Fast Cluster Detection in Point Sets

Brendan Mumey
brendan@cs.washington.edu
Department of Computer Science
University of Washington, Seattle, WA 98195

Abstract

A central question in astrophysics is "Why is matter in the universe distributed the way it is?" In particular, when we look at images of space we see long filamentous strands of matter and regions which are essentially void. A quantitative approach to measuring how "clustered" a point set is necessary to answer this question objectively; to prevent drawing artificial conclusions based on what the eye wants to see. One proposed approach is to use information contained in a *Euclidean Minimum Spanning Tree*, or EMST built on matter-representing points in space. Whereas the best known algorithms for finding a full EMST in two dimensions run in $O(n \lg n)$ [1] time [3], we present an algorithm for approximating only the information necessary to identify clusters of planar sets in $O(\lg(\frac{\Delta}{\epsilon\tau})n)$ time, where Δ is the sidelength of the smallest bounding box for the point set, τ is a threshold value controlling how far apart points may be and still be considered clustered, and ϵ is an accuracy parameter. For three dimensions, we can currently give an $O(\lg(\frac{\Delta}{\epsilon\tau})n + \sqrt{\frac{1}{\epsilon}}(\lg(\frac{1}{\epsilon}))^{1.5}n)$ time algorithm. Finally an $\Omega(n \lg n)$ lower bound in the algebraic decision tree model of computation is established.

1 Introduction

Generally speaking a spanning tree is subset of edges in a (combinatorial) graph which form a tree (with no restrictions on branching factor) and reach every vertex. If the edges in the graph also have weights associated with them, we say a spanning tree is minimum if it the sum of all weights in its included edges is the minimum possible. The Euclidean prefix means the graph involved is the normal complete graph formed on a point set, with the distance between two points taken as the weight of the edge between them. EMSTs have several properties which are useful in the context of measuring the degree of non-uniformity in point sets (branching factor, degree of balance). The property of EMSTs

[1]We adopt the convention $\lg \stackrel{\text{def}}{=} \log_2$.

which is applicable to cluster detection is the fact that they provide a convenient means of computing the *bottleneck distance* between pairs of points.

The bottleneck distance between two points is defined to be the minimum possible weight of the maximum weight edge in a path between them:

$$\delta_b(x, y) = \min_{\text{paths } P \text{ connecting } x \text{ and } y} \; \max_{\text{edges } e \in P} \text{length}(e)$$

That is, if you only charge according to most expensive edge traversed then the bottleneck cost is the cost of the least expensive path between a pair of points. Two points in a tightly clustered clump will be close under the bottleneck distance metric as there will be many nearby intermediate points to go through. Only the maximum length edge matters. Conversely, points which are isolated must incur large bottleneck costs to reach other points. It is this property which we exploit to define clusters. We will say that two points x and y are *τ-clustered* if and only if $\delta_b(x, y) \leq \tau$. This defines an equivalence relation on the point set and so implicitly partitions it into sets of related points; clusters. A property of EMSTs is that if one simply deletes all the edges above the threshold value, the remaining forest of connected components will correspond exactly to the set of clusters according to the bottleneck definition.

An unfortunate fact about EMSTs is that the best known methods for constructing them in two, three, or more than three dimensions take respectively $O(n \lg n)$, $O((n \lg n)^{1.5})$, or $O(n^2)$ time [3, 4, 5, 1] (n is the number of points). This is unfortunate because astrophysicists would like to run simulations of 10^7 or 10^8 points in order to adequately capture scale phenomena in the evolving universe. The extra n factor is prohibitive. In light of this it is natural to ask if a good approximation to the solution can be found more quickly. We say an approximate cluster finding algorithm is ϵ-*close* if:

1. $\delta_b(x,y) > (1+\epsilon)\tau \Rightarrow x$ and y are identified to be in different clusters

2. $\delta_b(x,y) \leq \tau \Rightarrow x$ and y are identified to be in the same cluster

There are methods for constructing approximations to the EMST in $O(n \lg n + \epsilon^{-d/2} \lg(\frac{1}{\epsilon})n)$ time [1], from which an ϵ-close cluster partition may be extracted (d is the dimensionality of the point set). However we do not need to fully construct an (approximate) EMST as we only use part of the information that it provides. We only need to know what the connected components are when the edges above the threshold are thrown away. In light of this, we develop a new algorithm which finds an ϵ-close cluster structure in $O(\lg(\frac{\Delta}{\epsilon\tau})n)$ time where the constant of proportionality is quite small. It is adapted to three dimensions to give an $O(\lg(\frac{\Delta}{\epsilon\tau})n + \sqrt{\frac{1}{\epsilon}}(\lg(\frac{1}{\epsilon}))^{1.5}n)$ time method. Although it is possible to generalize our techniques to higher dimensions, we will not consider this in this version of the paper.

