Tree-Weighted Neighbors and Geometric k Smallest Spanning Trees

David Eppstein

Department of Information and Computer Science
University of California, Irvine, CA 92717

Tech. Report 92-77

July 7, 1992

Abstract

We compute the k smallest spanning trees of a point set in the planar Euclidean metric in time $O(n \log n \log k + k \min(k,n)^{1/2} \log(k/n))$, and in the rectilinear metrics in time $O(n \log n + n \log\log n \log k + k \min(k,n)^{1/2} \log(k/n))$. In three or four dimensions our time bound is $O(n^{4/3+\epsilon} + k \min(k,n)^{1/2} \log(k/n))$, and in higher dimensions the bound is $O(n^{2-2/(\lceil d/2 \rceil+1)+\epsilon} + kn^{1/2} \log n)$.

1 Introduction

The k smallest spanning tree problem for graphs has been studied extensively [5, 6, 7, 8, 9, 10, 11, 12], but it was only recently that the author introduced the corresponding geometric problem [7]. In this problem, one is given a point set as input, and one must construct k different spanning trees that have the minimum total edge lengths among all possible spanning trees of the set. The trees need not be edge-disjoint. This problem is to be distinguished from the much harder one of finding the k smallest distinct spanning tree weights [12].

Since the geometric problem is equivalent to a graph problem on the complete graph of all the points, weighted by distance, the various known graph algorithms can all be applied to the geometric problem. The best known graph algorithm takes $O(m \log \beta(m,n) + k \min(k,n)^{1/2} \log(\min(k,m)/n))$ time [8]. For the complete graph, $m = \Theta(n^2)$ and this bound can be simplified to $O(n^2 + kn^{1/2} \log n)$.

This bound holds for any metric space. In order to improve it, we need to apply some geometry. From now on, we assume the Euclidean metric in the plane, unless stated otherwise. In our earlier paper [7], we showed that the k smallest spanning trees could be found in a graph with $O(kn)$ edges, constructed using the order k Voronoi diagram. The time for finding the spanning trees is dominated by that for finding the diagram, which is $O(kn^{1+\epsilon})$ [2] or $O(k^2 n + n \log n)$ [4].

In the same paper, we gave a different approach, which is closer to the ideas we apply here. We first used the order 3 Voronoi diagram to identify $O(k)$ edges of the minimum spanning tree which might be replaced in the other $k-1$ smallest trees. These edges split the MST into $O(k)$ blocks, and we found the nearest neighbor of each point in each block. We used this information to identify $O(k)$ points which might occur as endpoints of edges in the smallest spanning trees, other than those edges already in the MST. Then we solved the graph problem on the graph formed by adding $O(k^2)$ possible edges to the $n-1$ MST edges. The time for this procedure is $O(k^2 + kn \log(n/k))$.[1] This is always an improvement on the $O(kn^{1+\epsilon})$ Voronoi-diagram-based algorithm.

Since our paper appeared, Frederickson [9] and Eppstein et al. [8] improved the algorithm for general graphs, to the bounds stated above. However, their improvements only help the non-geometric algorithm for our

[1]We adopt the convention that $\log x = \log_2(2+x)$, so $\log x$ is always $\Omega(1)$ even if $x < 1$.

problem (valid for any metric space) giving the $O(n^2 + kn^{1/2} \log n)$ bound stated above. It seems natural to hope that their approach could also improve the times for the geometric (Voronoi diagram based) algorithms of [7], but this required new ideas, since those algorithms spend more time constructing graphs than they spend finding spanning trees.

