Approximate Range Searching Using Binary Space Partitions

Mark de Berg [a] and Micha Streppel [a,1],

[a] *Department of Computer Science, TU Eindhoven, P.O.Box 513, 5600 MB Eindhoven, the Netherlands.*

1. Introduction

Multi-functional data structures and BSP trees. In computational geometry, efficient data structures have been developed for a variety of geometric query problems: range searching, point location, nearest-neighbor searching, etc. The theoretical performance of these structures is often close to the theoretical lower bounds. In order to achieve close to optimal performance, most structures are dedicated to very specific settings. It would be preferable, however, to have a single *multi-functional geometric data structure*: a data structure that can store different types of data and answer various types of queries. Indeed, this is what is often done in practice.

Another potential problem with the structures developed in computational geometry is that they are sometimes rather involved, and that it is unclear how large the constant factors in the query time and storage costs are. Moreover, they may fail to take advantage of the cases where the input objects and/or the query have some nice properties. Hence, the ideal would be to have a multi-functional data structure that is simple and takes advantage of any favorable properties of the input and/or query.

A *binary space partition tree*, or *BSP tree*, is a space-partitioning structure where the subdivision of the underlying space is done in a hierarchical fashion using hyperplanes (that is, lines in case the space is 2D, planes in 3D, etc.) The hierarchical subdivision process usually continues until each cell contains at most one (or maybe a small number of) input object(s). BSP trees are used for many purposes; among these are range searching [1] and hidden surface removal with the painter's algorithm [11].

In some applications—hidden-surface removal is a typical example—the efficiency is determined by the size of the BSP tree. Hence, several researchers have proposed algorithms to construct small BSP trees in various different settings [2,5,13,15]. In this paper we focus on the query complexity of BSP trees.

Approximate range searching. Developing a multi-functional geometric data structure—one that can store any type of object and can do range searching with any type of query range—that provably has good performance seems quite hard, if not impossible. As it turns out, however, such results can be achieved if one is willing to settle for *ε-approximate range searching*, as introduced by Arya and Mount [3].

Here one considers, for a parameter $\varepsilon > 0$, the ε-extended query range Q_ε, which is the set of points lying at distance at most $\epsilon \cdot \mathrm{diam}(Q)$ from Q, where $\mathrm{diam}(Q)$ is the diameter of Q. Objects intersecting Q must be reported, while objects intersecting Q_ε (but not Q) may or may not be reported; objects outside Q_ε are not allowed to be reported. In practice, one would expect that for small values of ε, not too many extra objects are reported.

Our results. In this paper we show that it is possible to construct BSP trees for sets of disjoint segments in the plane, and for low-density scenes in any dimension, whose query time for approximate range searching is as good, or almost as good, as the best known bounds for point data [3,8,9]. More precisely, our results are as follows.

In Section 3 we study BSP trees for a set S of n disjoint line segments in the plane. We give a general technique to convert a BSP tree T_p for a set of points to a BSP tree T_S for S, such that the size of T_S is $O(n \cdot \mathrm{depth}(T_p))$, and the time for range searching remains almost the same.

In Section 4 we then consider low-density scenes.

Email addresses: m.t.d.berg@tue.nl (Mark de Berg), m.w.a.streppel@tue.nl (Micha Streppel).

[1] MS is supported by the Netherlands Organisation for Scientific Research (N.W.O.) under project no. 612.065.203.

We prove that any scene of constant density in $\mathbb{R}^d$ admits a BSP of linear size, such that range-searching queries with arbitrary convex ranges can be answered in $O(\log n + \min_{\varepsilon>0}\{(1/\varepsilon^{d-1}) + k_\varepsilon\})$, where k_ε is the number of objects intersecting Q_ε.

2. Preliminaries

In this section we briefly introduce some terminology and notation that we will use throughout the paper.

A BSP tree for a set S of n objects in $\mathbb{R}^d$ is a binary tree T with the following properties.

