query is formulated for each of the grid values, some of the segments of the polyline may be reported more than once, as they may

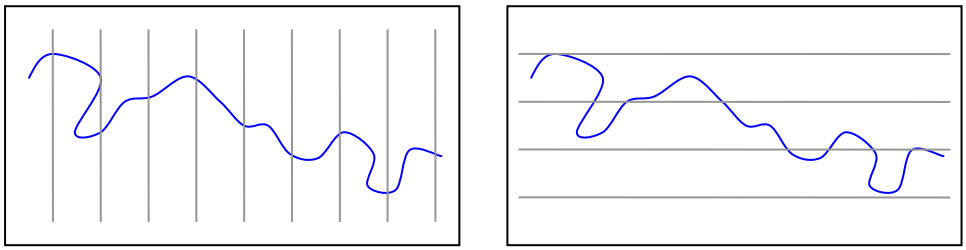


Figure 3.7: Multiple interval stabbing problem. A polyline being stabbed by multiple values both horizontally and vertically.

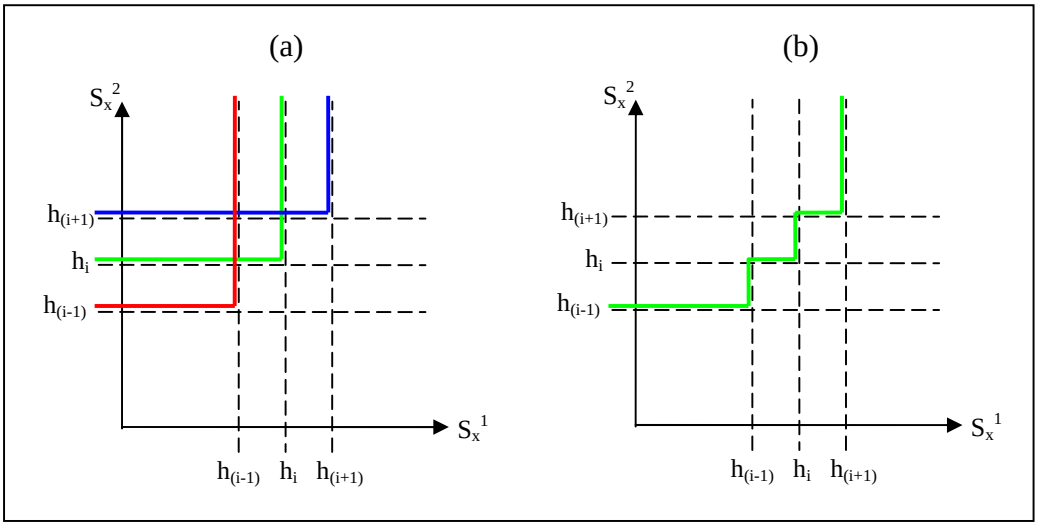


Figure 3.8: Graphical display of the transformed bucket query for multiple intervals in the same plane. (a) multiple instances of the single interval stabbing query from the previous section; (b) the multiple bucket queries are combined into one staircase shaped query.

be intersected by more than one grid value. To solve this problem, another transformation on the bucket query must be made.

Figure 3.8 (a) illustrates how the transformed bucket query will be displayed graphically for multiple intervals. As can be seen in the figure, all intervals that, when mapped to points, are placed above the bucket query for the greatest stabbing interval and to the left of the smallest, will be reported in all the bucket queries. The solution for this problem is to formulate a query that only reports the points that are within the outer boundary of all the bucket queries combined. This new query is carried out very similar to a regular bucket query, but instead of checking against a fixed y-value, the limit in the y-direction depends on where along the x-axis the point currently being checked is positioned.

The size of the rasters in the line simplification algorithm is fixed, and therefore so is the distance between the values in the grid. However, when creating the transformed bucket query, this is not

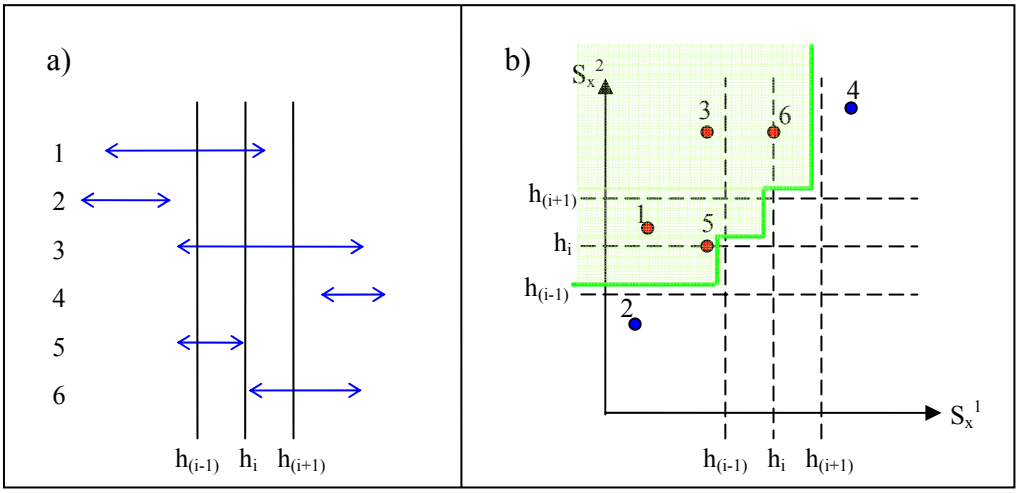


Figure 3.9: Mapping intervals to points, and answering a multiple interval stabbing query using a PST.

necessary. A search with a regular grid is shown in Figure 3.8 (b). Figure 3.9 gives a complete example of mapped intervals and multiple stabbing values put into a coordinate system, and the result when performing a staircase query is that the intervals that are stabbed by any of the stabbing values are reported, and they are reported only once. The fact that the grid does not have to be regular can, for instance, be utilized when creating an approximation of a map where the density of the features is varying.

3.3.1 "Automatic" Orthogonal Range Search

When the grid intersecting problem has been split into two separate problems, horizontal and vertical intersecting, and a staircase query has been performed in both dimensions, automatically, an orthogonal range query has been performed. But unfortunately, too many intervals have been reported. This is because the queries in both direction report every interval that contains one of the stabbing values, but the only once that are interesting are those that are within the range of the grid in both directions. Figure 3.10 illustrates which intervals are reported from these two queries and which must be discarded.

The immediate "brute-force" solution to this problem is to check all the reported intervals when the queries are completed, and keep only those intervals that are reported in both queries. However, this approach is not very efficient, and a smarter solution must be developed.

What is needed at this point is a way to limit the search in the other dimension, to avoid too many intervals being reported in the first place. Inspired by the general two-dimensional design principles, we realized that this limitation actually can be performed before the staircase queries are performed. We came up with a solution where the PST is incorporated in a new two-level data structure. In this structure the first level splits the polyline into ranges along one dimension. Each node in this dimension is then linked to a PST which organizes the intervals from the range of the node according to the other dimension. This data structure is described in the following chapter.

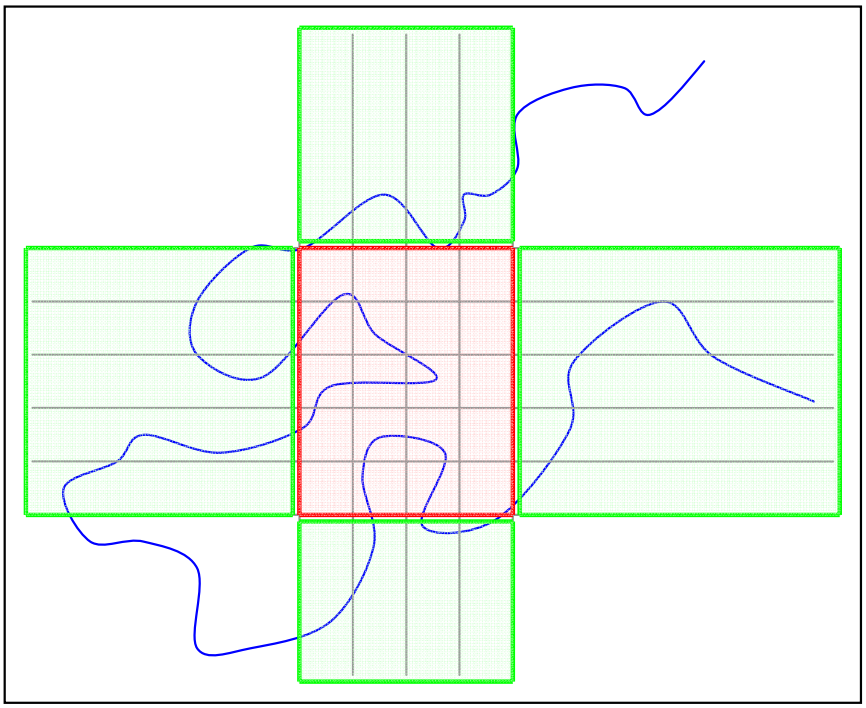


Figure 3.10: Illustrates which intervals are returned from a double staircase query. The red rectangle mark those that are of interest, the green rectangles are reported and must be discarded.

3.4 Summary

To sum up, we found that the PST can be used for more complex types of range search, and that it also can be an important part in solving the combined problem of creating an approximation of the result of the range query. The staircase shaped search algorithm for the PST, which we have developed, is of interest in this context. Thus, the theoretical complexity and practical performance of this type of query must be established.

Chapter 4

Staircase Searching

This chapter describes the data structure developed to solve the problem of reporting too many points from the staircase search. The first section provides a detailed description of the structure. After this there is a section where some theoretical analyses of the complexity of the structure are presented. In order to test the staircase query on sufficiently large data sets, an external implementation of the PST has been implemented. This implementation is described in the third section of this chapter. It is important to keep in mind that it was only developed to be able to run tests with large data sets. Therefore, no measures have been taken to improve the implementation's efficiency in terms of reducing I/O-operations or other. The last section presents empirical tests of the staircase search algorithm.

4.1 The Data Structure

This section describes the new two-level data structure that was developed as a solution to the problem of reporting too many points from the two staircase shaped PST queries presented in section 3.3 in the previous chapter. First a short description of the problem is provided, along with an outline of the proposed solution. This is followed by a section describing the first level of the new tree structure. The next section describes the role of the PST, which is the second level in this new data structure. Finally, the new two-level data structure is presented, where the first level and the second level are combined.

In the previous chapter, an approach to combining range search and line simplification using two staircase shaped searches in a PST was described. The problem with this approach is that the points reported are within a "cross-shaped" area (see Figure 3.10). This is because there are no limitations in the other direction when a staircase search is performed, hence, all points that are within the range in each direction are reported. As briefly mentioned in the previous chapter, the solution to the problem was to incorporate the PST in another, two-level, data structure. This way, the PSTs can be restricted to only contain intervals that are within the correct range in the other dimension. Consequently, the limitation in the other direction is performed before the staircase search is actually carried out.

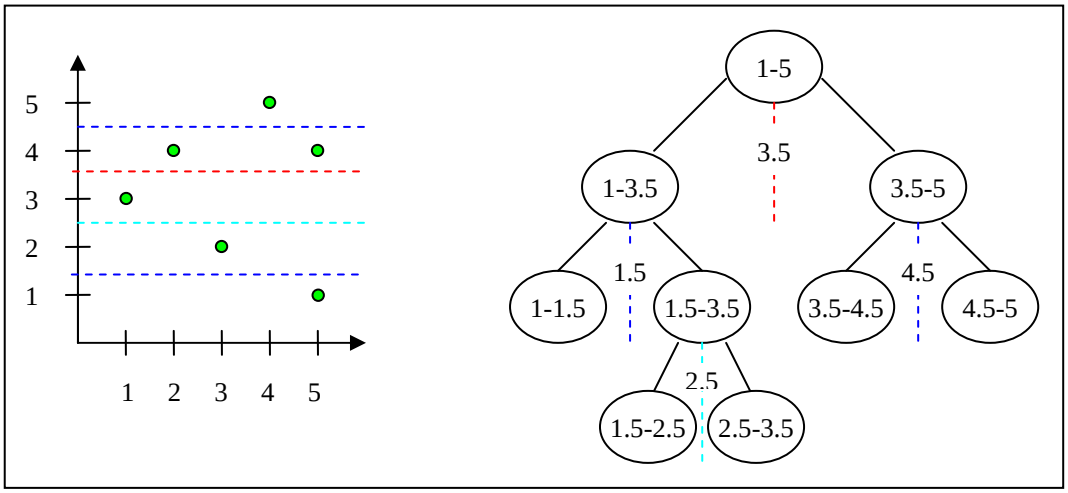


Figure 4.1: An input data set on the left and its corresponding y-range tree on the right. The horizontal lines in the coordinate system are the splitting values for the nodes in the y-range tree.

In a two-level data structure, each node on the first level is connected to another data structure. These other data structures are the second level of the two-level tree structure. We realized that in our range query problem we can recursively divide one dimension into ranges, and store these ranges in nodes in the first level of the tree. These range nodes will then each be connected to a PST, which organizes the data according to the other dimension. This way, the search is performed in two levels; first in the first level searching for nodes that have their entire range within the range of the search, and then in the second level, searching the PSTs that are connected to the first-level nodes that were found.

4.1.1 First Level - The Y-range Tree

In the first level of the two-level tree structure, the points are organized in ranges along the y-axis. Each node in this tree contains two values that are the minimum, min, and maximum, max, of all the y-values in that node's range. The root's min and max are the minimum and maximum of the entire range of y-values in the data set. The root's range is recursively split in half, and stored in the left and right subtrees. The lower half of the values is stored in the left, and the upper half in the right. This gives the root of the left subtree the same min as the root, and max equal to the value where the root's range was split in half. The root of the right subtree gets min equal to the root's split value, and a max equal to the root's max. Figure 4.1 gives an example of such a tree.

The recursive split is continued until there is only one point in each range or until all points in a range have the same y-value. The value at which the subtrees are split in each node must also be stored. This value is vital when a search is performed. The search procedure is described in section 4.1.3 where the entire two-level tree structure is presented.

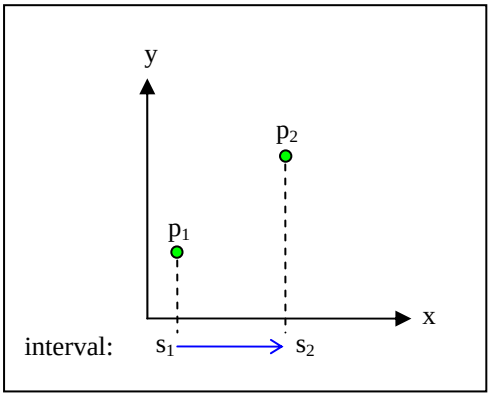


Figure 4.2: Retrieving an interval from two points.

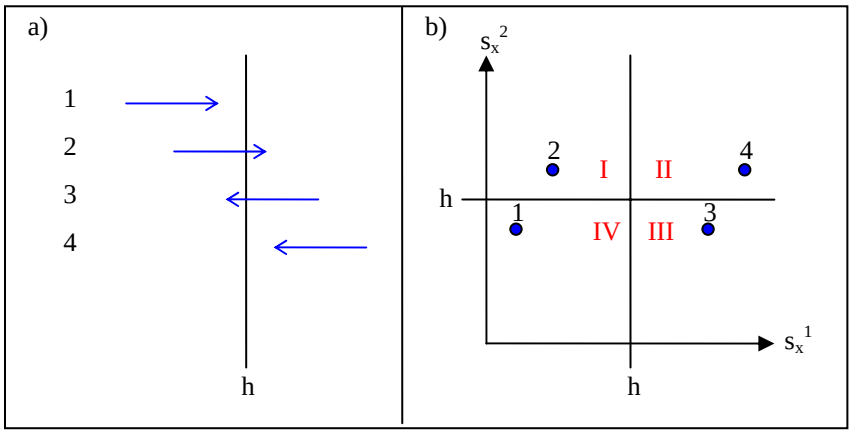


Figure 4.3: Directed intervals mapped to 2D points, using the mapping principle from Figure 3.5

4.1.2 Second Level - The PST

The polylines in the data set are originally stored as points with an x- and a y-coordinate. To be able to utilize the PSTs ability to answer stabbing queries, the polylines must be broken down to intervals. This is done as follows: Instead of keeping points p_1 and p_2, an interval is created by letting the x-coordinate of p_1 be the start value, s_1, of the interval, and the x-coordinate of p_2 be the end value, s_2. Figure 4.2 illustrates this procedure. The intervals that are created must have a reference to the y-coordinates from the original points. They must also have a sequence ID telling where in the original point sequence they where positioned. This information is important because the original points must be recreated when a search has been performed and the resulting polyline is to be presented.

