Perfect Binary Space Partitions *

Mark de Berg Marko M. de Groot Mark H. Overmars

Abstract

In this paper we discuss some results on perfect binary space partitions on sets of non-intersecting line segments in two dimensions. A binary space partition is a scheme for recursively dividing a configuration of objects by hyperplanes until all objects are separated. A binary space partition is called perfect when none of the objects is cut by the hyperplanes used by the binary space partition. Given a set of n non-intersecting line segments, our method constructs a perfect binary space partition, or decides that no perfect binary space partition exists for the arrangement of line segments, in $O(n^2 \log n)$ time.

1 Introduction

For geometric problems where the input is a set of objects in the plane or space, efficient algorithms are often based on recursive partitioning. The input is divided into two parts by splitting the set of objects with a line, in the 2-dimensional space, or with a plane in 3-dimensional space. The two resulting sets are then divided recursively until finally all objects are separated. The scheme as described above is called a binary space partition and was first considered in [2]. Since each division may split some of the objects into two parts, the process described above can lead to a proliferation of objects. Therefore we are motivated to search for schemes that cut the set of objects in such a way that fragmentation of these objects is minimized.

The problem of constructing binary space partitions of small size has been studied by Paterson and Yao [7]. For the 2-dimensional case they construct binary space partitions, BSPs in short, of size $O(n \log n)$ for sets of n line segments, using a $O(n \log n)$ time algorithm. They also present a special kind of BSP based on *auto-partitions*: a natural class of BSPs, that can be obtained by imposing the restriction that each cut hyperplane must be a hyperplane containing some facet of the input set. For any n disjoint line segments in the plane, an autopartition of size $O(n \log n)$ can be found in $O(n^2)$ time.

*Department of Computer Science,Utrecht University,P.O. Box 80.089,3508 TB Utrecht,The Netherlands. Research was supported by the ESPRIT Basic Research Actions of the EC under contract No. 7141 (project ALCOM II: *Algorithms and Complexity*). The first and third author were also supported by the Dutch Organization for Scientific Research (NWO).

For n facets in $\mathcal{R}^3$ they show that an autopartition of size $O(n^2)$ can be constructed in $O(n^3)$ time, in $\mathcal{R}^d$ autopartitions of $O(n^{d-1})$ size can de constructed in time $O(n^{d+1})$. Methods concerning *orthogonal* objects are presented by Paterson and Yao in [8]. For any set of n orthogonal non-intersecting line segments in the plane they construct an autopartition of $\Theta(n)$ size in time $O(n \log n)$. They also show that for any configuration of n axis-parallel line segments in $\mathcal{R}^3$, a BSP of size $\Theta(n^{3/2})$ can be found in time $O(n^{3/2})$. For a set of n axis-parallel rectangles, they achieve the same time and size bounds. Finally, they present bounds on the size of a BSP on sets of axis-parallel line segments in four or more dimensions.

In this paper however we restrict ourselves to *perfect* binary space partitions in the plane. A perfect binary space partition is defined to be a binary space partition that prevents any fragmentation of the objects in the input set. Finding a perfect binary space partition for a set of n line segments will be proved to be n^2*-hard*, a class of problems defined in [3], showing that it is probably hard to find a subquadratic solution. We also present an algorithm that constructs a perfect BSP in $O(n^2 \log n)$ time or that reports its non-existence.

2 Planar perfect binary space partition

Perfect binary space partitions, perfect BSPs in short, form a class of BSPs that can be obtained by imposing the restriction that each cut hyperplane be a hyperplane that does not intersect or contain any of the objects in the input configuration. The formal definition of a perfect BSP tree is as follows:

Definition 2.1 *A perfect binary space partition (perfect BSP) for a set S of objects in d-dimensional space is a binary tree $\mathcal{T}$ with the following properties:*

- *If $|S| \leq 1$ then $\mathcal{T}$ is a leaf; the set S is stored explicitly at this leaf.*
- *If $|S| > 1$ then the root ν of $\mathcal{T}$ stores a hyperplane h_ν, such that the set of objects that are contained in or intersected by h_ν is empty. The left child of ν is the root of a BSP tree $\mathcal{T}^+$ for the set $h_\nu^+ \cap S = \{s \in S : s \subset h_\nu^+\}$, where h_ν^+ is the region above h_ν, and $h_\nu^+ \cap S$ is non-empty. The right child of ν is the root of a BSP tree $\mathcal{T}^-$*

