Computing Fair and Bottleneck Matchings in Geometric Graphs

Alon Efrat* Matthew J. Katz†

May 15, 1996

Abstract

Let A and B be two sets of n points in the plane, and let M be a (one-to-one) matching between A and B. Let $\min(M)$, $\max(M)$, and $\Sigma(M)$ denote the length of the shortest edge, the length of the longest edge, and the sum of the lengths of the edges of M respectively. The *uniform matching* problem (also called the *balanced assignment* problem, or the *fair matching* problem) is to find M_U^*, a matching that minimizes $\max(M) - \min(M)$. A *minimum deviation matching* M_D^* is a matching that minimizes $(1/n)\Sigma(M) - \min(M)$. We present algorithms for computing M_U^* and M_D^* in roughly $O(n^{10/3})$ time. These algorithms are more efficient than the previous $O(n^4)$-time algorithms of Martello and Toth [19] and Gupta and Punnen [11], who studied these problems for general bipartite graphs.

We also consider the (non-bipartite version of the) Euclidean bottleneck matching problem in higher dimensions. We extend the planar results of Chang et al. [4] and Su and Chang [22], and show that given a set A of $2n$ points in d-space, it is possible to compute a bottleneck matching of A in roughly $O(n^{3/2})$ time, for $d \leq 6$, and in subquadratic time, for any fixed dimension d.

*School of Mathematical Sciences, Tel-Aviv University, Tel-Aviv 69982, Israel. *alone@cs.tau.ac.il*

†Department of Computer Science, Utrecht University, P.O.Box 80.089, 3508 TB Utrecht, The Netherlands. *matya@cs.ruu.nl*

1 Introduction

Let $G = (A \cup B, E)$ be a bipartite graph. A *Matching* M in G is a subset of E such that each $v \in A \cup B$ belongs to at most one edge of M. M is *maximum* if $|M| \geq |M'|$, for any other matching M' in G. M is *perfect* if each $v \in A \cup B$ belongs to exactly one edge of M. (Obviously, if a perfect matching exists then $|A| = |B|$.) The problem of computing a perfect matching in G has been studied extensively. The best known solution is due to Hopcroft and Karp [15]; it computes a perfect matching in G in time $O(|E|\sqrt{|V|})$. (See also [20].)

When weights are associated with the edges of G, it is often desirable to compute a perfect matching which is optimal with respect to some criterion. A *minimum weight matching* minimizes the sum of the weights of the edges of the matching, a *bottleneck matching* minimizes the maximum weight of an edge in the matching, a *most uniform matching* minimizes the difference between the maximum weight and the minimum weight of edges of the matching, and a *minimum deviation matching* minimizes the difference between the average weight and the minimum weight of the matching (alternatively, minimizes the difference between the maximum weight and the average weight). Much work has been done on the problems of finding efficient algorithms for computing these matchings; see e.g. [9, 11, 17, 18, 19].

In this paper we consider the last two of these four problems from a geometric point of view. That is, we assume that the sets of vertices A and B are sets of points in the plane, and that G is the complete bipartite graph over A,B. The weight associated with an edge (a, b) is the Euclidean distance between a and b. In many of the cases where an optimal matching of one of the above types is required, the underlying graph is indeed a Euclidean bipartite graph (below we mention a few concrete examples). For the first two of these problems, there exist algorithms for the geometric versions that are more efficient than the corresponding more general algorithms. The best algorithm for computing a minimum-weight Euclidean matching was given by Agarwal et al. [3]; it runs in time $O(n^{2+\varepsilon})$ as opposed to the $O(n^3)$-time algorithm that is obtained from the standard *Hungarian method* [17, 18]. (The algorithm of Agarwal et al. is based on a previous $O(n^{2.5}\log n)$-time algorithm of Vaidya [23].) Recently Efrat and Itai [7] have proposed an $O(n^{1.5}\log n)$-time algorithm for computing a Euclidean bottleneck matching, which is, again, faster than the best graph-theoretic algorithm by nearly a factor of n.

Martello and Toth [19] consider the problem of computing a most uniform matching (or a *balanced* assignment, as they call it) in general bipartite graphs, and present an $O(n^4)$-time solution. Here we present an $O(n^{10/3}\log n)$-time solution for this problem in the geometric setting. Our solution is based on a technique for batched range searching, where the ranges are congruent annuli [16], and it also borrows ideas from [7].

The problem of computing a most uniform Euclidean matching M_U^* is strongly related to the field of pattern matching. In Section 3 we show the connection between this problem and the following problem. Find a translation of a point set B that maximizes the similarity (or correspondence) between a large portion B' of B and a large portion A' of another point set A. Also many natural problems arising in the field of operations research boil down to the problem of computing M_U^*.

The problem of computing a minimum deviation matching for an arbitrary graph was considered by Gupta and Punnen [11] who gave an $O(n^4)$-time solution. For this problem we present an $O(n^{10/3+\varepsilon})$-time solution[1] in the geometric setting. This solution is based on the algorithm of [3] for computing the minimum-weight Euclidean matching.