As described in the previous chapter, the retrieved intervals must be mapped to points before they are included in a PST. In the former examples, all intervals had a start value, s_1, that was smaller than the end value, s_2. In real geographical data, this is not always the case. Figure 4.3 illustrates

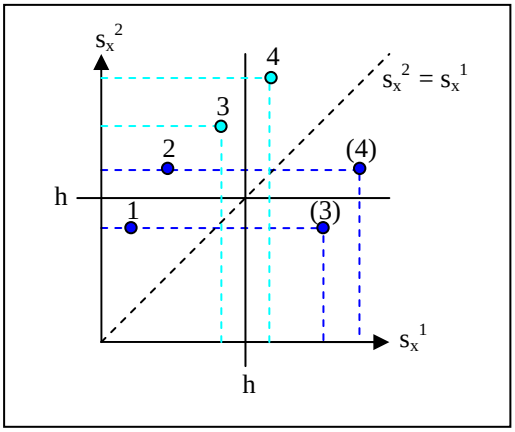


Figure 4.4: Folding the intervals where the $start$ value is greater than the $stop$ value across the diagonal where ${s_x}^1 = {s_x}^2$.

where the intervals end up in the coordinate system after they have been mapped to points, when their direction is taken into consideration. As can be seen in the figure, the points that intersect the stabbing value, h, end up in quadrants I and III, depending on which direction they have. When dealing with undirected intervals, only quadrant I has to be considered in the search in the PST, but as the intervals are directed, also those that end up in quadrant III must be reported from the search. To accomplish this, the intervals that have s_1 greater than s_2 must be folded along the diagonal where ${s_x}^1 = {s_x}^2$. Figure 4.4 illustrates how, when this is done, both the intervals that intersect the stabbing value end up in quadrant I.

Once the intervals have been retrieved and mapped to points, the PST can be built. The construction of a PST is described in the previous chapter, and constructing a PST for the new two-level structure is done in the same way. The only difference is that instead of the y-coordinate determining which point becomes the root of each subtree, the end values of the intervals (the ${s_x}^2$-coordinates of the points) are used. This, however, does not affect the construction procedure once the intervals are retrieved and mapped to points.

Pseudo code for the staircase search in the PST is provided below. x', x'' and $yLow$ limits the original search area. $currentY$ is the y-limit that is calculated. n is the point (interval) currently being checked, and in the first call to this method $n = root$.

```
1. if (n = null) return;
2. if (n.y < yLow) return;
3. calculate currentY;
4. if ((x' <= n.x <= x'') and (n.y > currentY)) report n;
5. if(n.y < currentY) decrement x'';
6. if(x' < n.split) recursively search left (with new x'');
7. if(n.split < x'') recursively search right (with new x'');
```

Instead of checking against a fixed lower y-value, the y-limit must be calculated according to where along the x-axis the point is located. Because the search area has the shape of a staircase, a point near the upper x-limit (x") of the search area must have a higher y-value to be included than

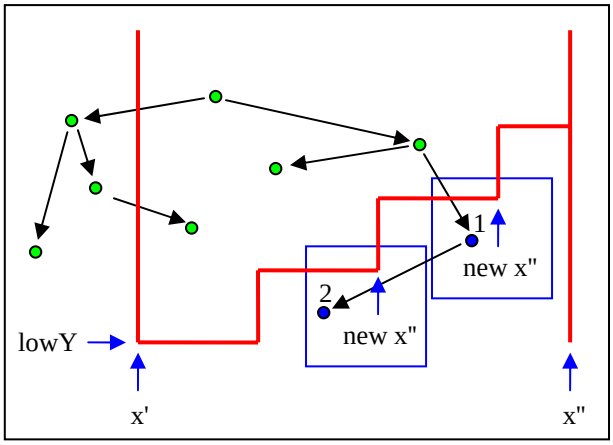


Figure 4.5: The nodes 1 and 2 are within the range of the search, but below the current y-value. When they are reached, x'' can be decremented.

one near the lower limit of the search area (x').

The other difference in the staircase query compared to the bucket query is that when one node is found that is below the search area's lower y-limit, x" can be decremented. In every subtree the root is the node with the highest y-value. This means that when the root is below the current y-limit, all the other nodes in this subtree will be below as well. Because of this, the x" can be reduced to whatever was the last vertical value in the staircase. See Figure 4.5 for a graphic display of this decrement.

Figures 4.6 and 4.7 illustrate the difference in the number of nodes visited in a search algorithm where x" is decremented, and one where it is not. The red nodes are the ones that are reported from the search, the yellow nodes are visited but not reported, and the green nodes are not affected by the search at all.

4.1.3 Combining the Two Levels - The 2D Range Simplification Tree

The 2D Range Simplification Tree is a two-level tree combining the two levels described in the two previous sections. The ranges on the y-values each are connected to another binary tree which is a PST. In these PSTs there are some restrictions as to which intervals are included. Only the intervals that originally were retrieved from points that had a y-coordinate within the y-range of the node the PST is connected to, are included. Figure 4.8 illustrates how this tree structure is constructed. The original geographical dataset is shown in the upper left corner. The horizontal lines represent the values where the ranges on the y-values are split in half, and these lines can also be seen in the model of the tree structure in the upper right corner. The PSTs are represented by triangles attached to the nodes in the first level y-range tree, and they are also shown below the data set and the y-range tree, in the second "row" of the figure.

A search in this structure is performed by recursively checking the nodes in the y-range tree. When one node that has its entire range within the y-range of the search is found, a staircase search is performed in the PST that is connected to this node. From the search in this PST, all the points

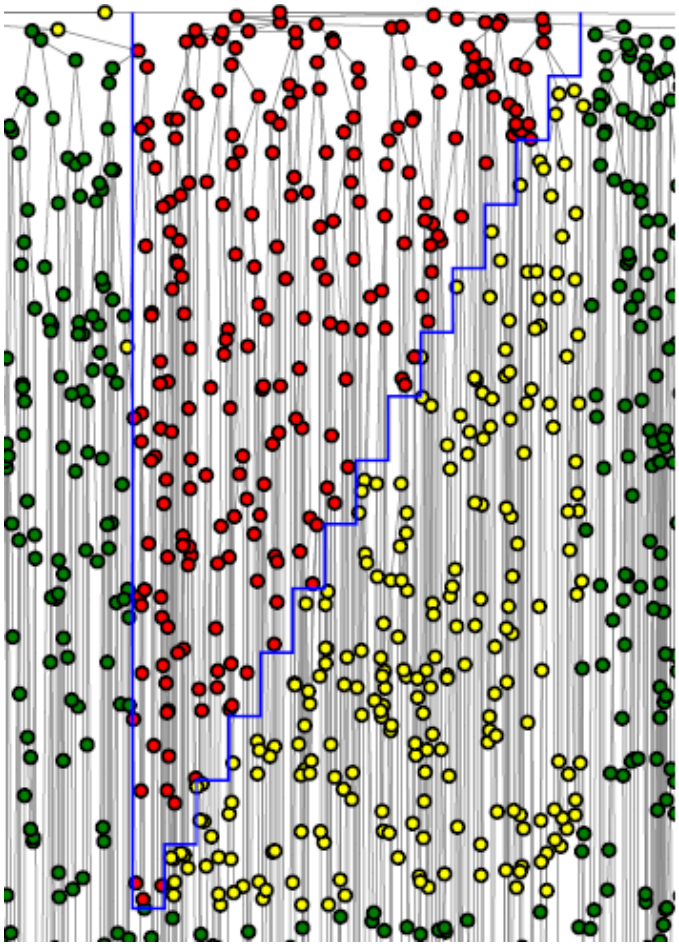


Figure 4.6: If the x” is not decremented, the search visits the same number of nodes as if it had been a bucket search. The red nodes are reported, the yellow are visited, the green are not affected by the search.

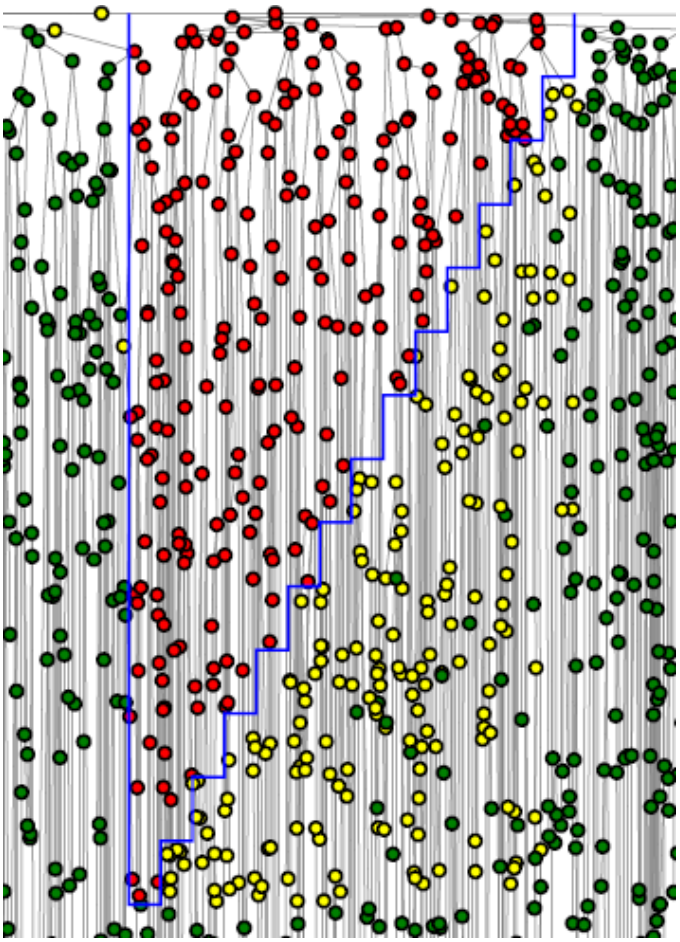


Figure 4.7: When the x" is decremented, the number of visited nodes is reduced. The red nodes are reported, the yellow are visited, the green are not affected by the search.

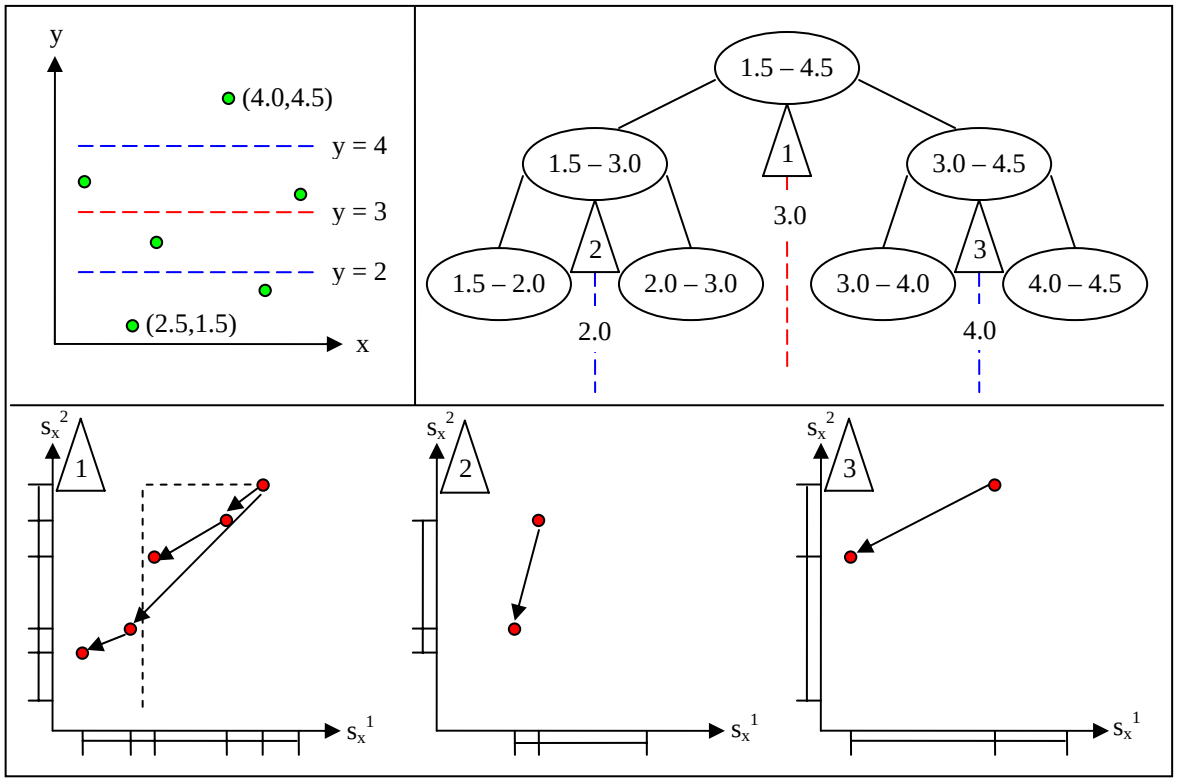


Figure 4.8: Building the 2D Range Simplification Tree. In the upper left corner is the input data set, and to the right is the y-range tree. Below is each of the PSTs that are connected to the nodes in the y-range tree.

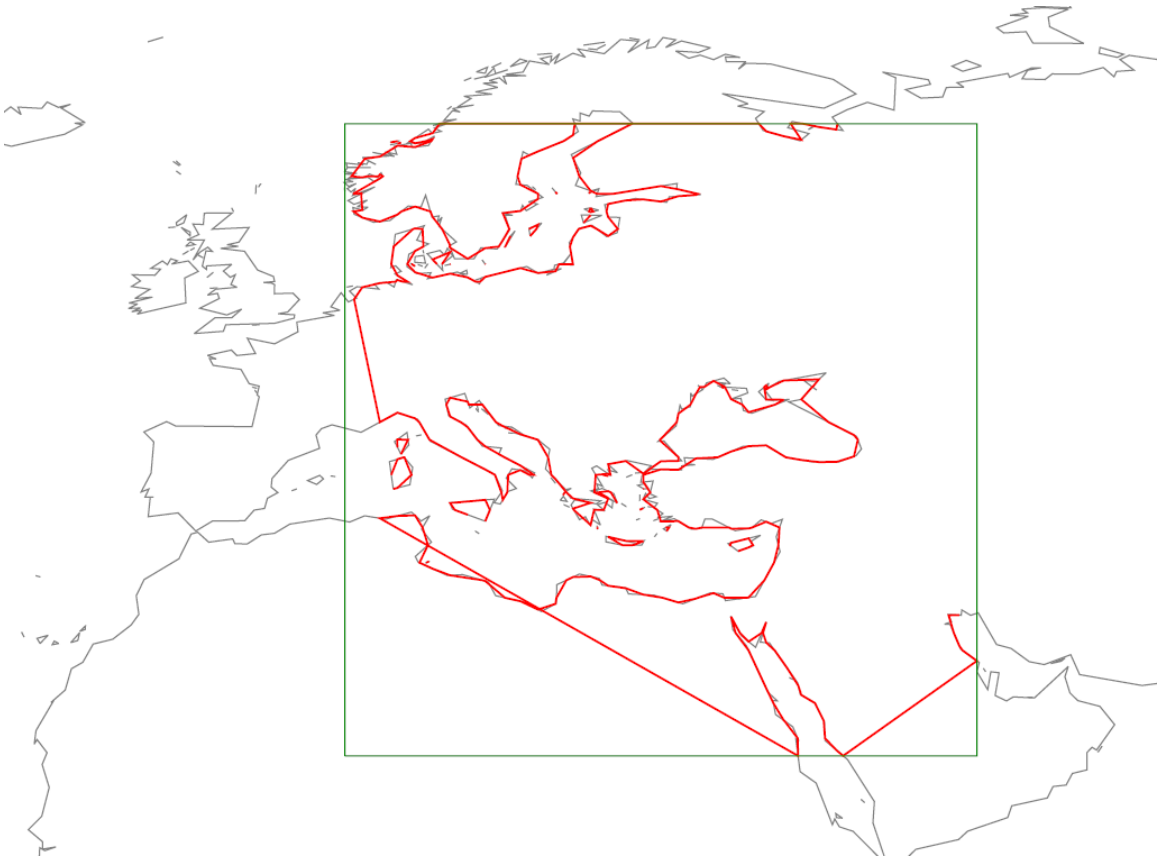


Figure 4.9: The result from a query in a 2D Range Simplification Tree.

that intersect one of the stabbing values in the grid are reported. The recursion in the y-range tree is carried out as long as the current node does not have its entire range within the range of the search. The left subtree is searched as long as the minimum y-value of the range of the search is lower than the node's split value, and the right subtree is searched as long as the maximum y-value of the range of the search is higher than the node's split value. The pseudo code for this search algorithm is provided below. n is the node currently being checked, and $n = root$ in the initial call to this method. $ymin$ and $ymax$ are the upper and lower y-limits of the orthogonal search area. $n.min$ and $n.max$ are the upper and lower y-limits of the range stored in the node n, and $n.split$ is the value at which n's subtrees are split in half.

```
1. if (n = null) return;
2. if ((n.min >= ymin) and (n.max <= ymax))
3.       search n.PST
4. else // n's range is too wide
5.       if (ymin < n.split) search left subtree
6.       if (ymax > n.split) search right subtree
```

Figure 4.9 shows an example of output from performing a query in a 2D Range Simplification

Tree.