In the next section we describe the approximate algorithm and prove its running time bound and correctness. The following section will prove an $\Omega(n \lg n)$ lower bound on the general problem. (An $O(n \lg n)$ exact algorithm can be given for the plane by first building the full EMST using [3], but the author does not know how to achieve $O(n \lg n)$ time in higher dimensions.) In the next section a rough comparison of the efficiency of our method versus alternative approaches is given for a typical problem. We conclude and state some open questions in the final section.

2 The Approximate Algorithm

The algorithm is divided into two phases. In the first phase the algorithm constructs an θ-*resolved quad tree*. Here we are assuming a two dimensional point set. If it were three, we would have an oct tree instead. For ease of explanation, we will discuss the algorithm for the plane and note any discrepencies in the algorithm adapted for higher dimensions as they come up. A *quad tree* is a recursive subdivision of the point set. It is based on first finding the smallest bounding cube for the points. Let the sidelength of this cube be Δ. The points are added one by one into the box. Whenever two points land in the same box, the box is recursively divided into quadrants and the points placed in the appropriate quadrant. Many particle simulation codes rely on an oct tree as the central data structure [12, 13]. We expect the algorithm to be used particle simulator, so it is likely that an oct tree on the data will already be present. If this is not the case, they are easy to construct.

We say a quad tree is *θ-resolved* if we stop dividing cells when their sidelength goes below θ. We call these smallest cells *θ-cells*. Each θ-cell has a list of the all points which are added to it. On each recursive subdivision the sidelength is reduced by a factor of two. We start with a sidelength of Δ and stop at when the sidelength reaches θ. Thus the maximum depth a θ-resolved quad tree will reach when constructed with a starting box of sidelength Δ is $\lceil \lg(\frac{\Delta}{\theta}) \rceil$. As will be explained later, we choose $\theta = \Theta(\epsilon\tau)$. The cost of adding a new point to the quad tree in construction is just the number of recursive steps it takes to reach an empty or θ-sized cell. This will be bounded by the maximum depth of the tree. So the cost of the first phase of the algorithm will be $O(\lg(\frac{\Delta}{\epsilon\tau})n)$, which is just the cost of adding n points.

The second phase of the algorithm is more complicated. It uses the underlying quad tree as a means to filter points. The basic idea is to start a point x and determine which points are contained within a ball centered on x of radius τ. These points can be added to the cluster containing x. If a ball is expanded around every point it is fairly easy to argue that all the clusters will be found correctly. If two points x and y have a bottleneck distance less than the threshold τ, then we know there will be a path through points in the point set connecting them such that the distances between adjacent points in the path are all less than τ. Since we expand a ball around each point in the point set we will merge together all the points in the path, including x and y into the same cluster. So x and y will be correctly identified as being clustered.

One detail missing in the above approach is what order the points should be expanded in. There are two approaches. The first is a *merge-based* approach akin to Kruskall's algorithm for finding minimum spanning trees [9]. When two different clusters are merged together (as the result of a ball around a point in one intersecting a point in the other), we need to have a way to quickly "inform" all the points in boths clusters that they are now in one larger common cluster. A good way to do this is use a union-find data-structure. As is shown in [7], this can be done very efficiently. The amortized time cost per merge is $O(\alpha(n))$, where α is a very slow growing function (in some sense it is the inverse of Ackermann's function [9]). The other approach lacks some of the simplicity of the merge-based approach but avoids the use of a union-find data-structure as it builds up each cluster sequentially. We call this approach *search-based.* It is similar to Prim's algorithm for MST-finding, as it begins with a single point and incrementally adds all the points which are clustered with it. When there are no more points to add, the algorithm proceeds to a new unvisited point and completely builds its cluster. An advantage of this method is that there is never any need to revise what cluster a given point is in. Clusters are never merged together. This means that the ability to merge clusters quickly is unnecessary and so we do not need to use a union-find data-structure. The algorithm presented will use the search-based approach.

