SIAM J. COMPUT.
Vol. 18, No. 6, pp. 1201-1225, December 1989

GEOMETRY HELPS IN MATCHING*

PRAVIN M. VAIDYA†

Abstract. A set of $2n$ points on the plane induces a complete weighted undirected graph as follows. The points are the vertices of the graph, and the weight of an edge between any two points is the distance between the points under some metric. The problem of finding a minimum weight complete matching (MWCM) in such a graph is studied. An $O(n^{2.5}(\log n)^4)$ algorithm is given for finding an MWCM in such a graph, for the L_1 (*manhattan*), the L_2 (*Euclidean*), and the L_∞ metrics. The bipartite version of the problem is also studied, where half the points are painted with one color and the other half with another color, and the restriction is that a point of one color may be matched only to a point of another color. An $O(n^{2.5} \log n)$ algorithm for the bipartite version, for the L_1, L_2, and L_∞ metrics, is presented. The running time for the bipartite version can be further improved to $O(n^2(\log n)^3)$ for the L_1 and L_∞ metrics.

Key words. weighted matching, computational geometry, optimization

AMS(MOS) subject classifications. 68Q20, 68Q25, 68U05

1. Introduction.

Given a complete weighted undirected graph on a set of $2n$ vertices, a complete matching is a set of n edges such that each vertex has exactly one edge incident on it. The weight of a set of edges is the sum of the weights of the edges in the set, and a minimum weight complete matching (MWCM) is a complete matching that has the least weight among all the complete matchings.

We study the problem of finding an MWCM in the complete graph induced by a set of $2n$ points on the plane. The points are the vertices of the graph, and the weight of an edge between any two points is the distance between the points under some metric. We shall investigate two common metrics: the L_1 (*manhattan*) metric, and the L_2 (*Euclidean*) metric. (We note that the L_∞ metric can be converted to the L_1 metric by rotating the coordinate system by 45°, and so any algorithm for the L_1 metric can be trivially modified to work for the L_∞ metric.) The input consists of $2n$ points that specify the locations of the vertices on the plane. Each point p is given as an ordered pair (p_x, p_y), where p_x and p_y denote the x and y coordinates of p, respectively. The L_r distance between two points p and q is given by $(|p_x - q_x|^r + |p_y - q_y|^r)^{1/r}$. (Note that the L_1 distance between p and q is given by $|p_x - q_x| + |p_y - q_y|$.) We shall assume that the metric defining the edge weights is fixed.

We also study the bipartite version of the MWCM problem for points on the plane. In the bipartite version, half the points are painted with one color and the other half another color, and the restriction is that a point of one color can be matched only to a point of the other color.

The complete graph induced by a set of $2n$ points on the plane is entirely specified by the locations of the vertices. So the problem of finding an MWCM in such a graph differs from the problem of finding an MWCM in a general complete graph in that the size of the input is $O(n)$ rather than $\Omega(n^2)$. The input is sparse since the edge weights are implicitly defined by the underlying geometry. It is interesting to investigate if the geometric nature of the MWCM problem for points on the plane can be exploited

* Received by the editors November 30, 1987; accepted for publication (in revised form) December 5, 1988.

† Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 W. Springfield Avenue, Urbana, Illinois 61801. Present address, AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974.

to obtain an algorithm for its solution that is faster than the $\Theta(n^3)$ algorithm [6], [11] for general graphs. At this point we note that several heuristics for finding a good complete matching (not necessarily of minimum weight) on points on the plane have been developed [2], [9], [17], but the only known way to find an MWCM on $2n$ points on the plane is to run the MWCM algorithm for general graphs that requires $\Theta(n^3)$ time. In this paper we show that geometry does help to obtain a faster algorithm. We give an $O(n^{2.5}(\log n)^4)$ algorithm for finding an MWCM in the complete graph induced by a set of $2n$ points on the plane, for the L_1 and L_2 metrics. For the bipartite version of the MWCM problem for $2n$ points on the plane, we give an $O(n^{2.5} \log n)$ algorithm for the L_1 and L_2 metrics. For the bipartite case, the running time of the MWCM algorithm can be further improved to $O(n^2(\log n)^3)$ for the L_1 metric. The space requirement of all the algorithms is $O(n \log n)$ in the case of L_2 metric and $O(n(\log n)^2)$ in the case of L_1 metric.

The algorithms described in this paper will be essentially the well-studied primal-dual algorithms for weighted matching, namely, the Hungarian method [10], [11], [14] for bipartite matching, and Edmond's algorithm [4], [11], [14] for general matching. The primal-dual algorithms for weighted matching associate a dual variable with each vertex of the given graph, and the *slack* associated with an edge is the weight of the edge minus the sum of the dual variables associated with the end vertices of the edge. The algorithms can be substantially speeded up for points in the plane by the application of two key ideas. First, associating a weight with each vertex (point) that is suitably related to the dual variable corresponding to the vertex and that changes much less frequently than the dual variable, and implicity maintaining the dual variable using the weight. Second, reducing the computation of the minimum slack for certain subsets of edges to geometric query problems that involve the weights associated with the vertices and that can be efficiently solved using known data structures in computational geometry.

In § 2 we shall discuss some geometric query problems that arise naturally in the implementation of the primal-dual weighted matching algorithm for points on the plane, and see how known data structures in computational geometry can be used to solve them efficiently. In § 3 we shall give the algorithm for the bipartite version of the MWCM problem for points in the plane. The bipartite case is easier, and serves to illustrate the main ideas that are used in developing the algorithm for the general case. In § 4 we describe the algorithm for finding an MWCM in the complete graph induced by a set of points on the plane.

We shall assume a real RAM model of computation [15] standard in computational geometry, so arithmetic operations (i.e., addition, subtraction, multiplication, division), memory access operations, and comparison operations, on real numbers require constant time. (Actually, it is not necessary to assume that division requires constant time; the restricted assumption that division by two takes constant time is adequate.) For the case of the L_2 metric, we must make the additional assumption that either square roots can be computed in constant time (so that edge weights can be obtained in constant time) or that the edge weights have been precomputed and are available at the start of the algorithm.

Next, we introduce some notation and definitions.

A *priority queue* [1] is an abstract data structure consisting of a collection of elements, each element being associated with a real-valued *priority.* A priority queue supports the three operations—insert an element with some priority, delete an element, and find an element with the minimum priority—in time proportional to the logarithm of the number of elements in the priority queue.

We let $d(p, q)$ denote the distance between points p and q. (The metric under consideration will be clear from the context.) Let $H_x(a, b)$ denote the set of all points p on the plane such that $a \leqq p_x \leqq b$. Similarly, let $H_y(a, b)$ denote the set of all points p on the plane such that $a \leqq p_y \leqq b$. For a pair of real numbers a and b, $[a, b)$ denotes the set of all real numbers c such that $a \leqq c < b$. With respect to an ordered sequence $a_1 < a_2 < \cdots < a_m$, $[a_i, a_j)$ denotes the set of all a_k such that $i \leqq k < j$.

With respect to a set of points P such that there is a weight $w(p)$ associated with each point p in P, we define the following terms. For subsets P_1, P_2 of P, *shortest*$[P_1, P_2]$ denotes an edge (p_1^*, p_2^*), $p_1^* \in P_1$, $p_2^* \in P_2$, such that

$$d(p_1^*, p_2^*) - w(p_1^*) - w(p_2^*) = \min_{p_1 \in P_1,\, p_2 \in P_2} \{d(p_1, p_2) - w(p_1) - w(p_2)\}.$$

For a point q, *nearest*$[q, P]$ denotes a point $p^* \in P$ such that

$$d(q, p^*) - w(p^*) = \min_{p \in P} \{d(q, p) - w(p)\},$$

and *shortest*$[q, P]$ denotes the edge $(q, \textit{nearest}[q, P])$.

Finally, we shall use the term *vertex* when referring to a graph, and the term *node* when referring to a data structure.

2. Geometric query problems arising in matching on the plane. The primal-dual algorithms for weighted matching associate a dual variable with each vertex of the given graph, and the slack associated with an edge is the weight of the edge minus the sum of the dual variables associated with the end vertices of the edge. During the execution of the matching algorithm, we are repeatedly required to compute the minimum slack for certain subsets of edges. To perform this computation efficiently we shall need a good solution to the following query problems.

PROBLEM 1. Given a set of points P and a weight $w(p)$ for each point p in P, preprocess P so that for a given query point q, *nearest*$[q, [p_i, p_j)]$ can be found quickly.

PROBLEM 2. Given a set of points P, an ordering $p_1 < p_2 < \cdots < p_{|P|}$ of the points in P, and a weight $w(p)$ associated with each point p in P, preprocess P, so that given a query point q, and an interval $[p_i, p_j)$ such that $1 \leqq i < j \leqq |P| + 1$, *nearest*$[q, [p_i, p_j)]$ can be computed quickly.

In Problems 1 and 2 the set P is static. We shall also require a solution to the *semidynamic* version of Problems 1 and 2. In the semidynamic version, a new point can be added to P but a point can never be deleted from P. Furthermore, P is totally ordered by the following rule. For a pair of points $p, p' \in P$, $p < p'$ if and only if p was added to P before p'.

Problem 1 comes up in the bipartite case as well as the general case, and its solution enables us to quickly compute the minimum slack for various subsets of edges. Problem 2 and the semidynamic versions of the two problems arise because of certain subsets of vertices of odd cardinality, called blossoms, in Edmond's algorithm for general weighted matching. One type of blossom corresponds to intervals in some ordering on the set of vertices (points), and given a vertex q and a blossom B of this type, we are required to compute the minimum slack over all edges between q and vertices in B. This leads to Problem 2. The semidynamic versions arise because of blossoms merging to form bigger blossoms.

In §§ 2.1 and 2.2 we describe solutions to Problems 1 and 2, respectively. In § 2.3 we shall describe how to suitably modify the data structures for Problems 1 and 2 to handle the semidynamic case where new points may be inserted into P but no point may be deleted from P. The solutions use segment trees [15], a data structure common in computational geometry. So we shall first briefly discuss segment trees. The segment tree for the interval $[i, j)$ is a rooted binary tree defined as follows:

(1) $j > i+1$. The interval $[i, j)$ is associated with the root of the segment tree. The left subtree at the root is the segment tree for $[i, \lceil (i+j)/2 \rceil)$ and the right subtree at the root is the segment tree for $[\lceil (i+j)/2 \rceil, j)$.

(2) $j = i+1$. With the root is associated the interval $[i, i+1)$, and the left and right subtrees are both empty.

(3) $j \leqq i$. The segment tree is empty.

The segment tree has $O(\log(|j-i|))$ levels, and the disjoint union of all the intervals at a specific level in the tree is the interval $[i, j)$. Any subinterval of $[i, j)$ with integer endpoints can be expressed as the disjoint union of $O(\log(|j-i|))$ intervals in the segment tree. The segment tree data structure extends naturally to an ordered sequence $a_1 < a_2 < \cdots < a_m$ via the correspondence between the interval $[i, j)$ and the interval $[a_i, a_j)$.

2.1. Problem 1. For the case of the Euclidean (L_2) metric the weighted Voronoi diagram (WVD) [5], [16] of the points in P provides an adequate solution to Problem 1. Such a Voronoi diagram divides the plane into $|P|$ regions (some possibly empty), there being a region $Vor(p)$ for each point $p \in P$. $Vor(p)$ is the region given by

$$Vor(p) = \{p'' : \forall p' \in P, d(p'', p) - w(p) \leqq d(p'', p') - w(p')\}.$$

The WVD of P can be constructed in $O(|P| \log(|P|))$ time [5]. Furthermore, in $O(|P| \log(|P|))$ additional time it can be preprocessed, so that given a query point q, in $O(\log(|P|))$ time we can find a point $\bar{p}$ in P such that $q \in Vor(\bar{p})$ [3], [12]. So once the WVD of P is available, *nearest*$[q, P]$ can be obtained in $O(\log(|P|))$ time for any point q.

