SIAM J. DISCRETE MATH.
Vol. 15, No. 2, pp. 252–267

EXACT SIZE OF BINARY SPACE PARTITIONINGS AND IMPROVED RECTANGLE TILING ALGORITHMS*

PIOTR BERMAN†, BHASKAR DASGUPTA‡, AND S. MUTHUKRISHNAN§

Abstract. We prove the following upper and lower bounds on the exact size of binary space partition (BSP) trees for a set of n isothetic rectangles in the plane:

- An upper bound of $3n - 1$ in general, and an upper bound of $2n - 1$ if the rectangles tile the underlying space. This improves the upper bounds of $4n$ in [V. Hai Nguyen and P. Widmayer, *Binary Space Partitions for Sets of Hyperrectangles,* Lecture Notes in Comput. Sci. 1023, Springer-Verlag, Berlin, 1995; F. d'Amore and P. G. Franciosa, *Inform. Process. Lett.,* 44 (1992), pp. 255–259]. A BSP satisfying the upper bounds can be constructed in $O(n \log n)$ time.
- A worst-case lower bound of $2n - o(n)$ in general, and $\frac{3n}{2} - o(n)$ if the rectangles form a tiling.

The BSP tree is one of the most popular data structures in computational geometry, and hence even "small" factor improvements of $\frac{4}{3}$ or 2 on the previously known upper bounds that we show improve the performances of applications relying on the BSP tree. As an illustration, we present improved approximation algorithms for certain dual rectangle tiling problems using our upper bounds on the size of the BSP trees.

Key words. binary space partitions, exact bounds, tiling, approximation algorithms

AMS subject classifications. 68Q01, 68W25, 68W40

PII. S0895480101384347

1. Introduction.

Binary space partitioning (BSP) for a collection of geometric objects in the two-dimensional plane[1] is defined as follows. The plane is divided into two parts by cutting objects with a line if necessary. Each fragment of the object belongs solely to one of the parts it falls in. The two resulting parts of the plane are divided recursively in a similar manner; the process continues until at most one two-dimensional fragment of the original objects remains in any part of the plane.[2] This division process can be naturally represented as a binary tree (BSP tree) where a node represents a part of the plane and stores the cut that splits the plane into two parts that its two children represent; each leaf of the BSP tree represents the final partitioning of the plane and stores at most one fragment of an input object. Since a cut at some node may split an object into two, the number of regions in the final configuration, equivalently the number of leaves in the BSP tree, may exceed the number of input objects. Note that quadtrees, octtrees, and grid files are all related to BSPs. Figure 1.1 shows a binary space partition for a set of four rectangles and the corresponding tree.

*Received by the editors January 30, 2001; accepted for publication (in revised form) January 15, 2002; published electronically April 8, 2002.

http://www.siam.org/journals/sidma/15-2/38434.html

†Department of Computer Science, Pennsylvania State University, University Park, PA 16802 (berman@cse.psu.edu). This author's research was supported in part by NSF grant CCR-9700053 and by NLM grant LM05110.

‡Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60607-7053 (dasgupta@cs.uic.edu). This author's research was supported in part by NSF grants CCR-9800086 and CCR-0296041.

§AT& T Labs—Research, 180 Park Avenue, Florham Park, NJ 07932 (muthu@research.att.com).

[1]We restrict ourselves to the two-dimensional plane throughout this paper.

[2]The objects have to be *disjoint,* since otherwise no BSP exists in which each final part of the plane contains at most one fragment of an object.

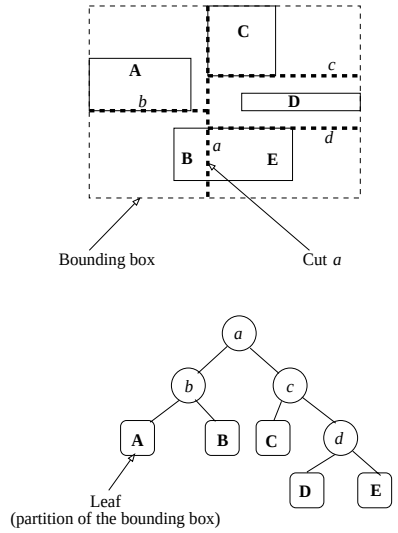

FIG. 1.1. *A binary space partition for four isothetic rectangles and the corresponding BSP tree. a, b, c, and d denote the four cuts in the partition, whereas A, B, C, D, and E denote the partitions of the bounding box of these rectangles generated by these cuts such that at most one fragment of each given rectangle is in each such partition.*

Since the introduction of BSP trees in [FKN80], they have become one of the most popular data structures. They present a way to implement a geometric divide-and-conquer strategy. They have found numerous applications in graphics (hidden surface removal [D94], shadow generation [CF89]), computational geometry (ray-tracing [NT86], visibility problems [T92], solid geometry [TN87]), robotics (motion planning [B93]), spatial databases [S90, van90] and approximation algorithms [KMP98, M90]. They are used in computer games such as DOOM and Quake, and there is a lot of publicly available code for different kinds of BSPs.

The fundamental parameter of interest about BSPs in all their applications is the *size*, that is, *the number of leaves* of a BSP tree. In two seminal papers, Paterson and Yao [PY90, PY92] established the first and essentially optimal bounds on the size of some BSPs. In two dimensions, which is of interest to us, they proved that any set of n line segments has a BSP of size $O(n \log n)$; for axis-parallel line segments, they proved that BSPs of $O(n)$ size exist. Very recently, Tóth [T01] showed that there exists n disjoint line segments in the plane such that any BSP of these segments must have a size of at least $\Omega(\frac{n \log n}{\log \log n})$. Some results are in [deGO] for some more general class of objects on the plane. The problem of bounding the size of BSPs remains an active research area, e.g., in higher dimensions, for objects with bounded aspect ratio, etc. [NW95, AG+96].

In this paper, we focus on axis-parallel (isothetic) rectangles as do Paterson and Yao in [PY92]. Such rectangles form a very important class of objects in application domains because complex objects are often replaced by their bounding rect-

angles. They also arise naturally in planar tiling problems such as constructing two-dimensional histograms and as subproblems when higher-dimensional hyperrectangles are projected on two dimensions. Our focus is on proving bounds on the *exact* size of BSP trees in the worst case, that is, not just asymptotic size but also the exact constants involved. In this paper, we prove the following results:

1. There exists a BSP tree of size at most $3n - 1$ for collection of n isothetic rectangles. If the rectangles form a tiling (that is, the given set of rectangles partition a rectangular region), then we prove an improved upper bound of $2n - 1$. A BSP satisfying the upper bounds can be constructed in $O(n \log n)$ time in either case.
   Paterson and Yao proved an upper bound of $12n$ in [PY92]; subsequent improvements have led to the current best upper bound of $4n$ [NW95, dAF92].
2. We also present lower bounds on the size of a BSP tree for n isothetic rectangles in the worst case: we prove a lower bound of $2n - o(n)$. If the rectangles must form a tiling of the space, we show a lower bound of $\frac{3n}{2} - o(n)$.