In this paper, we provide such an improvement. As in the second algorithm of our previous paper, we find $O(k)$ endpoints of spanning tree replacement edges; however we speed up this stage by computing the *tree-weighted nearest neighbor* of each point; that is, we alter the standard Euclidean metric by subtracting from each distance $d(x, y)$ the length of the longest edge on the MST path from x to y, and compute nearest neighbors with these altered distances. We then thin down the $O(k^2)$ possible edges to a set of $O(k)$ edges, in time $O(k^{1+\epsilon})$, using a modification of the tree-weighted neighbor idea. As a result, we find the k smallest spanning trees in time $O(n \log n \log k + k \min(k, n)^{1/2} \log(k/n))$

Our algorithm improves our previous geometric algorithms for values of k in the range $(\log n \log \log n)^{1/2} < k < n$, and it is the first to apply geometry to improve the previous purely graph theoretic algorithms in the range $n < k < n^2$. Our algorithm is nearly optimal for $k < n^{2/3}$, avoiding optimality only by a factor of $O(\log k)$. For larger k, our algorithm's time is dominated by that for finding the k smallest spanning trees in graphs; thus we are poised to take advantage of any further development in the graph algorithms.

We also adapt our algorithm to the rectilinear (L_1 or L_∞) metric. In this metric, the time is $O(n \log n + n \log \log n \log k + k \min(k, n)^{1/2} \log(k/n))$, which always improves the previous $O(n \log n + kn \log \log(n/k))$ bound of our previous paper.

Finally, we discuss generalizations of our algorithm to higher dimensions. Our previous geometric algorithms found the spanning trees in sparse graphs constructed using Voronoi diagrams, but in higher dimensions the corresponding graphs may be dense, so those algorithms do not generalize. Our new algorithms rely primarily on the ability to search for several nearest neighbors in a point set, for which data structures are known in any dimension. Thus we find the first geometric algorithms for k smallest spanning trees in any dimension. Our algorithm takes time $O(n^{4/3+\epsilon} + k \min(k, n)^{1/2} \log(k/n))$ in three and four dimensions. In any higher dimension d the bound is $O(n^{2-2/(\lceil d/2 \rceil+1)+\epsilon} + kn^{1/2} \log n)$. For small k these bounds match the best known times for computing a single minimum spanning tree [1]. For large k the time is dominated by that for finding k smallest spanning trees in graphs, and is poised to take advantage of improvements in the graph algorithms.

2 Tree-weighted neighbors

Suppose we are given a point set S, and an edge-weighted tree T spanning the points. For example, the tree could be the minimum spanning tree of the point set. Define the *tree-weighted distance* $d_T(x,y)$ as follows. Given points x and y, there is some path in T connecting them. Let e be the edge of maximum weight on that path. Then $d_T(x,y) = d(x,y) - |e|$. The tree-weighted distance "normalizes" the Euclidean distance by effectively pushing closer together pairs of points connected by a long edge in T. However note that after this weighting, the distance need no longer satisfy the triangle inequality, so it will not necessarily be a metric. If T is the MST, the tree-weighted distance $d(x,y)$ measures how much the length of the spanning tree would increase if we forced xy to be an edge in the tree.

The *nearest tree-weighted neighbor* of a point x is then simply that other point y minimizing the tree-weighted distance $d_T(x,y)$. Again, if T is the MST, this would seem to be the same as the best way of adding an edge with x as endpoint to the MST, to get another tree with as small a weight as possible. However, in this case the points connected by MST edges to x will have tree-weighted distance 0, and other points will in general have positive distance, so if x and y are nearest neighbors xy will already be an MST edge. We will see how to avoid this difficulty, and use tree-weighted neighbors for computing the k smallest spanning trees, later. For now we will simply compute the neighbors.

Given any tree T, if we remove any point x the remaining edges form a graph with several connected components; let $L(x)$ be the number of points in the largest such component. If $L(x) > n/2$, let y be the neighbor of x in the single component with size $L(x)$; then $L(y) \leq L(x) - 1$. Therefore we can always find a *centerpoint* $c(T)$ such that $L(c(T)) \leq n/2$. We can compute $L(x)$ for each point x, and hence find a centerpoint, in linear time.

Our algorithm uses centerpoints to perform a divide and conquer recursion on the tree T. We first find a centerpoint $c(T)$ for the entire tree. This partitions the tree into components of size at most $n/2$. We merge pairs of small components until we have at most three components, still of size at most $n/2$. Then for each point x we find the nearest tree-weighted neighbor y, only among those points for which path xy crosses the centerpoint. After this is done we recursively examine each component of $T-c(T)$, and compute nearest neighbors in the corresponding point sets. The nearest neighbor to x overall will either be y or it will be the neighbor to x in the component of $T-c(T)$ that contains x. At each stage of the recursion the size of the tree decreases by half. Therefore after $O(\log n)$ stages of recursion we will have computed the nearest neighbor for each point.