4.1.4 General Problem

Describe the fact that this can be reduced to a general problem, where points with two coordinates can be reported and they can be limited using another value. For instance, weather reports, pollution analyses and so on.

4.2 Theoretical Analyses

This section provides analyses of the theoretical complexity of building, storing and searching the 2D Range Simplification Tree.

4.2.1 Building

The 2D Range Simplification Tree is a two-level tree structure, and therefore, there are two separate tree structures that must be built. The complexity of building the 2D Range Simplification Tree is the combined complexity of constructing each of the two levels. In this section the analysis of these complexities is presented. First the complexity of building the first level, the y-range tree, is described. Second the process of constructing the PST is analyzed, and finally the two are combined in the complexity of constructing the entire two-level tree.

Lemma 4.2.1. *A 2D Range Simplification Tree can be constructed in $O(N^2 log^2 N)$ operations.*

When building the first level, which is the y-range tree, the first process that has to be carried out is sorting the points in the data set according to their y-coordinates. The lower bound for sorting is $O(NlogN)$, where N is the number of points in the data set. Once the points are sorted, the data set is recursively split in half until only one point remains in each node. This is done in $O(N)$ operations. When the two operations are combined, the complexity becomes $O(NlogN + N)$. The sorting is the dominant factor, and, hence, the complexity of building the y-range tree is $O(NlogN)$.

The second level of the 2D Range Simplification Tree is the PST. The PST is built by sorting the points in the data set according to their x-coordinates, which, as stated above, has a lower bound of $O(NlogN)$. After the points are sorted, the data set is recursively split in half. For every split, the point with the largest y-coordinate must be found, because this is the point that will be stored in the root of the current subtree. Since the PST has height $O(logN)$, and there are N points that must be searched through at every level to find the roots, this operation takes $O(NlogN)$ operations. With the sorting and the splitting combined, constructing the PST is done in $O(2NlogN)$ operations. The constant is ignored, and $O(NlogN)$ remains.

When the y-range tree takes $O(NlogN)$ operations to construct, and the PSTs also take $O(NlogN)$ operations to construct, the combined construction complexity becomes $O(N^2 log^2 N)$. To reach the desired bound of $O(NlogN)$ construction complexity for the 2D Range Simplification Tree, some issues concerning the construction of the PSTs must be solved. These issues are discussed in chapter 6.

4.2.2 Storage

Lemma 4.2.2. *A 2D Range Simplification Tree requires $O(NlogN)$ storage.*

All the nodes in the data set are stored as intervals in exactly one of the PSTs at every level of the 2D Range Simplification Tree. This means that for each level of the tree, the PSTs use $O(N)$ storage. The depth of tree is $O(logN)$, and, hence, the required storage for the 2D Range Simplification Tree is $O(NlogN)$.

4.2.3 Searching

Determining the exact complexity of querying the 2D Range Simplification Tree, is not trivial. One issue is that the search has an output sensitive complexity. This means that the number of nodes visited depends on the number of nodes reported. Another issue is that the theoretical complexity of the staircase search in the PST is difficult to determine.

In the y-range tree, $O(logN + k)$ nodes are visited, where k is the number of range nodes where the PSTs must be searched. A bucket search in a PST also has a complexity of $O(logN + k)$, where k is the number of nodes reported [25]. In the staircase query, the case is a bit different. As a starting point we could say that the complexity of a staircase search is the same as for a bucket search that has the same width as the widest part of the staircase area. However, because of the decrement of the x'' in the query (described in section 4.1.2), the complexity is a bit better than this. It does not reach $O(logN + k)$, but it is difficult to determine the exact complexity of this query.

When we combine the $O(logN + k)$ of the y-range tree, and the close to $O(logN + k)$ of the PST, we end up with approximately $O(log^2N + k)$. Unfortunately, because of the inaccuracy in determining the complexity of the staircase search, this is not the final complexity of a search in a 2D Range Simplification Tree. Nevertheless, it is probably the closest we can get in this theoretical analysis.

4.3 Implementation

This section describes the process of externally building and searching the two level tree structure presented in the first section of this chapter. The section starts with an introduction to external memory algorithms followed by an explanation of how the different parts of the construction and search algorithms have been externalized. After this there is a subsection that describes the actual implementation of the data structure.

4.3.1 External Memory Algorithms

In an increasing number of problem areas in the field of computer science, the amount of data to be processed is too massive to fit in the computer's internal memory. Examples of such areas are computer graphics, geological and meteorological databases and GIS. When a system involves massive data sets that are too large to fit in the computer's internal memory, the algorithms must be adjusted to use external memory.

In computers today the different types of memory often are organized in hierarchies like the one in Figure 4.10. The further away from the CPU, the larger and slower are the storage 'units'. Accessing the internal memory is fast, but the amount of this type of memory is limited because it is also expensive. Disk memory, however, is cheap and the amount of this type of memory can reach several terabytes. Unfortunately, accessing disk memory is extremely slow compared to accessing the internal memory (it is actually approximately one million times slower [21]). Therefore, the main objective of external algorithms is to minimize the number of disk accesses [17].

In ordinary in-memory algorithms, the efficiency is determined by comparing the number of calculations performed to the size of the problem. In an external memory algorithm, however, this measurement will probably make no sense. This is because accessing elements in external memory takes so much more time that the number of disk accesses becomes the dominant part of the algorithm in terms of efficiency.

A lot of research has been conducted in the field of external algorithms, but not all of it is backed up by implementations or test results. In many cases this is because the proposed theories are so complex that developing a functional prototype would be extremely difficult and probably too resource-demanding. With this in mind, the present project does not intend to revolutionize the area of external algorithms. Instead, the main goal for this project is to implement a functioning prototype of the previously implemented data structure, in order to facilitate the extensive testing of large amounts of data.

4.3.2 External Tree Structures

Building a two-dimensional binary tree involves a number of processes like sorting, selecting and storing. In an internal tree, the selection and storage processes are rather straightforward, but in an external tree, the type of storage used must also be taken into consideration. The data can be stored on tapes or on disks, and as sequential or random access files. In the present prototype the data are stored on disk and in random access files.

Random access files were chosen for this project because they can be treated similar to in-memory arrays. In-memory arrays provide direct access to the elements, and so does random access files in a way. These files have a method that allows reading from any byte on the file, which means that as long as the size of the objects is known, direct access to any object is provided. The operation

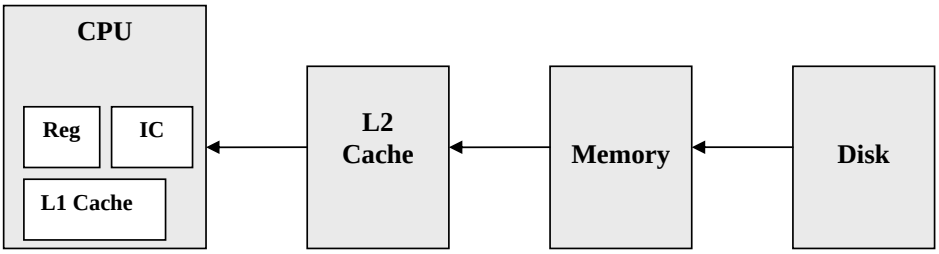


Figure 4.10: Memory hierarchy of a single processor system, including registers, instruction cache, data cache (level 1 cache), level 2 cache, internal memory and disk.

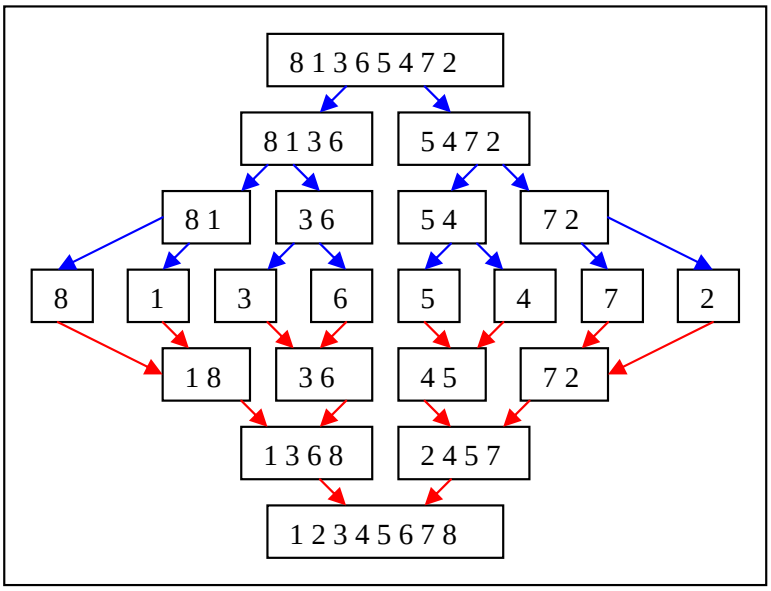


Figure 4.11: Illustrating two-way merge sort.

that allows reading from any byte on the file is called a seek operation. This seek operation always starts the seeking from the beginning of the file. The random access files also have a skip method where any given number of bytes can be skipped. Where this is practical, the time spent seeking in the file can be reduced by using this skip method.

External Sorting

One of the main problems that need to be solved efficiently when building a binary tree externally is sorting. In multi-level tree structures this is even more important as the points are sorted in several different ways. In the two level tree structure described here, the points are first sorted according to the y-coordinates in the first dimension, then according to the x-coordinate in the second dimension, and finally according to curves and indexes.

In this process an inefficient sorting algorithm will be a substantial bottleneck, and will impair the performance of the entire structure. External sorting is the term for sorting elements that are stored as mentioned above, while internal sorting describes sorting elements in an array in RAM. The main concern when developing an external sorting algorithm is to limit the number of disk accesses since this takes more time than accessing an item in RAM.

Most external sorting algorithms are based on merge sort. The principle of the merge sort algorithms is “divide and conquer”. In an internal merge sort algorithm, the data are recursively split in half and stored in separate arrays. When only one element remains in each array, the arrays are merged in pairs until all the data once again are gathered in one array, see Figure 4.11.

The external merge sort algorithms generally split the original large data file into shorter run lists that are sorted internally using an efficient internal sorting algorithm. The run lists are stored on disk and merged until only one large file remains. This process can be a two-way merge or a multiway

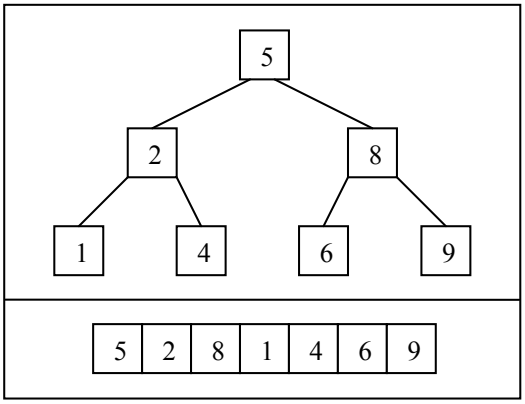


Figure 4.12: A binary tree with its corresponding array representation.

merge. In a two-way merge, two and two files are merged into one larger file. In a multiway merge multiple files can be merged at each "level". The two-way merge is the simpler method, while the multiway merge can be a more efficient approach as the number of merges can be reduced.

Selecting

Selecting an element in a random access file involves a seek operation in order to set the file pointer to the desired index, followed by a read operation. When the file pointer is placed at the correct index, the object can be read. The only way to read an object of a given type is to know the size of the object, create a byte array of the same size, and fill this array with bytes read from the file. When the bytes that constitute the object have been read, the array must be converted to an object of the correct type. Only when the byte array has been converted to an object is it possible to verify whether this was the wanted object or not. If not, the process of reading and converting must be repeated until the object is found.

When building a PST, the original objects are stored on a file and sorted using their x-value. To find the root, which is the object with the largest y-value, all the nodes must be checked. To find the roots of the sub trees the process is repeated recursively on the lower and the upper half of the original file.

To limit the number of I/O-operations, most external memory algorithms are based on reading blocks of data from the disk. Each block contains B objects, and accessing one disk block requires only one seek operation. The size of the blocks should be as large as possible, without exceeding the amount of internal memory that can be dedicated to this process.

Storage

The tree must also be stored externally, which requires an array representation. Figure 4.12 displays a simple example of a balanced binary tree and its corresponding array representation. The root is stored at index 0, and the root of the left and right sub trees are stored at index $(2 * 0 + 1) = 1$ and

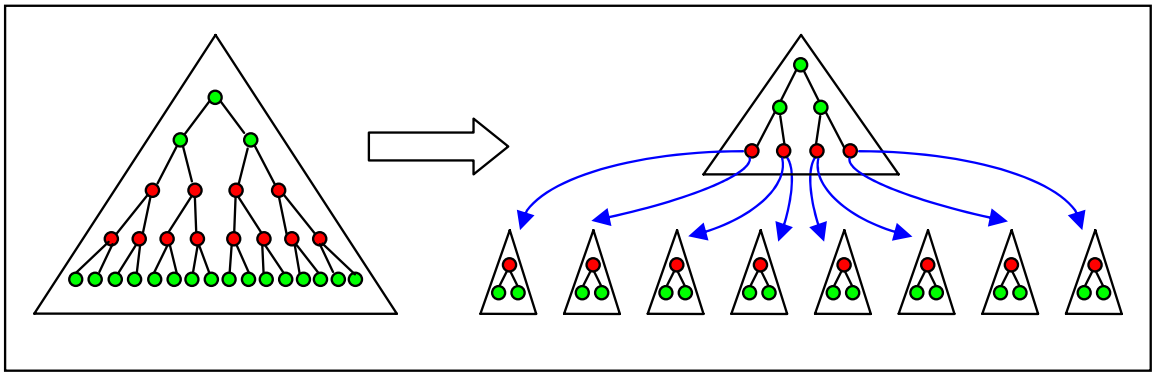


Figure 4.13: Splitting one large tree into sub trees stored on separate files.

$(2 * 0 + 2) = 2$ respectively. The left child of the node at index i is stored at index $(2i + 1)$ and the right child at $(2i + 2)$.

