# Bottleneck Segment Matching

Arita Banik* Matthew J. Katz Marina Simakov*

Department of Computer Science, Ben-Gurion University, Israel
aritrabanik@gmail.com {matya,simakov}@cs.bgu.ac.il

## Abstract

Let $R$ be a set of $n$ red segments and $B$ a set of $n$ blue segments, we wish to find the minimum value $d^*$, such that there exists a perfect matching between $R$ and $B$ with bottleneck $d^*$, i.e., the maximum distance between a matched red-blue pair is $d^*$. We first solve the corresponding decision problem: Given $R$, $B$ and a distance $d > 0$, find a maximum matching between $R$ and $B$ with bottleneck at most $d$. We begin with the simpler case where $d = 0$ and then extend our solution to the case where $d > 0$. We focus on the settings for which we are able to solve the decision problem efficiently, i.e., in roughly $O(n^{1.5})$ time. The most general of these, is when one of the sets consists of disjoint arbitrary segments and the other of vertical segments. We apply similar ideas to find a matching in the setting in which the vertical segments are replaced by points in the plane.

After solving the decision problem, we explain how to find the minimum value $d^*$. Finally, we show how to compute a shortest path tree for a given set of $n$ orthogonal segments and a designated root segment in $O(n \log^2 n)$ time.

## 1 Introduction

The *maximum matching* problem is a fundamental problem in graph theory. Given a graph $G = (V, E)$, find a matching in $G$ of maximum cardinality, where $M \subseteq E$ is a *matching* in $G$ if each vertex of $V$ is adjacent to at most one edge of $M$. If $|V|$ is even and $|M| = |V|/2$, then $M$ is *perfect*. The maximum matching problem has received a lot of attention, see below for some of the known algorithmic results. When $G$ is a weighted graph, i.e., when each edge of $E$ is assigned some weight, then one often is interested in a *minimum weight matching* or *bottleneck matching* in $G$, i.e., in a perfect matching that minimizes the sum of the edge weights or the weight of the heaviest edge, respectively. In this context, bipartite graphs received special attention, since many of the motivating problems are naturally modeled using bipartite graphs.

*Work by A. Banik and M. Simakov was partially supported by the Lynn and William Frankel Center for Computer Sciences.

Bottleneck matching and minimum weight matching in geometric graphs, i.e., in graphs induced by geometric settings, are well-studied topics. The most common geometric setting is of course points in the plane. For a set $P$ of points in the plane, the bottleneck matching problem (alternatively, the minimum weight matching problem) is to compute a bottleneck matching (resp., a minimum weight matching) in the complete graph induced by $P$, where the weight of an edge $e = \{p, q\}$ is the Euclidean distance between points $p$ and $q$. In the bipartite version of this problem, one is given two sets of points, a red set $R$ and a blue set $B$, each of size $n$, and the induced bipartite graph consists of all red-blue edges.

In this paper we study maximum matching problems in bipartite graphs induced by a pair of sets of line segments, i.e., a red set $R$ and a blue set $B$, each consisting of $n$ line segments. We are not aware of any previous results dealing with these problems. We are mainly interested in the variants for which a maximum matching can be found efficiently, i.e., in time roughly $O(n^{1.5})$.

In the first variant that we study, $R$ is a set of vertical segments and $B$ is a set of arbitrary disjoint segments, and there is an edge between $r \in R$ and $b \in B$ if and only if the two segments intersect. The goal is to compute a maximum matching in this graph. One could do this by applying one of the known graph algorithms for maximum matching in bipartite graphs, however, the running time of these algorithms is in general superquadratic; it is either $O(\sqrt{n}m)$ [4], or $O(n^{2.376})$ [8], or $\tilde{O}(m^{10/7})$ [7], where $m$ is the number of edges in the graph. We present an $O(n^{1.5} \log^2 n)$-time algorithm for this variant.

In the second variant, we are also given a distance $d > 0$, such that there is an edge between $r \in R$ and $b \in B$ if and only if the distance between the two segments is at most $d$; we denote this graph by $G(R, B, d)$. We would like to compute a maximum matching in this graph. This variant is more difficult than the former, and we present an $O(n^{1.5+\varepsilon})$-time algorithm for it.