Size of the BSP trees determines the time and space of all applications reliant on them; improvements even by "small" factors ($\frac{4}{3}$ or 2) are highly desirable. Even construction time in securing these improvements is not a bottleneck because our upper bounds can be achieved by efficient algorithms taking $O(n \log n)$ time. Our constructions are modifications of the original algorithm in [PY92], matching it in their running times. The bulk of our technical achievement is our detailed amortized analyses of the algorithm.

Concurrent as well as subsequent to our work, there has been considerable interest in proving tight lower and upper bounds on the exact size of the BSP tree for various objects. In particular, subsequent to our initial submission of this manuscript, Dumitrescu, Mitchell, and Sharir [DMM01] have proved an asymptotic lower bound of $2n - o(n)$ for BSPs for n orthogonal line segments and an improved lower bound of $\frac{9n}{4} - o(n)$ for sizes of BSPs for n isothetic rectangles.[3] However, this last lower bound of [DMM01] does not seem to apply to the case when the rectangles must form a tiling of the underlying space.

We show an application of our results for the BSP tree. The dual tiling problem is to cover a two-dimensional array with nonoverlapping rectangles so that the total (or maximum) "weight" of the tiling is bounded by a specified threshold. (The weight of a tiling is determined by different applications in spatial data structures, databases, parallel load balancing, video compression, etc.) The goal is to minimize the number of rectangles used in the tiling. Using our upper bounds on the BSP trees for isothetic rectangles, we present approximations for the dual tiling problem which significantly improves known results in approximation factor and/or running time. This technique is general and applies to different "weight" functions and optimization criteria.

Map. In the rest of the paper, we first present the upper bounds (section 2) followed by the lower bounds (section 3). We then present applications of our results to the dual rectangle tiling problems in section 4.

2. Upper bounds. Given a rectangular region $\mathbf{R}$ containing a set of n disjoint isothetic rectangles, a BSP of $\mathbf{R}$ consists of recursively partitioning $\mathbf{R}$ by a horizontal or vertical line into two subregions and continuing in this manner for each of the two subregions until each obtained region intersects at most one rectangle. If a rectangle

[3] They also announced in the same conference that the lower bound of $\frac{9n}{4} - o(n)$ can be further improved to $\frac{7n}{3} - o(n)$.

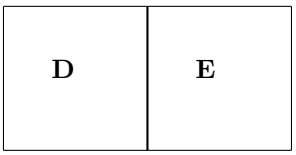

FIG. 2.1. *A cut of* **C**.

is intersected by a cutting line, it is then split into disjoint rectangles whose union is the intersected rectangle. The *size* of a BSP is the number of regions produced. Equivalently, a BSP of a set of rectangles contained in a rectangular region **R** is a binary tree, where each node is a rectangular subregion of **R**, each internal node is the union of its two children, the intersection of each leaf with the rectangles in our collection has at most one rectangle, and the root is the rectangular region **R**. The partition of **R** that corresponds to a BSP is the collection of the leaves of this tree, and the size of the BSP is the number of leaves in the tree. A set of rectangles form a tiling if they partition some rectangular region **R**. It is shown in [KMP98, MPS99] that a BSP of **R** with the minimum number of leaves can be computed using dynamic programming technique in $O(n^5)$ time.

THEOREM 2.1. *We can compute a BSP of* **R** *of size at most* $3n-1$ *in* $O(n \log n)$ *time.*

Restricting the set of rectangles in our collection to be a tiling of **R** yields even better bounds on the size of the BSP.

THEOREM 2.2. *Assume that the rectangles in our collection form a tiling of* **R**. *Then, we can compute a BSP of* **R** *of size at most* $2n-1$ *in* $O(n \log n)$ *time.*

In the rest of this section, we will prove both theorems.

2.1. General schema for the proofs of Theorems 2.1 and 2.2. Both Theorems 2.1 and 2.2 use the accounting method of amortized analysis for tighter bounds on the size of their respective BSPs. First, we describe the general schema and then later provide details of how this schema can be applied to each case.

Suppose that a node (region) **C** in a BSP has two children, say **D** and **E**. Then the boundary between **D** and **E** is a line segment, horizontal or vertical, that extends between two points on the boundary of **C**. This segment is called a *cut* of **C** (see Figure 2.1).

Our proof is by induction. We use $\mathcal{N}$ to denote the given collection of the disjoint rectangles. For a subrectangle of **R**, say **C**, we define induced collection $\mathcal{N}_{\mathbf{C}}$, which consists of nonempty intersections of rectangles from $\mathcal{N}$ with **C**.

We start by declaring one of the sides of **R** to be *unbroken* and the others to be *broken*. Then we initially give money to the elements of $\mathcal{N} = \mathcal{N}_{\mathbf{R}}$ according to the following formula (for some x and y, with $x \geq y \geq 1$): a rectangle not adjacent to a broken side gets x dollars, a rectangle adjacent to only one broken side gets y dollars, and any other rectangle (adjacent to two or more broken sides) gets one dollar. We will say that $\mathcal{N}_{\mathbf{R}}$ is *properly endowed* if the associated distribution of dollars satisfies this formula. The choice of x and y will vary for the two problems.

Our inductive step is the following. If a rectangle **C** has three broken sides and $\mathcal{N}_{\mathbf{C}}$ is properly endowed, then

- either $|\mathcal{N}_{\mathbf{C}}| = 1$;
- or we can cut **C** into two rectangles **D** and **E**, declare three broken sides for these rectangles, and provide proper endowment for $\mathcal{N}_{\mathbf{D}}$ and $\mathcal{N}_{\mathbf{E}}$ by re-

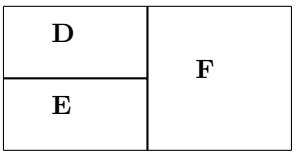

FIG. 2.2. *Cutting* **C** *twice into* **D**, **E**, *and* **F**.

distributing enough of the money from the proper endowment of $\mathcal{N}_{\mathbf{C}}$ (see Figure 2.1);

- or we can cut **C** twice, into three rectangles **D**, **E**, and **F** (see Figure 2.2), and make the declarations of broken sides and redistribution of dollars for all three new rectangles (one child and two grandchildren of **C**).

