Approximation algorithms for bipartite and non-bipartite matching in the plane *

Kasturi R. Varadarajan† Pankaj K. Agarwal‡

July 7, 1998

Abstract

In the approximate Euclidean min-cost perfect matching problem, we are a given a set V of $2n$ points in the plane, and a real number $\varepsilon > 0$, and we want pair up the points (into n pairs) so that the sum of the distances between the paired points is within a multiplicative factor of $(1+\varepsilon)$ of the optimal. The recent work of Arora [2] gives a Monte-Carlo algorithm for this problem that with probability at least $1/2$ returns a solution that is within $(1+\varepsilon)$ of the optimal; the running time of the algorithm is $O(n \log^{O(1/\varepsilon)} n)$. We present a Monte-Carlo algorithm that also returns, with probability at least $1/2$, a solution with $(1+\varepsilon)$ of the optimal; the running time of our algorithm is $O(n \log^6 n/\varepsilon^3)$. Thus, the running time of our algorithm is polynomial in $1/\varepsilon$ in contrast to Arora's algorithm. Our algorithm combines Arora's geometric partitioning scheme with a divide-and-conquer approach that we developed for exact matching of points in the plane [14].

In the bipartite version of this problem, we are given a set R of 'red' points, a set B of 'blue' points in the plane, and a real number $\varepsilon > 0$. We want to pair up each red point with a blue point so that the sum of the distances between paired points is within $(1+\varepsilon)$ of the optimal. For this problem, we do not know of any work that improves upon the near-quadratic algorithm of Agarwal et al. [1] that solves the exact version of the problem. One reason is that the techniques used for the non-bipartite case do not seem to apply here. We present an algorithm for this problem that runs in $O(n^{3/2} \log^5 n/\varepsilon^2)$ time.

* Work on this paper has been supported by National Science Foundation Grant CCR-93–01259, by an Army Research Office MURI grant DAAH04-96-1-0013, by a Sloan fellowship, by an NYI award, by matching funds from Xerox Corporation, and by a grant from the U.S.–Israeli Binational Science Foundation.

† Department of Computer Science, Box 90129, Duke University, krv@cs.duke.edu

‡ Department of Computer Science, Box 90129, Duke University, pankaj@cs.duke.edu

1 Introduction

In the approximate (Euclidean) min-cost matching problem, we are given a set V of $2n$ points in the plane, and a real number $\varepsilon > 0$. A *matching* of V is a collection M of unordered pairs of V such that no point in V is incident on more than one pair in M. A *perfect* matching of V is a matching M in which every point in V is incident on *exactly* one pair of M. Note that a perfect matching of V has n pairs. We define the *cost* of a matching M to be the sum of the Euclidean distances between the paired points. The problem is to find a perfect matching whose cost is at most $(1+\varepsilon)$ times the cost of a min-cost perfect matching.

In the approximate (Euclidean) bipartite min-cost matching problem, we are given a set R of n 'red' points and a set B of n 'blue' points in the plane, and a real number $\varepsilon > 0$. Here, the pairs of the matching are restricted to be red-blue pairs. The problem is to find a perfect red-blue matching of $R \cup B$ whose cost is at most $(1+\varepsilon)$ times the cost of a min-cost perfect red-blue matching.

These problems have applications in operations research, pattern recognition, shape matching, statistics, and VLSI. The first polynomial time algorithm (on general graphs) for (exact) min-cost bipartite matching is due to Kuhn [9], and for min-cost non-bipartite matching is due to Edmonds [4]. The fastest known implementations of these algorithms run in $O(|V|^3)$ time on dense graphs (see Lawler [10]) and roughly $O(|E||V|)$ time on sparse graphs [8]. For the Euclidean (planar) versions of these problems, Vaidya [13] showed that geometry can be exploited to get algorithms running in $O(n^{5/2}\log^{O(1)} n)$ time for both the bipartite and non-bipartite versions. Agarwal et al. [1] improved the running time for the bipartite case to $O(n^{2+\delta})$, for any $\delta > 0$. Very recently, Varadarajan [14] gave a divide-and-conquer algorithm for planar non-bipartite matching that runs in $O(n^{3/2}\log^{O(1)} n)$ time.

For approximate min-cost matching in the plane, Vaidya [12] gave an algorithm that runs in roughly $O(n^{3/2}/\varepsilon^3)$ time. Recently, Arora [2] gave a Monte-Carlo algorithm for this problem that runs in $O(n\log^{O(1/\varepsilon)} n)$ time and returns a correct solution with high probability. Building on his approach, Rao and Smith [11] give a Monte-Carlo algorithm that runs in $O(n\log n)$ time and produces (with probability atleast $1/2$) a matching whose cost is within a constant factor of the optimal. For work on approximation algorithms for non-bipartite minimum matching that use an absolute measure, see the survey by Avis [3], and the references therein. We are not aware of any previous work on approximation algorithms for min-cost bipartite matching in the plane.

Our results We present an algorithm for approximate min-cost matching that runs in $O(n\log^6 n/\varepsilon^3)$ time; the algorithm returns a matching that is within $(1+\varepsilon)$ of the optimal with probability atleast $1/2$. Of course, the probability of success can be increased by iterating the algorithm and taking the smallest matching returned. Arora's algorithm [2] achieves the same result, but with a running of $O(n\log^{O(1/\varepsilon)} n)$, which is exponential in $1/\varepsilon$. The running time of our algorithm is polynomial in $1/\varepsilon$. We achieve this by using a divide-and-conquer approach developed in [14] on top of the partitioning scheme of Arora. (Arora's technique uses dynamic programming on top of his partitioning scheme, which is why there is the exponential dependence on $1/\varepsilon$; Of course, his scheme is general and works for strongly NP-hard problems like the travelling salesman problem, whereas we are taking advantage of the fact that matching is solvable in polynomial time.)

