FINDING SETS COVERING A POINT WITH APPLICATION TO MESHFREE GALERKIN METHODS*

XIAOXU HAN†, SUELY OLIVEIRA‡, AND DAVID STEWART§

Abstract. For a collection of sets in $\mathbf{R}^d$ we consider the task of finding all sets in the collection that cover or contain a given point. The algorithms introduced in this paper are based on quadtrees and their generalizations to $\mathbf{R}^d$. The advantages of our new splitting algorithm to find the covering sets of a point over the basic algorithm are detailed by means of hit rates and the expected depth traversed in the quadtree search, numerically and theoretically. This solves a difficult problem faced by meshfree discretizations of partial differential equations.

Key words. Quadtree and octree search methods, computational geometry.

AMS subject classifications. Primary: 68U05, 65D18; Secondary: 65N30

1. Introduction.

The task of this paper is, given a collection of support sets $S_i \subset \mathbf{R}^d$, $i = 1, 2, \ldots, N$, to efficiently answer the query "*Given a point $x \in \mathbf{R}^d$, find all S_i such that $x \in S_i$.*" This task arises when assembling the matrix for a meshfree discretization of a partial differential equation. The sets S_i are the support sets of the basis functions used to solve the partial differential equation. They are typically rectangles with modest aspect ratios, or circles/spheres in $\mathbf{R}^d$ with d being two, three, or four.

One way to answer this query quickly, is to organize the S_i using quadtrees and octrees [5, 9, 13]. Clearly the time needed to answer this query is at least $\Omega(\#\{\, i \mid x \in S_i \,\})$ where $\#A$ is the cardinality of the set A. The quantity $\#\{\, i \mid x \in S_i \,\}$ is the covering number of x; it is the number of support sets containing (that is, covering) the point x. Quadtrees and octrees have been found useful in geometric tasks such as finding which points from a collection belong to a given *range* [3]. For a task involving preprocessing a collection of points, quadtrees and octrees can be considered as *tries,* such as are used for string matching [2]. However, the task studied in this paper involves preprocessing a collection of *sets* $\{\, S_i \subset \mathbf{R}^d \mid i = 1, 2, \ldots, N \,\}$, rather than a collection of points. Also note that this task is not the point location problem previously studied [11, 12], in that here the sets S_i are usually not disjoint. Our analysis is also different to that of [11, 12] in that here we are concerned with the expected time for a random query point x, rather than with worst case analysis. While the data structures described here are those of Samet [15], Samet considers their use only for the intersection problem: find all pairs (i, j) where $S_i \cap S_j \neq \emptyset$. Here we wish, given x, to find all i where $x \in S_i$.

For each point x and support set S_i, we assume that we can test if $x \in S_i$ in $\Theta(1)$ time. The point about using quadtrees/octrees or other similar data structures, is that we can avoid doing unnecessary tests. The total time to answer a query can be roughly divided up into the time needed to traverse the data structure plus the time to

---

*This resarch has been supported by NSF/DARPA grant DMS-9874015.

†Applied Mathematics and Computational Science Program, The University of Iowa, Iowa City, IA 52242, USA.

‡Department of Computer Science and Depatment of Mathematics, The University of Iowa, Iowa City, IA 52242, USA. Phone: +1(319)335-0731. Fax: +1(319)335-3624. E-mail: oliveira@cs.uiowa.edu

§Department of Mathematics, The University of Iowa, Iowa City, IA 52242, USA. Phone: +1(319)335-3832. Fax: +1(319)335-0627. E-mail: dstewart@math.uiowa.edu

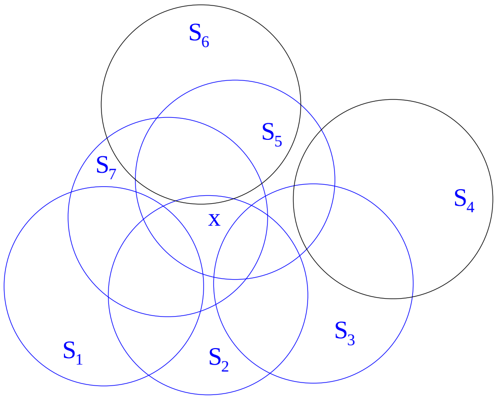

FIG. 1.1. *Example of distribution of support sets S_i*

perform the tests to identify the support sets containing x. Since the best we can do, is to have a "hit" (i.e., $x \in S_i$) whenever we test for "$x \in S_i$", we are interested in the "hit rate", which is the ratio of hits to the total number of tests, averaged over all the integration points. In particular, we don't want this to go to zero; and if the hit rate can be bounded away from zero, then the number of tests is, at worst, proportional to the average covering number of the sets $\{ S_i \mid i = 1, 2, \ldots, N \}$. We will show that the average time to traverse the quadtree is no worse than $O((\log N)/d)$ for both algorithms that we will present, so that the traversal time is a more easily controlled cost.

Worst-case deterministic bounds for the hit rate can be arbitrarily bad: the hit rate is zero if x is not covered by any support set and at least one test is done for x. If there are no restrictions on the shape of the support sets then quadtrees/octrees can be made to perform badly, for example, by making the S_i's curves or thin ribbons. In practice, the support sets S_i are discs, balls, rectangles, or rectangular solids, or similar shapes, with modest aspect ratios (the ratio between the longest and smallest dimension).

Our bounds on the hit rate are proven under the assumption that x is chosen randomly and uniformly over a set S which contains the union of the S_i's.

1.1. Application. Various new methods for solving partial differential equations have been introduced which do not require construction of grids or triangulations of regions in order to compute approximate solutions. These are known as meshfree methods; there are a number of types of meshfree methods such as *Partition of Unity Finite Element Methods* (PUFEM), *Reproducing Kernel Particle Methods* (RKPM), *Element Free Galerkin* (EFG) methods and *Moving Least Squares* (MLS) methods. An overview of meshfree methods in general can be found in [4].

