Approximation Algorithms for Bipartite and Non-Bipartite Matching in the Plane *

Kasturi R. Varadarajan† Pankaj K. Agarwal‡

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 to 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. We present a Monte-Carlo algorithm that returns, with probability at least $1/2$, a solution within $(1+\varepsilon)$ of the optimal; the running time of our algorithm is $O((n/\varepsilon^3)\log^6 n)$.

In the bipartite version of this problem, we are given a set R of n *red* points, a set B of n *blue* points in the plane, and a real number $\varepsilon > 0$. We want to match each red point with a blue point so that the sum of the distances between paired points is within $(1+\varepsilon)$ times that of an optimal matching. We present an algorithm for this problem that runs in $O((n/\varepsilon)^{3/2}\log^5 n)$ time.

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 so 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; 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.

There has been considerable amount of work on hueristics for Euclidean matching; see the survey by Avis [3], and the references therein. Much of this work considers the case in which the points are in a unit square, and aims at producing a matching whose absolute cost is small. In contrast, Vaidya [12] gave an algorithm for approximate min-cost matching in the plane 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 at least $1/2$) a matching whose cost is within a constant factor of the optimal. We are not aware of any previous work on approximation algorithms for min-cost bipartite matching in the plane.

Our results. This paper contains two main results. First, we present a Monte Carlo algorithm for computing an approximate min-cost non-bipartite matching that runs in $O((n/\varepsilon^3)\log^6 n)$ time; the algorithm returns a matching that whose cost is within $(1+\varepsilon)$ times that of of an optimal, with probability at least $1/2$. Of course, the probability of success can be increased by

* Work on this paper was done when the first author was at Duke University. This work has been supported by National Science Foundation research grants CCR–9732287 and EIA–9870724, by Army Research Office MURI grant DAAH04–96–1–0013, by a Sloan fellowship, by a National Science Foundation NYI award, and by a grant from the U.S.-Israeli Binational Science Foundation. National Science Foundation

†DIMACS, Rutgers University, Piscateway, NJ 08854 Email: krv@dimacs.rutgers.edu

‡Center for Geometric Computing, Department of Computer Science, Duke University, Durham, NC 27708. Email: pankaj@cs.duke.edu

iterating the algorithm and taking the smallest matching returned. Arora's algorithm [2] achieves the same result, but with a running of $O(n \log^{O(1/\varepsilon)} n)$. Unlike Arora's algorithm, the running time of our algorithm is polynomial in $1/\varepsilon$. We achieve this improvement by combining the divide-and-conquer approach developed in [14] for computing an optimal Euclidean matching in the plane along with the the partitioning scheme of Arora.

The techniques used by Vaidya [12] and Arora [2] for approximate min-cost non-bipartite matching do not extend to the bipartite case. No subquadratic algorithm is known even for a constant-factor approximate bipartite min-cost matching. We present a determinsitic algorithm for computing an $(1+\varepsilon)$-approximate Euclidean bipartite matching that runs in $O((n/\varepsilon)^{3/2} \log^5 n)$ time. The crux of our algorithm is an efficient implementation in geometry of the scaling algorihm of Gabow and Tarjan [7]. For this, we partition the red-blue edges into $O(\frac{\log n}{\varepsilon})$ 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. We offer our conclusions in Section 4. 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 $\mathcal{G}$.

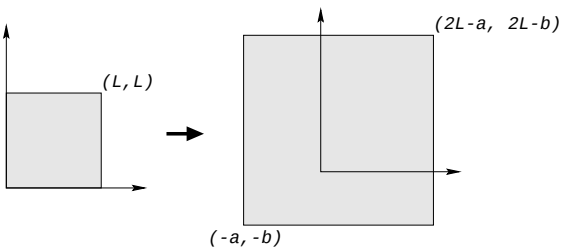

Figure 1: Original square and the expanded square.