This inductive argument is sufficient (with $x = 3$ and $y = \frac{3}{2}$ for Theorem 2.1 and with $x = 2$ and $y = 1$ for Theorem 2.2), because every leaf in the resulting BSP gets at least one dollar, and initially there are at most $xn - 1$ dollars; thus we have less than $xn - 1$ leaves in our BSP since we can obviously assume that the root **R** is the smallest bounding rectangle of the given set of rectangles.[4] Finally, it is easy to eliminate empty regions in the resulting BSP, if so desired, by ensuring that each region intersects at least one rectangle of our collection: if the region for a nonroot node does not intersect any rectangle, then simply cover it by extending its sibling in the BSP and, if necessary, the descendants of this sibling.

The inductive step is easy if we can cut **C** without splitting any of the rectangles in $\mathcal{N}_{\mathbf{C}}$, i.e. , we have $\mathcal{N}_{\mathbf{C}} = \mathcal{N}_{\mathbf{D}} \cup \mathcal{N}_{\mathbf{E}}$, $\mathcal{N}_{\mathbf{C}} \neq \mathcal{N}_{\mathbf{D}}$, and $\mathcal{N}_{\mathbf{C}} \neq \mathcal{N}_{\mathbf{E}}$. A side of the child rectangle that is a (part of a) broken side of **C** is declared broken. If a child has two undeclared sides, we arbitrarily declare one of them to be broken. Since $x \geq y \geq 1$, it is easy to see that no rectangle from $\mathcal{N}_{\mathbf{C}}$ needs more money after this partition. We refer to this case as the *easy case* of the inductive step.

For ease of discussion, we assume that the unbroken side of **C** is the right side of **C** (by rotating the coordinate axes, if necessary).

2.2. Application of the general schema. Now, we give details of the application of the general schema for each of the theorems.

2.2.1. Proof of Theorem 2.1. Set $x = 3$ and $y = 3/2$. Our algorithm is a modification of the algorithm discussed in [PY92]. Refer to Figure 2.3. Let X be the longest rectangle which is adjacent to the broken left side of **C**, or, in the absence of any such rectangle, the rectangle in **C** whose left side is closest to the left side of **C** (break ties arbitrarily). Slide the right vertical side (segment) $\overline{a, b}$ of X until it either hits a rectangle in **C** or hits the unbroken side of **C**.[5] If $\overline{a, b}$ hits the unbroken side of **C**, then this is the easy case of the inductive step. Otherwise, let $\overline{c, d}$ be the position of $\overline{a, b}$ when it hits a rectangle Y of **C**. Notice that if Y is adjacent to two broken (horizontal) sides of **C**, then we have the easy case of the inductive step; hence we assume that this is not the case.

Our first cut is vertical through point d. If this cut does not split any rectangles of $\mathcal{N}_{\mathbf{C}}$, we have the easy case. Otherwise, we will assume that this cut splits a rectangle above line $\overline{b, d}$. The sides of **D**, **E**, and **F** contributed by the first cut are

[4] It is easy to see that in the beginning at least one rectangle will be adjacent to at least one broken side.

[5] The notation $\overline{a, b}$ denote the segment joining points a and b.

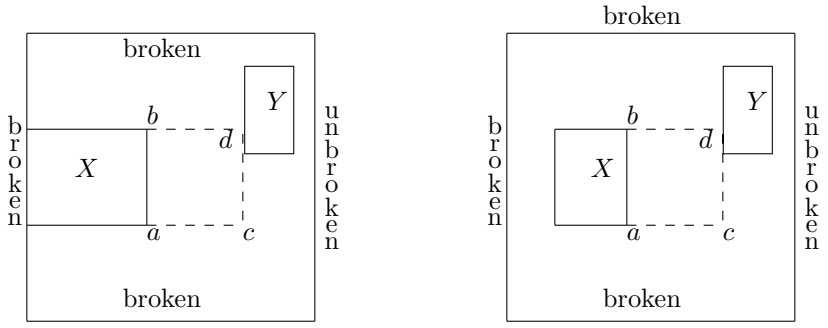

FIG. 2.3. *The placement* $\overline{c,d}$ *of slided* $\overline{a,b}$.

declared broken. (Rectangles **D**, **E**, and **F** were defined in the discussion of the general schema.) Our second cut will be the horizontal line $\overline{b,d}$ extended to the left side of **C**. The sides of children rectangles that are adjacent to that line are declared unbroken. The second cut is the easy case of the inductive step in the sense that it does not need to be analyzed in terms of its effects on the endowment but was essential because it provided the left products of the cuts with unbroken sides. While checking if we have enough money for the endowments, we analyze the first cut only. It splits a number of rectangles in **C**, among which at most two are adjacent to one broken side of **C** (one adjacent to the top side and the other to the bottom side of **C**). Every other rectangle that is cut had three dollars before the cut, and after the cut each of the two pieces needs $\frac{3}{2}$ dollars, which can be obtained by distributing the three dollars evenly between them. Hence, there are two cases to consider:

- Only one rectangle adjacent to a broken side of **C** is cut. Before the cut, the rectangle had $\frac{3}{2}$ dollars. After the cut, each of the two pieces need one dollar; hence it needs an extra $\frac{1}{2}$ dollar for proper endowment. However, because the status of rectangle Y is changed (that is, because Y becomes adjacent to at least one more broken side), it has a surplus of at least $\min\{x-y, y-1\} \geq \frac{1}{2}$ dollars, which can be used for this purpose.
- Two rectangles adjacent to a broken side of **C** are cut. Then, each of these rectangles needs an additional $\frac{1}{2}$ dollar. However, in this case, Y could not have been adjacent to any broken side of **C** before the cut but becomes adjacent to one broken side after the cut; hence it has a surplus of at least $x - y = \frac{3}{2}$ dollars, which can be used to provide the additional one dollar.