We also study the (non-bipartite version of the) Euclidean bottleneck matching problem in higher dimensions, that is, given a set A of $2n$ points in d-space, compute a bottleneck matching in G, the (complete) Euclidean graph over A. (The weight associated with an edge (a, b) is the Euclidean distance between a and b.) The best known algorithm for computing a bottleneck matching in a general graph is due to Gabow and Tarjan [9]; it runs in time $O((n \log n)^{1/2} m)$. Chang et al. [4] have shown for $d = 2$ that a linear-size subgraph of G, called the *17 relative neighborhood graph* of A and denoted 17RNG (see definition in Section 5), already contains a bottleneck matching of G. Thus, by applying the algorithm of Gabow and Tarjan to the 17RNG of A instead of to G, a bottleneck matching can be computed in $O(n^{3/2} \log^{1/2} n)$ time, provided that the 17RNG of A can be computed within the same time bound. This however is quite simple (assuming general position), see also [22].

We extend the main result of Chang et al. and show that for any fixed dimension d there exists some constant $k = k(d)$ such that the kRNG of A contains a bottleneck matching of G. Since the size of a kRNG remains linear in higher dimensions, and since it (or related linear-size supergraphs of it) can be computed efficiently, we obtain subquadratic algorithms for computing a bottleneck matching in any fixed dimension. In particular we show that up to dimension 6, a bottleneck matching in G can be computed in roughly $O(n^{3/2})$ time.

2 Computing a Most Uniform Matching

We assume some familiarity of the reader with the notion of augmenting paths in the context of computing a maximum (cardinality) matching in a bipartite graph. See [18] for further details.

Let A and B be two sets of n points in the plane. Let $G[r, r']$ denote the bipartite graph whose set of vertices is $A \cup B$, and there is an edge between $a \in A$ and $b \in B$ if $r \leq ||a - b|| \leq r'$, where $||a - b||$ is the Euclidean distance between a and b. Let $d^{(1)} \ldots d^{(n^2)}$ denote the n^2 distances between points of A and points of B, in increasing order. We refer to them as *critical distances*, and we assume, for simplicity of exposition, that they are distinct. We will maintain a maximum matching in $G[r, r']$, where r, r' are critical distances that vary. We start with $G[d^{(i)}, d^{(j)}]$ for $i = j = 1$ and with the empty matching. The top level of the algorithm consists of the following loop. If there is no perfect matching in $G[d^{(i)}, d^{(j)}]$ (i.e., if the current maximum matching is of cardinality less than n) increase j by one, else increase i by one; in either case compute a maximum matching in the new graph, and repeat. Increasing j adds a single edge to the graph, and we must check whether the size of the maximum matching increases by one. Increasing i deletes a single edge from the graph, and, if this edge was in the current maximum matching, we must check whether the size of the maximum matching remains as before or decreases by one. Both these conditions can be checked by trying to

[1]Throughout the paper, ε stands for a positive constant which can be chosen arbitrarily small with an appropriate choice of other constants of the algorithms.

compute an augmenting path for the current matching (in the latter case, after deleting the edge corresponding to the distance $d^{(i)}$ from the current matching). If such a path exists then the answer is positive and we update the current maximum matching, otherwise, the answer is negative. If a perfect matching was found, then we compare the appropriate difference, i.e., either $d^{(j+1)} - d^{(i)}$ or $d^{(j)} - d^{(i+1)}$, with the difference corresponding to the best matching found so far. Clearly, the most uniform matching will be discovered in this way, and the number of times we need to compute an augmenting path is $O(n^2)$.

We next describe how to compute an augmenting path in $G[r, r']$ (if such a path exists) in time $O(n^{4/3} \log n)$. Let $S \subseteq A$ be the set of all vertices in A that are currently unmatched (exposed). If we can find an augmenting path (i.e., a path whose odd edges are edges of $G[r, r']$ that are not in the current matching, and whose even edges are edges of the current matching) from some vertex $a \in S$ to an exposed vertex $b \in B$, then we can augment the current matching. Otherwise, the current matching is also a maximum matching of the new graph.

We need a data structure $\mathcal{D}_{r,r'}(P)$ over a set of points P, supporting the following operations.

- **neighbor**$_{r,r'}(P, q)$: For a query point q, return a point $p \in P$ whose distance from q is between r and r'. If no such p exists, then **neighbor**$_{r,r'}(P, q) = \emptyset$.
- **delete**$_{r,r'}(P, p)$: Delete the point p from P.

Using these operations, we can compute an augmenting path, if such exists, with the following simple procedure, which is a variant of a procedure that appeared in [7]. The underlying data structure is described immediately afterwards.

```
procedure FindAugmentingPath
$L_1 \leftarrow S$ ;
$L_2 \leftarrow \emptyset$ ;
$\mathcal{D} \leftarrow \mathcal{D}_{r,r'}(B)$ ;
Repeat forever
  For each $a \in L_1$ Do
    /* Find all $b$'s that are neighbors of $a$ in $G[r, r']$ */
    While neighbor$_{r,r'}(\mathcal{D}, a) \neq \emptyset$
      $b \leftarrow$ neighbor$_{r,r'}(\mathcal{D}, a)$ ;
      If $b$ is exposed, then stop. An augmenting path was found.
      Add $b$ to $L_2$ ;
      delete$_{r,r'}(\mathcal{D}, b)$ ;   /* prevent re-finding $b$ */
    End
  End
  If $L_2 = \emptyset$, then stop. No augmenting path exists.
  $L_1 \leftarrow$ all vertices connected to some $b \in L_2$ by a matching edge.
  $L_2 \leftarrow \emptyset$.
End
```

Note that this procedure performs $O(n)$ operations of $\mathbf{neighbor}_{r,r'}$ and $\mathbf{delete}_{r,r'}$, and finds an augmenting path (if such exists). The procedure actually computes a forest of trees whose roots are the exposed vertices of A (i.e., the vertices in S). When an exposed vertex b of B is reported, the path leading to b from the root of the tree to which b belongs is an augmenting path. Below we describe the underlying data structure $\mathcal{D}_{r,r'}(B)$ that enables us to perform the operations $\mathbf{neighbor}_{r,r'}$ and $\mathbf{delete}_{r,r'}$ in amortized time $O(n^{1/3}\log n)$. The data structure can be constructed in $O(n^{4/3}\log n)$ time. Thus, an augmenting path can be computed in total time $O(n^{4/3}\log n)$, and, since we repeat this $O(n^2)$ times, we obtain an $O(n^{10/3}\log n)$-time algorithm for computing a most uniform matching.