When $R$ is also a set of arbitrary disjoint segments, the time bound increases to roughly $O(n^{11/6})$ (which is still subquadratic), for both variants above.

Our algorithms are based on the algorithm of Efrat et al. [2] for computing a maximum matching in a bi-

partite graph induced by a set of $n$ red points and a set of $n$ blue points in the plane, where there is an edge between a red and a blue point if and only if the distance between them is at most $d$, for a given parameter $d$. For each of the variants above, we need to replace the "oracle" component in their algorithm with a different oracle that meets our needs, see below for more details.

Using similar ideas, we also obtain an efficient algorithm for the following "mixed" problem. Given a set of $n$ arbitrary disjoint segments and a set of $n$ points, compute a maximum matching in the induced bipartite graph, where there is an edge between a segment and a point if and only if the distance between them is at most $d$, for a given parameter $d$. This problem can be viewed as a special case of the second variant above.

The second variant above is actually the decision version of the following optimization problem: Given $R$ and $B$, find the minimum distance for which there exists a perfect matching in the graph $G(R, B, d)$, or, in other words, find a bottleneck matching in the graph $G(R, B, \infty)$. We show that this optimization problem can be solved within the same time bound, i.e., in $O(n^{1.5+\varepsilon})$ time.

Finally, we present an efficient algorithm for computing a shortest path tree from a designated segment $s$. Consider a scene consisting of $n$ horizontal and vertical segments and let $s$ be one of the segments. Let $G$ be the bipartite graph induced by the scene (i.e., there is an edge between a horizontal and a vertical segment if and only if they intersect). The distance in $G$ between two segments is the length (in terms of number of edges) of a shortest path between them. We wish to compute a shortest path tree $T$ from $s$, that is, a tree rooted at $s$, such that, for any other input segment $s'$, the length of the path in $T$ between $s$ and $s'$ is the distance between them in $G$. We show how to construct a shortest path tree from $s$ in $O(n \log^2 n)$ time.

## 2 Solving the decision problem

### 2.1 The basic segment matching problem

The basic segment matching problem is defined as follows: let $R$ be a set of $n$ vertical segments and $B$ a set of $n$ disjoint arbitrary segments, find a maximum matching between $R$ and $B$, where two segments may be matched only if they intersect. This is the decision problem of the extended problem for the case $d = 0$.

Consider the corresponding bipartite intersection graph $G = (V, E)$, where each segment in $R \cup B$ corresponds to a vertex in $V$, and the edge $(r, b)$ between a red vertex $r$ and a blue vertex $b$ is in $E$ if and only if the segments intersect. Computing $G$ explicitly requires $O(n^2)$ time, and the best known graph-theoretic bipartite matching algorithm runs in $O(n^{2.376})$ time [8] (or $O(\sqrt{n}m)$, where $m = |E|$ [4]). We seek a subquadratic solution.

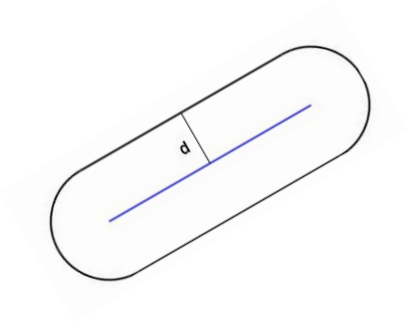



Figure 1: A segment's arena (race track) of radius d.

**Theorem 1** *The basic segment matching problem can be solved in time $O(n^{1.5} \log^2 n)$.*

**Proof.** Efrat et al. [2] show how to find a maximum matching without constructing the entire graph, by implicitly computing augmenting paths until a maximum matching is obtained. Their solution relies on an oracle, which is actually a data structure supporting a certain type of queries and an update operation. They show that if each of these operations (i.e., handling a query and deletion) can be performed within $T(n)$ time, then the maximum matching problem can be solved in $O(n^{1.5} \cdot T(n))$.

