One-Sided Crossing Minimization in Two-Layered Bipartite Graphs

Kristoffer Sandvang (jzc153) & Mateusz Filipowski (vwk910)

Supervisor: Mikkel Vind Abrahamsen

University of Copenhagen

June 10th 2024

1 Abstract

This report addresses the one-sided crossing minimization problem (OCMP) in two-layered bipartite graphs in the PACE 2024 challenge. The OCMP requires precise information about the number of edge crossings to confirm that a reduction in crossings has occurred. Therefore, this report also explored the counting of crossings, focusing on the Plane Sweep method. The OCMP has been proven to be NP-hard, even when one level of vertices is fixed. We explored the heuristic algorithms that compute a permutation of the non-fixed level. The known algorithms examined in this report are comprised of the assignment heuristic, the median heuristic, and the barycenter heuristic. For both the barycenter and median heuristics, we have proved that barycenter has a lower bound of $O(\sqrt{n})$ and median has a lower bound of $3 - O(\frac{1}{n})$ and an upper bound of 3. Furthermore, this report also explores the development of new heuristic algorithms, such as the parent heuristic and the bogo heuristic. We also explored tie-breaking methods for the median and barycenter heuristics. We performed tests on each heuristic, to determine the optimal conditions for each heuristic. This showed that the permutation tie-breaking method for the barycenter and median heuristics produced a solution with the fewest crossings. Ultimately, we have developed a hybrid algorithm that leverages the strengths of the median and barycenter heuristics and their tie-breaking variants under PACE2024-specific graph conditions and time constraints.

Contents

2 Introduction

This year's PACE 2024 challenge involves the one-sided crossing minimization problem (OCMP) [1]. The bipartite graph is drawn such that all vertices in G are on two horizontal lines, where the vertices in V_a are on the upper level, and the vertices in V_b are on the lower level. V_a is kept fixed, thus we only have to compute the optimal order of V_b to minimize the total number of edge crossings in G. The goal of the challenge is to compute a permutation of the non-fixed layer that produces the fewest crossings. An example of the one-sided crossing minimization problem and solution can be seen in figure 1. Here the original graph is shown in fig. 1a has 5 crossings. However, the number of edge crossings can be reduced by swapping vertices b_2 and b_3, thus producing 3 edge crossings (As shown in fig. 1b).

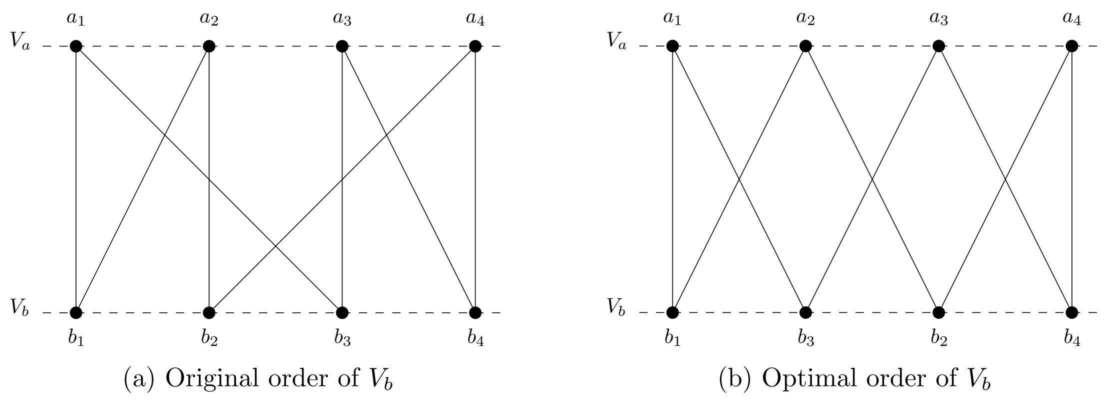


Figure 1: Computing the optimal order of V_b where vertices b_2 and b_3 are swapped to minimize the total number of crossings in G.

This year's PACE challenge has three tracks for solving the OCMP; a track for `Exact` algorithms for instances with few crossings that require an optimal solution for each given graph, a track for `Heuristic` algorithms that might require many crossings and should compute a good solution quickly, and a track for `Parameterized` algorithms for instances with small cutwidth [1]. The one-sided crossing minimization problem involves arranging nodes of one layer in a two-layered bipartite graph. This is an NP-hard problem, as proved by [2], and thus justifies the usages of heuristics. We have entered the `Heuristic` track and this report will therefore only address heuristic methods to solve this problem.

The minimization of the number of edge crossings in two-layered bipartite graphs is one of the most widely studied problems in the field of graph drawing [3], where several heuristic algorithms for this problem already exist, such as the barycenter and median heuristic. Both of these heuristics are discussed later in this report. Our approach to solving the one-sided crossing minimization problem ranges from implementing our own crossing minimization algorithms to exploring the many ways of optimizing the already known algorithms for the one-sided crossing minimization problem. We have used a GitHub repository to keep track of the implementation progress of our solver for the competition. We have implemented our solution in `C++`.

2.1 Overview

In section 3, we will explore two approaches to counting crossings. We describe each approach and analyze its time complexity. The first approach is the naive approach and the second approach is the plane sweep approach.

In section 4, we examine the assignment heuristic for edge-crossing minimization. Here we look at edge crossing minimization as an assignment problem and explain how the Hungarian reduction algorithm works. The assignment heuristic is also illustrated using an example graph.

In section 5, we describe how the barycenter heuristic works. We also prove that it has a lower bound of $O(\sqrt{n})$ and that it will always find a solution with zero crossings if one exists.

In section 6, the median heuristic is described. We prove the lower bound of the median heuristic is $3 - O(\frac{1}{n})$, and an upper bound of 3. Lastly, we will prove that if a bipartite graph has a solution with zero crossings, then the median heuristic will find it.

In section 7, we describe two of our self-developed heuristics to reduce crossings. The bogo heuristic and the parent heuristic. The parent heuristic stems from a suggestion by our thesis supervisor, Mikkel Vind Abrahamsen, and the bogo heuristic is inspired by the Bogo Sort algorithm (See [4]).

In section 8, we describe the test graphs, then we examine the different tie-breaking methods for the barycenter and median heuristics and try to determine the best tie-breaking method. Furthermore, we will explore the limitations of the permutation tie-breaking methods. Ultimately, we try to examine under what conditions the barycenter or median heuristic produces a permutation with the fewest crossings.

In section 9, we describe how our hybrid algorithm, which uses the barycenter and median heuristic and their different tie-breaking methods, works and why we have created it as such.

2.2 Notation

Throughout this report, we will refer to bipartite graphs as $G = (V_a \cup V_b, E)$. V_a will be the fixed level and is depicted as being the upper level of the bipartite graph. V_b will be the free level and is depicted as the lower level. Thus we will try to find a permutation of V_b, which will be denoted as π_{V_b}, that produces the fewest edge crossings. This is done to follow the input format for the PACE 2024 competition [1].

A vertex v will have two properties, *value* and x. The value property will contain the value of the vertex. V_a consist of vertices with values $\{0 \ \dots \ n0\}$ and V_b consists of vertices with values $\{n0 + 1 \ \dots \ n1\}$. This also allows us to determine if a vertex is in V_a or V_b in constant time. The x property contains the x-coordinate of the vertex.

3 Counting Crossings

A vital part of reducing crossings is counting the number of crossings in a two-layered drawing of a bipartite graph as optimizations of some algorithms require knowledge of the exact number of crossings.

3.1 Naive Approach

The naive approach of counting crossings will iterate through each edge uv and all other edges wz in the graph, where $u, w \in V_a$ and $v, z \in V_b$. It will then check if the following condition holds:

$$(u.x < w.x \ \wedge \ v.x > z.x) \ \vee (w.x < u.x \ \wedge \ z.x > v.x)$$

If the condition holds, then the edges will produce an edge crossing and otherwise, they will not cross each other. This counts crossings by ensuring that if uv starts before wz and ends after wz. It must mean that uv will cross wz. This also holds symmetrically when wz starts before uv and ends after uv. It is clear that this has a time complexity of $\mathrm{O}(|E|^2)$.

3.2 Plane Sweep Approach

In general, the plane sweep algorithm is an algorithm that counts the intersection of line segments in a set. To find these segments the algorithm performs a downward sweep using a horizontal line l starting from a position above all segments. The sweep line algorithm makes use of an event queue, which stores the events below the sweep line, and a balanced search tree, to store the line segments that intersect with the sweep line. Furthermore, the segments are ordered from left to right as they intersect the sweep line, meaning that the algorithm only looks at the segments that are adjacent in the horizontal ordering and only tests a new segment against its neighboring line segments. It then handles the events in order of the queue and updates the balanced search tree as line segments intersect or end. Thus achieving a time complexity of $\mathrm{O}(n \log n + I \log n)$, where I is the number of intersections [5].

However, in the case of counting crossings in a two-layered bipartite graph, we can use a simplified approach. The ordering of vertices in a bipartite graph can be stored in a doubly linked list, thus already forming the horizontal ordering of the vertices in each level, thus the line segments do not need to be ordered and the lower bound $\Omega(|E|log|E|)$ for the general case, does not hold here. Furthermore, the line segments all share the same y-interval as a line segment always goes from the upper level (V_a) of G to the lower level (V_b) of G or vice versa. Thus there is no need to use a balanced search tree to keep track of neighboring vertices. This allows a reduction of the time complexity by the factor $log|E|$ [6].

3.2.1 Description Of The Algorithm

The pseudo-code for this algorithm can be seen in algorithm 1. The pseudo-code in alg. 1 uses a function *Ord(v)*, this function returns the horizontal position of v that is odd for vertices in the upper level, V_a, and even for vertices in the lower level, V_b [6]. This means that the algorithm assumes that the x-coordinates are odd in the upper level, and even in the lower level. Furthermore, this algorithm considers the normally undirected edges in the bipartite graph to be directed from the left to the right, meaning that the vertex with the lowest x-value is considered the starting vertex of the edge and the other vertex to be the end vertex of the edge [6].
In our implementation we did not use this function, however, we were able to determine if a vertex

was in the upper or lower level based on the value of the vertex. This is due to the fact, that V_a consist of vertices with values $\{0 \ \ldots \ n0\}$ and V_b consists of vertices with values $\{n0 + 1 \ \ldots \ n1\}$. Thus we were able to implement the same function that also runs in $O(1)$ time. This means that the sweep line in our implementation touched both the upper and lower vertices at the same time, but treated the vertices in V_a as if they were touched first. The value property of the vertices also allowed us to use a hash map for the implementation of the *last_occurence* field for the vertices.