The data structure

The data structure is based on the following theorem.

Theorem 2.1 *(Katz [16]) Let $\mathcal{M}$ be a set of m congruent annuli and A a set of n points in the plane. One can compute the set of pairs of the form (c, a), where $c \in \mathcal{M}$, $a \in A$, and a lies inside c, as a collection $\{\mathcal{M}_u \times A_u\}_u$ of complete edge-disjoint bipartite graphs, in $O((m^{2/3}n^{2/3} + m + n)\log m)$ time and space. The number of graphs obtained is $O(m^{2/3}n^{2/3} + m + n)$, and we have $\sum_u |A_u|, \sum_u |\mathcal{M}_u| = O((m^{2/3}n^{2/3} + m + n)\log m)$. Each such point-annulus containment pair appears in exactly one of these graphs.*

For each $b \in B$ draw the annulus of radii r and r' that is centered at b. Let $\mathcal{M}$ be the set of these annuli. Clearly, $r \le ||q - b|| \le r'$ for some point q iff q lies inside the annulus associated with b. We apply Theorem 2.1 to the sets A and $\mathcal{M}$ and obtain in time $O(n^{4/3}\log n)$ a collection of $O(n^{4/3})$ bipartite graphs H_u such that $\sum_u |A_u|, \sum_u |\mathcal{M}_u| = O(n^{4/3}\log n)$. We now create a few auxiliary linked lists so that we don't report a point in B more than once. First, we convert the vertex sets of the graphs H_u into doubly linked lists. Next, for each $a \in A$ we create a doubly linked list of pointers to the occurrences of a in the lists A_u, and have each such occurrence of a point back at the pointer to it. For each point $b \in B$ we create a linked list of pointers to its occurrences in the lists $\mathcal{M}_u$. All this can be done in $O(n^{4/3}\log n)$ time and space by traversing the vertex sets of the graphs H_u.

Now in order to report all the neighbors of a point $a \in L_1$ that have not been reported yet for some other point, that is, in order to execute the While loop in the procedure above, we proceed as follows. Traverse the list associated with a. For each occurrence of a in a list A_u report all the points b in $\mathcal{M}_u$. For each such reported b, remove all occurrences of b in the lists $\mathcal{M}_u$. For each vertex a' in a list A_u containing a, remove from the linked list associated with a' the pointer to this occurrence. Clearly, the overall number of basic operations that are performed during the construction of the forest is only $O(n^{4/3}\log n)$.

Remark 2.2: Note that if the underlying norm is L_∞, then, even if the input points are in d-space for some $d > 2$, it is easy to find a most uniform matching in time $O(n^3\text{polylog}n)$. This is done by constructing a two level orthogonal range tree, where the first level enables us to find the points of B lying at distance at least r from a query point, as a polylogarithmic number of disjoint subsets of B, and the second level enables us to find out of the points found in the first level those lying at distance at most r' from the query point. Details are standard and hence omitted from this version. Summarizing, we have:

Theorem 2.3 *Let A and B be two sets of n points. It is possible to compute a most uniform matching in time $O(n^{10/3} \log n)$ when the points are in $\mathbb{R}^2$ and the underlying norm is L_2, or in time $O(n^3 polylogn)$ when the points are in d-space, $d \geq 2$, and the underlying norm is L_∞.*

3 Applications to Pattern Matching

Note that an obvious variant of the algorithm of the preceding section (with the same running time) finds a matching M^* for which $\max(M) - \min(M)$ is minimum among all matchings M between A and B of cardinality k, where $1 \leq k \leq n$ is some pre-determined parameter. This variant has the following interesting application.

Assume that the set $B' \subseteq B$ is a translated copy of the set $A' \subseteq A$, but every point in B' has been independently slightly perturbed. In addition, $A - A'$ and $B - B'$ consist of spurious points; that is, they were created as a result of noise. The problem is therefore to find A', B' and the translation by which B' was translated with respect to A'. Assume furthermore that we have some priori knowledge that the noisy points are spread randomly and that $|A'| \geq k$. One approach for solving this problem is to find a translation that minimizes the *Hausdorff distance* between large enough subsets of A and B; see [5, 6] for further details. This approach has the drawback that the computed matching is not necessarily one-to-one, and hence might not be appropriate, since we are certain in this case that a one-to-one matching between A' and B' exists.

On the other hand, it is reasonable to assume that the most uniform matching of size k will fit points of A' to their images in B', while most of the points of $A - A'$ and $B - B'$ will remain unmatched. (If we don't have an exact estimation of the size of A', we can compute such an estimation by a binary search, since we expect a significant increase in the differences corresponding to the most uniform matchings that are computed when moving from sizes below $|A'|$ to sizes above $|A'|$.) Hence our algorithm is most likely to identify the sets A' and B' and to give a useful initial translation to the algorithms of Efrat and Itai [7], Heffernan and Schirra [13], or Heffernan [14] that compute the exact translation (once A' and B' are known). We are not aware of any other way to solve this problem in less than $O(n^{3.5})$ time.