Storing a one-dimensional binary tree like this is manageable as these trees consist of only one type of object, and objects of the same type have the same size. When the case is expanded to the storing of a two-dimensional binary tree, there will be problems because the second dimension includes trees of variable sizes. One solution is to store every second dimension tree in a separate file, and have implicit pointers to these files, using, for instance, the index of the "parent node" in the first dimension. This, however, will not be practical when the data set contains a large number of coordinates. If building the tree structure means creating tens of millions of new files, the computer probably will get into trouble. A solution to this might be to store all the internal PSTs in one large file, the pointers in the external nodes pointing to the index in this large file where the PST for that node is stored.

When the original data set is large, the file where the tree is stored will be extremely large as well, and the question is whether it might be more efficient to split the large file into several smaller files, see Figure 4.13. This can be done either during the process of building the tree, or by using a routine that runs through the tree after it has been built. This writes the nodes to different files when given "split level" are reached. For the present prototype the latter approach was chosen. The nodes on the last level of each file must then have an indicator saying that the next level is stored on another file. The filename must be something that can be recreated in the search algorithms, like the index the root node in the new file would have had if it had been stored in the original tree-file.

Searching

Performing an external search is very similar to performing an internal search. The biggest difference is that when the tree is stored in an internal array, direct access to the nodes is provided. When the tree is stored on a file, however, it is necessary to seek or skip bytes to get to the point where the nodes you want are stored.

The technique mentioned earlier where blocks of data are read to improve efficiency is difficult to use when searching an externally sorted tree. This is because the tree is laid out in level-order on

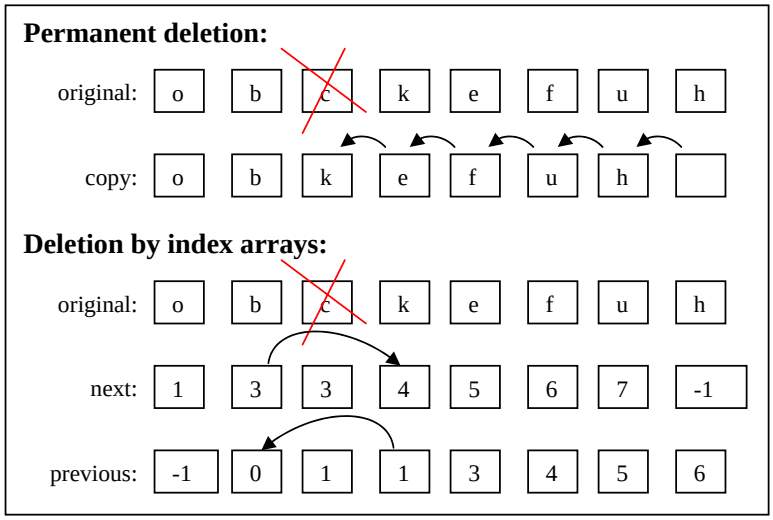


Figure 4.14: Different methods for deleting an element in an array.

the file, while search algorithms are based on pre-order traversal of each sub tree.

When reading blocks of data is out of the question, one operation of seeking, reading and converting must be performed for every node in the tree that needs to be checked. This means that the only way of improving efficiency when performing a search in an externally stored tree is to limit the number of nodes visited for each reported.

4.3.3 Externalizing the 2D Range Simplification Tree

This section describes the techniques, the problems and the solutions from the implementation of the external two level tree structure presented in the first section of this chapter. First the implementation of PST is presented, followed by the complete two-dimensional tree structure.

The PSTs

When building a PST, the first process that needs to be carried out is sorting the coordinates according to their s_1-value. The s_1-value is what would have been the x-value if the coordinates had been ordinary coordinates with an x-value and a y-value instead of intervals along the x-axis with a $start$- and a $stop$-value. This sorting must be performed externally, and any external sorting algorithm can be used. Merge sort has been used in the prototype, but without any particular focus on reducing the number of I/O operations. The goal is to build the tree once and dynamically update it as the geographical data is updated. Because of this, and because the prototype is mainly developed only to facilitate extensive testing, the efficiency of the sorting algorithm has had low priority in the implementation.

After the coordinates have been sorted, the node with the largest s_2-value ("y-value") must be found. This node will be the root of the PST. At this point it is wise to read blocks of data, since every node in the file must be checked to find the one with the largest s_2-value. When the node with

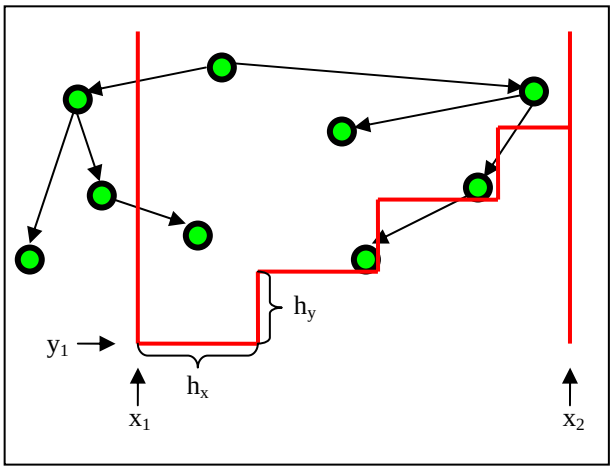


Figure 4.15: Display of the different values defining the search area in a staircase shaped search in a PST.

the largest s_2-value has been made the root of the PST, this node must be deleted from the file to avoid it being added to the PST more than once. The process of permanently deleting all nodes from a file or an array one after another has a worst case order of $O(N2)$, as the remaining nodes must be "moved down" to fill the gap. To avoid this, one can implement an alternative "deletion"-method using index files or arrays to hold a 'next'- and a 'previous'-pointer for every index in the original file or array. Thus, when "deleting" a node, the only thing that is done is marking the node 'deleted', and skip the node in the list by updating the next and previous pointers. Figure 4.14 illustrates these two different deletion methods.

The rest of the PST is constructed by recursively repeating the selecting and deleting on the left and right halves of the file. The completed PST is stored in level order in a random access file. The root is stored at index 0, while the left and right children of each node at index i are stored at index $(2i + 1)$ and $(2i + 2)$.

The search area in a PST is defined by the values x_1, x_2 and y_1 and two values, h_y and h_x, to update the y_1 and x_2 values so the search area gets a staircase shape. h_y and h_x indicate how much y_1 will increase and on what intervals of x, see Figure 4.15.

The search is started by reading the root node from the file. As the tree is stored in level order, the root is the first node on the file. Next, the current y_1-value is calculated based on the original y_1-value and the h_y- and h_x-values which define the staircase-shape. If the root's s_2-value is lower than the current y_1-value, every node in the PST will fall outside the search area because the root is the node with the highest s_2-value. This stops the search. If the search continues, the root's s_1-value is checked. If it is between x_1 and x_2, or equal to one of them, the coordinate is reported. Then the recursive search continues. If the root's s_1 value is larger than x_1, the left sub tree is searched, and if the root's s_1 value is smaller than x_2, the right sub tree is searched. The search continues until all nodes are searched, or the search has been stopped because the s_2-value is below y_1.

The Complete 2D Range Simplification Tree

The complete two-dimensional tree structure organizes the geographical coordinates according to their y-value in the first dimension, and according to their x-value in the second dimension. The two dimensions consist of two different types of nodes:

The nodes in the first dimension have two float values limiting the node's range along the y-axis, and a split value deciding which y-values belong in its left sub tree and which belong in its right. In addition it has a reference to the PST connected to the node. In the external version of the structure, the PST reference must be replaced by a reference to the file where the PST is stored.

The nodes in the second dimension (the PSTs) are originally coordinates with an x- and a y-value, but are combined, two and two, to intervals along the x-axis with a reference to a node storing their original y-values. In addition to the interval and a $split$ value, these nodes also have information of where in the original coordinate sequence they belong, and a pointer to the node which stores the y-values.

Building the two-dimensional tree structure is relatively easy once the PST is implemented. As can be seen in section 4.1, the outer dimension contains ranges along the y-axis, and the root of the tree holds a range containing all y-values in the data set. The root is connected to a PST containing all the intervals along the x-axis.

The process starts with the coordinates being sorted according to their y-value. After this the range is recursively split in half on every level of the outer dimension. The left child of each node gets a range of the lower half of its parent's y-values, and the right child the upper y-values. The PSTs for these children are constructed from all x-values that originally had y-values within the range of the node the PST is connected to. The range on the y-values is split in half until only one node is left within each range.

The PSTs must be stored in one big file because when testing the algorithm on data sets of 10 million coordinates, having one file for each PST becomes impractical. This leads to a few issues that need to be solved:

First of all, it is necessary to keep track over where in the file each PST is stored and how many "indexes" that are occupied by every PST. This information is stored in the nodes in the outer dimension. Therefore, instead of only having a reference to the file where the PST, these nodes must have two long values; one to know the "startindex" of this node's PST, and one to know which index is the last one occupied by this PST.

The formula used for storing a binary tree as a list ($(2i+1)$ and $(2i+2)$) uses every index when the tree is complete and the root is stored at index 0. When the root is stored at another index, this will lead to many empty indexes. To avoid this, the new formula for storing the left child will be $(2*(i-startindex)+1)+startindex$, and the right child $(2*(i-startindex)+2)+startindex$.

The PSTs must be stored sequentially in the file because there is no way of knowing how much space each PST will need. This means that in the process of building the tree structure, one must always keep track of the greatest "index" that is occupied in the file where the PSTs are stored. Figure 4.16 illustrates this problem. In the construction method, the index where the PST is to be stored is provided when the method is called, in the variable $pstIndex$:

```
private long makeTree(float min, float max, long pstIndex)
```

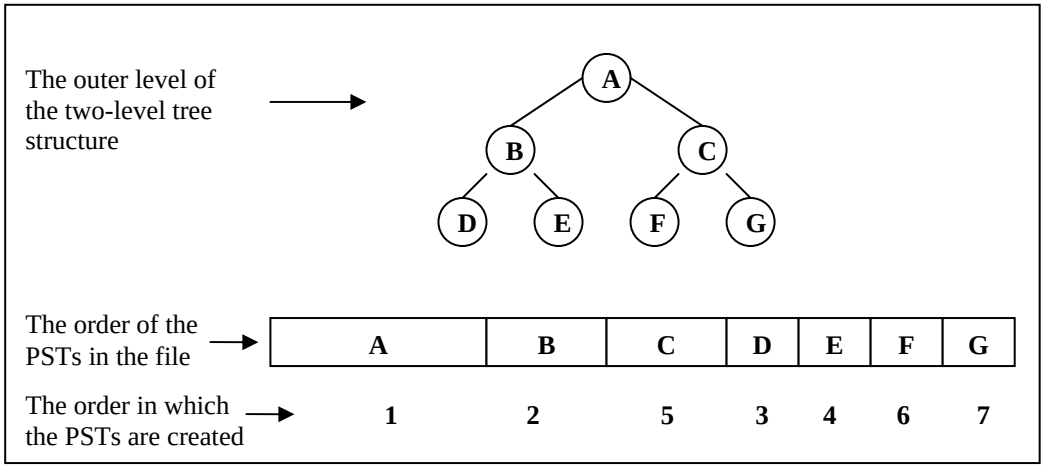


Figure 4.16: The PSTs are created recursively, but stored sequentially in the file.

```
   min – the lower end of this node's range
   max – the upper end of the range
   pstIndex – the index in the PST file where this node's PST is to be stored
```

When the left sub tree is created, the $pstIndex$ is easy to find because the left subtree is created first. The PST for the root of the left subtree can therefore be stored at the last index occupied by the PST just created + 1. Consequently, this last index occupied must also be passed on in the construction method. The value is stored in the variable $maxPstIndex$ which is also returned to be used when the call-stack is popped:

```
private long makeTree(float min, float max, long pstIndex) {
   long maxPstIndex = 0;
   <create current node>
   maxPstIndex = makeTree(min, newMax, <last index for this PST + 1>);
   maxPstIndex = makeTree(newMin, max, maxPstIndex + 1);
   return maxPstIndex;
}
```

The process that builds the PSTs that are connected to the nodes in the outer level returns the greatest index used by that particular PST. When the left subtree is created, this index $+1$ is used as the $pstIndex$. When the right subtree is created, the $maxPstIndex + 1$ returned from the left subtree process is used as the $pstIndex$. The $pstIndex$ and the $maxPstIndex$ for each PST is stored in the outer level node the PST is connected to, and are used in the search procedure.

The search is performed pre-order by iterating the sub trees until one node is reached that has its entire range within the y-range of the search area. When one such node is found, then this node's PST is searched to find the coordinates that have an x-value within the x-range of the search area. Because the PST that is connected to a node in the outer dimension only contains points that originally had a y-value within the range of the outer node, the nodes in this PST that are within the x-range of the search area, will also be within the y-range.

Because the PSTs are stored in one big file, one will in most cases find a node that belongs to another PST instead of a $null$ node when the current node is a leaf that should have no children. However, the nodes in the outer dimension "know" which index the root of their PST is stored at, and how many nodes this PST occupies. This means that the recursive search algorithm is terminated when the index reached is beyond the $maxPstIndex$ stored in the outer node, and not when $null$ is returned.

Summary

The external implementation described in this section is by no means a satisfactory external implementation of the data structure described in section 4.1. Nevertheless, it serves the purpose of giving the possibility to test the structure on substantially larger data sets than what is possible using only internal memory. It was also important to see that a functioning external implementation could be made, that actually is useful. In the next section the tests that have been carried out are described.

4.4 Testing

The reasons for developing the naive data structure prototype described in the previous section was to facilitate tests on large data sets to determine its efficiency. This section describes the data sets used for testing, and the hardware on the machine that carried out the tests. Then the tests and the results are presented and discussed.

4.4.1 The Data Sets

In order to establish the efficiency of this data structure, tests on large data sets must be performed. Because real life data sets seldom have an evenly distributed point density, random sets were created for the initial testing. These data sets all cover the same area, but contain different numbers of points. Table 4.4.1 tells how many points each data set contains, and how large the file is on disk. The table also gives information about how much data the structures that were built contain, and how much "excessive" data is created. The "excessive" data is the difference between the size of the file where the structure is stored and the accumulated size of the same number of points converted to Java objects. Each of these objects contains 137 bytes of data.

In real geographical data, the points seldom are as evenly distributed as in the data sets created above. It is also difficult to use real data sets when a specific problem is to be tested. Because of this, another set of data sets were created. The idea behind creating these data sets was to attempt to create some "worst case" scenarios, where the number of points within the query range was extremely low compared to the surrounding area. The total number of points in these new data sets were equal to the ones described above. Within the query range, which was known when the sets were created, only very few points were placed. This leads to a situation where the staircase search area has a very low density of points, while the rest of the area that would have been searched in a bucket query has a much higher density. Figure 4.17 shows an example of a data set where the query area contains only one point, while the surrounding area has a substantially higher point

Table 4.1: Size of the data sets used for testing. The second column gives the size of the input file, the third row gives the size of the PST that was constructed, and the fourth column gives the difference in the amount of data created in the PST and if the points were to be stored sequentially in a file.

Number of points in data set	**Size of data set**	**Size of structure**	**Size of structure - accumulated size of points**
1 million	19,7 MB	280,5 MB	143,5 MB
2 million	39,5 MB	561,2 MB	287,2 MB
5 million	98,6 MB	2.244,6 MB	1.559,6 MB
10 million	197,3 MB	4.489,2 MB	3.119,2 MB

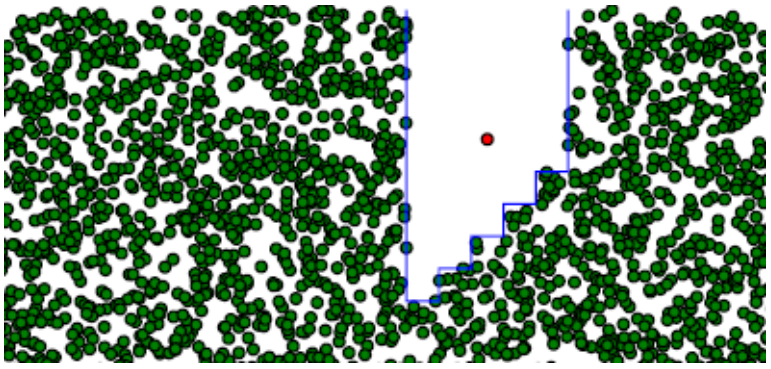


Figure 4.17: An area with extremely low point density, surrounded by areas with significantly higher point density.

density. What was to be tested with these data sets was how much the decrement in the upper x direction influence the efficiency of the staircase query.

