Optimal binary space partitions

Clairbois, X.J.L.

Award date:
2006

TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computer Science

Master's Thesis

Optimal Binary Space Partitions

by

X.J.L. Clairbois

Supervisor: prof. dr. M.T. de Berg

Eindhoven. August 2006

Abstract

A Binary Space Partition (BSP) is a scheme for recursively dividing a configuration of objects by hyperplanes until all objects are separated. Objects can be cut into fragments by this process. We present some results on optimal size Binary Space Partitions in two dimensions.

In this thesis we show that we cannot always get an optimal BSP for a set of disjoint line segments if we only use fixed partition lines, which are partition lines that go through at least two endpoints of fragments. We also proof that the best BSP that only uses fixed partition lines cuts at most 3 times as much line segments as the optimal BSP.

We provide an algorithm that computes an optimal BSP for a rectangular subdivision in $O(n^5)$ time – using dynamic programming – and discuss some heuristics to improve this. We generalize the algorithm so it works for a larger class of optimality criteria, including size and depth. We give experimental results based on randomly generated rectangular subdivisions.

Contents

Preface

This document presents my master's thesis for the Computer Science program at the Technische Universiteit Eindhoven. The research and work was done between May 2005 and August 2006 within the Algorithms group of the Computer Science Department. My supervisor was prof. dr. Mark de Berg. Other members of the committee were dr. Bettina Speckmann and dr. ir. Alex Telea.

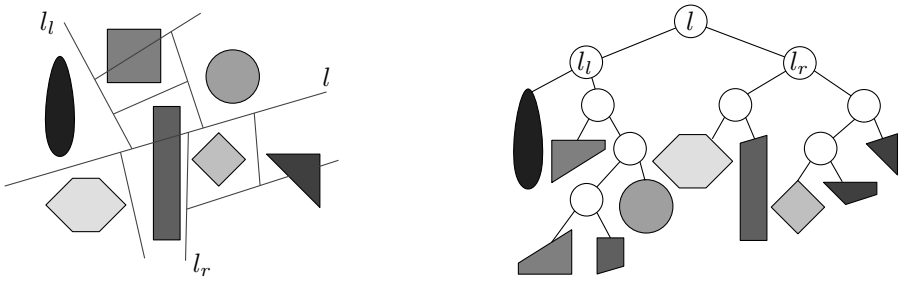
I wish to express my gratitude to Mark de Berg for his supervision during this period. Without his assistance and advice this thesis would never have been written.

I would also like to thank my girlfriend, family and friends for their support as well as during this master's thesis project as during the rest of my study.

Xavier Clairbois

1 Introduction

Many geometric problems can be solved easier if the underlying space is first partitioned into smaller subspaces. Often it is convenient to perform this partitioning in a hierarchical fashion. A popular scheme for this is a *Binary Space Partition*, or *BSP*, first introduced by Fuchs *et al.* [14]. In this scheme, one is given a set S of disjoint objects in a d-dimensional space. The space is then partitioned into two subspaces with a hyperplane. Both of the subspaces are then recursively partitioned. This continues until there is only one object – or perhaps some small number of objects – left in each subspace. Figure 1(a) shows an example of a BSP in 2-dimensional space. The line l is the first partition line. Notice that some of the objects are fragmented by the partitioning process. A natural way to model this process is by a binary tree, where the root stores the first partition hyperplane, its left child stores the partition hyperplane of the left subspace and its right child of the right subspace, and so on – see Figure 1(b). Such a tree is called a *Binary Space Partition tree*, or *BSP tree*.



(a) BSP (b) BSP tree

Figure 1: A BSP in the plane and its corresponding BSP tree.

BSPs play an important role in many application areas. In computer graphics, for example, when drawing a 3D scene one has to determine for each pixel which object is visible at that pixel; this is called *hidden-surface removal*. BSPs can be used to perform hidden-surface removal using the painter's algorithm [14]. In this algorithm the objects are "painted" in back-to-front order on the screen. So objects in the front are painted on top of objects in the back, which leads to a correct view of the scene. There can be cyclic overlap among the objects, but when they are stored in a BSP tree, a back-to-front order for the fragments in the tree can easily be obtained for any viewing direction by traversing the tree. When drawing a 3D scene, one also has to determine where shadows appear; this is called *shadow generation*. BSPs can be used to make shadow generation more efficient [5, 18]. BSPs are also frequently used in geographic information

systems (GIS) [22] and in robotics, for example for collision detection and motion planning [2, 3, 4].

The efficiency of algorithms that use BSPs depends primarily on the *size* of the BSP, that is the number of *fragments* that are created by the partitioning process. In some applications – for example when a BSP is used for point location or other queries in GIS – the depth of the BSP is important for the efficiency.

Previous work. There has been a lot of research on BSPs. For two dimensions for example, Paterson and Yao [16] proved that for any set of n line segments, there exists a BSP of size $O(n \log n)$, which can be computed in $O(n \log n)$ time. Tóth [20] showed that there exist a set of n line segments in the plane for which the smallest BSP must have size $\Omega(n \log n / \log \log n)$. De Berg *et al.* proved that for a set of fat objects, there exists a BSP of size $O(n)$ [7, 8] and that it can be checked if a BSP without any fragmentations exists in $O(n^2 \log n)$ time [9]. BSPs for line segments with a limited number of directions have also been studied [19, 21]. Some results for three dimensions are for example that there exists a BSP of size $O(n^{3/2})$ for a set of orthogonal line segments; this has been proved by Paterson and Yao [17]. BSPs for orthogonal rectangles have been considered by Agarwal *et al.* [1]. Then there are some results for multiple dimensions. For example, De Berg [7] proved that a BSP for an uncluttered scene of size $O(n)$ can be computed in $O(n \log n)$ time. BSPs for axis-parallel segments, rectangles and hyperrectangles were studied by Dumitrescu *et al.* [13]. In chapter 3 some of the existing algorithms to create BSPs for a set of disjoint line segments in the two dimensions are described in more detail.

Most of the existing algorithms for BSPs are optimized for the worst-case size BSP, not for the best size for the given input set S. This often leads to suboptimal results for the given input. For instance, while the result of Tóth [20] shows that some sets of n line segments in the plane require a BSP of size $\Omega(n \log n / \log \log n)$, it is likely that most input sets admit smaller BSPs (of linear size). Hence, we would like an algorithm that constructs, given an input set S, a BSP that is (close to) optimal for S, that is, a BSP of minimal size for S. This is the goal of our work: we want to develop algorithms that, given an input set S, construct a BSP whose size is (close to) optimal.

Our results. In chapter 4 we examine arbitrary disjoint line segments in the plane. We prove that an optimal BSP does not always exist when we use only *fixed* partition lines, which are partition lines that go through at least

two endpoints of fragments. Then we prove that there exists a BSP using only fixed partition lines that cuts no more than three times as many line segments as the optimal BSP.

In chapter 5 we give an algorithm that computes an optimal BSP for a rectangular subdivision in $O(n^5)$ time by using dynamic programming. We also present some heuristics to improve the performance of this algorithm. After this we generalize the algorithm so it can handle different optimality criteria and we give an example of an application where our algorithm is used.

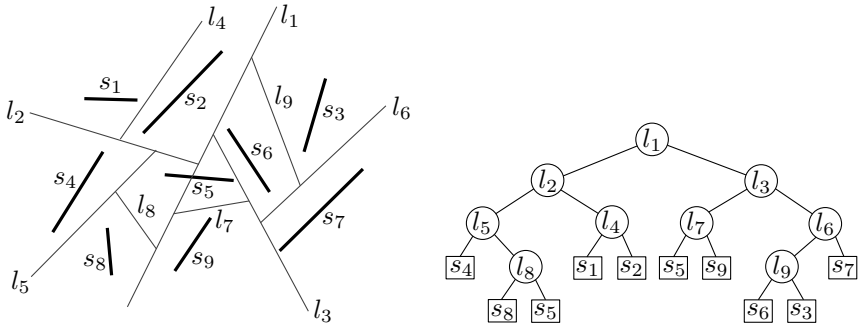
Results of experiments using this algorithm are presented in chapter 6. We show how efficient the algorithm is in practice and we see which effect the heuristics have on the running time and memory usage. In the last chapter we state our conclusions and provide notes on future work.

2 Preliminaries

In this chapter we explain some of the basic techniques, definitions, notations and terminology that we use throughout this thesis.

2.1 Basic notation and terminology

As stated in the introduction, a BSP is a recursive partitioning of space using hyperplanes such that each final region in the partitioning contains only one (fragment of) an object. In this thesis we look at BSPs where the input is a set S of n disjoint line segments in the plane. Figure 2(a) shows a BSP of such a set of disjoint line segments in the plane. We use thin lines for partition lines and denote them by $l_1 ... l_m$, where m is the total number of partition lines. Line segments are denoted by $s_1 ... s_n$ and are shown as fat line segments. In Figure 2, l_1 is the first partition line. As illustrated, objects can be cut into two or more *fragments* by the partitioning process (in Figure 2, s_5 is cut into two fragments). Line segments which are not cut during the partitioning process are also called fragments. The number of fragments of a BSP is denoted by the *size* of a BSP. For example, the size of the BSP in Figure 2 is 10. The cutting of objects can lead to a large increase of the size of the BSP. Because of this, we want schemes that partition the objects in such a way that the fragmentation of objects is minimized.



(a) BSP (b) BSP tree

Figure 2: A BSP of a set of line segments in the plane and the corresponding tree.

A BSP is naturally modeled as a binary tree, which is called a *Binary Space Partition tree*, or *BSP tree*. Figure 2(b) shows the BSP tree corresponding to the BSP in Figure 2(a). Each leaf corresponds to a region in the final partitioning. The fragment that lies in that region is stored at that leaf. Each

internal node of the tree corresponds to a region and stores a partition line that cuts that region. So the root node corresponds to the whole plane, the two nodes under the root correspond to the two regions created by the first partition line, and so on. We denote the region corresponding to the node v by $R(v)$, and we use l_v to denote the partition line stored at node v.

Free splits. One simple optimization that can be used by most BSP algorithms is to make *free splits* when possible. A free split occurs when there is a fragment that crosses a region completely. When a region is partitioned, using the fragment as the partition line, it cannot cut any other fragment in that region because all fragments are disjoint.

Figure 3 shows an example of a free split. The grey region is the region that is being partitioned; fragment s crosses that grey region completely. Hence, it is possible to make a free split along s. In this case, the fragment along which the free split is made is stored with the node.

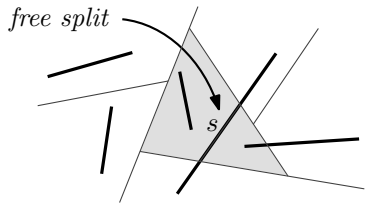


Figure 3: A free split

2.2 Types of BSPs

It is possible to distinguish different types of BSPs, depending on the restrictions put on the partition lines that they use. We consider the following types of partition lines:

- A *floating partition line* is a partition line l_v that does not go through any endpoints of fragments inside the region $R(v)$. – see Figure 4(a).
- A *semi-fixed partition line* is a partition line l_v that goes through exactly one endpoint of a fragment inside the region $R(v)$. – see Figure 4(b).
- A *fixed partition line* is a partition line l_v that goes through two or more endpoints of fragments inside the region $R(v)$ – see Figure 4(c).

- An *autopartition line* is a partition line that is an extension of a fragment. So an autopartition line contains a fragment – see Figure 4(d).
- A *rectilinear partition line* is a partition line that is horizontal or vertical, these partition lines are also called *orthogonal* partition lines – see Figure 4(e). Rectilinear partition lines are usually used when the input set S consists of only horizontal and vertical segments.

(a) Floating (b) Semi-fixed (c) Fixed (d) Autopartition (e) Rectilinear

Figure 4: Different types of partition lines

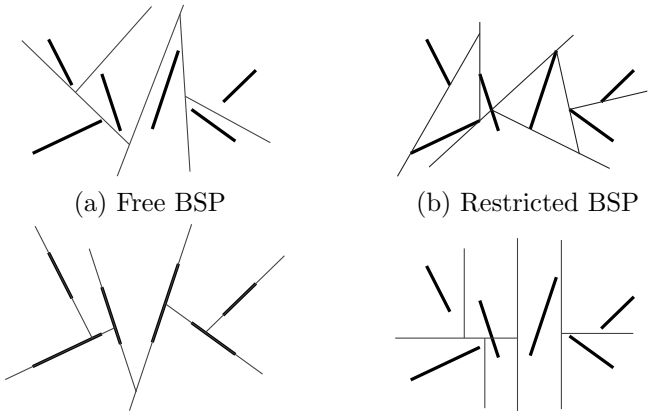
Based on these types of partition lines, we now define the following four types of BSPs.

