New Techniques for Some Dynamic Closest-Point and Farthest-Point Problems

Kenneth J. Supowit*

Abstract. We present techniques for solving geometric closest-point and farthest-point query problems, in the presence of deletions. Applications include efficient implementations of classical greedy heuristics for minimum-weight matching (where our result improves on that of Bentley and Saxe), and maximum-weight matching.

1 Introduction

This paper contains two new techniques for solving dynamic query problems in Euclidean space, when deletions are allowed. One is for closest-point query problems, whose applications include greedy minimum-weight matching and perhaps the traveling salesman and minimum spanning tree problems (see the "Other problems" section below). The other is for farthest-point problems whose applications include the analogous problems such as greedy maximum-weight matching. For the closest-point problems, the idea is essentially a dynamic version of multidimensional divide-and-conquer [Sh], [B1], [B2].

For the problem of minimum-weight matching in the Euclidean plane, the greedy heuristic iteratively matches a closest pair, deletes those two points, matches a closest pair among the remaining points, and so forth until all points are matched. The behavior of this heuristic (i.e., the quality of the solution produced) has received much study, for worst-case [RT], [A1], [SS] as well as expected performance [AD]. To implement the greedy heuristic, Bentley and Saxe [BS] describe an $O(n^{3/2} \log n)$ time algorithm. In the current paper, we present an $O(n(\log n)^2)$ time algorithm. Furthermore, the technique of [BS] is based on Voronoi Diagrams and hence does not generalize well to higher dimensions; ours extends to $O(n(\log n)^d)$ time for fixed dimension d. The fastest

* Dept. of Computer & Info. Science, Ohio State University, Columbus, Ohio 43210. This research was supported in part by the National Science Foundation grant number DMC-8451214.

algorithm known to solve the problem optimally requires $O(n^{5/2}(\log n)^4)$ time in the plane [V1], and $O(n^{3-1/c^d})$ in higher dimensions [V2], for some constant $c > 1$. Many other heuristics have been studied for Euclidean and more general weighted matching; the field is surveyed in [A2], where the complexity of implementing the greedy heuristic for the plane is stated as one of the open problems.

The greedy heuristic for the maximum-weight matching problem analogously matches a farthest pair, deletes those two points, matches a farthest pair among the remaining points, and so forth. The weight of the matching produced is at least half of the weight of the maximum matching, as is shown in [A2]. We obtain an $O(n^{3/2}\log n)$ time implementation for this heuristic, in the plane.

We describe the data structure for closest-point problems, together with its application to greedy minimum-weight matching in the plane, in some detail. Its extension to higher dimensions, as well as the farthest-point technique and its application to maximum-weight matching, is only sketched. For each of these algorithms, S is the given set of points, $n = |S|$.

2 Minimum-weight matching

We begin with a review of the divide-and-conquer closest pair algorithm [Sh]. The median x-value of S, denoted $\mathrm{m}(S)$, is found and S is partitioned into sets $A = \{p \in S : \mathrm{x}(p) \le \mathrm{m}(S)\}$ and $B = \{p \in S : \mathrm{x}(p) > \mathrm{m}(S)\}$. The problem is solved recursively for A and for B, letting $\delta(A)$ and $\delta(B)$ be the distance of a closest pair in A and B, respectively. The set

$$L(S) = \{p \in S : |\mathrm{x}(p) - \mathrm{m}(S)| < \delta\}$$

is constructed, where $\delta = \min\{\delta(A), \delta(B)\}$. Intuitively, $L(S)$ consists of all points of S within a vertical strip (which we call the *center strip of* S) of width 2δ. We call the vertical line through $\mathrm{m}(S)$ the *center line* of S. A closest pair ab such that $a \in A \cap L(S)$ and $b \in B \cap L(S)$ is found. If $\mathrm{dist}(a,b) < \delta$ then ab is a closest pair in S; otherwise the closer of the two pairs found in the recursive calls is a closest pair in S. The key to the efficiency of the algorithm is that the center strip is sparse; in particular, each point $p \in L(S)$ is within vertical distance δ of at most $O(1)$ other points in $L(S)$. Thus we may simply scan $L(S)$ by increasing y-value, checking for each point in $L(S)$ the distance to just a few of its predecessors and successors in the scan.