The techniques that have been used by Vaidya [12] and Arora [2] for approximate min-cost non-bipartite matching do not seem to carry over to the bipartite case. No algorithms for approximate bipartite min-cost matching are known whose running time dependence on n is better than the near-quadratic algorithm of Agarwal et al. [1] for the exact case. We present an algorithm for this problem

that runs in $O(n^{3/2} \log^5 n/\varepsilon^2)$ time. The crux of our algorithm is an efficient implementation in geometry of the scaling algorihm of Gabow and Tarjan [7]. The main bottleneck is here is that there are $\Omega(n^2)$ red-blue pairs that the algorithm has to consider. To overcome this, we partition the red-blue edges into $O(\log n/\varepsilon)$ classes depending on their (approximate) length, and work with clique covers [6] of these classes rather than with each red-blue pair explicitly.

In Section 2, we describe our algorithm for approximate non-bipartite matching, and in Section 3, we describe our algorithm for approximate bipartite matching. Throughout the paper, we will make the assumption the $\varepsilon > 1/n$; this will simplify our running time expressions. The justification is that for $\varepsilon < 1/n$, the exact algorithms are anyway faster than our approximation algorithms.

2 Approximating non-bipartite matching

We are given a set V of $2n$ points in the plane, and a real number $\varepsilon > 0$, and we want to find a matching of V whose cost is within a multiplicative factor of $(1+\varepsilon)$ of the min-cost perfect matching. We first describe the partitioning scheme of Arora [2], based on which we define a graph $\mathcal{G}$ whose vertices are V and some additional 'Steiner' points. We then describe our divide-and-conquer algorithm for computing a min-cost matching on $\mathcal{G}$.

Using standard techniques, such as the ones described by Arora [2] or Rao-Smith [11], we can assume that the minimum distance between any two points in V is 8, and V lies in an L by L square where L is bounded by a polynomial in n, say n^3. We also assume, without loss of generality, that L is a power of 2, and that the bounding $L \times L$ square is aligned with the integer grid. We choose random integers a and b from $[0, L]$. We grow the width of the bounding square by L by extending it on the left by a lines of the integer grid, and extending it on the right by $L - a$ lines. Similarly, we grow the height by using the random variable b. We now have a $2L \times 2L$ square containing the points V.

We construct a quad-tree on this square using the following recursive procedure. Any stage of the recursion begins with a square K. If K contains at most one point in V, it is a leaf of the quad-tree; the recursion terminates and we return. Otherwise, K is a non-leaf square of the quad-tree. We divide K into four equal squares $K_1, \ldots, K_4$ using a vertical median line and a horizontal median line. We place $O(\log n/\varepsilon)$ evenly spaced 'portals' on the vertical and horizontal lines exactly as in Arora's algorithm. We then recursively construct the quadtrees for $K_1, \ldots, K_4$.

We now construct a graph $\mathcal{G}$ that has for its vertices the set V and additional 'Steiner' points, which are the portals that we have placed. For each *leaf* square K of the quad-tree, we add an edge between ever pair of portals on the boundary of K. In addition, if K contains a point $v \in V$, we add an edge between v and each portal on the boundary of K. Since the quad-tree has depth $O(\log n)$, it has $O(n \log n)$ leaf squares overall, so the total number of vertices in $\mathcal{G}$ is $O(n \log^2 n/\varepsilon)$, and the total numer of edges is $O(n \log^3 n/\varepsilon^2)$. We define the *length* of an edge (p, q) in this graph to be the Euclidean distance between p and q. The *distance between two vertices p and q of $\mathcal{G}$* (where (p, q) is not necessarily an edge in $\mathcal{G}$) is defined to be the length of the shortest path between p and q in $\mathcal{G}$; we denote this by $\delta(p, q)$. In this section, we use the word 'edge' to denote an actual edge in the graph and the word 'pair' to denote any unordered pair $(u, v) \in V \times V$. The cost of a *matching M of V in $\mathcal{G}$* is defined to be the sum $\sum_{(u,v)\in M} \delta(u, v)$. (Note that the matching is a collection of pairs in $V \times V$.) The following lemma is an easy consequence of Arora's charging scheme:

Lemma 2.1 *With a probability of atleast* $1/2$*, the graph* $\mathcal{G}$ *produced by the above scheme has the property that the min-cost matching of* V *in* $\mathcal{G}$ *is within* $(1+\varepsilon)$ *of the min-cost matching of* V*.*

Our algorithm computes a min-cost matching of V in $\mathcal{G}$. Unlike the Arora algorithm, which uses dynamic programming to compute all possible solutions at each node of the quad-tree, our algorithm is a divide-and-conquer variant of Edmonds' matching algorithm. It is because of this alternative approach that we obtain a running time that is polynomial in $1/\varepsilon$, whereas Arora's algorithm is exponential in $1/\varepsilon$. Our algorithm can be viewed as an implementation on the graph $\mathcal{G}$ of the divide-and-conquer scheme that we developed for exact matching in the plane [14].

2.1 Min-cost matching of V in the graph $\mathcal{G}$

We will take the view that an edge (u, v) of the graph $\mathcal{G}$ is an actual link whose length is $d(u, v)$, the Euclidean distance between u and v. Thus, we will refer to 'the region of $\mathcal{G}$ whose distance from vertex v is at most r'; this region includes the vertices and all portions on the edges of $\mathcal{G}$ whose distance (in $\mathcal{G}$) from v is at most r. We will also call this region the *disk* of radius r centered at v.