Free BSP. In a free BSP, there are no restrictions on partition lines, that is, we can use any line as partition line. An example is given in Figure 5(a).

Restricted BSP. A restricted BSP only uses fixed partition lines. So for every node v, the partition line l_v goes through at least two endpoints of fragments of the form $s \cap R(v)$, $s \in S$ – see Figure 5(b).

Autopartition. An autopartition only uses autopartition lines. Figure 5(c) gives an example of an autopartition.

Rectilinear BSP. A rectilinear or orthogonal BSP only uses rectilinear partition lines. – see Figure 5(d).



(a) Free BSP (b) Restricted BSP

(c) Autopartition (d) Rectilinear BSP

Figure 5: Different types of BSPs

3 Existing algorithms for constructing BSPs

We describe some existing algorithms that create a BSP for a set of disjoint line segments in the plane such that one can prove good bounds on the size. In the last section we give some heuristics that do not have provable guarantees on the size of the produced BSPs, but which hopefully lead to small BSPs in most practical cases.

3.1 The trapezoid method

let V be the set of endpoints of the input set S of line segments in the plane. The trapezoid method by Paterson and Yao [16] first finds the two points of V with the maximum and the minimum y-coordinate. So all line segments lie between the horizontal lines l_{min} and l_{max} that go through these two points. Now we select an endpoint which has the median y-coordinate amongst all points of V. The horizontal line l_0 through that median splits V into two sets of points as illustrated by Figure 6.

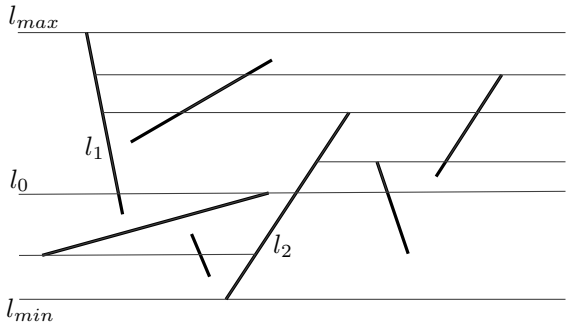


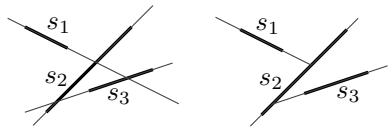
Figure 6: A BSP made by the trapezoid method

The line segments that intersect both l_{max} and l_0 can now be used to make free splits (l_1 in Figure 6). These free splits divide the region between l_{max} and l_0 into disjoint area's, and for each area we can repeat the partitioning separately until there is at most one fragment left in each region. The same applies to the region between l_0 and l_{min}. Paterson and Yao [16] proved that this method gives the following result.

Theorem 3.1. [16] *The size of a BSP created by the trapezoid method is* $O(n \log n)$ *and it can be created in* $O(n \log n)$ *time.*

3.2 A randomized algorithm for autopartitions

A method that is often used to construct a BSP for a set of disjoint line segments in the plane is a randomized algorithm for autopartitions. The problem with autopartition algorithms is that the right order to do the autopartitions has to be chosen: different orders can lead to different size of BSPs, as illustrated by Figure 7.



(a) size is 6 (b) size is 3

Figure 7: Different orders give BSPs of different sizes. In (a), the order s_1, s_3, s_2 is used, while in (b) the order s_2, s_1, s_3 is used.

It is proved that if we use a random segment to do the partitioning, we do well in the expected case. This algorithm can be improved by making free splits when possible. So the randomized algorithm for autopartitions makes free splits whenever it is possible, and otherwise it chooses a random line segment for the partitioning.

The expected size of a BSP created by this randomized algorithm for autopartitions is $O(n \log n)$, and the expected construction time is $O(n^2)$ as shown by Paterson and Yao [16]. We summarize their results in the following theorem.

Theorem 3.2. [16] *A BSP for a set of disjoint line segments in the plane of size $O(n \log n)$ can be computed with a randomized algorithm for autopartitions in $O(n^2)$ time.*

3.3 Orthogonal partition algorithm

Next we present an algorithm by Paterson and Yao [16] for orthogonal line segments in the plane. First we compute the bounding rectangle R for the set of line segments. A segment is said to be *anchored* on a side of the bounding rectangle if one of its endpoints lies on a side of R and the other endpoint in the interior of R. Let R_l and R_r denote the sets of line segments anchored on the left and right side respectively. Similarly, denote the sets anchored on the top and the bottom as R_t and R_b.

A *T-decomposition* of R is defined as follows. Let s_1 be the longest segment from R_l. Suppose that the line containing it intersects with some segments of R_t or R_b. Let s_2 be the first (i.e. leftmost) segment it intersects. We decompose R into three rectangles by first splitting along the line containing s_2, and then splitting the area to the left of s_2 along the line containing s_1 as shown in Figure 8.

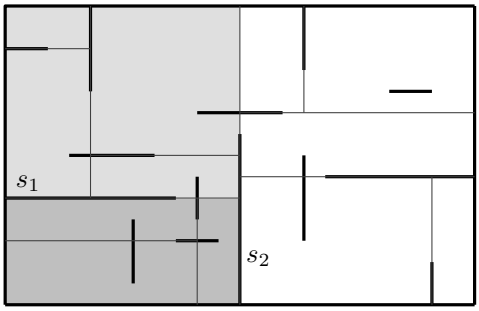


Figure 8: Orthogonal partition algorithm. The shading indicates the three regions resulting from the T-decomposition.

The partitioning of the set of line segments takes place by recursively making free splits or applying T-decompositions (when there is no free split possible). To ensure that each segment is cut at most a constant number of times (when doing T-decompositions), we rotate each region if needed. To do this, we attach a label to each anchored segment. If an unanchored segment gets cut, the two fragments both become anchored and get the label *green*. When a *green* segment is cut, the part between the cut and the place where it is anchored is no longer anchored – it is a free split – while the other remaining part is made *red*. It is proved that if the right side of the bounding rectangle has no *red* segment anchored and we apply a T-decomposition, each new region has at least one side without a *red* segment anchored. So we can rotate these new regions so that the right side has no *red* segment and we can apply another T-decomposition. Figure 8 shows an example partitioning obtained by this partitioning scheme.

This algorithm runs in $O(n^2)$ time. The size of this partitioning is at most $4n$, because any original segment can form at most two *green* segments and each green segment can only be cut once more. This is proved by Paterson and Yao in [16], where we can find a more detailed description of the algorithm. D'Amore and Franciosa [6] improved this algorithm with a slightly better constant for the combinatorial bound.

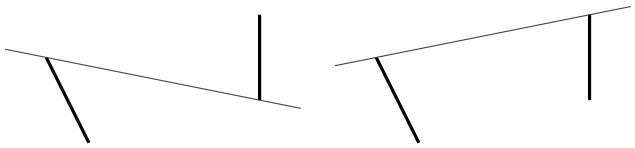
Theorem 3.3. [16] *For any set of n disjoint axis-parallel line segments in the plane, we can compute a BSP of size $O(n)$ in $O(n^2)$ time .*

3.4 Perfect BSPs

Here we describe an algorithm by de Berg *et al.* [9] that constructs a perfect BSP, or decides that no perfect BSP exists. A BSP is called *perfect* if none of its n line segments is cut by the partition lines, or in other words, when its size is exactly n.

First we explain the main idea of this algorithm. If we have a perfect BSP, no partition line cuts a line segment. We can then rotate each partition line until it is fixed without introducing any intersections, resulting in a restricted BSP. So first we create a set of all possible fixed partition lines and then we use that to create the BSP.

For this, we need a visibility graph. A visibility graph for a set S of line segments is a graph whose vertices are the endpoints of the given line segments and where two vertices u, v are connected by an edge if the line segment uv, that connects the two vertices, does not intersect the interior of any of the input line segments.

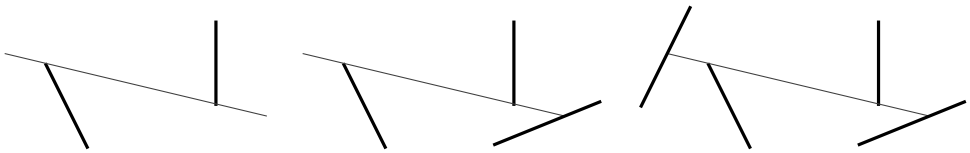


(a) inner tangent (b) outer tangent

Figure 9: An inner and an outer tangent edge.

We view this visibility graph as if it were a geometric structure. If there is a perfect BSP, then there is one that only uses fixed partition lines that are the extension of an inner tangent edge – see Figure 9(a). So partition lines of the set of line segments can be found by extending all inner tangent edges of the visibility graph; outer tangent edges – see Figure 9(b) – can be removed from the visibility graph. Then we extend each edge to both sides, with the restriction that it may not intersect any of the given line segments. By doing this we get lines, halflines and line segments – see Figure 10. Halflines and line segments are called *candidate partition lines*, lines are called partition lines.

Next we apply the following recursive algorithm. At each stage we choose a partition line from the set of partition lines and split the space into two subspaces. We divide the set of line segments between these two subspaces. We also divide our set of partition and candidate partition lines between the two subspaces. Some of the candidate partition lines can become a partition line for a new subspace, because they do not cut any line segment in that



(a) line (b) halfline (c) segment

Figure 10: A line, a halfline and a segment.

subspace. We keep doing this until there is only one line segment left in each subspace – we have found a perfect BSP – or until the set of partition lines in a subspace is empty and there is more than one line segment left in that subspace. In the latter case the partitioning can be ended and we conclude that there exists no perfect BSP for the set of given line segments. This algorithm runs in time $O(n^2 \log n)$, as proved in [9].

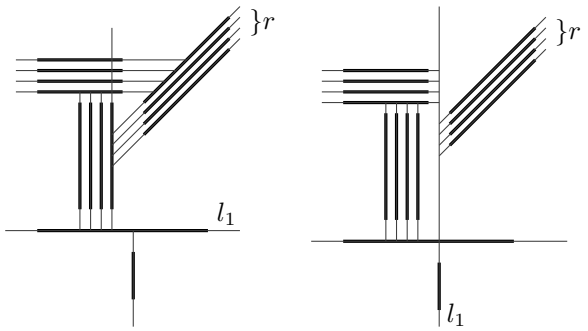
Theorem 3.4. [9] *If a perfect BSP exists for a set of disjoint line segments in the plane, it can be constructed in* $O(n^2 \log n)$ *time.*

3.5 Heuristics

There are a lot of heuristics for BSPs. In this section we briefly mention two heuristics that can be used to autopartition disjoint line segments in the plane.

One greedy algorithm is to always take the partition line that intersects the least line segments. This can lead to a bad result, as illustrated by Figure 11. It shows a set consisting of three subsets of r parallel line segments each, and two other line segments. The greedy BSP of (a) has size $4r + 2$ but, as shown in (b), it is also possible to get a BSP of size $3r + 3$.

Another heuristic is to find the line segment with the largest chance of being cut. The line on that line segment is then chosen as the next partition line to prevent it from being cut. So it tests each line segment that is candidate for the next partition line to see how much it is cut by the other candidate partition lines. Then the algorithm chooses the line containing the segment with the most potential cuts as the next partition line. Like most heuristics, there are cases in which this algorithm performs badly. For example, consider Figure 11 again. The lowest horizontal line segment gets intersected the most, so the heuristic chooses the line through that segment as the first partition line. If we continue using this algorithm, we get a BSP of at least size $4r + 2$ again, just like with the greedy algorithm.