In our case, the oracle data structure, $D(S)$, would be a segment tree for a set of segments $S \subseteq B$, which consists of disjoint arbitrary segments. The data structure requires $O(n \log n)$ space (see, e.g., [1]). The required operations are defined as follows:

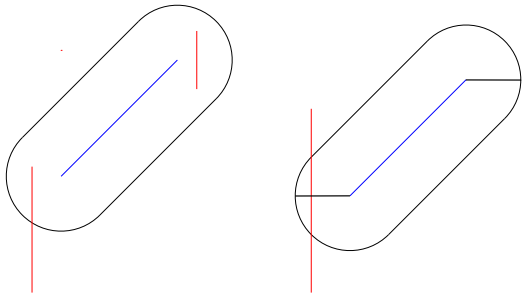
- **match(D(S), q)** — For a query segment $q \in R$, return a segment $s \in S$ such that $q$ intersects $s$, or *null* if no such segment exists.
- **delete(D(S), s)** — Delete the segment $s$ from $S$.

Since we are using a segment tree and the the query segments are vertical, each operation can be completed in $T(n) = O(\log^2 n)$ time [1]. Thus, by using the oracle we can compute a maximum matching in $O(n^{1.5} \cdot T(n)) = O(n^{1.5} \log^2 n)$. □

### 2.2 An extended segment matching problem

After solving the basic matching problem, we consider the case where $d > 0$. We define the extended segment matching problem: let $R$ be a set of $n$ vertical segments and $B$ a set of $n$ disjoint arbitrary segments, find a maximum matching between $R$ and $B$, where two segments may be matched if the distance between them is at most $d$.

Given a segment $b \in B$, denote the *arena* (or race track) of radius $d$ induced by it by $A_d(b)$ (see Figure 1).



(a) Case 2 (b) Case 3

Figure 2: Matching segments.

In order to solve the given problem, we require a data structure which supports the following operations:

- **match(D(S), q)** — For a query segment $q \in R$, return a segment $s \in S$, such that $q \cap A_d(s) \neq \emptyset$, or *null* if no such segment exists.
- **delete(D(S), s)** — For a segment $s \in S$, delete $s$ from $D(S)$ to prevent $s$ from being returned again.

If we can solve these queries efficiently, we can obtain an efficient maximum matching algorithm, by applying the scheme of Efrat et al. [2], as we did for the basic matching problem.

Let us distinguish between the different cases in which we can match a vertical segment $r$ to a segment $b$ (notice that there is no restriction on $b$):

1. The segments $r$ and $b$ intersect.
2. At least one of $r$'s endpoints lies inside $A_d(b)$ (see Figure 2(a)).
3. The most difficult case we have to consider is the one in which the segments do not intersect, and none of $r$'s endpoints lies inside $A_d(b)$. Let us draw two horizontal segments of length $d$ starting at each of $b$'s endpoints and extending away from $b$ (see Figure 2(b)). If $r$ intersects one of the horizontal segments we have added, then it can be matched with segment $b$.

**Lemma 2** *If $b$ and $r$ are at distance at most $d$ from each other, then they satisfy at least one of the conditions described above.*

**Proof.** Given segments $b$ and $r$ at distance at most $d$ from each other, we will prove that $r, b$ satisfy at least one of the conditions described above. Assume the segments do not satisfy the first and second conditions, we will show that they must satisfy the third one. Note that this means that the segments do not intersect, and none of $r$'s endpoints lies inside $A_d(b)$. Since the distance between $r$ and $b$ is at most $d$, segment $r$ must intersect $A_d(b)$ in at least one point. Also, since $r$ is a vertical segment which does not intersect segment $b$, we infer that it must intersect one of the horizontal segments we have added. We conclude that the segments satisfy the third condition. □

We conclude that by detecting each of the three cases, we can determine whether two segments can be matched. We will maintain three data structures, each one allowing the detection of one of the cases. Our goal is to use the oracle once again, so each data structure must support the *match, delete* operations. The following data structures will be required:

1. Data structure $D_1$: A segment tree for the segments in $B$, as in the previous section. This data structure allows us to detect segments which satisfy the first condition. *match, delete* operations are performed in $O(\log^2 n)$.
2. Dynamic data structure $D_2$: A structure for the arenas $A_d(S)$, induced by a set of segments $S \subseteq B$. Let us define the required operations for using the oracle:
   - $match(D_2(A_d(S)), q)$ — Given point $q$, return a segment s, such that $q \in A_d(s)$. For a vertical segment $r \in R$ we will perform two queries using the segment's endpoints, this way we can match segments which satisfy the second condition.
   - $delete(D_2(A_d(S)), s)$ — Given segment $s \in S$, remove the arena $A_d(s)$ from the data structure. Note that this operation requires $D_2$ to be a dynamic data structure.

