Bottleneck Bichromatic Plane Matching of Points

Ahmad Biniaz* Anil Maheshwari* Michiel Smid*

Abstract

Given a set of n red points and n blue points in the plane, we are interested to match the red points with the blue points by straight line segments in such a way that the segments do not cross each other and the length of the longest segment is minimized. In general, this problem in NP-hard. We give exact solutions for some special cases of the input point set.

1 Introduction

We study the problem of computing a bottleneck non-crossing matching of red and blue points in the plane. Let $R = \{r_1, \ldots, r_n\}$ be a set of n red points and $B = \{b_1, \ldots, b_n\}$ be a set of n blue points in the plane. A *RB-matching* is a non-crossing perfect matching of the points by straight line segments in such a way that each segment has one endpoint in B and one in R. The length of the longest edge in an RB-matching M is known as *bottleneck* which we denote by λ_M. The *bottleneck bichromatic matching* (BBM) problem is to find a non-crossing matching M^* with minimum bottleneck λ^*. Carlsson et al. [4] showed that the bottleneck bichromatic matching problem is NP-hard. Moreover, when all the points have the same color, the bottleneck non-crossing perfect matching problem is NP-hard [1].

Notice that, the bottleneck (possibly crossing) perfect matching of red and blue points can be computed exactly in $O(n^{1.5} \log n)$ time [5]. In addition, a non-crossing perfect matching of red and blue points always exists and can be computed in $O(n \log n)$ time by applying the ham sandwich cut recursively. In [3] the authors considered the problem of non-crossing matching of points with different geometric objects.

In this paper we present exact solutions for some special cases of the BBM problem when the points are arranged in convex position, boundary of a circle, and on a line. For simplicity, in the rest of the paper we refer to a RB-matching as a "matching".

2 Points in Convex Position

In this section we deal with the case when $R \cup B$ form the vertices of a convex polygon. Carlsson et al. [4] presented an $O(n^4 \log n)$-time algorithm for points on convex position. We improve their result to $O(n^3)$ time. Let P denote the union of R and B, that is $P = \{r_1, \ldots, r_n, b_1, \ldots, b_n\}$. We have the following observation:

Observation 1 *Let (r_i, b_j) be an edge in any RB-matching of P, then there are the same number of red and blue points on each side of the line passing through r_i and b_j.*

Using Observation 1, we present a dynamic programming algorithm which solves the BBM problem for P. For simplicity of notation, let $P = \{p_1, \ldots, p_{2n}\}$ denote the sequence of the vertices of the convex polygon in counter clockwise order, starting at an arbitrary vertex p_1; see Figure 1. By Observation 1, we denote (p_i, p_j) as a *feasible edge* if p_i and p_j have different colors and the sequence $p_{i+1}, \ldots, p_{j-1}$ contains the same number of red and blue points. In other words we say that p_j is a *feasible match* for p_i, and vice versa. Let F_i denote the set of feasible matches for p_i. Figure 1 shows that $F_1 = \{p_4, p_8, p_{10}\}$. Therefore, we define a weight function w which assigns a weight $w_{i,j}$ to each pair (p_i, p_j), where

$$w_{i,j} = \begin{cases} |p_i p_j| & : \text{if } (p_i, p_j) \text{ is a feasible edge} \\ +\infty & : \text{otherwise} \end{cases}$$

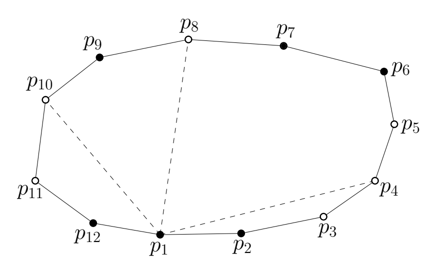


Figure 1: Points arranged on convex position.