All of these methods construct basis functions Ψ_i, $i = 1, 2, \ldots, N$ over the domain of interest $\Omega \subset \mathbf{R}^d$ from which a Galerkin discretization is developed. Thus, for example, for solving the partial differential equation

$$\begin{aligned} -\nabla^2 u(x) + u(x) &= f(x), \quad x \in \Omega, \\ u(x) &= 0, \quad x \in \partial\Omega, \end{aligned}$$

where $\partial\Omega$ is the boundary of Ω, we approximate $u \approx u^h$ where $u^h(x) = \sum_{i=1}^N u_i^h \Psi_i(x)$. Then we construct a linear system of equations for the unknown coefficients u_i^h:

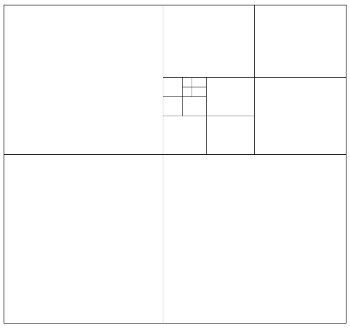
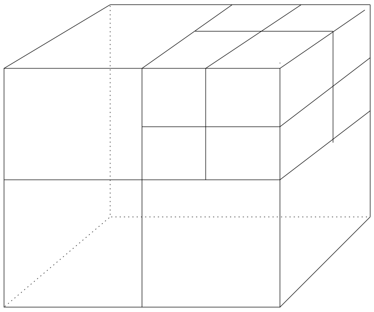

FIG. 1.2. *Quadtree and octree examples*

$\sum_{j=1}^{N} K_{ij} u_j^h = f_i$ for $i = 1, 2, \ldots, N$ where

$$K_{ij} = \int_\Omega \frac{1}{2} \left(\nabla\Psi_i(x)^T \nabla\Psi_j(x) + \Psi_i(x)\Psi_j(x) \right) \, dV(x).$$

For most meshfree methods, there are either no analytical formulas available for $\Psi_i(x)$, or they are too complex to compute these integrals directly. Therefore numerical approximations of the form

$$K_{ij} \approx \sum_{k=1}^{P} \frac{1}{2} \left(\nabla\Psi_i(x_k)^T \nabla\Psi_j(x_k) + \Psi_i(x_k)\Psi_j(x_k) \right) \omega_k$$

are used to construct the linear system. In our notation, P is the number of integration points. Let $S_i = \operatorname{supp} \Psi_i = \overline{\{ z \mid \Psi_i(z) \neq 0 \}}$, the support of Ψ_i. Note that Ψ_i and all of its derivatives are zero outside its support S_i. The integration points x_k are scattered essentially uniformly throughout the region Ω. Rather than first looping over i and j where $S_i \cap S_j \neq \emptyset$, and then finding all k such that $x_k \in S_i \cap S_j$, we can compute the matrix K more efficiently by looping first over k, and then finding all i where $x_k \in S_i$ by using quadtree/octree type data structures. Note that the former strategy leads to both an intersection finding problem and a range searching problem. While finding all i and j where $S_i \cap S_j \neq \emptyset$ can be solved using quadtree structures (see for example, Samet [15, pp. 199-225]), the costs are at least as high as for the task discussed here. The problem of finding all x_k where $x_k \in S_i \cap S_j$ is a range searching problem which requires construction of a range-searching tree on the integration points. Apart from the fact that the number of integration points is usually much larger than the number of support sets in practice, these structures are complex and the searching time complexity is $O(\log^{d-1} P + \#\{ k \mid x_k \in S_i \cap S_j \})$ in d dimensions where P is the number of integration points. By contrast, the latter strategy can be accomplished with methods which have an expected time complexity of just $O(\log N + \mathbf{E}(\#\{ i \mid x_k \in S_i \}))$ in any fixed number of dimensions.

1.2. Quadtrees and Octrees. Quadtrees and octrees are members of hierarchical decomposition data structures based on the principle of recursive decomposition of space.They are widely used in image compression [14, 13], GIS (Geographic Information Systems) [16, 6], computer vision [7], and computer graphics [8, 17]. Quadtrees can be used to represent a two-dimensional space. Similarly, octrees can be used to deal with the three-dimensional case. A quadtree is a tree data structure based on a rectangle with four children and an octree is a tree data structure based on a cube with eight children. However, both of them can be used in the generation of meshes for finite element analysis. Due to different applications, there are variants on this structure, such as region-based quadtrees, and point quadtrees.

A quadtree can be described in this way using Java, for example:

```
class QtreeNode{
   private Object data;
   private QtreeNode children[2][2];
   private QtreeNode parent;
   private int depth;
   private int coords[2];
     ...
   }
```

An octree tree can be described in a similar way using Java

```
class OctreeNode{
   private Object data;
   private OctreeNode parent;
   private OctreeNode children[2][2][2];
   private int depth;
   private int coords[3];
     ...
   }
```

We use the following notation and terminology to describe various aspects of quadtrees, octrees and their d-dimensional analogues.

- A *support set* S_i is simply a bounded subset of $\mathbf{R}^d$; the quadtree or octree construction algorithm is , given a collection $\{S_i \mid i = 1, 2, \ldots, N\}$ of these sets, to build a corresponding quadtree or octree representation of these support sets S_i.
- A *rectangle* is a set of the form $[a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$ in $\mathbf{R}^d$; a *rectangle in a quadtree or octree* has the special property that $a_i = \gamma_i 2^{-k}$, $b_i = (\gamma_i + 1)2^{-k}$ for all i with k independent of i, and γ_i an integer. A *subrectangle* of a rectangle R in a quadtree or octree is a rectangle in the quadtree or octree contained in the rectangle R.
- We use *quadtree* to refer to quadtrees and their generalizations to 2^d-trees in $\mathbf{R}^d$. The term thus includes octrees in three dimensions and binary trees in one dimension.
- The *width* of a set A is the diameter of the set in the infinity norm; that is, $\mathrm{width}(A) = \sup\{\, \|x - y\|_\infty \mid x, y \in A\,\} = \sup_{x,y \in A} \max_i |x_i - y_i|$.
- The *top* rectangle is the rectangle at the root of the quadtree or octree. It is usually considered to be the rectangle $[0, 1] \times [0, 1] \times \cdots \times [0, 1]$.
- The *expected value* of a random variable X is denoted $\mathbf{E}(X)$.
- The *d-dimensional volume*, or *d-volume* of a set S is denoted $\mathrm{vol}_d(S)$.

Each rectangle in the quadtree has an associated linked list containing references to support sets S_i; if R is a rectangle in the quadtree whose linked list contains a reference to S_i, we say that S_i *is linked to* R, or that R *links* S_i.