For the case of the L_1 metric we shall use the Willard-Lueker modification of the two-dimensional range tree [15] to provide a suitable solution to Problem 1. The range tree (RT) for P is as follows. At the top level is a segment tree for the nondecreasing sequence of x-coordinates of the points in P. At a segment tree node ψ associated with the interval $[a, b)$ of the x-axis is stored an ordered list of points in $P \cap H_x(a, b)$, with the points being ordered by y-coordinate. The segment tree also contains extra pointers from each internal node to its two children for efficient searching. The entire RT for P can be constructed in $O(|P| \log(|P|))$ time. To facilitate the search for *nearest*$[q, P]$ we shall store some additional information at each node. Let ψ be a node in the top level segment tree, and let $[a, b)$ be the interval associated with ψ. Let $p \in H_x(a, b) \cap P$. In addition to storing p in the ordered list at ψ, we store along with p the following points:

(1) *nearest*$[(a, p_y), P \cap H_x(a, b) \cap H_y(p_y, \infty)]$.

(2) *nearest*$[(a, p_y), P \cap H_x(a, b) \cap H_y(-\infty, p_y)]$.

(3) *nearest*$[(b, p_y), P \cap H_x(a, b) \cap H_y(p_y, \infty)]$.

(4) *nearest*$[(b, p_y), P \cap H_x(a, b) \cap H_y(-\infty, p_y)]$.

For a vertical strip $H_x(a, b)$ these additional points may be computed in $O(|P \cap H_x(a, b)|)$ time leading to a total of $O(|P| \log(|P|))$ for all the nodes in the tree. Suppose p, p' are adjacent points in the ordered list of points in $P \cap H_x(a, b)$ stored at node ψ. Also suppose that the query point q is such that $p_y \geqq q_y \geqq p'_y$. Then q satisfies the following two conditions:

(1) If $q_x \leqq a$ then one of the two points *nearest*$[(a, p_y), P \cap H_x(a, b) \cap H_y(p_y, \infty)]$, and *nearest*$[(a, p'_y), P \cap H_x(a, b) \cap H_y(-\infty, p'_y)]$, is *nearest*$[q, P \cap H_x(a, b)]$.

(2) If $q_x \geqq b$ then one of the two points $nearest[(b, p_y), P \cap H_x(a, b) \cap H_y(p_y, \infty)]$, and $nearest[(b, p'_y), P \cap H_x(a, b) \cap H_y(-\infty, p'_y)]$ is $nearest[q, P \cap H_x(a, b)]$.
So given a query point q, we visit the $O(\log(|P|))$ nodes in the segment tree corresponding to the decomposition of the two segments $[-\infty, q_x)$, and $[q_x, \infty)$, locate q in the ordered lists at these nodes, and compute $nearest[q, P]$ in $O(\log(|P|))$ time.

LEMMA 1. *Given a set of points P on the plane, and a weight $w(p)$ associated with each point p in P, P can be preprocessed in $O(|P| \log(|P|))$ time, so that given a query point q, $nearest[q, P]$ and $shortest[q, P]$ can be found in $O(\log(|P|))$ time.* □

2.2. Problem 2. The data structure for Problem 2 has two levels. At the top level is a segment tree for the ordered sequence $p_1 < p_2 < \cdots < p_{|P|}$ of the points in P. At a segment tree node associated with the interval $[p_k, p_l)$ is stored the WVD for the set $[p_k, p_l)$ in the case of L_2 metric, and the RT for the set $[p_k, p_l)$ in the case of L_1 metric. Given an interval $[p_i, p_j)$, we visit the $O(\log(|P|))$ nodes in the segment tree corresponding to the decomposition of $[p_i, p_j)$, and search the WVD/RT at each of these nodes, and thereby compute $nearest[q, [p_i, p_j)]$ in $O((\log(|P|))^2)$ time. The data structure may be easily constructed in $O(|P|(\log(|P|))^2)$ time.

LEMMA 2. *Let $P = \{p_1 < p_2 < \cdots < p_{|P|}\}$ be an ordered set of points on the plane, and let there be a weight $w(p)$ associated with each point p in P. P can be preprocessed in $O(|P|(\log(|P|))^2)$ time, so that given a query point q and an interval $[p_i, p_j)$ such that $1 \leqq i < j \leqq |P|+1$, $nearest[q, [p_i, p_j)]$ and $shortest[q, [p_i, p_j)]$ can be found in $O((\log(|P|))^2)$ time.* □

2.3. Dynamizing the data structures for insertion. In this section we shall describe how to dynamize the data structures for Problems 1 and 2 (described in §§ 2.1 and 2.2, respectively) to handle the semidynamic case where new points may be inserted into P, but no points may be deleted from P. P is totally ordered by the following rule. For a pair of points p, p' in P, $p < p'$ if and only if p was added to P before p'.

There are standard techniques [13] for dynamizing a static data structure to support insertion, and we shall briefly describe how to apply one of them to the weighted Voronoi diagram and the range tree.

Let

$$|P| = \sum_{0 \leqq i \leqq \log_2(|P|)} a_i 2^i, \qquad a_i \in \{0, 1\},$$

be the binary representation of $|P|$. Let $P_1, P_2, \cdots$ be a partition of P such that:

(1) $|P_i| = a_i 2^i, 0 \leqq i \leqq \log_2(|P|)$.

(2) If $i > j$ then each point in P_i was inserted into P before any of the points in P_j.

The *semidynamic* WVD (RT) is just a collection of WVDs (RTs), one for each nonempty P_i in the partition of P. Using the semidyanmic WVD (RT), given a query point q, $nearest[q, P]$ may be obtained $O((\log(|P|))^2)$ time. Furthermore, if we start with $P = \phi$ and there are a total of m insertions into P, then the total cost of maintaining the semidynamic WVD (RT) for P is $O(m(\log m)^2)$ operations [13].

The static data structure for Problem 2 described in § 2.2 may be dynamized to allow insertions into P in a similar manner. The dynamization increases the query time and the total time for all the insertions by a factor of at most $2\log_2(|P|)$ [13].

LEMMA 3. *Let P be a set of points such that new points may be added to P, but no point may be deleted from P. With each point $p \in P$ is associated a weight $w(p)$. Suppose in the beginning $P = \phi$, and in the end P contains m points.*

(1) *We can maintain a semidynamic* WVD/RT *for P such that*:

(1.1) *The total time for all the m insertions and thereby the total time for maintaining for the semidynamic* WVD/RT *is* $O(m(\log m)^2)$.

(1.2) *Given a query point q, nearest$[q, P]$ (and shortest$[q, P]$) can be found in* $O((\log m)^2)$ *time.*

(2) *Suppose P is totally ordered by the following rule. For a pair of points p, p' in P, $p < p'$ if and only if p was added to P before p'. Also, let $p_1 < p_2 < \cdots < p_{|P|}$ be the ordered sequence of the points in P. Then we can maintain a semidynamic data structure for P such that*:

(2.1) *The total time for inserting all the m points and thereby the total time for maintaining the semidynamic data structure is* $O(m(\log m)^3)$.

(2.2) *Given a query point q, and an interval $[p_i, p_j)$ such that $1 \leqq i < j \leqq |P|+1$, nearest$[q, [p_i, p_j)]$ (and shortest$[q, [p_i, p_j)]$) can be found in* $O((\log m)^3)$ *time.* □

3. Weighted bipartite matching on the plane. We are given two sets U and V each consisting of n points on the plane. U and V induce a complete bipartite graph whose vertices are the points in U and V, and the weight of an edge (u_i, v_j), $u_i \in U$, $v_j \in V$, is the distance between u_i and v_j under some metric. We consider two metrics, the L_1 metric and the L_2 metric. The problem is to find a minimum weight complete matching in the complete bipartite graph on U and V.

Let $u_1, \cdots, u_n$ be an enumeration of the vertices in U, and let $v_1, \cdots, v_n$ be an enumeration of the vertices in V. A linear programming formulation [11] of the above bipartite weighted matching problem is:

$$\min \sum_{i,j} d(u_i, v_j) x_{ij}$$

$$\text{subject to} \quad \sum_j x_{ij} = 1, \qquad i = 1, \cdots, n,$$

$$\sum_i x_{ij} = 1, \qquad j = 1, \cdots, n,$$

$$x_{ij} \geqq 0$$

with the understanding that (u_i, v_j) is in the matching X if and only if $x_{ij} = 1$. The dual linear program is

$$\max \sum_i \alpha_i + \sum_j \beta_j$$

$$\text{subject to} \quad \alpha_i + \beta_j \leqq d(u_i, v_j), \qquad 1 \leqq i \leqq n, \quad 1 \leqq j \leqq n,$$

$$\alpha_i, \beta_j, \quad \text{unconstrained}$$

α_i and β_j are the dual variables associated with u_i and v_j, respectively. Orthogonality conditions that are necessary and sufficient for optimality of primal and dual solutions are:

$$x_{ij} > 0 \Rightarrow \alpha_i + \beta_j = d(u_i, v_j) \tag{3.1}$$

$$\alpha_i \neq 0 \Rightarrow \sum_j x_{ij} = 1, \qquad i = 1, \cdots, n, \tag{3.2}$$

$$\beta_j \neq 0 \Rightarrow \sum_i x_{ij} = 1, \qquad j = 1, \cdots, n. \tag{3.3}$$

We will use the version of the Hungarian method [10] for weighted bipartite matching given in [11]. The Hungarian method maintains dual feasibility at all times, and in addition maintains satisfaction of all orthogonality conditions except (3.3). Initially, we start with the empty matching $X = \phi$, and a feasible dual solution $\beta_j = \min_i \{d(u_i, v_j)\}$, $\alpha_i = 0$. The method proceeds in phases, and during each phase the matching X is augmented by an edge. Thus during each phase the number of violations of (3.3) is decreased by one.

During each phase we focus on those edges (u_i, v_j) such that $\alpha_i + \beta_j = d(u_i, v_j)$. These are the *admissible* edges. The *exposed* vertices are those that are not matched by the current matching X. An alternating path is one that alternately traverses an edge in the matching X, and an edge not in the matching X. An augmenting path is one that is between two exposed vertices. A phase consists of searching for an augmenting path among the admissible edges.

For each exposed vertex in V, we grow an alternating tree rooted at the vertex. Each vertex in $U \cup V$ that is in an alternating tree is reachable from the root of the tree via an alternating path that uses only admissible edges. Each vertex in an alternating tree is *labelled*, and each vertex that is not in any of the alternating trees is *free*. The root of an alternating tree is given an s-label of the form $[s, \mathit{root}]$. A matched vertex $v_j \in V$ in an alternating tree is given an s-label of the form $[s, u_k]$ where u_k is the vertex to which v_j is matched. A vertex $u_i \in U$ in an alternating tree is given a t-label of the form $[t, v_j]$, where v_j is the vertex from which u_i was labelled. An s-vertex (t-vertex) is a vertex with an s-label (t-label). Furthermore, an s-vertex (t-vertex) is reachable from an exposed vertex in V by an even (odd) length alternating path that uses only admissible edges.