At each stage, we must find the nearest neighbor y to x on paths xy that cross c. There are two cases. In the first case, the longest edge on path xy is also on path xc, that is, on the portion of xy between x and c. Let e be that longest edge. Then for all points y not in the component of x, $d_T(x,y) \leq d(x,y) - |e|$, with equality exactly when e is longer than any edge on path cy. If we let y be the nearest neighbor to x in the unweighted Euclidean metric, then $d(x,y) - |e|$ is at least as large as the distance from x to its nearest tree-weighted neighbor, with equality exactly when y is that neighbor. Each such neighbor can be found in time $O(n \log n)$ using point location in a Voronoi diagram.

In the second case, the longest edge on path xy is also on path cy. For each possible point y, let e_y be the longest edge on path cy. Then for all points y not in the component of x, $d_T(x,y) \leq d(x,y) - |e_y|$, with equality exactly when e_y is longer than any edge on path xc. If we let y be the nearest neighbor to x in the weighted Euclidean metric $d'(x,y) = d(x,y) - |e_y|$, then $d(x,y) - |e_y|$ is at least as large as the distance from x to its nearest tree-weighted neighbor, with equality exactly when y is that neighbor. Each such neighbor can be found in time $O(n \log n)$ using point location in an additively weighted Voronoi diagram.

Thus after constructing two Voronoi diagrams and performing point location in each one, in time $O(n \log n)$ we have two candidates for the nearest neighbor to x, among those points across c from x. We can compare the two and choose the true nearest neighbor in constant time per point. After at most three repetitions of this algorithm (for the three components of $T - c$), we have computed all neighbors at this level of the recursion. This completes the proof of the following result.

Lemma 1. *Given a set of n points, and an edge-weighted tree on the points, we can compute the nearest tree-weighted neighbor of each point in time $O(n \log^2 n)$.* $\square$

In certain circumstances we can save a further factor in our time bound. Suppose that of the weights on the edges in T, most are $-\infty$, while the remaining k edges have positive weight. Then the nearest neighbor of any point will be across some positively weighted edge. We can find a modified center $c'(T)$ such that each component of $T - c'$ contains at most $k/2$ positive edges. If we use this modified center to perform a divide and conquer as above, we only perform $O(\log k)$ stages. Each stage takes $O(n \log n)$ time as above. Therefore we get the following result.

Lemma 2. *Given a set of n points, and an edge-weighted tree T on the points, such that k of the edges in T have positive weight and the remaining $n - k$ have weight $-\infty$, we can compute the nearest tree-weighted neighbor of each point in time $O(n \log n \log k)$.* □

Finally, we state a version of our results for the rectilinear (L_1 or L_∞) metric. In our previous paper [7], we showed how to perform point location in a Voronoi diagram for this metric in time $O(n \log \log n)$, if we know the sorted orders of the points by their x coordinates, and by their y coordinates. The same algorithm applies without modification to the additively weighted Voronoi diagram. The sorted orders can be computed once at the start of the algorithm, and then maintained in linear time as we split the point sets at each level of the recursion. This gives us the following result.

Lemma 3. *Given a set of n points with the planar rectilinear metric, and given an edge-weighted tree T on the points, such that k of the edges in T have positive weight and the remaining $n - k$ have weight $-\infty$, we can compute the nearest tree-weighted neighbor of each point in time $O(n \log n + n \log \log n \log k)$.* □

3 Finding replacable edges

As hinted above, all but k edges of the MST are forced to remain in all of the k smallest spanning trees. We now discuss how to find such a set of k edges efficiently. The results of this section are repeated from our previous paper [7], and so are stated without proof here.