2. Algorithms. In this Section we present two quadtree search algorithms to find the number of support sets covering each point x. In our application, x is an integration point and is part of a set of integration points x_k that are approximately

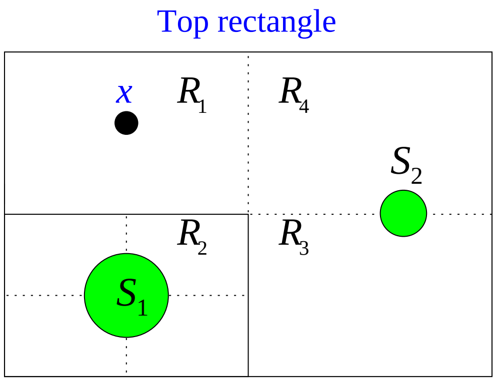

FIG. 2.1. *Example of the basic quadtree*

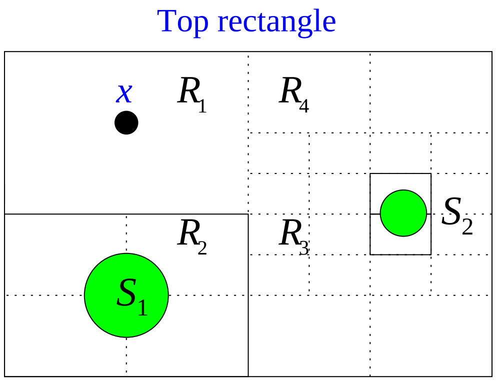

FIG. 2.2. *Example of the expanded quadtree*

uniformly distributed in a domain Ω. The first algorithm for building quadtree structures is the called the *basic algorithm*, and the second is called the *splitting algorithm*. These data structures constructed are discussed in Samet [15], where the former is called the MX-CIF quadtree and the latter the expanded MX-CIF quadtree. In this paper these will be referred to as the basic and expanded quadtrees, respectively. However, neither data structure was considered in Samet [15] in the context of identifying sets containing a given query point. The splitting algorithm has advantages over the basic algorithm in finding the number of support sets covering the integration point x. The total expected query time for randomly selected points x can be bounded in terms of the *hit rate* and the *expected depth traversed* in the quadtree. We will provide detailed theoretical analysis and numerical simulation in terms of these quantities later. We illustrate the basic and expanded quadtrees in Figures 2.1 and 2.2.

In both the basic algorithm and the splitting algorithm, each rectangle in the quadtree has a corresponding linked list. In the basic approach we try to link each S_i to a rectangle R in the quadtree if it is contained in R and there is no subrectangle of R in which S_i is completely contained. Note that S_i is not necessarily linked to a leaf node of the quadtree, even at the time of linking. Also note that there can be more than one support set linked to a single rectangle R, creating a linked list. In the interests of efficiency, we usually want these linked lists to be short. When using this

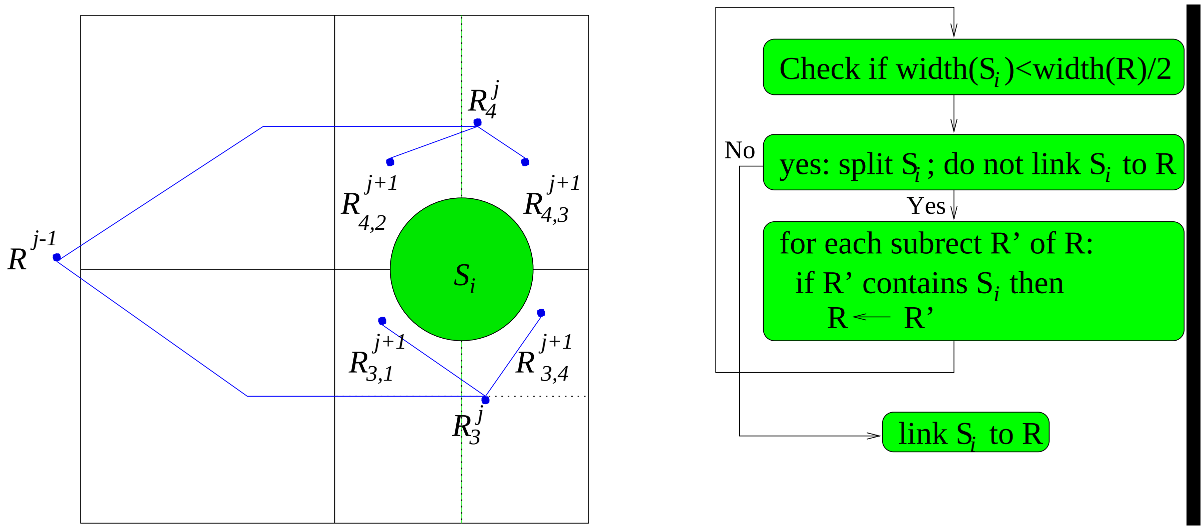

FIG. 2.3. *Example for describing the splitting algorithm*

algorithm, in the worst case, the construction of the quadtree takes $O(N \log(1/\varepsilon))$ time where $\varepsilon = \min_i \operatorname{diam} S_i$.

The basic algorithm constructs a quadtree which is the MX-CIF quadtree discussed by Samet [15, pp. 200–213] and the data structure discussed by Abel and Smith [1]. Unlike Samet, however, we are interested here in finding sets S_i containing query points, rather than intersection detection.

Pseudo-code for building the basic quadtree (basic algorithm)

```
void basic(S,top,N)
/* Constructing a quadtree for support sets S_i. */
for i = 1,2,...,N
    R ← top
    repeat
        for each subrectangle R' of R
            if S_i ⊆ R'
              R ← R' & break for loop
            end if
        end for
    until (no subrectangle of R contains S_i)
    link S_i to the rectangle R
end for
```

For example, in figure 2.1, S_1 is not linked to the top rectangle R but it is linked to the subrectangle R_2; since S_2 is not contained in any of the sub-rectangles of R, it is linked to R.

In our splitting algorithm (our second method), if a support set S_i is a subset of a rectangle R, but is not a subset of any sub-rectangle of R in the quadtree and if $\operatorname{diam}(S_i) < \frac{1}{2}\operatorname{width}(R)$, then we don't link S_i to R; rather we "split" S_i: for each subrectangle R' of R, if $S_i \cap R' \neq \emptyset$ and recursively call the splitting algorithm with $S_i \cap R'$. We give the pseudo-code for the splitting algorithm below.

Note that the splitting algorithm builds a recursive extension of the MX-CIF tree of Samet which is the expanded MX-CIF tree of Samet [15, pp. 213–215].

Pseudo-code for building the expanded quadtree (splitting algorithm)

procedure splitting(S_i,R)
/* Add support set S_i to quadtree
  using R as the root of the quadtree */
loop
  if R' is a subrectangle of R where $S_i \subset R'$
    $R \leftarrow R'$
  else if $\mathrm{diam}(S_i) < \frac{1}{2}\mathrm{width}\ (R)$
    for each subrectangle R' of R where $R' \cap S_i \neq \emptyset$ (allocate R' if necessary)
      call splitting($S_i \cap R'$,R')
  else
    link S_i to R and exit loop
end loop

The splitting algorithm is illustrated by Figure 2.3. The subscripts in this figure refer to the subrectangle number, being 1 (northwest), 2 (southwest), 3 (southeast), or 4 (northeast). In this figure $R^{j+p}_{k,l,\ldots}$ denotes the rectangle at level $j+p$ where $k, l, \ldots$ denotes the path from R^{j-1} to this rectangle within the quadtree.