4 Computing a Minimum Deviation Matching

In this section we show how to find M_D^*, the matching that minimizes $(1/n)\Sigma(M) - \min(M)$. We continue to use the same notation as in the previous section.

Definition 4.1 *Let M^i be the perfect matching between A and B whose sum of distances is minimum among all matchings M for which $\min(M) \geq d^{(i)}$.*

Lemma 4.1 *Assume that $d^{(i)}$ is the shortest edge of M_D^*. Then $M_D^* = M^i$.*

In [3] an $O(n^{2+\varepsilon})$-time algorithm for finding M^1 was proposed. This algorithm is a variant of Vaidya's algorithm for solving the same problem. We use a variant of the algorithm of [3] for finding M_D^*. First we find M^1, using the algorithm of [3] mentioned above. Next we perform

n^2-1 phases of the algorithm, where in the j-th phase we use M^{j-1} to find M^j, and compute $(1/n)M^j - d^{(j)}$. Lemma 4.1 shows that the smallest value encountered is M_D^*. It will be shown that the time needed for a phase (actually for obtaining M^j from M^{j-1}) is $O(n^{4/3+\varepsilon})$, so the total running time is $O(n^{10/3+\varepsilon})$.

Let G be a general weighted bipartite graph on $A \cup B$, where each edge (a_i, b_j) is associated with a weight $d(a_i, b_j)$. The bipartite weighted matching problem can then be formulated as a linear program:

$$\begin{array}{ll}
\min \ \sum_{i,j} d(a_i,b_j)x_{ij}\,, \quad \text{subject to} & \\
\sum_{j=1}^{n} x_{ij} = 1, & i = 1, \ldots, n, \\
\sum_{i=1}^{n} x_{ij} = 1, & j = 1, \ldots, n, \\
x_{ij} \geq 0, & i,j = 1, \ldots, n,
\end{array}$$

where (a_i, b_j) is an edge of M if and only if $x_{ij} = 1$. The dual linear program is

$$\begin{array}{ll}
\max \ \sum_{i} \alpha_i + \sum_{j} \beta_j\,, \quad \text{subject to} & \\
\alpha_i + \beta_j \leq d(a_i,b_j), & i,j = 1, \ldots, n,
\end{array}$$

where α_i (resp. β_j) is the dual variable associated with the point a_i (resp. b_j). A necessary and sufficient condition for the optimality of the solution is that the orthogonal conditions below hold:

$$x_{ij} > 0 \ \Rightarrow \alpha_i + \beta_j = d(a_i,b_j) \quad i,j = 1 \ldots n \tag{1}$$
$$\alpha_i \neq 0 \ \Rightarrow \textstyle\sum_j x_{ij} = 1 \quad i = 1 \ldots n \tag{2}$$
$$\beta_j \neq 0 \ \Rightarrow \textstyle\sum_i x_{ij} = 1 \quad j = 1 \ldots n \tag{3}$$

Assume that in the last phase we have computed M^{k-1}. That is, we have found values $x_{i,j}$, α_i, β_j (for $1 \leq i,j \leq n$) such that all orthogonal conditions (1), (2), and (3) hold. We now seek M^k. If the edge $(a_{i'}, b_{j'})$ whose length is $d^{(k-1)}$ is not in M^{k-1} (and then all its edges are of length at least $d^{(k)}$), then surely $M^k = M^{k-1}$, and this phase is done. Otherwise we want to delete this edge, and make sure that no matching in the future will use it. This will be achieved by setting $d(a_{i'}, b_{j'}) \equiv \infty$, $x_{i',j'} = 0$ (that is, delete $(a_{i'}, b_{j'})$ from the matching), $\alpha_{i'} = 0$, and $\beta_{j'}$ is not changed. Hence all conditions remain valid, except for (3), which is violated. Note that at this stage, for each pair a_i, b_j whose (Euclidean) distance $||a_i - b_j||$ is less than $d^{(k)}$, we have $d(a_i, b_j) = \infty$. For all other edges $d(a_i, b_j) = ||a_i - b_j||$.

The Hungarian method computes a matching in n stages, each of which augments the matching by one edge and updates the dual variables. Let X be the current matching, i.e., $X = M^{k-1} - \{(a_{i'}, b_{j'})\}$. An edge (a_i, b_j) is called *admissible* if $d(a_i, b_j) = \alpha_i + \beta_j$. Due to the orthogonal conditions, all the edges of X are admissible. As in the previous section, we use an augmenting path to increase the cardinality of the matching back to n.

We search for an augmenting path consisting only of admissible edges, as follows. From the unique exposed vertex $b_{j'} \in B$, we grow (in an implicit manner) an 'augmenting tree' whose paths are augmenting paths starting at $b_{j'}$. More precisely, each point of $A \cup B$ in the augmenting tree is reachable from $b_{j'}$ by an augmenting path that consists only of admissible edges. For a point w of A (resp. B), the path leading to w ends at an edge not in X (resp. in X). Let $S \subseteq B$ and $T \subseteq A$ denote the set of points of B and of A that lie in the augmenting tree. At the beginning of a phase, $S \subseteq B$ is the singleton containing the exposed vertex $b_{j'}$, and $T = \emptyset$. Let

$$\delta = \min_{a_i \in A-T, b_j \in S} \{d(a_i, b_j) - \alpha_i - \beta_j\}.$$

At each step, the algorithm takes one of the following actions, depending on whether $\delta = 0$ or $\delta > 0$:

Case 1: $\delta = 0$. Let (a_i, b_j), for $a_i \in A - T$ and $b_j \in S$, be an admissible edge ($\delta = 0$ implies that such an edge must exist). If a_i is the exposed vertex $a_{i'}$, then an augmenting path has been found. Otherwise (a_i is matched), let b_k be the vertex matched (in X) to a_i. We add the edges (a_i, b_j) and (a_i, b_k) to the augmenting tree, the point a_i to T, and the point b_k to S.