Lemma 4. *Let G be any graph, with minimum spanning tree T, and let xy be an edge in T. Then the minimum spanning tree of $G - xy$ can be found by removing xy from T, and adding the shortest edge in $G - T$ that reconnects the two components left when xy was removed. All such replacement edges can be found in time $O(m + n \log n)$.* □

Lemma 5. *Let S be a planar point set, and let uv be the replacement edge for MST edge xy. Then if the circle with uv as diameter contains any other input point, that point is one of x and y, and either u or v is the other of x and y. Therefore all possible replacement edges can be found in time $O(n \log n)$, using the order 3 Voronoi diagram.* □

Lemma 6. *Let S be a planar point set, with MST T, and for every edge e in T let $r(e)$ be its replacement computed as in Lemma 4. Each edge gives rise to a different tree $T - e + r(e)$, and we can calculate in linear time the trees in this set having the k smallest weights. If $T - e + r(e)$ is not among the $k-1$ smallest trees of this form, then e must be an edge in each of the k smallest spanning trees of the point set. Therefore we can compute in time $O(n \log n)$ a set of $n-k$ MST edges that must be in all k smallest spanning trees. The remaining $k-1$ edges may or may not be replaced in some of the k smallest spanning trees.* □

4 Finding endpoints of replacement edges

Continuing to follow our previous algorithm, we now find a set P of $O(k)$ points such that any edge in one of the k smallest spanning trees, other than MST edges, has both endpoints in P. In our previous algorithm, we did this in $O(kn \log(n/k))$ time using $O(k)$ nearest neighbor searches per point, one for each replacable edge in the MST. We now show how to speed this up using the idea of tree-weighted nearest neighbors.

The idea is simple: We find the MST of the point set, and a set of $k-1$ MST edges that may be replaced, as above. We include all endpoints of these $k-1$ edges in P. We weight the MST edges by their length, if the edge is in the set of $k-1$ edges, or by $-\infty$, if the edge is not in this set. We then find the tree-weighted nearest neighbor of each point. Among those points not already in P, we choose the $2k$ with the smallest tree-weighted distances to their nearest neighbors.

In this way we find a set of at most $4k$ points, in time $O(n \log n \log k)$. We now show that this set has the property we wish, that every non-MST edge in one of the k smallest spanning trees has both endpoints in P.

Lemma 7. *Let xy be an edge in one of the k smallest spanning trees, that is not an edge in the MST. Then x is in P.*

Proof: Let u be one of the $2k$ points in P that is not one of the endpoints of the $k-1$ replacable MST edges. Let v be the tree-weighted nearest neighbor of u. Since u is not an endpoint of a replacable edge, all MST edges adjacent to u have weight $-\infty$, and so v is not adjacent to u in the tree. If we add uv to the MST, and remove the longest edge on the MST path from u to v, we find another spanning tree with weight differing from that of the MST by the tree-weighted distance $d_T(u, v)$. The $2k$ points each give such a tree, and each such tree can be found in at most two ways, one for each endpoint of the new edge, so we have a set of at least k trees.

If one of the k smallest spanning trees contains xy, then the smallest weight spanning tree containing xy must be one of the k smallest spanning trees. That tree is found by adding xy to the MST, and removing the longest edge on the MST path from x to y. If this is to be one of the k smallest spanning trees, the removed edge must be a removable edge, and so the weight of the tree differs from that of the MST by $d_T(x, y)$. Since this is one of the k smallest spanning trees, this weight must be within the weights of the k trees described above. Therefore, if x is not an endpoint of a removable edge (which is automatically in P), $d_T(x, y)$ must be at most the $2k$ smallest value $d_T(u, v)$, among pairs (u, v) for which v is the nearest neighbor of u. If z is the nearest neighbor to x, then $d_T(x, z) \leq d_T(x, y)$ and z is also at most the $2k$ smallest value $d_T(u, v)$. Therefore x will be selected as one of the points in P. $\square$

5 Finding replacement edges

At this point we could simply find the k smallest spanning trees in the graph consisting of the MST, together with all $O(k^2)$ edges between points in the set P constructed above. This would give a total time of $O(n \log n \log k + k^2)$, which would be an improvement on previously known algorithms. However if we wish to apply the $O(k \min(k, n)^{1/2} \log(k/n)$ time graph algorithm, the time will be dominated by that for simply enumerating all $O(k^2)$ replacement edges. To further speed this up, we must prune this set of replacement edges to a smaller set which still contains all the replacements that actually occur in the k smallest spanning trees.