Let $S(T)$ denote the set of all the s-vertices (t-vertices), and let F denote the set of all the free vertices in U. (Thus $F = U - T$.) Initially all the vertices are free, and to start with each exposed vertex in V is given an s-label. Thus at the beginning of a phase, S consists of the exposed vertices in V and $F = U$. Let δ be defined as

$$\delta = \min_{u_i \in F, v_j \in S} \{d(u_i, v_j) - \alpha_i - \beta_j\}.$$

We use a variable Δ to keep track of the sum of dual changes δ, and associate a weight $w(v)(w(u))$ with each vertex v in V (u in U). The weights are used to implement a phase efficiently. At the beginning of a phase $\Delta = 0$; for each $u_i \in U$, $w(u_i) = \alpha_i$; and for each $v_j \in V$, $w(v_j) = \beta_j$. Depending on whether δ equals zero or exceeds zero, the alternating trees are grown or there is a dual variable change.

Case 1. $\delta = 0$ (add to alternating trees or augment). Let (u_i, v_j), $u_i \in F$, $v_j \in S$ be an admissible edge, i.e., $d(u_i, v_j) - \alpha_i - \beta_j = 0$.

If u_i is exposed, then we have discovered an augmenting path among the admissible edges, and we can construct such a path by backtracking from v_j to the root of the alternating tree containing v_j using the labels on the vertices in the tree.

If u_i is matched to v_k, then give u_i the t-label $[t, v_j]$, and give v_k the s-label $[s, u_i]$,

$$F := F - \{u_i\}, \quad T := T \cup \{u_i\}, \quad S := S \cup \{v_k\},$$

$$w(u_i) := \alpha_i + \Delta, \qquad w(v_k) := \beta_k - \Delta. \qquad \square$$

Case 2. $\delta > 0$ (dual variable change). $\Delta := \Delta + \delta$;

For each vertex $u_i \in T$, $\alpha_i := \alpha_i - \delta$;
For each vertex $v_j \in S$, $\beta_j := \beta_j + \delta$. □

We note that for each s-vertex v_j, β_j equals $w(v_j)+\Delta$, and for each t-vertex u_i, α_i equals $w(u_i)-\Delta$. So during a phase, it suffices to maintain Δ and the weights associated with all the vertices, and there is no need to explicitly update the dual variables α_i, β_j, every time δ exceeds zero. At the end of a phase the correct values of the dual variables may be computed using Δ and the weights. A useful property of the weights is that the weight of a vertex changes only when it is labelled, and once it has been labelled its weight does not change during the remainder of the phase. Thus we may write

$$\delta = \min_{u \in F, v \in S} \{d(u, v) - w(u) - w(v)\} - \Delta.$$

We will use the data structures used in the solution of Problem 1 in § 2, namely, the weighted Voronoi diagram (WVD) and the range tree (RT), to efficiently compute δ, and an edge (u, v), $u \in F$, $v \in S$, such that $d(u, v) - w(u) - w(v) = \delta + \Delta$. Throughout a phase, S is partitioned into S_1 and S_2 such that $|S_2| \leqq \sqrt{n}$. Also, F is partitioned into $F_1, F_2, \cdots, F_{\lceil n^{0.5} \rceil}$, (some of the F_i's possibly empty) such that $|F_i| \leqq \lceil n^{0.5} \rceil$, $1 \leqq i \leqq \lceil n^{0.5} \rceil$. We maintain following data structures:

(1) A priority queue containing the edge *shortest*$[u, S_1]$ for each $u \in F$. The priority of an edge (u, v) in this queue is $d(u, v) - w(u) - w(v)$.

(2) A priority queue containing the edges *shortest*$[v, F_i]$, $1 \leqq i \leqq \lceil n^{0.5} \rceil$, for each vertex v in S_2. The priority of an edge (u, v) in this queue is also $d(u, v) - w(u) - w(v)$.

(3) The WVD/RT for each of the sets $F_1, F_2, \cdots, F_{\lceil n^{0.5} \rceil}$.

δ and an edge for which δ is achieved can be obtained in $O(\log n)$ time by examining an edge with minimum priority in (1) and (2) above. A new vertex added to S always gets inserted into S_2. In order to maintain the condition that $|S_2| \leqq \sqrt{n}$, whenever the size of S_2 reaches the threshold of $\sqrt{n}$ we add all the vertices in S_2 to S_1 and reset S_2 to the null set. Then *shortest*$[u, S_1]$ must be recomputed for every $u \in F$. From Lemma 1 in § 2, this recomputation may be done in $O(n \log n)$ time using a WVD/RT for S_1, leading to total of $O(n^{1.5} \log n)$ operations for the recomputation for the entire phase. Next we shall see that an insertion into S_2, and a deletion from F, each cost $O(n^{0.5} \log n)$ operations. As there are $O(n)$ such insertions and deletions in a phase, this leads to a total of $O(n^{1.5} \log n)$ operations per phase.

(i) Suppose a vertex v is inserted into S_2. Then we must compute *shortest*$[v, F_i]$, and insert it into the priority queue in (2) above, for $i = 1, \cdots, \lceil n^{0.5} \rceil$. Using the maintained WVD/RT for F_i, *shortest*$[v, F_i]$ can be found in $O(\log n)$ time. Hence an insertion costs $O(n^{0.5} \log n)$ operations.

(ii) Suppose a vertex u is deleted from F, and suppose $u \in F_i$. Then recomputing the WVD/RT for F_i requires $O(n^{0.5} \log n)$ time, and recomputing *shortest*$[v, F_i]$ for all the vertices v in S_2 and maintaining the priority queue also requires $O(n^{0.5} \log n)$ time. Thus a deletion costs $O(n^{0.5} \log n)$ operations.

Finally, since a phase takes $O(n^{1.5} \log n)$ time, and as there are at most n phases, the total running time of the weighted bipartite matching algorithm for points in the plane is $O(n^{2.5} \log n)$. In § 3.1 we shall see how to further improve the running time of the bipartite matching algorithm to $O(n^2(\log n)^3)$ for the case of L_1 metric.

3.1. Improving the complexity for the L_1 metric. In this section we shall show that for the case of the L_1 metric, the running time of the weighted bipartite matching algorithm for points on the plane can be further improved to $O(n^2(\log n)^3)$. Since we shall be dealing with the L_1 metric only in this section, $d(p, q)$ will denote the L_1

distance between p and q throughout this section. Note that maintaining the dual variables can be accomplished in $O(n)$ time per phase. To efficiently compute δ, we shall maintain a data structure containing all the points in $F \cup S$ such that:

(1) *shortest*(F, S) can be obtained in $O(1)$ time.

(2) A point in S can be inserted (or deleted) in $O((\log n)^3)$ time, and a point in F can be deleted (or inserted) in $O((\log n)^3)$ time.

Suppose that *shortest*$(F, S) = (u, v)$. Depending on whether $d(u, v) - w(u) - w(v) - \Delta$ equals or exceeds zero there is a dual variable change, and then (u, v) becomes admissible. If u is exposed then there is an augmentation and the phase ends. Otherwise, both u and the vertex to which u is matched get labelled and enter an alternating tree. u is deleted from F and the vertex to which u is matched is inserted into S. Correspondingly, there is a deletion from and an insertion into the above-mentioned data structure. Thus, using the above data structure it takes $O((\log n)^3)$ operations to increase the number of labelled vertices by 2, thereby leading to $O(n(\log n)^3)$ operations per phase. This leads to a running time of $O(n^2(\log n)^3)$ for the L_1 case.

Next, we describe a data structure for maintaining *shortest*$[F', S']$ for the special case when F' and S' are separated by a vertical line. Then using the data structure for this special case we shall show how to implement the above-mentioned data structure for maintaining *shortest*$[F, S]$. Let U' and V' be fixed subsets of U and V, respectively, such that all the points in U' lie on one side of the vertical line $x = a$, and all the points in V' line on the other side of the line $x = a$. (Points in U' and V' could lie on the line $x = a$.) Let $F' = F \cap U'$, and let $S' = S \cap V'$. Let $m = |U' \cup V'|$. Let $b_1 \leqq b_2 \leqq \cdots \leqq b_m$ be the nondecreasing sequence of the y-coordinates of the points in $U' \cup V'$. The data structure consists of a segment tree for this sequence of y-coordinates together with extra information stored at each node in the segment tree. Consider a node ψ in the segment tree associated with the interval $[b_i, b_j)$ of the y-axis. Let $k = \lceil (i+j)/2 \rceil$. At node ψ we store the following:

(i) Four priority queues, one for each of the four sets $F' \cap H_y(b_i, b_k)$, $F' \cap H_y(b_k, b_j)$, $S' \cap H_y(b_i, b_k)$, and $S' \cap H_y(b_k, b_j)$. The priority of a point p in any of the four queues is given by $d(p, (a, b_k)) - w(p)$.

(ii) The edge *shortest*$[F' \cap H_y(b_i, b_j), S' \cap H_y(b_i, b_j)]$.

Next, we shall see that once the two edges *shortest*$[F' \cap H_y(b_i, b_k), S' \cap H_y(b_i, b_k)]$ and *shortest*$[F' \cap H_y(b_k, b_j), S' \cap H_y(b_k, b_j)]$ are available, using the priority queues stored at node ψ, *shortest*$[F' \cap H_y(b_i, b_j), S' \cap H_y(b_i, b_j)]$ may be computed in $O(\log m)$ time. This is seen as follows. First, note that for a pair of points u and $v, u \in F' \cap H_y(b_i, b_k)$, and $v \in S' \cap H_y(b_k, b_j)$,

$$d(u, v) = d(u, (a, b_k)) + d(v, (a, b_k)),$$

and hence

$$d(u, v) - w(u) - w(v) = d(u, (a, b_k)) - w(u) + d(v, (a, b_k)) - w(v).$$

Thus

$$\min_{u \in F' \cap H_y(b_i, b_k), v \in S' \cap H_y(b_k, b_j)} \{d(u, v) - w(u) - w(v)\}$$

$$= \min_{u \in F' \cap H_y(b_i, b_k)} \{d(u, (a, b_k)) - w(u)\} + \min_{v \in S' \cap H_y(b_k, b_j)} \{d(v, (a, b_k)) - w(v)\}.$$

The same relationship holds for $u \in F' \cap H_y(b_k, b_j)$ and $v \in S' \cap H_y(b_i, b_k)$. Thus *shortest*$[F'\cap H_y(b_i, b_k), S'\cap H_y(b_k, b_j)]$ and *shortest*$[F'\cap H_y(b_k, b_j), S'\cap H_y(b_i, b_k)]$ may be computed by examining the four points with minimum priority in the four priority queues described in (i) above. This requires $O(\log m)$ time. And once, *shortest*$[F'\cap H_y(b_i, b_k), S'\cap H_y(b_i, b_k)]$, *shortest*$[F'\cap H_y(b_k, b_j), S'\cap H_y(b_k, b_j)]$, *shortest*$[F'\cap H_y(b_i, b_k), S'\cap H_y(b_k, b_j)]$, and *shortest*$[F'\cap H_y(b_k, b_j), S'\cap H_y(b_i, b_k)]$, are available, *shortest*$[F'\cap H_y(b_i, b_j), S'\cap H_y(b_i, b_j)]$ is computable in constant time.

We now show that a point may be inserted into or deleted from the data structure for maintaining *shortest*$[F', S']$ in $O((\log m)^2)$ time. When a point p is added to S' (or F'), it is inserted into the priority queues at the nodes in the segment tree that lie on the path from the root to the leaf corresponding to the point p. So there are $O(\log m)$ insertions, each costing $O(\log m)$ operations. We then start from the leaf corresponding to the inserted point p and work our way up toward the root, updating in sequence the edge (as specified by (ii) above) that is stored at each of the $O(\log m)$ nodes on the path from the leaf to the root. Let ψ be a node on this path. Once the updates at nodes on the subpath from the leaf to ψ have been performed, the update at node ψ may be performed in $O(\log m)$ time using the priority queues stored at ψ. Thus, whenever there is an insertion, *shortest*$[F', S']$ is updated in $O((\log m)^2)$ operations. The same bound holds when a point is deleted from F' (or S').