We say that a subset $Q \subseteq V$ of V is an *odd subset* or an *odd-set* if $|Q|$ is odd and $|Q| \geq 3$. For $Q \subseteq V$, let $\xi(Q)$ denote the subset of pairs with exactly one endpoint in Q, that is, $\xi(Q) = \{(u, v) \in V \times V : |\{u, v\} \cap Q| = 1\}$.

Edmonds' algorithm is motivated by duality theory for linear programs; see [4] and [10] for a discussion of linear programming duality. His algorithm associates a 'dual variable' variable ω_v for each $v \in V$ and a dual variable ω_Q for each odd set Q. Sometimes, it will be convenient to denote ω_v by $\omega_{\{v\}}$. Corresponding to the pair (u, v), let $\pi_{uv} = \omega_u + \omega_v + \sum_{(u,v)\in\xi(Q)} \omega_Q$. From duality theory, it follows that a perfect matching M is optimal if there exist values ω_v, for each $v \in V$, and ω_Q, for each odd subset Q, such that the following conditions hold:

EDGE-FEASIBILITY: $\pi_{uv} \leq \delta(u, v)$ for each $(u, v) \in E$.

POSITIVE-DUAL: $\omega_Q \geq 0$ for each odd subset Q.

MATCHING-ADMISSIBILITY: $(u, v) \in M \Rightarrow \pi_{uv} = \delta(u, v)$.

MAXIMALITY: For each odd subset Q, if $\omega_Q > 0$, then the matching M is maximal within Q, that is, the number of pairs in M both of whose endpoints are in Q is $(|Q| - 1)/2$. Since M is a perfect matching, this is equivalent to $M \cap \xi(Q) = 1$.

Like Edmonds' algorithm, our approach also computes a perfect matching and a corresponding set of dual variables such that EDGE-FEASIBILITY, POSITIVE-DUAL, MATCHING-ADMISSIBILITY, and MAXIMALITY are satisfied. The difference is that unlike in Edmonds' algorithm, we use divide-and-conquer for doing this. Before describing our approach, we describe the important notion of blossoms that was introduced by Edmonds. Our description of blossoms and other standard components of the matching algorithm are based on the presentation of Galil et al.[8].

Definition 2.2 For any vertex $v \in V$, let $\lambda(v) = \omega_v + \sum_{v\in Q} \omega_Q$. A pair (u, v) is *feasible* if $\pi_{uv} \leq \delta(u, v)$. It is *admissible* if $\pi_{uv} = \delta(u, v)$.

2.2 Blossoms

During the course of our algorithm, certain odd subsets of V are designated as *blossoms*. The algorithm maintains the property that $\omega_Q > 0$ for an odd subset Q only if Q is a blossom. The set of blossoms at any stage have the following *nested* structure: For any two distinct blossoms B and B', either $B \cap B' = \emptyset$, or $B \subset B'$, or $B' \subset B$. Each $v \in V$ is a trivial blossom of size one. A

non-trivial blossom B is given by a sequence of blossoms $B_0, \ldots, B_r$, where $r = 2k$, for $k \geq 1$, and a sequence of admissible pairs $e_i = (u_{i-1}, v_i)$, for $i = 1, \ldots, r+1$, such that

1. $u_i, v_i \in B_{i \bmod (r+1)}$
2. For $1 \leq i \leq r+1$, $(u_{i-1}, v_i) \in M$ if i is even and $(u_{i-1}, v_i) \notin M$ if i is odd.

The blossoms $B_0, \ldots, B_r$ are referred to as the *subblossoms* of B. A blossom that is not a subblossom of any other blossom is called an *outermost* blossom. Clearly, the outermost blossoms induce a partition of V. It can be shown from the properties above that any blossom B contains an odd number of vertices, and that the matching M is maximal within B. The unique vertex of B that is not matched to any other vertex of B is called its *base*. The base can also be defined by induction on the structure of blossoms as follows. The base of a trivial blossom v is the vertex v itself. The base of a blossom B whose subblossoms are given by the sequence $B_0, \ldots, B_r$ (as above) is the base of B_0.

An *alternating path between vertices* v_0 and v_r is a sequence of admissible pairs $e_i = (v_{i-1}, v_i)$, for $i = 1, \ldots, r$, such that for $i = 1, \ldots, r-1$, $e_i \in M$ if and only if $e_{i+1} \notin M$. In other words, it is a path in which alternate pairs are in the matching. An *alternating path between outermost blossoms* B_0 and B_r is given by a sequence of admissible pairs $e_i = (u_{i-1}, v_i)$, for $i = 1, \ldots, r$, and a sequence of outermost blossoms $B_0, \ldots, B_r$, where $u_i, v_i \in B_i$, and for $i = 1, \ldots, r-1$, $e_i \in M$ if and only if $e_{i+1} \notin M$. We say that a vertex v is *exposed* if no pair of the matching M is incident on v; an outermost blossom B is exposed if no pair of the matching M is incident on the base of B. An alternating path between two exposed vertices is called an *augmenting path.*

Lemma 2.3 *Let u and v be points in different outer blossoms. The pair (u, v) is feasible (that is, $\pi_{uv} \leq \delta(u, v)$) iff $\lambda(u) + \lambda(v) \leq \delta(u, v)$. The pair (u, v) is admissible (that is, $\pi_{uv} = \delta(u, v)$) iff $\lambda(u) + \lambda(v) = \delta(u, v)$.*

Proof: Follows from the fact that if u and v are in different outer blossoms, $\pi_{uv} = \lambda(u) + \lambda(v)$. □

We can show that throughout our algorithm, $\lambda(v) \geq 0$ for any $v \in V$. We define disk(v), the *disk* of vertex v, to be the disk of radius $\lambda(v)$ centered at v; this disk is *not* the Euclidean disk but is defined on the graph $\mathcal{G}$. Since $\lambda(v) \geq 0$, disk(v) is well defined. Lemma 2.3 tells us that if u and v are vertices in different blossoms, feasibility of (u, v) means that disk(u) and disk(v) do not overlap (although they can touch); admissibility of (u, v) means disk(u) and disk(v) do not overlap but touch. (Note that we are using here the fact that the triangle inequality holds for distances along $\mathcal{G}$.)