Our basic strategy is to follow our previous algorithm for tree-weighted neighbors, to find several neighbors for each point. After including the corresponding edges in our set of replacements, we will then be able to reduce the size of our set of endpoints, and repeat the process finding even more neighbors per point.

There are two main differences between our algorithm here and our algorithm for finding single tree-weighted nearest neighbors. First, since we have already eliminated all but $O(k)$ points as candidates for replacement edge endpoints, we can perform the nearest neighbor search on a set of $O(k)$ points instead of our original n point set.

Second, our tree-weighted nearest neighbor algorithm performed two kinds of searches, one in a normal Voronoi diagram and one in an additively weighted Voronoi diagram. Each potential edge replacement would be considered twice, once per each endpoint, using each of the two types of search. Normal Voronoi diagrams can be generalized to higher order Voronoi diagrams in order to find several nearest neighbors. However we know of no similar results for additively weighted Voronoi diagrams. Since our original algorithm considers each edge once each way, we can ignore the portion of the algorithm that searches additively weighted diagrams, and only use normal Voronoi diagrams. However this will cause our algorithm, as it progresses in stages and reduces the potential number of replacement edge endpoints, to perform searches at this smaller number of endpoints, but still search in a diagram formed by the original set of $O(k)$ potential endpoints. As the number of neighbors we wish to find goes up, the time to construct this diagram would also increase. To avoid this increase, we use a data structure of Agarwal and Matoušek [3], which can be constructed in time $O(k^{1+\epsilon})$ and which allows efficient searches for any number of nearest neighbors.

Consider a potential replacement edge xy. In the divide-and-conquer structure of the tree-weighted nearest neighbor algorithm, x and y will remain in the same set for several levels of the recursion, until they are separated by some point c that is used to split the tree. Let e_{xy} be the longest replacable edge on path xc, and e_{yx} symmetrically be the longest replacable edge on path cy. The longer of these two edges gives us the true tree-weighted distance $d_T(x, y)$ and the smallest tree including edge xy. Instead of comparing these two and finding the single best replacement for edge xy, we will treat these two cases as two distinct possible replacements, and find $O(k)$ ordered pairs (x, y) minimizing the replacement weight.

We define a directed tree-weighted distance $d'_T(x, y) = d(x, y) - |e_{xy}|$. Each ordered pair (x, y) gives rise to a tree T_{xy} formed by adding xy to the MST and removing e_{xy}; $d'_T(x, y)$ measures the weight difference between this tree and the MST. If xy is one of the $O(k)$ replacable MST edges, T_{xy} will be the MST itself; otherwise each tree T_{xy} will be different from all other such trees. It follows that if xy is an edge in one of the k smallest spanning trees, $d'_T(x, y)$ must be one of the k smallest values among edges other than the replacable MST edges, and therefore must be one of the $2k$ smallest values over all.

As mentioned above, our algorithm proceeds in stages. At each stage i, we find the $2k$ smallest values of $d'_T(x, y)$, among those y that are among the 2^i nearest neighbors to x. This gives us a set of $2k$ edges which are passed as input to the next stage. At each stage, some x will have all their 2^i nearest neighbors in this set of $2k$ edges, and some will be the endpoint of fewer edges. We call points in the latter class *eliminated.* If the 2^i nearest neighbors of x are not all in the smallest $2k$ among the edges examined so far, no further away neighbors can be in the smallest $2k$ overall, so once a point is eliminated we need no longer generate any further edges from that endpoint.

Each uneliminated point contributes 2^i towards a total of $2k$ edges, so there can be at most $k2^{1-i}$ uneliminated points. For each such point, we compute the 2^{i+1} nearest neighbors, measuring distance by d'_T, among the $O(k)$ points of set P. This gives us a set of $O(k)$ potential replacement edges, among which we again compute the $2k$ smallest values. This set is passed as input to stage $(i + 1)$, and we continue until stage $\log k$ at which point we will have found $2k$ replacement edges among all possible directed edge replacements involving the $4k$ points in P.