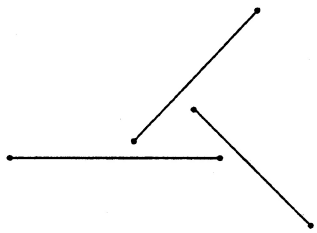


Figure 1: no perfect BSP

for the set $h_\nu^- \cap S$, where h_ν^- is the region below h_ν, and $h_\nu^- \cap S$ is non-empty.

Observe that it is not always possible to find a perfect BSP given a configuration of objects, since configurations can be constructed that do not permit a perfect BSP, see Figure 1.

In this paper we study perfect BSPs for sets of non-intersecting line segments in the plane.

2.1 The lower bound

There are many problems in computational geometry for which the best known algorithms take at least $\Theta(n^2)$ time in the worst case while only very low lower bounds are known. In [3] Gajentaan and Overmars describe a large class of so-called n^2-*hard* problems which they prove to be at least as difficult as the following base problem: Given a set S of n integers, determine the existence of three elements of S that sum up to zero. The best known algorithm for this base problem takes $\Theta(n^2)$ time. They prove that the class of n^2-hard problems includes the following problem, which they call GeomBase: Given a set of n points with integer coordinates on three horizontal lines $y = 0, y = 1$ and $y = 2$, determine whether there exists a non-horizontal line containing three of the points. We prove that this class of n^2-hard problems includes our problem of finding a perfect binary space partition by showing our problem is as least as hard as GeomBase. This means that it will be difficult to obtain an $o(n^2)$ running time.

Theorem 2.2 *It is n^2-hard to decide whether a set of n disjoint line segments in the plane admits a perfect binary space partition.*

Proof: We show that the problem of finding a perfect BSP is as least as hard as the problem GeomBase. Given a set of points on three horizontal lines $y = 0, y = 1$ and $y = 2$. Let the x-coordinates of the points on the first line A, ordered from left to right be $a_1, a_2, \ldots, a_i$. Similarly, let the points on the other lines B and C be $b_1, b_2, \ldots, b_j$ and $c_1, c_2, \ldots, c_k$. Let $\epsilon = \frac{1}{4}$. Now transform the points

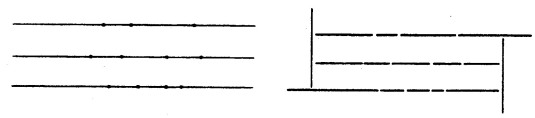


Figure 2: finding a perfect BSP is n^2-hard

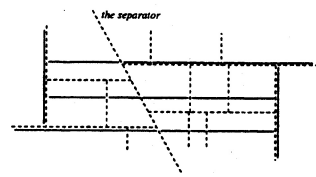


Figure 3: a perfect BSP

on A into horizontal line segments on A with x-intervals $[a_1 + \epsilon : a_2 - \epsilon], \ldots, [a_{i-1} + \epsilon : a_i - \epsilon]$. And add one very long line segment extending from $a_1 - \epsilon$ to the left and one very long line segment at $a_i + \epsilon$ extending to the right, see Figure 2. Similar for the sets B and C. Finally we place two vertical line segments at the left and right of our arrangement. Clearly this transformation can be done in time $O(n \log n)$. In [3] the same construction is used to prove the n^2-hardness of a separator problem.

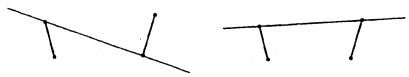
It is obvious that, when there is a line through the points a on A, b on B and c on C, a separator exists for the set of line segments that goes through the holes related to a, b and c. Once given such a seperator, a perfect BSP can easily be constructed, see Figure 3 where the dotted lines indicate the perfect BSP.