2.3 The divide-and-conquer algorithm

Let K be a square in the quad-tree, and let $U \subseteq V$ be the set of points lying within K. We will describe our divide-and-conquer scheme for the set U. Let P be the set of portals on the boundary of Q. Let $\mathcal{G}(K)$ denote the sub-graph of $\mathcal{G}$ induced by U and the portals lying inside or on the boundary of K. The goal in the sub-problem for U is to compute a (not necessarily perfect) matching M of U, a set of blossoms in U, and a set of dual variables ω_u for each $u \in U$, and ω_Q for each blossom Q (the dual variables of odd sets that are not blossoms are assumed to be 0), so that (1) the conditions EDGE-FEASIBILITY, POSITIVE-DUAL, MATCHING-ADMISSIBILITY, and MAXIMALITY hold for U, and (2) in addition, the following two conditions are also satisfied:

RADIUS-CONSTRAINT: For each $u \in U$ and each portal $p \in P$, $\lambda(u) \leq \delta(u,p)$.

EXPOSED-CONSTRAINT: We say that an exposed blossom Q of U is *constrained* at a portal $p \in P$, if there is a $q \in Q$ and a portal $p \in P$ such that $\lambda(q) = \delta(q,p)$. We say that Q is *unconstrained* if it is not constrained at any portal in P. The EXPOSED-CONSTRAINT condition is that every exposed blossom of U be constrained.

We remark that the RADIUS-CONSTRAINT allows us to restrict our attention to $\mathcal{G}(K)$ for solving the sub-problem for U, because it restricts the disks of points in U to within $\mathcal{G}(K)$.

If K is not a leaf of the quad-tree, our algorithm recursively solves the sub-problems for $K_1, \ldots, K_4$, the four children of Q. Let $U_i \subseteq U$ be the set of points lying in K_i. Suppose that the recursive calls return a matching, blossoms, and dual variables for U_i satisfying the six conditions for U_i. To begin the conquer step for U, we obtain an initial matching, dual variables, and blossoms by combining the matching, dual variables, and blossoms for $U_1, \ldots, U_4$. At this stage, it is easy to see that all the six conditions except the EXPOSED-CONSTRAINT are satisfied for U. (Here, we use the fact that the portals on the boundary of a square K_i 'separate' the vertices of $\mathcal{G}(K_i)$ from the vertices of $\mathcal{G}$ lying outside K_i.)

Observe that the EXPOSED-CONSTRAINT condition may be violated for a blossom Q of U. The 'conquer' stage of the divide-and-conquer algorithm for U eliminates the violations of the EXPOSED-CONSTRAINT, thus 'solving' the sub-problem for U. The 'conquer' stage consists of a series of phases; in each phase the number of exposed, unconstrained blossoms, is reduced by either one or two.

2.4 The conquer stage

As we indicated, the conquer stage consists of phases. Each phase begins with the current matching M, a set of dual variables, and a set of blossoms. Some of the exposed blossoms are constrained, and are called c-blossoms. The algorithm always maintains the five conditions EDGE-FEASIBILITY, POSITIVE-DUAL, MATCHING-ADMISSIBILITY, MAXIMALITY, and RADIUS-CONSTRAINT. In each phase, the number of exposed, unconstrained, blossoms is decreased by one or two. Thus, each phase decreases the number of violations of the sixth condition EXPOSED-CONSTRAINT, and so the algorithm terminates after a finite number of phases.

During a phase, some unconstrained outer blossoms are *labelled* as s-blossoms and t-blossoms. (An outer blossom is labelled as either an s-blossom or a t-blossom, but not both.) An unconstrained outer blossom which is not labelled is called a *free* blossom or f-blossom. (s-, t-, and f- prefixes are only for unconstrained blossoms.) A vertex is called an s-vertex, t-vertex, or f-vertex according to whether it belongs, respectively, to an s-blossom, t-blossom, or f-blossom. We let S, T, and F denote, respectively, the set of s-vertices, t-vertices, and f-vertices. For any $v \in V$, let $b(v)$ denote the outermost blossom containing V.

A phase is divided into $O(m)$ *sub-phases*. At the end of each sub-phase, the following invariants hold. An exposed, unconstrained blossom is always an s-blossom. For every s- or t- blossom B, there is an alternating path $\sigma(B', B)$ between an exposed, unconstrained blossom B' and B. If B is an s-blossom, $\sigma(B', B)$ has even length, that is, there are an even number of edges in the alternating path. If B is a t-blossom, $\sigma(B', B)$ has odd length. The s- and t-blossoms, together with the corresponding alternating paths, induce a forest of rooted trees, a tree being rooted at each exposed, unconstrained blossom. The trees are called *alternating trees*, and the forest is called an *alternating forest*. (The c-blossoms are not in the alternating forest.) The leaves of the alternating trees are always s-blossoms.

For every f-blossom B, there is another f-blossom C such that there is a pair in matching M

between the bases of B and C. That is, M induces a perfect matching on the bases of all the f-blossoms.