(a) Greedy autopartitioning (b) Optimal autopartitioning

Figure 11: A greedy and an optimal autopartitioning

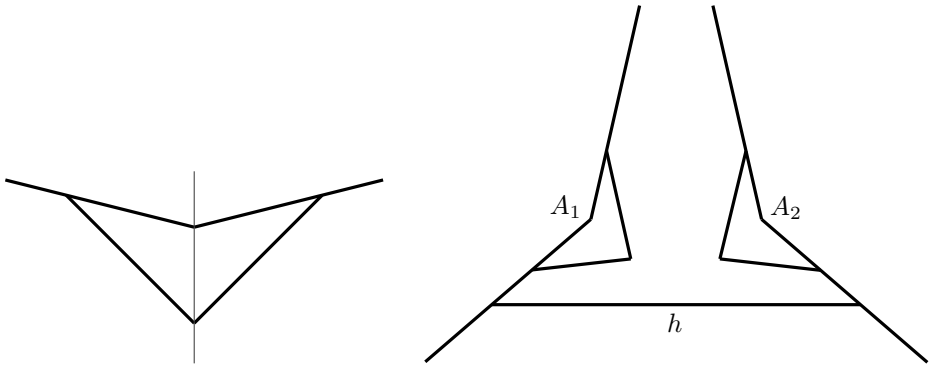
Of course there are more heuristics possible (e.g. see [15]). However, there are always input sets for which they work badly.

4 Optimal BSPs for arbitrary line segments

In this chapter we consider the problem of computing BSPs for a given set S of (arbitrarily oriented) disjoint line segments in the plane. We are interested in optimal BSPs, that is, BSPs with minimal size.

4.1 Optimal BSP using fixed partition lines

In [9] De Berg *et al.* made the following observation: each partition line of a set S of n line segments in the plane that does not cut any of these line segments and that partitions the set into two non-empty subsets, can be slightly rotated until it touches two lines segments each at a side. This means that to construct a perfect BSP for S we can restrict our attention to fixed partition lines. One might think that this is also true for optimal BSPs, that is, that an optimal restricted BSP is also optimal for the class of free BSPs. Unfortunately this is not true. We prove this by a counterexample that shows that we cannot always get an optimal free BSP when we only use fixed partition lines.



(a) Construction A (b) Construction B

Figure 12: (a) Construction A: A set of line segments with only 1 possible partition line that does not cut a line segment. (b) Construction B: A set of line segments that cannot be partitioned without cutting a line segment.

Figure 12(a) shows a set of four line segments – we call it construction A – and one partition line. This is the only possible partition line that does not cut any of the four line segments and still has segments on both sides. The next figure, Figure 12(b) – construction B – is a set of line segments that is constructed using two constructions A, called A_1 and A_2, and a line segment h. If we take the only possible partition line from A_1, we see that it cuts

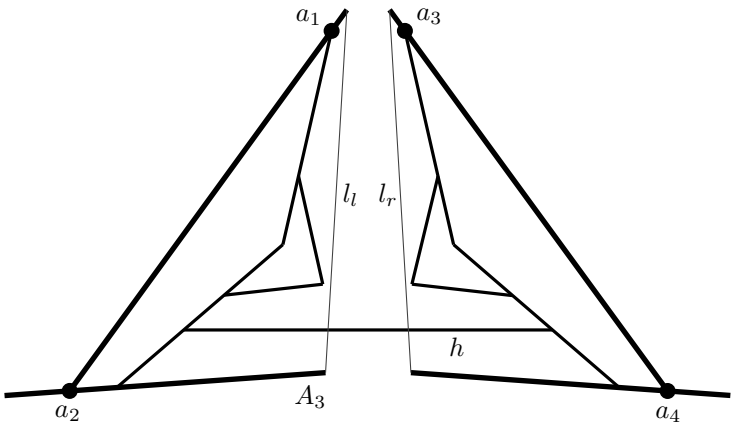


Figure 13: Construction C: A construction that has to be intersected at least once between l_l and l_r.

h. The same happens with the partition line from A_2. Moreover, h cannot be separated from A_1 and A_2 by a partition line that does not cut any line segment. Hence, construction B cannot be partitioned without cutting at least one line segment.

Next we put another larger construction A, called A_3, with some bigger gaps at the top and the bottom, around construction B. We get the construction with 13 segments as shown in Figure 13 – construction C, where the fat line segments form A_3.

Lemma 4.1.
(i) There exists a free BSP of size 14 for construction C.
(ii) Any restricted BSP for construction C has at least size 15.

Proof. (i) We show that there exists a free BSP of size 14 for construction C. Since there are 13 line segments, it means that only one line segment can be cut to get a BSP of size 14. Figure 14 shows that it is indeed possible to construct a BSP that cuts only one line segment, namely h.

(ii) To prove *(ii)*, we argue that any restricted BSP cuts at least two line segments (or cuts one line segment at least twice). Suppose for a contradiction that there is a restricted BSP that makes at most one cut. It is easy to see that the first partition line has to cut a line segment, because there is only one way to partition construction A_3 without intersecting one of its line segments. More precisely, the only way to partition A_3 without intersecting one of its segments is by a partition line in between l_l and l_r. Such a partition line cuts segment h.

Because it is only allowed to cut one line segment and the first partition line has to cut at least one line segment, the first partition line has to cut exactly one line segment.

The first partition line can cut a line segment from structure A_3 or it can cut a line segment from structure B. If it cuts structure A_3 it is not allowed to cut structure B. If we want to cut structure A_3 once, we can only do this by cutting structure A_3 above a_1 or a_3, to the left of a_2, or to the right of a_4 as shown in Figure 13. When we do this, one of the subproblems cannot be partitioned without making another cut. Hence, we do not want to cut structure A_3. The only way to do that is with a partition line between l_l and l_r. This cuts structure B once in h.

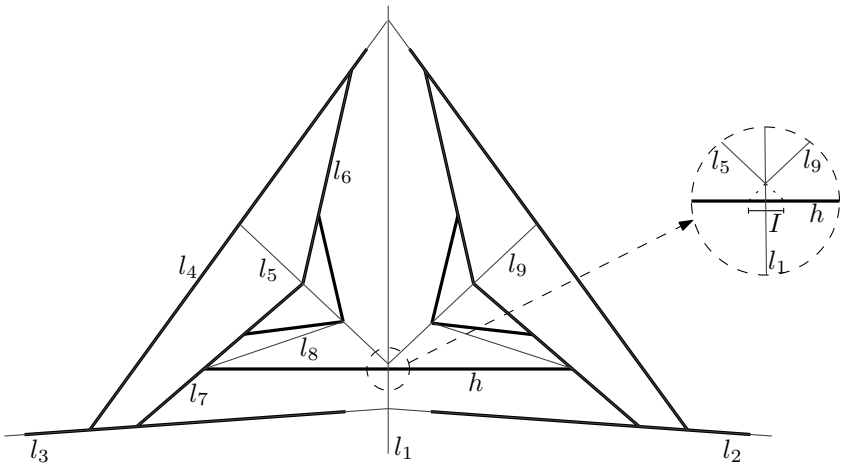


Figure 14: A partitioning of C

Figure 14 shows a partitioning of construction C of size 14. The first partition line is l_1, which has to go between l_l and l_r – see Figure 13. The next two partition lines are l_2 and l_3. It is not difficult to check that these are the only choices if we do not want to make another cut.

We only consider the left side of the construction, since the right side is the mirror image of the left side. After partition line l_3, the cuts are made at successively l_4, l_5, l_6, l_7 and l_8. For all those partition lines we do not have any other choice if we do not want any additional cuts and we want them to be fixed partition lines.

The partition lines that can be seen at Figure 14 all are fixed except partition line l_1. I is the line segment between the intersection of l_5 and h and the intersection of l_9 and h as shown in the right image in Figure 14. If we rotate partition line l_1 until it is fixed, it will not go through I anymore and then

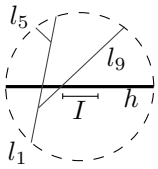


Figure 15: When l_1 is rotated a little

l_5 or l_9 cut h – as illustrated by Figure 15 – and we get two cuts. In order to get an optimal partitioning l_1 has to go through I and cannot be fixed. The optimal restricted BSP cuts two line segments and has size 15. □

Corollary 4.2. *There are sets of disjoint line segments in the plane for which an optimal restricted BSP is not an optimal free BSP.*

4.2 Approximating an optimal BSP

We now know that one cannot always get an optimal free BSP when only fixed partition lines are used. This makes it very hard to compute an optimal free BSP, because it is unclear which partition lines to use. In this section we therefore investigate if we can approximate an optimal free BSP using only fixed partition lines.

To this end, we consider an optimal free BSP. We want to rotate all floating partition lines to make them fixed. All floating partition lines that do not cut any line segment can be rotated until they are fixed without introducing any more cuts. The problem is that we also want the floating partition lines that cut line segments to be fixed. This can introduce extra fragments – as we saw in the previous section. Thus the goal is to move the floating partition lines in such a way that they become fixed, while trying to keep the number of extra fragments introduced as small as possible.

To achieve this we go through all partition lines in the BSP tree in top-down order. Each floating partition line that does not cut a line segment is rotated until it is fixed. When we encounter a floating partition line l that cuts at least one line segment, we perform the seven steps described below, which introduce some new partition lines. While describing this method we illustrate it with the example BSP from Figure 16(a). Figure 16(b) shows the results of making the first partition line l_1 fixed.

Step 1: Duplicate the floating partition line l and call the new partition lines l_a and l_b.

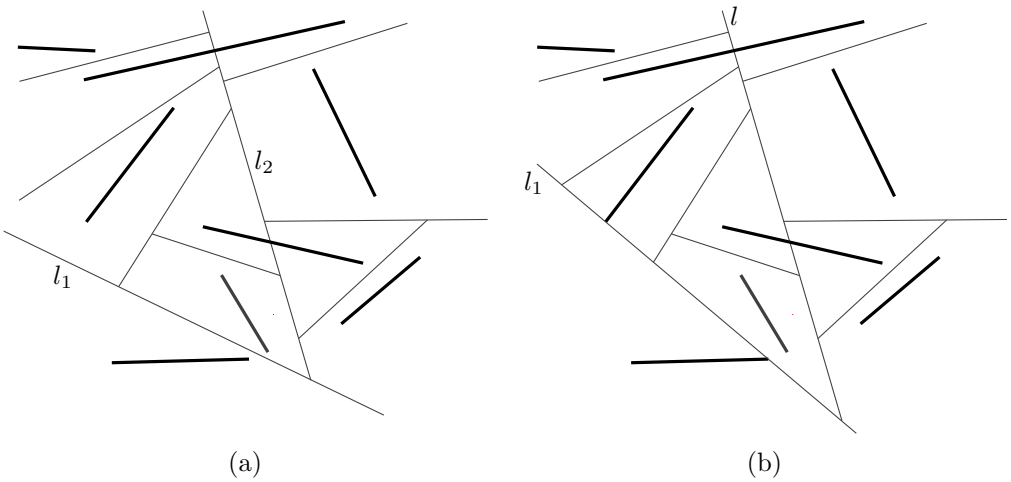


Figure 16: (a) shows a free BSP and (b) shows the BSP where l_1 has been made fixed and the next partition line that we want to be fixed is l.

Step 2: Rotate l_a clockwise around an arbitrary point on the line l until it hits an endpoint of a fragment which we call p_a^1. Then rotate l_a clockwise around p_a^1 until it hits a second endpoint of a fragment which are called p_a^2. l_a is fixed then. We can see how this looks for our example in Figure 17(a).

Step 3: Cut l_b into two at point p where l_a intersects l_b. One half is called l_b^1 and the other l_b^2.

Step 4: If l_b^1 does not cut a fragment we remove l_b^1. If it cuts a fragment we rotate it counterclockwise around p until it hits an endpoint of a fragment which we call p_b. We can see this applied to our example in Figure 17(b).

Step 5: l_b^1 is divided into two parts by p_b. With l_c^1 we denote the part between p and p_b. With l_c^2 we denote the other part. There are three options now.

(i) If l_c^1 cuts a fragment and l_c^2 does not, then rotate l_b^1 clockwise around p_b until it hits an endpoint of a fragment.

(ii) If l_c^2 cuts a fragment and l_c^1 does not, then rotate l_b^1 counterclockwise around p_b until it hits an endpoint of a fragment.

(iii) If l_c^1 and l_c^2 both cut a fragment, then duplicate the part of l_b^1 which we call l_c^1 and denote it by l_c. Then rotate l_b^1 clockwise around p_b until it hits an endpoint of a fragment. Next rotate l_c counterclockwise around p_b until it hits an endpoint of a fragment. In our example l_c^1 and l_c^2 both

partition a fragment, so we rotate l_b^1 clockwise and l_c counterclockwise around p_b and get the result that is shown in Figure 18(a).