It remains to prove the reverse. If a perfect BSP exists of the set of line segments constructed, then the first partition line must be a separator of the set of line segments. From the construction it is clear that this partition line must run through three holes $\langle a - \epsilon : a + \epsilon \rangle$ on A, $\langle b - \epsilon : b + \epsilon \rangle$ on B and $\langle c - \epsilon : c + \epsilon \rangle$ on C. So $(a + \delta_1) + (b + \delta_2) = 2(c + \delta_3)$ for δ_1, δ_2 and δ_3 between $-\epsilon$ and ϵ. Because $\epsilon = \frac{1}{4}$ and a, b and c are integers, this is only possible when $a + b = 2c$, thus when there is a line through points on A, B and C. □

Hence, it is probably hard to find a subquadratic solution of the construction of a perfect BSP of a set of non-intersecting line segments in the plane.

2.2 The algorithm

In our algorithm to compute perfect BSPs, visibility graphs play a crucial role. The visibility graph G_S of a set of non-



4.i: inner tangent 4.ii: outer tangent

Figure 4:

5.i: line 5.ii: segment 5.iii: halfline

Figure 5: line segment extension

intersecting line segments S is a graph structure, where the vertices of the graph equal the endpoints of our input line segments and where two vertices are adjacent if the interior of the segment connecting the two corresponding endpoints does not intersect any of our input line segments except in their endpoints. The usage of the visibility graph G_S of the set of line segments S is built on the following observation: each partition line of the set S of line segments that does not intersect any of these line segments and that partitions the set in two non-empty subsets, can be slightly rotated until it touches two line segments each at a side. The line segment defined by the two points, in which the rotated partition line touches the line segments, is an edge of G_S. We can formalize this observation as follows:

Observation 2.3 *If a line exists that partitions the set of line segments S in exactly two non-empty subsets without intersecting any of the line segments of S, then there also exists such a line that contains an edge of G_S.*

We will not view this visibility graph G_S as a graph structure but as a geometrical structure, i.e. when we speak of an edge e of G_S, we mean the line segment connecting the corresponding endpoints. Partition lines of our set S can be found by extending each edge e of G_S, where the extension ℓ_e of an edge e is the line containing e. Notice we only have to extend those edges of G_S that are inner tangent edges of G_S, because the outer tangent edges correspond to partition lines that do not split the set of input line segments in two non-empty subsets. See Figure 4.

Let $\tilde{G}_S$ be that subgraph of G_S from which all outer tangent edges are removed. We extend each edge e of $\tilde{G}_S$ to a line ℓ_e, with the restriction that ℓ_e may not intersect any of the line segments of S, see Figure 5. So if it is not possible to extend an edge to a line, we extend it to a line segment that is as long as the configuration of the set S of line segments permits, see Figure 5.ii. A half-line will be the result of an extension that is restricted at just one side, see Figure 5.iii. We denote the resulting set of extensions by E. We define two types of extensions. One type denotes the edges of which the extension is a line, those we will call partition lines. We denote the set of all partition lines by E_p. The other type denotes the edges of which the extension is a half-line or a line segment, those we will call candidate partition lines. The set of all candidate partition lines is denoted E_c. For our method it is necessary that also these candidate partition lines are registered, because from this set the partition lines for future subspaces can be chosen.

Each stage of our recursive algorithm is initiated by the choice of a partition line ℓ from the set of available partition lines E_p. Let our initial space be denoted by R, the half space above ℓ be called R_a, and the halfspace below ℓ be called R_b. The following two actions have to be taken:

1. the set of line segments S has to be subdivided against ℓ, resulting in two subsets: S_a, the set of all line segments above ℓ, and S_b, the set of line segments below ℓ.
2. we have to update and split our sets of partition lines E_p and our set of candidate partition lines E_c, creating a set of partition lines and candidate partition lines for each of the subspaces R_a and R_b.