We now turn to the implementation of our algorithm. All the steps of our algorithm can be implemented in a manner similar to those in [dAF92], except the step that requires finding the placement $\overline{c,d}$ of the slided vertical side $\overline{a,b}$ of X if it was stopped by the left vertical side of some rectangle Y. We describe the implementation, assuming that the side $\overline{a,b}$ is vertical and the unbroken side of **C** is the right side of **C**; the other cases are similar. The x and y coordinates of the corners of the given rectangles can be preprocessed in $O(n \log n)$ time by sorting them and then replacing them by their rank in the respective sorted lists to ensure that each coordinate of any corner point of any rectangle is an integer from the set $\{1, 2, \ldots, 2n\}$. Whenever we generate a new rectangular partition, we also maintain the coordinates of its top left and bottom right corners. We now need to maintain a data structure on the left vertical sides of all rectangles in our collection **R** such that, given a query vertical segment (which is a part of the right vertical segment of some rectangle in our collection **R**) connecting

two points with coordinates (x, y_0) and (x, y_1), the query returns the left vertical side of a rectangle connecting points with coordinates (x', y'_0) and (x', y'_1) such that the interval $[y_0, y_1]$ overlaps the interval $[y'_0, y'_1]$, $x' \geq x$ and x' is the minimum possible; then we need to check if the segment connecting the points (x', y_0) and (x', y_1) is inside our current bounding rectangular region $\mathbf{C}$ using the coordinates of its top left and bottom right corners of $\mathbf{C}$. (Otherwise, the segment $\overline{a, b}$, when slid to the right, would hit the right unbroken side of $\mathbf{C}$.)

Therefore, it suffices for our purpose to build a data structure $\mathcal{D}$ on a set of n two-dimensional points $\{(x, y) \mid x, y \in \{1, 2, \ldots, 2n\}$ in $O(n \log n)$ time and using $O(n \log n)$ space such that a query range (a, b, c) must return a point in $\mathcal{D}$ (if any) in $O(\log n)$ time with $y \leq b \leq c$, $a \geq x$ and a is the minimum possible.[6] We modify the priority search tree (PST) data structure (e.g., see [M85]) for this purpose. A PST is a data structure on n two-dimensional points such that given a query range (a, b, c) it is possible to find in $O(\log n)$ time a point (x, y) in the PST with $x \geq a$ (or, $x \leq a$) and $b \leq y \leq c$; the first dimension is called the priority dimension and the remaining two dimensions are called the search dimensions. Moreover, it is possible to build a PST in $O(n \log n)$ time using $O(n)$ space. To design $\mathcal{D}$, we first build a binary search tree T on the y-coordinates of the given points. At each node v we store a horizontal line H_v which represents a value in between the maximum y-coordinate value in the left subtree and the minimum y-coordinate value in the right subtree. At the left (respectively, right) child of v we store an appropriate PST whose search dimension is based on the x-coordinate values and whose priority dimension is based on the y-coordinate values of the given points. Given a query range (x, y_0, y_1) with the first dimension being the priority dimension, we search in T and find the highest node v such that H_v is contained in $[y_0, y_1]$. H_v splits the range into two subranges that are both unbounded with respect to the priority dimension. (For example, the part of the range below H_v is unbounded in the negative y-dimension and similarly for the other range.) So, we simply use the PSTs to find the point in each subrange which has the minimum value in the search dimension (i.e., the x-dimension). From these two queries we can infer our desired point. We do this query twice at a single node v, after searching down T. So the query time is $O(logn)$. The storage per level of T is $O(n)$, so it is $O(nlogn)$ overall. If we are given the points in sorted order along the search dimension, then a PST on those points can be built in linear time. Thus, we can presort all the points once and then build the entire structure bottom-up (merging the sorted lists at the left and right child of v to get the sorted list at v). So the total preprocessing time is also $O(n \log n)$.

2.2.2. Proof of Theorem 2.2. Set $x = 2$ and $y = 1$. For a subrectangle $\mathbf{U}$ of $\mathbf{C}$ let $\mathcal{B}_{\mathbf{U}}$ to be the union of the boundary segments (sides) of the rectangles of $\mathcal{N}_{\mathbf{U}}$ that are not contained in the boundary of $\mathbf{U}$. We pick a horizontal line segment $\overline{a, b}$ with the following properties (see Figure 2.4):

- a belongs to the left side of $\mathbf{C}$ (remember that the left vertical side of $\mathbf{C}$ is broken);
- $\overline{a, b}$ is a horizontal line segment;
- $\overline{a, b}$ is a subset of $\mathcal{B}_{\mathbf{C}}$ (that is, $\overline{a, b}$ consists of line segments each of which is an element of $\mathcal{B}_{\mathbf{C}}$);
- no point on $\overline{a, b}$, except possibly the points a and b, lie on a side of $\mathbf{C}$;
- $\overline{a, b}$ is a *longest* segment with the above properties.

[6] We would like to thank Prof. Ravi Janardan for helping us with this part of the implementation.

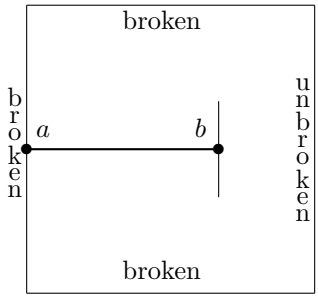

FIG. 2.4. *Picking the horizontal line segment* $\overline{a, b}$.

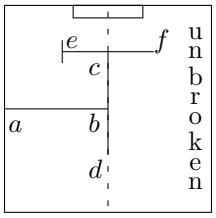

FIG. 2.5. *Extending* $\overline{b, c}$ *upwards towards the boundary of* $\mathbf{C}$.

Assume that b is not on a side of $\mathbf{C}$; otherwise we would have the easy case as described before. Also, if there is no such segment $\overline{a, b}$ of length greater than zero, then again we would have the easy case, so we may assume that such a segment $\overline{a, b}$ of length greater than zero exists. Our first cut is vertical through point b. The sides of $\mathbf{D}$, $\mathbf{E}$, and $\mathbf{F}$ contributed by this cut are declared broken. The second cut is (a, b) itself, and the sides of $\mathbf{D}$ and $\mathbf{E}$ contributed by this cut are declared unbroken. As before, any side of $\mathbf{D}$, $\mathbf{E}$, and $\mathbf{F}$ that is part of a broken side of $\mathbf{C}$ is also declared broken. It remains to show that the endowment of $\mathcal{N}_{\mathbf{C}}$ is sufficient to provide the endowments of $\mathcal{N}_{\mathbf{D}}$, $\mathcal{N}_{\mathbf{E}}$, and $\mathcal{N}_{\mathbf{F}}$.

Since the given collection of rectangles form a tiling, point b must lie in the interior of a vertical segment from $\mathcal{B}_{\mathbf{C}}$ that is perpendicular to $\overline{a, b}$. We extend this segment maximally vertically in both directions, so it is a union of segments $\overline{c, b}$ and $\overline{b, d}$ (see Figure 2.5). If both c and d are on the boundary of $\mathbf{C}$, then the cut $\overline{c, d}$ makes it an easy case. Thus we can assume that at least one of them, say c, is located in the interior of $\mathbf{C}$ and thus in the interior of some maximal horizontal segment (i.e., a horizontal segment that cannot be extended in either direction) from $\mathcal{B}_{\mathbf{C}}$, say $\overline{e, f}$. Assume that e is to the left of f.