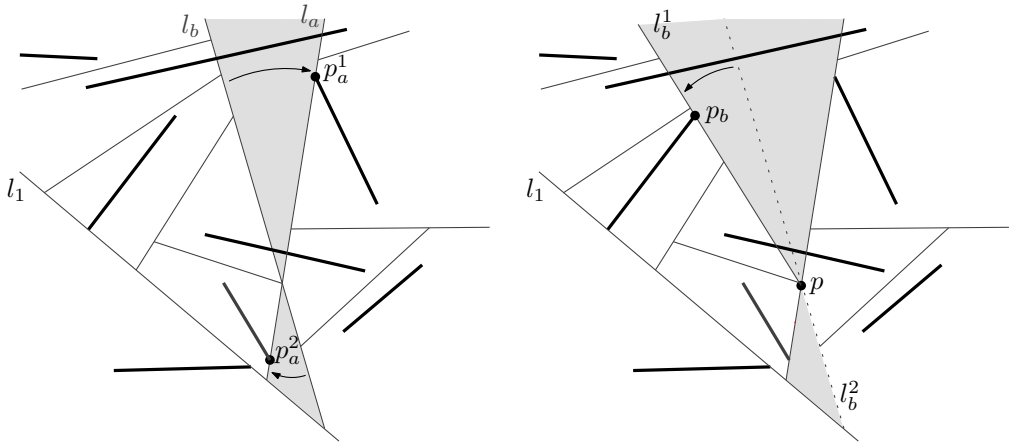


Figure 17: (a) shows the BSP after step 2 and (b) shows the BSP after step 4.

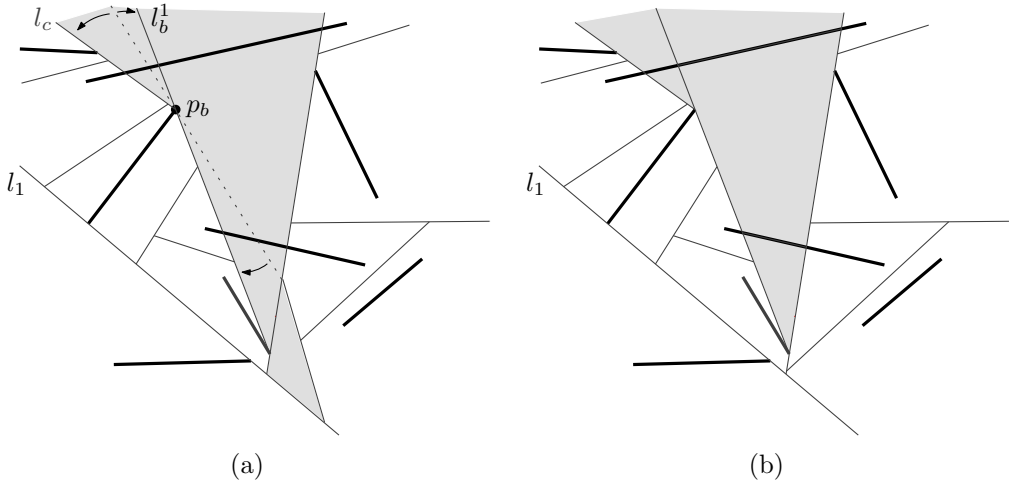


Figure 18: (a) shows the BSP after step 5 and (b) shows the BSP after step 6.

Step 6: Repeat step 4 and 5, but now with l_b^2 instead of l_b^1. In our example this means that we have to remove l_b^2, since l_b^2 does not cut a fragment and we get the result that is showed in Figure 18(b).

Step 7: For each fragment that was cut by l, there is a fragment of that fragment that crosses the whole grey area now. We have to make free splits for all those fragments. In our example this means we make 3 free splits.

After we have done these steps in our example we can continue with rotating partition lines that do not cut line segments. Since in our example no other partition line cuts a line segment, we do not have to go through the steps described above again. Figure 19 shows the restricted BSP that we have created this way.

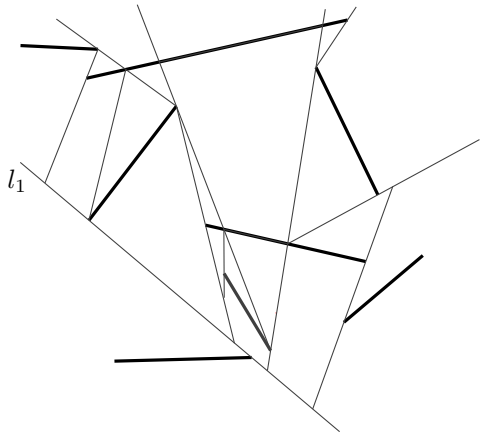


Figure 19: A restricted BSP created from the free BSP from Figure 16(a)

Lemma 4.3. *When the optimal free BSP of a set of n disjoint line segments has a size of $n + i$, the method described above gives a restricted BSP with a maximum size of $n + 3i$.*

Proof. The idea of this method is that we create an area – the grey area in the example figures – that does not has any endpoints of line segments in it. We have to make sure that the intersection points between l and the line segments it cuts are kept inside the grey area. This way no other partition line can suddenly cut one of these line segment – as happened in the previous section, see Figure 15 – and we can prevent a large increase in the number of fragments. We create this area by replacing l with a couple of fixed partition lines in the way the method describes.

The method only rotates partition lines until they hit an endpoint of a fragment. So this cannot introduce extra cuts. The only thing that can introduce extra cuts is the duplicating of partition lines.

This happens at the point where we duplicate l and a part of l_b^1 and l_b^2. In the worst case we duplicate a part of both lines l_b^1 and l_b^2. Both lines originate from the line l_b and they are both duplicated from another part of l_b. They cannot overlap, since they are both on another side of p. It is impossible for a line

segment to intersect l_b twice, so l_b^1 and l_b^2 cannot cut the same line segment. Their rotating does not introduce cuts, because they are rotated until they hit an endpoint of a fragment. Hence, in total we can only get three times as much cuts as l had in the beginning. In the worst case, all i cuts in the optimal free BSP are made with floating partition lines. So we get a maximum of $3i$ cuts in the restricted BSP.

With this method we rotate each floating partition line until it is semi-fixed, and we rotate each semi-fixed partition line around the endpoint of a fragment that it goes through until it is fixed, so the final BSP is restricted. □

5 Optimal BSPs for rectangular subdivisions

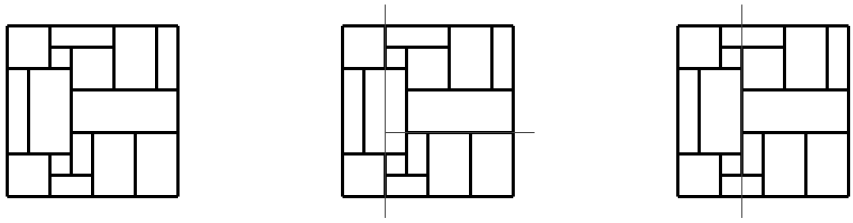
A *rectangular subdivision* is a subdivision of a rectangle into a set R of smaller rectangles – see Figure 20(a) for an example. We assume without loss of generality that all edges of the rectangles are horizontal or vertical. In this chapter we present an algorithm to compute an optimal BSP for a rectangular subdivision, where we assume that we can only use horizontal and vertical partition lines. When constructing a BSP for a rectangular subdivision, we consider the rectangles of the subdivision as the objects. Thus we want to construct a BSP for the subdivision such that each final region in the BSP is contained in a single rectangle of the subdivision.

Notice that partition lines from an optimal partitioning always lie on the edges of rectangles. If for example a horizontal partition line does not lie on a horizontal rectangle-edge, we can move it vertically until it lies on a horizontal rectangle-edge. When we do this some vertical partition lines get extended. But before they can cut a new rectangle we encounter a horizontal rectangle-edge and we stop, so they cannot introduce new cuts. The horizontal partition line that we are moving cannot introduce more cuts, because it first meets a horizontal edge before it can start partitioning a new rectangle. This is similar for vertical partition lines.

5.1 Greedy algorithm

The first algorithm that comes to mind is a greedy algorithm. We check all possible places for partition lines – all horizontal and vertical edges of all rectangles – then we take one that cuts the fewest rectangles, and recurse on the resulting subproblems. In Figure 20(b) the first partition line is the vertical line that only cuts one rectangle; all other possible partition lines cut at least two rectangles. On the right side of that partition line there are two possibilities for the next partition line that only cuts one rectangle; all other options cut more rectangles. It does not matter which one is chosen since it is symmetric. The next partition line in the upper right region has to be a partition line that cuts one rectangle. All remaining partition lines do not cut any rectangles, so in total this greedy partitioning cuts three rectangles.

In Figure 20(c) an optimal partitioning of this rectangular subdivision is shown. Here another partition line is chosen as the first partition line. This partition line cuts two rectangles. After that, it is possible to choose all remaining partition lines so that they do not cut any rectangle. Hence, this partitioning cuts only two rectangles.



(a) Rectangular subdivision (b) Greedy partitioning (c) Optimal partitioning

Figure 20: A rectangular subdivision with a greedy partitioning and an optimal partitioning

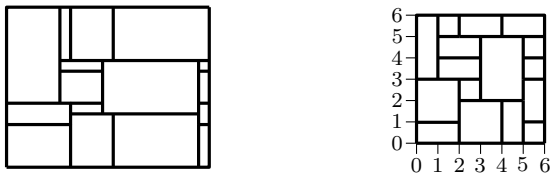
So a greedy algorithm will not always give the best solution and we have to look for another algorithm.

5.2 Dynamic programming

Instead of using a greedy approach, we can apply a brute-force approach, that is, check all possible BSPs. This is finite, because we only have a finite number of places to place partition lines – only on edges of rectangles. But this requires exponential time. So we are looking for a better solution.

Another approach is dynamic programming. The first reason why we think about dynamic programming is because this problem has optimal substructures: If we take a subproblem of the optimal solution – a smaller region that has to be partitioned – then we need an optimal partitioning for this subproblem to get the optimal partitioning of the total problem. The second reason is that this problem has overlapping subproblems: if we have a recursive algorithm for this problem, it solves the same subproblems over and over again. One small region can be a subproblem of a lot of different BSPs. A dynamic programming algorithm takes advantage of this by solving each subproblem only once. The solution is stored in a table, so it is available next time.

Representing the subdivision. First we describe how we represent the rectangular subdivision. We are only interested in the places where we can put partition lines and in how much rectangles they cut. So the exact size of the rectangles does not matter for our algorithm. For example, Figure 21(a) and (b) have the same optimal partitioning. In the second figure, we replaced each coordinate by its *rank*. We call this the *normalized subdivision*. We use n_x to denote the maximum rank among the x-coordinates, and n_y to



(a) Rectangular subdivision (b) The normalized version of (a).

Figure 21: A rectangular subdivision and its normalized counterpart.

denote the maximum rank among the y-coordinates. We store this normalized subdivision in two arrays, an array V for the vertical edges and an array H for the horizontal edges. The first element of V is another array with all vertical edges at horizontal rank 0 in Figure 21(b). The second element of V is an array of all vertical edges at rank 1, etc. We represent a vertical edge by an array of two elements, the horizontal coordinate of its begin and endpoint. If two edges touch each other, we merge them to one longer edge. Array H contains all horizontal edges and is defined similarly. The arrays for Figure 21(b) then look like this:

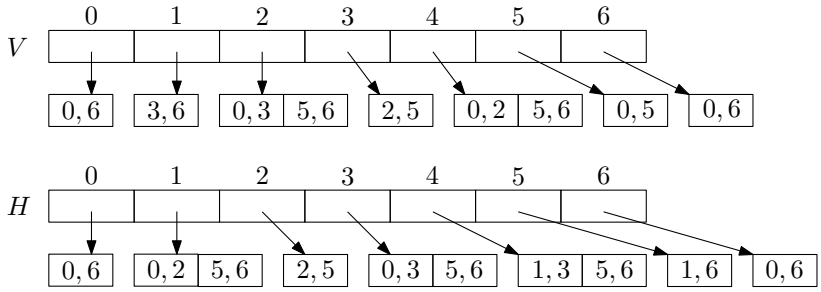


Figure 22: Representation of Figure 21(b) with arrays V and H

Recursive definition of an optimal solution. Define $R_{i,j,k,l}$ as the part of the given subdivision inside the rectangle with its vertical edges on rank i and j and its horizontal edges on rank k and l – see Figure 23. For integers i, j, k, l with $0 \leq i < j \leq n_x$ and $0 \leq k < l \leq n_y$ we define:

$$A[i, j, k, l] = \text{number of regions in an optimal BSP for } R_{i,j,k,l}$$

The number of regions in an optimal BSP for the whole rectangular subdivision $R_{0,n_x,0,n_y}$ would then be $A[0, n_x, 0, n_y]$.