To construct the complete quadtree, we apply `splitting` to all support sets, i.e.

for $i = 1, 2, \ldots, N$
  call splitting(S_i,top)

Note that in the expanded quadtree, each S_i can be linked to more than one rectangle in the quadtree. In fact, each S_i can be linked to at most 2^d rectangles. As given, the time and space requirements for the splitting algorithm are $O(2^d N\, h)$ where h is the depth (or height) of the expanded quadtree. Since splitting can produce very small sets in the computation of $R' \cap S_i$, the worst-case space and time requirements are unbounded. However, if *path compression* [2] is used so that rectangles with only one allocated subrectangle and no S_i linked to it, are removed from the quadtree, the worst-case space requirements reduce to $O(2^d N)$. Furthermore, the depth of the path compressed quadtree is bounded above by $c_{\max} + \lceil \log_2 N \rceil$, where $c_{\max} = \max_x \#\{\, i \mid x \in S_i \,\}$ is the maximum covering number. In applications, the maximum covering number is bounded, or grows polylogarithmically in N.

The path-compressed quadtree can be built incrementally using a number of different techniques. A path-compressed version of the basic quadtree can start by finding the depth and co-ordinates of the smallest enclosing quadtree rectangle. For each S_i we find a bounding box $[a_1, b_1] \times \cdots \times [a_d, b_d] \subseteq [0, 1]^d$, and expand each coordinate a_i and b_i in binary. The number of leading bits of a_i and b_i that match is the depth of the smallest enclosing quadtree rectangle, and the matching bits are the (integer) coordinates of that rectangle. Using the Java data structures above, the current path-compressed quadtree can be traversed downwards until a rectangle is found at the correct depth, or the correct place to insert a new such rectangle can be found. Once this is found, standard techniques can insert a suitable rectangle in $O(1)$ time. A similar approach can be used to find the $\leq 2^d$ rectangles which are linked to a support set for the splitting algorithm by first finding matching bits. This makes the time to construct the quadtree for the splitting algorithm with path compression $O(2^d N(c_{\max} + \log N + d\, \log\log(1/\varepsilon))$ where $\varepsilon = \min_i \mathrm{diam}(S_i)$ assuming $O(1)$ time bit operations (and, or, invert, and shift) on integers. Alternatives to pointer-based data structures include balanced trees and hash-tables using the tuple (depth,coord_1,...,coord_d) as the key, where coord_i is the ith co-ordinate of the rectangle.

Alternatively, if in the splitting algorithm the set $R' \cap S_i$ is expanded to have diameter no less than $0 < \eta \leq 1/2$ times the diameter of S_i, the maximum depth of the expanded quadtree is no more than $\log_2(1/\varepsilon) + \log_2(1/\eta)$. Choosing a particular value (e.g., $\eta = 1/4$) gives an upper bound on the depth of the expanded quadtree generated by the modified splitting algorithm with the same asymptotic behavior. Furthermore, the performance bounds for the search given below remain unaltered for the modified splitting algorithm.

Once the quadtree is constructed, we need to use it to find the support sets covering a given point x. Below is our pseudo-code for doing this with a quadtree constructed using either the basic or splitting algorithms.

```
list function search(x,top)
/* Given x, find all S_i that cover x: x ∈ S_i */
R ← top
while R ≠ null
    for each S_i linked to R
    if x ∈ S_i then
        add S_i to list of covering support sets
        increment covering number
    R ←allocated subrectangle of R containing x, if any
    R ← null if there is no such subrectangle
end while
```

In Figure 2.1, we show the behavior of the basic algorithm. We need to make a test to determine if x is in S_2 or not because S_2 is linked to the top rectangle, though x is far from S_2.

In Figure 2.2, the advantages of the splitting algorithm are exemplified. We will test if a given $x \in S_2$ only if it is in the two smallest rectangles shown containing S_2.

2.1. The advantage of the expanded quadtree over the basic quadtree. Just as we discussed above, in the basic quadtree there are many unnecessary tests to obtain the sets covering each integration point. Here, we give the plot of the number of wasted tests (unnecessary tests) per integration point in two algorithms in Figure 2.4. Here, we randomly generate 2000 support sets (circles) with radius = 0.2 and use 10000 uniformly distributed integration points.

We observe that the number of wasted tests in the basic algorithm is 15–20 times the number in the expanded quadtree.

Figure 2.5 gives plots of the lengths of the linked lists in the basic and expanded quadtrees. We can see, for the chosen 200 support sets, that the linked lists are generally much shorter for the expanded quadtree, so that fewer tests are needed for each rectangle. Furthermore, the few rectangles with long linked lists in the basic quadtree are the top rectangles, and the rectangles near the top. All support sets linked to the top rectangle are tested for *any* integration point, making this data structure much less efficient.

2.2. The hit rate. The hit rate is the ratio between the average number of sets covering integration point x and the average number of tests for finding all covering sets for all integration points x, where x is a point randomly chosen from a uniform distribution of points over the top rectangle.