   **Lemma 3** *The complexity of the union of the arenas induced by $S$ is linear in $|S|$.*

   **Proof.** First we observe that any two arenas can intersect in at most two points, this is due to the assumption that the segments $b \in B$ are pairwise disjoint, and that the arenas are all of the same radius. By a result of Kedem et al. [6], we conclude that the complexity of the union of the arenas is linear in $|S|$. □

   Lemma 3 enables us to use the dynamic data structure of Efrat et al. [3], which is able to perform queries in $O(\log^2 n)$ time and deletions in $O(n^\varepsilon)$ time, for any constant $\varepsilon > 0$. Thus, the running time of a single *match, delete* operation in $D_2$ is bounded by $O(n^\varepsilon)$. The size of the data structure is near linear in $|S|$.
3. Data structure $D_3$: A segment tree for the horizontal segments of length $d$, starting at the endpoints of the segments in $B$ and extending away

from them. For each segment $b \in B$, we add two segments, thus the size of the segment tree remains $O(n \log n)$. This data structure allows the detection of segments which satisfy the third condition. Let us define the required operations:

- *match*$(D_3(S), q)$ — Given a vertical segment $q$, return a segment $s \in S$ such that $q$ intersects $s$.
- *delete*$(D_3(S), s)$ — Delete segment $s$ from $S$. When deleting $s$ we must also delete its 'twin' segment, so that the corresponding segment in $B$ would not be matched twice.

These operations can be implemented in time $O(\log^2 n)$ per operation.

Now, given a vertical segment $r \in R$, we will conduct at most four queries for each *match* operation of the oracle, and exactly four removals for each *delete* operation of the oracle. The running time of each of the oracle's operations is determined by the maximum running time in the three data structures, which is bounded by $O(n^\varepsilon)$. Thus, using the oracle, we conclude that the extended segment matching problem can be solved in $O(n^{1.5+\varepsilon})$ time. The following theorem summarizes the main result of this section.

**Theorem 4** *The extended segment matching problem can be solved in time $O(n^{1.5+\varepsilon})$.*

### 2.3 Maximum matching between segments and points

Let $d > 0$. An important special case of the extended segment matching problem is the problem of computing a maximum matching between a set of $n$ disjoint arbitrary segments and a set of $n$ points, both in the plane, where we match match a point to a segment if the Euclidean distance between them is at most $d$.

**Theorem 5** *The segments and points matching problem can be solved in time $O(n^{1.5+\varepsilon})$.*

**Proof.** We observe that the distance between point $p$ and segment $s$ is at most $d$ if and only if $p$ lies inside $A_d(s)$. Thus, this is the only case in which a matching between $p$ and $s$ is valid. By maintaining a single data structure, similar to $D_2$ in the previous section, we can detect all relevant matchings satisfying this condition. Since each operation is bounded by $O(n^\varepsilon)$, the overall matching problem can be solved in $O(n^{1.5+\varepsilon})$. □

### 2.4 The general case

The most general setting of the segment matching problem is when both sets consist of disjoint arbitrary segments. This case requires more sophisticated data structures, and balancing between the space allocation and the total time required for query processing in one round of the matching algorithm. This case can be solved in roughly $O(n^{11/6})$ time using $O(n^{4/3})$ space, which is still subquadratic, but significantly worse than our goal of roughly $O(n^{1.5})$ time.

### 2.5 Optimization

After solving the decision problem, we focus on the optimization problem which is defined as follows: let $R$ be a set of $n$ vertical segments and $B$ a set of $n$ disjoint arbitrary segments, find $d^*$, which is the smallest value $d$ for which there exists a perfect matching with bottleneck $d$.