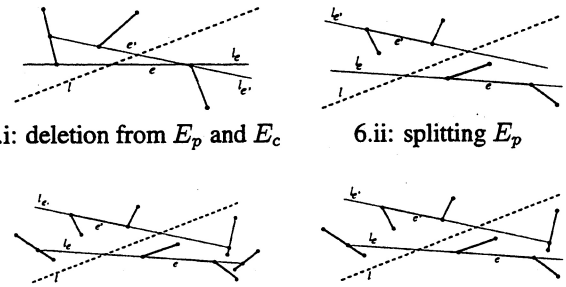
Let the set of partition lines E_p be split in two disjoint subsets $E_{p,a}$ for R_a and $E_{p,b}$ for R_b. Let the set of candidate partition lines E_c be split likewise in two disjoint subsets $E_{c,a}$ for R_a and $E_{c,b}$ for R_b. When we elaborate the second action above, the following steps have to be taken, Figure 6:

delete step From E_p and E_c all partition lines ℓ_e have to be removed, for which e is cut by the partition line ℓ, see Figure 6.i.

split step E_p is to be split in two subsets $E_{p,b}$ and $E_{p,a}$, where the partition line $\ell_e \in E_p$ is a member of $E_{p,b}$ if e lies below ℓ, see Figure 6.ii. Otherwise ℓ_e is a member of $E_{p,a}$. Similarly, E_c is to be split in two subsets $E_{c,b}$ and $E_{c,a}$, where the candidate partition line $\ell_e \in E_c$ is a member of $E_{c,b}$ if e lies below ℓ, see Figure 6.iii. Otherwise ℓ_e is a member of $E_{c,a}$.

update step $E_{c,b}$ and $E_{p,b}$ are to be updated, because some ℓ_e, that were candidate partition lines for R, can be partition lines for R_b and therefore have to be removed from $E_{c,b}$ and added to $E_{p,b}$, see Figure 6.iv. Likewise, $E_{c,a}$ and $E_{p,a}$ are to be updated, because some ℓ_e, that were candidate partition lines for R, can be partition lines for R_a and therefore have to be removed from $E_{c,a}$ and added to $E_{p,a}$.

After these steps we have two half-spaces with the line segments, partition lines and candidate partition lines inside



6.i: deletion from E_p and E_c 6.ii: splitting E_p

6.iii: splitting E_c 6.iv: updating

Figure 6:

them, on which we recurse. Recursion is continued until one of two situations occurs:

- The half space under consideration contains only one single line segment of the set S. No further partitioning of this subspace is necessary.
- The half-space under consideration contains more than one line segment from S, but the set of partition lines corresponding to this subspace is empty. In this case the total partition can be ended, with the conclusion that there exists no perfect binary space partition for this particular arrangement of line segments.

Using this strategy, we have to be sure that no partition line is overlooked. Our previously mentioned Observation 2.3 garantees this. But what if at a certain stage in the recursion more than one partition line is available? Can we just pick one of the partition lines available, or do we have to take some order or priority into consideration? The next lemma implies that this is not necessary.

Lemma 2.4 *If a perfect binary space partition exists for the set of line segments S, then a perfect binary space partition exists for any subset of S.*

Proof: Let S' be any subset of S. Let $\mathcal{T}_S$ be a binary space partition of S. Given $\mathcal{T}_S$, we can easily construct a binary space partition $\mathcal{T}_{S'}$ of S' as follows. We leave out all leaves of $\mathcal{T}_S$ that contain a segment not in S'. Next we visit each internal node ν of $\mathcal{T}_S$ in a bottom up fashion, checking the presence of both its children: when ν has no children we remove ν from $\mathcal{T}_S$, and in case ν has one child we replace ν by this child. Otherwise ν contains a partition line splitting S' in two non-empty subsets, and is an internal node of $\mathcal{T}_{S'}$. □

Thus, within our method we can define two parts. The first part finds the initial partition lines E_p and candidate partition

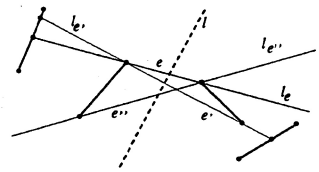


Figure 7: delete e from E_p resp. E_c

lines E_c, by construction of the visibility graph G_S on the set S of input line segments and initiates the partition. The second part recursively applies the partition strategy described above. Next we show how to implement this second part efficiently. The method presented here is built on the concept of *tandem search*, and is deduced from a method for computing depth orders as described in [1]. Tandem search is a scheme to partition a set of objects into two subsets in time that is dependent on the size of the *smaller* of the two subsets. This means that the more unbalanced the partitioning is, the faster it is performed, leading to a good worst-case running time for the algorithm.