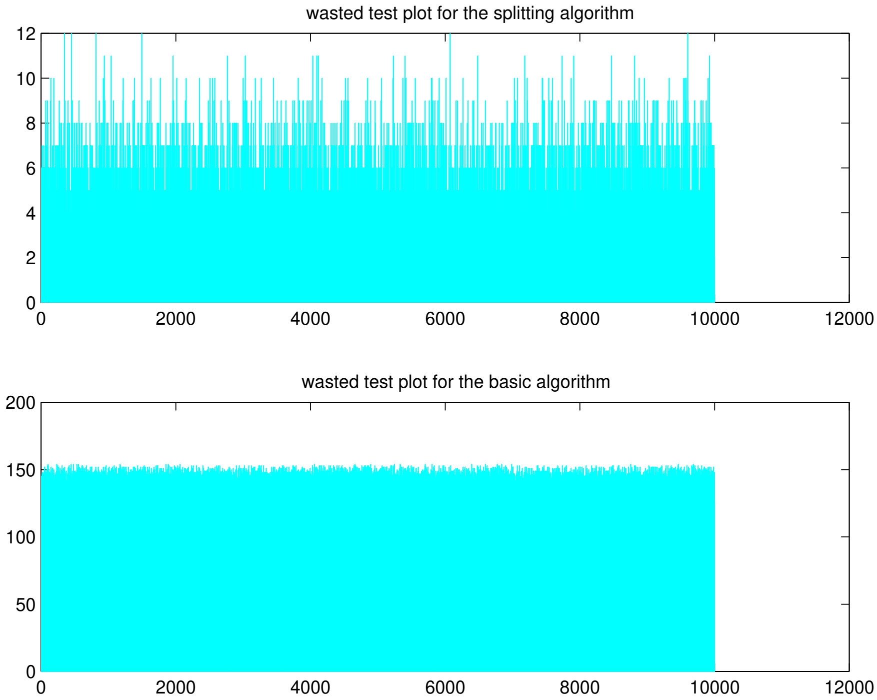

Fig. 2.4. *The number of wasted tests in the basic and expanded quadtrees*

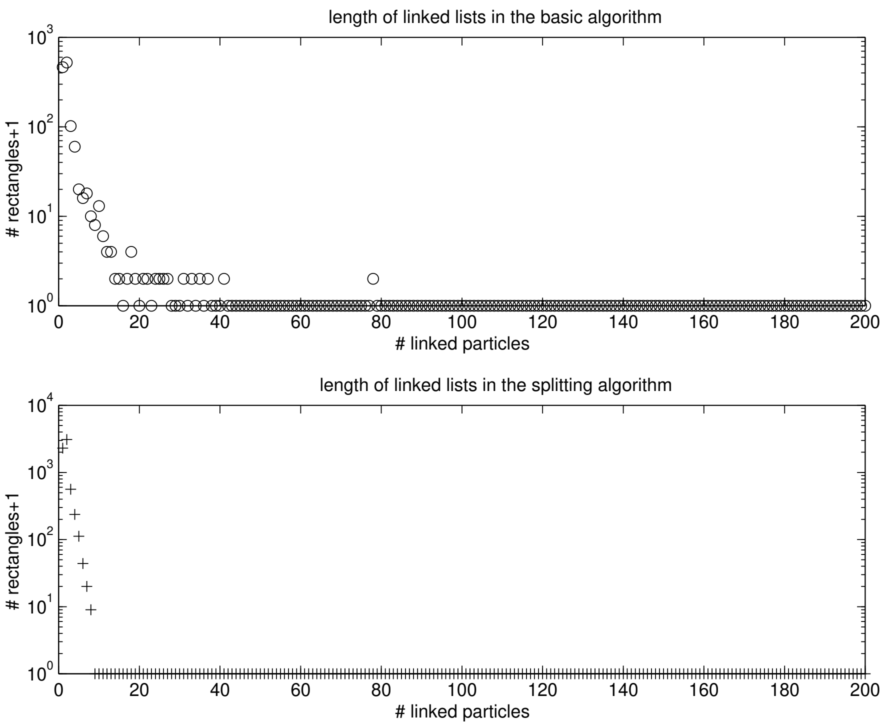

Fig. 2.5. *The length of linked lists in the basic and expanded quadtrees*

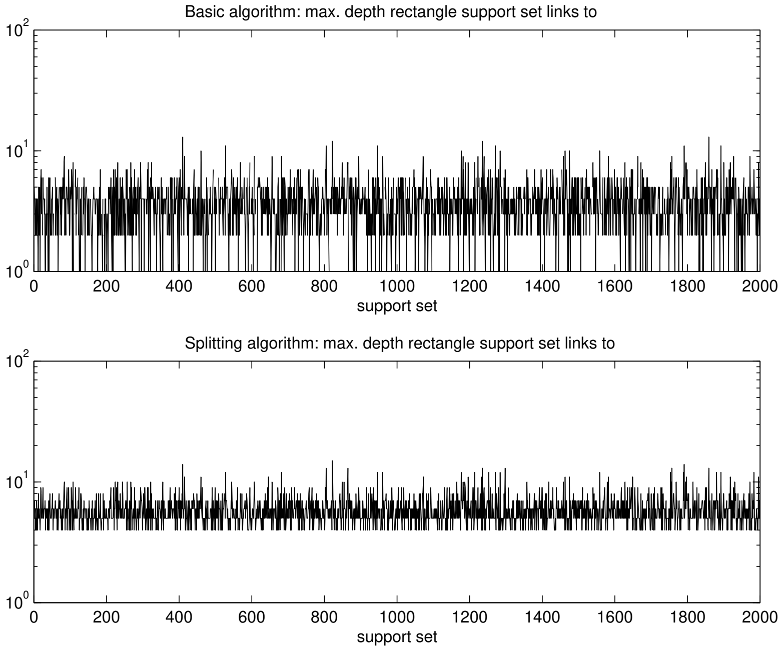

FIG. 2.6. *Depth of rectangle linked to each support set*

We give a more precise definition of the hit rate:

$$\text{hit rate} = \frac{\mathbf{E}(covering\#)}{\mathbf{E}(\#tests)}$$

Here *covering#* is the number of sets covering the randomly chosen point x, while *#tests* is the number of tests needed in the algorithms to determine the list of sets covering x. That is, hit rate is the ratio between the mathematical expectation of the covering (support sets) numbers of the integration points and the expectation of the number of tests to get the coverings for the randomly chosen and uniformly distributed integration points.

3. Analysis of the expected search time. We randomly choose integration points x and go down through the quadtree to find the sets S_i which contains x in our integration region. The total search time will depend on how many tests we will do for the integration points we use, and how deeply we traverse the tree structure. So our expected search time is:

$$\mathbf{E}(\text{search time}) = O(\mathbf{E}(depth) + \mathbf{E}(\#tests)),$$

where *depth* is the depth traversed in the quadtree when searching for sets covering a given point x, and *#tests* is the number of tests $x \in S_i$ that are performed during the search. Since $\mathbf{E}(\#tests) = (\text{hit rate}) \times \mathbf{E}(covering\#)$, provided the hit rate is bounded away from zero, we get $O(\mathbf{E}(depth) + \mathbf{E}(covering\#))$ expected search time per query point.

We will show that, for both the splitting and basic algorithms, the expected depth is bounded above by $(\log N)/d$. *i.e.* $\mathbf{E}(depth) = O((\log N)/d)$. For the basic quadtree, the hit rate can go to zero. In fact, for the basic quadtree the hit rate is $O(N^{-1/2})$ in two dimensions in practice. The expanded quadtree, however, has a hit rate that is bounded away from zero independently of N, although it can decrease exponentially as the dimension d increases. In the following sections we will analyze both the hit rate and the expected depth traversed in the quadtree.

3.1. Analysis of the hit rate. We assume bounds on the *aspect ratio*, which is the ratio of the longest to the shortest side. First, we give the lower and upper bounds of $\mathbf{E}(covering\#)$, then we get the inequality of reflecting relationship between $\mathbf{E}(covering\#)$ and $\mathbf{E}(\#tests)$. We combine these to bound the hit rate.