At the start of the phase, we label each exposed, unconstrained blossom as an s-blossom; every other unconstrained outer blossom is an f-blossom. A sub-phase consists of the following loop, which is repeated until a termination condition for the phase is met. The above invariants hold at the end of each iteration of the loop. Let

$$\delta_1 = \min_Q \text{ a nontrivial } t\text{-blossom } \omega_Q \qquad \delta_2 = \min_{u\in S, v\in F}(\delta(u,v) - \pi_{uv})$$
$$\delta_3 = \min_{u,v\in S;\ b(u)\neq b(v)}(\delta(u,v) - \pi_{uv})/2 \qquad \delta_4 = \min_{u\in S, v \text{ a } c\text{-vertex}}(\delta(u,v) - \pi_{uv})$$
$$\delta_5 = \min_{u\in S, p\in P}(\delta(u,p) - \lambda(u))$$

and let $\delta = \min\{\delta_1, \delta_2, \delta_3, \delta_4, \delta_5\}$.

Dual change: Let ω_Q be the dual variable corresponding to the blossom Q. (If Q is a trivial blossom consisting of a vertex v, then $\omega_Q = \omega_v$.) For each s-blossom Q, we increase ω_Q by δ, and for each t-blossom Q, we decrease ω_Q by δ. After the dual change, one of δ_1, δ_2, δ_3, δ_4, or δ_5 becomes zero. In case of a tie, we pick an arbitarary δ_i that is zero. For technical reasons, δ_4 gets precedence over δ_5. We will be terse about some of the following cases, which are standard; see [8]

$\delta_1 = 0$: In this case, the dual variable ω_B corresponding to a (non-trivial) t-blossom B becomes zero. We expand B, that is, we stop regarding it as a blossom and make its subblossoms outer blossoms. Some of these new outer blossoms become s-blossoms, some become t-blossoms, and some f-blossoms.

$\delta_2 = 0$: In this case, a pair (u, v), which is now admissible, has been discovered; u is an s-vertex and v an f-vertex. Two f-blossoms are added to the alternating forest, one as a t-blossom and the other as an s-blossom.

$\delta_3 = 0$: A pair (u, v) which is now admissible has been discovered, where u and v are s-vertices. Either a new s-blossom is formed, or an alternating path between two exposed, unconstraned blossoms is discovered. The latter subcase ends the phase and is handled in a manner similar to the case where $\delta_4 = 0$.

$\delta_4 = 0$: A pair (u, v), which is now admissible, has been discovered; u is an s-vertex and v a c-vertex. Let A (resp. B) be the s-blossom (resp. c-blossom) containing u (resp. v). Let A' be the exposed, unconstrained, blossom which is the root of the alternating tree containing A, and let $\sigma(A', A)$ denote the corresponding even-length alternating path between A' and A. Note that $\sigma(A', A)$, the edge (u, v), and the blossom B together constitute an alternating path between the exposed blossoms A' and B. We expand this to an alternating path π between the exposed bases of A' and B. We augment the current matching M by excluding all pairs of M belonging to π and including the other pairs of π. Note that the cardinality of the matching M increases by one, and the number of exposed, unconstrained, blossoms falls by one since A' is now no longer exposed. We also change appropriately the bases of all the blossoms through which the augmenting path passes. This ends the current phase of the algorithm.

$\delta_5 = 0$: In this case, $\lambda(u)$ has increased to $\delta(u, p)$, where u is an s-vertex and p a portal on the boundary of Q. Let A be the s-blossom containing u. Let A' be the exposed, unconstrained, blossom which is the root of the alternating tree containing A, and let $\sigma(A', A)$ denote the corresponding even-length alternating path between A' and A. We expand $\sigma(A', A)$ to an even-length alternating path π between the bases of A' and A. We alter the current matching M by excluding all pairs of M belonging to π and including the other pairs of π. We change appropriately the bases of all the blossoms through which the augmenting path passes. This ends the current phase of the algorithm. We can show that the cardinality of the matching M remains unchanged, and the number of exposed, unconstrained, blossoms falls by one. Note that in the next phase, A is constrained.

This completes the description of a phase. At the end of the phase, we (recursively) expand all outer blossoms whose dual variable is zero.

This also completes our description of the overall divide-and-conquer scheme for min-cost perfect matching of V in $\mathcal{G}$. The following observation is useful in bounding the number of phases in the conquer stage.

Lemma 2.4 *The following invariant holds after each phase of the conquer stage for $U = K \cap V$: At each portal $p \in P$ at the boundary of the square K, at most one exposed blossom of U is constrained.*

Proof: The proof is based on the observation that when a second blossom is about to be constrained at a portal, we will be in the case $\delta_4 = 0$ (which we give precedence over $\delta_5 = 0$). □

Lemma 2.5 *The number of phases in the conquer step for U is $O(\log n/\varepsilon)$.*

For lack of space, we are unable to descibe how a phase of the conquer step is implemented. The main task is to detect when δ_i becomes zero; in particular, to detect when disks of two vertices in different blossoms of V touch, or when a disk touches a portal. This is not difficult once the notion of disks in $\mathcal{G}$, and some of their properties are understood. The ideas are quite similar, only simpler, than the near-linear time scheme we give in [14] to implement a phase of the exact matching algorithm for a set of n points in the plane. What we can show is that we can use the data structures similar to Galil et al. [8] to implement a single phase at a cost of $O(\log n)$ per edge of $\mathcal{G}$. This gives us an $O(n \log^4 n/\varepsilon^2)$ time algorithm for a single phase, and since there are $O(\log n/\varepsilon)$ phases, an $O(n \log^5 n/\varepsilon^3)$ for the conquer stage. We conclude with the main result of this section:

Theorem 2.6 *Given a set V of $2n$ points in the plane, and a real number $\varepsilon > 0$, we can compute a perfect matching of V whose cost is at most $(1+\varepsilon)$ times the optimal using an algorithm that runs in $O(n \log^6 n/\varepsilon^3)$ time.*

3 Approximating the min-cost bipartite matching

We are given a set R of n red points and a set B of n blue points in the plane, and a real number $\varepsilon > 0$; we wish to compute a perfect red-blue matching that is within a multiplicative factor of $(1+\varepsilon)$ of the optimal. We first 'clean up' the given instance of the problem. We then partition the red-blue edges into $O(\log n/\varepsilon)$ 'classes', and compute a clique cover of the edges in each class. We then show how the clique covers can be used to efficiently implement the scaling algorithm of Gabow and Tarjan on an appropriately defined graph.