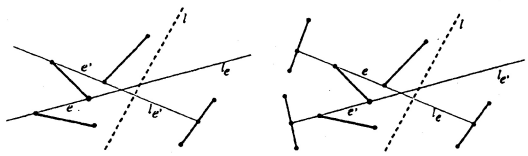
Let P_S denote the set of the endpoints of all line segments of S, P_E be the set of endpoints of all the extensions in E, and ℓ the partition line chosen. To split our sets P_S and P_E with ℓ, we build the convex hulls $\mathcal{CH}(P_S)$ and $\mathcal{CH}(P_E)$. Let $P_{S,a}$ be those points of P_S above ℓ and $P_{S,b}$ the points of P_S below ℓ. Let $P_{E,a}$ and $P_{E,b}$ be defined likewise for P_E. The points of $\mathcal{CH}(P_S)$ are visited by means of a tandem search, that is organized as follows. Two queries are made on $\mathcal{CH}(P_S)$ for two points furthest away from ℓ, one above ℓ and the other below ℓ. We remove these two points from $\mathcal{CH}(P_S)$ and again query for two points furthest away, each on either side of ℓ. As soon as no two such points can be found, the search ends and the smaller of the sets $P_{S,a}$ and $P_{S,b}$ is identified. Assume $P_{S,a}$ is the smaller set, and let $\mathcal{CH}(P'_S)$ be the remainder of the convex hull. To complete $P_{S,b}$ we have to add P'_S to it. The case where $P_{S,b}$ is the smaller set is completely symmetrical.

For each point of $P_{S,a}$, each extension ℓ_e incident to this point is tested on intersection with ℓ, as follows:

- If ℓ_e is intersected by ℓ in a point on e, then ℓ_e is removed from E_p resp. E_c and its endpoints from $\mathcal{CH}(P_E)$, see Figure 7.

In case ℓ_e is intersected by ℓ in a point not on e and e lies above ℓ, then one of the two following actions is performed:

- ℓ_e is removed from E_p or E_c and made member of $E_{p,a}$, see Figure 8.i. The endpoints of ℓ_e below ℓ are removed from $\mathcal{CH}(P_E)$.



8.i: add to $E_{p,a}$ 8.ii: add to $E_{c,a}$

- ℓ_e is removed from E_c and made member of $E_{c,a}$, Figure 8.ii. The endpoints below ℓ are removed from $\mathcal{CH}(P_E)$, endpoints of ℓ_e above ℓ are removed from $\mathcal{CH}(P_E)$ and added to $P_{E,a}$.

Let the remainder of $\mathcal{CH}(P_E)$ be called $\mathcal{CH}(P'_E)$. After all points of $P_{S,a}$ and each extension containing one of those points are visited, there still may remain some points of P'_E above ℓ. These points are the endpoints of extensions ℓ_e that cross ℓ while e lies below ℓ and that are not visited yet. To find these points we use $\mathcal{CH}(P'_E)$. We query for the point that is furthest away from ℓ and above ℓ, and we remove it from $\mathcal{CH}(P'_E)$. We continue querying for more such points until no more points are left on $\mathcal{CH}(P'_E)$ above ℓ. Note that the remainder of P'_E equals $P_{E,b}$. For each endpoint visited we update the classification of ℓ_e incident to this endpoint as described above. All extensions $\ell_e \in E_p$ and $\ell_e \in E_c$ not visited in previous steps are added to $E_{p,b}$ resp. $E_{c,b}$.

At this stage the set of endpoints P_S, thus also the set of input line segments S, and the set P_E are each split by ℓ. The set of partition lines E_p and our set of candidate partition lines E_c are also split in two disjoint subsets by ℓ, one for the region R_a above ℓ and one for the region R_b below ℓ.

