Fast Spatial Decomposition and Closest Pair Computation for Limited Precision Input

John H. Reif*
Department of Computer Science
Duke University
Durham, NC 27708

reif@cs.duke.edu

Stephen R. Tate†
Department of Computer Science
University of North Texas
Denton, TX 76208

srt@cs.unt.edu

Abstract

In this paper we show that if the input points to the geometric closest pair problem are given with limited precision (each coordinate is specified with $O(\log n)$ bits), then we can compute the closest pair in $O(n \log\log n)$ time. To make use of the limited precision of the input points, we use a reasonable machine model that allows "bit shifting" in constant time — we believe that this model is realistic, and provides an interesting way of beating the $\Omega(n \log n)$ lower bound that exists for this problem using the more typical algebraic RAM model.

1 Introduction

Closest pair problems play a prominent role in computational geometry and have been studied from many different perspectives. These studies have led to important insights and results in such diverse areas as lower bounds for geometric problems, randomized algorithms, dynamic data structures, and parallel algorithms. This paper continues the study of closest pair problems by considering the effect of input representation, and in particular limited precision input representation, on the complexity of closest pair problems.

The standard model used in computational geometry problems is that of algebraic computation: point coordinates can be manipulated by the standard algebraic operations (addition, subtraction, multiplication, etc.), but no particular representation is assumed for these coordinates. One large benefit of this approach is that strong lower bound techniques can be used to lower bound the number of algebraic operations required for certain geometric problems. In particular, for deterministic computation it is known that finding the closest pair of points from among n supplied

*Supported by NSF Grant NSF-IRI-91-00681 and Rome Labs Contract F30602-94-C-0037.
†Supported in part by NSF Grant CCR-9409945.

points requires $\Omega(n \log n)$ time in this model (see, for example, Theorem 5.2 from the book by Preparata and Shamos [7]).

While any deterministic algebraic algorithm must take $\Omega(n \log n)$ time in the worst case, faster algorithms can be obtained using different models of computation. In particular, by removing the "deterministic" part of the model, a classic result of Rabin shows that there exists a randomized algebraic algorithm that has worst-case expected complexity $O(n)$ [8], thus beating the best possible deterministic algebraic algorithm. Randomization allows operations such as hashing to be performed in constant expected time, and in fact this is a key element of Rabin's algorithm. To demonstrate this dependence, Fortune and Hopcroft gave an $O(n \log \log n)$ time deterministic algorithm by augmenting their model with an operation that is essentially hashing in constant time [3].

In this paper, we consider only deterministic algorithms, but assume that input points are represented as fixed point binary values with $O(\log n)$ bits. In addition, we augment our model with the floor function; equivalently, we could allow constant time binary shift and mask operations. Both of these assumptions seem reasonable given today's computing hardware, and in fact seem more realistic than the algebraic model's assumption that arbitrary precision real numbers can be stored and manipulated. Under such a model, we show that the simple nearest neighbor and k-nearest neighbors problems can be solved in $O(n \log \log n)$ time. This is currently the only algorithm to beat the $\Theta(n \log n)$ bound that does not use either randomization or the strongly augmented model of Fortune and Hopcroft that allows constant time hashing.

Improved time complexity based on input representation is not a new concept. This idea has produced results such as radix sort, which beats the $\Omega(n \log n)$ lower bound for sorting, and the priority queue data structure of van Emde Boas, Kaas, and Zijlstra, which beats the $\Omega(\log n)$ lower bound on priority queue operations [11]. In fact, our algorithm uses a data structure that draws from the priority queue of van Emde Boas et al. to compute a spatial decomposition in time $O(n \log \log n)$ time. This algorithm is then incorporated into the closest pair work of Callahan and Kosaraju [1]. Our improved decomposition was in fact devised while inventing new algorithms for the n-body problem, which was also discussed by Callahan and Kosaraju, and we give in this paper a simple application of our decomposition to the n-body problem. A more in-depth treatment, including greatly improved algorithms for the n-body problem itself, is given in a separate paper [9].

2 Creating the Decomposition Tree

Our initial goal will be to construct a hierarchical space decomposition and a corresponding decomposition tree that has certain properties. The decomposition techniques we describe work for arbitrary dimension d, and in this section we treat d as a variable so that the dependence on dimension of our decomposition algorithm is apparent from the complexity estimates. In later sections, where we apply our decomposition techniques to several problems, we will treat d as a constant, as is usual for these problems. We fix an enumeration of dimensions $d_1, d_2, \cdots, d_d$, and we refer to dimensions by their position in this sequence ("dimension i" refers to d_i, for example).

Let R denote a rectilinear region of d-dimensional space, and let $l_i(R)$ denote the length of this region along dimension i. We use $\max(R)$ to denote the dimension number of the longest side[1] of R, and use $l_{\max}(R)$ as short-hand for $l_{\max(R)}(R)$.

Our spatial decomposition works in a fairly standard manner: it starts with an "initial region" containing all of the input points (in our case, the initial region will always be a d-cube), and then produces non-overlapping subregions that cover all the input points. Next, these subregions may be similarly subdivided. A tree naturally corresponds to such a decomposition, where there is a one-to-one mapping between tree nodes and spatial regions, and a region/node is the parent of all of the subregions produced when it was subdivided.