The data structure for maintaining *shortest*$[F, S]$ is a two level data structure. At the top level is a segment tree for the nondecreasing sequence $a_1 \leqq a_2 \leqq \cdots \leqq a_{2n}$ of the x-coordinates of the points in $U \cup V$, and at each node is a data structure for the special case mentioned above. Specifically, let ψ be a node in the top level segment tree and let $[a_i, a_j)$ be the interval of the x-axis associated with ψ. Let $k = \lceil (i+j)/2 \rceil$. At node ψ we store the following:

(I). A data structure as described above for maintaining each of the two edges *shortest*$[F\cap H_x(a_i, a_k), S\cap H_x(a_k, a_j)]$ and *shortest*$[F\cap H_x(a_k, a_j), S\cap H_x(a_i, a_k)]$. This is the secondary data structure stored at ψ.

(II). The edge *shortest*$[F\cap H_x(a_i, a_j), S\cap H_x(a_i, a_j)]$.

We note that the edge stored at the root (as specified by (II) above) is *shortest*$[F, S]$. When a point is inserted, we start at the leaf in the top level segment tree corresponding to the inserted point, and perform updates in sequence at the nodes on the path from this leaf to the root. Suppose ψ is a node on this path. Once the updates at nodes on the subpath from the leaf to ψ have been performed, the edges *shortest*$[F\cap H_x(a_i, a_k), S\cap H_x(a_i, a_k)]$ and *shortest*$[F\cap H_x(a_k, a_j), S\cap H_x(a_k, a_j)]$, are available. The updates in the secondary data structure at ψ are performed in $O((\log n)^2)$ time, and then *shortest*$[F\cap H_x(a_i, a_k), S\cap H_x(a_k, a_j)]$, and *shortest*$[F\cap H_x(a_k, a_j), S\cap H_x(a_i, a_k)]$, are also available. Then *shortest*$[F\cap H_x(a_i, a_j), S\cap H_x(a_i, a_j)]$ is computable in constant time. Since there are $O(\log n)$ nodes on the path from a leaf in the top level segment tree to the root, the total cost of an insertion is $O((\log n)^3)$ operations. The same bound holds for deleting a point.

4. Weighted general matching on points in the plane.

We are given a set V of $2n$ points in the plane. The set of points V induces a complete graph whose vertices are the points in V, and the weight of an edge between two points is the distance between the points under some metric. We shall consider two cases, one where the weight of an edge is given by the Euclidean (L_2) distance between the two endpoints of the edge, and the other where the weight of an edge is given by the L_1 distance between the endpoints of the edge. We let $v_1, v_2, \cdots, v_{2n}$ denote the $2n$ vertices (points) in V. Let O denote the set of all those subsets of V having cardinality that is odd and greater than one.

The MWCM problem in the complete graph on V is formulated as a linear program [4], [11], [14]:

$$\min \sum_{1 \leqq i < j \leqq 2n} d(v_i, v_j) x_{ij},$$

$$\text{subject to} \quad \sum_{1 \leqq j \leqq 2n, j \neq i} x_{ij} = 1, \qquad i = 1, \cdots, 2n$$

$$\forall B \in O, \quad \sum_{v_i, v_j \in B} x_{ij} \leqq \frac{|B|-1}{2},$$

$$x_{ij} \geqq 0, \qquad 1 \leqq i < j \leqq 2n$$

with the understanding that $x_{ij} = 1$ if and only if (v_i, v_j) is in the matching X.

The dual linear program is given by

$$\max \sum_{i=1}^{2n} \alpha_i + \sum_{B \in O} z(B)$$

$$\text{subject to} \quad \alpha_i + \alpha_j + \sum_{v_i, v_j \in B, B \in O} z(B) \leqq d(v_i, v_j), \qquad 1 \leqq i < j \leqq 2n$$

$$\forall B \in O, \quad z(B) \leqq 0.$$

α_i is the dual variable associated with vertex v_i, and $z(B)$ is the dual variable associated with the odd set B. Orthogonality conditions that are necessary and sufficient for optimality of primal and dual solutions are:

$$x_{ij} > 0 \Rightarrow \alpha_i + \alpha_j + \sum_{v_i, v_j \in B, B \in O} z(B) = d(v_i, v_j), \tag{4.1}$$

$$\alpha_i \neq 0 \Rightarrow \sum_{1 \leqq j \leqq 2n, j \neq i} x_{ij} = 1, \tag{4.2}$$

$$z(B) < 0 \Rightarrow \sum_{v_i, v_j \in B} x_{ij} = \frac{|B|-1}{2}. \tag{4.3}$$

Edmond's algorithm [4], [11], [14] for finding a minimum weight complete matching maintains dual feasibility at all times, and in addition maintains the satisfaction of all orthogonality conditions, except conditions (4.2). The algorithm starts with the empty matching $X = \phi$, and a feasible dual solution given by $\alpha_i = \frac{1}{2} \min_{v_j \in V - \{v_i\}} d(v_i, v_j)$, $i = 1, \cdots, 2n$, and $z(B) = 0$, for all $B \in O$. The algorithm proceeds in phases. During each phase the cardinality of the matching X increases by one thereby decreasing the number of violations of (4.2) by one.

An $O(nm \log n)$ implementation of Edmond's algorithm for finding a minimum weight complete matching in a graph with n vertices and m edges is given in [8]. We shall utilize the underlying geometry together with some of the ideas in [8] to obtain an $O(n^{2.5}(\log n)^4)$ algorithm for finding an MWCM on $2n$ points on the plane. First, we shall sketch Edmond's algorithm, and then show how to efficiently implement it for points in the plane.

4.1. Blossoms and their representation. As the algorithm proceeds, it discovers certain subsets of V (of odd size) called *blossoms*. It is convenient to consider the vertices of V as (trivial) blossoms of size one. For a pair of blossoms B, B', either $B \cap B' = \phi$, $B \subseteq B'$, or $B' \subseteq B$, and a blossom that is not a subset of any other blossom is a *maximal* blossom. The algorithm has $z(B) < 0$ only for those $B \in O$ that are nontrivial blossoms. Consequently, the number of nonzero $z(B)$'s is $O(n)$.

The discussion below about blossoms follows [8]. The representation that we use for blossoms is identical to the one in [8] but we shall describe it here for completeness.

An edge (v_i, v_j) is *admissible* if and only if

$$\alpha_i + \alpha_j + \sum_{v_i, v_j \in B, B \in O} z(B) = d(v_i, v_j).$$

An *alternating path* from v_0 to v_r is a sequence of edges $\{e_i = (v_{i-1}, v_i)\}_{i=1}^{r}$ such that for $i = 1, \cdots, r-1$, $e_i \in X$ if and only if $e_{i+1} \notin X$. An *alternating path* from a maximal blossom B_0 to a maximal blossom B_r (possibly $B_0 = B_r$) is a sequence of edges $\{e_i = (u_{i-1}, v_i)\}_{i=1}^{r}$ such that for $i = 0, \cdots, r$, $u_i, v_i \in B_i$, where $B_0, \cdots, B_r$ are distinct maximal blossoms, and for $i = 1, \cdots, r-1$, $e_i \in X$ if and only if $e_{i+1} \notin X$. When the algorithm discovers an odd length alternating path, that uses only admissible edges, from a maximal blossom B_0 to itself (i.e., $B_0 = B_r$, $e_1, e_r \notin X$), a new blossom B is formed. $B_1, \cdots, B_r$ are the subblossoms of the new blossom B. (Note: A blossom that is a proper subset of B_i, for some i, $1 \leqq i \leqq r$, is not a subblossom of B.) Each blossom has a *base* vertex. The base of a trivial blossom is the unique vertex in it. The base of the blossom B defined above is the base of B_0 $(=B_r)$.

A nontrivial blossom B is represented by the cyclic double-linked list $\{(B_i, e_i)\}_{i=1}^{r}$, and by its base. (From an entry (B_i, e_i) in the list there are bidirectional links to (B_l, e_l) where $l = (i \bmod r) + 1$.) B_i, $i = 1, \cdots, r$ are the subblossoms of B, and B_r is the distinguished subblossom that contains the base of B. For each i, $1 \leqq i \leqq r-1$, $(e_1, e_2, \cdots, e_i)$ and $(e_r, e_{r-1}, \cdots, e_{i+1})$ are alternating paths that use admissible edges only from B_r to B_i. The one of even length is the one whose last edge is in X.

A blossom B of size $2k+1$ has the following two properties. First, there are exactly k matched edges (i.e., edges in X) both of whose endpoints are in B, and the base of B is that vertex in B not an endpoint of one of these k edges. Second, there is an even length alternating path, which uses only admissible edges, from the base of B to every vertex in B. As a consequence an alternating path, which uses admissible edges only, from a maximal blossom B to a maximal blossom B' can be expanded into an alternating path, which uses admissible edges only, from the base of B to the base of B'.

The structure of a maximal blossom B can be represented by a tree. In this tree the sons of a blossom B are its subblossoms $B_1, \cdots, B_r$, and the leaves are the vertices of B. This tree will be denoted as the *structure tree* of B. The structure tree of B is implicitly represented by the cyclic double-linked lists $\{(B_i, e_i)\}_{i=1}^{r}$ corresponding to B, its subblossoms, the subblossoms of its subblossoms, and so on. The tree implies a total order on the vertices in B: $v_i < v_j$ if and only if v_i is to the left of v_j in the tree. The base of B is largest vertex in this ordering. Furthermore, a blossom B' that is a subset of B corresponds to an interval in this ordering. Also note that the base of a blossom changes only when the matching X is augmented, and so during a phase of the algorithm the ordering of the vertices in an existing blossom does not change.

During the course of the matching algorithm, given a vertex v_j we shall need to know the identity of the maximal blossom containing v_j. To be able to obtain this identity efficiently, we also represent the maximal blossoms as ordered sets of vertices. These ordered sets are implemented as *concatenable queues* [1]. Such queues support the operations of find, split, and concatenate [1] in time proportional to the logarithm of the number of items in the queue. So given a vertex v_j, the base of the maximal blossom containing v_j and thereby the identity of the maximal blossom containing v_j can be computed in $O(\log n)$ time. When a blossom B splits into r subblossoms, the splitting of the concatenable queue for B into the concatenable queues for the subblossoms of B may be carried out in $O(r \log n)$ time. Similarly, when a new blossom

B is formed from r blossoms, the concatenation of the r concatenable queues of the subblossoms of B to obtain the concatenable queue for B may also be carried out in $O(r \log n)$ time.

4.2. A phase of the algorithm. An edge in the matching X is said to be matched, and an edge not in X is said to be unmatched. A vertex is *matched* if there is an edge in X incident to it. A vertex is *exposed* if there is no edge in X incident to it. A blossom is exposed (matched) if and only if its base is exposed (matched). An *augmenting path* is an alternating path between two distinct exposed maximal blossoms or between two distinct exposed vertices. A phase consists of looking for an augmenting path, that uses admissible edges only, between two exposed maximal blossoms.

We grow an alternating tree rooted at each exposed maximal blossom. The nodes in an alternating tree are themselves maximal blossoms. Each node in an alternating tree is reachable from the root by an alternating path that uses only admissible edges. Each maximal blossom that is a node in an alternating tree is *labelled*, and each maximal blossom that is not in any of the alternating trees is *free*. (Note that only a maximal blossom can be labelled or free.) A node in an alternating tree that is reachable by an even (odd) length alternating path from the root is given an s-label (t-label). An s-label (t-label) given to a node B is of the form $[s, (v_i, v_j)]([t, (v_i, v_j)])$ where $v_j \in B$ and v_i is the first vertex (point) on the alternating path from B to the root of the alternating tree containing B. A node with an s-label (t-label) is called an s-blossom (t-blossom). A vertex in an s-blossom (t-blossom) is referred to as an s-vertex (t-vertex). A vertex in a free blossom is called a free vertex. We let $S(T)$ denote the set of all the s-vertices (t-vertices), and let F denote the set of free vertices. (Note that $V = S \cup T \cup F$.)