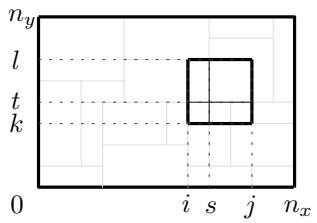


Figure 23: A rectangular subdivision $R_{0,n_x,0,n_y}$ with a subproblem $R_{i,j,k,l}$. s and t are potential places to put the next partition line.

We can define $A[i,j,k,l]$ recursively as follows. If $i = j-1$ and $k = l-1$, the rectangle $R_{i,j,k,l}$ has no more edges inside, so $A[i,i+1,k,k+1] = 1$. Otherwise, the optimal BSP either uses a vertical partition line through a vertical edge on rank s for some $i < s < j$ or a horizontal partition line through a horizontal edge on rank t for some $k < t < l$.

There are only $j-i-1$ possible choices for s and $l-k-1$ possible choices for t – see Figure 23; we need to check them all to find the best. So our recursive definition for the minimum number of cuts becomes:

$$A[i,j,k,l] = \begin{cases} 1 & \text{if } i = j-1 \text{ and } k = l-1 \\ \min \begin{cases} \min\limits_{i<s<j}(A[i,s,k,l] + A[s,j,k,l]) \\ \min\limits_{k<t<l}(A[i,j,k,t] + A[i,j,t,l]) \end{cases} & \text{if } i < j-1 \text{ or } k < l-1 \end{cases}$$

A recursive algorithm that is based on this recurrence takes exponential time, which is not any better than the brute-force method of checking each possible BSP. An observation that we make is that we have relatively few subproblems: one for each choice of i, j, k and l satisfying $0 \le i < j \le n_x$ and $0 \le k < l \le n_y$. Hence, there are $\Theta(n_x^2 n_y^2)$ subproblems. A recursive algorithm may encounter each subproblem many times in different branches of its recursion tree (overlapping subproblems), so we store the values in a table. Since we have four variables, we have a 4-dimensional table.

The algorithm. We provide a dynamic programming algorithm that computes the number of final regions in an optimal partitioning. First we have the algorithm *MemoizedPartitioning*. This algorithm fills the table $A[0..n_x - 1, 1..n_x, 0..n_y - 1, 1..n_y]$ with the value ∞ initially to indicate that the entry has yet to be filled in. Then it calls the recursive algorithm *LookupPartitioning* that fills in the table. After we have described and explained these algorithms,

we present an algorithm that computes an optimal BSP itself, instead of only the number of regions in an optimal solution.

```
Algorithm MemoizedPartitioning(V, H)
1.  n_x ← length(V)
2.  n_y ← length(H)
3.  for i ← 0 to n_x − 1
4.      do for j ← i + 1 to n_x
5.          do for k ← 0 to n_y − 1
6.              do for l ← k + 1 to n_y
7.                  do A[i, j, k, l] ← ∞
8.  return LookupPartitioning(0, n_x, 0, n_y)
```

```
Algorithm LookupPartitioning(i, j, k, l)
1.  if A[i, j, k, l] < ∞
2.    then return A[i, j, k, l]
3.  for s ← i + 1 to j − 1
4.      do for x ← 0 to length(V[s]) − 1
5.          do if (V[s][x][0] < l and V[s][x][1] > k)
6.              then q ← LookupPartitioning(i, s, k, l)
                       + LookupPartitioning(s, j, k, l)
7.                   A[i, j, k, l] ← min(q, A[i, j, k, l])
8.  for t ← k + 1 to l − 1
9.      do for y ← 0 to length(H[t]) − 1
10.         do if (H[t][y][0] < j and H[t][y][1] > i)
11.             then q ← LookupPartitioning(i, j, k, t)
                         + LookupPartitioning(i, j, t, l)
12.                  A[i, j, k, l] ← min(q, A[i, j, k, l])
13. if A[i, j, k, l] = ∞
14.   then A[i, j, k, l] ← 1
15. return A[i, j, k, l]
```

The algorithm *LookupPartitioning* first checks if $A[i, j, k, l]$ has already been computed. If this is the case ($A[i, j, k, l] < \infty$), it returns the value stored in the table (line 1-2). At line 3 a loop starts that goes through all possible ranks s for vertical partition lines. It checks all line segments in $V[s]$ (line 4). Here it checks if at least a part of the line segment lies within the subproblem. If it does it computes the subproblems on both sides of that partition line separately. After that there is a check if it is better than the best solution until then. If it is, the value is stored in $A[i, j, k, l]$ (line 5-7).

The next lines (line 8-12) do exactly the same for the horizontal partition lines. Then it checks if a partitioning has been found, or if $A[i, j, k, l]$ is still ∞. If it is, it means that there are no line segments within this subproblem, and there are no partition lines needed for this subproblem. The number of regions in an optimal BSP for this subproblem is then 1 (line 13-14). Finally $A[i, j, k, l]$ is returned.

We do not have an optimal BSP at this moment, we only have the size of an optimal BSP. We make a little adjustment first that we need to find the optimal BSP. Every time we store a value in $A[i, j, k, l]$, we store the rank (s or t) together with a number to indicate if it is a vertical or a horizontal partition line in another table $B[i, j, k, l]$. We do this by multiplying the rank with 10, and then adding 1 if it is vertical and 2 if it is horizontal. This way we only need to store one integer each time. When 1 is stored in $A[i, j, k, l]$ – it cannot be partitioned further – we store 0 in $B[i, j, k, l]$.

We can find the optimal BSP by calling the recursive algorithm *GetOptimalBSP*$(0, n_x, 0, n_y)$. This algorithm is given below.

Algorithm *GetOptimalBSP*(i, j, k, l)
1. **if** $(B[i, j, k, l] \neq 0)$
2. **then if** $(B[i, j, k, l] \mod 10 = 1)$
3. **then** $s \leftarrow (B[i, j, k, l] - 1)/10$
4. Partition $R_{i,j,k,l}$ on the vertical line on rank s
5. *GetOptimalBSP*(i, s, k, l)
6. *GetOptimalBSP*(s, j, k, l)
7. **if** $(B[i, j, k, l] \mod 10 = 2)$
8. **then** $t \leftarrow (B[i, j, k, l] - 2)/10$
9. Partition $R_{i,j,k,l}$ on the horizontal line on rank t
10. *GetOptimalBSP*(i, j, k, t)
11. *GetOptimalBSP*(i, j, t, l)

Running time. Next we analyze the running time of this algorithm. The number of rectangles in the rectangular subdivision is denoted by n. Then $n_x = O(n)$ and $n_y = O(n)$. Hence, the table that the algorithm fills has $O(n^4)$ entries.

Each of the $O(n^4)$ table entries is initialized once in line 7 of *MemoizedPartitioning*. In *LookupPartitioning* we go through all horizontal and vertical edges at most once, which takes $O(n)$ time. It also needs the results from other table entries, but each table entry is computed only once. After that the value is given directly from the table instead of computing it again. So we have to

call *LookupPartitioning* – which takes $O(n)$ time – for each of the $O(n^4)$ table entries only once. *GetOptimalBSP* does not go through a table entry more than once, so it takes less time than the filling of the table.

Theorem 5.1. *An optimal BSP of a rectangular subdivision with n rectangles can be computed in $O(n^5)$ time.*

5.3 Heuristics

In this section we present some heuristics that can improve the running time of the algorithm from last section.

Free splits. A heuristic that comes to mind is to make free splits whenever it is possible. A free split never cuts a rectangle, therefore we can always make such a free split without losing optimality. This means that we do not have to try any other option for a partition line when we can make a free split. Because we want to know if it is possible to make a free split before we start solving subproblems, we first go through all possible positions for partition lines and check if we can make a free split. If we cannot make a free split, we proceed as before.

We further improve this, because if we do not find a free split, we go through all positions to place partition lines twice. To prevent this, while we are looking for free splits at a certain position, we also check if that position has a line segment inside the subproblem we are looking at. We store it in an array W if it does. If we then do not find any free splits, we only have to go through the array W instead of through all line segments. The first loop (line 3-7) from the algorithm *LookupPartitioning* from previous section then looks as follows:

```
1.  c ← 0
2.  for s ← i + 1 to j − 1
3.      do for x ← 0 to length(V[s]) − 1
4.             do if (V[s][x][0] ≤ k and V[s][x][1] ≥ l)
5.                    then A[i, j, k, l] ← LookupPartitioning(i, s, k, l)
                           + LookupPartitioning(s, j, k, l)
6.                         return A[i, j, k, l]
7.             else  if (V[s][x][0] < l and V[s][x][1] > k)
8.                      then W[c] ← s
9.                           c ← c + 1
```

This code is for the vertical partition positions; the horizontal positions are handled similarly. Then, after these two loops, the following code is inserted to try all vertical partition places that are stored in W.

```
1.  for i ← 0 to c − 1
2.      do q ← LookupPartitioning(i, W[i], k, l)
           + LookupPartitioning(W[i], j, k, l)
3.         A[i, j, k, l] ← min(q, A[i, j, k, l])
```

For the horizontal positions a similar loop is inserted.

Pruning. Another heuristic that can be applied is *pruning*. This means that we avoid solving subproblems of which we know that they cannot lead to optimal results. We do this by first computing the size z of the greedy BSP – which can be done in $O(n^2)$ time. With n we denote the number of rectangles in the rectangular subdivision. We define g to be $z - n$, which is the number of times a rectangle is cut by the greedy partitioning. We add g to the parameter list of *LookupPartitioning*. The first call is then *LookupPartitioning*$(0, n_x, 0, n_y, g)$. We need to change the definition of $A[i, j, k, l]$. Instead of the number of regions in an optimal BSP for $R_{i,j,k,l}$, we define $A[i, j, k, l]$ as the number of times a rectangle is cut by the partitioning.

Every time we try a partition line s, we calculate how much rectangles that partition line cuts – which we denote by $cuts(s)$. The first two lines of the algorithm *LookupPartitioning* from the previous section then changes to the following lines.

```
1.  if g < 0
2.     then return ∞
3.  if A[i, j, k, l] < ∞
4.     if g < A[i, j, k, l]
5.        then return ∞
6.     else  return A[i, j, k, l]
```

At line 1-2, if g is below zero, it means that we have cut more rectangles than the greedy partitioning did. This never leads to an optimal partitioning, that is why we return ∞. At line 3-5, if $A[i, j, k, l]$ already is computed, and g is smaller than that value, it means that we still need more cuts than we are allowed to make, so we return ∞ again.

The following lines replace line 6-7 of the algorithm *LookupPartitioning* in the previous section.

```
1.  noResults ← false
2.  q1 ←LookupPartitioning(i, s, k, l, g − cuts(s))
3.  if q1 < ∞
4.     then q2 ←LookupPartitioning(s, j, k, l, g − cuts(s) − q1)
5.          if q2 < ∞
6.             then q ←q1 + q2 + cuts(s)
7.                  A[i, j, k, l] ← min(q, A[i, j, k, l])
8.          else  noResults ← true
9.  else  noResults ← true
```

At line 1 the boolean variable *noResults* gets initialized as false. The use of this variable is explained later. The number of cuts needed for $R_{i,s,k,l}$ is q_1. We calculate this value, and as parameter for g we give $g - cuts(s)$. Since we were allowed to cut g rectangles in $R_{i,j,k,l}$, and we cut at position s, which cuts $cuts(s)$ rectangles, we are allowed to only cut $g - cuts(s)$ rectangles in $R_{i,s,k,l}$.

Then we check if q_1 returns ∞. If it does, it means that it needs more cuts than are allowed and we can go to the next position for a partition line. If it does not, we continue and compute q_2. Since we already partitioned q_1, we are only allowed to cut $g - cuts(s) - q_1$ rectangles in $R_{s,j,k,l}$. Then we check if this does not return ∞. If it does not, we assign the value of this partitioning $(q_1 + q_2 + cuts(s))$ to q and check if it is better than the result until then. Notice that the cuts gets added to the value since we are counting the number of cuts now instead of all regions.