The general behavior of the algorithm is to begin with a point x and determine points which fall inside a ball of radius τ around it. These points are marked to be in the same cluster as x. We will call them the *neighbors* of x. Next we consider balls around each of these neighbors in succession. Points which have been expanded once are not redone. In this way we iteratively build up the cluster. At some point we will run out of new points to expand and thus have found the complete cluster. The algorithm proceeds to a new unvisited point and starts building the cluster which contains it. We will tolerate some errors in this process. Namely we are sure to include points which are within radius τ, but may also admit some points which are only within radius $(1+\epsilon)\tau$. Points which fall outside the ball of radius $(1+\epsilon)\tau$ will not be included. Because of this, we will never overestimate the bottleneck distance between two points (by incorrectly identifying them to belong to separate clusters), but may underestimate it by a factor of $1+\epsilon$. Note that this means the clustering found will be ϵ-close.

Assume that two points x and y are each in θ-cells (if either of them is a singleton point in a larger cell in the tree the distance approximation will be more accurate, as we know at least one of their locations exactly). We will estimate the distance between them by the distance between the centers of the θ-cells. Call this distance l. If l is no more than some threshold L, we will say that x and y are clustered.

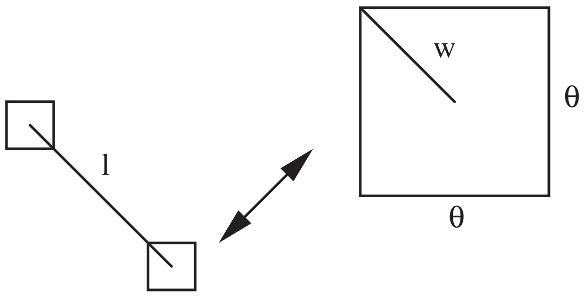

Figure 1: When are two θ-cells clustered?

We can derive L and θ as follows. Consider the case when the point set is planar (the analysis for higher dimensions is completely analogous). Let w be the maximum distance from the center of a θ-cell to another point in the same cell (this will be a corner). It is easy to see from Figure 1, that for $d = 2$, $w = \frac{\theta}{2\sqrt{2}}$. In general, for d dimensions $w = \frac{\theta}{2d^{1/d}}$. $|x, y| \leq l + 2w$ and $|x, y| \geq l - 2w$. Assume x and y are said to be clustered; that is $l \leq L$. We require the $|x, y| \leq (1 + \epsilon)\tau$. As l may equal L and $|x, y|$ may equal $l + 2w$, by substituting these values into the first inequality we have

$$L + 2w \leq (1 + \epsilon)\tau. \tag{1}$$

A similar analysis in the case where x are said to be in different clusters shows that

$$L - 2w \geq \tau. \tag{2}$$

Adding these two inequalities gives $w \leq \frac{\epsilon\tau}{4}$. So we have the restriction $\theta \leq \frac{1}{2}d^{1/d}\epsilon\tau$ in for our choice of θ. Cells in the quad tree have sidelengths of the form $2^{-z}\Delta$, where z is a nonnegative integer. We will choose $\theta = \max\{2^{-z}\Delta : 2^{-z}\Delta \leq \frac{1}{2}d^{1/d}\epsilon\tau$ and z is a nonnegative integer $\}$. Substituting this value for θ back into the two inequalities yields $\tau + \frac{\theta}{d^{1/d}} \leq L \leq (1 + \epsilon)\tau - \frac{\theta}{d^{1/d}}$. Any L in this range can be chosen.

A first observation is that we will make the most progress in finding the entire cluster if we choose to expand the neighbors of a given point x which are furthest away from x first. This way new territory (area not currently included in the cluster under construction) will be examined more quickly. This is advantageous as it is possible to reduce the cost of expanding points which are completely inside the cluster. Thus the sooner the cluster is expanded the better, as we may spend less effort processing the interior points. Another observation is that it is possible to mark off entire regions of points as being in the same cluster by marking internal nodes of the quad tree. When a box of the quad tree is discovered to belong to the cluster being built we can check to see if all its siblings also belong to the cluster, and if so mark the parent node as belonging to the cluster too. We these ideas in mind we present the second phase of the algorithm fully.

The cluster finding phase of the algorithm sweeps through the leaf nodes of the quad tree, and at each one whose cluster has not already been identified proceeds to construct it in full. When constructing the cluster, a coarse *search mesh* of boxes is erected over the region of the cluster which is being determined. The search mesh is used to guide the search for new cells of the tree which may merge into the cluster. This mesh is coarse in the sense that the box sidelengths are proportional to τ, not θ. In particular we choose the sidelength, which will call ϕ, of the mesh boxes from the range $\frac{1}{2}\alpha\tau$ to $\alpha\tau$. We will choose it so that it equals a box sidelength present in the quad tree. $\alpha \in (0, 1)$ is a fixed constant which presumably can be optimized, but we have not done so yet. We also align the search mesh so that mesh boxes align exactly with boxes of the that size in the quad tree. Of course not every box in the search mesh will actually be found in the quad tree, because not every mesh box will contain a point. Mesh boxes which do not contain any points are marked as being empty.