Lines 2-5 in alg. 1 initializes the crossing count to be 0. Furthermore, the doubly linked lists UL and LL are initialized to be empty and the size of UL and LL are zero. UL contains the end vertices of active edges that end in V_a, while LL contains the end vertices of active edges that end in V_b. An edge is considered active, if the sweep line has touched its starting vertex, but not its end vertex. Once the sweep line has touched an edge's end vertex, the edge is no longer considered active. In both lists, the vertices are sorted by the edge's starting vertex's x-coordinates, and if multiple edges originate from the same vertex, they will be sorted according to the end vertex's x-coordinate of the edges [6]. Finally, the field *last_occurrence(w)* is initialized to be *nil* for all vertices in the graph. The *last_occurrence* field contains the index of the last occurrence of w in either UL or LL and is updated when a vertex is inserted into either UL or LL. Line 6 moves the algorithm's vertical sweep line horizontally through the bipartite graph, having the sweep line touch a vertex alternating between V_a, the upper level of the graph, and V_b, the lower level of the graph. Lines 7 and 27 check if the vertex is in V_a or V_b. We will only explain what happens if the vertex is in V_a, as it is done symmetrically for when the vertex is in V_b.

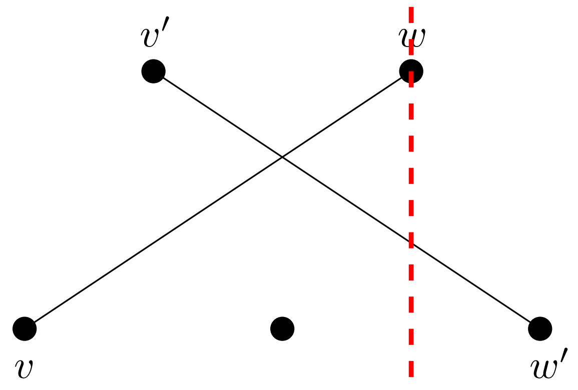
The algorithm counts crossings by handling two situations [6]:

Situation 1: If w in UL is touched by the sweep line, there are as many edges with endpoint w, as there are occurrences of w in UL. All these edges are crossed by all active edges with an end vertex in the lower level. This can be counted by looking at the size of LL. This is done symmetrically for if w is in LL. See fig. 2a

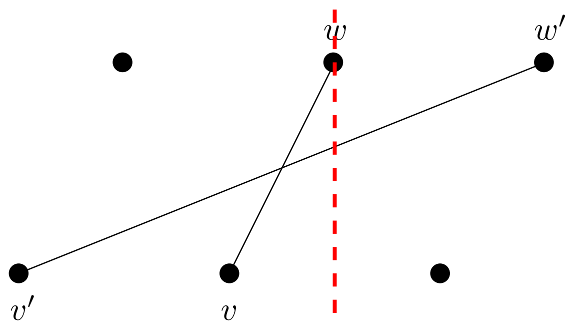
Situation 2: For each occurrence of w in UL there exist an edge e. All vertices $w' \neq w$ before this instance of w in UL are endpoints of active edges, and thus will cross e. This also holds symmetrically for if w is in LL. See fig. 2b

To avoid counting crossings twice, the algorithm inspects the crossings of an edge when the sweep line touches its end vertex. If the end vertex w of an edge e is in V_a, then e is crossed by all edges e' starting before w in V_a and ending after w in V_b. This is situation 1 in fig. 2a. Note that crossings of e with edges e' that end before w are already counted. the edge e is crossed by all edges starting before the start point v of e at the lower level, and ending after w at the upper level 2b.

In lines 8-19, the algorithm counts crossings if the sweep line touches a vertex that appears in UL. On line 8, 3 variables are initialized to be zero, k_1, k_2, and k_3. k_1 is the number of occurrences of a vertex w in UL, k_2 is the number of predecessors of w in UL that are active. Thus the number of crossings produced by the edge with endpoint w. k_3 is the sum of crossings produced by situation 2 for w [6]. On line 9, the algorithm checks if w appears in UL, and thus is an end vertex of an active edge. On lines 10-18, the algorithm iterates through the vertices v in UL from the beginning of UL till the last occurrence of w in UL. If $v = w$ then k_1 is incremented by one, k_3 is incremented by k_2, and then v is deleted from UL, and *UL_size* is updated. If $v \neq w$ then k_2 is incremented by one, as v must be a predecessor to w. On line 19 we count the crossings produced by edges ending in w by handling the two situations mentioned above. Situation one is $k_1 \cdot LL\_size$



(a) Situation 1 from [6]. It is clear that edges vw and $v'w'$ cross. In this example $LL = [w']$ and $LL\_size = 1$, and $UL = [w]$. Thus this produces $1 \cdot LL\_size$ crossings.



(b) Situation 2 from [6]. It is clear that edges vw and $v'w'$ cross. In this example $UL = [w', w]$, and $k_2 = k_3 = 1$ Thus this produces k_3 crossings.

Figure 2: Crossings situations from [6]

and situation two is k_3. Therefore the number of crossings produced by the edges ending in w is calculated as $k_1 \cdot LL\_size + k_3$. To preserve the precomputed number of crossings we update c as $c = c + k_1 \cdot LL\_size + k_3$.

Lines 21-26 iterates through each edge originating in w, and defines the edge's endpoint to be w'. It then adds w' to LL and updates the size field of LL and $last\_occurence(w')$ accordingly.

Once the algorithm has been through every vertex in $V_a \cup V_b$ it will terminate and return the total number of crossings, c, in the bipartite graph.

3.2.2 Local count crossings

We were able to modify this algorithm to also perform a local count crossings. To do so we modified the loop adding the endpoints of edges to LL (Line 21-26) in alg. 1. The modified loop can be seen below:

```
1: for all edges $e$ with start point $w$ in order according to Ord of their end points do
2:     let $w'$ = endpoint of $e$;
3:     if ($w'$ is in $V_b'$) then
4:         Add $w'$ at the end of $LL$;
5:         $LL\_size = LL\_size + 1$;
6:         $last\_occurence(w')$ = this new instance of $w'$ in $LL$;
7:     end if
8: end for
```

This uses linear search to determine if w' is in V_b', where $V_b' \subseteq V_b$, and adds a computational complexity of $O(|V_b'|)$. However, we knew this would only be used for the permutation tie-breaking method, which will be discussed in section 8.2.1, and thus $|V_b'| < 8$. Therefore it would not have a great impact on the time complexity of the original algorithm.

Algorithm 1 Count crossings sweep from [6]

```
procedure COUNT_CROSSINGS($V_a$, $V_b$)
    c = 0; UL_size = 0; LL_size = 0;
    UL = LL = *empty*;
    for $w$ in $V_a \cup V_b$ do $last\_occurence(w) = nil$;
    end for
    for $w$ alternating from $V_a$ and $V_b$, in the order of $Ord(w)$ do
        if ($Ord(w)$ is odd) then
            $k_1 = 0$; $k_2 = 0$; $k_3 = 0$;
            if ($last\_occurrence(w) \neq nil$) then
                for $v$ in $UL$ from start of $UL$ to $last\_occurrence(w)$ including, in that order do
                    if ($v == w$) then
                        $k_1 = k_1 + 1$;
                        $k_3 = k_3 + k_2$;
                        delete $v$ from $UL$;
                        $UL\_size = UL\_size - 1$;
                    else $k_2 = k_2 + 1$;
                    end if
                end for
                $c = c + k_1 \cdot LL\_size + k_3$;
            end if
            for all edges $e$ with start point $w$ in order according to Ord of their end points do
                let $w' =$ endpoint of $e$;
                Add $w'$ at the end of $LL$;
                $LL\_size = LL\_size + 1$;
                $last\_occurence(w') =$ this new instance of $w'$ in $LL$;
            end for
        else $Ord(w)$ is even.
            This is then done symmetrically to when $Ord(w)$ is even.
        end if
    end for
    return $c$
end procedure
```

3.2.3 Example

If we consider the graph G shown in fig. 3, and we initialize the linked lists UL and LL to be empty, and c to be 0. The sweep line algorithm would then sweep through G from left to right and populate

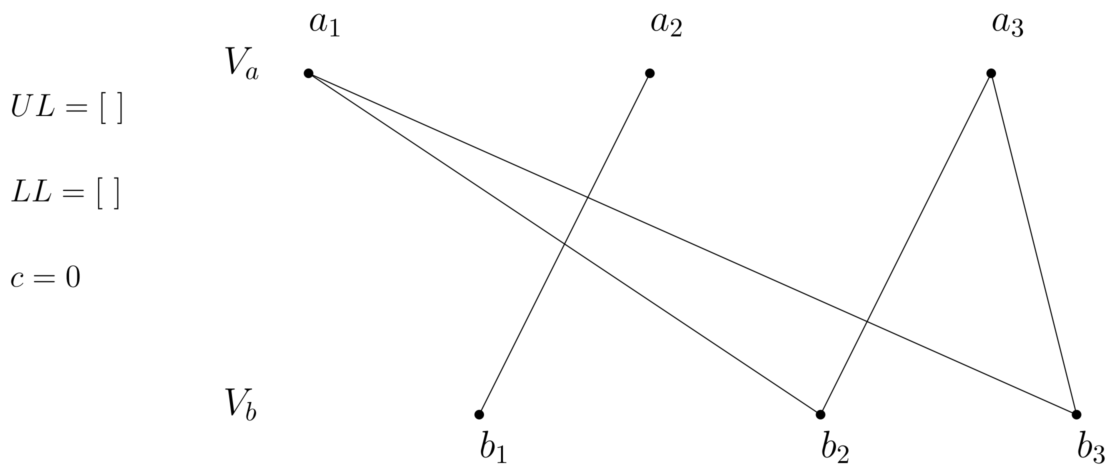
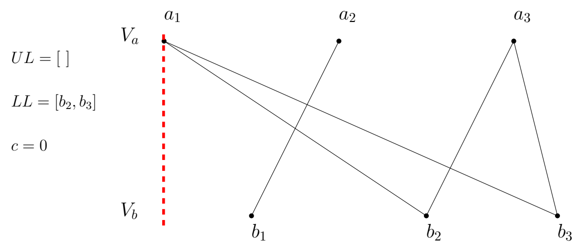


Figure 3: A bipartite graph G and the linked lists UL and LL initialized to be empty, and the number of crossings represented with c to be initialized to 0.