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 $L \times L$ square, where $L \leq n^3$. We also assume, without loss of generality, that $L = 2^k$, for some integer $k \geq 0$, and that the square is aligned with the integer grid, i.e., the bottom-left corner is at the origin and the top-right corner is (L, L). We expand the square to a $2L \times 2L$ square as follows. We choose random integers $a, b \in [0, L]$. We move the bottom-left corner to $(-a, -b)$ and the top-right corner to $(2L-a, 2L-b)$; see Figure 1.

We construct a quad-tree on the resulting 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, as in Arora's algorithm. We then recursively construct the quadtrees for $K_1, \ldots, K_4$.

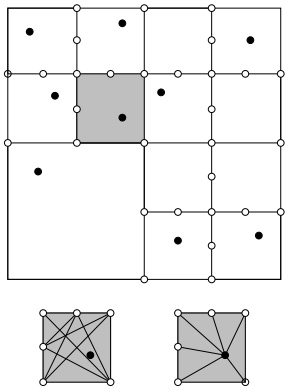

Figure 2: A quadtree for the points indicated in bold. The empty circles are the portals. The two kinds of edges in $\mathcal{G}$ corresponding to the shaded leaf of the quadtree are shown.

We now construct a graph $\mathcal{G}$ whose vertices are the points in V, called *sites*, and the portals, also called *Steiner points*, that we have placed. For each *leaf* square K of the quad-tree, we add an edge between every pair of portals on the boundary of K. In addition, if the leaf square K contains a point $v \in V$, we add an edge between v and each portal on the boundary of K. See Figure 2 for an illustration. 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 $O(\frac{n}{\varepsilon} \log^2 n)$, and the total numer of edges is $O(\frac{n}{\varepsilon^2} \log^3 n)$. We define the *length* of an edge (p, q) in this graph to be the Euclidean distance between p and q. The *distance* between any two vertices p and q of $\mathcal{G}$, denoted by $\delta(p, q)$, is the length of the shortest path between p and q in $\mathcal{G}$. 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 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 at least* $1/2$*, 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 was developed by Varadarajan [14] for exact matching in the plane.

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

We 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. We define the distance between a vertex w and a point z on a link (u, v), denoted by $\delta(w, z)$, as the length of the shortest path along the links, i.e., $\delta(w, z) = \min\{\delta(w, u) + d(u, z), \delta(w, v) + d(w, z)\}$. We define the *disk* of radius r centered at a site v to be the set of all vertices and all portions on the edges of $\mathcal{G}$ whose distance (in $\mathcal{G}$) from v is at most r. In other words, a disk is a one-dimensional network consisting of vertices and portions of edges of $\mathcal{G}$. By definition of δ, if a portion of an edge is in the disk, then at least one of its endpoints is also in the disk.

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], [10] for a discussion of linear programming duality. His algorithm associates a "dual" 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 $(u, v) \in V \times V$.

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 and alternating paths

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. (See Figure 3 for an illustration of a blossom.) 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.

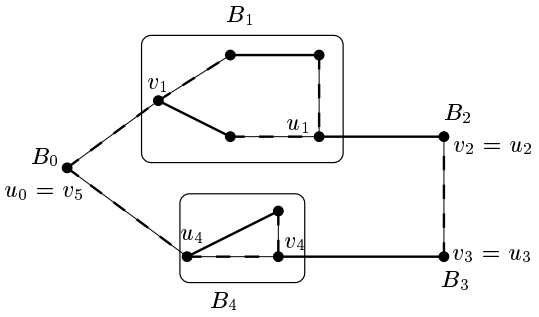

Figure 3: A blossom with sub-blossoms $B_0, \ldots, B_4$. The solid edges represent pairs in the matching and the dashed edges represent the other admissible pairs making up the blossom.

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 iff $\lambda(u) + \lambda(v) \leq \delta(u, v)$. The pair (u, v) is admissible iff $\lambda(u) + \lambda(v) = \delta(u, v)$.*

Using the triangle inequality for $\delta(.,.)$, we can show that throughout our algorithm, $\lambda(v) \geq 0$ for any $v \in V$. For a site $v \in V$, we use disk(v) to denote the disk of radius $\lambda(v)$ centered at v, as defined above. Since $\lambda(v) \geq 0$, disk(v) is well defined. Lemma 2.3 implies 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), i.e., no point on a link lies in the relative interior of both disk(u) and disk(v); admissibility of (u, v) means disk(u) and disk(v) do not overlap but touch.