Observe that e must be located in the interior of $\mathbf{C}$, since otherwise the segment $\overline{e, f}$ would be chosen, rather than $\overline{a, b}$. Thus, since the given collection of rectangles form a tiling, the rectangle of $\mathcal{N}_{\mathbf{C}}$ that is adjacent to segments $\overline{e, c}$ and $\overline{b, c}$ is not adjacent to the boundary of $\mathbf{C}$, and therefore it was endowed with two dollars. During our cuts this rectangle is not subdivided, and it is adjacent to our new broken side; thus we can take one dollar from its endowment to be used elsewhere. To perform the first cut, we extend $\overline{b, c}$ upwards toward the boundary of $\mathbf{C}$. We may cut a number of rectangles from $\mathcal{N}_{\mathbf{C}}$. The last one is adjacent to a broken boundary segment of $\mathbf{C}$, so it has only one dollar, and after the cut we need one dollar for each of its two

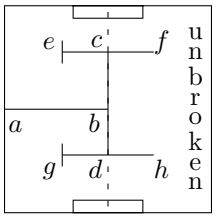

FIG. 2.6. *Extending* $\overline{b,d}$ *downwards towards the boundary of* $\mathbf{C}$.

parts. This is where we used a dollar that we have extracted a moment ago. Other rectangles which are cut cannot be adjacent to a broken boundary, since the only possibility is if the rectangle is adjacent to the boundary segment where a is located, but then such a rectangle would provide a better (longer) choice for $\overline{a,b}$. Thus each of these rectangles has two dollars, but after the cut their parts are adjacent to the new broken boundary segment, so they need one dollar each.

Now we extend $\overline{b,d}$ downwards toward the boundary of $\mathbf{C}$ (see Figure 2.6). If d is on the boundary of $\mathbf{C}$, then this extension does not cut any rectangles of $\mathcal{N}_{\mathbf{C}}$. Otherwise, we cut some rectangles. By the same reasoning as before, we need money (one dollar) only for the last rectangle that is cut. It is easy to see that in this case point d is in the interior of a maximal segment from $\mathcal{B}_{\mathbf{C}}$, say $\overline{g,h}$. By the same reasoning as applied before to points c, e, and f, we can conclude that the rectangle adjacent to segments $\overline{b,d}$ and $\overline{g,d}$ is not adjacent to a broken side of $\mathbf{C}$, so it had two dollars, and becomes adjacent to a broken side after the partition, so it provides us with that additional one dollar.

We now turn to the implementation of our algorithm. All the steps of our algorithm can be implemented in a manner similar to those in [dAF92], except the step that requires finding the segment $\overline{a,b}$. We describe the implementation assuming that the side $\overline{a,b}$ is horizontal and the unbroken side of $\mathbf{C}$ is the right side of $\mathbf{C}$; the other cases are similar. Note that we do not need to know or enumerate the rectangles whose sides contributed to the segment $\overline{a,b}$. Consider the set of horizontal boundary segments of all rectangles in our collection $\mathbf{R}$ and join two segments if they share an endpoint. This will give us a set of segments in which each segment is a result of the joining of horizontal boundary segments of one or more rectangles. This can be easily obtained in $O(n \log n)$ time by sorting all the horizontal boundary segments of all rectangles in our collection $\mathbf{R}$ by their distance from the y-axis. Finding the longest segment $\overline{a,b}$ and maintaining them for each partition that we generate can now be done by the same implementation as in [dAF92].

3. Lower bounds. In this section, we prove the following lower bounds.

THEOREM 3.1. *The following lower bounds hold:*

(a) *There exists a configuration of* n *isothetic rectangles that form a tiling of the space such that any BSP tree for them is of size at least* $\frac{3n}{2} - o(n)$.

(b) *If the rectangles are not required to form a tiling, there exists a configuration of* n *rectangles such that any BSP tree for them is of size at least* $2n - o(n)$.

Proof. We first focus on part (a) when the rectangles form a tiling.

We will construct the configuration in *stages*. See Figure 3.1. Stage 1 is the five rectangle configuration shown in the bottom left part of Figure 3.1. We construct stage 1 from stage 2 as follows. We reflect a stage 1 configuration along each axis as