The decomposition we're interested in has some additional restrictions. In particular, all regions are rectilinear and have exactly two non-empty subregions. Furthermore, dimensions are always divided in half (or a larger power of two) in order to subdivide them, and the regions have to be "almost square". More specifically, every region in the spatial decomposition must satisfy the following properties:

- For each dimension i, either $l_i(R) = l_{\max}(R)$ or $l_i(R) = \frac{1}{2}l_{\max}(R)$.
- Every subdivided region contains exactly two subregions.
- Every region contains at least one input point.

If the input points were perfectly uniformly distributed, we could construct such a decomposition very easily: simply split the initial region along dimension 1, and then the resulting regions along

[1] In cases of a tie, we can disambiguate the term "longest side" to refer to minimum dimension number among the maximum-length sides.

dimension 2, and then dimension 3, etc. Eventually, we split along dimension d, and start over with dimension 1. We stop subdividing when we reach regions with just one input point, or when the input precision is exhausted so that we can no longer distinguish between points. In two dimensions, this is essentially a quad-tree, except that we split in the two dimensions separately. We call this decomposition technique the *standard decomposition sequence*. If we stop after k steps, then this is the *k-level standard decomposition*.

If the points are not uniformly distributed, then the standard decomposition sequence will produce an invalid decomposition. To understand why, consider an input in which half of the points are clustered together in one corner of the initial region, and the other half are at the opposite corner. The first split, as described above, works to separate the two halves; however, when we make the second split, we end up with regions containing no points, violating one of our requirements.

One modification to the above procedure allows us to find the desired decomposition tree: rather than blindly splitting a region along a dimension and using the two resulting regions, we keep splitting along the sequence of dimensions until a split results in two non-empty regions. These two regions (the union of which may be much smaller than the region we are subdividing) form the subregions of the current region, and are the children of the current node in the decomposition tree. Notice that not all of the space of the current node will necessarily be covered by subregions — in other words, our decomposition is not a partitioning. However, all of the input points are covered by the subregions, which is all that we required of our decomposition.

The decomposition tree resulting from this procedure is exactly the decomposition tree we would obtain by naively using the standard decomposition sequence and then trimming out all leaves that contain no input points, along with their parents (this is similar to the "rake" operation in tree contraction [6]). Given this correspondence, we refer to this trimmed tree as simply the *standard decomposition tree*; furthermore, we can stop this procedure after a fixed number k of dimension splits (also counting those that produce no tree nodes) which we refer to as the *k-level standard decomposition tree*. Since each internal tree node has exactly two non-empty children, this tree clearly has $O(n)$ nodes.

Figure 1 shows an example of ten points in two dimensions, and shows the standard decomposition and corresponding tree.

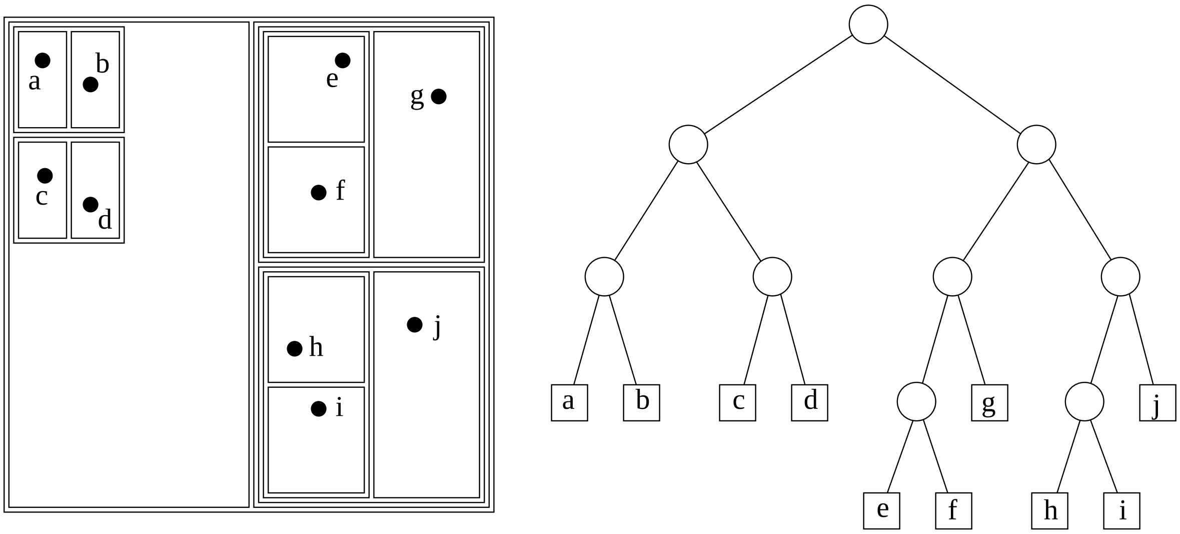

Figure 1: Example decomposition tree. Near-by box borders should actually coincide, but are shown separate to emphasize the hierarchy of regions.