Finally we also wanted to test the algorithms on a real data set. To accomplish this, we extracted 8.5 million points from the east coast area of the United States. The data covers roads, rivers, railways, coastlines, state boundaries and so on, and was extracted from the vmap1 data set [15]. This input data set contain 482,9 MB of data, and the PST that was built contains 2.244,6 MB. This data set is stored in another format. The previously mentioned sets were SVG (scalable vector graphics), while this new one is GML (geography markup language). The reason for the last data set containing more data, even though the number of points does not indicate that it should, is that GML files contain more overhead than SVG files.

4.4.2 Hardware and Software

The machine on which the tests were carried out is a Dell Optiplex GX270 with one Pentium 4/2.80GHz processor, 512MB of RAM and a WDC WD800BB-75DKA0 disk with a speed of 7200 RPM. The software is written in Java 1.4.2 [23].

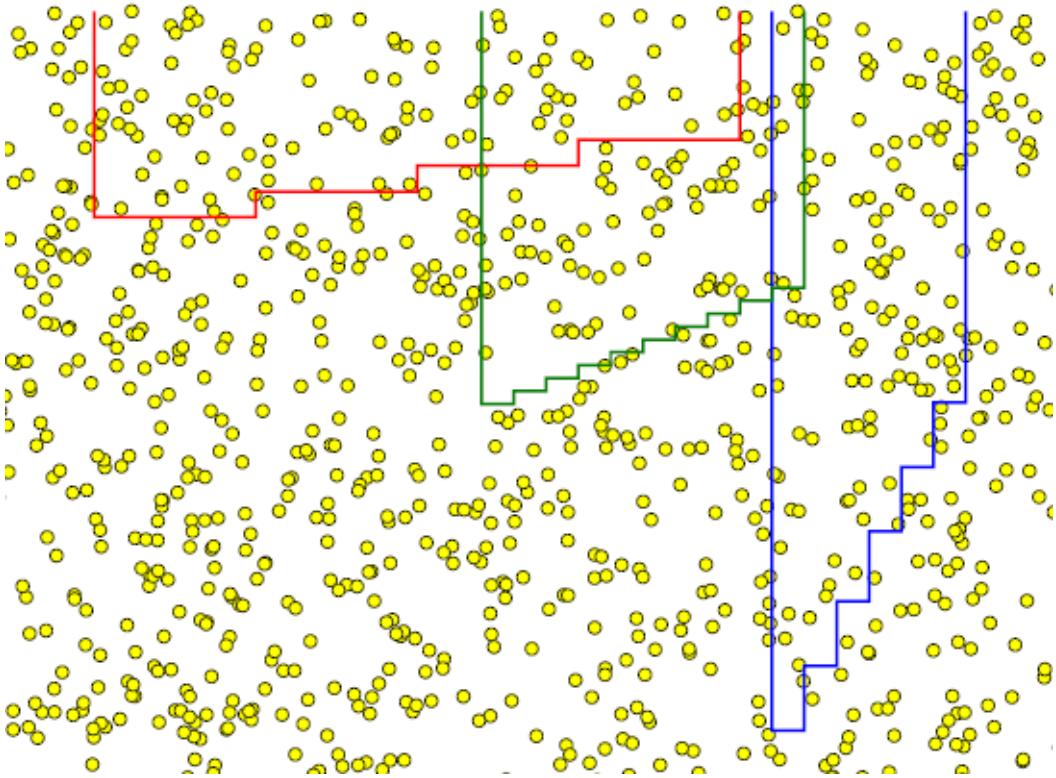


Figure 4.18: Examples of different staircase query ranges covering the same size area.

4.4.3 Test Results

What is most interesting to test is the performance of the new staircase search algorithm for the PST. There are two different aspects of efficiency that need to be tested with the prototype developed. First the complexity of the search, and second the processing time.

The complexity of the search is determined by the number of nodes visited compared to the number of nodes reported. The complexity of a bucket query in a PST is $O(logN + k)$, and the question is how much worse the complexity of the staircase search is.

The processing time is important because the typical computer user becomes impatient if he or she has to wait too long for information before it is displayed. Therefore, if the time used for searching the structure and displaying the result becomes longer than what an average computer user will accept, this approach can not be used. Although the prototype is implemented in a very naive and straightforward way, the search time will still give an indication of how much more time is needed to access disk compared to accessing internal memory.

In the PSTs built from the data sets described in section 4.4.1, staircase queries of different sizes and shapes have been performed. Query ranges covering from 0.25% to 8% of the total area covered by the data sets were used. For each of these query sizes (0.25% to 8%), different shaped staircases were used. Figure 4.18 shows three different staircase shaped search ranges covering the same size

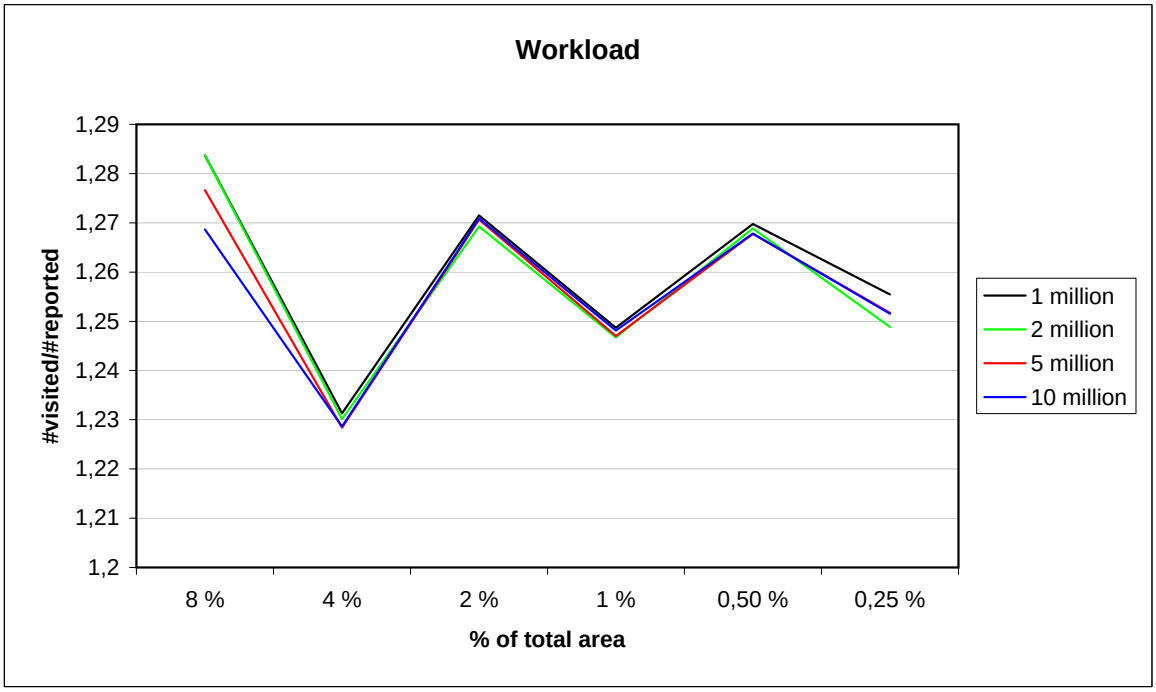


Figure 4.19: The efficiency of the staircase shaped search. The lines show the number of visited nodes for each node that was reported from the search. The searches have been carried out in the data sets described in table 4.4.1, and the size of the search areas have been from 0.25% to 8% of the total area covered by the data set.

area.

Staircase Search Complexity

To determine the complexity of the staircase query, several tests have been performed. For each of the four data sets presented in Table 4.4.1, queries covering six different sized areas were carried out. For each of these area sizes, ten different ranges were used. This adds up to a total of 240 tests for all four data sets.

Figure 4.19 shows the results from running the staircase search on the data sets. The result presented here is the average of the ten different queries on each range size. The tests show that for each point reported, approximately 1.25 nodes are visited. This number is surprisingly constant in all the tests in all the different data sets. In the ratios of the visited points and the reported points for each single query, the highest was 1.56 and the lowest 1.11.

As the data sets used in the tests described above had evenly distributed points, some tests in other data sets were carried out as well. Table 4.4.3 shows the results from these tests. The first column says how many points the total data sets contain, and the other column how many points were in the query range. The query ranges were of 8% of the total area covered by the data set. The third column presents the number of visited nodes when performing the staircase query, and the

Table 4.2: Results from querying an area containing very few points surrounded by an area with very high point density. The ratio states what percentage the nodes visited in the staircase query is of the number of nodes visited in the bucket query.

Total # points	**# points in query area**	**Staircase query**		**Bucket query**		**Ratio of visited nodes in bucket and staircase**
		Visited	Reported	Visited	Reported	
1 million	1	558	1	29358	29324	1.9%
	10	546	10	29527	29494	1.85%
	100	1527	100	29798	29766	5.12%
2 million	1	561	1	59144	59107	0.95%
	10	651	10	58837	58804	1.11%
	100	1869	100	59538	59506	3.13%
5 million	1	708	1	147432	147397	0.48%
	10	758	10	147432	147394	0.51%
	100	2117	100	147222	147184	1.44%
10 million	1	771	1	294233	294194	0.26%
	10	850	10	294505	294463	0.29%
	100	2358	100	295236	295195	0.8%

fourth the number of points reported. Naturally, this number is equal to the number of points in the query range, given in the second column. The fifth column presents the number of visited nodes in a regular bucket search with the same limits in the x- and y-direction as the staircase search. Column six presents the number of nodes reported from this bucket search.

As can be seen from the table, the ratio of visited and reported points is less stable when the points are not evenly distributed. However, what is interesting to see here, is the effect the decrement in the upper x-limit of the search has on the performance of the staircase search. Without this decrement, the number of nodes visited in the staircase query, would be equal to the number of nodes visited in the corresponding bucket query. The last column in Table 4.4.3 tells how many percent of the points visited in the bucket query, were also visited in the staircase query. As can be seen, this percentage is generally very low. Another interesting fact that can be read from these numbers, is that the bucket query is very close to maintaining its logarithmic complexity, even though large parts of the query range contains next to no points at all.

The tests performed in the real geographical data set, could not be carried out "scientifically" as the ones in the random data sets. This is because the density of the points were very high in some areas of the set, and very low in others. Because of this, the results tended to give no meaning. However, some tests have been performed, and the results are presented in Table 4.4.3.

From these results we can see that in some cases, very few redundant points are visited, while in other cases, the number is higher. Overall, we can say that the staircase search algorithm has a surprisingly good performance. This statement is backed by both these results, the results from the "worst case" scenarios presented in Table 4.4.3, and the results from the evenly distributed data sets given in Figure 4.19.

Table 4.3: Results from performing staircase queries on data representing real geographical areas.

Size of query range in % of total area	# points visited	# points reported	Ratio of visited nodes pr reported node
5.9	7321825	7321821	1.0000005
4.12	592	575	1.02957
3.8	6784123	6784112	1.0000016
3.48	2045399	2045382	1.0000083
0.17	315016	237815	11.3254
0.13	600867	400510	1.50025

Staircase Search Processing Time

The other aspect of the structure's efficiency that needed to be tested, was that of the amount of time used for answering a query. The data sets contain more points than what fits in the computer's internal memory, and therefore, the absolute quickest way to retrieve the points is to read them sequentially from a file. Because of this, we found it interesting to see how much more time is needed to search for a number of points in a PST compared to reading them sequentially from a file. For these tests, the data sets described in Table 4.4.1 were used, and Table 4.4.3 shows what the results were. The tests show that, on average, searching the data structure takes approximately 5 times longer than reading the points sequentially from a file.

Figure 4.20 illustrates the increased time needed to search compared to read the points sequentially. The time is given in milliseconds pr point reported from the queries.

If we compare these results to the ones presented in Figure 4.19, we can recognize some of the same tendencies. The amount of nodes visited pr reported is higher in the 2% areas and lower in the 1% areas, and so is the increase in the amount of processing time in the 2, 5 and 10 million data sets. The results in processing time for the 1 million data set does not have the same features as the other three, and this is probably because the data set is too small to get stable results.

Table 4.4: Results from tests run to find the increase in processing time used for searching the data structure compared to reading the same number of points sequentially from a file.

# points	Size of query area in %	Average increase for ten queries
1 million	0.25	987%
	0.5	471%
	1	472%
	2	478%
	4	384%
	8	345%
2 million	0.25	496%
	0.5	405%
	1	380%
	2	386%
	4	366%
	8	362%
5 million	0.25	488%
	0.5	466%
	1	393%
	2	507%
	4	455%
	8	414%
10 million	0.25	602%
	0.5	567%
	1	509%
	2	621%
	4	523%
	8	479%

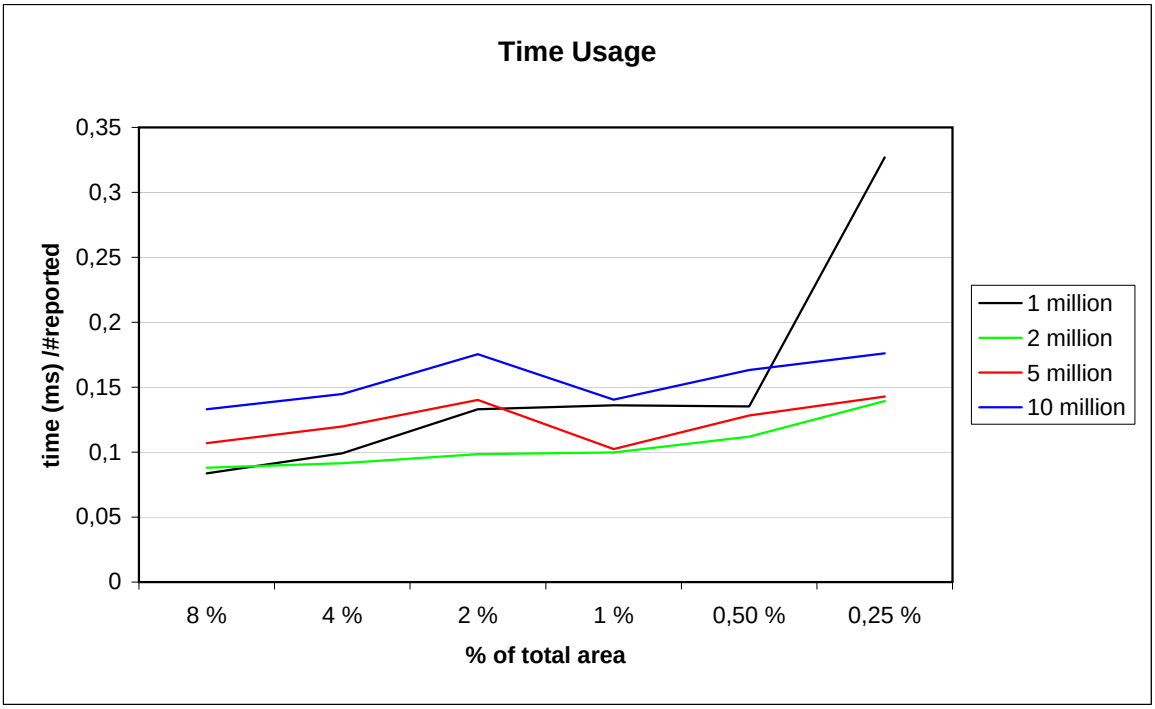


Figure 4.20: Statistics for the increase in time used for searching compared to reading the same number of points from a file. The time is presented in milliseconds pr reported point for each of the four data sets, in ranges covering from 0.25% to 8% of the total area.

Chapter 5

Further Work

This chapter gives some suggestions to further development of the 2D Range Simplification Tree and using the PST for performing range queries. First of all, it would be interesting to develop a dynamic version of the new structure, and some preliminary work on this topic is presented. Second it is clear that the PST is a very interesting data structure, with many possible applications. The second section in this chapter provides some thoughts on how to utilize the PST in order to develop a method for answering non-orthogonal range queries as well.