We also do this for the horizontal partition lines and change lines 11-12 from *LookupPartitioning* in a similar way.

It remains to explain the use of the boolean variable *noResults*. As can be seen at line 8 and 9 this variable becomes true if q_1 or q_2 is ∞. This is done for the following reason. If at the end nothing is filled in for $A[i, j, k, l]$ – so the value is ∞, line 1 of the lines below) – it is possible that all possible partition lines gave a result worse than the greedy algorithm. If that is the case, *noResults* has been put to true and in it should return ∞. So line 13 from the algorithm *LookupPartitioning* from the previous section gets replaced by the lines 1-3 that are presented below. Because we are counting the cuts instead of the regions, line 14 from the algorithm *LookupPartitioning* from the previous section (and line 4 in the lines below) should return 0 instead of 1.

```
1.  if A[i,j,k,l] = ∞
2.    then if noResults
3.           then return ∞
4.         else  A[i,j,k,l] ← 0
```

A way to improve the pruning heuristic is by updating the result from the greedy partitioning that we are using with a better result as soon as we find one. We do this by storing the value of the greedy partitioning at a variable, denoted by g_1. Then every time we look to a subdivision where $g = g_1$, which means that no rectangle has been cut yet, we check if $A[i, j, k, l]$ is filled. If it is and it is better than g, we update g and g_1 with the value from $A[i, j, k, l]$.

We can combine the pruning heuristic with the free splits heuristic. To do this, we have to make some adjustments to the free splits heuristic. When making a free split, we call one subproblem with g and check if q_1, the value it returns, is ∞. If it is, we return ∞. If it is not, we call the other subproblems with $g - q_1$ and check if q_2, the value it returns, is ∞. If it is we return ∞. If it is not, we return the value that we would normally return.

Hash tables. If we apply the previous two heuristics, then in many cases we do not have to test all subdivisions $R_{i,j,k,l}$. This should improve the computation time significantly. But we can also use this to reduce the storage space that is needed. We can use the hash table $T[0..p]$ instead of the table $A[0..n_x - 1, 1..n_x, 0..n_y - 1, 1..n_y]$. The hash function h maps a four-tuple (i, j, k, l) with $0 \leq i \leq n_x - 1$, $1 \leq j \leq n_x$, $0 \leq k \leq n_y - 1$ and $1 \leq l \leq n_y$ to an index in the range $0..p$. The value of $A[i, j, k, l]$ then is stored in $T[h(i, j, k, l)]$. Table B can be replaced by a hash table similarly.

5.4 Generalization

The algorithm for getting an optimal BSP for a rectangular subdivision described above can be generalized to an algorithm that can compute optimal BSPs for a wider class of cost functions.

Weights and costs. We make some changes to accomplish this. First we give each rectangle r a weight, denoted by $weight(r)$. With this weight we favor certain rectangles to be cut over other rectangles, which can be useful for some optimality criteria. Instead of counting the number of regions in the end, we count the cost of each node in the BSP tree. The cost of a node of a BSP tree is defined as follows:

- The cost of a leaf node (which corresponds to a region that need not be partitioned further) is the same as the weight of the rectangle where the corresponding region is in.
- The cost of an internal node (which corresponds to a region that can be partitioned further) is a function F of the costs of its children: $cost(v) = F(C_l, C_r)$, where C_l and C_r are the cost of the left and the right child of node v.

The cost of a BSP is defined as the cost of the root node of the BSP tree. To get an optimal BSP, its cost needs to be the minimum of all possible BSPs.

In order for the algorithm to work, the function F needs to be monotone in the following sense: if we go higher in the BSP tree, the costs cannot decrease. This means we have the following restriction on the function F.

- For any a, a', b, b' with $a \leq a'$ and $b \leq b'$ we have $F(a, b) \leq F(a', b)$ and $F(a, b) \leq F(a, b')$.

To get, just as in the previous sections, a BSP of minimum size, we set the weight of each rectangle to one, and we define $F(a, b) = a + b$. We can also model other optimality criteria. A cost function to obtain a BSP of minimum depth is $F(a, b) = \max(a, b) + 1$, where the cost of each rectangle is one again.

By changing the weights of the rectangles, we can further adjust the algorithm. If we give one rectangle a very large weight (while the others get weight one for example), we make sure that the rectangle with the large weight (almost) never gets cut. Another change is to give some rectangles weight zero. This means that it does not increase the cost of the BSP if we cut them, and cutting such a rectangle can then be seen as a free split. We can use this if we want to make a BSP of a set of rectangles – with all horizontal and vertical edges – that do not form a rectangular subdivision. First we put a big rectangle around the rectangles, then we give all rectangles weight one, and at last we fill all other space with rectangles which we give weight zero.

Algorithm. We are changing the algorithm given in the previous chapter so it works with our generalization, using the function F and weights for regions. First we define for integers i, j, k, l with $0 \leq i < j \leq n_x$ and $0 \leq k < l \leq n_y$:

$$C[i, j, k, l] = \text{minimum cost of a BSP for subdivision } R_{i,j,k,l}$$

Next we give the new recursive definition. Here r is the rectangle that contains region $R_{i,j,k,l}$ for the case $i = j - 1$ and $k = l - 1$.

$$C[i, j, k, l] = \begin{cases} weight(r) & \text{if } i = j - 1 \text{ and } k = l - 1 \\ \min \begin{cases} \min\limits_{i<s<j}(F(C[i, s, k, l], C[s, j, k, l])) \\ \min\limits_{k<t<l}(F(C[i, j, k, t], C[i, j, t, l])) \end{cases} & \text{if } i < j - 1 \text{ and } k < l - 1 \end{cases}$$

The algorithm itself does not change much, it still uses dynamic programming and has the same running time. *MemoizedPartitioning* stays the same, only it now uses $C[i, j, k, l]$ instead of $A[i, j, k, l]$). *LookupPartitioning* has some adjustments as shown below.

Algorithm *LookupPartitioning*(i, j, k, l)
1. **if** $C[i, j, k, l] < \infty$
2. **then return** $C[i, j, k, l]$
3. **for** $s \leftarrow i + 1$ **to** $j - 1$
4. **do for** $x \leftarrow 0$ **to** $length(V[s]) - 1$
5. **do if** ($V[s][x][0] < l$ and $V[s][x][1] > k$)
6. **then** $q \leftarrow F(LookupPartitioning(i, s, k, l),$ $LookupPartitioning(s, j, k, l))$
7. $C[i, j, k, l] \leftarrow \min(q, C[i, j, k, l])$
8. **for** $t \leftarrow k + 1$ **to** $l - 1$
9. **do for** $y \leftarrow 0$ **to** $length(H[t]) - 1$
10. **do if** ($H[t][y][0] < j$ and $H[t][y][1] > i$)
11. **then** $q \leftarrow F(LookupPartitioning(i, j, k, t),$ $LookupPartitioning(i, j, t, l))$
12. $C[i, j, k, l] \leftarrow \min(q, C[i, j, k, l])$
13. **if** $C[i, j, k, l] = \infty$
14. **then** $C[i, j, k, l] \leftarrow weight(r)$
15. **return** $C[i, j, k, l]$

Heuristics. With our generalized algorithm, free splits cannot always be made. It depends on the cost function that is used. We can for example still use it to find a BSP of minimum size, but we cannot use it if we want to find a BSP with minimum depth. Further more, if we give regions weight zero, those regions can be cut without increasing the cost, so they should also be treated as free splits to improve the running time. Pruning can still be done. It is always useful to stop the current partitioning if it will not lead to a better result than an already found partitioning. It can be applied in the same way, but the cost of a partition line is now dependent on the weight of the rectangles that it cuts and the cost function. Hash tables can still be used in the same way as described in the previous chapter.

5.5 Application to rectilinear cartograms

An example of an application where this algorithm is used is to compute high-quality rectilinear cartograms [11].

A cartogram is a thematic map that visualizes statistical data of some set of regions. The area of such a region corresponds to a particular geographic variable. In Figure 24 we can for instance see a map of Europe (the regions are countries), where the area of each region is proportional to the population of the corresponding country (the population is the geographic variable).

A rectilinear cartogram is a cartogram where each region is a rectangle, L-shape or any other type of rectilinear polygon. It has been proven [10] that it is always possible to construct rectilinear cartograms with zero cartographic error (so each region has exactly the size the variable gives) and correct adjacencies (so they still have the same neighbours). The algorithm of [10] needs a subroutine to compute a BSP for a rectangular subdivision. To get the best result, an optimal BSP is used that uses a specialized cost function. Since the cost function is monotone, the algorithm described above can be used. See the paper by De Berg *et al.* [11] for details.

Figure 24: Cartogram of the population of Europe [11]

6 Implementation and experiments

We implemented the algorithm described in the previous chapter to get an optimal BSP for a rectangular subdivision. To be able to experiment with this algorithm, random rectangular subdivisions were generated, as described in the next section.

6.1 Generation of random rectangular subdivisions

To generate a random rectangular subdivision, we first create a rectangle with random vertical line segments in it. We set a width w and a height h for the normalized rectangular subdivision. For each vertical rank a random number of line segments of random length and at random places are created. Each endpoint of those line segments has to be on a horizontal rank. We make sure that no line segments overlap and that they all fit.

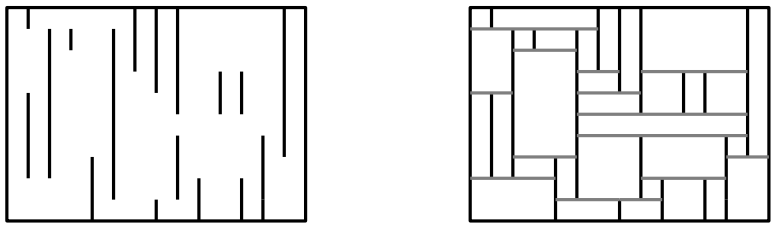
We use the algorithm *GenerateLineSegments* to generate the vertical line segments (x_1, y_1, x_2, y_2) and add them to a list $verSegments$ with all the vertical line segments.

Algorithm *GenerateLineSegments*(w, h)
1. **for** $i \leftarrow 1$ **to** $w - 1$
2. **do** $nrSeg \leftarrow$ **random**$(1, \lceil h/8 \rceil)$
3. $start \leftarrow 0$
4. **for** $j \leftarrow 1$ **to** $nrSeg$
5. **do** $nrSegLeft \leftarrow nrSeg - j$
6. $y_1 \leftarrow start +$ **random**$(0, h - (nrSegLeft * 7) - last)$
7. $y_2 \leftarrow y_1 + 1 +$ **random**$(0, h - (nrSegLeft * 7) - y_1)$
8. Add (i, y_1, i, y_2) to the list $verSegments$.
9. $start \leftarrow y_2 + 1$

First we have a loop that goes through all the vertical ranks (line 1). At each rank we randomly choose a number of segments that is generated (line 2). With *start* we denote the y-coordinate of the first free position where we can place the next segment. We initialize it at zero (line 3). Then we have another loop that goes through all the segments for a certain rank (line 4). The number of segments that we still have to generate on a certain rank are denotes by $nrSegLeft$ (line 5). Then we randomly choose the first (y_1) and second (y_2) y-coordinate of the segment (line 6-7) in such a way that there is still enough room left for the other segments that we still have to generate for

that rank. At last we add the segment to our segment list *verSegment* (line 8) and update *start* with the first free position for the next segment.

The number of segments that we choose for a rank and the length and position of each segment were determined experimentally.



(a) Random vertical line segments (b) Random rectangular subdivision

Figure 25: A rectangular subdivision (b) is created from some random vertical line segments (a).

The rectangular subdivision is obtained by a horizontal decomposition on the line segments in *verSegments*. This can be done with a plane sweep in $O(n \log n)$ time, as shown in [12]. In Figure 25(b) a random rectangular subdivision created from the vertical line segments from Figure 25(a) is given. Figure 26 shows a rectangular subdivision of 118 rectangles which has been created by the described algorithm.