In the rest of this section, we will sometimes refer to an outermost blossom as simply a blossom, and use the term 'sub-blossom' when specifically referring to a blossom that is not outermost.

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 sites 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. Let m (resp. η) denote the number of vertices (resp. edges) in $\mathcal{G}(K)$. The goal in the subproblem 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, so that the conditions EDGE-FEASIBILITY, POSITIVE-DUAL, MATCHING-ADMISSIBILITY, and MAXIMALITY hold for U, and 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: For every exposed blossom Q of U, there is a $q \in Q$ and a portal $p \in P$ such that $\lambda(q) = \delta(q, p)$.

We say that a blossom Q of U is *constrained* at a portal $p \in P$ if Q is an exposed blossom, and there is a $q \in Q$ such that $\lambda(q) = \delta(q, p)$. We say that Q is *unconstrained* otherwise. We emphasize that only an exposed blossom can be constrained; a blossom that is not exposed is by definition unconstrained. The EXPOSED-CONSTRAINT condition is that every exposed blossom of U is constrained. We remark that the RADIUS-CONSTRAINT allows us to restrict our attention to $\mathcal{G}(K)$ for solving the subproblem 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 subproblems for $K_1, \ldots, K_4$, the four children of Q. Let $U_i \subseteq U$ be the set of sites 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 subproblem for U. The 'conquer' stage consists of a series of phases; in each phase the number of unconstrained exposed 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. The algorithm always maintains the five conditions EDGE-FEASIBILITY, POSITIVE-DUAL, MATCHING-ADMISSIBILITY, MAXIMALITY, and RADIUS-CONSTRAINT. In each phase, the number of unconstrained exposed 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.

For brevity, we will call a constrained blossom a c-blossom. 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, f-vertex, or c-vertex according to whether it belongs, respectively, to an s-blossom, t-blossom, f-blossom, or c-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 $O(m)$ *subphases*. At the end of each subphase, the following invariants hold. An unconstrained exposed blossom is always an s-blossom. For every s- or t- blossom B, there is an alternating path $\sigma(B', B)$ between an unconstrained exposed blossom B' and B. If B is an s-blossom (resp. t-blossom), $\sigma(B', B)$ has even (resp. odd) length, that is, there are an even (resp. odd) number of edges in the alternating path. The s- and t-blossoms, together with the corresponding alternating paths, induce a forest of rooted trees, a tree being rooted at each unconstrained exposed 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 so that bases of B and C are matched in M, i.e., $(\text{base}(C), \text{base}(C)) \in M$. Therefore, M induces a perfect matching on the bases of all the f-blossoms.

At the start of a phase, we label each unconstrained exposed blossom as an s-blossom; every other unconstrained outer blossom is an f-blossom. A subphase 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

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

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

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 unconstraned exposed 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 unconstrained exposed 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 augmenting 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 unconstrained exposed blossoms reduces 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 unconstrained exposed 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. This event reduces the number of unconstrained exposed blossoms by one without changing the cardinality of M. Note that in the next phase, A will be 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.*

The proof uses 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 $O((\log n)/\varepsilon)$.*

Proof: Let $\mathcal{E}$ denote the number of unconstrained exposed blossoms at the beginning of the conquer step. Since each phase decreases the total number of unconstrained exposed 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, without loss of generality, 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 portals on $\partial U_1 \setminus \partial U$ is $O((\log n)/\varepsilon)$. By Lemma 2.4, this bounds $|\mathcal{E}|$. □

For a fast implementation of one phase of the conquer algorithm for U, we need a mechanism to quickly compute when δ_i becomes zero. The following theorem results from a careful implementation of a phase such as the one described by Galil et al. [8] or Vaidya [13].