5.1 Developing a Dynamic 2D Range Simplification Tree

This section presents the work that already has been done in order to develop a dynamic 2D Range Simplification Tree, and some thoughts and ideas on how to complete the process. Because very little information on dynamically updating two-level tree structures exist, this section is mostly concerned with the second level, the PSTs.

As already described in chapter 2 presenting related work, dynamically updating multi-dimensional data structures efficiently is a complicated process and has proven difficult. In a two-dimensional structure there is only one level of associated structures, but still, rebuilding the associated structures from scratch is not an option. Another solution must be found. Chiang and Tamassia's approach [4] involves rebuilding only a very small amount of the associated structures from scratch, and hopefully it is possible to use that principle on this structure as well.

5.1.1 Dynamic Priority Search Tree

The following section is based on a lecture given by Professor Robert Tamassia on dynamic PSTs [26]. The lecture provides an overview over the techniques used for building and maintaining a dynamic PST. This section is an attempt to explain the processes more thoroughly, by using more complete examples and figures.

A PST can be efficiently constructed using a bottom-up construction method like the method used for constructing heaps as binary trees. When building a PST by the bottom-up method, the

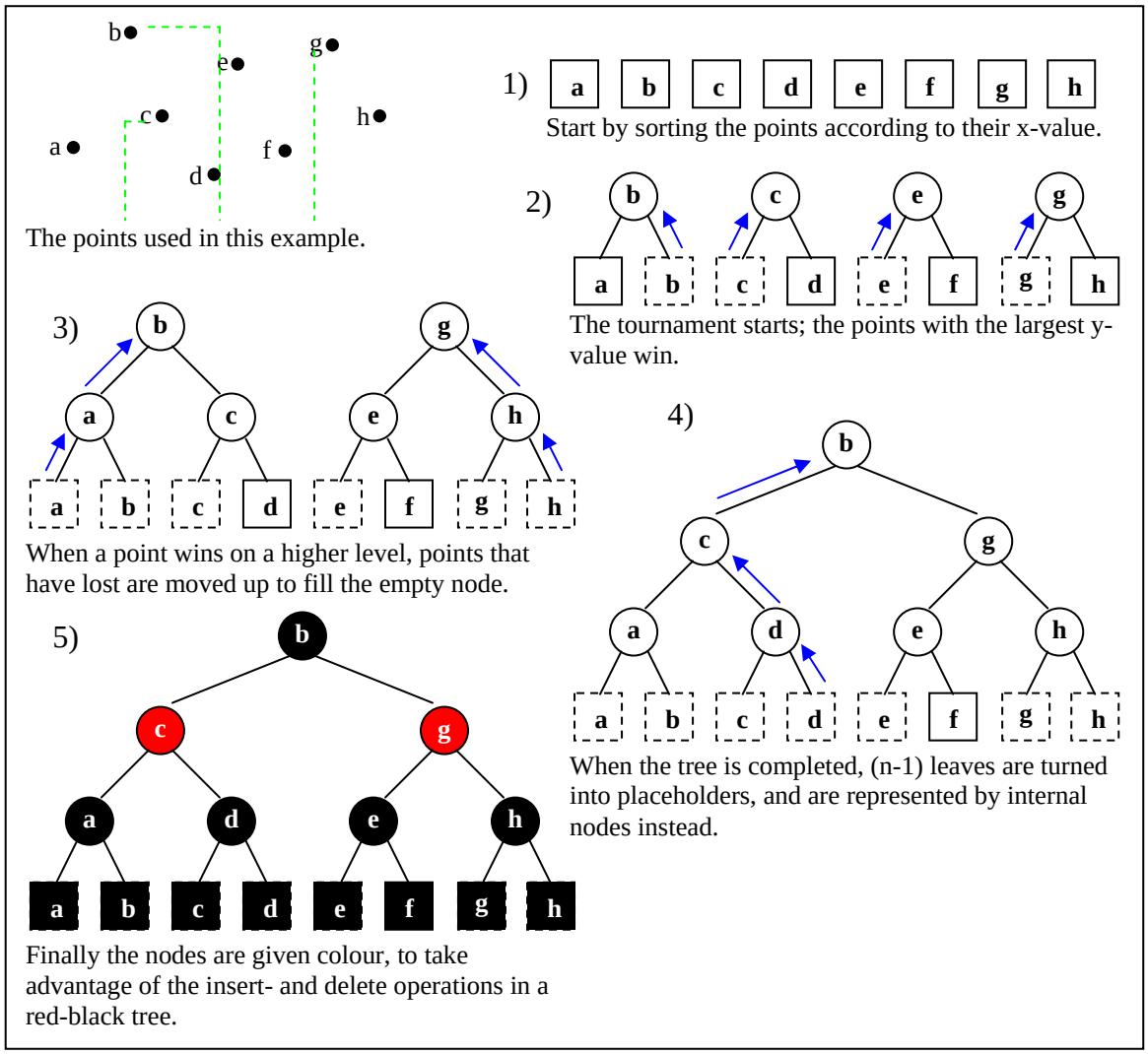


Figure 5.1: Bottom-up tournament construction of a PST.

first step is to make all the points leaves, and sort them according to their x-value. After this the tree is constructed using a "tournament-principle" similar to the one used to build tournament trees.

In the tournament the points "compete" two and two, and the one with the highest y-value wins the competition and becomes the root of the subtree. On the next level of the tournament, the winners of the last level compete for the position as the root on this level. The points that win leave empty nodes behind, and these nodes are filled by points that previously have lost in the competition. This may lead to other nodes being emptied, and so the process is repeated until there are no more points that can fill the nodes. When a point is represented by an internal node in the tree, the leaf for this point is made a "placeholder". When the tree is completed, $(N-1)$, where N is the number of points in the data set, of the leaves are made placeholders. The leaves are always sorted according to the point's x-values. Figure 5.1 illustrates the construction process using a small example of eight points.

Constructing the tree this way is done in $O(N)$ time. However, as the first step in the process is to sort the points, and sorting takes $O(NlogN)$ time, the entire construction process takes $O(NlogN)$ time.

After the tree has been constructed, the nodes are colored according to the rules of a red-black tree. The red-black tree is a binary search tree that has efficient methods for inserting and deleting elements. The reason for using the red-black tree's coloring scheme is that it provides efficient methods for inserting and deleting elements. These processes are described in the following sections.

Insert

Inserting a new point in a dynamic PST is a three step process:

```
1. Add a new leaf which will be a placeholder for the new point.
2. Push the new point down the tree, starting at the root.
3. Rebalance the tree if necessary.
```

As the leaves of the tree always are sorted according to the points' x-values, the placeholder for the new point will have to be added at the correct place in this sequence. To find this place, start at the root and iterate down the correct subtrees by comparing the new point's x-value to the x-values of the root of each subtree. If the new point has an x-value that is larger than the x-value of the root, proceed down the right subtree. If it is smaller, proceed down the left subtree. The first tree in figure 5.2 illustrates this process.

When the correct place for the new placeholder-leaf is found, the leaf already stored in this place must be replaced by a new (empty) internal node. The new leaf and the replaced leaf becomes the children of this new node, as can be seen in the second tree in figure 5.2. The new node is colored red according to the rules for insertion in a red-black tree.

After the placeholder and a new internal node have been added to the tree, it is time to find the correct position for the new point in the tree. To find this position, a push-down operation is carried out. This operation starts at the root of the tree, and the y-values of the root and the new node is compared. If the new point has a y-value that is smaller than the root's y-value, the push-down operation is continued in the correct subtree decided by comparing the new point's x-value to the splitvalue of the root. If the new point has a y'value that is larger than the root, the push-down

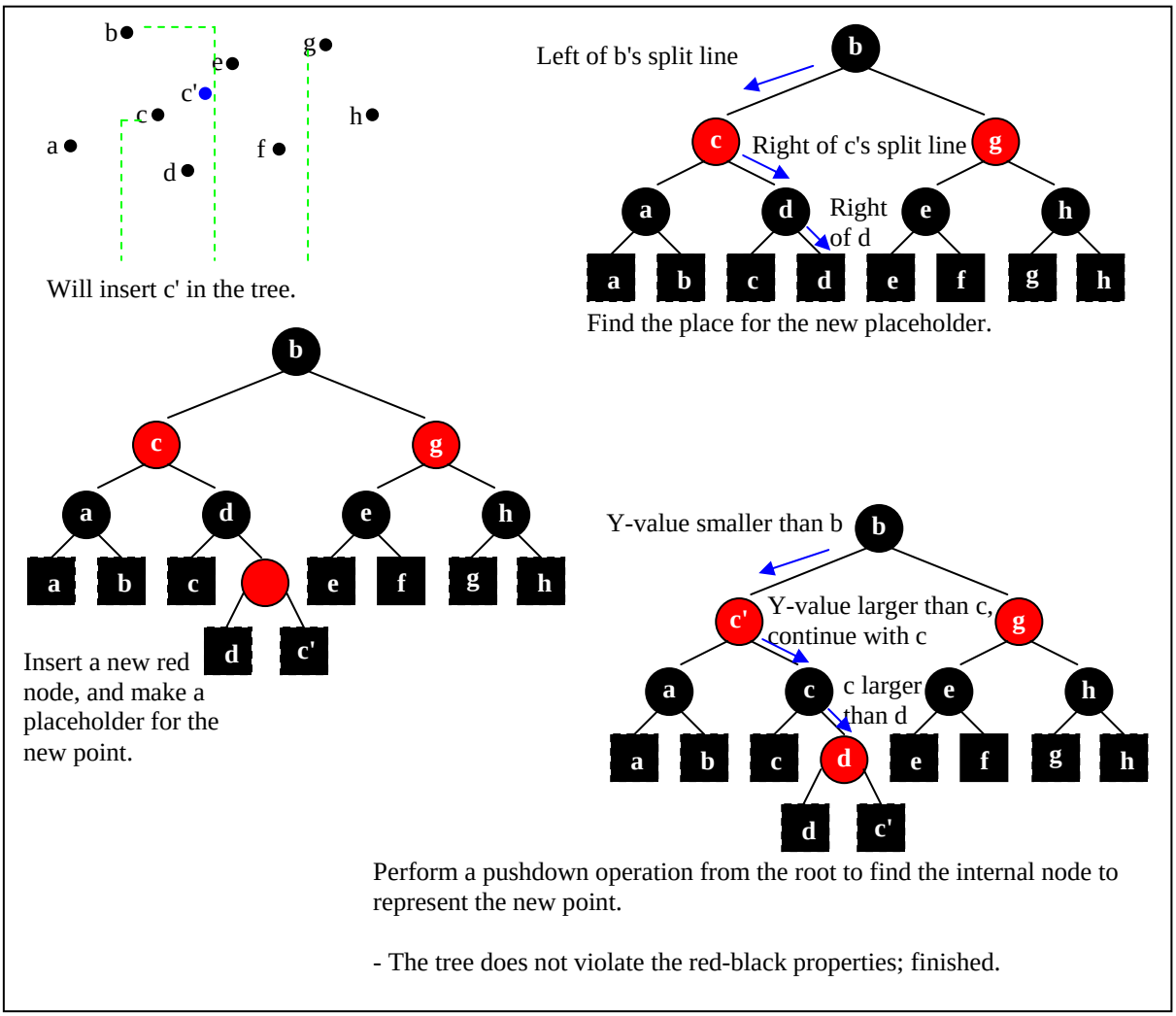


Figure 5.2: Insert a new point in the PST, no restructuring needed.

operation is continued with the old root as the ”new point”. This process is recursively repeated until all points are placed in a node. The third tree in figure 5.2 illustrates this process.

Once the new node is inserted, all affected splitvalues must be updated. The splitvalue of a node is placed in the middle of the x-values rightmost child in its left subtree and the leftmost child in its right subtree. When a new node is inserted, these conditions may have been changed, and hence an update may be required.

After the insertion, the tree must be balanced according to the properties of a red-black tree. If any of these properties are violated after having inserted a new point, the tree must be restructured. In the example in figure 5.2 this is not the case, but in figure 5.3, the tree is incorrect and must be repaired.

A new node inserted in a red-black tree is always painted red, and the rule of having the same number of black nodes in every path from the root to a leaf is therefore not threatened. The rule that says no red node can have a red child, however, can easily be violated when inserting a new red node. When such a “red-red violation” occurs, the colors of the adjacent nodes decide what to do next. If both the parent and the “uncle” (the grandparent’s other child) are red, repaint the parent and the uncle black and the grandparent red. This might lead to the grandparent having a red-red violation with it’s parent. If that is the case, the process is repeated on this level. If only the parent is red, and the uncle is black, the answer is to perform a rotation.

In figure 5.3 the points c' and c'' has been inserted into the tree that was constructed in figure 5.1. The next step is to insert c''' into this tree. The leaf placeholder is placed between that of c'' and d, and the point itself is placed as the right child of c''. This leads to a red-red violation between c''' and c''. Both the parent (c'') and the uncle (d) are red in this case, so c'', d and c are repainted. This leads to another red-red violation, this time between the original grandparent c and its parent c'. In this case only the parent is red, the uncle (a) is black, and then a rotation must be performed.

A rotation in a dynamic PST can not be performed in exactly the same way as in a red-black tree. This is because in a red-black tree the points are organized only by their x-values. In a dynamic PST, however, the y-values must also be considered. A rotation in a red-black tree leads to the subtree getting a new root, while in a PST the root must remain the same. This means that instead of rotating, the subtrees must be reorganized. A single left rotation is illustrated in figure 5.4. A single right rotation is symmetrical to a left rotation, and double rotations consist of two single rotations.

The rotation principle in figure 5.4, is explained as follows: $root$ is the root of this portion of the tree both before and after the rotation has been carried out. Before the rotation u is the root of the left subtree, $child$ is the root of the right subtree, and v is the root of $child$’s left subtree. After the rotation, subtrees 1 and 2 will both be in the left subtree of $root$, while subtree 3 remains as $root$’s right subtree. The point that originally was $root$’s right child, is now $root$’s left child, $child$, but this might not be correct. To make sure the correct point is placed in this node, the point with the greatest y-value of u and v is chosen, and a push-down operation starting at node $child$ is performed with this point.