- Every (internal or leaf) node ν corresponds to a subset region(ν) of $\mathbb{R}^d$, which we call the *region* of ν. These regions are not stored with the nodes. When ν is a leaf node, we sometimes refer to region(ν) as a *cell*. The root node root(T) corresponds to $\mathbb{R}^d$.
- Every internal node ν stores a hyperplane $h(\nu)$. The left child of ν then corresponds to region(ν) $\cap$ $h(\nu)^-$, where $h(\nu)^-$ denotes the half-space below $h(\nu)$, and the right child corresponds to region(ν) $\cap$ $h(\nu)^+$, where $h(\nu)^+$ is the half-space above $h(\nu)$.

  A node ν stores, besides the splitting hyperplane $h(\nu)$, a list $L(\nu)$ with all objects contained in $h(\nu)$ that intersect region(ν).
- Every leaf node μ stores a list $L(\mu)$ of all objects in S intersecting the interior of region(μ). In our case the lists have a constant length.

The *size* of a BSP tree is defined as the total number of nodes plus the total size of the lists $L(\nu)$ over all nodes ν in T. Finally, for a node ν in a tree T, we use $T(\nu)$ to denote the subtree of T rooted at ν, and we use depth(T) to denote the depth of T.

3. BSPs for segments in the plane

Let S be a set of n disjoint line segments in the plane. In this section we describe a general technique to construct a BSP for S, based on a BSP on the endpoints of S. The technique uses a segment-tree like approach similar to, but more general than, the deterministic BSP construction of Paterson and Yao [13]. The range-searching structure of Overmars et al.[12] uses similar ideas, except that they store so-called long segments—see below—in an associated structure, so they do not construct a BSP for the segments. The main extra complication we face is that we must ensure that we only work with the relevant portions of the given tree during the recursive construction, and prune away irrelevant portions. The pruning has to be done carefully, however, because too much pruning can have a negative effect on the query time. Next we describe the construction in more detail.

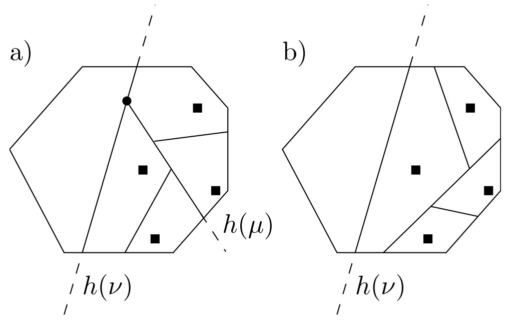

Figure 1. Illustration of the pruning strategy. The black squares indicate input segments. a) There is a T-junction on $h(\nu)$: a splitting line in the subtree ends on $h(\nu)$. Pruning $h(\nu)$ would partition the empty part of the region, which might have a negative effect on the query time. b) There is no T-junction on $h(\nu)$ and $h(\nu)$ can be pruned.

Let P be the set of $2n$ endpoints of the segments in S, and let T_P be a BSP tree for P. We assume that T_P has size $O(n)$, and that the leaves of T_P store at most one point from P. Below we describe the global construction of the BSP tree for S. Some details of the construction will be omitted here.

The BSP tree T_S for S is constructed recursively from T_P, as follows. Let ν be a node in T_P. We call a segment $s \in S$ *short* at ν if region(ν) contains an endpoint of s. A segment s is called *long* at ν if (i) s intersects the interior of region(ν), and (ii) s is short at parent(ν) but not at ν. In a recursive call there are two parameters: a node $\nu \in T_P$ and a subset $S^* \subset S$, clipped to lie within region(ν). The recursive call will construct a BSP tree T_{S^*} for S^* based on $T_P(\nu)$. Initially, $\nu = \text{root}(T_P)$ and $S^* = S$. The recursion stops when S^* is empty, in which case T_{S^*} is a single leaf.

Let $L \subset S^*$ be the set of segments from S^* that are long at ν. The recursive call is handled as follows.

(i) If L is empty, we compute $S_l = S^* \cap h(\nu)^-$ and $S_r = S^* \cap h(\nu)^+$.

If both S_l and S_r are non-empty, we create a root node for T_{S^*} which stores $h(\nu)$ as its splitting line. We then recurse on the left child of ν with S_l and on the right child of ν with S_r to construct respectively the left and the right subtree of the root.