The algorithm also maintains a FIFO queue of boxes in the mesh which we term the *exterior queue*. Initially it contains the a single mesh box containing the starting leaf node (or the singleton point if the leaf is not a θ-cell). The general approach is to remove a box from the boundary queue and expand a shell around it. We will call the box the *active box*, and the shell the *include shell*. The include shell will be just close enough to the active box so that if another box falls completely inside the shell, we know that any points it contains must be in the same cluster. This means the distance from any point on the include shell to the *furthest* point (a corner) in the active box will be exactly L. We also define another shell around the active box called the *exclude shell.* It will be defined so that any mesh box which falls completely outside will be known to be too far away to be directly clustered with a point from the active box. Thus the distance from any point on the exclude shell to the *nearest* point on the active box will be L. There will be a certain number of boxes which are neither completely inside the include shell, nor completely outside the exclude shell. We will term these boxes *ambivalent.*

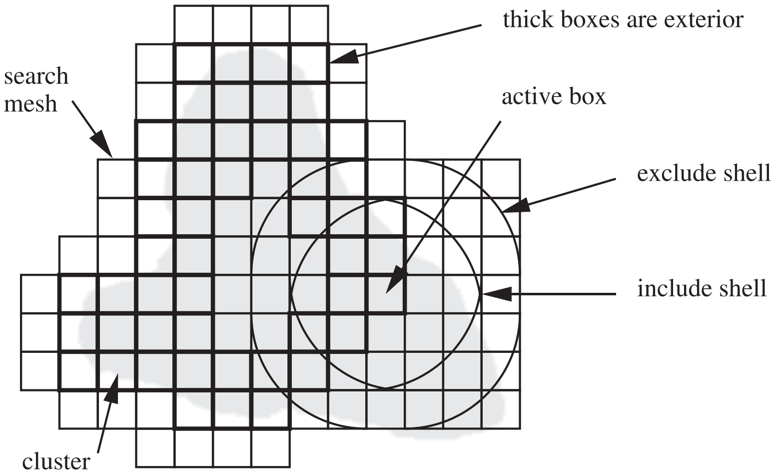

Figure 2: The search mesh

When the shells expand into new territory, the the search mesh is expanded to cover it. It is easy to verify that there are no more than $\lceil 2/\alpha + 1 \rceil^d$ boxes in the exclude shell by counting the number of search meshs cells in a box containing the exclude shell. Ambivalent boxes must be further checked to see if they also can be included in the cluster. There are only $O(1)$ boxes to check. (This constant will be exponential in $d-1$, but we leave the analysis to a future version of this paper.) This checking will be done by a subroutine which will calculate the minimum distance between a non-empty θ-cell from the ambivalent box and a non-empty θ-cell from the active box, up to the tolerated error. We describe this subroutine in §2.1. We will show that if there are p points in the ambivalent box and q

points in the active box, this can be done in $O((p+q)\lg(\frac{1}{\epsilon}))$ time. If the distance computed is not more than L, then the ambivalent box is marked as belonging to the cluster. If the distance is more than L, then we cannot directly add the ambivalent box to the cluster. When all the appropriate mesh boxes have been marked we will have expanded the search-mesh out to include a new region of the cluster. All the boxes which are on the boundary of this region are added into the exterior queue provided they have not been expanded before. A box is on the *boundary* if it is marked as belonging to the cluster, but incident with a box in the search mesh which is not. For reasons which be discussed later we will also consider boxes which are marked and adjacent to a boundary box as *exterior*. The remaining marked boxes are said to be *interior* to the cluster. Any boxes which are on the exterior queue but are now interior are removed. As we will see, there is no point in expanding a shell around them as no new territory can be discovered. When the boundary queue is empty the cluster has been completely determined and this information is propagated down to the leaf-nodes in each marked cell. If any unidentified leaf cells remain the above process is repeated to determine the clusters in which they are located.

In point form, the approximate algorithm is:

1. Construct a θ-resolved quad tree.

2. Start with any leaf cell in the quad tree. Find an active box containing the leaf cell and erect a search mesh out from it, far enough to enclose the exclude shell.

3. All non-empty mesh boxes which fall completely inside the include shell are marked as belonging to the cluster. A subroutine is invoked for those mesh boxes which are ambivalent towards the active box to determine if any of them may also be directly added to the cluster.