The only remaining problem with this decomposition technique is that if the above decomposition technique is used directly, and if the dimensions are given to arbitrary precision, then the procedure could take a large amount of time (if the input coordinates are given to p bits, then it could require $\Theta(ndp)$ time). If this input is given to $O(\log n)$ bits, as assumed in this paper, we still require $\Theta(dn \log n)$ time in order to decompose a region containing n points. In the next sections, we describe a method for constructing such a decomposition tree for input points which are expressed with $O(\log n)$ bits of precision, and accomplishes this using only $O(d^2 n \log \log n)$ time.

2.1 Partial Decomposition Trees

In this subsection, we consider the problem of constructing a decomposition tree for m points in d-dimensional space, where the coordinates of each point are given using $\lfloor \frac{1}{d} \log n \rfloor$ bits[2]. The tree we will compute is exactly the $d\lfloor \frac{1}{d} \log n \rfloor$-level standard decomposition tree described above.

Essentially what we need is a way of constructing the standard decomposition tree with all long single-child paths contracted out of the tree. Since such paths can have length $\Theta(\log n)$, we need some way of building the decomposition tree without having to consider all nodes on these paths. The basic idea is this: Starting with a leaf node, do a binary search on the tree levels in order

[2] All logarithms in this paper are base 2.

to find the lowest two-child ancestor of this leaf — everything between this ancestor and the leaf must be a single-child path, so can be ignored. Since there are $O(\log n)$ levels to the tree, a binary search on the levels will require $O(\log\log n)$ time per search, giving us the improved time bounds that we need.

Unfortunately, while this idea is easy to state, the reality of the implementation is a bit messy. We need to build up and use an auxiliary data structure that we call the "support tree" which will allow us to perform binary search on tree levels. The support tree is largely inspired by the efficient priority queue data structure of van Emde Boas, Kaas, and Zijlstra (using what they call a "stratified tree structure") [11, 10].

The support tree is actually a full $h = d\lfloor \frac{1}{d}\log n\rfloor$-level binary tree, and the nodes of the tree correspond exactly with the full standard decomposition sequence described in the preceding section; however, only a subset of those nodes will be "active" (i.e., non-empty) at any given time. In particular, we will show that in a support tree for points that occupy m leaves of the tree, at most $O(m\log\log n)$ nodes will be active.

Since the support tree is a full binary tree, we can use standard techniques (usually used in conjunction with heaps) to map the tree nodes into an array with indices $1..2^h-1$; however, the ordering of the nodes within a level of this array requires further explanation, and is discussed in the following section.

2.1.1 Mapping regions to support tree nodes

We first consider how to take an arbitrary input point and find the region containing that point on any given level of the tree. As is standard practice, tree levels are numbered from 0 (corresponding to the root) to level $h-1$ (corresponding to the level containing the leaves). On level k, we have divided the original root region k times. The number of times we have subdivided in dimension i is exactly the same as the number of most significant bits we use from the dimension i coordinate to determine the region at level k; we can define the function $bits(i,k)$ to represent this value as follows.

$$bits(i,k) = \begin{cases} \lfloor k/d\rfloor + 1 & \text{if } i \le k - d\lfloor k/d\rfloor, \\ \lfloor k/d\rfloor & \text{otherwise.} \end{cases}$$

Since our machine model allows us to strip out arbitrary bit positions and to shift by arbitrary amounts within a single cycle, we can build up an array index (giving a support tree node) as follows, which we call procedure LevelIndex: For each dimension i (with $1 \le i \le d$) strip out the most significant $bits(i, k)$ bits from that coordinate. Next, concatenate these bit sequences together into one word, with a single bit with value 1 concatenated into the most significant position. This procedure gives the desired index into our array representation of the tree, and has time complexity $\Theta(d)$.

2.1.2 Support tree definitions

As mentioned above, operations on the support tree work by doing a binary search on levels of the tree. In a binary search of levels, you are initially given two levels, a lower bound lo and an upper bound hi, and depending on the outcome of a computation involving level $mid = \lfloor (lo+hi)/2 \rfloor$ the process either stops or continues with a smaller range of either $lo..mid-1$ or $mid+1..hi$. This process defines what we refer to as "valid ranges":

- Range $0..h-2$ is a valid range.
- If range $lo..hi$ is valid, and $mid = \lfloor (lo+hi)/2 \rfloor$, then $lo..mid-1$ and $mid+1..hi$ are valid (if they are non-empty).

Notice that for any level k, it is the "mid level" for exactly one valid range, which we denote by $lo_k..hi_k$ (this implies that $k = \lfloor (lo_k + hi_k)/2 \rfloor$), and we refer to this range as the "range for level k". Given any support tree node x, we can also talk about the range for that *node*, written $lo_x..hi_x$, which simply means the range corresponding to node x's level.

Now we can define exactly what the support tree is. Each active node x of the support tree contains the following information:

- An unordered list of references to all non-empty support tree descendants on level $hi_x + 1$, and a count of how many items are on this list.
- An additional counter and pointer to a decomposition tree node (*not* a support tree node), which will be used by later processing routines.

In addition to this per-node information, we keep track of a global list of all active nodes for each level of the tree.