Case 2: $\delta > 0$. The algorithm updates the dual variables, as follows. For each vertex $a_i \in T$, it sets $\alpha_i = \alpha_i - \delta$ and, for each $b_j \in S$, it sets $\beta_j = \beta_j + \delta$. Note that every edge of the augmenting tree remains admissible.

The algorithm repeats these steps until it reaches an exposed vertex of A, thereby obtaining an augmenting path. If an augmenting path Π is found, we delete the edges of $\Pi \cap X$ from the current matching X and add the other edges of Π to X (thereby increasing the size of the current matching by 1) to obtain the new matching M^k. Note that (1), (2), and (3) hold and hence the optimality is provided. This completes the description of the algorithm. Further details of the algorithm and the proof of its correctness can be found in [18, 23].

Vaidya suggested the following approach to expedite the running time of each step. Maintain a variable Δ and associate a weight with each point in $A \cup B$. In the beginning of each phase, $\Delta = 0$ and $w(a_i) = \alpha_i$, $w(b_j) = \beta_j$ for each $1 \leq i, j \leq n$. During each step, the weights and Δ are updated, but the values of the dual variables remain unchanged. This is done as follows. If Case 1 occurs, then we set $w(a_i) = \alpha_i + \Delta$ and $w(b_k) = \beta_k - \Delta$, and do not change the value of Δ. (Note that $a_i \notin T$ and $b_k \notin S$, so the values of α_i and β_k are the same as at the beginning of the phase.) If Case 2 occurs, then we set $\Delta = \Delta + \delta$, and do not change the weights. Notice that, for each $b_j \in S$, the current value of β_j is equal to $w(b_j) + \Delta$, and, similarly, for each $a_i \in T$, the current value of α_i is equal to $w(a_i) - \Delta$. Also, the current values of the dual variables for other points are equal to their values at the beginning of the phase. At the end of each phase, the values of the dual variables can be computed from Δ and from the weights of the corresponding points. The weight of each point changes only once during a phase, namely, when it is added either to S or to T. Moreover, at any time during a phase,

$$\delta = \min_{a_i \in A-T, b_j \in S} \{d(a_i, b_j) - w(a_i) - w(b_j)\} - \Delta\,.$$

For a parameter $r > 0$, let $\mathbf{F}_r$ denote the family of n functions whose i-th function is

$$d_i(x) = \begin{cases} ||x - a_i|| + w(a_i) & \text{if } ||x - a_i|| \geq r, \\ \infty & \text{Otherwise.} \end{cases}$$

Using obvious abuse of notation, we say that a point $a_i \in A - T$ is the closest in $A - T$ to a point $q \in \mathbb{R}^2$ if $d_i(q) \leq d_j(q)$, for $a_j \in A - T$. Our problem in thus to maintain the closest pair (using the distance functions defined above) between $A - T$ and S. To bound the running time of each operation, we need to argue about the complexity of the lower envelope of $\mathbf{F}_r$. The best bound we are aware of is $O(n^{2+\varepsilon})$, which results from [12, 21]. Hence we can maintain δ using the following theorem.

Theorem 4.2 *(Agarwal et al. [3, Thm. 6.8]) The closest pair between S and $A - T$, when the distance is measured using the distance functions $\mathbf{F}_{d(k)}$, can be maintained dynamically in time $O(n^{1/3+\varepsilon})$ for each operation. This data structure can be built in time $O(n^{4/3+\varepsilon})$.*

Since each step requires at most two update operations (inserting a point into S and deleting a point from $A - T$), each phase can be performed in time $O(n^{4/3+\varepsilon})$ by Theorem 4.2. Moreover, at the beginning of the k-th phase, we can build the data structure for $\mathbf{F}_{d(k)}$ in time $O(n^{4/3+\varepsilon})$. Thus, we have:

Theorem 4.3 *Given sets $A, B \subseteq \mathbb{R}^2$ of n points, we can find M_D^* in time $O(n^{10/3+\varepsilon})$, for every $\varepsilon > 0$.*