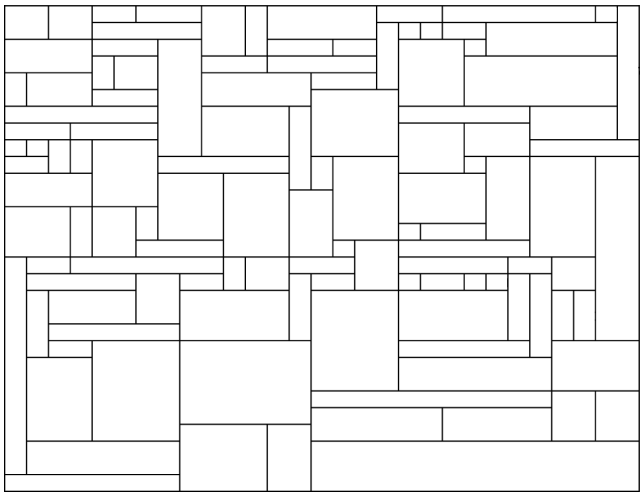


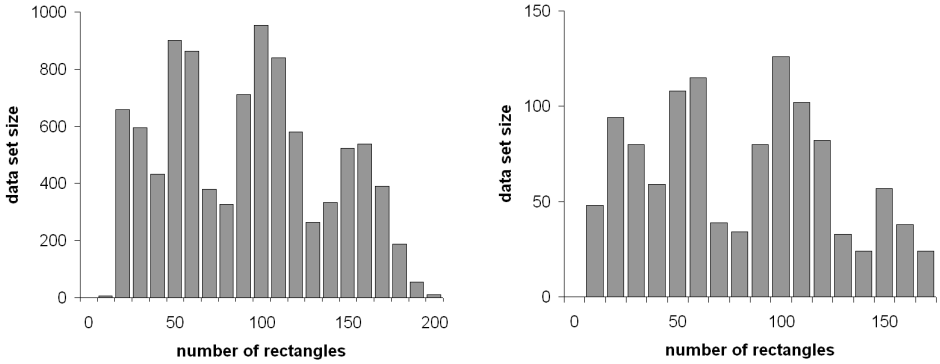
Figure 26: A randomly generated rectangular subdivision

6.2 Experiments

In this section we give our experimental results and explain and comment on some of the results. First we give the specification of how and on which platform we did the experiments.

Specifications. The experiments have been done on an *AMD Athlon XP 1700+ (1.11 GHz)* computer with *256 MB of RAM* that runs the operating system *Microsoft Windows XP Professional*. The programming has been done in *Java* using the IDE *JCreator*. The memory and time usage measurement are done by using native Java functions.

In all our experiments we used the free-split heuristic because this can only influence the results in a positive way. We have done experiments with and without the pruning and hash table heuristics. We used 2 randomly generated data sets. One – which we call data set 1 – of about 9500 rectangular subdivisions with sizes between 5 and 204 rectangles. And another smaller data set – which we call data set 2 – of about 1150 rectangular subdivisions with sizes between 5 and 174 rectangles. In order to get smoother charts, we grouped together the data in groups of ten. Figure 27 shows the number of rectangular subdivisions within each group, so for example in data set 1, there are about 700 rectangular subdivisions that have a number of rectangles varying between 85 and 94. The results given later are averaged for each of these groups of ten (5-14, 15-24, 25-34, etc.)



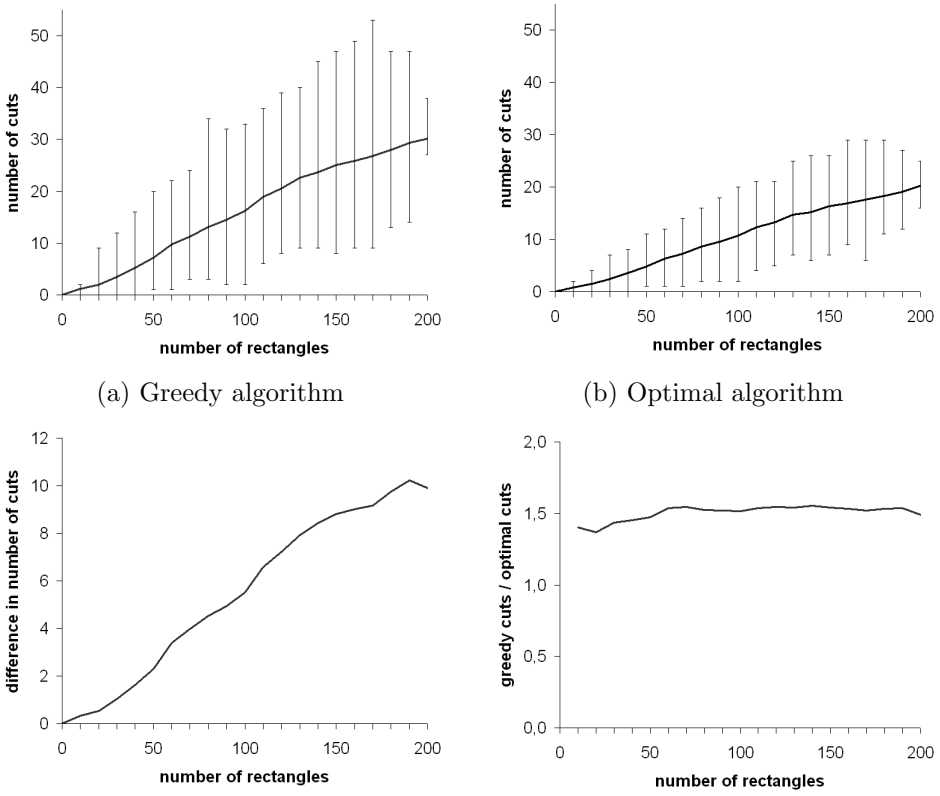
(a) Data set 1 (b) Data set 2

Figure 27: Distribution of the data sets

Data set 2 is smaller because the algorithm using the pruning heuristic takes a lot of time (as shown later) when the data set gets larger. That is also why

data set 2 contains rectangular subdivisions with at most 174 rectangles. We only used data set 2 when we did experiments with the pruning heuristic. For all the other experiments we used data set 1.

Greedy and optimal algorithm. We look at the difference in the number of cuts between the greedy and the optimal algorithm to partition rectangular subdivisions. Figure 28(a) shows the average number of cuts that the greedy algorithm needs to partition a rectangular subdivision with a certain number of rectangles. The thin vertical line segments describe the range of the cuts (so the top is the maximum and the bottom is the minimum number of cuts). For example a rectangular subdivision with a number of rectangles between 95 and 104 needs an average of 16 cuts with a minimum of 2 and a maximum of 33.



(a) Greedy algorithm

(b) Optimal algorithm

(c) Difference in number of cuts

(d) Relative difference in number of cuts

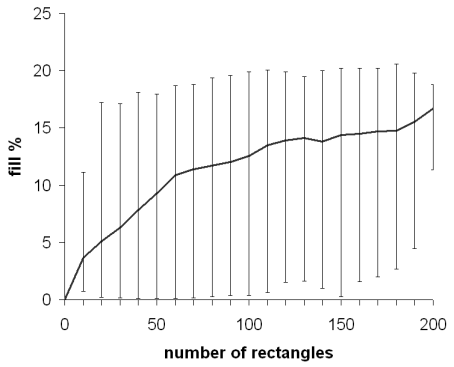
Figure 28: The number of cuts that the greedy an the optimal algorithms do and their difference.

The results for the optimal algorithm are shown in Figure 28(b). To get a better view of the difference between the two figures, we show the difference in Figure 28(c). It shows that the difference keeps increasing approximately linear. Figure 28(d) shows the relative difference in the number of cuts by dividing the number of cuts of the greedy algorithm by the number of cuts of the optimal algorithm. On average the greedy algorithm does 1.5 times as much cuts as the optimal algorithm.

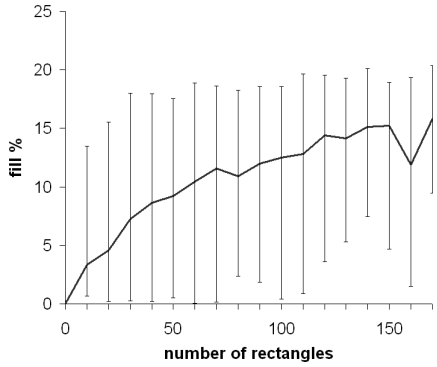
Note that the thin vertical line segments are smaller at the end of our graphs. This is explained by the fact that there were less experiments within the largest group of rectangular subdivisions as can be seen in Figure 27(a).

Fill percentage. The fill percentage is the percentage of table entries filled by the dynamic programming in the optimal algorithm. Since we use the free-split heuristic in all our experiments, a lot of subproblems are not computed, therefore the table never is full. Even without the free-split heuristic, more than half of the table is not filled, since $i < j$ and $k < l$. Figure 29(a) shows the fill percentage against the number of rectangles in the rectangular subdivision. For small subdivisions the fill percentage is lower, because more free splits can be done. The average fill percentage is approximately logarithmic and goes toward 15. In the end the line goes up a little. This can be explained as noise, since fewer experiments are done with a high rectangle number. Notice that the maximum fill percentage does not get much higher than 20%. The fewer experiments at a higher rectangle number also explain why the minimum is higher there. It is always possible to have a rectangular subdivision which you can partition by only making free splits which will lead to a low fill percentage.

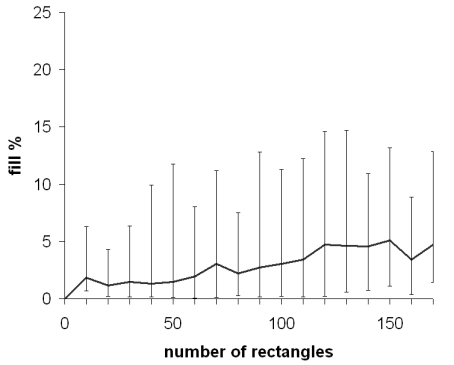
We are also interested in how our pruning heuristic effects the fill percentage. We used data set 2 for this, like stated earlier. Figure 29(b) shows the results of the algorithm without pruning for data set 2. It gives about the same results as data set 1 – Figure 29(a). It only has more noise, because we did less experiments. Figure 29(c) shows the results of the algorithm with pruning for data set 2. Figure 29(d) shows the difference in fill percentage between the algorithm with and without pruning. This is around 10% for larger subdivisions, which means that the algorithm without pruning fills the table about three times as much as the algorithm with pruning. Also the maximum fill percentage is lower, around 15%. A lower fill percentage leads to less memory usage if we use hash tables. We see this in the next paragraph.



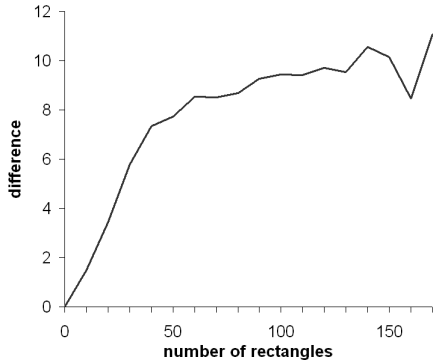
(a) Fill % without pruning (data set 1)



(b) Fill % without pruning (data set 2)



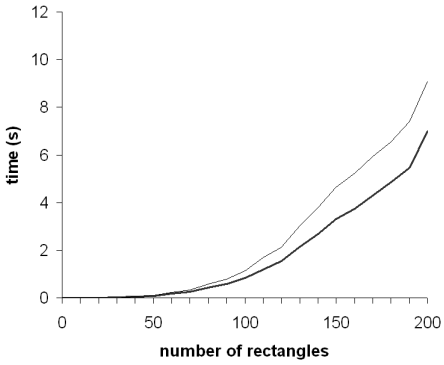
(c) Fill % with pruning (data set 2)



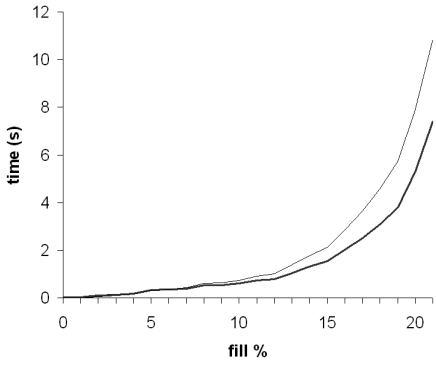
(d) Difference between the algorithms with and without pruning (data set 2)

Figure 29: The different fill percentages for the algorithm with and without pruning.