4. There will now potentially be some new cells in the seach mesh which have been identified as belonging to the cluster. Any new exterior boxes are added to the exterior queue and any boxes on the exterior queue which are now interior to the cluster are removed.

5. If the exterior queue is non-empty remove the first element and make it the active box; go to step three. If it is empty, the cluster has been completely found. Propagate this information to each leaf cell in the cluster.

6. If any unidentified leaf cells remain in the quad tree, pick one and go to step two. Otherwise stop.

2.1 The Box-Box Subroutine

We invoke the box-box subroutine whenever a non-empty ambivalent box is encountered when expanding an active box. These two boxes in the search mesh are separated by approximately τ. We would like to find out the distance of the closest pair of θ-cells, where the first one comes from the active box and the second comes from the ambivalent box. We will adapt a standard algorithm from computational geometry for the planar case. This method finds the closest pair of points between two sets which can separated by a straight line in $O(m \lg m)$ time, where m is the total number of points in both sets. It is based on first computing a Voronoi diagram for the point sets and then "walking" along the edges of the Voronoi diagram to find the minimum distance pair. See Exercise 11 in §5.8 of [10] for more details.

We have yet to determine the best way to solve this problem in dimensions greater than two. The box-box separation problem is a restricted version of the *closest bichromatic pairs* *problem*, which has been previously studied [5, 2]. Unfortunately, to the authors knowledge, none of its solutions can be adapted to solve the box-box problem and still maintain the same asymptotic time bound. It is possible, using a method from [5] for the box-box subroutine, to give an $O(\lg(\frac{\Delta}{\epsilon\tau})n + \sqrt{\frac{1}{\epsilon}}(\lg(\frac{1}{\epsilon}))^{1.5}n)$ time approximate cluster-finding algorithm for three dimensions. We omit the details. It is not clear that this is the best approach to take for three dimensions. On the surface it seems that the box-box separation problem is much simpler than the the general bichromatic pairs problem, so perhaps a better solution will be found. We leave this exploration to a future version of the paper.

The total number of points in a mesh box is bounded by the ratio of the volume of the mesh box to the volume of a θ-cell. This is is bounded above by $(\frac{K}{\epsilon})^d$ for some K (which depends on α). Thus for two dimensions, $\lg m = O(\lg(\frac{1}{\epsilon}))$, and the total running time of any invocation of the subroutine is $O(m \lg(\frac{1}{\epsilon}))$.

2.2 Proof of Correctness

To establish that the above algorithm is correct we need to prove that the clustering it produces satisfies the definition of ϵ-close. We will accomplish this with three lemmas.

LEMMA 2.1. *If $|x, y| < \tau$ then x and y will be identified as belonging to the same cluster.*

Proof. At some point the mesh box containing one of the points (say x) will be considered as part of cluster; this is guaranteed as the algorithm does not stop until every point is identified as belonging to some cluster. There are two cases to consider. Either the mesh box containing x (let us call it X) is the active box at some point in the execution of the algorithm or not. If it is, there is nothing to show because the box containing y (call it Y) will be discovered to belong to the same cluster as x during the shell expansion phase; either directly or using the box-box subroutine. If X is never the active box, it means that X is in the interior of the cluster. We need to argue that there is no need to expand shells around X to discover Y. To see this, consider the straight line s between x and y. s intersects a number of mesh boxes, starting with X and ending with Y. Let these boxes be, in order along s, $B_1 = X, B_2, \ldots, B_k = Y$. Let $r = \max\{i : B_i \text{ is in the same cluster as } X\}$. As X is interior, $r > 2$. Since B_r is on the exterior of the cluster, we know that at some point it will be the active box. We rely on Figure 3 to convince the reader that Y will be discovered as belonging to the same cluster as X when this occurs.

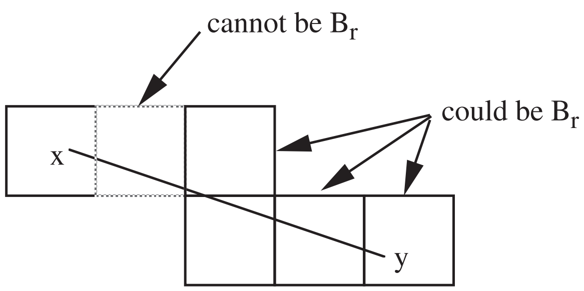

Figure 3: x and y will be clustered

LEMMA 2.2. $\forall x, y \; \delta_b(x, y) \leq \tau \;\; \Rightarrow \;\; x, y$ *will be identified as belonging to the same cluster.*

Proof. Assume $\delta_b(x, y) \leq \tau$. There will be a path $P = (p_1 = x, p_2, p_3, \ldots, p_k = y)$ of points from the point set such that $|p_i, p_{i+1}| \leq \tau$ for $i = 1, \ldots, k-1$. By the previous lemma, adjacent points in P will be identified as belonging to the same cluster. Thus x and y are identified as belonging to the same cluster.