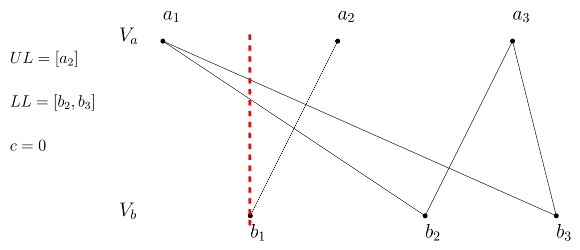
UL and LL. When the sweep line touches a_1 (fig. 4a), since $a_1 \notin UL$ it will add all edges starting in a_1 to LL, recall that this algorithm considers the otherwise undirected edges to be directed from left to right. Then the sweep line would continue and touch b_1 (fig. 4b) since $b_1 \notin LL$ it would add all edges starting in b_1 to UL, and continue to a_2. As the sweep line touches a_2 (fig. 4c), it will iterate through all elements in UL before the last occurrence of a_2 and let k_1 be the number of appearances of a_2 in UL, this means that $k_1 = 1$. It will then delete a_2 from UL, and calculate the number of crossings as $c = c + k_1 \cdot LL\_size + k_3$. In this case $k_3 = 0$, as no other vertices than a_2 appear in UL. Thus we have that $c = 0 + 1 \cdot 2 + 0 = 2$. Since there is no edges originating in a_2 the sweep line continues to b_2. When the sweep line touches b_2 (fig. 4d), it will iterate through all elements in LL before the last occurrence of b_2 and let k_1 be the number of appearances of b_2 in LL, this means that $k_1 = 1$. Since b_2 is the first element of LL the loop will terminate after deleting b_2 from LL and thus $k_2 = k_3 = 0$. Thus we have $c = c + k_1 \cdot LL\_size + k_3 = 2 + 1 \cdot 0 + 0 = 2$, and thus the number of crossings is not changed. Then a_3 is added to UL and the sweep line continues to a_3. As the sweep line touches a_3 (fig. 4e), it will iterate through all elements in UL before the last occurrence of a_3 and let k_1 be the number of appearances of a_3 in UL, this means that $k_1 = 1$. Since a_3 is the only element of UL the loop will terminate after deleting a_3 from UL and thus $k_2 = k_3 = 0$. Thus we have $c = c + k_1 \cdot LL\_size + k_3 = 2 + 1 \cdot 1 + 0 = 3$. Then b_3 is added to LL and the sweep line continues to b_3. Once the sweep line reaches b_3 (fig. 4f) it will iterate through all elements in LL before the last occurrence of b_3 and let k_1 be the number of appearances of b_2 in LL, this means that $k_1 = 2$. Since b_2 is the only element of LL the loop will terminate after deleting b_2 from LL twice and thus $k_2 = k_3 = 0$. Thus we have $c = c + k_1 \cdot LL\_size + k_3 = 3 + 2 \cdot 0 + 0 = 3$, and thus the number of crossings is not changed. Then the sweep line algorithm terminates as there are no more vertices in either layer and the total number of crossings $c = 3$.

3.2.4 Analysis of the algorithm

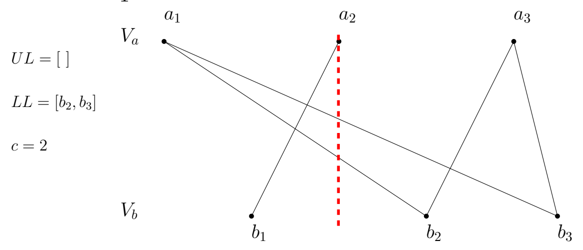
Insertion at the end of the list and deletion from UL and LL have a time complexity of O(1) since UL and LL are doubly linked lists. Lines 2-5, 6-8, and 27 are performed $O(|V_a| + |V_b|)$ times. All



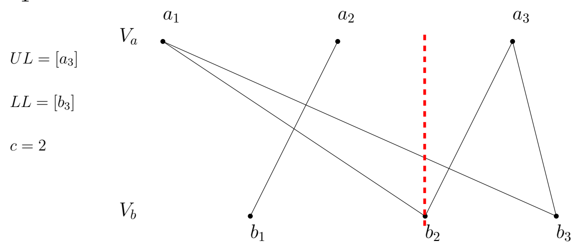
(a) b_2 and b_3 are added to LL, as the sweep line touches a_1.



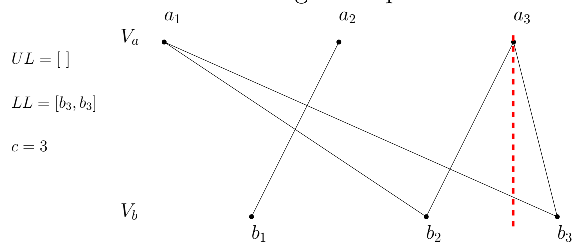
(b) a_2 is added to UL, as the sweep line touches b_1.



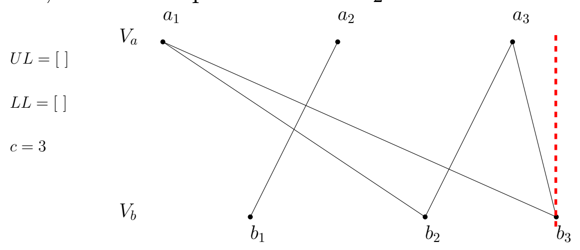
(c) As the sweep line touches a_2, a_2 is removed from UL and the crossings are updated to $c = 2$.



(d) a_3 is added to UL and b_2 is removed from LL, as the sweep line touches b_2.



(e) As the sweep line touches a_3, a_3 is removed from UL and the crossings are updated to $c = 3$.



(f) b_3 is removed from LL as the sweep line touches b_3.

Figure 4: An example of the sweep line algorithm for the graph G in fig. 3.

vertices are inserted into UL or LL $|E|$ times, as the lists represent active edges and no edge is active twice. Resulting in, lines 12-15 and 22-25 being performed O($|E|$) times. Line 16 is performed when there is a crossing of situation 2. Since there is at least one other element in UL or LL (because of line 9), we know that k_2 will be added to k_3, which in turn is added to the total number of crossings c. This ensures that line 16 is performed at most c times. The algorithm therefore has a time complexity of O($|V_a| + |V_b| + |E| + c$), where c is the number of crossings [6].

4 Assignment Heuristic

The assignment heuristic is based on the assignment problem. The assignment problem consists of determining the most cost efficient way to assign the ith worker to the jth job, based on a cost matrix c, which is of size $m \times m$. Catarci explains that the assignment problem can be solved in $O(m^3)$, using the Hungarian reduction algorithm [7], which will be discussed in section 4.2. Formulating edge crossing minimization as an assignment problem will be explained in section 4.1. If we can formulate the edge crossing minimization as an assignment problem, we can then minimize the total cost using the Hungarian reduction algorithm, and thus minimize the number of edge crossings.

4.1 Edge crossing minimization as an assignment problem

If we consider a bipartite graph, $G = (V_a \cup V_B,\ E)$, where $|V_a| = n$, $|V_b| = m$, and V_a is the fixed layer, and let A be the $n \times m$ adjacency matrix for G. The value a_{ij} will be 1 if there is an edge connecting vertex i and j, and 0 otherwise. Finally, we also consider a four-dimensional matrix B with size $m \times n \times m \times n$, where B_{abcd} is 1 if $(c > a$ **and** $d < b)$ **or** $(c < a$ **and** $d > b)$, and 0 otherwise, where $a, c \in V_a$ and $b, d \in V_b$. It is then clear that B_{abcd} will be 1 if there is a crossing between the edges ab and cd [7]. It is clear to see that the construction of B is unnecessary, as we can compute the value of each element in $O(1)$ time, and thus by not creating the matrix, but computing each element directly we will save $O(m^2n^2)$ space. However, it serves as a great illustration of explaining the initialization of the B matrix. B represents all possible edge crossings of a complete bipartite graph, where a bipartite is complete if each node $v \in V_a$ has m edges, one edge (v, w) for each $w \in V_b$. We can therefore construct the C matrix of size $m \times m$, as follows [7]:

$$c_{ij} = \sum_{h=1}^{n} \left[a_{hi} \cdot \left(\sum_{c=1}^{m} \sum_{d=1}^{n} B_{jhcd} \right) \right] \tag{1}$$

Each value $\in C$ is computed by assuming the rest of the G is complete, and thus c_{ij} represents an upper bound for the number of edge crossings it would produce to have vertex i in position j. Thus the permutation π_b computed by the assignment heuristic is not necessarily the optimal permutation with respect to minimizing edge crossings. However, the idea of the assignment heuristic is that the potential error caused by the assumption when computing c_{ij}, is compensated by minimizing the total cost and thus the number of crossings. This minimization allows for a mutual interaction of all vertices in V_b, to be considered, compared to the median and barycenter heuristic where their interaction is not considered [7]. An example of the construction of the C matrix and the cost minimization can be seen in section 4.3.

4.2 Hungarian reduction algorithm

The Hungarian reduction algorithm takes a non negative cost matrix, C, ($n \times n$, where n is the number of jobs and workers), and consists of six steps. The steps in the Hungarian reduction algorithm are described by [8] as:

Step 1: For each row, subtract the minimum element from each element in the row.

Step 2: For each row in the reduced matrix, find a zero, z. If there is no starred zero in its row or column star z.

Step 3: Cover each column containing a starred z. If n columns are covered, the starred zeros represent the optimal assignment, and the algorithm terminates. Otherwise, we proceed to step 4.

Step 4: Find a non covered z and prime it. If there is no starred zero in the row containing this primed z, continue to step 5. Otherwise, cover the row and uncover the column containing the starred zero. Continue this until there are no uncovered zeros left. Let the smallest uncovered value be s and continue to Step 6.

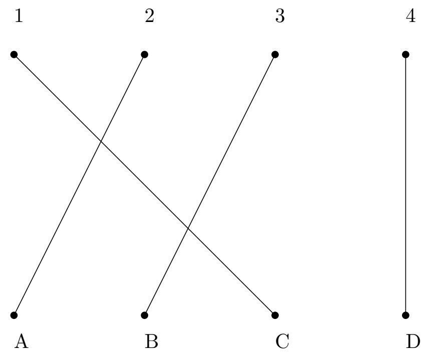
Step 5: Create a series of alternating primed and starred zeros as follows. Let z_0 be the uncovered primed zero found in step 4. Let z_1 be the starred zero in the column of z_0 if one exists. Let z_2 be the primed zero in the row of z_1. Continue this until the series terminates at a primed zero that has no starred zeros in its column. Then un-star each starred zero in the series and star each primed zero in the series. Delete all primes and uncover the matrix, and go to step 3.

Step 6: Add s to every non-zero element of each covered row, and subtract it from every non-zero element of each uncovered column. Then go to step 4.

Once the Hungarian reduction algorithm terminates, the assignments are indicated by the positions of the starred zeros in C. If c_{ij} is a starred zero, then the element represented by row i is assigned to the element represented by column j [8].

4.3 Example

If we consider the bipartite graph G shown in fig. 5a, where $V_a = \{1, 2, 3, 4\}$ and $V_b = \{A, B, C, D\}$ and keeping V_a fixed. We also consider the adjacency matrix A shown in fig. 5b.