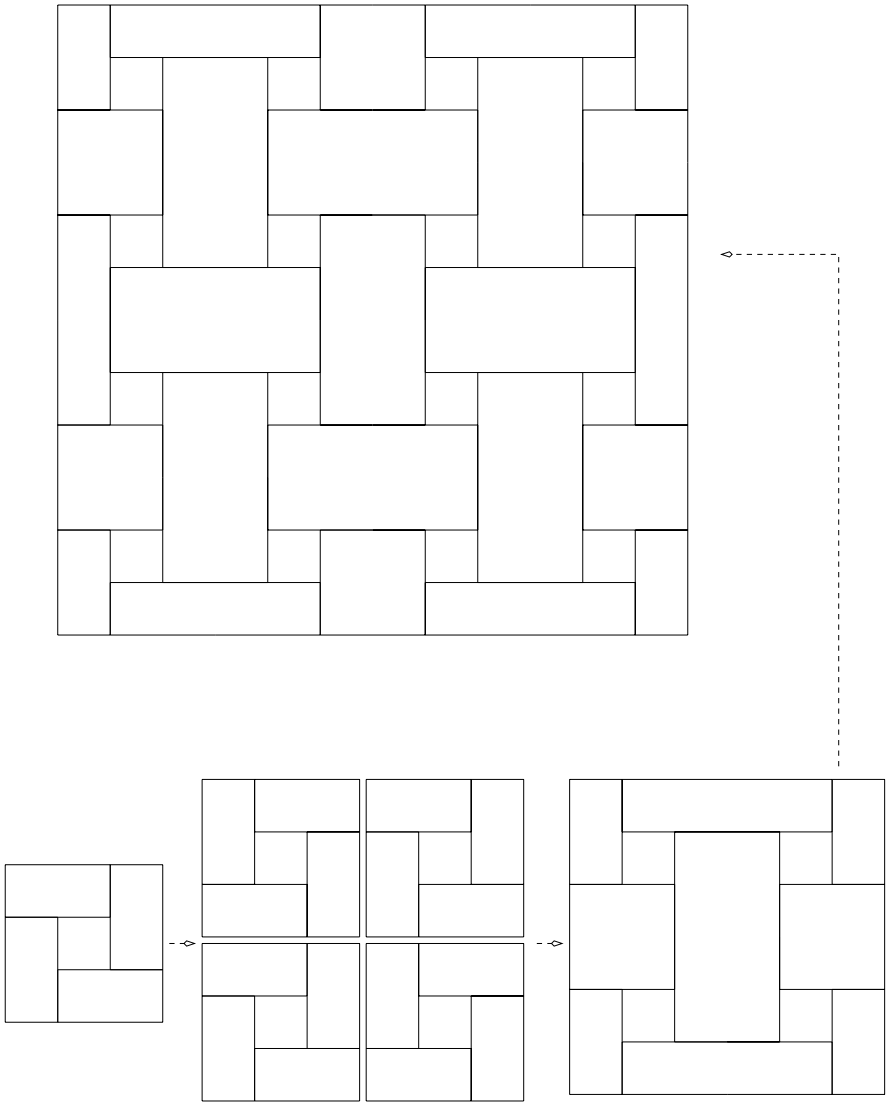

FIG. 3.1. *Lower-bound construction for a set of isothetic rectangles that form a tiling. Construction of stages* 1*,* 2*, and* 3*.*

well as across the lower right corner to get a configuration of four copies of stage 1. We merge any two mirror reflections abutting the borders of the four copies into one rectangle, whenever possible. The resulting configuration is shown in the bottom right part of Figure 3.1 with 13 rectangles. That is stage 2. Stage 3 consists of four copies of the configuration from stage 2 reflected and merged along the borders just as before, and so on.

Define B_i to be the number of rectangles on the boundary of any one side of a stage i configuration (from inside) and R_i to be total number of rectangles. For example, for Stage 1 $B_1 = 2$ and $R_1 = 5$ and for stage 2 $B_2 = 3$ and $R_2 = 13$.

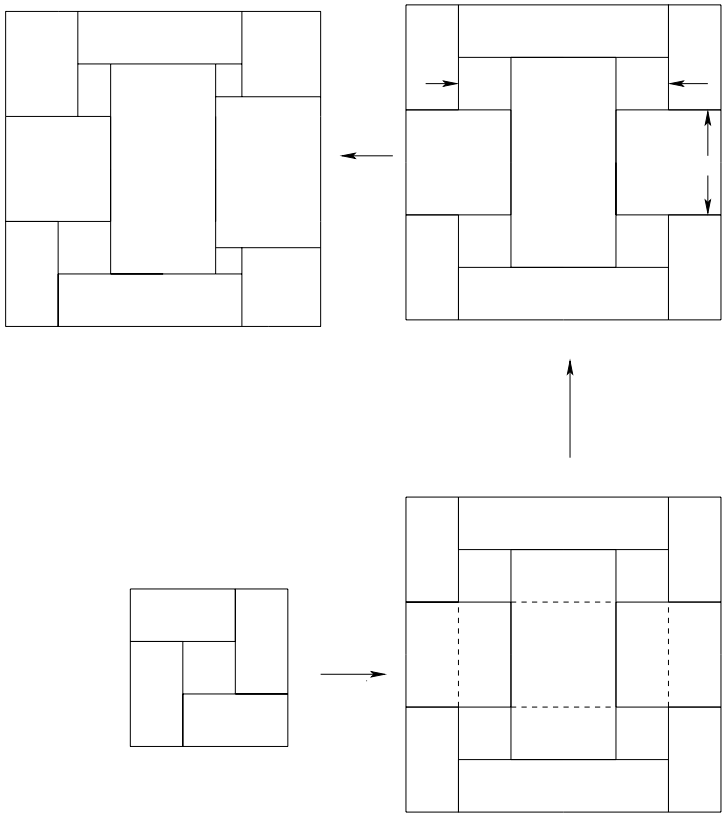

FIG. 3.2. *Modifying stage* 2.

Furthermore, we have $B_i = 2B_{i-1} - 1$; this is because any boundary of Stage i consists of two copies of stage $i-1$, but a single rectangle that abuts their joint is collapsed in the construction. By symmetry, all sides of stage i have B_i rectangles on the boundary. Also, note that

$$R_i = 4R_{i-1} - (4B_{i-1} - 1),$$

because in stage i we have the rectangles from four copies of stage $i-1$, but on each of the four joints of the copies we lose one copy of the rectangles on the boundary and gain the one rectangle at the center. Solving the recurrences, we have $B_i = 2^i - 2^{i-1} + 1$, and $R_i = 2(4^{i-1}) + 2^i + 1$. The total number of rectangles in the final stage is n.

We will now modify the stages slightly, but it is critical.

Any BSP for stage 1 needs to cut one of the rectangles and therefore generates at least one additional rectangle. Our goal is to claim that any BSP for stage 2 needs at least four additional rectangles (for reasons that will be clear soon). The intuition is that each of the four copies of stage 1 that was used to generate stage 2 needs a separate BSP cut and therefore will generate a separate additional rectangle in the BSP. This is not true as such for the 13 rectangle configuration shown in Figure 3.1 and reproduced as the bottom-right configuration in Figure 3.2. As the two parallel

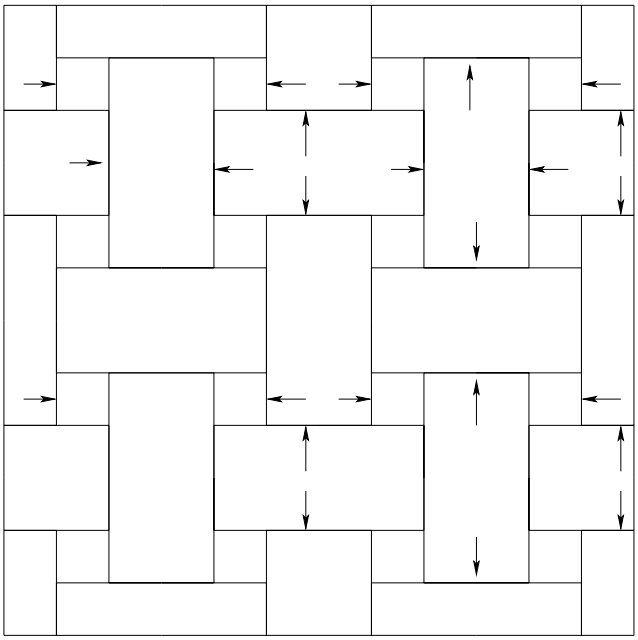

FIG. 3.3. *Modifying stage* 3. *Solid arrows indicate in what direction the side of a rectangle has to be moved slightly.*

sets of dotted lines show, a single cut can serve as the BSP cut of two copies juxtaposed horizontally or vertically. This is because of the symmetry of our construction. In order to break this symmetry, we need to make sure that the vertical line of one of the top two copies of stage 1 does not align with the vertical line in the copy of stage 1 directly underneath it. This is accomplished by moving the vertical lines marked in the figure a little along the direction of the arrow. The process is similar for the horizontal lines of the right two copies of stage 1. This results in the modified stage 2 shown in Figure 3.2. Now we need at least four additional rectangles in any BSP tree of Stage 2.