3.1.1. The hit rate for the basic algorithm. First, we note that the *hit rate* can be arbitrarily bad for the basic quadtree. In two dimensions, if every support set S_i was chosen to intersect either the horizontal or vertical lines bisecting the *top* rectangle, then every support set would be linked to the *top* rectangle of the quadtree. That would mean that all the support sets would have to be tested for all integration points $x \in top$. This would give a hit rate of $\mathbf{E}(covering\#)/N$, which can be arbitrarily close to zero, and goes to zero if the average covering number is bounded as $N \to \infty$.

Even for uniform or random distributions of the sets S_i, if $\mathrm{diam}(S_i)$ are all near $\varepsilon > 0$, the number of sets linked to the *top* rectangle is $\Theta(N\varepsilon)$; if $\varepsilon = \Omega(N^{-1/d})$ as we would expect if the S_i's covered *top*, then the number of sets linked to the *top* rectangle would be $\Omega(N^{1-(1/d)})$. While this is an improvement over the naive approach of directly testing a point x against all sets S_i, it is far from optimal. This confirms the the $O(\sqrt{N})$ query time found in [1] for planar queries.

3.1.2. Using the number of links to express $\mathbf{E}(\#test)$ for the expanded quadtree. We work in d dimensions ($d = 2$ for quadtrees, $d = 3$ for octrees). Let $N_j = \#$links at level j. Also, let $\widehat{N}_j$ be the number of rectangles which have at least one support set linked at level j. Note that $\widehat{N}_j \le N_j$, and every support set has at most 2^d links to it. So, for all support sets N for the integration point x , we have

$$\sum_{j=0}^{\infty} N_j \le 2^d N.$$

Every rectangle at level j has d-volume 2^{-jd} times the volume of the *top* rectangle and each link from a rectangle requires a test for any point contained in that rectangle. Suppose that the rectangles at level j which are linked to a support set are $R_{j,1}, R_{j,2}, \ldots, R_{j,\widehat{N}_j}$. Since there are N_j links at level j, then the expected number of links to search from level j is:

$$\begin{aligned}\mathbf{E}(\text{tests at level } j) &= \sum_{k=1}^{\widehat{N}_j} Pr(x \in R_{j,k}) \times (\#\text{support sets linked to } R_{j,k}) \\ &= \sum_{k=1}^{\widehat{N}_j} 2^{-jd}(\#\text{support sets linked to } R_{j,k}) = 2^{-jd} N_j.\end{aligned}$$

So, $\mathbf{E}(\#tests)$ is the sum of the expected number of links in all levels: $\mathbf{E}(\#tests) = \sum_{j=0}^{\infty} 2^{-jd} N_j$.

3.1.3. Bounds on $\mathbf{E}(covering\#)$ for the expanded quadtree. Adding total volumes of all support sets S_i, $i = 1, 2, ..., N$ in d dimensions, we get the value of $\mathbf{E}(covering\#) = \mathbf{E}(\#\{\, i \mid x \in S_i \,\})$. Let a be the maximum *aspect ratio (the ratio of the largest side to the shortest side for rectangles)* of the support sets. Assume that $\mathrm{vol}_d(top) = 1$. Then,

$$\begin{aligned}\mathbf{E}(covering\#) &= \sum_{i=1}^{N} \mathrm{vol}_d(S_i) \geq \sum_{i=1}^{N} \frac{1}{a^{d-1}} \mathrm{width}(S_i)^d \\ &= \frac{1}{a^{d-1}} \sum_{i=1}^{N} \mathrm{width}(S_i)^d\end{aligned}$$

We suppose that, at level j, S_i has n_{ij} links. Then the total links of S_i in the whole tree is less than 2^d:

$$\sum_{j=0}^{\infty} n_{ij} \leq 2^d \text{ for all } i.$$

On the other hand, the total number of links in level j for all supports of our integration points are: $N_j = \sum_i n_{ij}$. Remember that for the expanded quadtree a support set S_i is not linked at level j if the width of S_i is no more than half the width of a rectangle at level j. That is,

$$n_{ij} = 0 \text{ if } \mathrm{width}(S_i) \leq 2^{-j-1}\mathrm{width}(top).$$

Since in the expanded quadtree, the volume of S_i is split into disjoint parts, if we add all volumes in each linked rectangle for S_i, we have the following bound:

$$\mathrm{vol}_d(S_i) \leq \sum_{j=0}^{\infty} n_{ij} 2^{-jd} \mathrm{vol}_d(top).$$

Assuming $\mathrm{vol}_d(top) = 1$, we have

$$\mathrm{vol}_d(S_i) \leq \sum_{j=0}^{\infty} n_{ij} 2^{-jd}.$$

An upper bound for $\mathbf{E}(covering\#)$ is easily obtained:

$$\begin{aligned}\mathbf{E}(covering\#) &\leq \sum_{i=1}^{N} \sum_{j=0}^{\infty} n_{ij} 2^{-jd} = \sum_{j=0}^{\infty} (\sum_{i=1}^{N} n_{ij}) 2^{-jd} \\ &= \sum_{j=0}^{\infty} N_j 2^{-jd}.\end{aligned}$$

However, what we really need is a lower bound. For each set S_i let $j_i^* = \min\{\, j \mid S_i \text{ is linked to a rectangle } R \text{ at level } j \,\}$. For at least one rectangle R on level j_i^*, the aspect ratio of $S_i \cap R$ is no more than $2a$. Then $\mathrm{vol}_d(S_i) \geq \mathrm{vol}_d(R \cap S_i) \geq (2a)^{1-d}\mathrm{width}(R \cap S_i)^d \geq (2a)^{1-d} 2^{-d} \mathrm{width}(S_i)^d \geq 2a(4a)^{-d} 2^{-(j_i^*+1)d}$. Since $\sum_{j=0}^{\infty} n_{ij} \leq 2^d$ and $n_{ij} = 0$ for $j < j_i^*$, $\mathrm{vol}_d(S_i) \geq \sum_{j=0}^{\infty} 2a(4a)^{-d} 2^{-(j+1)d} \times 2^{-d} n_{ij}$.

Then,

$$\text{vol}_d(S_i) \geq 2a(16a)^{-d} \sum_{j=0}^{\infty} n_{ij} 2^{-jd}$$

So, we can get a lower bound of $\mathbf{E}(covering\#)$ as follows:

$$\begin{aligned}\mathbf{E}(covering\#) &= \sum_{i=1}^{N} \text{vol}_d(S_i) \geq 2a(16a)^{-d} \sum_{j=0}^{\infty} \sum_{i=1}^{N} n_{ij} 2^{-jd} \\ &= 2a(16a)^{-d} \sum_{j=0}^{\infty} N_j 2^{-jd} = 2a(16a)^{-d} \mathbf{E}(\#tests)\end{aligned}$$

Thus the hit rate for the expanded quadtree is bounded away from zero for fixed d.