We have referred several times to "active nodes" in the tree, but have not described how a node becomes active. The non-empty nodes that are active are closely tied to positions of "branchings" (internal nodes in the compete tree that have more than one child). The conditions required for a node to be active can be easily described and are called the *support tree conditions*.

Specifically, any leaf node corresponding to a non-empty region is active, and a non-empty internal node x is active if and only if either

(a) $lo_x = 0$,

or (b) there is a branching in the subtree of levels $lo_x - 1..hi_x$ that includes x.

In the remainder of this discussion we assume we are given an initialized, empty support tree at the beginning. More will be said about this assumption in a later section.

2.1.3 Building the support tree

To build the support tree, we sequentially step through all input points performing the following actions for each point x: we first determine which support tree leaf this point belongs in by calling procedure LevelIndex (outlined in section 2.1.1). Next, we add this point to the list of points maintained by that leaf node. Finally, if this was the first point in this leaf node we call AddPoint$(node, nil, 0, h-2)$, where procedure AddPoint adjusts the internal nodes of the support tree, and is defined in Figure 2.

Lemma 2.1 Given m points whose coordinates are given to $\lfloor \frac{1}{d} \log n \rfloor$ bits of precision, Algorithm AddPoint creates a support tree whose active nodes satisfy the support tree conditions in time $O(dm \log\log n)$.

Proof: In order to prove the correctness of the algorithm, we introduce some terminology dealing with steps in the algorithm. An "update" is the response to a request to add a new point to the support tree, including all recursive calls made to satisfy this request. A "step" is a single call of the code in AddPoint(excluding recursive calls); therefore, an update is made up of many steps. We prove the correctness of the algorithm by verifying the following invariant: at the beginning of any step, all levels outside of the range $lo..hi$ are properly updated with the current update request. At the last step of an update, the $lo..hi$ range vanishes, and so all levels of the tree are properly updated.

For any step of the algorithm, one of five conditions are satisfied (labeled as (1) through (5) in Figure 2), and corresponding actions are taken that cause a recursive call on a smaller range. We

```
Algorithm  AddPoint(node, alt, lo, hi):

    Input: References to two support tree nodes, node and alt, and level numbers lo and hi.
    Output: Updated support tree to include new node node.

    if lo ≤ hi then
        mid ← ⌊(lo+hi)/2⌋;
        midnode ← LevelIndex(mid, node);
        Add node to reference list of midnode;
        if alt = nil then
            if midnode reference list contains 1 item (which must be node) then
(1)             AddPoint(midnode, nil, lo, mid − 1);
            else if midnode reference list contains 2 items then
(2)             alt ← reference other than node in midnode list;
                AddPoint(node, alt, mid + 1, hi);
            else
(3)             AddPoint(node, nil, mid + 1, hi);
        else
            midalt ← LevelIndex(mid, alt);
            Add alt to reference list of midalt;
            if midnode = midalt then
(4)             AddPoint(node, alt, mid + 1, hi);
            else
(5)             AddPoint(midnode, midalt, lo, mid − 1);
```

Figure 2: Algorithm AddPoint (labels on the left are for reference in the text).

will show that in each case all nodes that newly satisfy one of the support tree conditions and are on a level that is excluded from the smaller range are properly updated, thereby proving that the invariant is maintained.

Case (1): Our new node is the only reference in *midnode*, and so there are no branchings in levels $mid..hi$, and no node y on any level in $mid + 1..hi$ can newly satisfy support tree condition (b). Furthermore, no node y on any level in $mid + 1..hi$ can have $lo_y = 0$, so no nodes on these levels have newly satisfied support tree condition (a). In conclusion, the only place on levels $mid..hi$ where a node may newly satisfy a support tree condition is on level mid, and the only change on that level is node *midnode*, which is activated. Therefore, all nodes on levels excluded from the next recursive call are properly updated in this case.

Case (2) or (3): If *midnode* contains references to two or more nodes, including the new one, then *midnode* was active before this update and there are no new structural changes in the full tree above level *mid*. This means that there are no branchings introduced on levels $lo..mid-1$, and hence no node on levels $lo..mid-1$ newly satisfies condition (b). Furthermore, since any node y on levels $lo..mid-1$ that has $lo_y = 0$ and is non-empty after this update must have been non-empty before the update also, no nodes on levels $lo..mid-1$ newly satisfy condition (a). Finally, the only node on level *mid* that changes is *midnode*, which was in fact previously activated, and so all nodes on the excluded levels $lo..mid$ are properly updated before the end of this step.

Case (4): In this case, $midnode = altnode$ and so the entire path from *midnode* up to its ancestor on level $lo-1$ was non-empty before this update, and this hasn't changed due to this update. Since the empty/non-empty status of no nodes on levels $lo..mid-1$ have changed due to this update, and no new branchings have been introduced on those levels, there can be no nodes that newly satisfy either condition (a) or condition (b). There must be a branching somewhere in the range $lo-1..hi$, and so *midnode* is activated by algorithm AddPoint, as required by the support tree conditions. Therefore, in case (4), all nodes in the excluded levels $lo..mid$ are properly updated by this step.