At the start of a phase each maximal exposed blossom is given an s-label, and each maximal matched blossom is free. (So initially, $T = \phi$.) A phase consists of several steps and at each step one of following four things must happen. Either there is an admissible edge between two s-vertices not in the same s-blossom or there is an admissible edge between an s-vertex and a free vertex or the dual variable corresponding to a t-blossom is zero or there is a dual variable change. In the first case an alternating tree can be grown, in the second case an augmenting path is found or a new blossom is discovered, in the third case a t-blossom expands, and an occurrence of the fourth case is always followed by an occurrence of one the first three cases in the next step. Let

$$\delta_1 = \min_{v_i \in S, v_j \in F} \{d(v_i, v_j) - \alpha_i - \alpha_j\},$$

$$\delta_2 = \min_{v_i, v_j \in S, v_i, v_j \text{ not in the same } s\text{-blossom}} \left\{\frac{d(v_i, v_j) - \alpha_i - \alpha_j}{2}\right\},$$

$$\delta_3 = \min_{B \text{ a } t\text{-blossom}} \left\{\frac{-z(B)}{2}\right\},$$

and

$$\delta = \min\{\delta_1, \delta_2, \delta_3\}.$$

A detailed description of the four cases that can occur at a step during a phase is given below. Δ accumulates the dual variable changes δ during a phase, and at the start of a phase $\Delta = 0$.

Case 1. $\delta_1 = 0$ (grow an alternating tree). In this case there is an admissible edge between an s-vertex and a free vertex. Let (v_i, v_j) be such an edge where v_i is an

s-vertex and v_j is in a free blossom B. Let b be the base of B and let (b, v_k) be the matched edge incident on b. Then v_k is also a free vertex, and is in some free blossom B'. B gets the label $[t, (v_i, v_j)]$, and B' gets the label $[s, (b, v_k)]$.

By using the representations of the blossoms as ordered sets (i.e., the concatenable queues), the maximal blossoms containing v_i, v_j, and v_k, may be found in $O(\log n)$ time, and so once the edge (v_i, v_j) is available the labelling takes $O(\log n)$ time. □

Case 2. $\delta_2 = 0$ (discover a new blossom or augment). In this case there is an admissible edge between two s-vertices not in the same s-blossom. Let (v_i, v_j) be such an edge, and let v_i, v_j, be in distinct s-blossoms B, B', respectively. Using the labels on blossoms, we backtrack along the alternating path from B to the root of the alternating tree containing B. Simultaneously, we also backtrack along the alternating path from B' to the root of the alternating tree containing B'. We make a careful backtrack by backtracking one blossom on both paths each time, marking blossoms along the way. Either we discover a new blossom or find an augmenting path.

(i) A new blossom is found. Suppose the new blossom has r subblossoms. Then we will visit at most $2r$ blossoms before finding the first common blossom C on both paths. We use the parts of the paths from C to B and from C to B' to generate the cyclic double-linked list $\{(B_i, e_i)\}_{i=1}^{r}$ for the new blossom, where $B_r = C$ and e_i are taken from the labels on the two paths. The base of the new blossom is the base of C. The new blossom gets an s-label, and the subblossoms of the new blossom get unlabelled and stop being s-blossoms/t-blossoms (since they are no longer maximal blossoms). The dual variable associated with the new blossom is initialized to zero. Using the concatenable queues and the labels, finding the common blossom C and constructing the new cyclic double-linked list requires $O(r \log n)$ operations. The concatenable queue for the new blossom is obtained by concatenating together the concatenable queues for the r subblossoms in $O(r \log n)$ time.

(ii) An augmenting path is discovered between two exposed blossoms. If we discover an augmenting path π between two exposed blossoms, then we construct an augmenting path between the base vertices of the two exposed blossoms as follows. For each blossom on the path π, we recursively find an even length alternating path between the base of the blossom and the vertex by which the path π leaves the blossom. Once the even length alternating paths within all the blossoms that lie on the path π are available, we connect up these paths using the edges on π to give an augmenting path between the base vertices of the two exposed blossoms. The resulting augmenting path uses admissible edges only. We switch the status of the edges on the resulting augmenting path from matched to unmatched and vice versa. This augments the matching X and the current phase ends. We note that for each blossom through which the augmenting path passes, the base vertex changes and the ordering of the vertices in the blossom that is implied by the structure tree also changes. □

Case 3. $\delta_3 = 0$ (expand a t-blossom). Let B be a t-blossom such that the dual variable $z(B)$ associated with B is zero, and let B be labelled $[t, (v_j, v_i)]$. Suppose that $v_i \in B_i$ where $B_1, \cdots, B_r$ are the subblossoms of B and B_r contains the base of B. The blossom B expands, the blossoms on the odd length alternating path from B_i to B_r become free, and the blossoms on the even length alternating path from B_i to B_r get alternating labels starting and ending with a t-label. The labels are generated using the edges e_i in the cyclic double-linked list for B. The concatenable queues for the subblossoms of B are obtained by splitting for $i = 1, \cdots, r-1$ each B_i from B according to its base, which is its largest element.

Using the cyclic double-linked lists and the concatenable queues, the expansion of a t-blossom B with r subblossoms can be carried out in $O(r \log n)$ time. □

Case 4. $\delta > 0$ (dual variable change). In this case the dual variables are changed as follows:

For every s-vertex v_i, $\alpha_i := \alpha_i + \delta$;

For every t-vertex v_j, $\alpha_j := \alpha_j - \delta$;

For every s-blossom B, $z(B) := z(B) - 2\delta$;

For every t-blossom B', $z(B') := z(B') + 2\delta$;

$\Delta := \Delta + \delta$. □

At the end of a phase we construct the augmenting path, and augment the matching X by switching the status of the edges on the augmenting path from matched to unmatched and vice versa. We then change the base of each blossom through which the augmenting path passes, and rebuild the concatenable queues for all the maximal blossoms through which the augmenting path passes, since changing the base changes the ordering of the vertices in the blossom. Finally, we expand all maximal blossoms B such that $z(B) = 0$. (This ensures that at the start of the next phase the dual variable associated with each maximal blossom has nonzero value.)

We have the following useful lemma concerning the behavior of blossoms in a phase, which follows directly from Cases 1–4 described above.

LEMMA 4. (a) *Each blossom that gets an s-label some time during the phase, corresponds to a unique node in the structure tree of some maximal blossom at the end of the phase. Each blossom that gets a t-label or becomes free some time during the phase, corresponds to a unique node in the structure tree of some maximal blossom at the beginning of the phase.*

(b) *A free blossom is either free from the beginning of the phase or becomes free because it is the subblossom of some t-blossom that expands. Once a blossom receives a t-label, either it stays as a t-blossom until the phase ends, it expands, or it loses its label because it becomes the subblossom of a new s-blossom, in which case it stays unlabelled until the phase ends. Once a blossom receives an s-label, it either remains an s-blossom, until the phase ends, or it loses its label because it becomes the subblossom a new s-blossom, in which case it stays unlabelled until the phase ends.*

(c) *The total number of free blossoms plus s-blossoms plus t-blossoms in a phase is* $O(n)$. □

The remainder of § 4 will be devoted to the efficient implementation of a phase. We shall assign a weight $w(v_i)$, suitably related to the dual variable α_i, to each vertex v_i in V, a weight $w(B)$ to each s-blossom/t-blossom B, and also an offset $\mu(B)$ to each blossom B, which measures the change in the value of the dual variable of a vertex in B. In § 4.3 we shall describe how to compute these weights and offsets, and show how to efficiently maintain the dual variables using these weights and offsets. In §§ 4.4 and 4.5 these weights and offsets will be used along with the underlying geometry to construct a scheme for efficiently maintaining δ_1 and δ_2, respectively. The running time per phase may be broken down as follows:

(1) The computation at the end of a phase (i.e., constructing the augmenting path, etc.) requires $O(n \log n)$ time.

(2) In § 4.3 it is shown that the time for computing the weights and the offsets and for maintaining the dual variables using the weights and the offsets is $O(n^{1.5})$ per phase.

(3) In § 4.4 we show that the time required for maintaining δ_1, and an appropriate edge between an s-vertex and a free vertex for which δ_1 is achieved, is $O(n^{1.5}(\log n)^2)$ per phase. Furthermore, since Case 1 can occur $O(n)$ times during one phase, $O(n \log n)$ extra time is spent in Case 1 per phase in addition to the time required for maintaining δ_1. So the total time spent in Case 1 is $O(n^{1.5}(\log n)^2)$ per phase.

(4) In § 4.5 we show that the time required to maintain δ_2, and an appropriate edge between two s-vertices not in the same s-blossom for which δ_2 is achieved, is $O(n^{1.5}(\log n)^4)$ per phase. From Lemma 4 above it follows that $O(n \log n)$ extra time is spent in Case 2 per phase in addition to the time required for maintaining δ_2. So the total time spent in Case 2 is $O(n^{1.5}(\log n)^4)$ per phase.

(5) To maintain δ_3 the t-blossoms are kept in a priority queue, and the priority of a t-blossom B in this queue is given by $-w(B)$. Each t-blossom B satisfies the condition $z(B) = w(B) + 2\Delta$, and so the t-blossom with the largest weight is also the t-blossom with largest value for the dual variable $z(B)$. Then from Lemma 4 above it follows that $O(n \log n)$ time is spent in Case 3 per phase in addition to the time for computing the weights $w(B)$ of blossoms.

Thus the time required by the algorithm per phase is $O(n^{1.5}(\log n)^4)$. Since the number of phases is at most n, the total running time is $O(n^{2.5}(\log n)^4)$.

4.3. Weights, offsets, and dual variables. It is too time consuming to explicitly update the dual variables whenever there is a dual variable change (i.e., whenever Case 4 in § 4.2 occurs), so they are implicitly maintained by associating weights w with the vertices and the blossoms, and offsets μ with the blossoms. Similar ideas are also used in [8] to implicitly maintain the dual variables. Let $w(v_i)$ denote the weight associated with vertex v_i, and let $\mu(B)$ denote the offset associated with blossom B. We divide blossoms into *large* and *small* blossoms according to their size: a large blossom is one that contains at least $\sqrt{n}$ vertices, and a small blossom is one that contains less than $\sqrt{n}$ vertices. As mentioned in § 4.2, Δ denotes the sum of the dual variable changes δ since the beginning of the phase. The weights associated with vertices and the offsets associated with blossoms are updated so that after each step of a phase (that consists of executing one of the four cases in § 4.2) the following four relationships are maintained:

(1) For each s-vertex v_i, $\alpha_i = w(v_i) + \Delta$.
(2) For each vertex v_j in a t-blossom B, $\alpha_j = w(v_j) + \mu(B) - \Delta$.
(3) For each vertex v_k in a large free blossom B', $\alpha_k = w(v_k) + \mu(B')$.
(4) For each vertex v_l in a small free blossom B'', $\alpha_l = w(v_l)$.

At any time during a phase, to compute the value of the dual variable associated with a t-vertex or a free vertex we find the maximal blossom containing the vertex and then use the above relations. So at any time during the phase, the value of the dual variable associated with a vertex is computable in $O(\log n)$ time. At the end of a phase the values of the dual variables associated with all the vertices are explicitly computed.

At the beginning of a phase the weights and offsets are initialized as follows.

For each $v_i \in V$, $w(v_i) := \alpha_i$;
For each blossom B, $\mu(B) := 0$.