Consider any subsequence $P_{i,j} = \{p_i, \ldots, p_j\}$ of P, where $1 \leq i < j \leq 2n$. Let $A[i, j]$ denote the bottleneck of the optimal matching in $P_{i,j}$ if $P_{i,j}$ has an RB-matching; otherwise, $A[i, j] = +\infty$. So $A[1, 2n]$ denotes the optimal solution for P. We use dynamic programming to compute $A[1, 2n]$. We derive a recurrence for $A[i, j]$. For a feasible edge (p_i, p_k) where $i + 1 \leq k \leq j$ and $p_k \in F_i$, the values of the sub-problems to the left and right of (p_i, p_k) are $A[i + 1, k - 1]$ and $A[k + 1, j]$.

*School of Computer Science, Carleton University, Ottawa, Canada. Research supported by NSERC.

We match p_i to a feasible point p_k which minimizes the bottleneck. Thus,

$$A[i,j] = \min_{\substack{i+1 \le k \le j \\ p_k \in F_i}} \{\max\{w_{i,j}, A[i+1,k-1], A[k+1,j]\}\}.$$

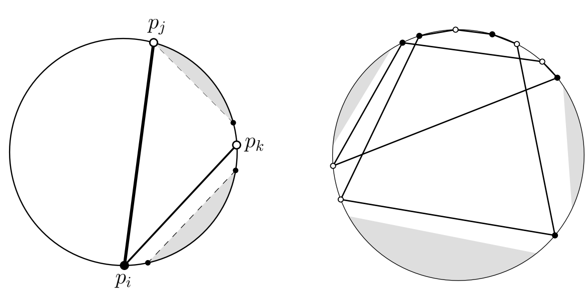
The size of A (which is the total number of sub-problems) is $O(n^2)$. For each sub-problem $A[i,j]$ we have at most $k = j - i$ lookups in A. Therefore, the total running time is $O(n^3)$.

Theorem 1 *Given a set B of n blue points and a set R of n red points in convex position, one can compute a bottleneck non-crossing RB-matching in time $O(n^3)$ and in space $O(n^2)$.*

Note that in [1] the authors showed that for points in convex position and when all the points have the same color, a bottleneck plane matching can be computed in $O(n^3)$ time and $O(n^2)$ space via dynamic programming. In the journal version of their paper [2] they extended their result and obtained the same time and space complexities for the bichromatic set of points.

2.1 Points on Circle

In this section we consider the BBM problem when the points in R and B are arranged on the boundary of a circle. Clearly, we can use the same algorithm as for points in convex position to solve this problem in $O(n^3)$ time. But for points on a circle we can do better; we present an algorithm running in $O(n^2)$ time. Consider $P = \{p_1, \ldots, p_{2n}\}$ as the sequence of the points in counter clockwise order on a circle. We prove that there is an optimal matching M^*, such that each point $p_i \in P$ is connected to its first feasible match in the clockwise or counter clockwise order from p_i.



(a) (b)

Figure 2: (a) illustrating the proof of Lemma 2, (b) the resulting graph of procedure CompareToOpt.

Lemma 2 *There is an optimal RB-matching for a point set P on a circle, such that each $p_i \in P$ is connected to its first feasible match in the clockwise or counter clockwise order from p_i.*

Proof. Consider an optimal matching M^* with an edge (p_i, p_j). Consider two arcs $\widehat{p_ip_j}$ and $\widehat{p_jp_i}$. W.l.o.g. let $\widehat{p_ip_j}$ be the smaller one. Clearly, the distance between any two points on $\widehat{p_ip_j}$ is at most $|p_ip_j|$. If $p_{i+1}, \ldots, p_{j-1}$ contains no feasible match for p_i, then p_j is the first feasible match to the right of p_i. Otherwise, let p_k be the first feasible match for p_i in $\widehat{p_ip_j}$; see Figure 2(a). By connecting p_i to p_k we have two smaller arcs $\widehat{p_{i+1}p_{k-1}}$ and $\widehat{p_{k+1}p_j}$. Obviously, $|p_ip_k| < |p_ip_j|$, and any matching of the vertices on $\widehat{p_{i+1}p_{k-1}}$ and $\widehat{p_{k+1}p_j}$ have the bottleneck smaller than $|p_ip_j|$. By repeating this process for all edges of M^* and all new edges, we obtain a matching M which satisfies the statement of the lemma and $\lambda_M \le \lambda^*$. □