LEMMA 2.3. $\forall x, y \; \delta_b(x, y) > (1+\epsilon)\tau \;\; \Rightarrow \;\; x, y$ *will be identified as belonging to different clusters.*

Proof. Let x and y be two points such that $\delta_b(x, y) > (1+\epsilon)\tau$. Consider the mesh boxes containing x and y. Call them X and Y as above. Assume that the cluster containing

X is constructed first. The only way new points are added to the cluster is by expanding shells around exterior points and marking new boxes in the search mesh as belonging to the cluster. The only two ways a new box is marked is by either falling completely inside the include shell or by the box-box subroutine. In either case the bottleneck distance between any pair of points where the first point is from the active box and the second is from the newly marked box is guaranteed to be at most $(1+\epsilon)\tau$. Thus (by induction) all new points added into the cluster are within $(1+\epsilon)\tau$ of x (under the δ_b-metric). So if y were added into the cluster containing x, we would have $\delta_b(x, y) \leq (1+\epsilon)\tau$. Contradiction.

The last two lemmas proof that the clustering found will be ϵ-close.

2.3 Proof of Time bound

In this section we give a proof that the above algorithm runs in $O(\lg(\frac{\Delta}{\epsilon\tau})n)$ time. We have already argued that the time to build the quad tree is $O(\lg(\frac{\Delta}{\epsilon\tau})n)$, so we will now account for the second phase.

Once the quad tree is built the algorithm performs a single scan through all the leaf cells. At every unmarked leaf cell this scan is interrupted to find the cluster containing that leaf. As there are at most n leaf cells in the tree, the scan will require $O(n)$ time. We need to account for the costs of finding each cluster. The general approach will be to "charge" points for any work the algorithm does. This will let us give a total bound on the amount of work done by summing over the points, the work charged to each of them.

When we find a new leaf cell we need to walk up the tree until we reach a node whose sidelength is ϕ. (If the leaf is a singleton point in a larger box, not as much work will be necessary.) This will take $\lceil \lg(\frac{\phi}{\theta}) \rceil = O(\lg(\frac{1}{\epsilon}))$ time. After we have found this mesh cell we need to determine the search mesh out far enough to include all the boxes which intersect the exclude shell. In general we will have do this only for cells on the boundary of the cluster, but we will assume this cost applies to all non-empty boxes in the search-mesh. As we have seen there will be $O(1)$ such boxes. (This constant will be exponential in the dimension, d of the point set.) The time to find any one of them is $O(\lg(\frac{\Delta}{\tau}))$, as in general it will be necessary to walk up the tree as far a the root and then back down to nodes of size ϕ. Hence the total cost for finding the mesh box and constructing a shell around it will be $O(\lg(\frac{\Delta}{\tau}) + \lg(\frac{1}{\epsilon}))$ time.

In addition to the cost of being found and a constructing a shell, a mesh box will also incur the cost of determining which cluster it belongs to and then passing this information on to every point that it contains. The cost of passing the information on will just be $O(m \lg(\frac{1}{\epsilon}))$, where m is the number of points in the mesh box. A mesh box will be determined to belong to a given cluster in to different ways. It is either found to be inside the include shell of some active box, or is found to be close enough to some active box via the box-box subroutine. The worst case occurs when box is surrounded by boxes just slightly too far away to be merged in. In this case, the box-box subroutine can be invoked for all $O(1)$ of them. (This constant will be exponential in $d-1$.) The cost of each invocation is $O((m+q)\lg(\frac{1}{\epsilon}))$ time, where q is the number of points in the other box. We distribute this cost proportionately between the two boxes. Hence the box with m points is charged $O(m \lg(\frac{1}{\epsilon}))$ time in total. Thus the largest total cost a given mesh box will ever sustain in determining its cluster is $O(m \lg(\frac{1}{\epsilon}))$ time.