In the example in figure 5.4, the root is the point c' and it has a red-red violation with its right child c. The left child, a, is black, and hence a rotation is required. The rotation is performed by making the right child, c, the new left child, and c’s right child, d, becomes the new right child of c'. The left child, a, becomes the left child of c, and the former left child of c becomes the new right child. When comparing a, c and c'' (u, v and ri from the principle), we find that the new left child

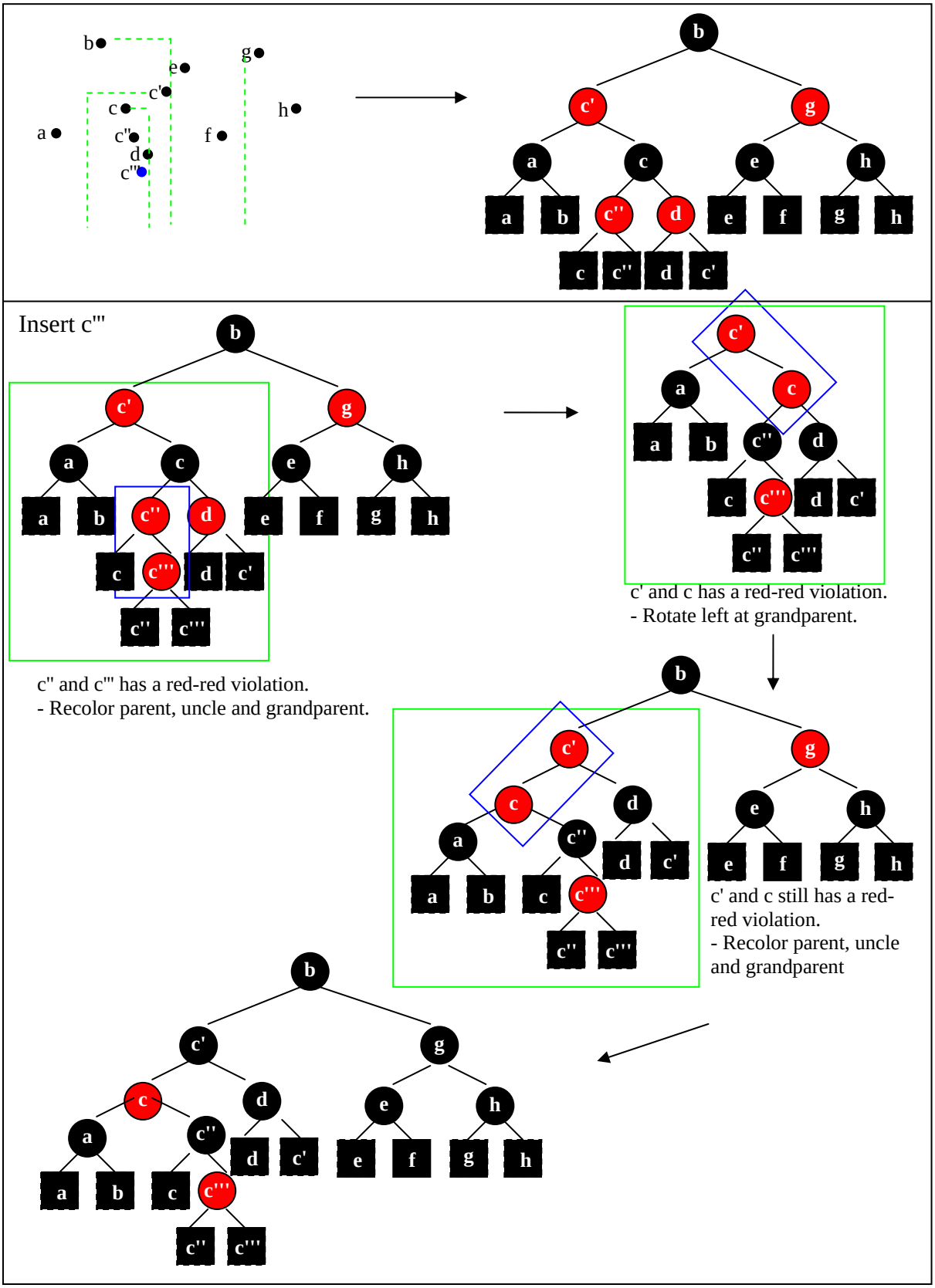


Figure 5.3: Insert a new point and restructure a dynamic PST.

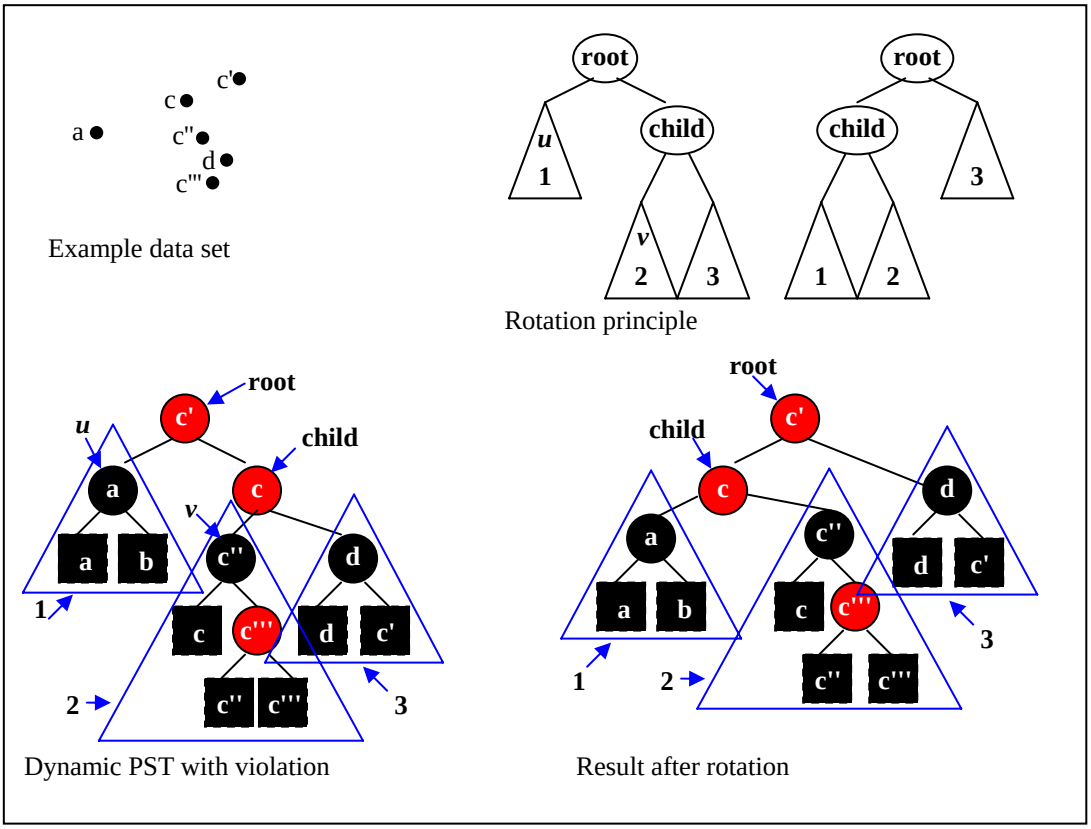


Figure 5.4: A rotation in a dynamic PST.

c, also is the point with the greatest y-value, and hence the rotation is complete.

In figure 5.4, after the rotation is performed, there still is a red-red violation between c and its parent c'. In this case the uncle, g, is also red, and c', g and the grandparent b is repainted. This again leads to a violation of the rule which states that the root is always black. To repair this, the root is painted black, one black node is added to all the paths, and the balance is restored.

Delete

To delete a point from a dynamic PST, these steps must be followed:

```
1. Locate the node that represents the point to be deleted.
2. Remove the point from this node, and fill it by replaying a portion
   of the tournament among the points in its subtree.
3. Delete the placeholder-leaf.
4. Rebalance the tree according to the properties of a red-black tree.
```

Figure 5.5 shows an example of deleting a point from a dynamic PST. The point to be deleted is stored in the left child of the root, and after the tournament is replayed, the node to be deleted is black. When deleting a black node, the fifth property of the red-black tree is violated, as no longer

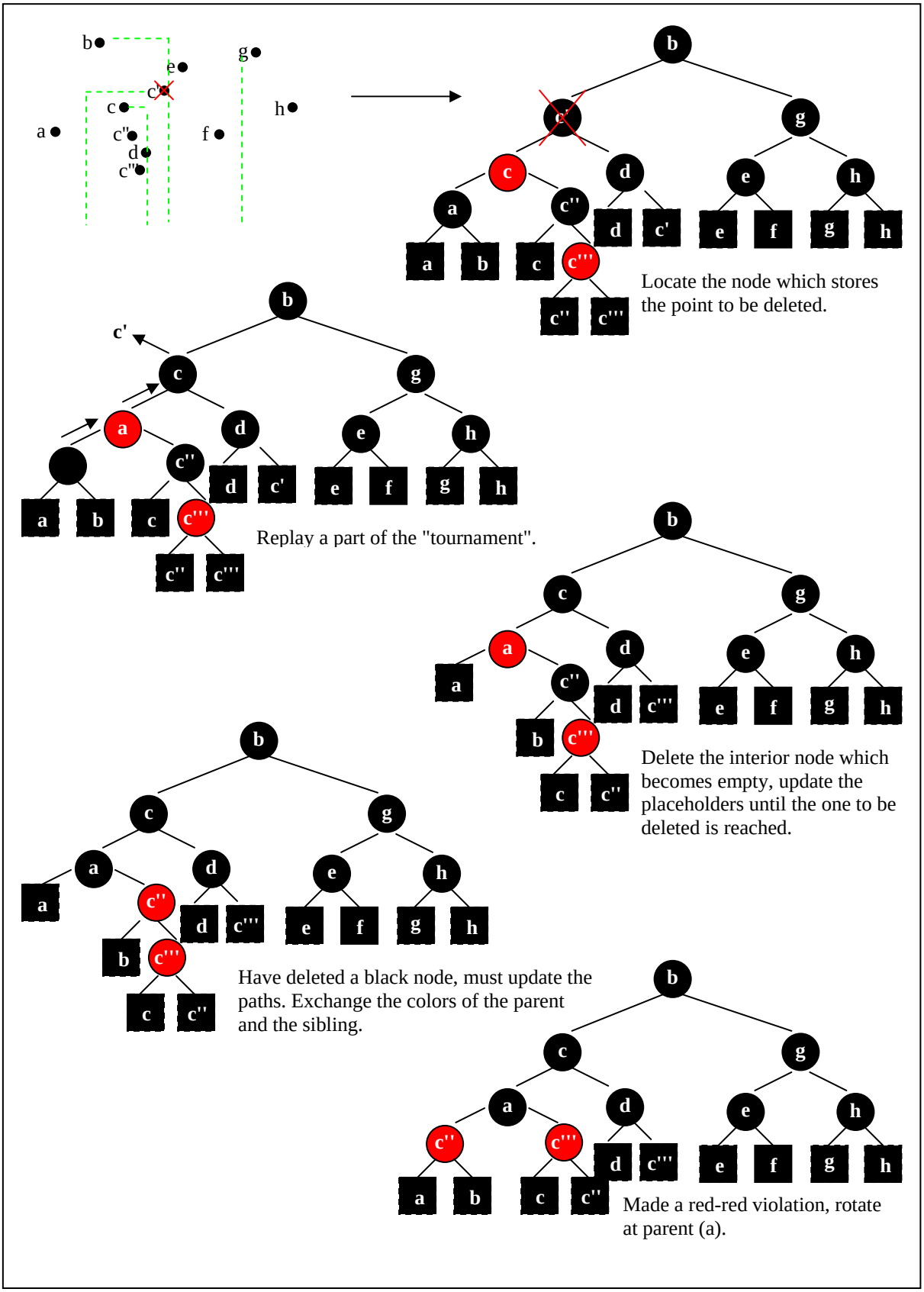


Figure 5.5: Example of deleting a point from a dynamic PST.

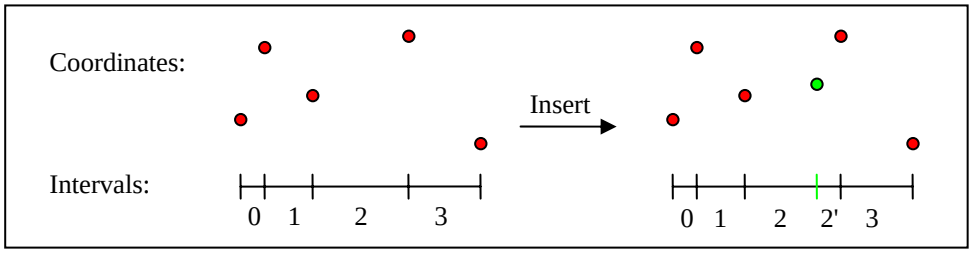


Figure 5.6: Inserting a new point in a interval sequence.

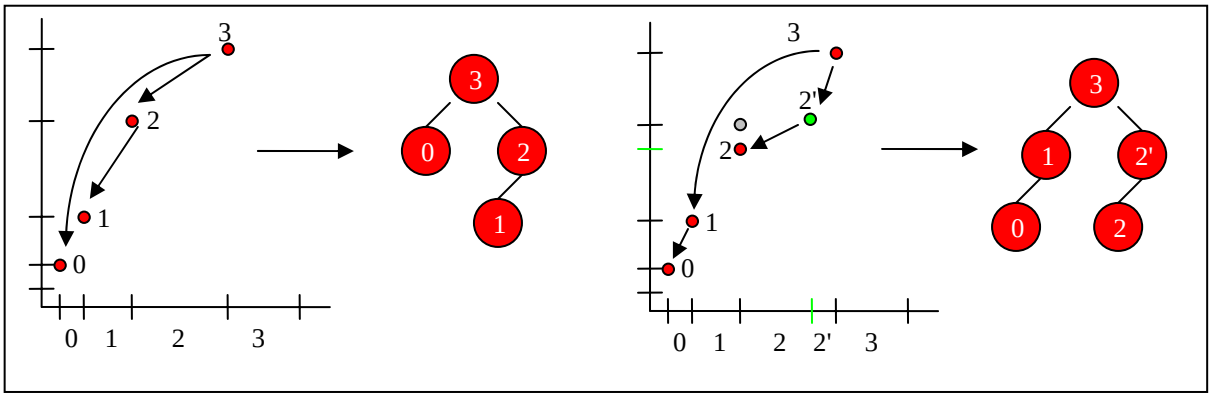


Figure 5.7: Inserting a new point in an interval PST.

all the paths from the root to a leaf contains the same number of black nodes. Then the rules for rebalancing when deleting a black node with a red parent and a black sibling with a red right child are followed. The rules say to rotate left at the parent. Rotation in a dynamic PST does not involve rotating the root of the subtree, but the colors are exchanged as if the root had taken part in the rotation.

5.1.2 Dynamic 2D Range Simplification Tree

When updating a 2D range tree [4], the new point is inserted as a leaf in the outer dimension, and then inserted in all the associated structures in the path from that leaf to the root of the tree. In our 2D Range Simplification Tree, there is one more consideration to make: The new point that is inserted must be converted to an interval along with the other x-values in every PST, see figure 5.6. This means that before one can start worrying about where in the associated structures the new point is to be inserted, the point must be created by finding its place in the interval sequence. Not only will this lead to one more interval in the sequence, it will also lead to not one but two intervals being changed. As can be seen in 5.7, this may lead to the entire PST being changed, and a “simple” insert will not do the trick. In addition to this, as different coordinates are included in every PST, these sequences are always different. Hence, this procedure must be repeated for every PST where the point is to be inserted.

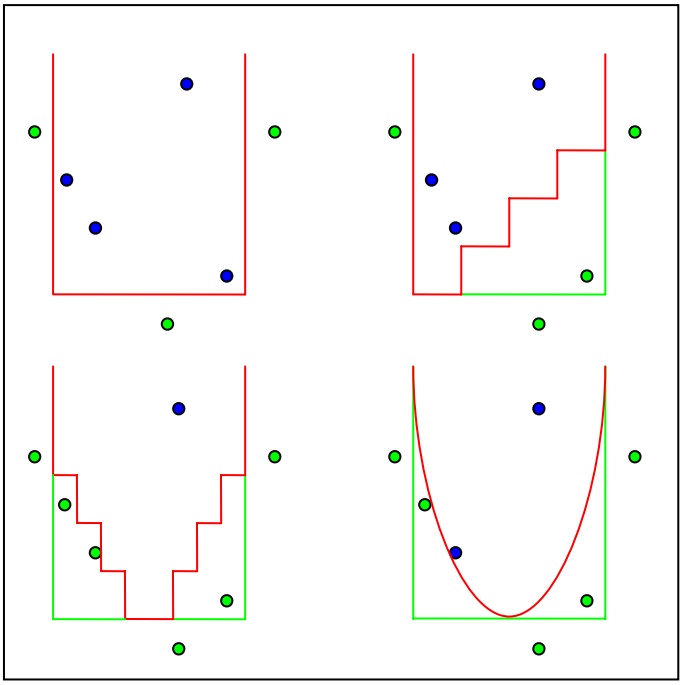


Figure 5.8: Examples of different adjustments that can be made on the bucket search.

Inserting the new range in the outer dimension can be done in the same way as in a regular binary search tree; recursively iterate until the place for the new node is found, create a new node, and balance the tree by rotating. If the outer tree is implemented as a red-black tree, this can be done in $O(logN)$ time, where N is the number of nodes in the tree, which again is the number of ranges along the y-axis that the data set contains.

The structure is meant to be used for storing large amounts of geographical data. Updating geographical data usually involves large changes in the data set. Because of this, an update procedure must be efficient also when dealing with larger updates, otherwise it will be better to rebuild the entire structure from scratch when updating. Whether or not such an efficient update procedure can be developed is an issue for further work.

5.2 Non-orthogonal Range Search

In this thesis it has been shown that the bucket search of a PST can be transformed into querying a range shaped like a staircase. When it is proven possible to adjust the search area on one side of the bucket, it is likely that the other side can be adjusted too. Figure 5.8 shows examples of different adjustments made to the traditional bucket search.