As a result of Lemma 2, for each point $p_i \in P$, we consider at most two feasible matches in F_i. Thus, using the dynamic programming idea of the previous section, for each sub-problem $A[i,j]$ we have at most two lookups in A. Thus, it takes $O(n^2)$ time to fill the table A. By preprocessing P, for each point $p_i \in P$ we can find its first matched points in $O(n^2)$ time. Thus, the total running time of the algorithm is $O(n^2)$.

2.1.1 A Faster Algorithm

Let $R = \{r_1, \ldots, r_n\}$ be a set of n red points and $B = \{b_1, \ldots, b_n\}$ be a set of n blue points on the boundary of a circle C. Without loss of generality let $P = \{p_1, \ldots, p_{2n}\}$ be the clockwise ordered set of all the points. In this section we present an $O(n \log n)$ time algorithm which solves the BBM problem for P.

Let F_i denote the first feasible matches of p_i in clockwise and counter-clockwise order. Note that $|F_i| \le 2$. We describe how one can compute F_i for all points in P in linear time. First, consider the case that we are looking for the first clockwise-feasible match for each red point. We make a copy P' of P. Consider an empty stack, and start from an arbitrary red point r_{start} and walk on P' clockwise. If we see a red point, push it onto the stack. If we see a blue point p_j and the stack is not empty, we pop a red point p_i from the stack and add p_j to F_i, and delete p_i and p_j from P'. If we see a blue point p_j and stack is empty, we do nothing. The process stops as soon as we find the proper match for each red vertex. As we visit each point in P at most twice, this step takes linear time. We can do the same process for the counter-clockwise order. Therefore, F_i for all $1 \le i \le 2n$ can be computed in $O(n)$ time.

Let F denote the set of all feasible edges, in sorted order of their lengths. Let G be the graph with vertex set P and edge set F. Note that the degree of each vertex in G is at most two and hence the total number of edges is $2n$. Let G_λ be the subgraph of G containing all the edges of length at most λ. Our algorithm performs a binary search on the edges in G and for each considered edge e, we use the following procedure to decide whether G_λ, where $\lambda = |e|$, has a non-crossing perfect matching. The running time of the algorithm is $O(n \log n)$.

For each edge $e = (p_i, p_j)$ in G_λ let I_e be the set of all vertices of P in the smaller arc between p_i and p_j, including p_i and p_j. Let P_0 and P_1 be the lists of vertices of degree zero and one in G_λ, respectively. If P_0 is non-empty, then it is obvious that a perfect matching does not exist. If P_0 is empty and P_1 is non-empty, then for each point $p \in P_1$, do the following. Let $e = (p, q)$ be the only edge incident to p. It is obvious that any perfect matching in G_λ should contain e. In addition, (p, q) is a feasible edge, and then all the points in I_e can be matched properly. Thus, we can remove the points of I_e from G_λ. Note that this changes the lists P_0 and P_1. The algorithm CompareToOpt receives G_λ as input and decides whether it has a perfect non-crossing matching.

Algorithm 1 CompareToOpt(G_λ)

Input: a graph G_λ
Output: TRUE, if G_λ has a non-crossing perfect matching, FALSE, otherwise

```
P_0 ← vertices of degree zero in G_λ
P_1 ← vertices of degree one in G_λ
while P_0 ≠ ∅ or P_1 ≠ ∅ do
    if P_0 ≠ ∅ then return FALSE
    p ← a vertex in P_1
    q ← the vertex adjacent to p in G_λ
    for each r in I_(p,q) do
        remove r and its adjacent edges from G_λ
        update P_0 and P_1
return TRUE
```

The algorithm CompareToOpt consider each vertex and each edge once, so it executes in linear in the size of G_λ. At the end of the while loop, we have $P_0 = P_1 = \emptyset$. All the vertices of the remaining part of G_λ have degree two and this case is the same as the problem that we started with (BBM problem) and by Lemma 2, it has a perfect non-crossing matching, thus we return TRUE. See Figure 2(b).