Before entering the next stage of recursion, we only have to construct the convex hulls for the sets at either side of ℓ. Where we can build the $\mathcal{CH}(P_{S,a})$ from scratch, since there are at most half as many points in $P_{S,a}$ as in P_S, we have to be careful in building the convex hull of the other subset $P_{S,b}$ of P_S. We cannot afford to build the datastructures that we need for the recursive call for the large set from scratch. At this stage however, we have $\mathcal{CH}(P'_S)$. We can therefore reconstruct the convex hull $\mathcal{CH}(P_{S,b})$ of the bigger set by reinsertion of points from P_S into $\mathcal{CH}(P'_S)$, that were removed during the tandem search from $\mathcal{CH}(P_S)$ and put into $P_{S,b}$. Note that there will be as many insertions as there are points in $P_{S,a}$, the smaller set. Recall that we already have $\mathcal{CH}(P_{E,b})$ at this stage. So we only have to build the convex hull $\mathcal{CH}(P_{E,a})$ of those points of P_E, that remained above ℓ.

Note that each extension ℓ_e is visited at most four times at each stage of the recursion: once for each of its endpoints and once for each endpoint of e.

Theorem 2.5 *The scheme as described above can be implemented to run in $O(n^2 \log n)$ time.*

Proof: The first step in our algorithm is to construct a visibility graph G_S on the set of input line segments S in order to find (candidate) partition lines. In [6] a method is presented that, given a set S of n non-intersecting line segments in the plane, computes its visibility graph G_S in time $O(n^2)$. The algorithm can easily be adapted to construct all extensions of the line segments of the visibility graph without extra asymptotic overhead. We spend only constant time at each extension to classify it as a partition line or candidate partition line. So the first part takes $O(n^2)$ time.

The second phase of the algorithm calculates recursively the set of partition lines, while updating our search structure at every stage of the recursion. The search is guided by the convex hulls of the sets P_S and P_E. Clearly our scheme needs a method for maintaining convex hulls in a dynamical way. The best bound currently known for the dynamic convex hull problem is due to Overmars and van Leeuwen [5]. They prove the existence of a dynamic structure for solving the 2-dimensional convex hull searching problem, such that queries can be done in $O(\log n)$ time, insertions and deletions can be done in $O(\log^2 n)$ time. Building this structure takes $O(n \log n)$ time and uses $O(n)$ storage. We use this structure to store $\mathcal{CH}(P_S)$. For $\mathcal{CH}(P_E)$, we can use an even more efficient structure, because we only delete points from P_E. In [4] Hershberger and Suri describe a datastructure for maintaining the convex hull of a set of n points, that can be built in time $O(n \log n)$ using $O(n \log n)$ space and that allows deletions of points in $O(\log n)$ amortized time per deletion. Queries can be done in time $O(\log n)$. Notice that the set P_E initially contains $O(n^2)$ points, so we actually need $O(n^2 \log n)$ time to build the data structure for $\mathcal{CH}(P_E)$.

Next we analyze the time taken in a single stage in the recursion. Finding the smaller subset of P_S using $\mathcal{CH}(P_S)$ will take time $O(k(\log n + \log^2 n))$, with k the size of the smaller subset. Visiting all extensions containing a point of the smaller subset takes time $O(kn)$. Deletion of the endpoints of the crossing extensions using $\mathcal{CH}(P_E)$ takes time $O(m \log n)$, where m is the number of such crossing extensions. What remains is the time needed to rebuild our search structures. Building the convex hull structure of the smaller subset of P_S from scratch takes $O(k \log k)$ time. The convex hull of the larger subset of P_S is constructed by reinserting the k elements of $P_S - P'_S - P_{S,smaller}$ into $\mathcal{CH}(P'_S)$, taking time $O(k \log^2 k)$. Finally, we have to build the convex hull structure of the endpoints of the extensions that are at the same side of ℓ as the smaller subset of P_S, taking time $O(k^2 \log k)$. Thus the total cost $P(n)$ at each stage equals $P(n) = O(kn + m \log n + k^2 \log k)$. The total running time can be bounded by the recursion: $T(n) \leq \max_{0 \leq k \leq \frac{n}{2}} P(k) + T(k) + T(n-k-1)$. We defined m to be the number of crossing extensions. Note that when we find a crossing extension, we delete a point

from P_E and $\mathcal{CH}(P_E)$, which will never be inserted again. Because we have at most n^2 extensions, the total number of crossing extensions that can occur over all stages is at most n^2. Thus, we can bound the total time spent on visiting those edges by $O(n^2 \log n)$. For the complexity that remains, we charge $c(n)$ to each point of the smaller subset of P_S, where $c(n) = \frac{1}{k}O(kn + k^2 \log k) = \frac{1}{k}O(kn + kn \log n) = O(n \log n)$. Every time an element of the smaller set gets charged, the size of the smaller set has at least been halved. Therefore the total charge on a single element can be bounded by: $c(n) + c(\frac{n}{2}) + c(\frac{n}{4}) + \cdots = n \log n + (\frac{n}{2}) \log(\frac{n}{2}) + (\frac{n}{4}) \log(\frac{n}{4}) + \cdots = O(n \log n)$. In total we have charged $O(n^2 \log n)$ time.