We can now add up all the costs and give a bound on the amount of time spent on each point. First there is the cost of finding the shell. We overestimate this cost at $O(\lg(\frac{\Delta}{\tau}) + \lg(\frac{1}{\epsilon}))$ time per point. Next there is the cost of deterimining the cluster that each point is in. This is done by finding the cluster of the mesh box containing the point.

From above, this costs $O(\lg(\frac{1}{\epsilon}))$ time per point. Thus the total cost of the second phase of the algorithm is $O(\lg(\frac{\Delta}{\tau}) + \lg(\frac{1}{\epsilon}))$ time per point. The total cost of both phases of the algorithm is $O((\lg(\frac{\Delta}{\epsilon\tau}) + \lg(\frac{\Delta}{\tau}) + \lg(\frac{1}{\epsilon}))n)$ time. This simplifies to $O(\lg(\frac{\Delta}{\epsilon\tau})n)$ time.

3 Lower Bound for the Exact Problem

To prove the lower bound for the exact problem, we will make use of Ben-Or's theorem [8] on the cost of solving geometric problems in the algebraic tree model of computation. A program in this model consists of a three-way tree, where each internal node has an associated multinomial and the computation proceeds deterministically to one of the child nodes depending on whether the multinomial evaluated on the input is less than, equal to, or greater than zero. Each leaf node has one label; either accept or reject. We state the theorem below, but the reader is refered to [8] and [10] for the full details. Note that there is no dependence on the degree of the multinomials, so this is quite a strong computation model and thus good for proving lower bounds in.

THEOREM 3.1. (BEN-OR) *: Let W be a set in the cartesian space E^n and let T be an algebraic decision tree that solves the membership problem in W. If h is the depth of T and $\#(W)$ is the number of disjoint connected components of W, then $h = \Omega(\lg \#(W) - n)$.*

Let $W = \{(x_1, \ldots, x_n) : x_1, \ldots, x_n \in R$ and form n distinct 1-clusters$\}$. Clearly if we had a cluster-finder we could decide membership in W in by spending an additional $O(n)$ time. So any superlinear lower bound about deciding membership in W will also apply to cluster-finding in general.

LEMMA 3.1. *Let $\vec{x}, \vec{y} \in W$. Then $\vec{x}$ and $\vec{y}$ are connected in $W \iff \forall i, j\ x_i < x_j \Leftrightarrow y_i < y_j$.*

Proof. "$\Leftarrow$" direction: Assume $\forall i, j\ x_i < x_j \Leftrightarrow y_i < y_j$. $\exists$ a permutation π such that $x_{\pi(1)} < x_{\pi(2)} < \ldots < x_{\pi(n)}$ and $y_{\pi(1)} < y_{\pi(2)} < \ldots < y_{\pi(n)}$. The parametrized path $P = P(t) = t\vec{y} + (1-t)\vec{x},\ t \in [0,1]$ connects $\vec{x}$ and $\vec{y}$. ($P(0) = \vec{x}$ and $P(1) = \vec{y}$.) We need to check that P remains in W as t varies from 0 to 1. Let $\vec{p} \in P$. $p_{\pi(i)} = ty_{\pi(i)} + (1-t)x_{\pi(i)} < ty_{\pi(i+1)} + (1-t)x_{\pi(i+1)} = p_{\pi(i+1)}$. It follows that there will be n distinct clusters at each point $\vec{p} \in P$ provided that the gap between $p_{\pi(i)}$ and $p_{\pi(i+1)}$ is never less than 1. $|p_{\pi(i+1)} - p_{\pi(i)}| = t(x_{\pi(i+1)} - x_{\pi(i)}) + (1-t)(y_{\pi(i+1)} - y_{\pi(i)}) \geq t(1) + (1-t)(1) = 1$. So P provides a path connecting $\vec{x}$ and $\vec{y}$ in W.

"$\Rightarrow$" direction: Assume $\vec{x}$ and $\vec{y}$ are connected in W. For the sake of obtaining a contradiction, assume $\exists i, j$ such that $x_i < x_j$ but $y_i > y_j$ (note $y_i = y_j$ is ruled out by $\vec{y} \in W$). Consider the hyperplane in R^n, $z_i = z_j$. Clearly this is disjoint from W. But $\vec{x}$ lies on one side, and $\vec{y}$ lies on the other, so they are must be in disconnected portions of W.