Modifying stage 3 is similar. Refer to Figure 3.3. We start with the modified stage 2 and repeat the construction of stage 3 as before. Again, we need to make sure that the horizontal lines in the top (bottom) left copy do not line up with those in the top (bottom) right copy and the vertical lines in the top left (right) copy do not line up with the bottom left (right) copy. This is accomplished by moving various line segments a little *further* along the direction shown in the figure. In this manner, we can ensure that the individual BSP cuts of each of the four copies of stage 2 are all needed in the BSP of stage 3. As a result, we have $4 * 4 = 16$ additional rectangles for the BSP of modified stage 3. It is easy to see that we can repeat this procedure to construct modified stage i such that any BSP of it will need 4^i additional rectangles.

That completes the description of the configuration. Clearly, B_i's and R_i's remain unchanged during the modification of stages.

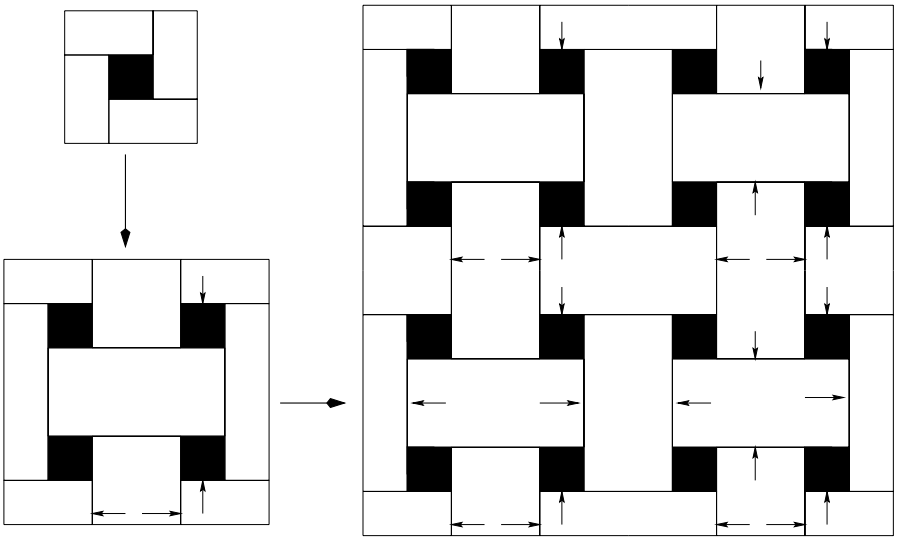

FIG. 3.4. *Lower bound construction for a set of isothetic rectangles, not necessarily forming a tiling. The black region is a hole.*

We will now complete the lower bound calculation. Define C_i to be the number of additional rectangles (in excess of R_i) needed in any binary space partition for modified stage i. From our construction, $C_i = 4C_{i-1}$; solving the recurrence, we get $C_i = 4^{i-1}$. Define the lower bound *factor* on the BSP tree size for a given set of rectangles to be the ratio of the size of a *minimum-size* BSP tree to the given number of rectangles. Clearly, for our construction, the lower bound factor on the BSP tree size is $\frac{C_i+R_i}{R_i} = 1 + \frac{C_i}{R_i}$. From the formulas for C_i and R_i, we have that the lower bound factor on the size of the BSP tree is at least

$$1 + \frac{4^{i-1}}{2(4^{i-1}) + 2^i + 1} = \frac{3}{2} - \frac{2^i + 1}{2(4^{i-1}) + 2^i + 1} \geq \frac{3}{2} - \frac{\sqrt{2n} + 1}{n} = \frac{3}{2} - o(1).$$

For the case when the rectangles are not required to be a tiling, we repeat the same construction as above, except that we leave the central square as a hole. The first two stages are shown in Figure 3.4, where the shaded region is a hole. Repeating the argument above (with $n = 4^{i-1} + 2^i + 1$), we obtain the lower bound factor to be at least

$$1 + \frac{4^{i-1}}{4^{i-1} + 2^i + 1} = 2 - \frac{2^i + 1}{4^{i-1} + 2^i + 1} \geq 2 - \frac{2\sqrt{n} + 1}{n} = 2 - o(1). \qquad \square$$

4. Improved rectangle tiling algorithms. We present improved approximation algorithms for several dual rectangle tiling problems. These rectangular tiling problems and our improved approximation algorithms for them are of interest in many areas: databases, parallel load balancing, spatial data structures, video compression, etc. As an example, consider the following problem [MPS99].

Dual rectangle tiling (DRTILE) problem. Given an $n \times n$ array A of positive integers and a parameter $\delta > 0$, the problem is to tile (partition) A with the minimum number of axis-parallel rectangular tiles such that the maximum weight of any tile is at most δ. The *weight* of a tile is the sum squared error of each entry from the average of the entries in that tile.

The DRTILE problem is known to be NP-hard [MPS99]. Our new results are as follows. The previously best known algorithm for (a) in the following theorem also used no more than twice as many tiles as needed, but its running time was $O(n^{\geq 10})$ [KMP98]! The previously best known algorithm for (b) in the following theorem with the same running time used $4p^*O(1/\epsilon^2)$ tiles[7] and hence was *worse* by a factor of 2 in approximation ratio [MPS99].

THEOREM 4.1. *The following results hold for the above DRTILE problem:*

(a) *There is an $O(n^5)$ time algorithm to find a tiling with maximum weight of a tile being at most δ using no more than twice as many tiles as needed.*

(b) *There is an $O(n^{2+\epsilon})$ time algorithm that uses at most $2p^*O(1/\epsilon^2)$ tiles, where p^* is the minimum number of tiles needed and $\epsilon > 0$ is arbitrarily small.*

Proof. Proofs of both (a) and (b) depend on the following argument. Assume that the optimum tiling has p^* tiles. Theorem 2.2 implies that there is a BSP tree for those tiles of size at most $2p^*$. Since the weight of a subtile of a tile is no more than that of the tile itself,[8] this means that there exists a BSP of the tiling with at most $2p^*$ tiles in which the weight of each tile at most δ. We can find a BSP of *minimum* size among BSPs of all possible tiling of the array A in which the weight of each tile at most δ in $O(n^5)$ time by dynamic programming [KMP98, MPS99]. This approximates p^* by a factor of 2 and proves (a). For part (b), we use the same reasoning as before but apply the sparse and rounded dynamic programming techniques in [MPS99] to reduce the running time at the expense of increasing the size of the computed BSP. □

Remark 1. Our lower bound result in Theorem 3.1(a) show that alternate approach is needed in order to get significantly better approximations for this problem.