Our algorithm for dynamic closest pair (with deletions) initially performs the closest pair algorithm as stated above, while saving the tree of subsets $S' \subseteq S$ obtained in the recursive calls, along with the values $\mathrm{m}(S')$, $\delta(S')$ and $L(S')$. In particular, the center strip $L(S')$ is saved as a balanced tree (e.g. an AVL tree), ordered by y-value. We use two additional data structures:

1. For each S' in the tree, a heap (also known as priority queue) of pairs:

$$Heap(S') = \{ab : a \in A \cap L(S'), b \in B \cap L(S') \text{ and } a \text{ and } b \text{ are within 6 positions of each other in } L(S')\},$$

ordered by dist(a,b).

2. A heap consisting of the minima of all the heaps:

$$Winners = \{\min(Heap(S')) : S' \text{ is a node in the recursion tree}\}$$

also ordered by dist(a,b).

We alternately use the symbol S' to refer to a node in the recursion tree and to the corresponding subset of S.

When a point p is deleted, we visit all nodes S' that contain it, updating all relevant data structures. Note that $\delta(S')$ may grow, but may never shrink; hence a point $p \in S'$ may be brought into $L(S')$ at most once.

Thus the greedy matching is obtained by iteratively outputting the minimum pair pq from the *Winners* heap, and then deleting p and q and updating all the data structures accordingly. The algorithm is shown as Figure 1.

The correctness of the algorithm follows from the simple observation that the closest pair pq among S must be the minimum of some *Heap*; in particular, pq is the minimum of $Heap(S')$ where S' is the least common ancestor of $\{p\}$ and $\{q\}$ in the recursion tree.

We now sketch the proof of the algorithm's complexity. Each point $p \in S$ is inserted into $O(\log n)$ sets $L(S')$, and is eventually deleted from it. A particular p can be inserted into a particular $L(S')$ at most once because the widths (the values δ) used in defining the center strips grow but do not shrink. When p is inserted into S', it causes $O(1)$ insertions into $Heap(S')$. Thus, the algorithm uses $O(n \log^2 n)$ time and $O(n \log n)$ storage. Of course, for each $p \in S$ we must maintain a list of pointers to entries for p in the various $L(S')$ as well as to all pairs containing p in the various $Heap(S')$. Furthermore, we must pre-sort by the x-dimension, so that no extraneous points a are examined in line (14).

For dimensions $d \geq 3$, our algorithm uses Bentley's multidimensional divide-and-conquer idea [B1], [B2], dynamicized with data structures that generalize those outlined above for the plane. In particular, one of the data structures we use is a tree (of height d) each of whose nodes corresponds to a heap, whose entries are the minima of its sons' heaps. The result is an $O(n(\log n)^d)$ greedy matching algorithm.

```
(1) W ← the empty heap;
(2) Initialize the data structures ;
(3) FOR i ← 1 TO n/2 DO
(4)     BEGIN
(5)         pq ⇐ Winners ;
(6)         [[ delete p ]]
(7)             S' ← father({p}) ;
(8)             WHILE S' ≠ S DO
(9)                 BEGIN
(10)                    Delete each pair containing p from Heap(S') ;
(11)                    Delete p from L(S') if it's there ;
(12)                    A ← leftson(S') ;      B ← rightson(S') ;
(13)                    δ ← min{δ(A), δ(B)} ;
(14)                    FOR each a ∈ A − L(S') s.t. m(S') − δ ≤ x(a) DO
(15)                        BEGIN
(16)                            Insert a into L(S') ;
(17)                            FOR each b ∈ B ∩ L(S')
                                        s.t. b is within 6 positions of a in L(S') DO
(18)                                Insert ab into Heap(S')
(19)                        END ;
(20)                    FOR each b ∈ B − L(S') s.t. x(b) ≤ m(S') + δ DO
(21)                        analogously
(22)                    IF Heap(S') is empty THEN δ(S') ← δ
(23)                                         ELSE δ(S') ← min{δ, min(Heap(S'))} ;
(24)                    Update Winners if min(Heap(S')) has changed ;
(25)                    S' ← father(S')
(26)                END ;

(27)        [[ delete q ]]
(28)            analogously
(29)    END.
```