The clean-up phase Let OPT be the cost of a min-cost red-blue matching of $V = R \cup B$. We first compute a crude approximation α to OPT such that $\alpha \leq OPT \leq n * \alpha$. One approach to do this is to use the algorithm of Efrat and Itai [5] to compute a *bottleneck matching* of R and B, which is a perfect red-blue matching that minimizes the *maximum* length red-blue pair in the matching. Using their algorithm, we can compute in $O(n^{3/2} \log n)$ time a bottleneck matching of R and B in the L_1 metric; let α be the largest distance among all pairs in the matching. Since an optimal min-cost matching must use a pair whose length is at least α, we have $\alpha \leq OPT$. Since the cost (under our measure) of the optimal bottleneck matching is at most $n * \alpha$, we have $OPT \leq n * \alpha$.

Consider a grid in the plane of spacing α/n^2, and move each point in $R \cup B$ to the nearest grid-point. We will use R and B to denote this new set of points. The cost of the new optimal

matching of R and B differs from the original one by a factor of at most $2n * OPT/n^2 = OPT/2n$, which is okay. We can now discard in pairs red and blue points that have moved to the same grid point. By scaling so that the grid-spacing is one, we now assume that the minimum red-blue distance is 1, and that the cost of the optimal red-blue matching is at most n^3; we can discard from consideration red-blue pairs that are more than n^3 apart. Let us therefore call the red-blue pairs whose length is between 1 and n^3 the *interesting pairs*. This ends the clean-up phase.

Clique covers of the interesting edges Instead of the Euclidean metric, it will be convenient to measure distances between points using a polygonal metric defined by a centrally-symmetric convex polygon having $O(1/\varepsilon)$ sides. Such a metric approximates the Euclidean metric to within a factor of $(1+\varepsilon)$.

We can partition the interval between 1 and n^3 on the real line into $O(\log n/\varepsilon)$ sub-intervals $I_1, \ldots, I_k$ so that the upper endpoint of each interval is within a factor of $(1+\varepsilon)$ of the lower end-point. For example, this can be done by partitioning the interval $[1,2)$ into $1/\varepsilon$ equally-spaced sub-intervals, the interval $[2,4)$ into $1/\varepsilon$ equally spaced sub-intervals, and so on. These sub-intervals induce a partition of the set of interesting pairs into $O(\log n/\varepsilon)$ classes— corresponding to each sub-interval I_i is the class C_i consisting of all interesting pairs whose lengths fall within the sub-interval I_i.

We compute a *bipartite clique cover*, or simply a clique cover, for the set of pairs in class C_i. This is a collection $\{(R_1, B_1), \ldots, (R_l, B_l)\}$ so that

1. $R_j \subseteq R$ and $B_j \subseteq B$, for $1 \leq j \leq l$
2. Every pair in $R_j \times B_j$ belonds to the class C_i
3. For each pair (r,b) in class C_i, there is a unique (R_j, B_j) such that $(r,b) \in R_j \times B_j$.

The size of the clique cover is defined to be $\sum_j (|R_j| + |B_j|)$. The size bounds the space needed to compactly represent the pairs in class C_i using a clique cover. Using standard range searching structures, we can compute a clique cover of the pairs in class C_i in $O((n/\varepsilon)\log^2 n)$ time [15]. The size of the clique cover is also $O((n/\varepsilon)\log^2 n)$. We compute $O(\log n/\varepsilon)$ clique covers, one for each class.

The Scaling Algorithm We approximate the lengths of all the pairs belonging to a class C_i by a single number n_i, which we choose to be the middle-point of the sub-interval I_i. By scaling all the numbers, we assume that the n_i's are all integers. Consider a graph $\mathcal{G}$ defined on $R \cup B$: $\mathcal{G}$ contains an edge (r,b) corresponding to each interesting pair (r,b); the cost of the edge (r,b) is set to n_i if (r,b) belongs to class C_i.

We use the scaling algorithm of Gabow and Tarjan [7] to compute a min-cost perfect matching in the graph $\mathcal{G}$; Clearly, this will yield a solution to our overall goal. However, we cannot afford to run the scaling algorithm on the graph $\mathcal{G}$ explicitly as the graph may have $\Omega(n^2)$ edges. Instead, we will show how the clique covers can be used to implement the scaling algorithm efficiently. We begin by giving a brief description of the scaling algorithm on the bipartite graph $\mathcal{G}$.

The algorithm associates a dual variable ω_w with each vertex $w \in R \cup B$. Let $c(e)$ denote the cost of an edge e. The costs on edges are integers; we will let N denote the largest cost. In our case, $N = O(n^3/\varepsilon)$.

A 1-*feasible matching* consists of a matching M and dual variables ω_w such that for any pair $(u, v) \in R \times B$,

$$\omega_u + \omega_v \leq c(u, v) + 1,$$
$$\omega_u + \omega_v = c(u, v), \text{ for } uv \in M$$

A 1-*optimal matching* is a perfect matching that is 1-feasible. If the $+1$ term is omitted from the first inequality, these are the usual complementary slackness conditions for a minimum perfect matching [10].