(a) A bipartite graph G, where $\{1, 2, 3, 4\} = V_a$ and $\{A, B, C, D\} = V_b$. This graph has two edge crossings.

	A	B	C	D
1	0	1	0	0
2	0	0	1	0
3	0	1	1	1
4	1	0	1	0

(b) The adjacency matrix A for G

Figure 5: A bipartite graph G shown in (a), and its adjencency matrix A shown in (b).

We can then construct the cost matrix C shown in fig. 1, by using the formula in (1), where $n = |V_a| = 4$ and $m = |V_b| = 4$. To calculate $c_{A1} = c_{11}$:

$$c_{11} = \sum_{h=1}^{4}\left[a_{h1} \cdot \left(\sum_{c=1}^{4}\sum_{d=1}^{4} B_{1hcd}\right)\right]$$
$$c_{11} = 3$$

	1	2	3	4
A	3	4	5	6
B	6	5	4	3
C	0	3	6	9
D	9	6	3	0

Table 1: The cost matrix C for G (shown in fig. 5a)

As the initial cost matrix C has now been constructed, we can then use the Hungarian reduction algorithm to determine where to place each vertex:

$$\begin{bmatrix} 3-3 & 4-3 & 5-3 & 6-3 \\ 6-3 & 5-3 & 4-3 & 3-3 \\ 0 & 3 & 6 & 9 \\ 9 & 6 & 3 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 3 & 2 & 1 & 0 \\ 0 & 3 & 6 & 9 \\ 9 & 6 & 3 & 0 \end{bmatrix}$$

Step 1: row reduction.

$$\begin{bmatrix} 0^* & 1 & 2 & 3 \\ 3 & 2 & 1 & 0^* \\ 0 & 3 & 6 & 9 \\ 9 & 6 & 3 & 0 \end{bmatrix}$$

Step 2: each zero that does not contain another zero in its row or column has been marked with a star, and we continue to step 3.

$$\begin{bmatrix} 0^* & 1 & 2 & 3 \\ 3 & 2 & 1 & 0^* \\ 0 & 3 & 6 & 9 \\ 9 & 6 & 3 & 0 \end{bmatrix}$$

Step 3: We cover each column with a starred zero with a line and then since $|lines| \neq n = 4$ we must continue to step 4

$$\begin{bmatrix} 0^* & {\color{red}1} & 2 & 3 \\ 3 & 2 & 1 & 0^* \\ 0 & 3 & 6 & 9 \\ 9 & 6 & 3 & 0 \end{bmatrix}$$

Step 4: Since there is no uncovered zeros in the matrix, we find s, the smallest uncovered value, which is marked with red and continue to step 6.

$$\begin{bmatrix} 0^* & 0 & 1 & 3 \\ 3 & 1 & 0 & 0^* \\ 0 & 2 & 6 & 9 \\ 9 & 5 & 2 & 0 \end{bmatrix}$$

Step 6: we subtract s from each uncovered column, since they are no covered rows we then continue to step 4.

$$\begin{bmatrix} 0^* & 0' & 1 & 3 \\ 3 & 1 & 0' & 0^* \\ 0' & 2 & 6 & 9 \\ 9 & 5 & 2 & 0 \end{bmatrix}$$

Step 4: we prime the uncovered zero in row 1 and uncover column 1 and cover row 1. Then we prime the zero in row 2 and uncover column 4 and cover row 2. Then we prime the zero in row 3 and continue to step 5.

$$\begin{bmatrix} {\color{red}0} & {\color{red}0^*} & 1 & 3 \\ 3 & 1 & 0' & 0^* \\ {\color{red}0^*} & 2 & 6 & 9 \\ 9 & 5 & 2 & 0 \end{bmatrix}$$

Step 5: We then construct the series of alternating primed zeros and starred zeros, we marked it in red. Then we star the primed zeros in the series, and unstar the starred zeros, before removing every line and prime and returning to step 3.

$$\begin{bmatrix} 0 & 0^* & 1 & 3 \\ 3 & 1 & 0 & 0^* \\ 0^* & 2 & 6 & 9 \\ 9 & 5 & 2 & 0 \end{bmatrix}$$

Step 3: We cover each column of the matrix containing a stared zero. Since $|lines| \neq n = 4$ we must continue to step 4.

$$\begin{bmatrix} 0 & 0^* & 1 & 3 \\ 3 & 1 & 0' & 0^* \\ 0^* & 2 & 6 & 9 \\ 9 & 5 & 2 & 0' \end{bmatrix}$$

Step 4: we prime the uncovered zero in row 2, and uncover column 4 and cover row 2. Then we prime the zero in row 4 and continue to step 5.

$$\begin{bmatrix} 0 & 0^* & 1 & 3 \\ 3 & 1 & \color{red}{0^*} & \color{red}{0} \\ 0^* & 2 & 6 & 9 \\ 9 & 5 & 2 & \color{red}{0^*} \end{bmatrix}$$

Step 5: We then construct the series of alternating primed zeros and starred zeros, we marked it in red. Then we star the primed zeros in the series, and unstar the starred zeros, before removing every line and prime and returning to step 3.

$$\begin{bmatrix} 0 & 0^* & 1 & 3 \\ 3 & 1 & 0^* & 0 \\ 0^* & 2 & 6 & 9 \\ 9 & 5 & 2 & 0^* \end{bmatrix}$$

Step 3: since $n = |lines|$, the algorithm terminates and the starred zero represents an optimal assignment.

After having done the hungarian reduction, we get an optimal assignment of the vertices in V_b to be $\{C, A, B, D\}$, and the resulting graph can be seen in figure 6.

4.4 Zero crossing proof

Catarci claims that the assignment heuristic always finds a solution with zero crossings if one exists [7]. However, if we consider the graph G and its C matrix shown in fig. 7, we can show that it will not find the solution of zero crossings, like in [9].

If we then perform the hungarian reduction, which was explained in section 4.2, we get:

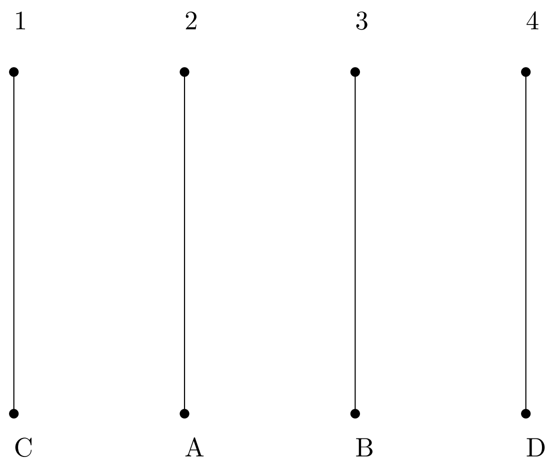
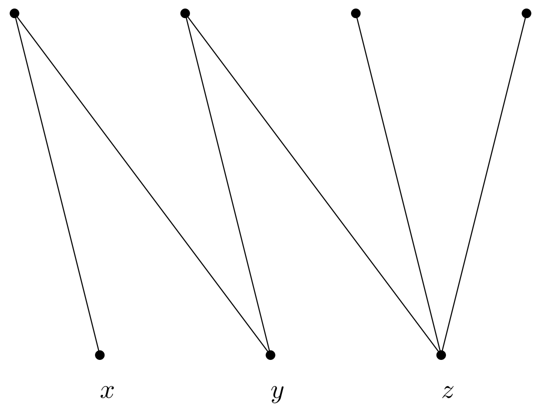


Figure 6: The resulting graph G' with a permutation of V_b that has 0 edge crossings.



$$\begin{array}{c|ccc} \text{x} & 0 & 3 & 6 \\ \text{y} & 2 & 6 & 10 \\ \text{z} & 12 & 9 & 6 \end{array}$$

Figure 7: A graph G and its C matrix from [9]

Step 1

$$\begin{array}{c|ccc} x & 0-0 & 3-0 & 6-0 \\ y & 2-2 & 6-2 & 10-2 \\ z & 12-6 & 9-6 & 6-6 \end{array} =$$

Step 2

$$\begin{array}{c|ccc} x & 0 & 0^* & 6 \\ z & 0^* & 1 & 8 \\ y & 6 & 0 & 0^* \end{array}$$

Since we can now cover all the starred zeros with 3 lines which corresponds to the number of columns, we are done reducing. Looking at the assignment heuristic will produce the permutation $\pi_{V_b} = \{z, x, y\}$. This means that the assignment heuristic would swap x and y, and thus would create a crossing. Therefore the assignment heuristic is not always able to find solutions with zero crossings if one exists.

5 Barycenter Heuristic

A common heuristic to reduce the number of crossings in a bipartite graph is the barycenter heuristic. The method will choose the x-coordinate of each vertex in the V_b of the bipartite graph, as the average of the x-coordinates of its neighbors. The average x-coordinate is referred to as the barycenter value. [2].

5.1 Performance ratio

We will examine the lower bound of the barycenter heuristic's performance ratio, as the upper bound of the barycenter heuristic is still unknown [7]. Considering the graph shown in fig. 8, which shows the worst case for the barycenter heuristic [10], where the $k+m$ nodes in V_a are labelled the same as their x-coordinates. The performance ratio is computed as $\frac{bc(G,\pi_{V_b})}{opt(G,\pi_{V_b})}$.

As explained, the x-coordinates for the vertices $\in V_b$ are determined by their barycenter values. Letting $bc(x, \pi_{V_b})$ be the barycenter value of the x vertex based on its neighbors in π_{V_b}. Assuming $bc(u, \pi_{V_b}) \leq k-1 = bc(v_1, \pi_{V_b})$ the barycenter heuristic would then produce $k-1+sm$ crossings. As $e(u,0)$ will cross edges $e(t,1)$, $e(t,2)\dots e(t,k-2)$ and $e(t,k-1)$ thus resulting in $k-1$ crossings. Additionally the edges $e(u,k)$, $e(u,k+1)\dots e(u,k+m-1)$ would cross the edges $e(v_1,k-1)$, $e(v_2,k-1)\dots e(v_s,k-1)$ resulting in m edges crossing s edges this produces sm crossings. Thus in the worst case, the barycenter heuristic would produce $k-1+sm$ crossings.