5 Computing a Bottleneck Matching in Higher Dimensions

Let A be a set of n points in $\mathbb{R}^d$, where n is even and $d \geq 3$. In this section, we show how to compute a bottleneck matching M_B^* in the (complete) Euclidean graph over A. As in the preceding sections, we assume that the $\binom{n}{2}$ distances between pairs of points in A are distinct. Let $B_r(p)$ denote the ball of radius r centered at point p. For two points $p, q \in \mathbb{R}^d$, let $lune(p, q)$ denote the region $B_{||p-q||}(p) \cap B_{||p-q||}(q)$, where $||p - q||$ is the distance between p and q. The *k relative neighborhood graph* of A, denoted kRNG, is a graph (A, E), where a pair of points $(p, q) \in E$ if and only if the number of points of A other than p and q that lie in $lune(p, q)$ is less than k. In the plane, Chang et al. [4] have proven that the 17RNG of A contains a bottleneck matching. Thus, in order to compute M_B^* in this case, it is enough to consider the 17RNG whose size is only $O(n)$ rather than the complete Euclidean graph over A whose size is $O(n^2)$. We then apply to it the general $O((n \log n)^{1/2} m)$ algorithm of Gabow and Tarjan [9]. Chang et al. compute the 17RNG in time $O(n^2)$, but it is easy to see that it can be computed in $O(n \text{ polylog}(n))$ time. Su and Chang [22] describe how to construct in $O(n \log n)$ time another linear-size graph, called the 17-Gabriel graph, that contains the 17RNG. Thus in both cases M_B^* can be computed in total time $O(n^{1.5} \log^{0.5} n)$.

We show below that for any fixed dimension d, there exists a constant $k = k(d)$ such that the kRNG of A contains a bottleneck matching. Since the size of a kRNG for a constant k remains linear also in higher dimensions (if the distances between pairs of points are distinct [2]), and since it can be computed in subquadratic time, M_B^* can also be computed in total time that is subquadratic even in higher dimensions. Moreover, we show that if the underlying norm is the L_∞ norm, then M_B^* can be computed in time $O(n^{1.5} \log^{0.5} n)$ in any fixed dimension.

[2]This size grows in degenerate situations; see [1]

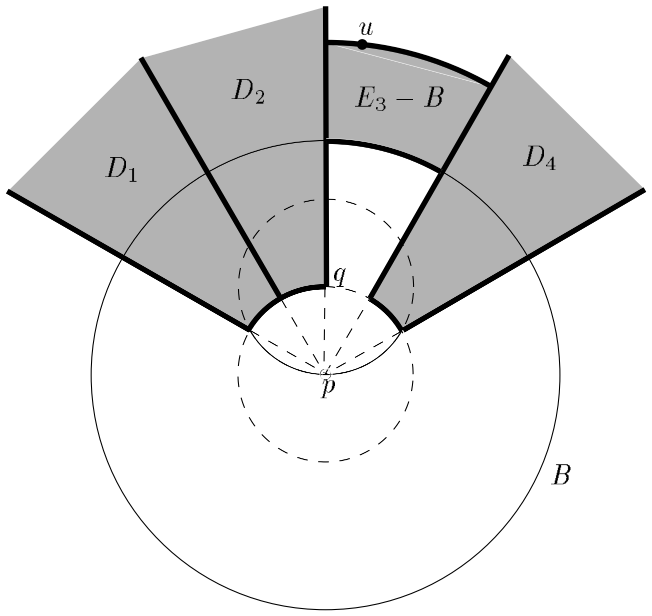

Figure 1: R_1 divided into pyramid-like regions

The proof of Chang et al. [4] to the assertion $M_B^* \subseteq k$RNG applies also in higher dimensions, provided that the following lemma, which is proven in [4] for the planar case, remains correct for an appropriate constant $k = k(d)$. Below we show that indeed this is the case.

Let M be a bottleneck matching, and let (p, q) be an edge of M. Let S be the subset of A consisting of the points that are matched with points lying in the interior of $lune(p, q)$, and assume that $S \cap lune(p, q) = \emptyset$ (since, if $u \in S$ lies inside $lune(p, q)$, then we could replace the edges (p, q) and (u, v) of M, where v is the point inside $lune(p, q)$ matched to u, by the edges (p, u) and (q, v) and obtain an improved matching). Hence, we assume that all the points in (the interior of) $lune(p, q)$ are matched with points lying outside of $lune(p, q)$. For a point $u \in S$, let n_u be its nearest point on the boundary of $lune(p, q)$ and put $r = ||p - q||$.

Lemma 5.1 *If for every $u, v \in S$*
(i) $d(u,p) > r$ *and* $d(u,q) > r$,
(ii) $d(u,v) > r$, *and*
(iii) $d(u,v) > d(u,n_u)$ *and* $d(u,v) > d(v,n_v)$,
then the number of points in S is some constant k depending only on the dimension d.

Proof: For clarity, we restrict ourselves in the proof to $\mathbb{R}^3$, however, the same proof with obvious modifications also holds for $d > 3$. We divide $\mathbb{R}^3 - lune(p, q)$ into three regions. Let C_p (resp. C_q) be the cone whose origin is p (resp. q) and its boundary contains the boundary of the disk $(B_r(p) \cap B_r(q))$. The first region R_1 is $C_p - lune(p, q)$, the second region R_2 is $C_q - lune(p, q)$, and the third region R_3 is $\mathbb{R}^3 - C_p \cup C_q$. We will show that the number of points in each region is less than some constant. The difficult case is to show this for the regions R_1, R_2, since the boundary of these regions contains a 2-dimensional face of $lune(p,q)$, while the boundary of R_3 contains only a 1-dimensional face of $lune(p,q)$. We will show this for R_1 (the proof for R_2 is completely analogous). Assume that p is the origin of our coordinate system and that the segment pq lies on the z-axis. We partition R_1 into a constant number of pyramid-like regions as follows. Consider the planes containing the y-axis that form angles of $30+\alpha$, $30+2\alpha$, etc. with the xy plane, and the planes containing the x-axis forming these angles with the xy plane, for some sufficiently small but fixed angle α. These planes together with the boundary of C_p partition the first region R_1 into a constant number of pyramid-like regions (see Figure 1 for an illustration in the plane). We will prove that each of these pyramids D_i contains only a constant number of points. Let $B = B_\rho(p)$, where ρ is an appropriate large constant. Let u be the point in $D_i \cap S$ that is furthest from p, and let $E_i = D_i \cap B_{||p-u||}(p)$. If $u \in B$ then so are all the other points in D_i, and clearly the number of these points is bounded by some constant, due to assumption (ii) above. Otherwise, u lies outside of B, but then $B_{||u-n_u||}(u) \supseteq E_i - B$, so there can be no other point in D_i outside of B, since such a point would necessarily lie inside $B_{||u-n_u||}(u)$ violating assumption (iii) above. □

In the next Lemma, whose proof appears in the appendix, we show that Lemma 5.1 is also true when the underlying norm is L_∞.