Thus the computation of replacement edges reduces to finding nearest neighbors with distance function d'_T. Let m denote the number of neighbors we are searching for. As in our original tree-weighted neighbor algorithm, we proceed in a divide and conquer fashion down the tree (the same divide and conquer that produces the centerpoints used to define edges e_{xy} and e_{yx}, and thus defines d'_T itself). At each level of the tree, we find for each point x the m nearest neighbors among those points across c from x. This gives us $O(m \log k)$ neighbors, among which we can use a linear time selection algorithm to find the m nearest.

To find the m nearest neighbors among a set of p points, we use a data structure of Agarwal and Matoušek [3], which can be constructed in time $O(p^{1+\epsilon})$, and which finds m nearest neighbors to any query point in time $O(\log^3 p + m \log^2 p)$. This data structure can be constructed once for each level of the divide and conquer tree, and need not be reconstructed for each stage of the algorithm outlined above. The total time for constructing the data structure is then $O(k^{1+\epsilon})$, and the total time per stage querying the data structure is $O(\log^4 k + m \log^3 k)$ per point for each of the $O(k2^{1-i})$

query points. Adding the time for all stages gives $O(k \log^4 k)$ for all queries.

This dominates the time spent comparing edge replacement weights, which is $O(k)$ per stage for $O(k \log k)$ overall, and the time spent selecting m nearest neighbors out of $O(m \log k)$ potential neighbors, which is $O(k \log k)$ per stage for $O(k \log^2 k)$ overall. Thus we have the following result.

Lemma 8. *Given a set P of $O(k)$ candidate replacement edge endpoints constructed as in the previous section, we can find in time $O(k^{1+\epsilon})$ a set of $O(k)$ edges such that any non-MST edge in one of the k smallest spanning trees must belong to this set.* $\square$

This technique still applies even when $k > n$, in which case P consists of the whole input set. In this case the time bound can be tightened to $O(\min(n,k)^{1+\epsilon} + k \log^4 k)$, and with some further care (finding candidate edges in weighted order using a priority queue) to $O(\min(n,k)^{1+\epsilon} + k \log^2 k)$. However such an improvement would not be useful until faster algorithms are developed for the graph k smallest spanning trees problem.

At this point we can apply the graph algorithm of Eppstein et al. [8] on the graph consisting of the $n-1$ MST edges and the $O(k)$ candidate replacement edges. Combining the bounds in the various lemmas above gives our main results:

Theorem 1. *We can find the k smallest spanning trees of a planar point set in time $O(n \log n \log k + k \min(k,n)^{1/2} \log(k/n))$ for the Euclidean metric, or $O(n \log n + n \log \log n \log k + k \min(k,n)^{1/2} \log(k/n))$ for the rectilinear metric.*

6 Higher dimensions

We now discuss generalizations of our algorithms to higher dimensions. This was not possible for our previous geometric k smallest spanning trees algorithms, because they were based on the use of Voronoi diagrams, which may give rise to dense graphs in dimensions as low as three. Therefore any Voronoi-diagram-based algorithm could be no better in the worst case than the pure graph algorithm on the complete graph of the input points.

For similar reasons, Lemma 6 is not useful to us, so we will not easily be able to reduce the number of replacable edges from n to $O(k)$.

The next part of our algorithm, reducing the number of potential endpoints of replacement edges from n to a set P of $O(k)$ points, is again not available to us. In higher dimensions, we lack an efficient data structure for performing nearest neighbor searches in the additively weighted Euclidean metric, so Lemma 2 is no longer available to us.

The remaining step in our algorithm is reducing the set of replacement edges to a set of $O(k)$ such edges, and this is the step we generalize. The algorithm here depends only on the existence of a data structure for finding several nearest neighbors of points in the Euclidean metric. Agarwal and Matoušek [3] provide such a generalization of their data structure. In three or four dimensions, if one is searching for sets of m nearest neighbors among a set of p points, their data structure (with the appropriate choice of parameters) takes time $O(p^{4/3+\epsilon})$ to initialize, and answers queries in time $O(p^{1/3+\epsilon} + m \log^2 p)$. In higher dimensions d, the preprocessing time is $O(p^{2-2/(\lceil d/2 \rceil+1)+\epsilon})$ and the query time is $O(p^{1-2/(\lceil d/2 \rceil+1)+\epsilon} + m \log^2 p)$.