**Theorem 6** *The segment matching optimization problem can be solved in time $O(n^{1.5+\varepsilon})$.*

**Proof.** We perform a binary search in the set of potential values. This set consists of all the distances between a segment in $R$ and a segment in $B$. Such a distance, if not 0, is determined by the distance between an endpoint of one of the segments and the other segment. The size of the set is $O(n^2)$, so we cannot afford to compute it explicitly. Instead, we slightly adapt the distance selection algorithm of Katz and Sharir [5], so that given $k$, it returns the $k$'th smallest distance in roughly $O(n^{4/3})$ time. For each potential value we run the algorithm for the decision problem described in section 2.2, each run requires $O(n^{1.5+\varepsilon})$, and there would be at most $O(\log n)$ potential values examined until $d^*$ is found. Thus, we reach the total running time of $(O(n^{4/3}) + O(n^{1.5+\varepsilon})) \cdot \log n = O(n^{1.5+\varepsilon})$. □

## 3 Computing a shortest path tree

The shortest path tree problem in our setting is defined as follows: Given $S$, a set of $n$ orthogonal segments and a segment $s \in S$, compute a shortest path tree $T$ rooted at $s$; (see Figure 3). We say that there is a *path* from $u$ to $v$ if there exist segments $u = s_1, s_2, \ldots, s_n = v \in S$, such that any two consecutive segments intersect and have different orientations. We show how to construct $T$ efficiently.

**Theorem 7** *Given $S$, a set of $n$ orthogonal segments, and a segment $s \in S$, we can compute a shortest path tree rooted at $s$ in $O(n \log^2 n)$ time.*

**Proof.** Our algorithm is based on the well-known BFS algorithm. We maintain two segment trees: one for

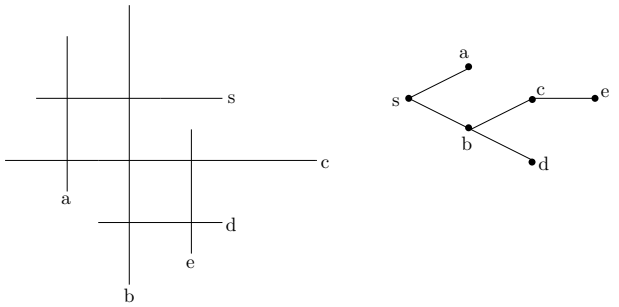



Figure 3: A shortest path tree for a set of orthogonal segments and root segment $s$.

the horizontal segments, $D_1$, and one for the vertical segments, $D_2$. Assume $s$ is a vertical segment, we can find all the segments at distance 1 from $s$ by performing a query in $D_1$, let us denote all these segments by $S_1$. In a similar manner, we can find all the segments at distance 2 from $s$ by performing a query in $D_2$ with each of the segments in $S_1$. A segment that is found during a query, is deleted from the appropriate data structure in $O(\log^2 n)$ time. We repeat the previous step, until no new intersections are found, implying that our shortest path tree is complete.

**Running time:** for a given segment $t$, a query requires $O(\log^2 n + k)$ time, where $k$ is the number of segments (remaining in the data structure) intersecting $t$. We perform at most $n$ queries for each data structure and each segment is returned at most once, thus overall the construction takes $O(n \log^2 n)$ time. □

**Acknowledgments** The authors would like to thank Alon Efrat for helpful discussions.

## References

[1] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry — Algorithms and Applications*. Springer-Verlag, 3rd edition, 2008.

[2] A. Efrat, A. Itai, and M. J. Katz. Geometry helps in bottleneck matching and related problems. *Algorithmica* 31 (2001), 1–28.

[3] A. Efrat, M. J. Katz, F. Nielsen, and M. Sharir. Dynamic data structures for fat objects and their applications. *Computational Geometry* 15 (2000), 215–227.

[4] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing* 2(4) (1973), 225-231.

[5] M. J. Katz and M. Sharir. An expander-based approach to geometric optimization. *SIAM Journal on Computing* 26(5) (1997), 1384–1408.

[6] K. Kedem, R. Livne, J. Pach, and M. Sharir. On the union of Jordan regions and collision-free translational motion amidst polygonal obstacles. *Discrete Comput. Geom.* 1 (1986), 59–71.

[7] A. Madry. Navigating central path with electrical flows: From flows to matchings, and back. In *Proc. 54th IEEE Symp. Foundations of Computer Science* pp. 253-262, 2013.

[8] M. Mucha and P. Sankowski. Maximum matchings via Gaussian elimination. In *Proc. 45th IEEE Symp. Foundations of Computer Science*, pp. 248-255, 2004.