Fig. 1
The algorithm for greedy minimum-weight matching.

3 Maximum-weight matching

Although this problem's statement is similar to that of minimum-weight matching, there are some fundamental differences between nearest-point and farthest-point problems. Perhaps chief among them is that

$$|\{q \in S : p = NN(q)\}| = O(1)$$

whereas

$$|\{q \in S : p = FN(q)\}| = \Theta(n)$$

in the worst case, where $NN(q)$ and $FN(q)$ denote q's nearest and farthest neighbor in S, respectively. Therefore the technique of [BS] that simply partitions S into $\sqrt{n}$ search structures (in this case, farthest-point Voronoi Diagrams [PS]) does not work, because each deletion might require us to search for the new farthest neighbor of $\Theta(n)$ points. Point sets can be constructed for which this unfortunate situation occurs $\Theta(n)$ times during the greedy maximum matching. Therefore we cannot afford to keep a data structure that stores the farthest neighbor of each point, because such an approach would require $\Omega(n^2)$ in the worst case.

Our strategy, rather, is to maintain the farthest neighbor of only certain special points that we call "boundary points". A point p is a *boundary point* if it has at least two antipodal neighbors (q is said to be p's *antipodal neighbor* if there exist parallel lines ℓ_1 and ℓ_2 that contain p and q, respectively, such that all points of S lie between ℓ_1 and ℓ_2 [PS]). Now it may turn out that $\Theta(n)$ points are boundary points, but that does not hurt in the analysis. The key facts (easily shown) for our algorithm are:

1. For each $p \in S$, p has at most two antipodal neighbors that are boundary points.

2. If ab are a farthest pair of S then either a or b is a boundary point.

We assume in this discussion that no four points form a trapezoid; such a situation slightly complicates the algorithm, but does not increase its complexity. By Fact 1, we need update only $O(1)$ points for each deletion, which is important in the complexity analysis. Fact 2 implies that this strategy indeed works; i.e. despite limiting our knowledge of farthest neighbors to the boundary points, we still can find a farthest pair at each iteration.

Our algorithm maintains:

(i) the convex hull of S by the method of [Ov],

(ii) the set of boundary points p by a new method,

(iii) a heap of pairs $(p, FN(p))$ for boundary points p, ordered by maximum distance,

(iv) about $\sqrt{n}$ farthest-point diagrams (in the manner of [BS]) whose sites are points on the convex hull.

The total time turns out to be $O(n^{3/2} \log n)$.

4 Other problems

Consider the greedy heuristic for the traveling salesman problem that starts at an arbitrarily chosen point p_1, visits its closest neighbor p_2, then visits p_2's closest neighbor $p_3 \in S - \{p_1\}$, and in general visits the closest neighbor of the point most recently visited among all those points not yet visited. This simple heuristic has been very widely used; its worst-case behavior was studied in [RS].

The greedy traveling salesman heuristic may be viewed as iteratively finding the nearest neighbor q of the current point p, deleting p, and making q the current point. Thus it is a special case of the all-nearest-neighbors dynamic query problem, with deletions.

The ideas in [BS] can easily be adapted to obtain an $O(n^{3/2}\log n)$ time implementation of this TSP heuristic in the plane; perhaps some modification of our technique for dynamic closest pair would yield a more efficient algorithm here.

The table below summarizes the applications of various dynamic query problems to greedy algorithms for some classical optimization problems.