Since we no longer know a set of $O(k)$ replacable edges, we must assume that all edges are replacable. Therefore at each stage of the algorithm, we retain the $n + k$ ordered pairs giving the best swaps, instead of the $2k$ such pairs. At most n of these pairs will involve swapping an edge for itself, so the remaining k swaps will give distinct spanning trees.

At each stage of the algorithm, and each level of the divide and conquer recursion on the minimum spanning tree, we perform queries from at most n points, asking for their nearest neighbors among sets of at most n points. The total number of neighbors asked for never exceeds $2(n + k)$. Thus the time per level is $O(n^{2-2/(\lceil d/2 \rceil+1)+\epsilon} + k \log^2 n)$. The total time for the algorithm is $O(n^{2-2/(\lceil d/2 \rceil+1)+\epsilon} + k \log^4 n)$ (as before, this could be reduced to $O(k \log^2 n)$ if required). After we have performed this stage of the algorithm, we have a set of $n + k$ possible replacement edges which, together with the $n-1$ edges in the original MST, give a graph with $O(n+k)$ edges. Applying the graph k smallest spanning trees algorithm of Eppstein et al. [8] to this graph leads to the following result.

Theorem 2. *We can find the k smallest spanning trees of a 3- or 4-dimensional point set in time $O(n^{4/3+\epsilon} + k \min(k, n)^{1/2} \log(k/n))$, and of a d-dimensional set in time $O(n^{2-2/(\lceil d/2 \rceil+1)+\epsilon} + kn^{1/2} \log n)$.*

References

[1] P.K. Agarwal, H. Edelsbrunner, O. Schwarzkopf, and E. Welzl. Euclidean minimum spanning trees and bichromatic closest pairs. 6th ACM Symp. Comput. Geom. (1990) 203–210.

[2] P.K. Agarwal, D. Eppstein, and J. Matoušek. Dynamic algorithms for half-space reporting, proximity problems, and geometric minimum spanning trees. 33rd IEEE Symp. Foundations of Computer Science (1992) to appear.

[3] P.K. Agarwal and J. Matoušek. Ray shooting and parametric search. 24th ACM Symp. Theory of Computing (1992) 517–526.

[4] A. Aggarwal, L.J. Guibas, J. Saxe, and P.W. Shor, A linear time algorithm for computing the Voronoi diagram of a convex polygon, 19th ACM Symp. Theory of Computing (1987) 39–47.

[5] R.N. Burns and C.E. Haff. A ranking problem in graphs. 5th Southeast Conf. Combinatorics, Graph Theory and Computing 19 (1974) 461–470.

[6] P.M. Camerini, L. Fratta, and F. Maffioli. The k shortest spanning trees of a graph. Int. Rep. 73-10, IEEE-LCE Politechnico di Milano, Italy (1974).

[7] D. Eppstein. Finding the k smallest spanning trees. 2nd Scand. Worksh. Algorithm Theory, Bergen, Norway, 1990. Springer LNCS 447 (1990) 38–47, and *BIT* 32 (1992) 237–248.

[8] D. Eppstein, Z. Galil, G.F. Italiano, and A. Nissenzweig. Sparsification—A technique for speeding up dynamic graph algorithms. 33rd IEEE Symp. Foundations of Computer Science (1992) to appear.

[9] G.N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. 32nd IEEE Symp. Foundations of Computer Science (1991) 632–641.

[10] H.N. Gabow. Two algorithms for generating weighted spanning trees in order. *SIAM J. Comput.* 6 (1977) 139–150.

[11] N. Katoh, T. Ibaraki, and H. Mine. An algorithm for finding k minimum spanning trees. *SIAM J. Comput.* 10 (1981) 247–255.

[12] E.W. Mayr and C.G. Plaxton. On the spanning trees of weighted graphs. Manuscript, 1990.