Notice that, if the procedure returns FALSE for some λ, then we know that $\lambda < \lambda^*$. Let e be the shortest edge for which the procedure returns TRUE. Thus $|e| \geq \lambda^*$, and a bottleneck RB-matching is contained in G_λ, where $\lambda = |e|$.

Theorem 3 *Given a set B of n blue points and a set R of n red points on a circle, one can compute a bottleneck non-crossing RB-matching in time $O(n \log n)$ and in space $O(n)$.*

3 Blue Points on Straight Line

In this section we deal with the case where the blue points are on a horizontal line and the red points are on one side of the line. Formally, given a sequence $B_{1,n} = b_1, \ldots, b_n$ of n blue points on a horizontal line ℓ and n red points above ℓ, we are interested to find a non-crossing matching M between the points in R and B, such that the length of the longest edge in M is minimized. We show how to build dynamic programming algorithms that solve this problem. In Section 3.1 we present a bottom-up dynamic programming algorithm that solves this problem in $O(n^5)$ time. In Section 3.2 we present a top-down dynamic programming algorithm for this problem running in $O(n^4)$ time.

3.1 First algorithm

In this section we present a dynamic programming algorithm for the problem. We define a subproblem (R', B') in the following way: given a quadrilateral Q with one face on ℓ, we are looking for a bottleneck RB-matching in Q, where $R' = R \cap Q$ and $B' = B \cap Q$. For simplicity, we may refer to the sub-problem (R', B') as its bounding box Q. In the top level we imagine a bounding quadrilateral which contains all the points of R and B. See Figure 3(a). Let $b(Q)$ denote the bottleneck of the sub-problem Q. If Q is empty, we set $b(Q) = 0$. If Q is not empty but $|R'| \neq |B'|$, we set $b(Q) = +\infty$, as it is not possible to have a RB-matching for (R', B'). Otherwise, we have $|R'| = |B'| > 0$; let r_t be the topmost red point in R' in Q. It has at most $|B'|$ possible matching edges. Each of the matching edges defines two new independent sub-problems Q_l and Q_r to its left and right sides, respectively. See Figures 3(b) and 3(c). Thus, we can compute the bottleneck of a sub-problem Q, using the following recursion:

$$b(Q) = \min_{b_k \in B'} \{\max\{|r_t b_k|, b(Q_l), b(Q_r)\}\}.$$

Note that the y-coordinate of all the red points in Q_l and Q_r are smaller than y-coordinate of r_t. If we recurse this process on Q_l and Q_r, it is obvious that each sub-problem (R', B') is bounded by the left and right sides of its corresponding quadrilateral. Thus, each sub-problem is defined by a pair of edges (or possibly the edges of the outer bounding box).

Note that the total number of edges is $n^2 + 2$ (including the edges of the outer box). The dynamic programming table contains n^2+2 rows and n^2+2 columns, each corresponds to an edge. The cells correspond to sub-problems. The dynamic programming table contains $O(n^4)$ cells, and for each we have at most n pairs of possible sub-problems, which implies at most $2n$ lookups in the table. Therefore, the algorithm runs in time $O(n^5)$ and space $O(n^4)$.