If one of S_l and S_r is empty, it seems the splitting line $h(\nu)$ is useless in T_{S^*}. We have to be careful, however, that we do not increase the query time: the removal of $h(\nu)$ can cause other splitting lines, which used to end on $h(\nu)$, to extend further. Hence, we proceed as follows. Define a *T-junction*, see Fig. 1, to be a vertex of the original BSP subdivision induced by T_P . To decide whether or not to use $h(\nu)$, we check if $h(\nu) \cap R$ contains a T-junction in its interior, where R is the region that corresponds to the root of T_{S^*}. If this is the case, we do not prune.

(ii) The second case is when L is not empty. Now the long segments partition $region(\nu)$ into $m := |L| + 1$ regions, $R_1, \ldots, R_m$. We take the following steps.
   (i) We split $S^* \setminus L$ into m subsets $S_1^*, \ldots, S_m^*$, where S_i^* contains the segments from S^* lying inside R_i.
   (ii) We construct a binary tree T with $m-1$ internal nodes whose splitting lines are the lines containing the long segments. The leaves of T correspond to the regions R_i, and will become the roots of the subtrees to be created for the sets S_i^*. T is balanced by the sizes of the sets S_i^*, as in the trapezoid method for point location [14].

   The tree T_{S^*} then consists of the tree T, with, for every $1 \leqslant i \leqslant m$, the leaf of T corresponding to R_i replaced by a subtree for S_i^*. More precisely, each subtree T_i is constructed using a recursive call with node ν and S_i^* as parameters.

The following theorem states the performance of the BSP. Its proof is omitted in this extended abstract.

Theorem 1 *Let R be a family of constant-complexity query ranges in $\mathbb{R}^2$. Suppose that for any set P of n points in $\mathbb{R}^2$, there is a BSP tree T_P of linear size, where each leaf stores at most one point from P, with the following property: any query with a range from R intersects at most $v(n,k)$ cells in the BSP subdivision, where k is the number of points in the query range. Then for any set S of n disjoint segments in $\mathbb{R}^2$, there is a BSP tree T_S such that*
(i) the depth of T_S is $O(depth(T_P))$
(ii) the size of T_S is $O(n \cdot depth(T_P))$
(iii) any query with a range from R visits at most $O((v(n,k) + k) \cdot depth(T_P))$ nodes from T_S, where k is the number of segments intersecting the range.
The BSP tree T_S can be constructed in $O(n \cdot (depth(T_P))^2)$ time.

Several of the known data structures for range searching in point sets are actually BSP trees. Examples are ham-sandwich trees [10], kd-trees [7], and BAR-trees [8,9]. Here we focus on the application using BAR-trees, as it gives good bounds for approximate range searching for line segments in the plane.

One can construct BAR-trees with logarithmic depth, such that the number of leaves visited by a query with a convex query range Q is bounded by $O((1/\varepsilon) + k_\varepsilon)$, where k_ε is the number of points inside the extended query range Q_ε.

Combining this with theorem 1 (and a specialized construction algorithm that speeds up the construction time by a logarithmic factor) we get the following results.

Corollary 2 *Let S be a set of n disjoint segments in $\mathbb{R}^2$. In $O(n \log n)$ time one can construct a BSP tree for S of size $O(n \log n)$ and depth $O(\log n)$ such that a range query with a constant-complexity convex range can be answered in time $O(\min_{\varepsilon>0}\{(1/\varepsilon)\log n + k_\varepsilon \log n\})$, where k_ε is the number of segments intersecting Q_ε.*

4. BSPs for low-density scenes

Let S be a set of n objects $\mathbb{R}^d$. For an object o, we use $\rho(o)$ to denote the radius of the smallest enclosing ball of o. The *density* of a set S is the smallest number λ such that the following holds: any ball B is intersected by at most λ objects $o \in S$ with $\rho(o) \geqslant \rho(B)$ [6]. If S has density λ, we call S a λ-low-density scene. In this section we show how to construct a BSP tree for S that has linear size and very good performance for approximate range searching if the density of S is constant. Our method combines ideas from de Berg [5] with the BAR-tree of Duncan et al.[9] Our overall strategy, also used by de Berg [5], is to compute a suitable set of points that will guide the construction of the BSP tree. Unlike in [5], however, we cannot use the bounding-box vertices of the objects in S for this, because that does not work in combination with a BAR-tree. What we need is a set G of points with the following property: any cell in a BAR-tree that does not contain a point from G in its interior is intersected by at most κ objects from S, for some constant κ. We call such a set G a *κ-guarding set* [4] against BAR-tree cells, and we call the points in G *guards*.