The scaling algorithm starts by computing a new cost $\overline{c}(e)$ for each equal e, equal to $n + 1$ times the given cost. Consider each $\overline{c}(e)$ to be a signed binary number $b_1 b_2 \cdots b_k$ having $k = \lfloor \log((n+1)N) \rfloor + 1$ bits. The scaling algorithm runs in k *scales*. It maintains a variable $c(e)$ for each edge e, equal to its cost in the current scale. At the beginning each $c(e)$ and each ω_w is set to 0. Then the following loop is executed with the loop index s going from 1 to k.

1. For each edge e, $c(e) \leftarrow 2c(e) +$ bit b_s of $\overline{c}(e)$. (Basically, $c(e)$ is set to the binary number represented by the the first s bits of $\overline{c}(e)$.) For each vertex v, $y(v) \leftarrow 2y(v) - 1$.

2. Call the procedure MATCH to find a 1-optimal matching with the current costs.

Each iteration of the above loop is called a scale. Thus, there are $O(\log(nN))$ scales. Gabow and Tarjan show that the 1-optimal matching computed at the end of the last scale is the min-cost perfect matching in $\mathcal{G}$. Before describing the procedure MATCH, we need a few definitions. Given a matching M, we call an edge (u, v) *eligible* if (1) $(u, v) \in M$ and $\omega_u + \omega_v = c(u, v)$, or (2) $(u, v) \notin M$ and $\omega_u + \omega_v = c(u, v) + 1$. In other words, an eligible edge is one for which the 1-feasibility constarint holds with equality.

Procedure MATCH

Initialize the matching at the current scale to $\emptyset$. (We throw away the 1-optimal matching computed at the previous scale.) Then repeat the following steps until Step 1 halts with a perfect matching.

1. Find a maximal set $\mathcal{A}$ of vertex-disjoint augmenting paths of eligible edges. For each path $P \in \mathcal{A}$, augment the matching along P, and for each vertex $v \in B \cap P$, decrease ω_v by 1. (This makes the new matching 1-feasible.) If the new matching is perfect, halt.

2. Do a Hungarian search to adjust the duals (maintaining 1-feasibility) and find an augmenting path of eligible edges.

Gabow and Tarjan show the Steps 1 and 2 of procedure MATCH are iterated $O(\sqrt{n})$ times at each scale. They also show how Steps 1 and 2 can be implemented in $O(m)$ time, where m is the number of edges in $\mathcal{G}$. This yields an $O(\sqrt{n}m)$ algorithm at each scale, and since there are $O(\log(nN))$ scales, an $O(\sqrt{n}m \log(nN))$ time algorithm for computing a min-cost perfect matching in $\mathcal{G}$. We now show how the clique covers can be used to implement steps 1 and 2 of procedure match in $O(n \log^4 n/\varepsilon^2)$ time.

Implementing Step 1 Step 1 of procedure MATCH begins with the current matching M and set of dual variables ω_v for each $v \in R \cup B$. The matching or dual variables are *not* changes in this step. Let us call a vertex *free* if it is not incident on any edge of M. Step 1 finds a maximal set of

vertex disjoint augmenting paths by doing a depth-first search. To do this, it *marks* every vertex reached in the search. It initializes a path P to a free unmarked vertex of R, and marks the vertex. To grow P from the last vertex x of P (which will always be in R), it asks a query: Is there an eligible edge from x to an unmarked vertex in B? If so, return such a vertex y. If no vertex is returned by the query, the last two edges of P (one matched and one unmatched) are deleted from P; if P has no edges, another augmenting path is intialized. Suppose a vertex y is returned by the query. If y is free, another augmenting path has been found; in this case y is marked, the path is added to $\mathcal{A}$, and the next path is intialized. The remaining possibility is that y is matched to a vertex z. In this case y and z are marked; edges (x, y) and (y, z) are added to P; and the search is continued from z.

It is clear that apart from the time needed to answer the queries, the procedure can be implemented in $O(n)$ time. We will show how the clique covers can be used to answer the queries. Consider the clique cover $\{(R_1, B_1), \ldots, (R_l, B_l)\}$ of the edges in class C_i; all these edges have the same cost, say n_i. Consider a given (R_j, B_j); we order the unmarked vertices in B_j according to increasing order of their dual variables. Suppose that for an $r \in R_j$, we want to know if there is an eligible edge (r, b) to an unmarked vertex in B_j. In other words, we want to know if there is an unmarked vertex $b \in B_j$ such that $\omega_r + \omega_b = n_i + 1$. The simple but important observation is that we can do this by binary searching the unmarked vertices in B_j using $n_i + 1 - \omega_r$ as the key.

Our data structure for answering the queries is built around this observation. We describe the data structure just for the edges in a single class C_i with clique cover $\{(R_1, B_1), \ldots, (R_l, B_l)\}$. We store the unmarked vertices in each B_j in sorted order according to their dual variables. When a vertex in B is marked, it is deleted from the structures in all the B_j's it is present in. Clearly, the total time needed to build and maintain this structure is proportional to $\sum_j |B_j| \log n$.

Each vertex $r \in R$ has a list of all the R_j's that contain r and that potentially have an eligible edge from r to an unmarked vertex in B_j. Given a query with r, we pick one of the R_j's from the list, and find in $O(\log n)$ time if there is an eligible edge from r to an unmarked vertex of B_j. If there is such a vertex, we return it. We charge this query to the returned vertex, which is going to be marked. If there is no such vertex, we discard the R_j from the list for r and continue with a new R_j from the list. It is clear that the time for answering all the queries is proportional to $\sum_j R_j \log n$, plus an $n \log n$ that is charged to the marked vertices. Summing over the $O(\log n/\varepsilon)$ classes, we conclude that Step 1 of procedure MATCH can be impelented in $O(n \log^4 n/\varepsilon^2)$ time.