3.1.4. The lower bound of the average hit rate for the expanded quadtree. By the definition of the hit rate, we have lower bound for average hit rate:

$$\text{hit rate} = \frac{\mathbf{E}(covering\#)}{\mathbf{E}(\#tests)} \geq 2a(16a)^{-d}$$

From here, we can see it is impossible to get zero hit rate in our quadtree search. That is to say, if each set S_i is a square in two dimensions, for each set S_i covering x, there will be an average of at most 128 tests using the splitting algorithm. However, this bound is not sharp: the hit rate is usually much larger, as can be seen in Figure 3.1 and in Table 3.1.

3.2. The simulation results on hit rate. From the point of view of the *hit rate*, we can also demonstrate that the advantage of the expanded over the basic quadtree. For 10000 randomly chosen and uniformly distributed integration points and 2000 support sets with radius 0.2, we get the following results. The average hit rate in the expanded quadtree is almost 30 times larger than that of in the basic case: *expanded quadtree: 0.1694; basic quadtree: 0.0057.* This result agrees with the result of the wasted tests counted in the two algorithms. Note that in this simulation, many of the support sets overlap, which can cause difficulties for other data structures.

Figure 3.1 shows the hit rate plots for the basic and expanded quadtrees.

Computational results were also obtained for actual meshfree methods. In particular, for a two-dimensional elasto-plastic problem computations were done using a number of different levels of refinement. The particular meshfree method used was the RKPM (Reproducing Kernel Particle Method) [10]. The basis functions were tensor products of B-splines (that is, $\Psi_i(x) = B_{i,1}(x_1)B_{i,2}(x_2)$ where $B_{i,1}$ and $B_{i,2}$ are cubic splines), so that the support sets were axis-aligned rectangles. The aspect ratios of the support sets were all similar, and not extreme. However, each support set was relatively large, resulting in a great deal of overlap; the average covering number is around 15. Table 3.1 shows the main results. Note that "$\#R$" is the number of rectangles allocated for the quadtree, N is the number of sets, P is the number of integration (i.e., query) points, and h is the depth (or height) of the quadtree. The naive algorithm tests each point against each support set. This example does not show the full power of the data structures since the problems are relatively small and two-dimensional. However, the difference between the basic and expanded quadtrees is clearly evident. The average covering numbers for the problems are shown

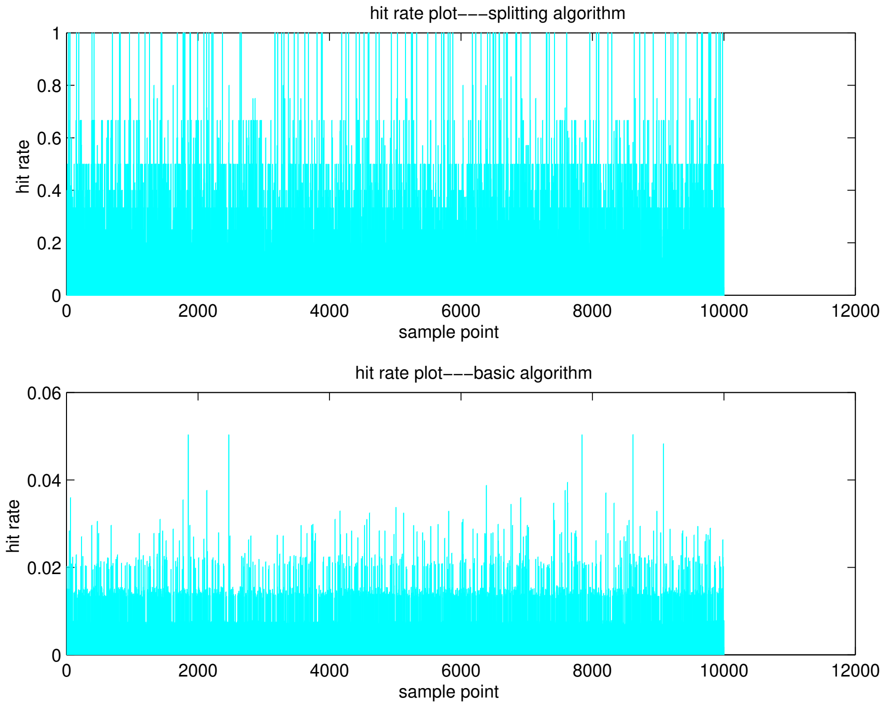

FIG. 3.1. *Hit rate plot in the basic and expanded quadtrees*

Data structure			Basic			Expanded			Naive
Problem	N	P	$\#R$	hit rate	h	$\#R$	hit rate	h	hit rate
coarse	71	557	4	0.2372	2	16	0.3904	5	0.1998
medium	152	1286	5	0.1443	2	42	0.3033	7	0.1058
fine	349	3139	12	0.0775	3	106	0.2781	7	0.0419
fine2	605	5582	20	0.0516	3	223	0.2442	10	0.0246
fine3	985	8262	20	0.0396	3	240	0.1937	10	0.0155

TABLE 3.1
Table showing quadtree statistics for meshfree problems.

in Table 3.2. As expected, the basic quadtree's hit rate is roughly proportional to $N^{-1/2}$, and the naive algorithm's hit rate is roughly proportional to N^{-1}, while the expanded quadtree's hit rate is relatively stable. There is clearly additional overhead in setting up the quadtree using the splitting algorithm, but the number of rectangles allocated is much less than the number of support sets, and the depth of the resulting quadtree grows only slowly, probably logarithmically, in N. Note that the implementation of the splitting algorithm did not use any modifications to avoid deep quadtrees. Even so, path compression would achieve little here since the number of rectangles not linked by support sets for the splitting algorithm were, for the various problems: coarse 2; medium 6; fine 9; fine2 16. Thus for the largest problem (fine2), path compression could save no more than 16 of the 223 allocated rectangles. Thus it appears that for our application, path compression will save little in space or time.

Problem	coarse	medium	fine	fine2	fine3
Avg. covering #	14.18	16.09	14.61	14.90	15.22

TABLE 3.2
Table showing average covering numbers.

3.3. The upper bounds of $\mathbf{E}(depth)$. As in the previous section, we let N_j denote the number of links at level j; $\widehat{N}_j$ denotes the number of rectangles linked to at least one support set in level j; and $\widetilde{N}_j$ denotes the number of allocated rectangles in the quadtree at level j. Clearly, $\widehat{N}_j \leq N_j$. Since every allocated rectangle must have a subrectangle that is linked to a support set, $\widetilde{N}_j \leq \sum_{k=j}^{\infty} \widehat{N}_k \leq \sum_{k=j}^{\infty} N_k$. On the other hand, $\widetilde{N}_j \leq 2^{dj}$ since there are at most 2^{dj} rectangles in a quadtree at level j.