Case (5): Notice that whenever *alt* is non-nil, there are exactly two non-empty regions on level $hi+1$ that are descendants of the current subtree. Since $midnode \neq altnode$, each of *midnode* and *altnode* must have exactly one active descendant on level $hi+1$. This means that no new branchings are introduced in levels $mid..hi$, and hence no nodes in $mid+1..hi$ newly satisfy support tree condition (b). Furthermore, none of these nodes y have $lo_y = 0$, and so no nodes on levels $mid+1..hi$ can satisfy condition (a). Since the only two non-empty nodes on level *mid* are activated by this step, all nodes on the excluded levels $mid..hi$ are properly updated by the end of the step.

Since all cases lead to the conclusion that all nodes on excluded levels are properly updated, we see that the invariant is maintained and the algorithm correctly constructs the support tree with all nodes satisfying the support tree conditions being active.

For the time complexity, notice that all operations in AddPoint are constant time except for the call to LevelIndex and the recursive call. Since LevelIndex takes time $O(d)$ and there are at most $O(\log h)$ levels of recursion, the total time for procedure AddPoint is $O(d \log h) = O(d \log\log n)$. The total time to add all m inputs points is therefore $O(dm \log\log n)$. ∎

2.1.4 Building the partial decomposition tree

In order to use our support tree to build the desired decomposition tree, we first must compute for each active internal node the number of active leaves that are descendants. Notice that each node already holds the number of active leaves in its canonical subtree, but we do not know the number of active leaves for the *entire* subtree rooted at that node. Fortunately, this is easy to compute: For range $lo..hi$ we assume that the number of active leaves in subtrees at level $hi+1$ has already been computed and using this information we can easily compute the number of active leaves for each active node on level $mid = \lfloor (lo+hi)/2 \rfloor$ (simply walk through each node's reference list and add up the number of active leaves for each referenced node). Next we recursively call this procedure for ranges $lo..mid-1$ and $mid+1..hi$. With an initial call for range $0..h-2$ this procedure correctly computes the number of active leaves for each node, and the time is proportional to the combined size of all reference lists in the support tree. Since the tree was computed in time $O(dm \log\log n)$, this clearly bounds the size of the reference lists, and hence also bounds the size of this procedure.

We will refer to the computed number of active leaves for a node x by the notation x.count. Using these values, given any node x we can find the lowest node in the full decomposition tree that is above x and has more than one active child (one of which will be an ancestor of x). This node corresponds to the parent of node x in the standard decomposition tree, and so we define a procedure FindParent as shown in Figure 3. For a node x this procedure is initially called as FindParent$(x, 0, h-2)$.

Lemma 2.2 FindParent$(x, 0, h-2)$ correctly returns node x's desired ancestor (i.e., the parent of region x in the standard decomposition tree), or nil if such an ancestor doesn't exist. The time complexity of FindParent is $O(d \log h) = O(d \log\log n)$.

Proof: The correctness of FindParent follows from the following easily verified invariant: If the desired ancestor is on level ℓ, then at all times (and in all recursive calls) in this procedure we have

Algorithm FINDPARENT(x, lo, hi):

INPUT: Support tree node x and level numbers lo and hi.
OUTPUT: Parent node of x in reduced tree.

```
if lo > hi then
    par ← level hi ancestor of x (nil if hi < 0);
else
    mid ← ⌊(lo+hi)/2⌋;
    if level(x) ≤ mid then
        par ← FINDPARENT(x, lo, mid − 1);
    else
        midnode ← level mid ancestor of x;
        if midnode.count ≤ x.count then
            par ← FINDPARENT(x, lo, mid − 1);
        else
            par ← FINDPARENT(x, mid + 1, hi);

return par;
```

Figure 3: Algorithm FINDPARENT.

$lo - 1 \le \ell \le hi$. Since this level ℓ ancestor is a branching, this implies that *midnode* for this $lo..hi$ range is active (by the support tree conditions), and so the algorithm only queries active nodes. Furthermore, at the very last recursive call, we must have $lo = hi+1$, so we have $lo-1 = hi \le \ell \le hi$, which means that the ℓ must be the same as hi at this stage. The algorithm returns the ancestor of x on this level, so the correctness follows.

For the time complexity, notice that every operation except for the recursive calls and the line

$$midnode \leftarrow \text{ level mid ancestor of } x;$$

takes constant time. This line uses procedure LEVELINDEX to find the appropriate ancestor in $O(d)$ time, and due to the binary search on levels there are only $O(\log h)$ recursive calls. Thus the time complexity of FINDPARENT is $O(d \log h) = O(d \log \log n)$. ∎

Since FINDPARENT can determine, for any node in the decomposition tree, its parent in the standard decomposition tree, we can build the entire procedure by repeatedly calling FINDPARENT. This is shown explicitly in Figure 4.