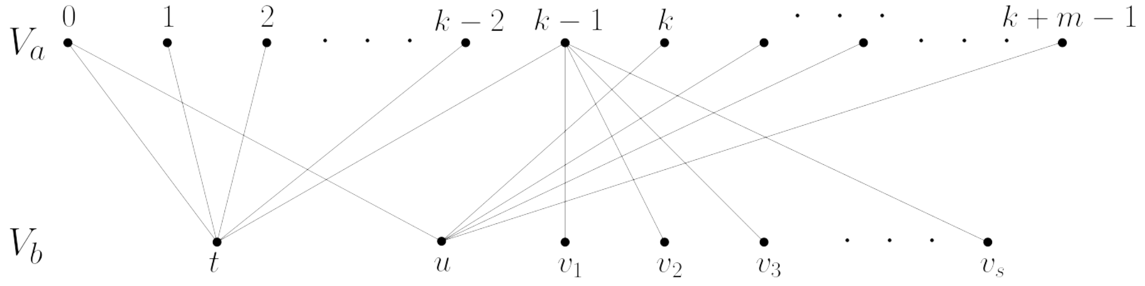


Figure 8: The worst case for barycenter taken from [10]

The best case for the barycenter heuristic shown in figure 9, the optimal solution is achieved by moving u to the right of v_s [10]. This results in that the only edge originating from u that crosses another edge is $e(u,0)$. $e(u,0)$ will cross the edges $e(v_1,k-1)$, $e(v_2,k-1)\dots e(v_s,k-1)$ and $e(t,1)$, $e(t,2)\dots e(t,k-2)$ and $e(t,k-1)$. Thus resulting $k-1+s$ crossings.

We can then calculate the performance ratio by $bc(G,\pi_{V_b})/opt(G,\pi_{V_b})$, where $bc(G,\pi_{V_b}) = k-1+sm$

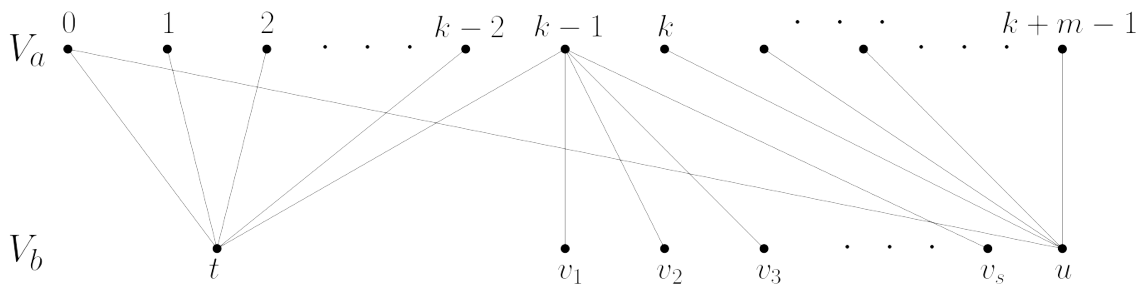


Figure 9: The best case for barycenter taken from [10]

and $opt(G, \pi_{V_b}) = k - 1 + s$. If we let $s = k - 1$, the performance ratio will be:

$$\begin{aligned}\frac{bc(G, \pi_{V_b})}{opt(G, \pi_{V_b})} &= \frac{k-1+(k-1)m}{k-1+(k-1)} \\ &= \frac{(k-1)(1+m)}{2(k-1)} \\ &= \frac{1+m}{2}\end{aligned}$$

m is constrained by the earlier assumption that: $bc(u, \pi_{V_b}) \leq k-1 = bc(v_1, \pi_{V_b})$, this means that we can calculate the barycenter value of u by:

$$\begin{aligned}(0 + \sum_{i=k}^{k+m-1} i)/(m+1) &\leq k-1 \\ \frac{(k+m-1+k)(k+m-1-k+1)}{2} \cdot \frac{1}{m+1} &\leq k-1 \\ \frac{(2k+m-1)(m)}{2(m+1)} &\leq k-1 \\ \frac{2km+m^2-m}{2(m+1)} &\leq k-1 \\ 2km+m^2-m &\leq (k-1)\cdot(2m+2) \\ 2km+m^2-m &\leq 2km+2k-2m-2 \\ m^2+m &\leq 2k-2\end{aligned}$$

This means we are able to conclude that m is $\Omega(\sqrt{k})$. Remembering that $n = k + m$ and that the ratio is $(m + 1)/2$. As k is a part of $n = k + m$, this means that the growth of n is at least as fast as $\sqrt{k}$, meaning that the ratio is $\Omega(\sqrt{n})$. $\square$

5.2 Zero Crossing Proof

Lemma 1. *If a bipartite graph G has a permutation π_{V_b} that produces zero edge crossings. Then the barycenter heuristic will find it.*

Proof. If we consider π_{V_b} to be the permutation of V_b shown in fig. 10, we must recall that the barycenter heuristic chooses the x-coordinate of the nodes in V_b by computing the average of the x-coordinates of its neighbors in V_a.

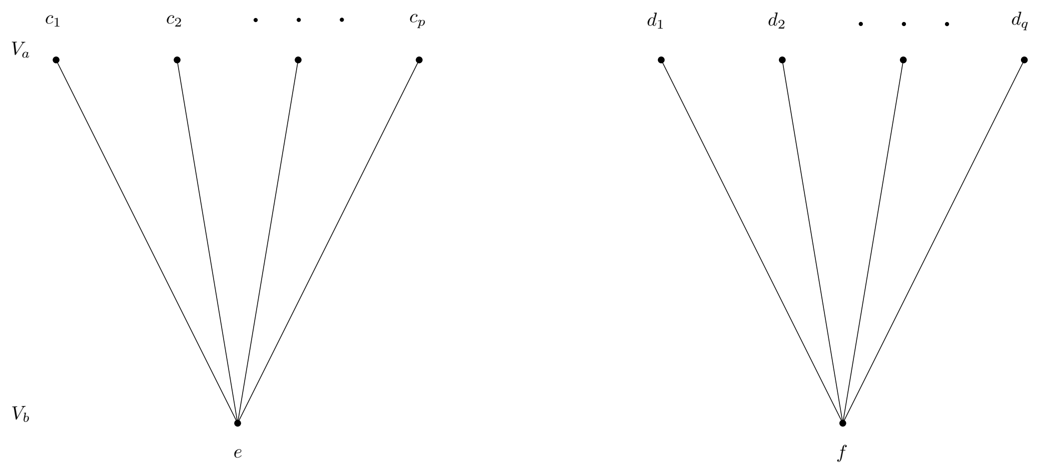


Figure 10: A simple graph, G, where the barycenter heuristic can find a correct solution with zero crossings.

Assuming that $c_i \leq d_j$ we want to prove that the barycenter value of $e \leq f$. This means that:

$$\frac{\sum_{i=1}^{p} c_i}{p} \leq \frac{\sum_{j=1}^{p} d_1}{p}$$

We can then reduce the right side of $\leq$ using the summation rules, as follows:

$$\begin{aligned}\frac{\sum_{j=1}^{p} d_1}{p} &= \frac{p \cdot d_1}{p} \\ &= d_1 \\ &= \frac{q \cdot d_1}{q} \\ &\leq \frac{\sum_{j=1}^{q} d_j}{q}\end{aligned}$$

This means that:

$$\frac{\sum_{i=1}^{p} c_i}{p} \leq \frac{\sum_{j=1}^{q} d_j}{q}$$

Thus we have proved that the barycenter value of $e \leq f$, meaning that e will always be placed to the left of f.

Furthermore, we can consider the permutation of G' shown in fig. 11, where the vertices s and u both have an edge to d. If we assume $c_i \leq d \leq e_j$, we can then prove that the barycenter heuristic will always produce the permutation shown in fig 11 and the barycenter values for s, t_i, and u have

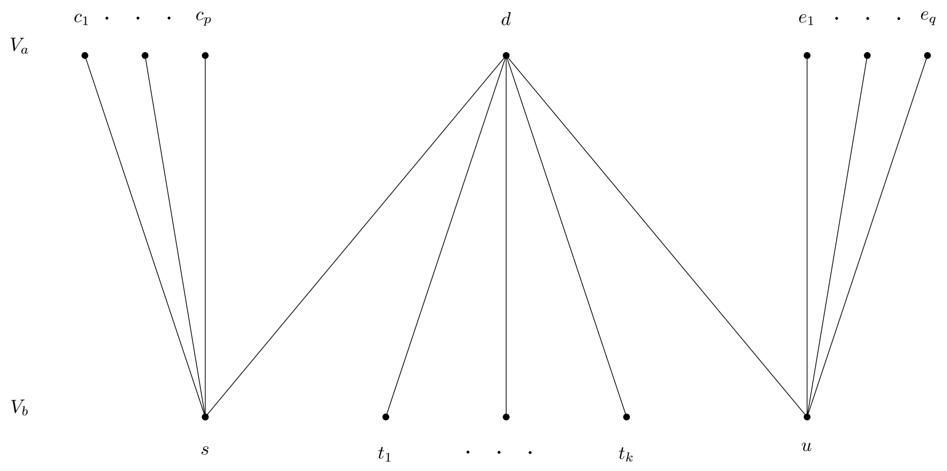


Figure 11: A permutation of graph G', where the barycenter heuristic has computed a solution with 0 crossings.

a relation as follows: $s \leq t_i \leq u$. We assumed that $c_i \leq d$, using this assumption we get that:

$$\frac{\sum_{i=1}^{p} c_i}{p} \leq \frac{\sum_{i=1}^{p} d}{p}$$

$$\sum_{i=1}^{p} c_i \leq \sum_{i=1}^{p} d$$

$$\sum_{i=1}^{p} c_i \leq p \cdot d$$

We then add d to both sides

$$(\sum_{i=1}^{p} c_i) + d \leq p \cdot d + d$$

$$(\sum_{i=1}^{p} c_i) + d \leq (p+1)d$$

$$\frac{(\sum_{i=1}^{p} c_i) + d}{p+1} \leq d$$

Thus our assumption that $c_i \leq d$ implies that the barycenter value of s is less than or equal to the barycenter value of t_i, where $1 \leq i \leq k$. For the second part of the relation, $d \leq e_1$, we can make a similar argument:

$$\frac{\sum_{i=1}^{q} d}{q} \leq \frac{\sum_{i=1}^{q} e_i}{q}$$

We then perform the same operations as above and we get that:

$$d \leq \frac{(\sum_{i=1}^{q} e_i) + d}{q+1}$$

Thus our assumption that $d \leq e_i$ implies that the barycenter value of t_i, where $1 \leq i \leq k$, is less than or equal to the barycenter value of u. Therefore we have proved that the barycenter values (here denoted as bc) will always be $bc(s) \leq bc(t_i) \leq bc(u)$. This proves that the barycenter heuristic will always compute a π_{V_b} that has zero crossings if such a permutation exists. $\square$

6 Median Heuristic

Another common approach to the one-sided crossing minimization problem in a bipartite graph is the median heuristic. Similar to the barycenter heuristic, in the median heuristic an x-coordinate is chosen for each vertex $v \in V_B$ to be a median of the x-coordinates of its neighbors. Let N_v be the neighboring vertices of v and if $N_v = w_1, w_2, w_3, \ldots, w_i$ with $x_0(w_1) \leq x_0(w_2) \leq x_0(w_3) \leq \cdots \leq x_0(w_i)$, then we define $med(v) = x_0(w_{\lfloor i/2 \rfloor})$. If N_v is empty, then $med(v) = 0$. It separates two vertices with the same median, by the degree of the vertices. If one vertex has an odd degree and the other even, the odd degree vertex will have a lower x-value than the even degree vertex [2].