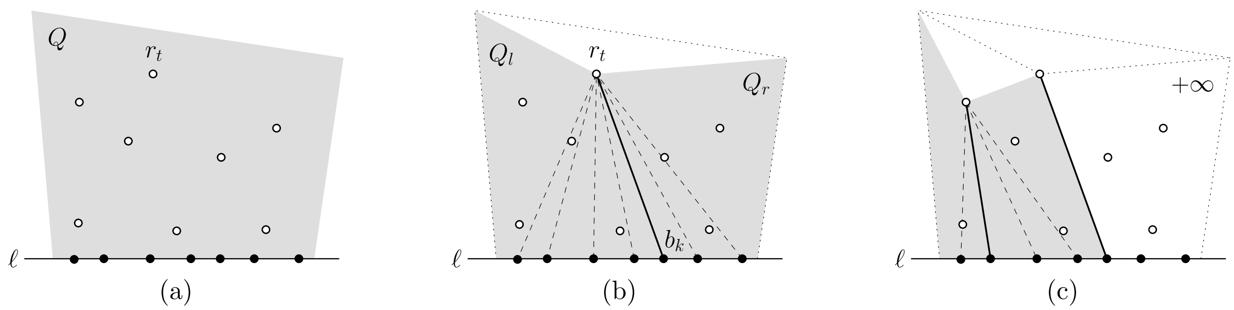


Figure 3: (a) definition of a sub-problem, (b) possible matching edges for r_t, and (c) Q_r returns $+\infty$ as it does not contain a matching; recurse on Q_l.

3.2 Second algorithm

In this section we present a top-down dynamic programming algorithm that improves the result of Section 3.1. Consider the problem (R, B), where $B = B_{1,n} = \{b_1, \ldots, b_n\}$. Let r_t be the topmost red point. In any solution M to the problem, consider the edge $(r_t, b_k) \in M$ which matches r_t to a point b_k in B, then there is no edge in M that intersects (r_t, b_k). Thus, (r_t, b_k) is a feasible edge if on each side of (r_t, b_k) the number of red points equals the number of blue points. In this case, b_k is a feasible match for r_t. Recall that F_t denotes the set of all feasible matches for r_t. See Figure 4(a). In other words,

$$F_t = \{k : (r_t, b_k) \text{ is a feasible edge}\}.$$

For a feasible edge (r_t, b_k), let R_l (resp. R_r) and B_l (resp. B_r) be the red and blue points to the left (resp. right) of (r_t, b_k), respectively. That is, the edge (r_t, b_k) divides the (R, B) problem into two subproblems (R_l, B_l) and (R_r, B_r), where $|R_l| = |B_l|$ and $|R_r| = |B_r|$. Clearly, $B_l = B_{1,k-1} = \{b_1, \ldots, b_{k-1}\}$ and $B_r = B_{k+1,n} = \{b_{k+1}, \ldots, b_n\}$. We develop the following recurrence to solve the problem:

$$b(R, B) = \min_{k \in F_t} \{\max\{|r_t b_k|, b(B_l, R_l), b(B_r, R_r)\}\}.$$

Let l_t denote the horizontal line passing through r_t. Note that the y-coordinate of all red points in R_l and R_r is smaller than the y-coordinate of r_t, and hence they lie below l_t. This implies that the left (resp. right) subproblem is contained in a trapezoidal region T_l (resp. T_r) with bounding edges ℓ, l_t, and (r_t, b_k). See Figure 4(a). Since, in each step we have two sub-problems, in the rest of this section we describe the process for the right sub-problem; the process for the left sub-problem is symmetric. Note that r_t is the top-left corner of the right sub-problem. Thus, given B_r and r_t, we know that r_t is connected to a blue point immediately to the left of B_r. In addition, we can find the red points assigned to the right sub-problem in the following way. Stand at a blue point immediately to the right of B_r and scan the plane clockwise, starting from ℓ. Count the red points in T_r while scanning, and stop as soon as the number of red points seen equals the number of blue points in B_r. These red points form the set R_r. See Figures 4(b) and 4(c).