Theorem 2.6 *Suppose the total time spent in detecting when δ_i becomes zero, over an entire phase for the subset U of sites, is $O(\tau)$. Then, one phase can be implemented in $O(|U| \log |U| + \tau)$ time.*

We describe below the main ideas of an implementation that detects when δ_i becomes zero at a total cost of $O(\eta \log m)$ per phase, where m (resp. η) denotes the number of vertices (resp. edges) in $\mathcal{G}(K)$. Since there are $O((\log n)/\varepsilon)$ phases in the conquer step, this gives a running time of $O(\frac{\log n}{\varepsilon} \cdot \eta \cdot \log m)$ for the conquer step. Putting everything together, we conclude with the main result of this section:

Theorem 2.7 *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 $O((n/\varepsilon^3) \log^6 n)$ time.*

2.5 Implementing a phase

We show below that detecting when δ_i becomes zero essentially reduces to detecting events when an inequality of one of the following forms becomes tight:

1. $\omega_Q \geq 0$, for a t-blossom Q.
2. $\lambda(u) \leq c_1$, for an s-vertex u and a real number c_1.
3. $\lambda(u) \geq c_1$, for an f- or t-vertex u and a real number c_1.
4. $\lambda(u) + \lambda(v) \leq c_1$, for a real number c_1, where either (i) u and v are s-vertices in different outermost blossoms, (ii) u is an s-vertex and v is a t-vertex, and (iii) u is an s-vertex and v is a c-vertex.

We will refer to these as events of the first, second, third, and fourth type, respectively. The total number of such events that we need to detect per phase is only $O(\eta)$. We will also need a data-structure which will allow us to compute the value of $\lambda(u)$, for any site u. This data-structure is queried $O(\eta)$ times. With straightforward modifications, the priority queues of Galil et al. [8] will let us implement the above data structures using a total of $O(\eta \log m)$ time per phase.

To detect when δ_1 becomes zero, we need to detect $O(m)$ events of the first type. For δ_5, we precompute $\delta(u,p)$, the distance in $\mathcal{G}(K)$ between each vertex $u \in U$ and portal $p \in P$, and then the closest portal to each vertex in U. To detect when δ_5 becomes zero, we now need to detect events of the second type. Again, there are only $O(m)$ such events. The cost of precomputing the distances $\delta(u,p)$ is absorbed in the running time of the conquer phase.

For detecting when δ_i becomes zero, for $2 \leq i \leq 4$, we need to detect when disk(u) and disk(v) touch, for sites u and v of U in *different* outer blossoms of U. For this, we first need to establish certain simple but useful properties of disks.

Lemma 2.8 *If u and v are sites in the same outer blossom, $\lambda(u)$ and $\lambda(v)$ change at the same rate.*

The following observation depends on the fact that the algorithm increases the dual variables corresponding to the s-blossoms, decreases the dual variables corresponding to the t-blossoms, and does not change the dual variables corresponding to the f-blossoms. It also expresses a property of the algorithm's labelling scheme.

Lemma 2.9 *During a phase, a site $u \in U$ may change its status from an f-vertex to a t-vertex (and vice versa) a number of times. In this part of the phase, $\lambda(u)$ can only decrease. However, once u becomes an s-vertex, it remains an s-vertex until the end of the phase. In this part of the phase, $\lambda(u)$ can only increase. If u belongs to a c-blossom, $\lambda(u)$ does not change at all during the phase.*

In view of the above lemma we refer to an s-vertex (resp. f/t-vertex) as a *growing* (resp. *shrinking*) vertex, and its disk as a growing (resp. shrinking) disk. The following lemma can be derived from Lemma 2.3 and the fact that $\lambda(u) \geq 0$ for any vertex $u \in U$.

Lemma 2.10 *For any two distinct sites $u, v \in U$, $\lambda(u) \leq \lambda(v) + \delta(u, v)$.*

For any vertex p of $\mathcal{G}(K)$, we define the *nearest* site ν_p of p to be $u \in U$ that minimizes $\delta(u, p) - \lambda(u)$; we refer to disk($\nu_p$) as the *nearest* disk to p. We say that ν_p (or disk(ν_p)) *owns* p if disk(ν_p) contains p, i.e., $\delta(\nu_p, p) \leq \lambda(\nu_p)$.