Note that $\Delta = 0$ at the beginning of a phase.

Next we shall describe how to update the weights and offsets during each of the four cases that occur at a step during a phase. (These cases were described in § 4.2.)

Case 1. $\delta_1 = 0$. In this case a free blossom B gets a t-label, and a free blossom B', such that a matched edge joins the base of B to a vertex in B', gets an s-label. The weights and offsets are updated as follows.

```
If newly labelled $t$-blossom $B$ is a large blossom
    then $\mu(B) := \mu(B) + \Delta$
    else $\mu(B) := \Delta$;
For each vertex $v_i$ in newly labelled $s$-blossom $B'$,
    If $B'$ is a large blossom
```

```
then $w(v_i) := w(v_i) + \mu(B') - \Delta$
else $w(v_i) := w(v_i) - \Delta$.   □
```

Case 2. $\delta_2 = 0$. The weights and offsets need to be updated only in the case when a new blossom is discovered in Case 2. Let B be the new blossom that is discovered. Note that B gets an s-label, and all the vertices in each subblossom of B, which had a t-label, switch status from t-vertices to s-vertices.

```
For each subblossom $B'$ of $B$ which had a $t$-label,
    For each vertex $v_i$ in $B'$, $w(v_i) := w(v_i) + \mu(B') - 2\Delta$.   □
```

Case 3. $\delta_3 = 0$. In this case a t-blossom B expands, and each of its subblossoms either becomes free or gets an s- or a t-label. We perform the following updates:

```
For each subblossom $B'$ of $B$,
    If $B'$ gets a $t$-label then $\mu(B') := \mu(B)$;
    If $B'$ gets an $s$-label then
        For each vertex $v_i$ in $B'$, $w(v_i) := w(v_i) + \mu(B) - 2\Delta$;
    If $B'$ is a large blossom and $B'$ becomes free then $\mu(B') := \mu(B) - \Delta$;
    If $B'$ is a small blossom and $B'$ becomes free then
        For each vertex $v_i$ in $B'$, $w(v_i) := w(v_i) + \mu(B) - \Delta$.   □
```

Case 4. $\delta > 0$. $\Delta := \Delta + \delta$.   □

It is easily verified that the updates to the weights and the offsets maintain the above-mentioned four relationships between the weights, the offsets, and the dual variables. A blossom switches status whenever one of three things happens: (1) it is free and gets labelled; (2) it is labelled and gets unlabelled because it becomes the subblossom of a new blossom; or (3) it becomes free or labelled because it is the subblossom of some blossom that expands. From Lemma 4 in § 4.2, the total number status switches of blossoms is $O(n)$ per phase. So the total number of updates to the offsets μ is $O(n)$, since $\mu(B)$ is updated only when B switches status. The weight of a vertex is updated when it switches status from t-vertex/free vertex to s vertex, and whenever it becomes free and is in a small free blossom. Since a vertex switches status to s-vertex at most once during a phase, since the total number of status switches of blossoms is $O(n)$ per phase, and since a small blossom contains at most $\sqrt{n}$ vertices, we get that the total number of updates to weights w of vertices is $O(n^{1.5})$ per phase. So maintaining the weights of all the vertices and the offsets of all the blossoms requires $O(n^{1.5})$ time per phase.

The values of the dual variables associated with s-blossoms and t-blossoms are also implicitly maintained by weights associated with these blossoms. Let $w(B)$ denote the weight associated with blossom B. When a blossom B receives a t-label we initialize its weight as

$$w(B) := z(B) - 2\Delta.$$

As long as B remains a t-blossom the value of $z(B)$ is implicitly given by the relation

$$z(B) = w(B) + 2\Delta.$$

When B stops being a t-blossom (by becoming a subblossom of a new blossom and thereby getting unlabelled) we explicitly compute the value of the dual variable $z(B)$. When a blossom B' receives an s-label we initialize its weight as

$$w(B') := z(B') + 2\Delta,$$

and for the duration of the period for which B' is an s-blossom the value of $z(B')$ is implicitly given by

$$z(B') = w(B') - 2\Delta.$$

When B' stops being an s-blossom (by becoming a subblossom of a new blossom and thereby getting unlabelled) we explicitly compute the value of the dual variable $z(B')$. Since the total number of times blossoms are labelled is $O(n)$ per phase, the weights of the blossoms can be maintained in $O(n)$ time per phase.

4.4. Maintaining δ_1 during a phase. In this section we shall describe how to maintain δ_1, and an edge between an s-vertex and a free vertex such that δ_1 is achieved for that edge. The slack associated with an edge (v_i, v_j) is given by the quantity $d(v_i, v_j) - \alpha_i - \alpha_j$, and denoted by $slack[(v_i, v_j)]$. Let

$$\delta_{11} = \min_{v_i \in S, v_j \text{ in a large free blossom}} \{slack[(v_i, v_j)]\},$$

and

$$\delta_{12} = \min_{v_i \in S, v_j \text{ in a small free blossom}} \{slack[(v_i, v_j)]\}.$$

Then clearly,

$$\delta_1 = \min\{\delta_{11}, \delta_{12}\}.$$

In §§ 4.4.1 and 4.4.2 we shall describe how to maintain δ_{11} and δ_{12} in $O(n^{1.5}(\log n)^2)$ and $O(n^{1.5} \log n)$ time per phase, respectively.

4.4.1. Maintaining δ_{11}. During the computation of δ_{11}, we will be required to quickly find $shortest[v_j, B]$ for an s-vertex v_j and a large free blossom B. A data structure for doing this is obtained as follows. From Lemma 4 (§ 4.2), a free blossom B is a blossom that existed at the beginning of the phase. Consider the maximal blossom B' that contained B at the start of the phase, and the ordering imposed on the vertices of B' by the structure tree of B'. The free blossom B corresponds to an interval in this ordering, in other words if $v_i, v_l \in B$ and $v_i < v_l$ then each vertex v_k in B' such that $v_i < v_k < v_l$ is also in B. So the problem is to preprocess each maximal blossom B' at the start of a phase, so that later on in the phase, given a vertex v_j and a blossom B that is a subset of B' (and therefore corresponds to an interval in the ordering on B'), $shortest[v_j, B]$ may be found quickly. This is precisely Problem 2 given in § 2. So, by Lemma 2 (§ 2.2) we can preprocess all the maximal blossoms B' that exist at the start of a phase in $O(n(\log n)^2)$ time, so that given a free blossom B later in the phase together with a vertex v_j, $shortest[v_j, B]$ can be found in $O((\log n)^2)$ time.

As in the bipartite case, the set of s-vertices S is partitioned into S_1, S_2, such that $|S_2| \leqq \sqrt{n}$. We maintain the following edges:

(1) For each large free blossom B, the edge $shortest[B, S_1]$. All these edges are in a priority queue, with the priority of an edge (v_i, v_j) being given by $slack[(v_i, v_j)]$.

(2) For each large blossom B' that is either a subset of a t-blossom or a subset of a free blossom, the edge $shortest[B', S_1]$.

(3) For each pair v_j, B, where $v_j \in S_2$ and B is a large free blossom, the edge $shortest[v_j, B]$. All these edges are also in a priority queue, with the priority of an edge being given by its slack.

We note that for each vertex v_i in a large free blossom B, $\alpha_i = w(v_i) + \mu(B)$ where $\mu(B)$ is the offset associated with B, and that for each s-vertex v_j, $\alpha_j = w(v_j) + \Delta$. Thus if B is a large free blossom and $S' \subseteq S$, then

$$slack[shortest[B, S']] = \min_{v_i \in B, v_j \in S'} \{slack[(v_i, v_j)]\}.$$

So, δ_{11} is achieved either for the edge in (1) above with minimum priority or for the edge in (3) above with minimum priority, and may be computed in $O(\log n)$ time by examining these two edges.

At the beginning of a phase, S_1 consists of the vertices in exposed blossoms and S_2 is empty. Note that a vertex is never deleted from S. A new s-vertex is always inserted into S_2. Whenever the size of S_2 reaches the threshold of $\lceil n^{0.5}\rceil$, we add all the vertices in S_2 to S_1 and reset S_2 to the empty set. When vertices are moved from S_2 to S_1 the edges in (1) and (2) above must be recomputed, and this may be accomplished in $O(n \log n)$ time as follows. We utilize the data structures for Problem 1 described in § 2.1. We construct the weighted Voronoi diagram/range tree (WVD/RT) for the points in S_1, and for each vertex v_i that is a free vertex or a t-vertex, we compute *shortest*$[v_i, S_1]$ by querying the WVD/RT for S_1. This requires $O(n \log n)$ time by Lemma 1 in § 2.1. Then using the structure trees of the t-blossoms and the free blossoms, the edges in (1) and (2) above can be computed in $O(n)$ extra time. As the recomputation is done at most $\sqrt{n}$ times per phase, the time for the recomputation is $O(n^{1.5} \log n)$ for the entire phase.

Note that for each vertex v_i in a large t-blossom or in a large free blossom, the weight $w(v_i)$ equals the value of dual variable α_i at the beginning of the phase, and so the edges in (1) and (2) above need to be recomputed only when S_1 changes.

Next, we show that the time required for updating the edges in (1), (2), and (3) above whenever a large blossom becomes free or a large free blossom gets labelled or a vertex is inserted into S_2 is $O(\sqrt{n}(\log n)^2)$.

(a) Suppose a t-blossom expands and one of its subblossoms, say B, becomes free. Also, suppose B is a large blossom. Then the edge *shortest*$[B, S_1]$, which is one of the edges in (2) above, is inserted into the priority queue in (1) above. For each vertex $v_j \in S_2$, the edge *shortest*$[v_j, B]$ is computed in $O((\log n)^2)$ time using the data structure described at the start of the section and is inserted into the priority queue in (3) above. This leads to a cost of $O\sqrt{n}(\log n)^2)$ operations when a large blossom becomes free.

(b) Suppose a large free blossom gets a t-label/s-label. Then the $O(\sqrt{n})$ edges that are incident to vertices in B are deleted from the priority queues in (1) and (3) above.

(c) Suppose a vertex v_j is inserted into S_2. Then for each large free blossom B, we find *shortest*$[v_j, B]$ in $O((\log n)^2)$ time, and insert it into the priority queue in (3) above. Since at most $\sqrt{n}$ large free blossoms can be present at any given time, an insertion into S_2 costs $O(\sqrt{n}(\log n)^2)$ operations.

From Lemma 4 (§ 4.2), it follows that the number of times blossoms become free or get labelled is $O(n)$, and the number of insertions into S_2 is also $O(n)$. So from (a)-(c) and the bound on the time for recomputing the edges in (1) and (2) above, we may conclude that the time for maintaining δ_{11} is $O(n^{1.5}(\log n)^2)$ per phase.

4.4.2. Maintaining δ_{12}. In this case too S is partitioned into S_1 and S_2 such that $|S_2| \leqq \sqrt{n}$. The free vertices in the small free blossoms are partitioned into $F_1, F_2, \cdots, F_{\lceil n^{0.5}\rceil}$ (some of the F_i's could be empty) such that for each small free blossom B

$$F_i \cap B \neq \phi \Rightarrow B \subseteq F_i, \qquad i = 1, \cdots, \lceil n^{0.5}\rceil,$$

and

$$|F_i| \leqq 2(\sqrt{n}+1), \qquad i = 1, \cdots, \lceil n^{0.5}\rceil.$$

We maintain the following edges and data structures:

(1) For each vertex v_k that is in a small free blossom, the edge *shortest*$[v_k, S_1]$. All these edges are in a priority queue with the priority of an edge being given by its slack.