Since, r_t defines the right (resp. left) and top boundaries of T_l (resp. T_r) which contains the left (resp. right) sub-problem, we call r_t a "boundary vertex". We define a sub-problem as a sequence $B_{i,j} = \{b_i, \ldots, b_j\}$ of blue points, a boundary vertex, r_t, connected to b_{j+1} (resp. b_{i-1}) for the left (resp. right) sub-problem. More precisely, a sub-problem $(B_{i,j}, r_t, d)$ consists of an interval $B_{i,j}$, a boundary vertex r_t, and a direction $d = \{left, right\}$ which indicates that r_t is connected to a point immediately to the left or to the right of $B_{i,j}$. For a sub-problem $(B_{i,j}, t, d)$, where $d = left$ we find the vertex set $R_{i,j}$ in the following way. Scan the plane by a clockwise rotating line s anchored at b_{j+1}. Count the red points in trapezoidal region formed by ℓ, l_t, and (r_t, b_{i-1}), and stop as soon as $j - i + 1$ red points have been encountered. These red points form the set $R_{i,j}$. See Figures 4(b) and 4(c).

In the top level, we add points b_0 and b_{n+1} on ℓ to the left and right of B, respectively. We add a point r_0 as the boundary vertex of the (R, B) problem in such a way that R and B are contained in the trapezoid formed by ℓ, l_0, and the line segment $r_0 b_0$. Thus in the top level we have the sub-problem $(B_{1,n}, r_0, left)$.

The dynamic programming table is a four-dimensional table $A[1..n, 1..n, 0..n, 1..2]$, where the first and second dimensions correspond to an interval of blue points, the third dimension corresponds a boundary vertex, and the fourth dimension corresponds to the directions. For simplicity we use l and r for $left$ and $right$ directions, respectively. Each cell $A[i, j, t, d]$ stores the bottleneck of the sub-problem $(B_{i,j}, r_t, d)$, and we are looking for $A[1, n, 0, l]$ which corresponds to the bottleneck of M^*. We fill A in the following way:

$$A[i, j, t, d] = \min_{k \in F_{t'}} \{\max\{|r_{t'} b_k|, A[i, k-1, t', r], A[k+1, j, t', l]\}\},$$

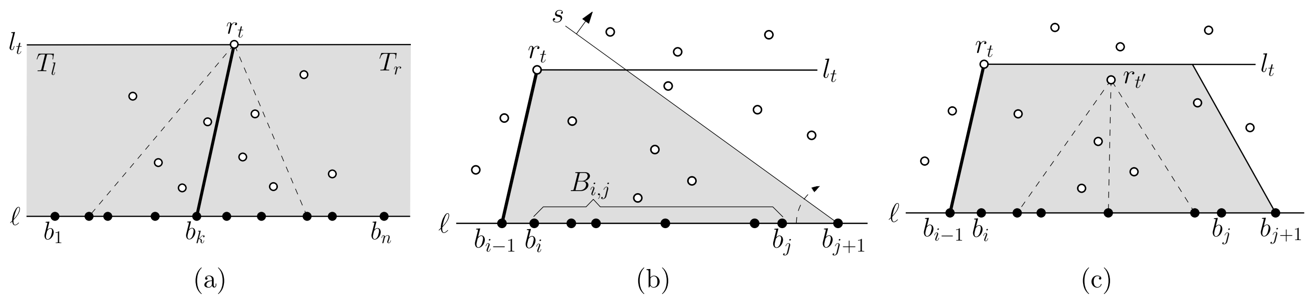


Figure 4: (a) feasible matches for r_t, (b) scanning the red points in the trapezoidal region (shaded area), and (c) the trapezoidal region which contains the same number of red and blue points.

where $r_{t'}$ is the topmost red point in the point set $R_{i,j}$ assigned to $(B_{i,j}, t, d)$.