Adding the $O(n^2 \log n)$ time needed to visit all crossing extensions and the $O(n^2 \log n)$ time to build our initial data structures, results in a total running time that can be bounded by $O(n^2 \log n)$. □

Given the method presented above, it can be proved that it will always find a perfect binary space partition whenever such a partition exists.

Theorem 2.6 *The scheme described above outputs a perfect binary space partition if it exists, and reports its non-existence otherwise.*

Proof: If there exists a perfect binary space partition for a set of line segments, then there is a line that divides the set of line segments in two nonempty and disjoint subsets: the first line used in the partition. Observation 2.3 states, that as long as there are any such partition lines, they are found by our algorithm. As stated by Lemma 2.4, we may at each stage choose our partition line from all lines available, because for any subset of our set of line segments the existence of a perfect binary space partition is assured. As soon as there are no more lines available, we are finished or report the non-existence of a perfect binary space partition for this set of line segments. □

We applied our scheme to a set of disjoint line segments. We will now extend it to other disjoint convex objects. Observe that each partition line ℓ of a set S of convex objects that does not intersect any of the objects of S, can be rotated slightly until it touches two objects of S, each at a side. We call the line segment defined by the two points, in which ℓ touches an object, a tangent edge. Let the structure containing all these tangent edges be called a tangent visibility graph. Using the following observation, we can apply our scheme also to other sets of objects:

Observation 2.7 *If a line exists that partitions the set S of convex objects in exactly two non-empty subsets, without intersecting any of the objects of S, then there also exists such a line that is the extension of an edge of the tangent visibility graph of S.*

Theorem 2.8 *Assuming that the tangent visibility graph of a set of n convex objects can be computed in time $T(n)$, then finding a perfect binary space partition of this set of objects or reporting its non-existence can be done in $O(T(n)+n^2 \log n)$ time.*

3 Conclusion

We presented a method that constructs a perfect binary space partition of a set of non-intersecting line segments in the plane or that concludes that no such partition exists for that particular arrangement. Our method runs in time $O(n^2 \log n)$. We also proved the problem to be n^2-hard.

Several questions remain open. First of all, it might be possible to improve our algorithm. It will be difficult however, to obtain an $o(n^2)$ running time, since the problem is n^2-hard. Probably the most important open problem in this area still is to (dis)prove the conjecture stated by Paterson and Yao in [7]: for a set of n line segments in 2-dimensional space it is always possible to find a binary space partition of size $O(n)$.

References

[1] M. de Berg, M. Overmars, and O. Schwarzkopf. Computing and verifying depth orders. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 138–145, 1992.

[2] H. Fuchs, Z. M. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. *Comput. Graph.*, 14(3):124–133, 1980.

[3] A. Gajentaan and M. H. Overmars. n^2-hard problems in computational geometry. Technical Report RUU-CS-93-15, Dept. of computer science, Utrecht university, 1993.

[4] J. Hershberger and S. Suri. Applications of a semi-dynamic convex hull algorithm. In *Proc. 2nd Scand. Workshop Algorithm Theory*, volume 447 of *Lecture Notes in Computer Science*, pages 380–392. Springer-Verlag, 1990.

[5] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Syst. Sci.*, 23:166–204, 1981.

[6] M. H. Overmars and E. Welzl. New methods for computing visibility graphs. In *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, pages 164–171, 1988.

[7] M. S. Paterson and F. F. Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete Comput. Geom.*, 5:485–503, 1990.

[8] M. S. Paterson and F. F. Yao. Optimal binary space partitions for orthogonal objects. *J. Algorithms*, 13:99–113, 1992.