(2) For each vertex $v_j \in S_2$, the edges *shortest*$[v_j, F_i]$, $1 \leqq i \leqq \lceil n^{0.5} \rceil$. All these edges are also in a priority queue with the priority of an edge being given by its slack.

(3) The weighted Voronoi diagram/range tree (WVD/RT) for S_1 and for each F_i, $1 \leqq i \leqq \sqrt{n}$. For each vertex v_k in a small free blossom, $\alpha_k = w(v_k)$, and for each vertex $v_j \in S$, $\alpha_j = w(v_j) + \Delta$. Thus if F' is a subset of the set of vertices in small free blossoms and $S' \subseteq S$, then

$$\mathit{slack}[\mathit{shortest}[F', S']] = \min_{v_k \in F', v_j \in S'} \{\mathit{slack}[(v_k, v_j)]\}.$$

Hence δ_{12} is achieved either for the edge in (1) above with minimum priority or for the edge in (2) above with minimum priority, and can be computed in $O(\log n)$ time by examining these edges.

A new s-vertex is always inserted into S_2. Whenever the size of S_2 reaches the threshold of $\sqrt{n}$, all the vertices in S_2 are added to S_1 and S_2 is reset to the null set, and the edges in (1) above are recomputed. By Lemma 1 (§ 2.1) the recomputation may be done in $O(n \log n)$ time using the WVD/RT for S_1. So the time for recomputation is $O(n^{1.5} \log n)$ per phase.

Next, we show that the time for updating the edges in (1) and (2) above when a small blossom becomes free or a small free blossom gets labelled or a vertex is inserted in S_2 is $O(\sqrt{n} \log n)$.

(a) Suppose a small blossom B becomes free. Let F_i be such that $|F_i| \leqq \sqrt{n}$. (By the pigeonhole principle, there always exists such an F_i.) All the vertices in B are added to F_i, the WVD/RT for F_i is recomputed, and then for each $v_j \in S_2$, *shortest*$[v_j, F_i]$ is recomputed in $O(\log n)$ time by querying the WVD/RT for F_i. For each vertex $v_k \in B$, the edge *shortest*$[v_k, S_1]$ is found in $O(\log n)$ time by querying the WVD/RT for S_1. So when a small blossom B becomes free it costs $O((\sqrt{n} + |B|) \log n) = O(\sqrt{n} \log n)$ operations.

(b) Suppose a small free blossom B gets a t-label/s-label. First, the $O(\sqrt{n})$ edges in the above priority queues that are incident to vertices in B are deleted. Let $B \subseteq F_i$. All the vertices in B get deleted from F_i, the WVD/RT for F_i is recomputed, and then for each $v_j \in S_2$, *shortest*$[v_j, F_i]$ is recomputed by querying the WVD/RT for F_i. So when a small free blossom gets labelled it costs $O(\sqrt{n} \log n)$ operations.

(c) Suppose a vertex v_j is inserted into S_2. Then we have to compute *shortest*$[v_j, F_i]$, $1 \leqq i \leqq \lceil n^{0.5} \rceil$. So the cost in this case is also $O(\sqrt{n} \log n)$ operations.
From Lemma 4 (§ 4.2) it follows that the number of times blossoms become free or get labelled is $O(n)$, and the number of insertions into S_2 is also $O(n)$. From (a)-(c), and the time bound for recomputing the edges in (1) above, we may then conclude that the time for maintaining δ_{12} is $O(n^{1.5} \log n)$.

4.5. Maintaining δ_2 during a phase. In this section we will describe how to maintain δ_2, and an associated edge (v_i, v_j) for which δ_2 is achieved, where v_i, v_j, are s-vertices in distinct s-blossoms. As before, the slack associated with an edge (v_i, v_j), is given by the quantity $d(v_i, v_j) - \alpha_i - \alpha_j$, and denoted by *slack*$[(v_i, v_j)]$. Note that for each $v_i \in S$, $\alpha_i = w(v_i) + \Delta$. Thus if $S' \subseteq S$ and $S'' \subseteq S$, then

$$\mathit{slack}[\mathit{shortest}[S', S'']] = \min_{v_i \in S', v_j \in S''} \{\mathit{slack}[(v_i, v_j)]\}.$$

So we can use the weights rather the dual variables in maintaining δ_2. Let

$$\delta_{21} = \min_{v_i, v_j, \text{ are in distinct large } s\text{-blossoms}} \{slack[(v_i, v_j)]\},$$

$$\delta_{22} = \min_{\substack{v_i \text{ is in a small } s\text{-blossom} \\ v_j \text{ is in a large } s\text{-blossom}}} \{slack[(v_i, v_j)]\},$$

and

$$\delta_{23} = \min_{v_i, v_j \text{ are in distinct small } s\text{-blossoms}} \{slack[(v_i, v_j)]\}.$$

Clearly,

$$\delta_2 = \tfrac{1}{2} \min \{\delta_{21}, \delta_{22}, \delta_{23}\}.$$

In §§ 4.5.1-4.5.3, we describe how to maintain δ_{21}, δ_{22}, and δ_{23}, in $O(n^{1.5}(\log n)^2)$, $O(n^{1.5} \log n)$, and $O(n^{1.5}(\log n)^4)$, time per phase, respectively.

Consider a specific phase. Let B be a blossom that has an s-label sometime during the phase. An s-subblossom (t-subblossom) of B is a subblossom of B that has an s-label (t-label), sometime during the phase, prior to B getting its s-label. We require the following easily proven lemma.

LEMMA 5. *Consider the s-blossoms that are present at the end of a particular phase. With each such s-blossom $\hat{B}$ is associated a tree whose root is $\hat{B}$ itself, and the leaves are all the vertices in $\hat{B}$. Each nonleaf node in this tree is a blossom that had an s-label sometime during the phase. The sons of a nonleaf node (blossom) B are as follows. Each son of B that is not a leaf is an s-subblossom of B. Each son of B that is a leaf is a vertex in B that has switched status from free vertex/t-vertex to s-vertex as a result of B getting an s-label. Let τ denote the forest of such trees associated with the s-blossoms at the end of the phase. Note that for a blossom (node) B in forest τ, the number of vertices in B (i.e., $|B|$) equals the number of leaves in the subtree rooted at B.*

For each node B in the forest τ, define $\sigma(B)$ as follows:

(1) *For a leaf B, let $\sigma(B) = 1$.*

(2) *Let B be a nonleaf node. Consider vertices to be trivial blossoms of size* 1. *Let B' be a son of B such that B' contains the largest number of vertices among all the sons of B. We let $\sigma(B) = |B| - |B'|$.*

Then

$$\sum_{B \text{ a node in forest } \tau} \sigma(B) = O(n \log n). \qquad \square$$

4.5.1. Maintaining δ_{21}. For each pair B, $\hat{B}$, where B and $\hat{B}$ are distinct large s-blossoms, we maintain the edge *shortest*$[B, \hat{B}]$. The edges are in a priority queue, with the priority of an edge (v_i, v_j) being *slack*$[(v_i, v_j)]$. δ_{21} is achieved for an edge with the minimum priority in this priority queue. We also maintain a semidynamic weighted Voronoi diagram/range tree (WVD/RT) described in § 2.3 for each large s-blossom. Note that once the semidynamic WVD/RT for a blossom B is available, *shortest*$[v_i, B]$ is computable in $O(\log n)^2)$ time for any vertex v_i. The shortest edges between large s-blossoms are updated as follows:

(1) Suppose a large blossom B gets an s-label in Case 1 or Case 3 in § 4.2. Then each vertex in B switches status to s-vertex. For each large s-blossom $\hat{B}$ (other than B), we compute the edge *shortest*$[v_i, \hat{B}]$ for all vertices v_i in B by querying the WVD/RT for $\hat{B}$, and use these edges to find *shortest*$[B, \hat{B}]$. We also build the semidynamic WVD/RT for B. As the number of large s-blossoms present at any time is $O(\sqrt{n})$, we spend $O(|B|\sqrt{n}(\log n)^2)$ time when B gets an s-label.

(2) Suppose a new large s-blossom B is created in Case 2 in § 4.2. Let C be the set of all vertices v in B such that v is either in a t-subblossom of B or in a

small s-subblossom of B. Note that each subblossom of B is either a t-subblossom or an s-subblossom, and that each vertex in each t-subblossom of B switches status to s-vertex when B receives an s-label.

(2.1) For each large s-subblossom B' of B, we delete from the above priority queue the $O(\sqrt{n})$ edges which are incident to vertices in B'.

(2.2) Let $\hat{B}$ be a large s-blossom other than B. For each vertex v_i in C, the edge *shortest*$[v_i, \hat{B}]$ is found by querying the semidynamic WVD/RT for $\hat{B}$. Using these edges together with the edges *shortest*$[B', \hat{B}]$, where B' are the large s-subblossoms of B, *shortest*$[B, \hat{B}]$ is obtained, and inserted into the above priority queue. Since at most $O(\sqrt{n})$ large s-blossoms may be simultaneously present and since the number of subblossoms of B is $O(|C|)$, the time to compute *shortest*$[B, \hat{B}]$ for all large s-blossoms $\hat{B}$ is $O(|C|\sqrt{n}(\log n)^2)$.

(2.3) Let B' be an s-subblossom of B such that B' has the greatest size among all the s-subblossoms of B.

(2.3.1) If B' is a small blossom, then $B = C$ and the semidynamic WVD/RT for B is constructed in $O(|C|(\log n)^2)$ time.

(2.3.2) If B' is a large blossom, then the semidynamic WVD/RT for B is obtained by inserting all the vertices in $B - B'$ into the semidynamic WVD/RT for B'. So there are $\sigma(B)$ insertions into a semidynamic WVD/RT where $\sigma(B)$ is as defined in Lemma 5.

A upper bound of $O(n^{1.5}(\log n)^2)$ on the time per phase for the computations in (1), (2.1), and (2.2) follows from the following observations. First, the total number of status switches of vertices to s-vertices is $O(n)$ per phase. Second, the number of blossoms labelled during a phase is $O(n)$, and so the number of s-subblossoms generated during a phase is $O(n)$. Third, for each vertex v, the condition that v is in a small s-subblossom of a large s-blossom can occur at most once during a phase. From these observations it also follows that the time spent in (2.3.1) is $O(n(\log n)^2)$ per phase. By Lemma 5, the total number of insertions into semidynamic WVD/RT's in (2.3.2) is $O(n \log n)$ per phase, and so the total time per phase spent in (2.3.2) is $O(n(\log n)^3)$ at the average rate of at most $O((\log n)^2)$ per insertion. Thus the time for maintaining δ_{21} is $O(n^{1.5}(\log n)^2)$ per phase.

4.5.2. Maintaining δ_{22}.

The procedure for maintaining δ_{22} is similar to the one for maintaining the minimum slack δ in the bipartite case discussed in § 3. Let S_L denote the set of those vertices in S that are in large s-blossoms, and let S_M denote the set of those vertices in S that are in small s-blossoms. S_L is partitioned into S_{L1}, S_{L2}, such that $|S_{L2}| \leqq \sqrt{n}$. S_M is partitioned into $S_{M1}, S_{M2}, \cdots, S_{Mr}$, $r = \lceil n^{0.5} \rceil$, such that $|S_{Mi}| \leqq 2(\lceil n^{0.5} \rceil + 1)$, $1 \leqq i \leqq \lceil n^{0.5} \rceil$. (Some of the S_{M_i}'s could possibly be empty.) The following information is maintained:

(i) For each $v_i \in S_M$, the edge *shortest*$[v_i, S_{L1}]$. These edges are in a priority queue, with the priority of an edge being its slack.

(ii) For each $v_j \in S_{L2}$, the edges *shortest*$[v_j, S_{Mi}]$, $1 \leqq i \leqq \lceil n^{0.5} \rceil$. These edges too are in a priority queue with the priority of an edge being given by its slack.