6.1 Performance Ratio

6.1.1 Lower bound

If we consider the worst case for the median heuristic, shown in fig. 12. In this graph $n = 2k + 2(k+1) = 4k + 2$ We can use this graph to calculate a lower bound for the median, by using the performance ratio, $\frac{median(G,\pi_{V_b})}{opt(G,\pi_{V_b})}$. In the graph shown there is $k^2 + 2k(k+1)$ crossings. The optimal solution appears when u and v switch places. This switch would result in $(k+1)^2$ crossings.

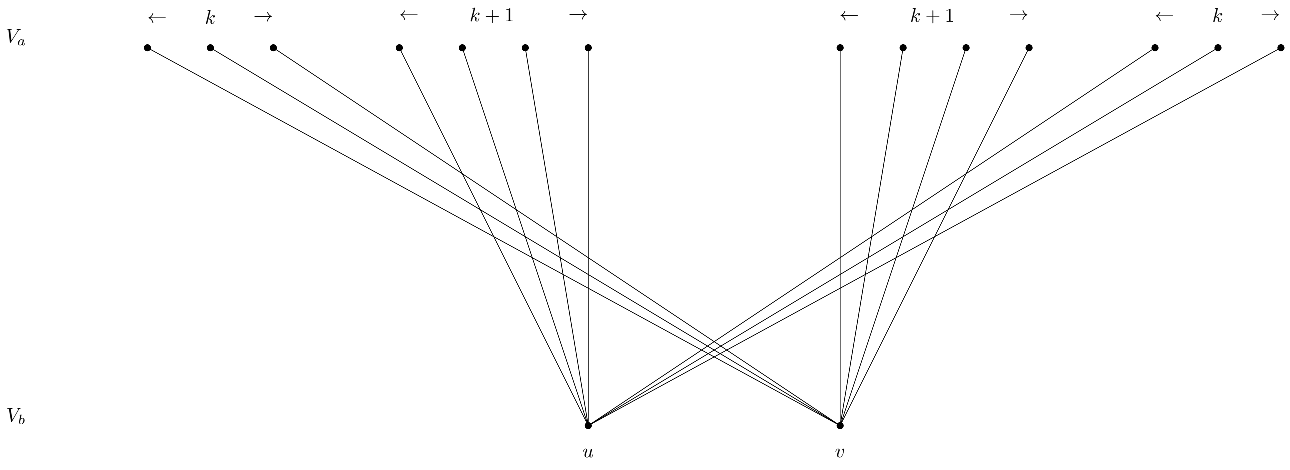

V_a
$\leftarrow$ k $\rightarrow$
$\leftarrow$ $k+1$ $\rightarrow$
$\leftarrow$ $k+1$ $\rightarrow$
$\leftarrow$ k $\rightarrow$
V_b
u
v

Figure 12: Worst case for the median heuristic. Taken from [11]

We can therefore calculate the performance ratio as follows:

$$\frac{median(G, \pi_b)}{opt(G, \pi_b)} = \frac{k^2 + 2k(k+1)}{(k+1)^2} \tag{2}$$

$$= \frac{3k^2 + 2k}{k^2 + 2k + 1} \tag{3}$$

Since k is O(n), and when n gets sufficiently large, we can substitute k with n.

$$\begin{aligned}\frac{median(G,\pi_b)}{opt(G,\pi_b)} &\geq \frac{3n^2+2n}{n^2+2n+1} \\ &\geq \frac{n(3n+2)}{n(n+2)+1} \\ &\geq \frac{3n+2}{n+3}\end{aligned}$$

This can be simplified to $3 - O(\frac{1}{n})$ as follows

$$\begin{aligned}\frac{3n+2}{n+3} &\geq 3 - \frac{c}{n} \implies \\ \frac{3n^2+2n}{n+3} &\geq 3n - c \implies \\ 3n^2+2n &\geq 3n^2 - cn + 9n - 3c\end{aligned}$$

This holds for any $c \geq 8$, and therefore, we have a lower bound as $\frac{median(G,\pi_b)}{opt(G,\pi_b)} \geq 3 - O(\frac{1}{n})$.

6.1.2 Upper bound

We now examine the upper bound of the median heuristic, where we show that the upper bound matches the lower bound. To do so, we need to use lemma 2, which is proven in [11].

Lemma 2 (From [11, p. 282]). *If $G = (V_a \cup V_b, E)$ is a two-layer bipartite graph, and π_a and π_b are orderings of V_a and V_b respectively, then*

$$cross(G,\pi_a,\pi_b) = \sum_{x_b(u)<x_b(v)} c_{uv} \tag{4}$$

Further

$$opt(G,\pi_a) \geq \sum_{u,v} min(c_{uv}, c_{vu}) \tag{5}$$

where the sum is over all unordered pairs u, v of vertices of the top later.

If we suppose u and v to be vertices $\in V_B$, and that the median heuristic has placed u to the left of v, we can then divide the edges into 4 groups as follows, this is also shown for a graph in fig. 13.

$$\begin{aligned}\alpha &= \{(u,w) \in E : x_a(w) \ < med(u)\} \\ \beta &= \{(v,w) \in E : x_a(w) > med(v)\} \\ \gamma &= \{(v,w) \in E : x_a(w) < med(v)\} \\ \delta &= \{(u,w) \in E : x_a(w) \ > med(u)\}\end{aligned}$$

u and v are both joined to a vertex at their median values, denoted by e_u and e_v.

We can then claim that:

$$c_{vu} \geq ab + a + b + \epsilon \tag{6}$$

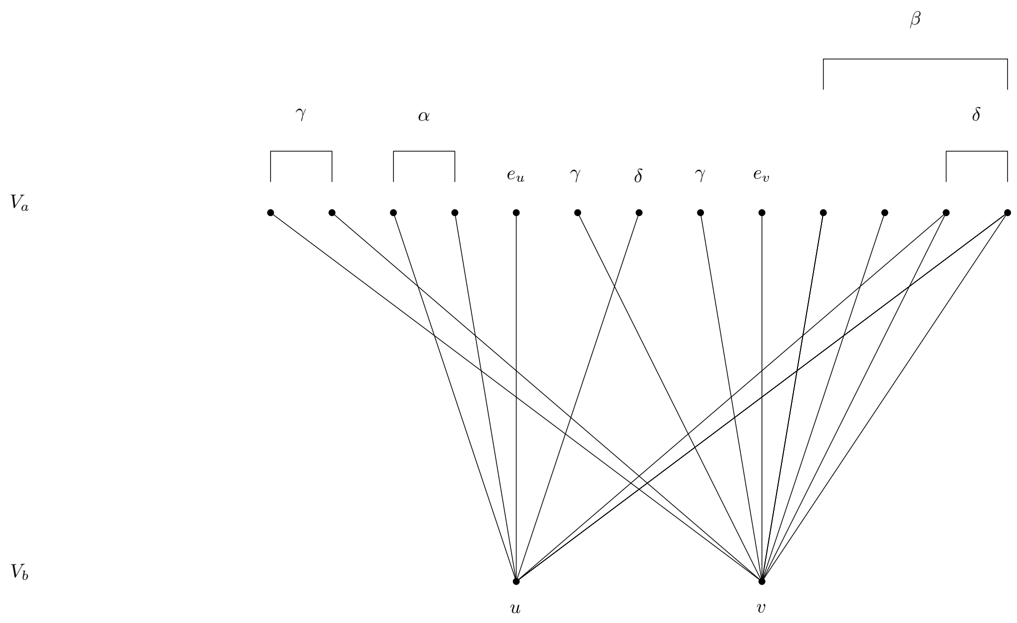


Figure 13: The grouping of edges shown. Taken from [2].

where, $a = |\alpha|$, $b = |\beta|$, $c = |\gamma|$, $d = |\delta|$, $\epsilon = 0$ if $med(u) = med(v)$ and 1 otherwise. c_{vu} is an entry in the crossing array, where the uvth entry contains the number of crossings between edges containing u and edges containing v when $x_a(u) < x_a(v)$, or $c_{vu} = |\{\{ut, vw\} \subseteq E : x_a(t) > x_a(w)\}|$ [2]. To prove this, suppose we switched u's and v's place in fig. 13. Then all edges in α, would cross all edges in β and e_v, this would result in $a \cdot b + a$ crossings. Furthermore all edges in β would cross e_u, resulting in b crossings. Finally, the edges e_v and e_u would not cross if $med(u) = med(v)$, otherwise the edges would cross. This case is expressed by the ϵ variable. Therefore the lower bound (6) follows.

We can also claim that:

$$c_{uv} \leq ac + cd + bd + c + d \tag{7}$$

Suppose u is placed to the left of v, as shown in fig. 13 then α cannot cross β, and e_u and e_v cannot cross either. This placement of u and v allows for the possibility of all edges in α crossing all edges in γ, resulting in $a \cdot c$ crossings. It also allows for all edges in δ to cross all edges in γ, which could result in a maximum of $c \cdot d$ crossings. Furthermore, it also allows for all edges in β and δ to cross resulting in $b \cdot d$ crossings. Finally, it also allows for the possibility that all edges in γ cross e_u, and that all edges in δ cross e_v, resulting in $c + d$ crossings.