For simplifying the discussion, assume that ν_p is always unique for every vertex p. By Lemma 2.10, we can assume that, for any site $u \in U$, $\nu_u = u$ always holds. Furthermore, since $\lambda(u) \geq 0$, we can assume that u always owns itself.

For any portal p, if a site $u \in U$ owns p at a particular time, then u continues to own p as long as $\lambda(u) \geq \delta(u, p)$. Indeed, this follows from the fact the disks of sites in different outer blossoms do not overlap (Lemma 2.3) and the fact that disks of sites in the same blossom grow at the same rate (Lemma 2.8). By Lemma 2.9, the owner of a steiner-point p can change only a constant number of times during a phase. To maintain the owner of a steiner point p during a phase, we mainly need to keep track of two kinds of information:

1. If a shrinking vertex u currently owns p, when does disk(u) stop containing p? This is an event of the third type, and there are $O(m)$ such events.

2. If p is currently without an owner, when does the disk of an s-vertex first hit p? For maintaining this information, we need to 'propogate' the disks of sites along the edges of the graph. This can be achieved using $O(\eta)$ events of the second type.

Finally, we are ready to describe how we can detect collisions between two disks of vertices in different outer blossoms of U. Suppose the point of contact of such a collision lies in the interior of an edge (p, q). Using Lemma 2.3 and Lemma 2.8, we can conclude that the disks that touch must be the owners of p and q. Therefore, to detect such a collision, we need to keep track of the event 'When do the disks that own p and q collide along the edge (p, q)'? This is an event of the fourth type. It is easy to see from the above discussion that there are only a constant number of such events for each edge in a phase.

3 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 whose cost is at most $(1 + \varepsilon)$ times that of an optimal matching. 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 use the clique covers to implement the scaling algorithm of Gabow and Tarjan efficiently on an appropriately defined graph.

3.1 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 rough approximation α to OPT such that $\alpha \leq OPT \leq n\alpha$. Using the algorithm by Efrat and Itai [5], we compute in $O(n^{3/2} \log n)$ time 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, under 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 and the cost of the optimal bottleneck matching is at most $n\alpha$, we have $\alpha \leq OPT \leq n \cdot \alpha$.

We draw a grid in the plane of the form $\{(i\alpha/n^2, j\alpha/n^2) \mid i, j \in \mathbb{Z}\}$, move each point in $R \cup B$ to the nearest grid point, and compute a matching of the resulting points. The cost of the optimal matching of perturbed points differs from the original one by at most $2n \cdot OPT/n^2 = OPT/2n$. Since $\varepsilon > 1/n$, it suffices

to compute an $(1+\varepsilon/2)$-approximate matching for the new set of points. If a red point r and a blue point b are moved to the same grid point, we match r with b and discard r and b. We thus assume that no grid point contains both red and blue points. Scaling the grid by factor n^2/α, we can 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 ignore from consideration red-blue pairs that are more than n^3 apart.

3.2 Clique covers of interesting pairs

Instead of the Euclidean metric, it will be convenient to measure distances between points using a polygonal metric, $d_P(\cdot,\cdot)$, defined by a centrally-symmetric regular convex polygon with $O(1/\sqrt{\varepsilon})$ edges.[1] Such a metric approximates the Euclidean metric to within a factor of $(1+\varepsilon)$. Let us call the red-blue pairs whose length, under d_P-metric, is between 1 and n^3 the *interesting pairs*.

For $1 \leq j \leq k = \lceil 3\log_{1+\varepsilon} n \rceil$, define I_j to be the interval $[(1+\varepsilon)^{j-1}, (1+\varepsilon)^j]$. Let $\mathcal{C}_j = \{(r,b) \in R \times B \mid d_P(r,b) \in I_j\}$. $\mathcal{C}_1, \ldots, \mathcal{C}_k$ partition the set of interestng red-blue pairs. We define a *bipartite clique cover*, or simply a clique cover for brevity, for the set of pairs in class C_i to be a family $\mathcal{F}_i = \{(R_1, B_1), \ldots, (R_l, B_l)\}$ with the following properties:

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, and
3. for every pair (r,b) in class C_i, there is an (R_j, B_j) such that $(r,b) \in R_j \times B_j$.