Algorithm 2 computes the bottleneck of each subproblem using top-down dynamic programming. In the top level, we execute LineMatching$(1, n, 0, l)$. Before running algorithm LineMatching, for each point, we presort the red points in the following way. For each red point r, we keep a sorted list of all the red points below l_r in clockwise order. For each blue point, we keep two sorted lists of red points in clockwise and counterclockwise orders. This step takes $O(n^2 \log n)$ time.

Algorithm 2 LineMatching(i, j, t, d)

Input: sequence $B_{i,j}$, top point r_t, and direction d.
Output: bottleneck of M^*.

if $A[i, j, t, d] > 0$ **then**
    return $A[i, j, t, d]$
if $i > j$ **then**
    return $A[i, j, t, d] \leftarrow 0$
$R_{i,j} \leftarrow j - i + 1$ red points assigned to $B_{i,j}$
$t' \leftarrow \mathit{top\text{-}index}(R_{i,j})$
if $i = j$ **then**
    return $A[i, j, t, d] \leftarrow |r_{t'} b_i|$
$b \leftarrow +\infty$
$F_{t'} \leftarrow$ indices of feasible blue points for $r_{t'}$
for each $k \in F_{t'}$ **do**
    $A[i, k-1, t', r] \leftarrow$ LineMatching$(i, k-1, t', r)$
    $A[k+1, j, t', l] \leftarrow$ LineMatching$(k+1, j, t', l)$
    $m \leftarrow \max\{|r_{t'} b_k|, A[i, k-1, t', r], A[k+1, j, t', l]\}$
    if $m < b$ **then**
        $b \leftarrow m$
return $A[i, j, t, d] \leftarrow b$

Lemma 4 *Algorithm LineMatching computes the bottleneck of M^* in $O(n^4)$ time.*

Proof. Each cell $A[i, j, t, d]$ corresponds to a subproblem formed by an interval $B_{i,j}$, a boundary vertex r_t, and a direction d. The total number of possible $B_{i,j}$ intervals is $\binom{n}{2} + n$ (i can be equal to j). For each interval, any of the n red points can be the corresponding boundary vertex, which can be connected to the left or right side of the interval. Thus, the total number of subproblems is $2n\binom{n}{2} + 2n^2 = O(n^3)$. In order to compute $R_{i,j}$ for each sub-problem, we use the sorted lists assigned to b_{i-1} (or b_{j+1}) and scan for the red points in the trapezoidal region. To compute the feasible blue vertices for $r_{t'} \in R_{i,j}$, we use the sorted list assigned to $r_{t'}$ and keep track of feasible matches for $r_{t'}$ in $B_{i,j}$. Thus, for each sub-problem, we can compute $R_{i,j}$, $r_{t'}$, and $F_{t'}$ in linear time. Therefore, the total running time of the algorithm is $O(n^4)$. □

Finally, we reconstruct M^* from A in linear time.

Theorem 5 *Given a set B of n blue points on a horizontal line ℓ, a set R of n red points above ℓ, one can compute a bottleneck non-crossing RB-matching in time $O(n^4)$ and in space $O(n^3)$.*

References

[1] A. K. Abu-Affash, P. Carmi, M. J. Katz, and Y. Trabelsi. Bottleneck non-crossing matching in the plane. In *ESA*, pages 36–47, 2012.

[2] A. K. Abu-Affash, P. Carmi, M. J. Katz, and Y. Trabelsi. Bottleneck non-crossing matching in the plane. *Comput. Geom.*, 47(3):447–457, 2014.

[3] G. Aloupis, J. Cardinal, S. Collette, E. D. Demaine, M. L. Demaine, M. Dulieu, R. F. Monroy, V. Hart, F. Hurtado, S. Langerman, M. Saumell, C. Seara, and P. Taslakian. Non-crossing matchings of points with geometric objects. *Comput. Geom.*, 46(1):78–92, 2013.

[4] J. Carlsson and B. Armbruster. A bottleneck matching problem with edge-crossing constraints. Manuscript, see *http://users.iems.northwestern.edu/~armbruster/2010matching.pdf*, 2010.

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