The algorithm is as follows.

(i) Construct a κ-guarding set G for S, as explained below. The construction of the guarding set is done by generating $O(1)$ guards for each object $o \in S$, so that the guards created for any subset of the objects will form a κ-guarding set for that subset. Using the results of [4] it can be shown that there exists a guarding set for λ-low-density scenes against BAR-tree cells. Using special properties of (corner-cut) BAR-tree cells a guarding set of size $12n$ can be given for the planar case. We will use object(g) to denote the object for which a guard $g \in G$ was created.

(ii) Create a BAR-tree T on the set G using the algorithm of Duncan et al.[9], with the following adaptation: whenever a recursively call is made with a subset $G^* \subset G$ in a region R, we delete all guards g from G^* for which object(g) does not intersect R. This pruning step, which was not needed in [5], is essential to guarantee a bound on the query time. This leads to a BSP tree whose cells can only contain guards whose corresponding objects do not intersect the cell.

(iii) Search with each object $o \in S$ in T to determine which leaf cells are intersected by o. Store with each leaf the set of all intersected objects. Let T_S be the resulting BSP tree.

The following theorem states the performance of the BSP. Its proof is omitted in this extended abstract.

Theorem 3 *Let S be a λ-low-density scene consisting of n objects in $\mathbb{R}^d$. There exists a BSP tree T_S for S such that*

(i) the depth of T_S is $O(\log n)$

(ii) the size is $O(\lambda n)$

(iii) a query range with a convex range Q visits takes $O(\log n + \lambda \cdot \min_{\varepsilon>0}\{(1/\varepsilon) + k_\varepsilon\})$ time, where k_ε is the number of objects intersecting the extended query range Q_ε.

The BSP tree can be constructed in $O(\lambda n \log n)$ time.

References

[1] P.K. Agarwal and J. Erickson. Geometric range searching and its relatives. In: B. Chazelle, J. Goodman, and R. Pollack (Eds.), *Advances in Discrete and Computational Geometry*, Vol. 223 of *Contemporary Mathematics*, pages 1–56, American Mathematical Society, 1998.

[2] P.K. Agarwal, E. Grove, T.M. Murali and J.S. Vitter. Binary space partitions for fat rectangles. *SIAM J. Comput.* 29:1422-1448, 2000.

[3] A. Arya, D. Mount, Approximate range searching, Comput. Geom. Theory Appl. 17 (2000) 135–152.

[4] M. de Berg, H. David, M. J. Katz, M. Overmars, A. F. van der Stappen, and J. Vleugels. Guarding scenes against invasive hypercubes. *Comput. Geom.*, 26:99–117, 2003.

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

[6] M. de Berg, M.J. Katz, A. F. van der Stappen, and J. Vleugels. Realistic input models for geometric algorithms. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 294–303, 1997.

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

[8] C.A. Duncan, Balanced Aspect Ratio Trees, Ph.D. Thesis, John Hopkins University, 1999.

[9] C.A. Duncan, M.T. Goodrich, S.G. Kobourov, Balanced aspect ratio trees: Combining the advantages of k-d trees and octrees, In *Proc. 10th Ann. ACM-SIAM Sympos. Discrete Algorithms*, pages 300–309, 1999.

[10] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer-Verlag, 1987.

[11] H. Fuchs, Z. M. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. *Comput. Graph.*, 14(3):124–133, 1980. Proc. SIGGRAPH '80.

[12] M.H. Overmars, H. Schipper, and M. Sharir. Storing line segments in partition trees. *BIT*, 30:385–403, 1990

[13] 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.

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

[15] C.D. Tóth. Binary Space Partitions for Line Segments with a Limited Number of Directions. *SIAM J. Comput.* 32:307–325, 2003.