Lemma 5.2 *Assume L_∞ is the underlying norm, and that p, q and S are defined as in Lemma 5.1. Then $|S|$ is a constant $k = k(d)$.*

The remarks preceding Lemma 5.1 also hold for the L_∞ case, thus these remarks together with Lemma 5.1 and Lemma 5.2 lead to the following theorem.

Theorem 5.3 *Let A be a set of n points in d-space. Then for some appropriate constant k, the $kRNG$ of A contains a bottleneck matching of A.*

Computing a bottleneck matching

Let A be a set of n points in d-space. We show how to compute a graph $G = (A, E)$, such that $|E| = O(n)$, and the $kRNG$ of A (where k is a constant) is contained in G. The containment follows from arguments similar to those given in [4]. We first deal with the L_∞ case. Recall that using a standard orthogonal multi-level range tree, we can count (and find) all points lying in a query range in time $O(\log^{d-1} n)$ (using fractional cascading). Let $u \in A$, and let R^+ denote the region of $\mathbb{R}^d$ consisting of all points x such that all the components of the vector $x - u$ are non-negative (e.g., in the plane, R^+ is the north-eastern quadrant of u). The idea is to find the k nearest neighbors of u in $A \cap R^+$. This process is repeated for all $u \in A$, and for all the 2^d regions of $\mathbb{R}^d$ symmetric to R^+.

To find these neighbors, we check how many points of A are contained in a cube B of (yet unknown) radius, whose most-negative corner coincides with u. We seek the one that contains exactly k points. To determine the size of the cube, we use the technique of Fredrickson and Johnson [8] to perform a binary search in the (implicitly defined) matrix M containing all differences of the form $u.x_i - v.x_i$, where $u, v \in A$, and $u.x_i$ is the i'th coordinate of u. Hence we can find G in time $O(n \log^d n)$. We now apply the algorithm of Gabow and Tarjan to the graph G to obtain a bottleneck matching of A in time $O(n^{1.5} \log^{0.5} n)$.

As to the L_2 case, Agarwal and Matoušek [1] show how to compute the kRNG, where k is some constant, in $O(n^{2(1-1/(d+1))+\varepsilon})$ time. They construct first a super graph of kRNG which is also of linear size, so we may take this supergraph as input to the algorithm of Gabow and Tarjan. The construction time of this supergraph is only $O(n^{4/3} \log^2 n)$ for $d = 3$, and $O(n^{2-2/(\lceil d/2 \rceil+1)+\varepsilon})$ for $d \geq 4$ (See [2]). Thus we can compute M_B^* in $O(n^{1.5} \log^{0.5} n)$ for $2 \leq d \leq 4$, and $O(n^{2-2/(\lceil d/2 \rceil+1)+\varepsilon})$ for $d \geq 5$. Hence we have

Theorem 5.4 *Let A be a set of $2n$ points in d-space. A bottleneck matching of A can be computed in $O(n^{1.5} \log^{0.5} n)$ time for $2 \leq d \leq 4$, in $O(n^{1.5+\varepsilon})$ time for $5 \leq d \leq 6$, and in $O(n^{2-2/(\lceil d/2 \rceil+1)+\varepsilon})$ for $d > 6$. Under the L_∞ norm, a bottleneck matching of A can be computed in $O(n^{1.5} \log^{0.5} n)$ time for any fixed $d \geq 2$.*

Remark 5.5: The following observation is due to Pankaj Agarwal, it implies that finding a faster algorithm for computing a bottleneck matching in d-space, for $d > 6$, is probably a non-trivial problem. Assume that the point set A consists of two disjoint subsets A_1 and A_2, such that $|A_1| = |A_2| = n/2$, where $n/2$ is odd. Assume furthermore that the diameter of A_1 and the diameter of A_2 are very small compared to the diameter of A. In this case, a bottleneck matching of A must match the closest mixed pair, i.e., a pair a_1, a_2 where $a_1 \in A_1$ and $a_2 \in A_2$. But this is actually the famous *bichromatic closest pair problem in d-space* (see [2]), which for several years now has no faster solution that the one we gave.

Acknowledgment

We would like to thank Pankaj Agarwal, Rafi Hassin, Alon Itai and Micha Sharir for helpful discussions on the contents of this paper.

References

[1] P.K. Agarwal and J. Matoušek, Relative neighborhood graphs in three dimensions, *Computational Geometry: Theory and Applications* 2 (1992), 1–14.

[2] P.K. Agarwal, J. Matoušek and S. Suri, Farthest neighbors, maximum spanning trees and related problems in higher dimensions, *Computational Geometry: Theory and Applications* 1 (1992), 189–201.