Algorithm PARTIALTREE:

INPUT: Set of m points in d-dimensions. Each coordinate is $\lfloor \frac{1}{d} \log n \rfloor$ bits.
OUTPUT: The $d\lfloor \frac{1}{d} \log n \rfloor$-level standard decomposition tree for these points.

```
Build support tree;

h ← d⌊(1/d) log n⌋;
for each x ∈ level h − 1 active list do
    v ← New decomposition tree node;
    x.dtreenode ← v;
    Put x in queue Q;

while Q is non-empty do
    x ← Dequeue(Q);
    v ← x.dtreenode;
    p ← FindParent(x, 0, h − 1);
    if p.dtreenode = nil then
        u ← New decomposition tree node;
        p.dtreenode ← u;
    else
        u ← p.dtreenode;
        Add p to queue Q;
    Set appropriate child of u to reference v;
    v.parent ← u;

Clear support tree;
```

Figure 4: Algorithm PARTIALTREE.

Lemma 2.3 Given an initially cleared support tree, algorithm PARTIALTREE correctly computes the $d\lfloor \frac{1}{d} \log n \rfloor$-level decomposition tree for m points in $O(dm \log\log n)$ time, and leaves the support tree cleared.

Proof: The correctness follows directly from the discussion of properties of the support tree. To determine the time complexity, consider the individual pieces: building and clearing the support tree both take $O(dm \log\log n)$ time, as discussed in previous sections. Each iteration of the main loop in PARTIALTREE consists of a call to FINDPARENT ($O(d \log\log n)$ time), and the remainder are constant-time operations. This loop is executed once per node of the *constructed* standard decomposition tree (to find its parent), and since there are $O(m)$ nodes of the standard decomposition tree, the total time for all loop iterations is $O(dm \log\log n)$. Putting all this together, the total time complexity of PartialTree is $O(dm \log\log n)$. ∎

Note: By requiring an initially cleared support tree and leaving the support tree cleared, we can re-use the same support tree structure over and over for arbitrarily many calls to PartialTree. Avoiding the otherwise $\Theta(n)$ time initialization of the support tree is important when, as in the next section, there may be many calls with $m \ll n$.

2.2 Putting It Together

The algorithm of the previous section works when the number of bits used to specify each coordinate is $\lfloor \frac{1}{d} \log n \rfloor$, but we would like to be able to compute the decomposition tree when the number of bits in each coordinate is $c \log n$ (for an arbitrary constant c). Unfortunately, simply scaling up PartialTree is not efficient, since the obvious technique would require $\Theta(n^c)$ space (and $\Theta(n^c)$ time to initialize this space, although this can be avoided). We avoid this problem by making repeated calls to PartialTree, rather than scaling it up. We begin with a high-level description of our full decomposition algorithm, and then fill in the details.

Let $b = \lfloor \frac{1}{d} \log n \rfloor$, which is the number of bits that can be handled by algorithm PartialTree. The main idea is to make several passes over the data, each pass processing b bits of each coordinate. We begin by sending all points to PartialTree, but sending only their most significant b bits. When PartialTree is finished, it will have produced a decomposition tree for this data and we will have easy access to all the leaves of the tree and their associated lists of points.

All points that are located in any single leaf of this tree must have the most significant b bits in common, and so we can send these points back to algorithm PartialTree, but now using the second most significant block of b bits. The tree that this produces can be linked into the tree from the first call to PartialTree, in place of the leaf that contained all those input points. We repeat this process for every leaf that contains more than one input point. Eventually, we reach a point where either every leaf has one input point or we run out of bits in the input coordinates. The final result is a single tree that contains the full spatial decomposition of the input points.

Pseudo-code for this algorithm is given in Figure 5. We make use of a queue of "sets", where each set contains a subset of input points, and some associated data referred to in the algorithm as the

```
Algorithm DECOMPOSE:

    INPUT: Set of points in d-dimensions. Each coordinate is c log n bits.
    OUTPUT: Decomposition tree for these points.

    b ← ⌊(1/d) log n⌋;
    m ← ⌈(c log n)/b⌉;
    Divide input coordinates into m blocks of b bits each.
    Blocks are numbered 1..m, with 1 being the most significant block.
    Make new set S with: S.block ← 0;
                         S.parent ← nil;
                         S.list ← List of all input points;
    Add S to queue Q;

    while Q not empty do
        S ← Dequeue(Q);
        T ← PARTIALTREE(S.list using coordinate bits from block S.block);
        Link root of T to S.parent;
        if S.block < m then
            for each leaf ℓ of T containing more than 1 point do
                Make new set R with: R.block ← S.block + 1;
                                     R.parent ← ℓ;
                                     R.list ← List of points in ℓ;
                Add R to queue Q;
```

Figure 5: Algorithm DECOMPOSE.

"block" and the "parent" of the set. Initially, the only set is the set of all input points, with the block field set to zero and the parent field set to "nil" (representing a non-existent parent value). Further decompositions of this set correspond to passing the data through algorithm PARTIALTREE to produce part of the decomposition tree. Each of these produced sets has its block field set to the number of the last block of coordinate bits that were sent to PARTIALTREE and its parent field set to the leaf node containing these input points in the full decomposition tree (a node which was created in a preceding call to PARTIALTREE.

The algorithm makes use of a queue of sets. It should be noticed that by keeping the points in linked lists, every operation except the call to PARTIALTREE requires only constant time. In order to satisfy this, the queue will only contain *references* to sets, rather than copies of the sets.