Note from the previous section that $\sum_{j=0}^{\infty} N_j \leq N$ for the basic quadtree, but $\sum_{j=0}^{\infty} N_j \leq 2^d N$ in the expanded quadtree.

3.4. Analysis of $\mathbf{E}(depth)$. Recall that *depth* is the depth of the quadtree traversed while searching for the sets covering a given point x, chosen at random. The basis of these calculations are bounds on the probability of *depth* having a particular value. A detailed discussion on related issues can be found in [2].

$$
\begin{aligned}
\mathbf{E}(depth) &= \sum_{j=0}^{\infty} j \Pr(depth = j) \\
&= \sum_{j=0}^{\infty} j \left[\Pr(depth \geq j) - \Pr(depth \geq j+1)\right] \\
&= \sum_{j=1}^{\infty} \Pr(depth \geq j).
\end{aligned}
$$

The problem now is to bound $\Pr(depth \geq j)$. This is the probability that x lies in an allocated rectangle at a depth j. A simple bound is

$$\Pr(depth \geq j) = 2^{-dj} \widetilde{N}_j \leq 2^{-dj} \min(2^{dj}, 2^d N) = \min(1, 2^{-d(j-1)} N).$$

This leads to bounds of the form $\mathbf{E}(depth) = O(\log N)$:

$$
\begin{aligned}
\mathbf{E}(depth) &= \sum_{j=1}^{\infty} \Pr(depth \geq j) \\
&\leq \sum_{j=1}^{\infty} \min(1, 2^{-d(j-1)} N).
\end{aligned}
$$

Suppose that $2^{d(j^*-1)} < N \leq 2^{dj^*}$. Then for $j \geq j^*$, $2^{-d(j-1)} N \leq 2^{-d(j-j^*-1)} \leq 1$. Thus

$$
\begin{aligned}
\mathbf{E}(depth) &\leq \sum_{j=1}^{j^*} 1 + \sum_{j=j^*+1}^{\infty} 2^{-d(j-j^*-1)} \\
&= j^* + \sum_{l=0}^{\infty} 2^{-dl} = j^* + \frac{1}{1-2^{-d}} \leq j^* + 2.
\end{aligned}
$$

Note that $j^* = \lceil(\log_2 N)/d\rceil$, and so

$$\mathbf{E}(depth) \leq \left\lceil \frac{\log_2 N}{d} \right\rceil + 2 = O\left(\frac{\log_2 N}{d}\right).$$

4. Conclusions. The expected total search time using the expanded quadtree for a random point $x \in top$ is $O(\mathbf{E}(depth)) + O(\mathbf{E}(\#tests))$. Since the number of tests needed is the covering number divided by the hit rate (which is bounded below by $2a(16a)^{-d}$) we get an expected search time which is $O((\log N)/d + (16a)^d \mathbf{E}(\#\{\, i \mid x \in S_i \,\}))$. For fixed d, this is $O(\log N + \mathbf{E}(\#\{i \mid x \in S_i\}))$. This means that for applications where the covering number grows at least as fast as $\log N$, we have an asymptotically optimal method. Otherwise, the expected time for the search grows like $O(\log N)$ which, while not asymptotically optimal, still gives good performance. By comparison, the expected total search time for the basic quadtree is $\Theta(N)$ for a worst case collection of sets $\{S_i\}$, and $\Theta(N^{1-1/d})$ for collections of sets which occur in practical problems.

Acknowledgments. We would like to thank Prof. J.S. Chen and his students Hui-Ping Wang and Cheng-Tang Wu for providing the meshfree method codes on which the computational results in Tables 3.1 and 3.2 are based. We would also like to thank the referees whose comments have helped to improve this paper.

REFERENCES

[1] D. J. Abel and J. L. Smith. A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Computer Vision, Graphics and Image Processing*, 24(1):1–13, 1983.

[2] Arne Andersson and Stefan Nilsson. Faster searching in tries and quadtrees—an analysis of level compression. In Jan van Leeuwen, editor, *Algorithms—ESA '94 (Utrecht)*, pages 82–93. Springer, Berlin, 1994. Lecture Notes in Computer Science #855.

[3] Arne Andersson and Kurt Swanson. On the difficulty of range searching. *Comput. Geom.*, 8(3):115–122, 1997.

[4] T. Belytschko, Y. Krongauz, D. Organ, M. Fleming, and P. Krysl. Meshless methods: An overview and recent developments. *Computer Methods in Applied Mechanics and Engineering*, 139:3–47, 1996.

[5] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational geometry*. Springer-Verlag, Berlin, 1997. Algorithms and applications.

[6] P. Gao and T. R. Smith. Space efficient hierarchical structures: Relatively addressed compact quadtrees for GISs. *Image and Vision Computing*, 7:173–177, 1989.

[7] I. Gargantini and H. H. Atkinson. Linear quadtrees: a blocking technique for contour filling. *Pattern Recognition*, 17:285–293, 1984.

[8] Donald Hearn and M. Pauline Baker. *Computer Graphics, 2nd Edition, C Version*. Prentice-Hall., 1994.

[9] Michael J. Laszlo. *Computational geometry and computer graphics in C++*. Prentice Hall, Inc., Upper Saddle River, NJ, 1996.

[10] S. Li and W. K. Liu. Reproducing kernel hierarchical partition of unity: Part I: Formulation and theory. *Internat. J. Numer. Methods Engineering*, 45(3):251–288, 1999.

[11] M. H. Overmars. Point location in fat subdivisions. *Infomation Proc. Letters*, 44:261–265, 1992.

[12] F. P. Preparata and R. Tamassia. Fully dynamic point location in a monotone subdivision. *SIAM J. Computing*, 18(4):811–830, 1989.

[13] Hanan Samet. The quadtree and related hierarchical data structures. *Comput. Surveys*, 16(2):187–260, 1984.

[14] Hanan Samet. *Applications of Spatial Data Structures*. Addison-Wesley, 1990.

[15] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Series in Computer Science. Addison–wesley, Reading, MA, 1990.

[16] Clifford A. Shaffer, Hanan Samet, and Randal C. Nelson. A geographic information system based on quadtrees. *International Journal of Geographic Information Systems*, 4,2:103–131, 1990.

[17] K. Yamaguchi, T. L. Kunii, K. Fujimura, and H. Toriya. Octree-related data structures and algorithms. *IEEE Computer Graphics and Applications*, 4(1):53–59, 1984.