THEOREM 3.2. *Deciding membership in W takes $\Omega(n \lg n)$ time.*

Proof. We show that W has $n!$ disjoint connected components. As $n! > (n/2)^{n/2}$, direct application of Ben-Or's theorem gives an $\Omega(n \lg n)$ lower bound. From the above lemma $W = \bigcup_{\pi \in S_n} W_\pi$ (S_n is the symmetric group on n elements), where $W_\pi = \{(x_1, \ldots, x_n) : x_1, \ldots, x_n \in R$, form n distinct clusters and $x_{\pi(1)} < \ldots < x_{\pi(n)}\}$. Furthermore the W_π are pairwise disconnected (by the lemma) and so W has exactly $|S_n| = n!$ connected components.

4 Efficiency

In this section we give a rough comparison between the approximate algorithm we have developed and some alternative approaches for determining clusters in a large, but relatively

Algorithm	$d = 2$	$d = 3$
exact EMST	$O(n \lg n)$	$O((n \lg n)^{1.5})$
approximate EMST	$O(n \lg n + 33n)$	$O(n \lg n + 332n)$
approximate cluster	$O(17n)$	$O(124n)$

Table 1: Running times for typical parameter values.

typical astrophysical simulation. This comparison will be made modulo the big-O notation which hides constants. It is our expectation that the constant for our approximate algorithm should be smaller than those of the other methods presented. We also mention some preliminary implementation results.

It has been communicated to the author that a typical parameters for a large astrophysical cluster problem might be $\Delta = 10^8$ parsecs, $\tau = 10^3$ parsecs, and $\epsilon = 10\%$ [14]. The results of plugging these values into the prefactors of the best known[2] exact EMST, approximate EMST, and our new approximate algorithm are shown in Table 1.

It appears from Table 1 the new approximate cluster finding algorithm should outperform other methods. Currently it has not been implemented so we have do not have concrete evidence of its practicality. An earlier algorithm in a similar vein to the one presented here (although merge-based and for three dimensional point sets) was implemented and showed excellent performance. On a workstation, a speed-up of around 30 was achieved over an algorithm based on building an EMST. This was on a million point set.

5 Conclusions and Open Problems

It should be straightforward to parallelize the approximate algorithm, but this has not been done yet. A good method for doing the box-box subroutine needs to be worked out for higher dimensions. Here good might just mean fast expected time performance, rather than

[2] To the author, at least.

a worst case upper bound. In addition to a basic serial implementation, an efficient parallel version should also be done as it is expected this algorithm will be used in conjunction with a parallel particle simulator.

References

[1] P. Callahan and S. Kosaraju. *Faster Algorithms for Some Geometric Graph Problems in Higher Dimensions*. Proceedings of the 1993 ACM/SIAM Symposium on Discrete Algorithms. January, 1993. Austin, Texas.

[2] N. Katoh and K. Iwano. *Finding k Farthest Pairs and k Closest/Farthest Bichromatic Pairs for Points in the Plane*. Proceedings of the 8th ACM Symposium on Computational Geometry. 1992.

[3] M. Shamos and D. Hoey. *Closest-point problems*. Proceedings of the 6th IEEE Symposium on Foundations of Computer Science. 1975.

[4] A. Yao. *On constructing minimum spanning trees in k-dimensional spaces and related problems*. SIAM Journal on Computing. 11 : 721-736. 1982.

[5] P. Agarwal, H. Edelsbrunner, O. Schwarzkopf and E. Welzl. *Euclidean Minimum Spanning Trees and Bichromatic Closest Pairs*. Discrete and Computational Geometry 6:407-422. 1991.

[6] P. Vaidya. *An $O(n \log n)$ Algorithm for the All-Nearest Neighbors Problem*. Discrete and Computational Geometry 4:101-115. 1989.

[7] R. Tarjan and J. van Leeuwen. *Worst-Case Analysis of Set Union Algorithms*. Journal of the ACM. 1984.

[8] M. Ben-Or. *Lower bounds for algebraic computation trees.* Proceedings of the 15th ACM Annual Symposium on Theory of Computation. pp. 80-86. May 1983.

[9] T. Cormen, C. Leiserson and R. Rivest. *Introduction to Algorithms.* McGraw-Hill. 1991.

[10] F. Preparata, M. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag. 1985.

[11] M. Dickerson and R. Drysdale. *Enumerating k distances for n points in the plane.* Proceedings of the 8th Annual ACM Symposium on Computational Geometry, 234-238, 1991.

[12] J. Barnes and P. Hut. *A hierarchical $O(n \log n)$ force-calculation algorithm.* Nature, 324, 446, 1986.

[13] J. Salmon and M. Warren. *Skeletons from the Treecode Closet.* Submitted to Journal of Computational Physics, July 31, 1992.

[14] R. Carlberg. University of Toronto. *Personal communication.* October, 1993.