Remark 2. The improvements described in Theorem 4.1 also hold for other tiling problems such as with either the maximum or the sum of weights of tiles, different weight functions, etc. The claims of Theorem 4.1 also apply to the special case (the DRTILE problem considered in [BDMR01, KMP98] and elsewhere) when the weight of a tile is the sum of all array elements in it. Many results exist for the DRTILE problem which use the special properties of the weight function and obtain better bounds than the ones we have quoted above. However, the strength of our results here is that they apply to many more complex weight functions, sum-squared-error being one example. The technical condition for the results in Theorem 4.1 to hold is that the weight functions have to be *superadditive*. Informally, this means that splitting the tiles in any optimum solution does not make it worse in the criteria. (A formal definition can be found in [MPS99].)

5. Concluding remarks. We have shown upper and lower bounds on the size of a BSP tree for a set of n isothetic rectangles. In addition, our results give improved approximation algorithms for rectangular tiling problems that arise in many application areas [BDMR01, MPS99]. We leave open the problem of closing the gap between the upper and lower bounds and of developing such detailed analyses for BSP trees for other objects.

[7] For specific values of ϵ, e.g., $\epsilon = \frac{3}{4}$, one can calculate the approximation ratio as well as the running time from [MPS99], and it is quite reasonable. However, for arbitrarily small but fixed ϵ, the bound appears to be substantially large. Hence, we retain just the expression $O(1/\epsilon^2)$ in the description.

[8] This is because the weight function is superadditive; see also the end of Remark 2.

Acknowledgments. We would like to thank the anonymous reviewers for helpful suggestions, which led to a substantially improved presentation of the results reported in this paper, and Prof. Ravi Janardan for help in the $O(n \log n)$ time implementation of the algorithm in Theorem 2.1.

REFERENCES

[AG+96] P. Agarwal, E. Grove, T. Murali, and J. Vitter, *Binary space partitions for fat rectangles,* in Proceedings of the 37th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1996, pp. 482–491.

[AS94] P. Agarwal and S. Suri, *Surface approximation and geometric partitions,* in Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, ACM, New York, 1994, pp. 24–33.

[B93] C. Ballieux, *Motion Planning Using Binary Space Partitions,* Technical report, Utrecht University, Utrecht, The Netherlands, 1993.

[BDMR01] P. Berman, B. DasGupta, S. Muthukrishnan, and S. Ramaswami, *Improved approximation algorithms for rectangle tiling and packing,* in Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms, ACM, New York, 2001, pp. 427–436.

[CF89] S. Chin and S. Feiner, *Near real-time shadow generation using BSP trees,* Comput. Graphics, 23 (1989), pp. 99–106.

[dAF92] F. d'Amore and P. G. Franciosa, *On the optimal binary plane partition for sets of isothetic rectangles,* Inform. Process. Lett., 44 (1992), pp. 255–259.

[BW80] J. L. Bentley and D. Wood, *An optimal worst-case algorithm for reporting intersections of rectangles,* IEEE Trans. Comput., 29 (1980), pp. 571–577.

[deGO] M. de Berg, M. de Groot, and M. Overmars, *New results on binary space partitions in the plane,* in Proceedings of the 4th Scandinavian Workshop on Algorithm Theory, Lecture Notes in Comput. Sci. 824, Springer-Verlag, Berlin, 1994, pp. 61–72.

[DMM01] A. Dumitrescu, J. Mitchell, and M. Sharir, *Binary space partitions for axis-parallel segments, rectangles and hyperrectangles,* in Proceedings of the 17th ACM Symposium on Computational Geometry, 2001, pp. 141–150.

[D94] S. E. Dorward, *A survey of object-space hidden surface removal,* Internat. J. Comput. Geom. Appl., 4 (1994), pp. 325–362.

[FKN80] H. Fuchs, Z. Kedem, and B. Naylor, *On visible surface generation by a priori tree structures,* Comput. Graph., 14 (1980), pp. 124–133.

[NW95] V. Hai Nguyen and P. Widmayer, *Binary Space Partitions for Sets of Hyperrectangles,* Lecture Notes in Comput. Sci. 1023, Springer-Verlag, Berlin, 1995.

[KMP98] S. Khanna, S. Muthukrishnan, and M. Paterson, *Approximating rectangle tiling and packing,* in Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, ACM, New YOrk, 1998, pp. 384–393.

[M85] E. M. McCreight, *Priority search trees,* SIAM J. Comput., 14 (1985), pp. 257–276.

[M90] J. Mitchell, *On maximum flows in polyhedral domains,* J. Comput. System Sci., 40 (1990), pp. 88–123.

[MPS99] S. Muthukrishnan, V. Poosala, and T. Suel, *Rectangular partitionings: Algorithms, complexity and applications,* in Proceedings of the Seventh International Conference on Database Theory, Lecture Notes in Comput. Sci. 1540, Springer-Verlag, Berlin, 1999, pp. 236–256.

[NT86] B. Naylor and W. Thibault, *Application of BSP Trees to Ray-Tracing and CSG Evaluation,* Technical report GIT-ICS 86/03, Georgia Tech, Atlanta, GA, 1986.

[PY90] M. Paterson and F. Yao, *Efficient binary space partitions for hidden-surface removal and solid modeling,* Discrete Comput. Geom., 5 (1990), pp. 485–503.

[PY92] M. Paterson and F. Yao, *Optimal binary space partitions for orthogonal objects,* J. Algorithms, 13 (1992), pp. 99–113.

[S90] H. Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing and GIS*, Addison-Wesley, Reading, MA, 1990.

[T92] S. Teller, *Visibility Computations in Densely Occluded Polyhedral Environments,* Ph.D. Thesis, Department of Computer Science, University of California, Berkeley, Berkeley, CA, 1992.

[TN87] W. Thibault and B. Naylor, *Set operations on polyhedra using binary space partitioning trees,* Comput. Graphics, 21 (1987), pp. 153–162.

[T01] C. D. Tóth, *A note on binary plane partitions,* in Proceedings of the 17th ACM Symposium on Computational Geometry, 2001, pp. 151–156.

[van90] P. van Oosterom, *A modified binary space partition for geographic information systems,* Int. J. GIS, 4 (1990), pp. 133–146.