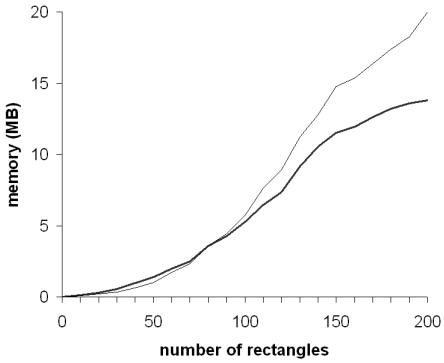
Time and storage for arrays and hash tables. We present our results for the time and storage space of the algorithms. We compare the use of arrays versus hash tables – we denote the algorithm that uses arrays by *ArrayAlg* and the algorithm that uses hash tables by $HashAlg$. First we look at the running time of our algorithms. Figure 30(a) shows the running time in seconds for *ArrayAlg* (the fat line) and for $HashAlg$ (the thin line). We see that $HashAlg$ has the longest running time, a little more than *ArrayAlg*. Figure 30(b) shows the time versus the fill percentage. *ArrayAlg* and $HashAlg$ have about the same running time if the fill percentage is lower than 7%. For higher fill percentages *ArrayAlg* has a better running time.



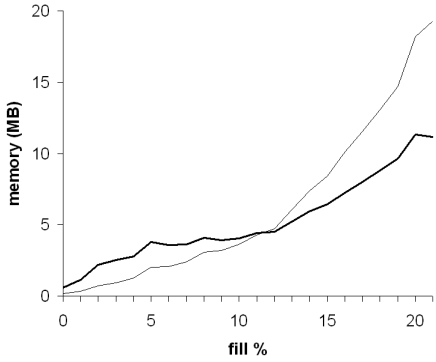
(a) Time versus number of rectangles



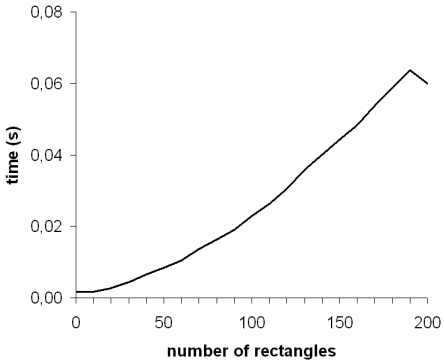
(b) Time versus fill percentage



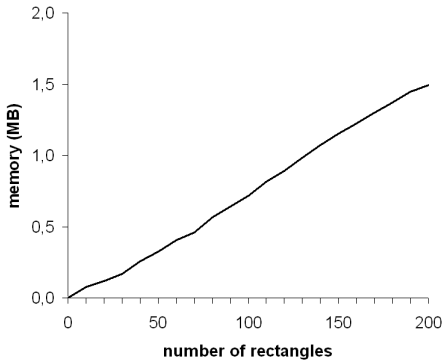
(c) Memory versus number of rectangles



(d) Memory versus fill percentage



(e) Greedy algorithm – time



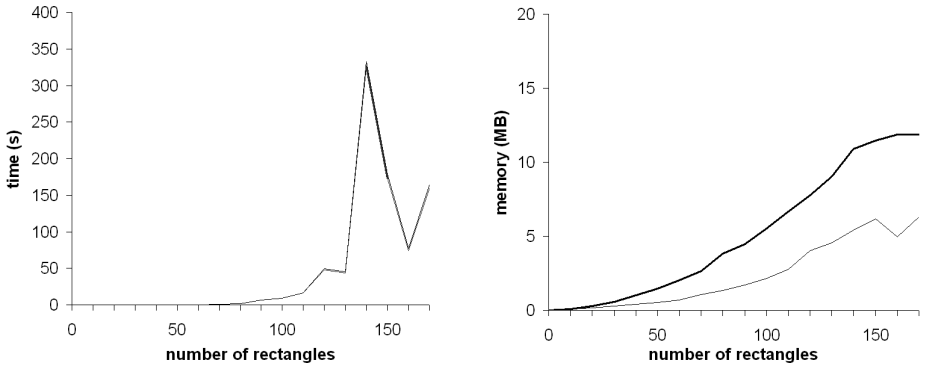
(f) Greedy algorithm – memory

Figure 30: Time and storage space measurements for the algorithms without pruning. In (a), (b), (c) and (d), the fat line is *ArrayAlg* and and the thin line is *HashAlg*.

Figures 30(c) and (d) show the results for the storage space in megabyte. $HashAlg$ uses less storage space for a low number of rectangles. This is explained by the fact that subdivisions with a low number of rectangles also have a lower fill percentage. In Figure 30(d), where we put the fill percentage against the memory usage, this gets more clear. For a low storage space, we better use $HashAlg$ when the fill percentage is under 12% and $ArrayAlg$ otherwise.

Figures 30(e) and (f) show the results of the time and storage space measurements for the greedy algorithm. It runs a lot faster and needs a lot less memory than the optimal algorithm.

Pruning heuristic. We now look at the results for time and storage space for the algorithms that use pruning. Figure 31(a) shows the time versus the number of rectangles for our algorithms with pruning. There is hardly any difference between $ArrayAlg$ and $HashAlg$ (they are on top of each other). They work well until about 100 rectangles. After that the time they need increases a lot. The big peak at 140 rectangles can be explained by the fact that the computation of the optimal BSP for one of the subdivisions needed one hour and 20 minutes. We notice that the pruning heuristic works bad for large rectangular subdivisions. This is explained by the fact that the pruning heuristic does not finish a subdivision which it is calculating if it gets worse than the best result until then. Because it is not finished, the whole subdivision is calculated again if it is a subproblem of another subdivision. So the pruning heuristic can partially undo one of the advantages of dynamic programming.



(a) Time versus number of rectangles (b) Memory versus number of rectangles

Figure 31: Time and storage space for the algorithms with pruning.

Figure 31(b) shows the memory usage of the algorithms with pruning. *ArrayAlg* is the fat line and *HashAlg* the thin line. *HashAlg* with pruning works great for storage space. At a large number of rectangles, it uses half of the storage space that the *ArrayAlg* uses. Pruning for *ArrayAlg* has no use with respect to storage, since it needs storage space for the whole table anyway. Figures 31(b) and 30(c) shows that the storage space for *ArrayAlg* is about the same with and without pruning.

Influence of zero-weighted recangles. In the previous chapter an example of a situation that uses rectangles with a weight of zero was given. It means that cutting such a rectangle is the same as a free split. This way a lot more free splits can be made which should lead to a lower fill percentage. We investigated the effect of zero-weighted rectangles. Figure 32 shows the results. The fill percentage decreases a lot when there are more zero-weighted rectangles. Notice that the fill percentage is only about 11.5 % when we have no zero-weighted rectangles. This is because it is the average fill percentage over all rectangular subdivisions in data set 1. Data set 1 contains a lot of small rectangular subdivisions which have a low fill percentage.

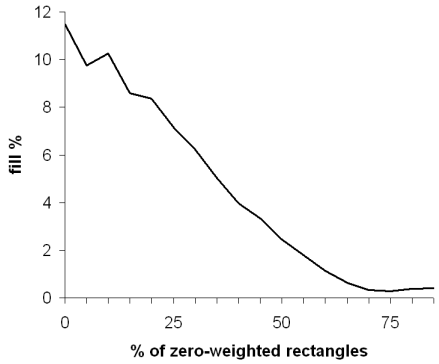


Figure 32: The influence of zero-weighted rectangles on the fill percentage.

7 Conclusion

We have proven that we can approximate an optimal free BSP for a set of arbitrary disjoint line segments in the plane with a restricted BSP which cuts no more than three times as many line segments as the free optimal BSP.

We presented an algorithm that computes an optimal BSP for a rectangular subdivision with n rectangles in $O(n^5)$ time. We showed some heuristics for this algorithm and did experiments to test the practical use of these heuristics. The free splits heuristic speeds things up a lot. Our experimental results demonstrate that the pruning heuristic can slow down the algorithm. It decreases the memory usage if we use pruning in combination with hash tables. So if memory usage is much more important than time, using the pruning heuristic can be useful. Maybe it is possible to do the pruning in a different way, which is an interesting topic for future work. The experimental results also showed that the use of hash tables instead of arrays is most useful if the fill percentage of the dynamic programming table is low, which is for example the case with the rectilinear cartogram application. This application uses rectangular subdivisions with a lot of zero-weighted rectangles which keep the fill percentage low as we saw from our results. We also notice that the given algorithms are not useful for huge data sets, because they need too much time and memory.

Future work. For future work we suggest making an algorithm that creates an optimal restricted BSP for a set of arbitrary disjoint line segments in the plane. It looks like this can be done with dynamic programming, but the problem is that there are an exponential number of subproblems. A second suggestion for future work is to come up with more and better heuristics for computing an optimal BSP for a rectangular subdivisions.

References

[1] P.K. Agarwal, T.M. Murali, and J.S. Vitter. Practical techniques for constructing binary space partitions for orthogonal rectangles. In *Proc. 13th Annual Symposium on Computational Geometry*, pages 382–384, 1997.

[2] S. Ar, B. Chazelle, and A. Tal. Self-customized BSP trees for collision detection. *Computational Geometry: Theory and Applications*, 15(1-3):91–102, 2000.

[3] C. Ballieux. Motion planning using binary space partitions. *Technical Report Inf/src/93-25*, Utrecht University, 1993.

[4] J. Baltes and J. Anderson. Flexible binary space partitioning for robotic rescue. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3144–3149, 2003.

[5] N. Chin and S. Feiner. Near real-time shadow generation using BSP trees. In *Proc. 16th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 99–106, 1989.

[6] F. d'Amore and P.G. Franciosa. On the optimal binary plane partition for sets of isothetic rectangles. *Information Processing Letters*, 44(5):255–259, 1992.

[7] M. de Berg. Linear size binary space partitions for uncluttered scenes. *Algorithmica*, 28(3):353–366, 2000.

[8] M. de Berg, M. de Groot, and M. Overmars. New results on binary space partitions in the plane. In *Proc. 4th Scandinavian Workshop on Algorithm Theory*, volume 824 of *Lecture Notes in Computer Science*, pages 61–72. Springer-Verlag, 1994.

[9] M. de Berg, M. de Groot, and M. Overmars. Perfect binary space partitions. *Computational Geometry: Theory and Applications*, 7(1-2):81–91, 1997.

[10] M. de Berg, E. Mumford, and B. Speckmann. On rectilinear duals for vertex-weighted plane graphs. In *Proc. 13th International Symposium on Graph Drawing*, volume 3843 of *Lecture Notes in Computer Science*, pages 61–72, 2005.

[11] M. de Berg, E. Mumford, and B. Speckmann. Optimal BSPs and rectilinear cartograms. In *Proc. 14th International Symposium on Advances in Geographic Information Systems*, 2006. (to appear).

[12] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2nd edition, 2000.

[13] A. Dumitrescu, J.S.G. Mitchell, and M. Sharir. Binary space partitions for axis-parallel segments, rectangles, and hyperrectangles. In *Proc. 17th Annual Symposium on Computational Geometry*, pages 141–150, 2001.

[14] H. Fuchs, Z.M. Kedem, and B.F. Naylor. On visible surface generation by a priori tree structures. In *Proc. 7th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 124–133, 1980.

[15] R. Krishnaswamy, G.S. Alijani, and S. Su. On constructing binary space partitioning trees. In *Proc. Annual Conference on Cooperation*, pages 230–235, 1990.

[16] M.S. Paterson and F.F. Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete & Computational Geometry*, 5(5):485–503, 1990.

[17] M.S. Paterson and F.F. Yao. Optimal binary space partitions for orthogonal objects. *Journal of Algorithms*, 13(1):99–113, 1992.

[18] M. Slater. A comparison of three shadow volume algorithms. *The Visual Computer: International Journal of Computer Graphics*, 9(1):25–38, 1992.

[19] P. Tobola and K. Nechvile. Linear BSP tree in the plane for set of segments with low directional density. In *Proc. 7th Annual Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, pages 297–304, 1999.

[20] C.D. Tóth. A note on binary plane partitions. In *Proc. 17th Annual Sumposium on Computational Geometry*, pages 151–156, 2001.

[21] C.D. Tóth. Binary space partitions for line segments with a limited number of directions. *Proc. 13th Annual Symposium on Discrete Algorithms*, 32:307–325, 2003.

[22] P. van Oosterom. A modified binary space partition for geographic information systems. *International Journal of Geographical Information Systems*, 4(2):133–146, 1990.