Theorem 2.1 Given n points in d dimensions, where the coordinates of each point are given using $c \log n$ bits, Algorithm DECOMPOSE computes the decomposition tree for these points in $O(d^2 n \log\log n)$ time.

Proof: The correctness of the algorithm is obvious. We will show that the complexity of the algorithm is $O(d^2 n \log\log n)$.

Consider block i of coordinate bits, where $0 \leq i < m$. During the execution of algorithm DECOMPOSE say that there are $n(i)$ different sets R created that have R.block $= i$, and let $k_1, k_2, \cdots, k_{n(i)}$ be the number of particles in each of these sets. Each input point can be in at most one of these sets, and so $k_1 + k_2 + \cdots + k_{n(i)} \leq n$. For a set with k_j points, Lemma 2.3 showed that the amount of time required to process this set is $O(dk_j \log\log n)$ for the call to PARTIALTREE, and $O(k_j)$ for everything else (there can be at most k_j leaves in the tree produced by this call to PARTIALTREE, and it takes a constant amount of time to process each leaf). This simplifies to just $O(dk_j \log\log n)$ time. Combining all this, we see that the amount of work required to process all of the sets R with R.block $= i$ is

$$O(dk_1 \log\log n + dk_2 \log\log n + \cdots + dk_{n(i)} \log\log n) = O(d(k_1 + k_2 + \cdots + k_{n(i)}) \log\log n)$$

$$= O(dn \log\log n).$$

Finally, there are $O(\frac{c \log n}{b}) = O(cd)$ blocks of bits, and so the overall time complexity of DECOMPOSE is $O(d^2 n \log\log n)$. ∎

3 Applications

In this section, we show how to use the decomposition tree constructed by Algorithm DECOMPOSE in order to solve two problems: geometric closest pair, and n-body potential field evaluation. The unified approach to these two problems is largely due to Callahan and Kosaraju [1], and we begin by making explicit the connection between our decomposition tree and their algorithms. As mentioned earlier, we now treat d as a constant, as is typical for these problems — in both cases the dependence on d is exponential, and so our new decomposition algorithm of the previous section has much smaller dependence on d than these applications.

3.1 Closest Pair and k-Nearest Neighbors

For any node v of the decomposition tree, let $R(v)$ denote the region corresponding to that node. Recall that in the standard decomposition tree produced by Algorithm Decompose the region $R(v)$ is subdivided into exactly two subregions. The two subregions do not necessarily cover region $R(v)$, but have the following two properties: they are *exactly* the same size, and they are separated by a single hyperplane that is adjacent to both subregions.

For any tree node v we can define a slightly different region $\widehat{R}(v)$ corresponding to this node as follows:

- If v is the root of the decomposition tree, then $\widehat{R}(v)$ is the entire initial d-cube.
- If v is not the root, and $p(v)$ is the parent of v, then $\widehat{R}(v)$ is obtained by splitting $\widehat{R}(p(v))$ using the same hyperplane that separates v from its sibling, and taking the side that includes $R(v)$.

Note that for any node v the region corresponding to that tree node is included in $\widehat{R}(v)$. The main difference between these "$\widehat{R}$ regions" and the actual regions corresponding to the tree nodes is that the "$\widehat{R}$ regions" do not leave any uncovered space — the sub-$\widehat{R}$-regions do in fact *cover* the parent region.

Finally, we define one last region for each tree node v. Define $Q(v)$ to be the smallest rectilinear region that includes all the inputs points contained by $R(v)$. This is some contraction of $R(v)$ to the smallest possible size, and so we note that of the three different regions defined we can say that $Q(v) \subseteq R(v) \subseteq \widehat{R}(v)$.

The first step in the decomposition-based algorithms of Callahan and Kosaraju [1] is the definition of what is called a "fair split tree". Given an internal node v in a decomposition tree, they call the split that produces that node's children a *fair split* if the hyperplane that separates the two children of v is at a distance of at least $l_{\max}(Q(v))/3$ from each of the two boundaries of $\widehat{R}(v)$ that are parallel to it. A *fair split tree* is one in which every internal node corresponds to a fair split.

Lemma 3.1 Algorithm Decompose produces a fair split tree.

Proof: Consider any internal node v of our tree. If a and b are the children of node v, then it was remarked before that $R(a)$ and $R(b)$ have *identical* size. Furthermore, we know that the hyperplane separating a and b exactly splits in half the region $R(a) \cup R(b)$, and that this split was along a maximum length dimension of $R(a) \cup R(b)$. Since $Q(v)$ must be entirely contained within $R(a) \cup R(b)$, it then follows that the distance from the separating hyperplane and either of the two parallel sides of $\widehat{R}(v)$ is *at least*

$$l_{\max}(R(a) \cup R(b))/2 \geq l_{\max}(Q(v))/2 > l_{\max}(Q(v))/3,$$

which shows that the split at node v is a fair split. Since v was chosen arbitrarily, all internal nodes must correspond to fair splits, and so the decomposition tree produced by Algorithm Decompose is a fair split tree. ∎

Finding a fair split tree is the first and most important step in the algorithms of Callahan and Kosaraju. The following lemma summarizes how this fair split tree is important, and is evident from the work of Callahan and Kosaraju [1]. In particular, refer to Theorem 4.3, the paragraph below the proof of Lemma 8.1, and Theorem 8.4 of the cited paper [1].