In theory, we can split any line into small segments shaped like the last example from Figure 5.8. If we draw this theory even further, any search area can be split in half, and make two such lines that

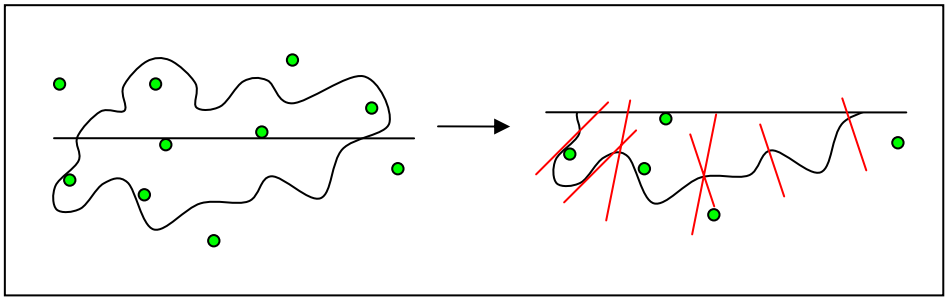


Figure 5.9: A simplified example of splitting a search area into segments.

can be split into segments. This means that if we can answer a semi-infinite query shaped like the one in Figure 5.8 efficiently, and find a way to efficiently limit the search in the infinite direction, we can, in theory, answer any range query. Figure 5.9 shows a very simplified example of such a situation.

The PST is a very interesting data structure when it comes to performing range queries, and it would therefore be interesting to see whether it can be used in developing an efficient method for performing non-orthogonal range search. However, there is no time for that in this project, and hopefully someone will see the potential and develop these theories further in the future.

Chapter 6

Discussion and Conclusions

This chapter sums up the work presented in this thesis. Whether or not the proposed solution actually solves the problem is discussed, and so are the results presented concerning the efficiency of this solution. In the second section of this chapter a conclusion is drawn, stating if the purpose for the project is fulfilled.

6.1 Discussion

This section provides discussions concerning the results that are presented in this thesis, and some of the problems that are yet to be overcome.

The tests that are carried out has shown two things; First of all they have shown that this approach to combining orthogonal range search and line simplification works. Second they have shown that the adjusted query developed for the PST, the staircase query, has an acceptable performance.

The bucket query, which is the type of query generally connected with the PST, has a complexity of $O(logN + k)$ where N is the number of points in the data set, and k is the number of points reported from the query. The staircase query is an adjusted bucket query where the lower y-limit and the upper x-limit varies. The worst case complexity of a staircase search is when one visits all the nodes that would have been visited in the corresponding bucket query. However, the tests that were carried out, which are described in section 4.4, show that the average complexity is much better than this. Even when the staircase query was tested in some "worst case scenario" data sets, that were created specifically to perform badly, never more than 5.5% of the nodes visited in the corresponding bucket query were visited in the staircase query.

When it comes to the complete 2D Range Simplification Tree, an external implementation was made. Unfortunately, it was completed too late, and the construction was too time consuming. Because of this, the time ran out, and no large tests could be conducted. However, based on the results from the tests of the staircase query algorithm in the PST, some conclusions can still be drawn.

Because the first level of the 2D Range Simplification Tree has the same structure as a regular balance binary tree, the complexity of a query in this level is worst case $O(logN)$, where N is the

number of ranges along the y-axis that the data set is split in. Because the search can return several of these ranges, the search has an output sensitive complexity, hence, $O(logN + k)$, where k is the number of ranges that are completely covered by the query range. From the tests performed on the staircase query in the PST, we have determined that the staircase query is quite efficient. We also know that even though the intervals in the PST are included in many first-level nodes, they are only included in one of the first-level nodes that are searched in each query that is performed. Because of this, we can safely say that the complexity of searching the complete 2D Range Simplification Tree is never worse than $O(logN + k)$ in the first level + the accumulated number of nodes visited in each of the PSTs that are connected to the reported first-level nodes. The average case is probably much better than this.

The following subsections presents two problems that should be overcome before if anyone wants to continue working with this problem and the solution presented in this thesis.

6.1.1 External Implementation

Because of the enormous amounts of data involved when dealing with GIS, it is crucial that the structure is built and stored externally. Even though one suggested external implementation is described in this thesis, this problem is not solved. The solution given here is not efficient enough, and was not even useful for performing tests on the complete 2D Range Simplification Tree, because it was too time consuming. However, it did show that an external implementation can be made, and, hopefully, some of the techniques described can be used in a future, more efficient implementation.

6.1.2 Constructing the 2D Range Simplification Tree

The current implementation of the PST is not optimal, because in every subtree, all the points must be checked to find the one with the greatest y-value. The base structure of a PST is a balanced binary tree, and for a tree to be balanced, it must have approximately the same number of nodes in each subtree. As the nodes are placed in the subtrees in a PST according to their s_1-value, and in theory, all the nodes can have the same s_1-value, the question of how to construct the tree arises. Since searching in a PST is an output sensitive algorithm, and the nodes will be reported no matter which subtree they belong to, the complexity of the search will be the same in either case. However, during the construction of the tree there is a difference.

When the nodes that have the same s_1-value are separated in different subtrees, a problem arises when selecting the node with the largest s_2-value to be the root. If all the nodes that have an s_1-value that is below a certain value are in one subtree, and the ones that are above in the other, the node with the largest s_2-value can be chosen from a list where the nodes are sorted by the s_2-value, and a simple check to see if the s_1-value is within the current subtree can be performed. If the nodes with the same s_1-values are separated into different subtrees, this check will not be sufficient, and one must check all the nodes in the subtree to select the one with the largest s_2-value. In the current implementation a balanced tree has been preferred, and, hence, the latter method chosen.

Another way to solve this problem is to figure out how the PST can be constructed using the bottom-up construction of a heap. This method is introduced in the first section of the previous chapter (Further Work), but it has not been implemented.

To avoid this, one can sort the points both according to their x- and their y-values, but that raises another issue. The base structure of a PST is a balanced binary tree, and for a tree to be balanced, it must have approximately the same number of nodes in each subtree. As the nodes are placed in the subtrees in a PST according to their x-value, and in theory, all the nodes can have the same x-value, the question of how to construct the tree arises. Since searching in a PST is an output sensitive algorithm, and the nodes will be reported no matter which subtree they belong to, the complexity of the search will be the same in either case. However, during the construction of the tree there is a difference.

There are also some issues concerning the selection of the splitting values in the y-range tree. It is important that points with the same y-coordinate are stored in the same subtree, but it is also important to maintain the tree balanced to ensure a logarithmic complexity in the search. As several points have the same y-coordinate, the middle of the list is not a good choice for the split value. What is done at the present time is that the two values in the middle of the list are compared. If these are different, the median of these two values are chosen. If they are the same two loops are run up and down the list to find the pair of values that are different and that are closest to the middle of the list. Then the median of these two different values are chosen to be the node's split value. This may lead to the tree being unbalanced, as there can be many points with the same y-value in the middle of the list. However, it is vital that all nodes with the same y-values end up in the same subtree, so maybe this possible imbalance must be accepted.

6.2 Conclusion

Based on the tests that were carried out, it is clear that the new data structure for combining orthogonal range search and line simplification is applicable. It is possible to solve these two problems simultaneously using the proposed approach, and queries can be answered within an acceptable amount of time. However, there are still some issues that must be solved.

It is clear to see that without an efficient external implementation of the 2D Range Simplification Tree, this method for combining range queries and line simplification can not be used. Nevertheless, it is important to keep researching this problem. More and more geographic data is made available, and updating layers of different resolutions becomes increasingly more resource demanding, and very easily lead to inconsistencies.

References

[1] Pankaj K. Agarwal, Lars Arge, and Ke Yi. An optimal dynamic interval stabbing-max data structure? SODA'05, Feb 2005.

[2] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, Sep 1975.

[3] Sean Mauch Caltech. Data structures for orthogonal range queries. Poster, Center for Simulation of Dynamic Response of Materials, Apr 2003. URL: `http://csdrm.caltech.edu/calendar/roadshow04/posters/cs3-orq.ppt`.

[4] Yi-Jen Chiang and Roberto Tamassia. Dynamic algorithms in computational geometry. *IEEE, Special Issue on Computational Geometry*, 80(9):1412–1434, Sep 1992.

[5] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry - Algorithms and Applications*. Springer, Berlin, 2nd revised edition, 1999.

[6] João Argemiro de Carvalho Paiva. Topological Equivalence and Similarity in Multi-representation Geographic Databases. URL: `http://www.dpi.inpe.br/teses/miro/`, Dec 1998.

[7] D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Canadian Cartographer*, 10(2):112–122, Dec 1973.

[8] John Hershberger and Jack Snoeyink. Speeding Up the Douglas-Peucker Line-Simplification Algorithm. Technical Report TR-92-07, University of British Columbia, 1992.

[9] T. Lang. Rules for robot draughtsmen. *The Geographical Magazine*, 42(1):50–51, 1969.

[10] Zhilin Li and Stan Openshaw. Algorithms for automated line generalization based on a natural principle of objective generalization. *International Journal of Geographical Information Science*, 6(5):373–389, 1992.

[11] Edvard M. McCreight. Priority Search Trees. *SIAM J. Comput.*, 14(2):257–276, 1985.

[12] R. B. McMaster and K. S. Shea. Generalization in Cartography. Association of American Geographers, 1992.

[13] Nirvana Meratnia and Rolf A. de By. A New Perspective On Trajectory Compression Techniques. ISPRS Commission II and IV, Oct 2003.

[14] Gunnar Misund, Knut-Erik Johnsen, and Mats Lindh. Hierarchical GML Modeling of Transportation Networks. GML Dev Days, Jul 2003.

[15] Tony Moore, Richard Glass, and David Wesloh Joyce A. Hoffman. Vector smart map level 1 - A new DMA product. *61st Annual Convention of the American Society for Photogrammetry and Remote seusing*, 2:83–92, 1995.

[16] J. Muller. AUTO CARTO 8: Optimum point density and compaction rates for the representation of graphic lines. Mar 1987.

[17] Kamesh Munagala. External memory algorithms. Technical Report cs361A, Stanford, November 2002.

[18] Nadia Shahriari Namini. A New Approach For Simplification Of Linear Vector Data For Internet_Based Gis Applications. Master's thesis, University of Calgary, Department of Geomatics Engineering, 2002.

[19] Octavian Procopiuc, Pankaj K. Agarwal, Lars Arge, and Jeffrey Scott Vitter. Bkd-tree: A dynamic scalable kd-tree, 2002.

[20] Hanan Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys (CSUR)*, 16(2):187–260, Jun 1984.

[21] Clifford A. Shaffer. *A Practical Introduction to Data Structures and Algorithm Analysis*, chapter 8 File Processing and External Sorting. Prentice-Hall, 1997.

[22] Cláudio Silva, Yi-Jen Chiang, Wagner Corrêa, Jihad El-Sana, and Peter Lindstrom. Out-Of-Core Algorithms for Scientific Visualization and Computer Graphics. URL: `http://www.cse.ogi.edu/~csilva/papers/silva-et-al-2003.pdf`, 2003.

[23] Inc. Sun Microsystems. Java 2 Platform, Standard Edition (J2SE Platform), version 1.4.2 - Performance. URL: `http://java.sun.com/j2se/1.4.2/reference/whitepapers/index.html`.

[24] Roberto Tamassia. Dynamic computational geometry. Lecture: ALCOM Summer School, Aarhus, Aug 1991.

[25] Roberto Tamassia. Lecture 9: Priority Search Trees - Part I, Mar 1993.

[26] Roberto Tamassia. Lecture 10: Priority Search Trees - Part II, May 1994.

[27] Peter van Oosterom. *Reactive data structures for geographic information systems*, chapter 6 The Reactive-Tree, pages 88–98. Oxford University Press, Oxford, 1993.

[28] Peter van Oosterom. *Reactive data structures for geographic information systems*, chapter 5 The Binary Line Generalization Tree, pages 83–88. Oxford University Press, Oxford, 1993.

[29] Peter van Oosterom. The GAP-tree, an approach to "On-the-Fly" Map Generalization of an Area Partitioning. In *GISDATA Specialist Meeting on Generalization*, Dec 1993.

[30] VBB Viak - Liqiu Meng (project leader). Automatic Generalization of Geographic Data. Technical report, 1997.

[31] M. Visvalingam and J.D. Whyatt. Line Generalisation by Repeated Elimination of Points. *Cartographic J.*, 30(1):46–51, 1993.

[32] Robert Weibel. *Algorithmic Foundations of Geographic Information Systems*, chapter Generalization of Spatial Data: Principles and Selected Algorithms, pages 99–152. 1997.

[33] J.D. Whyatt and P.R Wade. The Douglas-Peucker line simplification algorithm. *Bulletin of the Society of University Cartographers*, 22(1):17–27, 1988.

[34] B.S. Yang, R. S. Purves, and R. Weibel. Implementation of progressive transmission algorithms for vector map data in web-based visualization. XXth ISPRS Congress, Jul 2004.

List of Figures

List of Tables

Appendix A

Glossary of Terms

Amortized analyses: Bounds the cost of a sequence of operations and distributes this cost evenly to each operation in the sequence.
Approximation: A visualization of a geographical area presented with a lower resolution than the original data.
Area partitioning: A structure used when creating maps in which each point belongs to exactly one of the areas; hence, there are no overlaps or gaps.
Associated data structure: The second-level data structure associated with each node in the first-level in a two-level data structure.
Binary tree: A data structure consisting of nodes, where each node has no, one or two children. The top of the tree is a node called root.
BLG: Binary Line Generalization. A binary tree used for storing the results of performing the Douglas-Peucker line simplification algorithm on a polyline.
Bucket query: A semi-infinite range query. It is called a "bucket" query because the query range is undefined in the upper y direction.
Cartographer: A person who creates, revises and analyses maps and geographical data.
Complexity: The number of operations needed to complete a process in a computer program. Often formulated as number of operations compared to the number of nodes that is involved in the process.
Computational Geometry: A field of research, combining computer science and geometry which involves using computers in solving geometry related problems.
Dynamic data structure: A data structure that can have nodes added to it or deleted from it after it has been constructed.
Fan-out: The count of the number of subordinates for a module, for instance the number of children of a node.
Generalization: The process of reducing the size and complexity of a spatial data set with visual quality preserved or enhanced.
GIS: Geographic Information System. A system of hardware and software used for storage, retrieval, mapping, and analysis of geographic data.
Heap: A binary tree in which every node has a value that is higher than both its children.
I/O-operation: One operation of finding the correct index and reading information from a file.

MBR: Minimal Bounding Rectangle. The smallest rectangle that contains all subdivisions of what it is the MBR of.
Node: An object in a binary tree. Can have two or less children. A node with no children is called a leaf.
Orthogonal: Axis parallel. Used here for describing axis parallel range queries.
Output sensitive complexity: The expression for the efficiency of a search algorithm where the complexity depends on the number of nodes reported from the search.
PPP Purchasing Power Parities. Currency conversion eliminating the differences in price levels between countries.
PST: Priority Search Tree. A binary tree which is a hybrid of a heap and a balanced search tree.
Query key: The criterion the nodes reported from a search must satisfy. In the orthogonal range queries covered in this thesis, the query key is four values giving the upper and lower limits of the search area in both the x- and the y-direction.
Raster: A square. Often used in connection with computer graphics, photography and television.
Recursion: See Recursion.
Semi-infinite range query: A range query where the query key is infinite in at least one direction.
Stabbing-max query: Find the largest interval that intersects the stabbing value.
Stabbing query: The term for maintaining a set of intervals so that for any given value h, all intervals that contain h can be reported efficiently.
Staircase query: A semi-infinite range query with the shape of a staircase.
Subtree: A part of a binary tree that is a binary tree it self.
SVO: Smallest Visible Object. The size of the smallest object that shall be visible after performing a simplification of a line after one of the simplification methods presented by Li and Openshaw.
Two-level data structure: A data structure, for instance a binary tree, where all the nodes are connected to another complete data structure.