(iii) The WVD/RT for S_{L1} and for each S_{Mi}, $1 \leqq i \leqq \lceil n^{0.5} \rceil$.

Clearly, δ_{22} is achieved for either the edge in (i) with minimum priority or the edge in (ii) with minimum priority.

Initially, S_{L1} contains all the vertices in the large exposed blossoms, and $S_{L2} = \phi$. When a new vertex is added to S_L, it is always inserted into S_{L2}. When the size of S_{L2} reaches the threshold of $\lceil n^{0.5} \rceil$, all the vertices in S_{L2} are moved over to S_{L1}, and the edges in (i) are recomputed. By Lemma 1 (§ 2.1), using the WVD/RT for S_{L1} the recomputation may be performed in $O(n \log n)$ time, leading to a time of $O(n^{1.5} \log n)$ per phase for the recomputation.

Suppose there is an insertion into S_M, say v_i is inserted into S_{M_i}. Then the WVD/RT for S_{Mi} and the edges *shortest*$[v_j, S_{Mi}]$ for the vertices v_j in S_{L2} must be recomputed, and *shortest*$[v_i, S_{L1}]$ must be computed and inserted into the priority queue in (i). So an insertion into S_M costs $O(n^{0.5} \log n)$ time. Similarly, a deletion from S_M costs $O(n^{0.5} \log n)$ time. Insertion of a vertex v_j into S_{L2} also costs $O(n^{0.5} \log n)$ time, as the edges *shortest*$[v_j, S_{Mi}]$, $1 \leqq i \leqq \lceil n^{0.5} \rceil$ have to be computed. (Note that there are no deletions from S_L.) The number of insertions into S_L and S_M is $O(n)$ per phase. So the time for all the insertions into S_{L2}, and all the insertions into and deletions from S_M, is $O(n^{1.5} \log n)$ per phase.

Thus the time for maintaining δ_{22} is $O(n^{1.5} \log n)$ per phase.

4.5.3. Maintaining δ_{23}. Next, we show how to maintain δ_{23}. Let the vertices in S be ordered by the following rule. For $v_i, v_j \in S$, $v_i < v_j$ if and only if v_i was added to S before v_j. Note that for an s-blossom B, the ordering on the vertices in S induces a partition of $S - B$ into at most $|B| + 1$ intervals. Let $u_1 < u_2 < \cdots < u_{|S|}$ denote the ordered sequence of the vertices in S. By Lemma 3 (§ 2.3), we can maintain a semidynamic data structure for S (that is a dynamized version of the data structure for Problem 2 in § 2) such that:

(I) The total cost of inserting all the points in S, and thus the total time for maintaining the data structure for an entire phase is $O(n(\log n)^3)$.

(II) Given an interval $[u_i, u_j)$, $1 \leqq i < j \leqq |S| + 1$, and a vertex v_k, *shortest*$[v_k, [u_i, u_j)]$ may be computed in $O((\log n)^3)$ time.

For each small s-blossom B the following edges incident on the vertices in B are maintained. Let $|B| = r$, and let $u_{i_1} < u_{i_2} < \cdots < u_{i_r}$ be the ordered sequence of the vertices in the small s-blossom B. Then $S - B$ can be expressed as the disjoint union of the intervals $[u_1, u_{i_1})$, $[u_{i_1+1}, u_{i_2})$, $\cdots$, $[u_{i_{r-1}+1}, u_{i_r})$, $[u_{i_r+1}, u_{|S|+1})$. (Some of the intervals could possibly be empty.) We shall maintain the edge *shortest*$[B, [u_1, u_{i_1})]$, and the edges *shortest*$[B, [u_{i_k+1}, u_{i_{k+1}})]$, $1 \leqq k < |B|$. (Note that *shortest*$[B, [u_{i_r+1}, u_{|S|+1})]$ is not included.)

All these edges corresponding to the small s-blossoms are placed in a priority queue, with the priority of an edge given by its slack. It is easily seen that if (u_i, u_j) is an edge in this priority queue with the minimum priority then *slack*$[(u_i, u_j)] \leqq \delta_{23}$.

The computations required to maintain the above edges associated with small s-blossoms are broken down into two cases:

(a) An existing small blossom B gets an s-label. This happens in Case 1 and Case 3 in § 4.2, and the vertices in the newly labelled blossom B switch status from free vertices/t-vertices to s-vertices. All the vertices in B get added to S and so B corresponds to a single interval in the ordering on S. Moreover, $S - B$ is also an interval. For each vertex $u_{i_k} \in B$, *shortest*$[u_{i_k}, S - B]$ is found in $O((\log n)^3)$ time by querying the data structure for S, and using these edges *shortest*$[B, S - B]$ is obtained in $O(|B|)$ extra time. So the time spent is $O(|B|(\log n)^3)$.

(b) A new small blossom B is discovered and gets an s-label. This happens in Case 2 in § 4.2. Let B' be an s-subblossom of B such that B' has the largest size among all the s-subblossoms of B. Let $|B| = r$, and let $|B'| = m$. Let $u_{i_1} < u_{i_2} < \cdots < u_{i_r}$ be the vertices in B, and let $u_{j_1} < u_{j_2} < \cdots < u_{j_m}$ be the vertices in B'. Let J be the set of intervals defined by

$$J = \{[u_1, u_{i_1}), [u_{i_1+1}, u_{i_2}), \cdots, [u_{i_{r-1}+1}, u_{i_r})\},$$

and J' be the set of intervals defined by

$$J' = \{[u_1, u_{j_1}), [u_{j_1+1}, u_{j_2}), \cdots, [u_{j_{m-1}+1}, u_{j_m})\}.$$

Note that $(V-B)\cap[u_1, u_{i_r})$ can be expressed as the disjoint union of all the intervals in J, and that $(V-B')\cap[u_1, u_{j_m})$ can be expressed as the disjoint union of all the intervals in J'. Furthermore, $|J\cap J'| \geqq |B'|-(|B|-|B'|)$ and so $|J-J'| \leqq 2(|B|-|B'|)$. The edges *shortest*$[B, I]$ for all the intervals I in J may be obtained in $O(|B|(|B|-|B'|)\times(\log n)^3)$ time as follows. For each interval I' in $J-J'$, we compute *shortest*$[u', I']$ for all vertices u' in B'. For each interval I in J, we compute *shortest*$[u, I]$ for all vertices u in $B-B'$. Each edge is obtained by querying the data structure for S (described in § 2.3) in $O((\log n)^3)$ time, and all these edges may be obtained in $O(|B|(|B|-|B'|)\times(\log n)^3)$ time. Then as the edges *shortest*$[B', I']$, $I'\in J\cap J'$, are already available, the edges *shortest*$[B, I]$, $I\in J$, can be found in $O(|B|(|B|-|B'|)\log n)$ additional time. The time per phase for the computations in (a) above is $O(n(\log n)^3)$, since all the blossoms labelled by an s-label in Case 1 and Case 3 in § 4.2 are disjoint. As the size of a small s-blossom is at most $\sqrt{n}$, $O((|B|-|B'|)\sqrt{n}(\log n)^3)$ time is spent in (b) per each new blossom B, where B' is an s-subblossom of B that contains the largest number of vertices among all the s-subblossoms of B. In other words, $O(\sigma(B)\sqrt{n}(\log n)^3)$ time is spent in (b) above per each new s-blossom B, where $\sigma(B)$ is as defined in Lemma 5. Then we may apply Lemma 5 and conclude that the time per phase spent in performing the computations in (b) above is $O(n^{1.5}(\log n)^4)$. Thus the time per phase for maintaining δ_{23} is $O(n^{1.5}(\log n)^4)$.

5. Conclusion.

We have shown that the underlying geometry can be exploited to speed up algorithms for weighted matching when the vertices of the graph are points on the plane, and the weight of an edge between two points is the distance between the points under some metric. The techniques described in the paper can be used to speed up algorithms for related problems such as bottleneck matching [11] for points on the plane, and the transportation problem [11], [14] where the sources and the sinks are located on the plane and the cost of transporting from a source to a sink is proportional to the distance between the source and the sink. The techniques in the paper can also be utilized to speed up scaling algorithms [7], [18] for matching and related problems by a factor of about $\sqrt{n}$ for points on the plane. Finally, we note that for the L_1 and the L_∞ metrics the algorithms in the paper easily extend to the case where the vertices of the graph are points in d-dimensional space (d fixed) rather than points on the plane. For points in d-dimensional space we use d-dimensional range trees instead of two-dimensional range trees [15], and this increases the running time of the matching algorithms by at most $O(\log n)^d$.

Acknowledgments. The author thanks F. P. Preparata for suggesting the problem of matching on the plane, P. Shor, S. Fortune, and H. Edelsbrunner for helpful discussions, and M. R. Garey and D. S. Johnson for useful comments and suggestions concerning the presentation of this paper.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
[2] D. Avis, *A survey of heuristics for the weighted matching problem*, Networks, 13 (1983), pp. 475–493.
[3] H. Edelsbrunner, L. J. Guibas, and J. Stolfi, *Optimal point location in a monotone subdivision*, Tech. Report, DEC Systems Research Center, Palo Alto, CA, 1984.

[4] J. EDMONDS, *Maximum matching and a polyhedron with* 0, 1*-vertices*, J. Res. Nat. Bur. Standards, 69B (1965), pp. 125–130.

[5] S. FORTUNE, *A sweepline algorithm for Voronoi diagrams*, in Proc. ACM Annual Symposium Computational Geometry, Association for Computing Machinery, New York, 1986, pp. 313–322.

[6] H. N. GABOW, *An efficient implementation of Edmond's algorithm for maximum matching on graphs*, J. Assoc. Comput. Mach., 23 (1976), pp. 221–234.

[7] H. N. GABOW AND R. E. TARJAN, *Faster scaling algorithms for network problems*, Tech. Report, Department of Computer Science, Princeton University, Princeton, NJ, 1987.

[8] Z. GALIL, S. MICALI, AND H. N. GABOW, *Priority queues with variable priority and an $O(EV \log V)$ algorithm for finding a maximal weighted matching in general graphs*, in Proc. 22nd Annual IEEE Symposium Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1982, pp. 255–261.

[9] M. IRI, M. MUROTA, AND S. MATSUI, *Linear time heuristics for minimum-weight perfect matching on a plane with application to the plotter problem*, unpublished manuscript.

[10] H. W. KUHN, *The Hungarian method for the assignment problem*, Naval Res. Logist. Quart., 2 (1955), pp. 83–97.

[11] E. LAWLER, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.

[12] D. T. LEE AND F. P. PREPARATA, *Location of a point in a planar subdivision and its applications*, SIAM J. Comput., 6 (1977), pp. 594–606.

[13] K. MEHLHORN, *Data Structures and Algorithms* 3: *Multidimensional Searching and Computational Geometry*, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, Berlin, New York, pp. 2–9, 1984.

[14] C. H. PAPADIMITRIOU AND K. STEIGLITZ, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[15] F. P. PREPARATA AND M. I. SHAMOS, *Computational Geometry: An Introduction*, Springer-Verlag, Berlin, New York, 1985.

[16] M. SHARIR, *Intersection and closest-pair problems for a set of planar discs*, SIAM J. Comput., 14 (1985), pp. 448–468.

[17] K. J. SUPOWIT AND E. M. REINGOLD, *Divide-and-conquer heuristics for minimum weighted Euclidean matching*, SIAM J. Comput., 12 (1983), pp. 118–144.

[18] H. N. GABOW AND R. E. TARJAN, *Faster scaling algorithms for graph matching*, Tech. Report, Department of Computer Science, Princeton University, Princeton, NJ, 1987.