The *size* of the clique cover is $\sum_j (|R_j| + |B_j|)$. 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, in $O((n/\sqrt{\varepsilon})\log^2 n)$ time, a clique cover of $\mathcal{C}_i$ of size $O((n/\sqrt{\varepsilon})\log^2 n)$ [15]. Set $\mathcal{F} = \bigcup_i \mathcal{F}_i$. The total time spent in computing $\mathcal{F}$ is $O((n/\varepsilon^{3/2})\log^3 n)$.

3.3 The scaling algorithm

We approximate the lengths of all the pairs belonging to a class $\mathcal{C}_i$ by a single number $n_i = (1+\varepsilon)^{i-1}(1+\varepsilon/2)$, the middle-point of the subinterval I_i. By scaling all the numbers, we assume that all the n_i's are integers. Define $\mathcal{G} = (R \cup B, \bigcup_i \mathcal{C}_i)$; the cost of the edge (r,b) is set to n_i if $(r,b) \in \mathcal{C}_i$.

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 so that for any pair $(u,v) \in R \times B$,

$$\begin{aligned} \omega_u + \omega_v &\leq c(u,v) + 1, && \forall (u,v) \in R \times B, \\ \omega_u + \omega_v &= c(u,v), && \forall (u,v) \in M. \end{aligned}$$

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 begins by computing a new cost $\overline{c}(e)$ for each edge e, equal to $n+1$ times the given cost. Consider each $\overline{c}(e)$ to be a 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,

   $$c(e) \leftarrow 2c(e) + \text{ bit } b_s \text{ of } \overline{c}(e).$$

   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 constraint holds with equality.

procedure MATCH

I. Initialize $M = \emptyset$. (We throw away the 1-optimal matching computed at the previous scale.)

[1] For a centrally symmetric polygon P, the convex polygonal distance, d_P, between two points $p, q \in \mathbb{R}^2$ is defined as $d_P(p,q) = \inf\{\lambda \mid q \in \lambda P + p\}$.

II. Repeat the following steps until Step 1 halts with a perfect matching.

Step 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.

Step 2. Do a Hungarian search to adjust the duals (maintaining 1-feasibility) until there is an augmenting path of eligible edges.

Gabow and Tarjan show that Steps 1 and 2 can be implemented in $O(m)$ time, where m is the number of edges in $\mathcal{G}$, and that they are repeated $O(\sqrt{n})$ times at each scale. Since there are $O(\log(nN))$ scales, the total running time is $O(\sqrt{n}m\log(nN))$. We show below that Steps 1 and 2 of procedure match can be performed in $O((n/\varepsilon^{3/2})\log^4 n)$ time using clique covers. We therefore conclude the following:

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 in time $O((n/\varepsilon)^{3/2}\log^5 n)$.*

3.4 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* changed in the middle of this step. Let us call a vertex *free* if it is not incident on any edge of M. Step 1 finds a maximal set $\mathcal{A}$ of vertex disjoint augmenting paths, one by one, using a depth-first search. In the process, it *marks* every vertex reached during the search. The procedure terminates after no unmarked free vertex is left in R. The procedure begins by growing a path Π from some free unmarked vertex of R, and marks the vertex. Suppose it is growing a path $\Pi = \langle r_1, b_1, r_2, \cdots, b_{i-1}, r_i\rangle$. To grow Π from r_i, it asks the following query: *Is there an eligible edge from r_i to an unmarked vertex in B?* There are two cases:

(i) *There is such a vertex b_i.* If b_i is free, $\Pi = \langle r_1 \cdots b_i\rangle$ is an augmenting path. We add Π to $\mathcal{A}$, mark b_i, and start growing a new path from a different unmarked free vertex of R. If b_i is matched to another vertex r_{i+1}, set $\Pi = \langle r_1 \cdots r_i, b_i, r_{i+1}\rangle$, mark b_i and r_{i+1}, and grow Π at r_{i+1}.