[3] P.K. Agarwal, A. Efrat and M. Sharir, Vertical decomposition of shallow levels in 3-dimensional arrangements and its applications, *Proceedings 11 Annual Symposium on Computational Geometry*, 1995, 39–50.

[4] M.S. Chang, C.Y. Tang, and R.C.T. Lee, Solving the Euclidean bottleneck matching problem by k-relative neighborhood graphs, *Algorithmica* 8 (1992), 177–194.

[5] L.P. Chew, D. Dor, A. Efrat, and K. Kedem, Geometric pattern matching in d-dimensional space, *Proceedings of the Third Annual European Symposium on Algorithms* 1995, edited by Paul Spirakis, 264–279.

[6] L.P. Chew and K. Kedem, Improvements on geometric pattern matching problems, *Algorithm Theory – SWAT'92*, edited by O. Nurmi and E. Ukkonen, Lecture Notes in Computer Science, Vol. 621, Springer-Verlag, July 1992, 318–325.

[7] A. Efrat and A. Itai, Improvements on bottleneck matching and related problems, using geometry, *Proceedings 12 Annual Symposium on Computational Geometry*, 1996, to appear.

[8] G.N. Frederickson and D.B. Johnson, Generalized selection and ranking sorted matrices, *SIAM J. Computing* 13 (1984), 14–30.

[9] H.N. Gabow and R.E. Tarjan, Algorithms for two bottleneck optimization problems, *J. Algorithms* 9 (1988), 411–417.

[10] Z. Galil and B. Schieber, On finding most uniform spanning trees, *Discrete Applied Mathematics* 20 (1988), 173–175.

[11] S.K. Gupta and A.P. Punnen, Minimum deviation problems, *Oper. Res. Lett.* 7 (1988), 201–204.

[12] D. Halperin and M. Sharir, New bounds for lower envelopes in three dimensions, with applications to visibility of terrains, *Discrete and Computational Geometry* 12 (1994), 313–326.

[13] P.J. Heffernan and S. Schirra, Approximate decision algorithms for point set congruence, *SIAM J. Computing* 8 (1992), 93–101.

[14] P.J. Heffernan, Generalized approximate algorithms for point set congruence, *Proceedings 3 Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science* 1993, Vol. 709, Springer-Verlag, New-York–Berlin–Heidelberg, 373–384.

[15] J. Hopcroft and R. M. Karp, An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs, *SIAM J. Comput.* 2 (1973), 225–231.

[16] M. Katz, Improved algorithms in geometric optimization via expanders, *Proc. 3rd Israel Symposium on Theory of Computing and Systems*, 1995, 78–87. (See also, M.J. Katz and M. Sharir, An expander-based approach to geometric optimization, *SIAM J. Computing*, to appear.)

[17] H. Kuhn, The Hungarian method for the assignment problem, *Naval Research Logistics Quarterly* 2 (1955), 83–97.

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

[19] S. Martello and P. Toth, Linear assignment problems, Annals of Disc. Mathematics 31 (1987), 259–282.

[20] S. Micali and V.V. Vazirani, An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs, *Proceedings 21 Annual ACM Symposium on Theory of Computing*, 1980, 17–27.

[21] M. Sharir, Almost tight upper bounds for lower envelopes in higher dimensions, *Discrete and Computational Geometry* 12 (1994), 327–345.

[22] T.H. Su and R.C. Chang, The k-Gabriel graphs and their applications, *Proc. 1st Annu. SIGAL International Sympos. Algorithms*, LNCS 450, 1990, Springer-Verlag, 66–75.

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

Appendix

Proof: (of Lemma 5.2). Let $A_1 \subseteq A$ be the points lying at distance at most $2r$ from the boundary of $lune(p,q)$. Surely, by the second condition of the lemma, their number is bounded by some constant. Consider now the points in $A_2 \equiv A \setminus A_1$. Let f be a $(d-1)$-dimensional face of $\partial lune(p,q)$. Let $A_f \subseteq A$ denote those points of A_2 whose projection on $lune(p,q)$ lies on f, and let u be the point in A_f whose distance from f is the largest, among all points of A_f. Surely the projection of the ball $B_{||u-n_u||}$ (in L_∞-norm) on f covers f, hence as is easily seen, A_f contains only u ($d(u,n_u)$ is of radius at least $2r$). See Figure 2 for a demonstration.

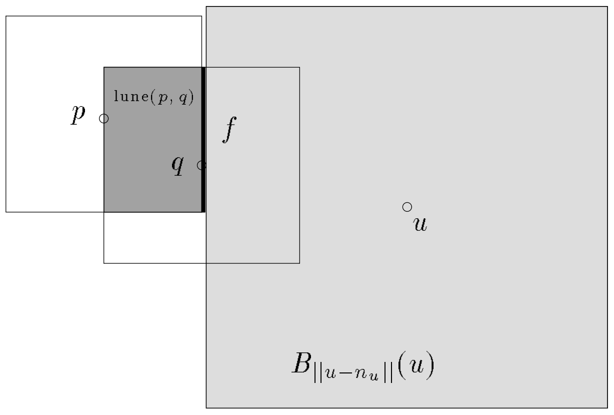

Figure 2: $lune(p,q)$ under the L_∞-norm

Next, let e be a $(d-2)$-dimensional face of $\partial lune(p,q)$, and let A_e be those points of A_2 whose projection lies on e. Repeating the very same argumentation as in the previous case, shows that $|A_e| \le 1$. Similarly, we treat faces of $\partial lune(p,q)$ of lower dimensions. □