Lemma 3.2 [1] Given a fair split tree, the closest pair of points can be computed in $O(n)$ time, and the k nearest neighbors can be computed in $O(kn)$ time.

Finally, combining this lemma with our algorithm Decompose (see Theorem 2.1) gives the following theorem, which is our main result.

Theorem 3.1 Given n points, where the coordinates of each point are given using $c \log n$ bits, we can compute the closest pair of this pointset in $O(n \log\log n)$ time, and can compute the k nearest neighbors of each point in $O(n \log\log n + kn)$ time.

3.2 n-Body Potential Field Evaluation

The n-body potential field evaluation problem, or simply the "n-body problem" is as follows. Given n point charges and their associated charge strengths, compute the value of the potential field (or the induced force) generated by these charges at each of the point charge locations. A breakthrough in this problem was made by Greengard and Rokhlin [4, 2, 5] who gave an algorithm for the two-dimensional n-body problem that ran in time $O(np \log p)$ when the input points met certain uniform-distribution constraints and with the condition that $p \geq \log n$. Callahan and Kosaraju [1] gave an algorithm for this same problem that had time complexity $O(n \log n + np \log p)$, and removed all of the assumptions and requirements imposed by the algorithm of Greengard and Rokhlin.

Similar to the results of the previous section, we can apply our decomposition tree to the problem of evaluating n-body potential fields using the techniques of Callahan and Kosaraju [1]. Our main result for the n-body problem is stated in the following theorem.

Theorem 3.2 Given n point charges whose coordinates are expressed using $c \log n$ bits, and whose charge strengths are given, let $T(p, n)$ be the time required to compute the n-body potential to p bits of accuracy given a fair split tree for the point charges. Then we can compute all the potentials required by the n-body problem in time $O(n \log \log n + T(p, n))$.

Using direct application of the algorithms of Greengard and Rokhlin [4, 2, 5] and Callahan and Kosaraju [1] gives $T(p, n) = O(np \log p)$, for an overall complexity of $O(n \log \log n + np \log p)$. However, this is not the best possible result for the n-body problem under our limited input precision restrictions! Using improved techniques for the actual potential field evaluation developed by the authors of this paper, it is possible to reduce $T(p, n)$ to $O(n \log^2 p)$, and hence solve the n-body problem for limited precision inputs in time $O(n \log \log n + n \log^2 p)$ time. As this result requires substantial modifications of the well-known n-body algorithm due to Greengard and Rokhlin, we describe this result in a separate paper that looks deeply into n-body potential field evaluation [9].

4 Conclusion

In the paper, we have shown how to exploit the input representation of certain geometric problems, under the realistic assumption that the points are give to a precision of $O(\log n)$ bits. We developed a new spatial decomposition technique, and applied this decomposition to show how to solve closest pair problems and n-body potential field evaluation problems more efficiently than is possible using the algebraic RAM model with no use of input representation. In particular, we give an $O(n \log \log n)$ algorithm for finding the closest pair of a pointset, beating the $\Theta(n \log n)$ time algorithms which are optimal under the algebraic RAM model. Similar improvements are given for the n-body potential field evaluation problem.

References

[1] Callahan, P. B., and Kosaraju, S. R. A decomposition of multi-dimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. ACM 42*, 1 (Jan. 1995), 67–90.

[2] Carrier, J., Greengard, L., and Rokhlin, V. A fast adaptive multipole algorithm for particle simulations. *SIAM Journal of Scientific and Statistical Computing 9*, 4 (1988), 669–686.

[3] Fortune, S., and Hopcroft, J. A note on Rabin's nearest-neighbor algorithm. *Inf. Process. Lett. 8*, 1 (Jan. 1979), 20–23.

[4] Greengard, L., and Rokhlin, V. A fast algorithm for particle simulations. *Journal of Computational Physics 73* (1987), 325–348.

[5] Greengard, L., and Rokhlin, V. On the efficient implementation of the fast multipole algorithm. Tech. Rep. Technical Report RR-602, Yale University, Department of Computer Science, 1988.

[6] Kosaraju, S. R., and Delcher, A. L. Optimal parallel evaluation of tree-structured computations by raking. In *Proceedings of the 3rd Aegean Workshop on Computing* (1988), pp. 101–110.

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

[8] Rabin, M. Probabilistic algorithms. In *Algorithms and Complexity, Recent Results and New Directions*, J. F. Traub, Ed. Academic Press, New York, 1976, pp. 21–39.

[9] Reif, J. H., and Tate, S. R. N-body simulation I: Fast algorithms for potential field evaluation and trummer's problem. Tech. Rep. N-96-002, University of North Texas, Department of Computer Science, 1996.

[10] van Emde Boas, P. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett. 6*, 3 (June 1977), 80–82.

[11] van Emde Boas, P., Kaas, R., and Zijlstra, E. Design and implementation of an efficient priority queue. *Math. Sys. Theory 10* (1977), 99–127.