The values a and d are closely related. If the degree of u is odd then $a = d$, as there will be the same number of edges on either side of u. However, if the degree of u is even then $a + 1 = d$, as there then would be an additional edge on the right side of u, per the definition of the median heuristic. This can be done similarly for c and b, where if the degree of v is even then $c + 1 = b$ and otherwise $c = b$. This means that in any case $d \leq a + 1$ and $c \leq b$ holds. We can now use these inequalities in (7) to allow us to deduce that:

$$c_{uv} \leq 3ab + 3b + a + 1 \tag{8}$$

Now we will show that $c_{uv} \leq 3c_{vu}$ to prove that $median(G, \pi_b) \leq 3 \cdot Optimal(G, \pi_b)$. We will do so by a proof by contradiction, such that $c_{uv} > 3 \cdot c_{vu}$, meaning that:

$$c_{uv} - 3 \cdot c_{vu} > 0 \tag{9}$$

This allows us to deduce:

$$\begin{aligned}(3ab + 3b + a + 1) - 3 \cdot (ab + a + b + \epsilon) &> 0\\ -2a - 3\epsilon + 1 &> 0\end{aligned}$$

This must therefore mean that

$$0 > 2a + 3\epsilon - 1$$

Since we know that a and ϵ per definition cannot be negative numbers, we can therefore imply that $a = \epsilon = 0$. Which in return implies that $\alpha = \emptyset$. Since $\alpha = \emptyset$ we know that the degree of u is a maximum of two, as it would mean that u only has edges that end in w, where $u.x < w.x \lor u.x = w.x$. For that to not conflict with the definition of the median heuristic u cannot have a degree higher than 2. We can show that (9) is impossible when the degree of u is at most one, as $\alpha = \delta = \emptyset$. Furthermore, if the degree of u is at most one it would mean that $a = d$. Thus:

$$\begin{aligned}(0)c + c(0) + b(0) + c + 0 - 3(0)b + 0 + b + \epsilon) &> 0\\ c - 3b &> 0\end{aligned}$$

Since we know that $c \leq b$, this inequality will never be true and it is therefore impossible for the degree of u to be one. We therefore know that the degree of u must be two, thus $d = 1$. If we recall that $a = \epsilon = 0$, we know that the $median(u) = median(v)$ and the degree of u must therefore be even and the degree of v odd, placing v to the left of u. This means that we can come to the conclusion that the degree of v is even. This means that $c = b - 1$. We can then use $a = 0$, $c = b - 1$, $d = 1$ and $\epsilon = 0$ in (6) and (7) to compute:

$$\begin{aligned}&c_{vu} \geq ab + a + b + \epsilon\\ &c_{vu} \geq 0b + 0 + b + 0\\ &c_{vu} \geq b\\ &c_{vu} = b\end{aligned}$$

$$\begin{aligned}&c_{uv} \leq ac + cd + bd + c + d\\ &c_{uv} \leq (0)(b-1) + (b-1)1 + b(1) + (b-1) + 1\\ &c_{uv} \leq 3b - 1\\ &c_{uv} = 3b - 1\end{aligned}$$

This contradicts our assumption (9), as it states $-1 > 0$, therefore $c_{uv} \leq 3 \cdot c_{vu}$. This implies that $c_{uv} \leq 3min(c_{uv}, c_{vu})$. By summing up the inequality, and then using both parts of lemma 2, we conclude that $cross(G, \pi_a, \pi_b) \leq 3 \cdot opt(G, \pi_a)$. $\square$

6.2 Zero Crossing Proof

Lemma 3. *If a bipartite graph G has a permutation π_{V_b} that produces zero edge crossings. Then the median heuristic will find it.*

Proof. Assuming that G is a bipartite graph, where a permutation π_{V_b} exists such that it produces no edge crossings, such as the graph is shown in fig 10. Recall that the Median heuristic orders the vertices in V_b by computing the median of its neighbors in V_a. We can then prove that the median heuristic will find a solution with 0 edge crossings, by assuming that $c_i \leq d_j$, it then follows that median($\{c_1 \dots c_p\}$) $\leq$ median($\{d_1 \dots d_q\}$). This means that the median heuristic will always put vertex e before vertex f, thus it can find the solution with zero crossings.

If we now consider the permutation shown in Fig. 11 and assume that the vertices in v_a have the following relation: $c_i \leq d \leq e_j$. It would then follow that vertex, s has a median($\{c_1 \dots c_p\} \cup \{d\}$) (denoted by med_s), t_m has median($\{d\}$) (denoted by med_t), and u has median($\{d\} \cup \{c_1 \dots c_q\}$) (denoted by med_u). Based on our assumption of the relation between the vertices in V_a, it would follow that the relation for their median values would be: $med_s \leq med_t \leq med_u$. This means that the median heuristic will always put vertex s before t, and t before u. Thus the median heuristic will always compute a permutation that produces zero crossings if one exists. □

7 Other Heuristics

In this section, we will discuss the development of the Parent heuristic algorithm, and the bogo heuristic algorithm for minimizing edge crossings. The parent heuristic stems from a suggestion by our thesis supervisor, Mikkel Vind Abrahamsen, and the bogo heuristic is inspired by the Bogo Sort algorithm (See [4]), and it was used mainly as a control in our tests, which will be described in section 8.

7.1 Parent heuristic

The parent heuristic algorithm creates a new ordering of the vertices in V_b based on V_a. The pseudo-code for the parent heuristic algorithm can be seen in alg. 2. For every $v \in V_a$ it will generate a list of v's neighboring vertices and sort them according to their degrees. Once sorted it will then append each of the neighboring vertices to π_{V_b} if they have not been appended previously. Checking if a vertex has been previously appended, on line 9 in alg. 2, can be implemented by a hashmap and can be done in constant time. In our implementation of the parent heuristic, we used a hashmap to map each vertex in V_b to a boolean value representing if the vertex had been previously appended or not. This allowed us to reduce the time complexity to $O(a \cdot e \log e)$, where $a = |V_a|$, and $e = |E|$.

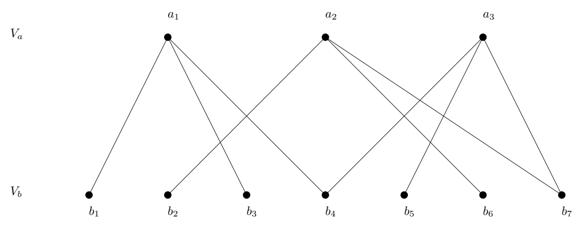
7.1.1 Example

If we were to apply the parent algorithm to the graph seen in figure 14a, the algorithm would first look at the vertices in V_b that are neighboring a_1, which are $\{b_1, b_3, b_4\}$. These vertices are then sorted based on their degree in ascending order, which results in the ordering $\{b_1, b_3, b_4\}$, and appended to π_{V_b} accordingly. Next, we look at a_2 where we repeat process, resulting in the ordering $\{b_1, b_3, b_4, b_2, b_6, b_7\}$. Lastly, when looking at the vertices neighboring a_3, we notice that some of them are also neighboring a_1 and a_2, meaning that the algorithm skips those vertices since they have already been added to the new π_{V_b}. Finally, we are left with one vertex b_5 that is neighboring

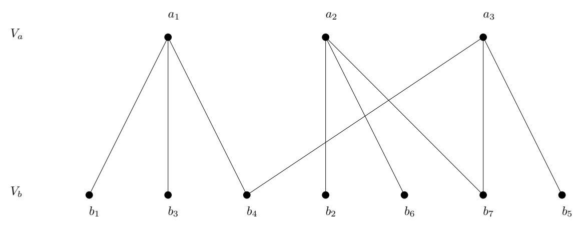
Algorithm 2 Parent heuristic

```
1: procedure Parent($G$)
2:     Input: A bipartite graph $G = (V_a \cup V_b,\ E)$
3:     Output: A bipartite graph $G = (V_a \cup \pi_{V_b},\ E)$
4:     let $\pi_{V_b}$ be an empty list
5:     for all $v \in V_a$ do
6:         let $N_v$ be a list of each of $v$'s neighbouring vertices
7:         Sort $N_v$ in ascending order according to their degrees.
8:         for all $u \in N_v$ do
9:             if $u$ is not in $\pi_{V_b}$ then
10:                Append $u$ to $\pi_{V_b}$
11:            end if
12:        end for
13:    end for
14:    return $(V_a \cup \pi_{V_b},\ E)$
15: end procedure
```

a_3, and since it has not been added to π_{V_b} yet, the algorithm sorts and adds it to π_{V_b}, resulting in the ordering $\{b_1, b_3, b_4, b_2, b_6, b_7, b_5\}$. The result of the parent heuristic can be seen in figure 14b.



(a) Before the parent heuristic



(b) After the parent heuristic

Figure 14: A graph before and after the parent heuristic is applied. After the parent heuristic the crossings in the original graph are reduced from 6 crossing to 3 crossings.

7.2 Bogo heuristic

The Bogo heuristic choses a random permutation of V_b by iterating over the array and generating a random index for each vertex $v \in V_b$. It therefore has a time complexity of $O(n)$, where $n = |V_b|$. There is therefore no performance bounds for it, and it is not useful in minimizing edge crossings. However, it serves as a control indicator in our tests. This means, that we are then able to ensure that the other mentioned heuristics, actually perform better than picking a random permutation of V_b.

8 Tests

The primary objective of the testing phase is to determine the most effective tie-breaking method for the median and barycenter heuristics in minimizing edge crossings and trying to determine conditions where one heuristic is better. We will discuss the testing process and the results in this section.

8.1 Test Graphs

We used the graphs provided by PACE [1] to perform our tests. We have tested on three sets of graphs: `tiny test set`, `medium test set`, `public test set`. The `tiny test set` contains tiny graphs, where the number of vertices in a graph varies from 8 to 20, the density varies from 0.12 to 1 and the number of edges varies from 4 to 20. The `medium test set` contains medium graphs, where the number of vertices in a graph varies from 32 to 1076, the density varies from 0.0046 to 0.31 and the number of edges varies from 48 to 1649. The `public test set` is representative of the graphs our solution will be tested on in the PACE 2024 competition, and thus the most interesting set. In this set the number of vertices in a graph varies from 512 to 262,124, the density varies from 0.000015 to 0.035 and the number of edges varies from 763 to 262,123.

Our tests involved applying each algorithm and their tie-breaking counterparts to each graph in the `tiny test set`, `medium test set`, and `public test set`. Then recording the original crossings, the duration of the algorithm, the number of edges in the graph, the number of vertices in V_a and V_b, the density of the graph, and the crossings after each algorithm.

8.2 Variations of the Barycenter and Median Algorithms

In both the barycenter heuristic and the median heuristic vertices with the same barycenter or median values are separated by infinitesimal space [2]. However, when implemented this is impossible, as we must decide an ordering. Thus our original approach preserves the original relative ordering of vertices with the same median or barycenter value. In the PACE 2024 competition, we must decide the optimal order of the vertices. We have therefore tested multiple tie-breaking methods to handle the situation that arises when two vertices have the same barycenter or median value and which case handles them best.

Various tie-breaking methods have already been developed and thoroughly tested [3]. We could have used these results, however, we have decided to implement and test the most promising tie-breaking methods on the provided public test graphs as these sparse graphs would not deviate from the graphs used in the competition, whereas the density in the graphs used in [3], were much denser.

8.2.1 Permutations

Our first thought when handling these cases was to iterate through all possible permutations of the vertices with the same barycenter or median values, and then compare the number of crossing each permutation would produce. However, this would mean a significant increase in the time complexity of both heuristics. When we tested on the public test set we found graphs that contained up to 125 vertices with the same barycenter value, meaning that the algorithm would have to go through 125! permutations and thus count the number of crossings 125! times. We therefore decided to enforce an upper bound of 8, meaning that if there were 8 or more vertices with the same median or barycenter value the algorithm kept their original order. However, if there were 7 or fewer the algorithm would find the optimal permutation by going through each one. Furthermore, we used the local count crossings discussed in section 3.2.2 to further reduce the computation time.