Implementing Step 2 Step 2 of procedure MATCH is the Hungarian search. It is well known [13] that the key component of the Hungarian search is the following problem of maintaining bichromatic closest pairs. We want to maintain a $R' \subseteq R$ and a $B' \subseteq B$. Initially, $R' = \emptyset$ and $B' = B$, and vertex each v in B' is assigned a weight $\sigma(v)$. The operations allowed on R' and B' are the following: A vertex u may be inserted into R' with a weight $\sigma(u)$; a vertex v may be deleted from B'. The problem is to maintain the bichromatic closest pair, which is the pair $(u, v) \in R' \times B'$ that minimizes $c(u, v) - \sigma(u) - \sigma(v)$.

We will show how to maintain the bichromatic closest pair over edges in a single class C_i, whose clique cover is $\{(R_1, B_1), \ldots, (R_l, B_l)\}$. Since all the edges in class C_i have the same cost, this boils down to simply maintaining the pair that minimizes $-\sigma(u) - \sigma(v)$ over all $(u, v) \in C_i \cap R' \times B'$. For an (R_j, B_j), let $R'_j = R_j \cap R'$, and $B'_j = B_j \cap B'$. We simply maintain the smallest $-\sigma(u)$ over all $u \in R'_j$, and the smallest $-\sigma(v)$ over all $v \in B'_j$. This gives us the pair that minimizes $-\sigma(u) - \sigma(v)$ over all $(u, v) \in R'_j \times B'_j$; we maintain the minimum over all the (R_j, B_j) in the clique cover by using a priority queue. It can now be shown that inserts in R', deletes in B', and

maintaining the bichromatic closest pair for class C_i takes $O(n \log^3 n/\varepsilon)$ time, and over all classes takes $O(n \log^4 n/\varepsilon^2)$ time. We conclude that step 2 of the procedure MATCH can be implemented in $O(n \log^4 n/\varepsilon^2)$ time.

Since there are $O(\sqrt{n})$ iterations of steps 1 and 2 of procedure MATCH the running time for procedure MATCH is $O(n^{3/2} \log^4 n/\varepsilon^2)$. This also bounds the running time for one scale of the matching algorithm. There are a total of $log(nN)$ scales. Since $N = O(n^3/\varepsilon)$, and we assume that $1/\varepsilon < n$, we get an overall running time of $O(n^{3/2} \log^5 n/\varepsilon^2)$ for the entire scaling algorithm. Putting everything together, we conclude:

Theorem 3.1 *Given a set R of n red points and a set B of n blue points in the plane, and a real number $\varepsilon > 0$, we can compute a perfect red-blue matching whose cost is at most $(1 + \varepsilon)$ times the cost of the optimal perfect red-blue matching using an algorithm whose running time is $O(n^{3/2} \log^5 n/\varepsilon^2)$.*

References

[1] P. K. Agarwal, A. Efrat, and M. Sharir. Vertical decomposition of shallow levels in 3-dimensional arrangements and its applications. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 39–50, 1995.

[2] S. Arora. Nearly linear time approximation schemes for Euclidean TSP and other geometric problems. In *Proc. 38th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 554–563, 1997.

[3] D. Avis. A survey of heuristics for the weighted matching problem. *Networks*, 13:475–493, 1983.

[4] J. Edmonds. Maximum matching and a polyhedron with (0,1) vertices. *J. Res. National Bureau of Standards*, ??:125–130, 1965.

[5] A. Efrat and A. Itai. Improvements on bottleneck matching and related problems using geometry. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 301–310, 1996.

[6] T. Feder and R. Motwani. Clique partitions, graph compression, and speeding up algoithms. In *Proc. 27th Annu. ACM Sympos. Theory Comput.*, pages 123–133, 1991.

[7] H. Gabow and R. Tarjan. Faster scaling algorithms for network problems. *SIAM J. Comput.*, 18:1013–1036, 1989.

[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 on Foundations of Computer Science*, pages 255–261, 1982.

[9] H. Kuhn. The Hungarian method for the assignment problem. *Naval Res. Logist. Q.*, 2:83–97, 1955.

[10] E. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart & Winston, New York, 1976.

[11] S. B. Rao and W. D. Smith. Improved approximation schemes for traveling salesman tours. In *Proc. 30th Annu. ACM Sympos. Theory Comput.*, page to appear, 1998.

[12] P. M. Vaidya. Approximate minimum weight matching on points in k-dimensional space. *Algorithmica*, 4:569–583, 1989.

[13] P. M. Vaidya. Geometry helps in matching. *SIAM J. Comput.*, 18:1201–1225, 1989.

[14] Kasturi R. Varadarajan. A divide-and-conquer algorithm for min-cost perfect matching in the plane. *To appear in FOCS 98.*

[15] Kasturi R. Varadarajan and Pankaj K. Agarwal. Algorithms for polygonal chain simplification. *manuscript.*

Appendix

We give a proof of Lemma 2.5

Lemma 2.5. The number of phases in the conquer step for U is $O(\log n/\varepsilon)$.

Proof: Let $\mathcal{E}$ denote the number of exposed, unconstrained blossoms at the beginning of the conquer step. Since each phase decreases the total number of exposed, unconstrained blossoms by one or two, the number of phases is at most $|\mathcal{E}|$. Hence it suffices to show $|\mathcal{E}| = O(\log n/\varepsilon)$. Let $Q \in \mathcal{E}$, and assume, w. l. o. g, that $Q \in U_1$. After the recursive call to U_1, Q must be constrained at a portal on the boundary of K_1; since it is unconstrained with respect to U, this cannot be a portal on the boundary of K. The number of such portals is $O(\log n/\varepsilon)$. By Lemma 2.4, this bounds $|\mathcal{E}|$. □