(ii) *There is no such a vertex.* If $i = 1$, we start growing a new path from a different unmarked free vertex of R. If $i > 1$, r_i and b_{i-1} are deleted from Π and we grow Π at r_{i-1}.

If we had all the m edges of $\mathcal{G}$, the above queries could be answer in a total of $O(m)$ time (summed over all queries). Since we only have a clique cover of the edges of $\mathcal{G}$, we preprocess them into a data structure so that the queries can be answered efficiently. For each $r \in R$, we maintain a subset J_r of indices of pairs in $\mathcal{F}$. Initially, we set $J_r = \{j \mid r \in R_j\}$. For each B_j, we maintain the unmarked vertices of B_j in a red-black tree with the values of their dual varaible as the key. Since all interesting pairs in a class $\mathcal{C}_i$ have the same cost n'_i, determining whether there is an eligible edge from $r \in R_j$ to an unmarked vertex in B_j is equivalent to determining whether there is a vertex $b \in B_j$ in the red-black tree with $\omega_b = n'_i + 1 - \omega_r$. Using our data structure, we can search for such a vertex b in $O(\log n)$ time. Hence, to answer a query for a vertex $r \in R$, we choose the first index j in the list J_r, and search in B_j with the appropriate value. If a vertex b is found, we mark b and delete it from all the red-black trees. Otherwise, we delete j from J_r and repeat the same procedure with the next index in the list. If no index in J_r is left and we have not found a desired vertex, we conclude that there is no eligible edge from r to an unmarked vertex in B. Since at each step we either delete an index from J_r or delete a vertex b from all trees, the total time spent in answer all queries is $\sum_j O((|R_j| + |B_j|)\log n) = O((n/\varepsilon^{3/2})\log^4 n)$.

Lemma 3.2 *A single iteration of Step 1 takes $O((n/\varepsilon^{3/2})\log^4 n)$.*

3.5 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 σ_v. The operations allowed on R' and B' are the following: A vertex u may be inserted into R' with a weight σ_v; 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 reduces to simply maintaining the pair that maximizes $\sigma_u + \sigma_v$ 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 largest σ_u over all $u \in R'_j$, and the largest σ_v over all $v \in B'_j$. This gives us the pair that maximizes $\sigma_u + \sigma_v$ over all $(u, v) \in R'_j \times B'_j$; we maintain the maximum 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 $O(n \log^3 n/\sqrt{\varepsilon})$ time, and over all classes takes $O((n/\varepsilon^{3/2}) \log^4 n)$ time. We conclude that step 2 of the procedure MATCH can be implemented in $O((n/\varepsilon^{3/2}) \log^4 n)$ time.

Lemma 3.3 *A single iteration of Step 2 takes* $O((n/\varepsilon^{3/2}) \log^4 n)$.

4 Conclusions

We have presented a Monte-Carlo algorithm for approximate planar min-cost matching that runs in $O((n/\varepsilon^3) \log^6 n)$ time and returns a matching whose cost is within $(1+\varepsilon)$ of the optimal. For the bipartite version of the problem, we presented a deterministic algorithm that runs in $O((n/\varepsilon)^{3/2} \log^5 n)$ time.

We conclude by mentioning two interesting open problems.

- Is there an algorithm for approximate planar bipartite matching whose running time dependence on n is near-linear?
- Is there a sub-quadratic time algorithm for exact planar bipartite matching?

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.*, 1995, 39–50.

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

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

[4] J. Edmonds. Maximum matching and a polyhedron with (0,1) vertices. *J. Res. National Bureau of Standards*, 69B (1965), 125–130.

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

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

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

[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*, 1982, 255–261.

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

[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.*, 1998, 540–550

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

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

[14] K. R. Varadarajan. A divide-and-conquer algorithm for min-cost perfect matching in the plane. In *Proc. 38th Annual IEEE Symposium on Foundations of Computer Science*, 1998.

[15] K. R. Varadarajan and P. K. Agarwal. Algorithms for polygonal chain simplification. Tech. Rept. CS-98-15, Dept. Computer Science, Duke University, 1998.