The minimum spanning tree problem is very interesting in this context, because for it various greedy algorithms actually give the optimal solution. The MST in the plane can be solved in $O(n \log n)$ time using a Voronoi Diagram, whereas in higher dimensions d the fastest known is almost $O(n^2)$, namely $O(n^{2-1/2^{d+1}}(\log n)^{1-1/2^{d+1}})$, making it still very much an open problem [Ya] [PS]. Implementing Prim's algorithm for MST may be viewed as a special case of a bipartite closest pair problem. That is, we have a partition of S into set A (initially empty) and B. At each iteration we find the closest pair ab such that $a \in A, b \in B$ and delete b from B and insert it into A. The difficulty in applying our technique for dynamic closest pair problems here is that the sparsity of the center strip no longer holds, because of the bipartiteness. Perhaps a weaker sparsity condition can be found which would be adequate to obtain a faster algorithm for higher dimensions. Such a result would be perhaps the most significant outcome from this line of research.

Another problem that almosts fit in here is minimum-weight triangulation, where the greedy algorithm may be viewed as a closest pair problem in which instead of deletions, certain obstacles are dynamically inserted which block off certain edges. Could some variation of our technique work for this problem?

Query Problem	Operations	Application
closest pair	deletions	greedy min. matching
diameter	deletions	greedy max. matching
all-nearest-neighbor	deletions	greedy TSP
bipartite closest pair	insertions and deletions	MST

REFERENCES

[A1] D. Avis, "Worst-case bounds for the Euclidean matching problem," *Comput. Math. Appl.*, Vol. 7 (1981), pp. 251–257.

[A2] D. Avis, "A survey of heuristics for the weighted matching problem," *Networks*, Vol. 13 (1983), pp.475–493.

[AD] D. Avis, B. Davis and J. M. Steele, "Probabilistic analysis of a greedy heuristic for Euclidean matching," *Probability in the Engineering and Informational Sciences*, Vol. 2 (1988), pp. 143–156.

[B1] J. L. Bentley, "Divide and conquer algorithms for closest point problems in multidimensional space," Ph. D. dissertation, Univ. of North Carolina, Chapel Hill, NC, 1976.

[B2] J. L. Bentley, "Multidimensional Divide-and-Conquer," *Comm. of the ACM*, Vol. 23, No. 4 (April 1980), pp. 214–229.

[BS] J. L. Bentley and J. B. Saxe, "Decomposable searching problems 1: Static to dynamic transformations," *J. Algorithms*, Vol. 1 (1980), pp. 301–358.

[Ov] M. H. Overmars and J. van Leeuwen, "Maintenance of configurations in the plane," *J. Comput. and Syst. Sci.*, Vol. 23 (1981), pp. 166–204.

[Pa] C. H. Papadimitriou, "The probabilistic analysis of matching heuristics," *Proc. 15th Allerton Conference on Communication, Control and Computing* (1977), pp. 368–378.

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

[RT] E. M. Reingold and R. E. Tarjan, "On a greedy heuristic for complete matching," *SIAM J. Comput.*, Vol. 10 (1981) pp. 676–681.

[RS] D. J. Rosenkrantz, R. E. Stearns and P. M. Lewis, "An analysis of several heuristics for the traveling salesman problem," *SIAM J. Comput.*, Vol. 6 (1977), pp. 563–581.

[Sh] M. I. Shamos, "Geometric complexity," *Proc. 7th ACM Symp. Theory of Comput.* (1975), pp. 224-233.

[SS] T. L. Snyder and J. M. Steele, "Worst case greedy matchings in the unit d-cube," *submitted for publication.*

[V1] P. Vaidya, "Geometry help in matching," *Proc. 20th ACM Symp. on Theory of Comput.* (1988), pp. 422–425.

[V2] P. Vaidya, personal communication.

[Ya], A. C. Yao, "On constructing minimum spanning trees in k-dimensional space and related problems," *SIAM J. Comput.*, Vol. 11, No. 4 (1982), pp. 721–736.