8.2.2 A Combination Of Both Algorithms

Our second approach is a combination of both algorithms `BarycenterMed` and `MedianBary`, where vertices that have the same barycenter value produced by the barycenter heuristic are evaluated and reordered by applying the median heuristic and vertices that have the same median value produced by the median heuristic are evaluated and reordered by applying the barycenter heuristic [3].

8.2.3 Reverse

The third and last approach is a `Barycenter and Median heuristic`, `BarycenterRev` and `MedianRev`, where we reverse the order of the vertices that have the same barycenter or median values [3].

8.2.4 Barycenter tie-breaking results

In table 2 the performance of the barycenter heuristics different tie-breaking methods is shown in terms of the number of victories, where one victory is assigned when the least amount of crossings is achieved. These results highlight the effectiveness of the different strategies.

Variation	Victories
Permutations	40
Barycenter	40
BarycenterMed	13
BarycenterRev	6

(a) Overall

Variation	Victories
Permutations	26
Barycenter	12
BarycenterMed	8
BarycenterRev	4

(b) Public

Table 2: The different barycenter heuristic tie-breaking methods and the number of victories they have each. Here a victory is achieved when it produces the fewest amount of crossings for a graph.

Table 2a shows the number of victories for each tie-breaking method when tested on all the test sets, whereas table 2b shows the results for the tie-breaking methods when tested only on the public set. In table 2a we can see that the best tie-breaking methods is to keep their original positioning, such that the order from the input graph is preserved for the vertices with the same barycenter value or to iterate through all possible permutations. The median and reverse tie-breaking approaches produce significantly fewer victories. Therefore, it is clear that either the original or permutation tie-breaking approach must be the best one. In table 2b we see that the permutation variation outperforms the original tie-breaking approach 14 times, as the permutation approach has 26 victories and the original barycenter approach has 12 victories. Since the `public test set` contains the largest graphs, this implies that the permutation approach is better for larger graphs, and the original approach is better for smaller graphs. However, we must keep in mind, that the smaller the graphs the less chance of vertices having the same barycenter value. Thus the barycenter can simply be outperforming the others, as there are no ties to be broken. Furthermore, since the `public test set` is most representative of the graphs that will be used in the PACE 2024 competition, we decided to weigh the performance on the `public test set` higher than the overall performance and therefore we can declare the permutation approach the best tie-breaking method for the barycenter heuristic, where the second best is to preserve the original relative ordering of the vertices.

Figure 15 shows the mean duration of each tie-breaking method when tested on the `public test set`. Here we can clearly see that the permutation method is on average much slower than the other tie-breaking method. This is to be expected as the permutation method goes through every possible ordering of the vertices with the same barycenter value, even with our upper limit at 8. This could prove to be a problem for bigger graphs where a lot of vertices have the same barycenter value. The three other variants roughly take the same amount of time, each taking less than half a second.

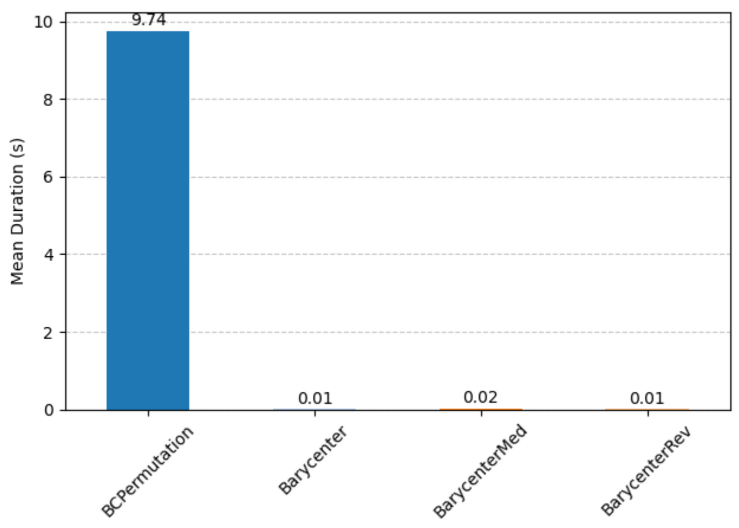


Figure 15: The mean duration of each tie-breaking method for the barycenter heuristic on the `public test set`.

Ultimately, the permutation tie-breaking method is the best tie-breaking method for the barycenter heuristic when it comes to bigger graphs, but this comes at the expense of the slower running time. As our program will have a 5-minute time limit, we must therefore identify when the permutation approach will take longer. This will be discussed more in section 8.2.6.

8.2.5 Median tie-breaking results

In table 3 the performance of the median heuristic's different tie-breaking methods is shown in terms of the number of victories, where one victory is assigned when the least amount of crossings is achieved. These results highlight the effectiveness of different strategies.

Variation	Victories
Permutations	35
MedianBary	23
Median	7
MedianRev	1

(a) Overall

Variation	Victories
Permutations	28
MedianBary	14
Median	5
MedianRev	1

(b) Public

Table 3: Combined table of victories achieved from different median variants.

The table shows that the `Permutation` variant leads with a significant number of victories. This indicates that the permutation method is the most effective method when resolving ties and finding a good solution in the majority of cases. Next, the `MedianBary` variant scores 23 victories overall and 14 victories in the public graphs meaning that this variant has a moderate level of effectiveness compared with the permutation variant. The original `Median` heuristic has 7 victories overall and 5 victories in the public test set while `MedianRev` only has one victory for both the public test set and overall.

Overall the results determine clear winners and losers in the performance of the different variants, where the permutation variant is the most effective for larger graphs and MedianRev the least.

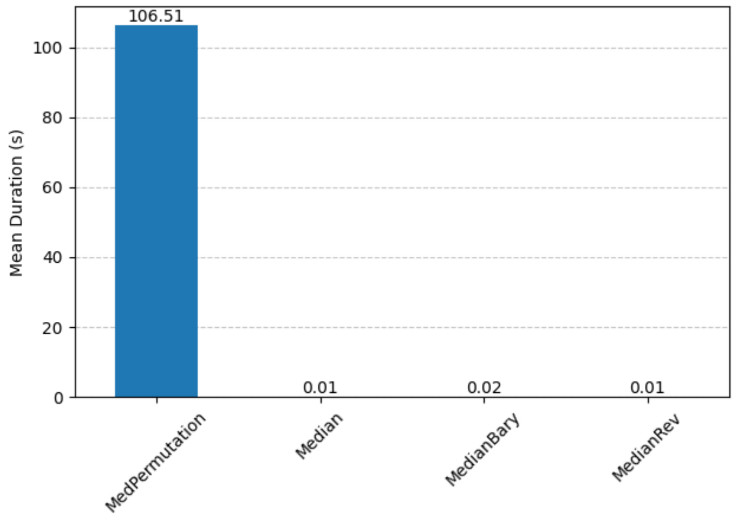


Figure 16: The mean duration of each tie-breaking method for the median heuristic on the `public test set`.

Figure 16 compares the mean duration of each variant. As seen in the figure, the permutation variant has a significantly higher mean duration compared to the other variants. The permutation variant takes longer to find a solution, however, if we recall 3b this variant has the most victories for larger graphs.

Ultimately, the permutation tie-breaking method is the best tie-breaking method for the median heuristic, but this comes at the expense of the slower running time. As our program will have a 5-minute time limit, we must therefore identify when the permutation approach will take longer. This will be discussed more in section 8.2.6.

8.2.6 Identifying time limits

We have tried to identify the limits on graphs for when it is possible to use the permutations tie-breaking method within the 5-minute time limit, which was described in section 8.2.1, for both the median and barycenter heuristic. It is a hard task to determine exactly when it will exceed the time limit, and we have chosen to be conservative in our approach, as if the time limit is exceeded we would receive zero points for the graph. Furthermore, we have examined the duration of the permutation tie-breaking method for both the median and barycenter heuristic on the `public test set`, as it contains the biggest graphs and is thus more likely to contain indicators for when the time limit will be exceeded. Note that the median has a bigger need for an effective tie-breaking method, as the median selects the median neighbor of each vertex and thus there might be more vertices with the same median neighbor. Compared to the barycenter value, which computes the average position based on the neighboring positions. We however, decided that they should share a limit, as the potential worst case for both of them would be $\frac{n}{7} \cdot 7!$, where $n = |V_b|$. This would mean that every vertex in a graph would have the same median or barycenter value as 6 other vertices.

Figure 17 shows a scatter plot where (x,y) = (total number of vertices in the graph, duration in seconds). The horizontal red line is set at 280 seconds, which indicates the time limit as we would like to add a safety margin and not risk exceeding the time limit of 300 seconds. The vertical red line is sat at 13,500 vertices, as this is before the first graph on which the time limit is exceeded by the median permutation method. We therefore decided that the median and barycenter permutation tie-breaking methods only should be used on graphs containing less than 13,500 vertices. This is conservatively placed, as we can see a point right of the line which is under the time limit. This led us to wonder if we could see a clearer limit by looking at the number of edges in a graph. Figure 18 shows exactly that. Here y is the same as in fig. 17, but x is now the total number of edges in the graph. The red horizontal line is the same as in fig. 17, but the red vertical line is sat at 13,000 edges, this is again a conservatively sat limit, as this is when the first point above the time limit occurs.

Ultimately, we were unable to find a clear limit as to when to use and not to use the permutation tie-breaking methods. However, we decided to use the limits we indicated by the red vertical lines, as this was the first point above the time limit. Thus we only use the permutation methods when the following condition is met:

$$|V_a| + |V_b| < 13500 \text{ and } |E| < 13000$$

8.3 Overall results

The two bar plots in figure 19 depict the mean number of crossings found by each heuristic for respectively the tiny (fig. 19a) and medium test (fig. 19b), which were described in section 8.1. A low number of crossings indicates a better performance. For both test sets, the worst heuristic is the bogo heuristic, which is as expected. This is due to the fact that it selects a random permutation of V_b and therefore does not attempt to minimize crossings in a meaningful way. However, this also means that the other heuristics have been implemented correctly. It does however make it harder to see the general performance by the bar plot and we thus decided to exclude the bogo heuristic to produce clearer plots.

The heuristic that produces the second most crossings is the assignment heuristic. This is somewhat unexpected, as Catarci claimed it would always produce a lower amount of crossings than the median heuristic, and especially when the density was greater than 30% [7]. The contradiction in our data could stem from either a wrongful implementation of the algorithm or that the graphs in our test sets are not dense enough for the assignment heuristic to perform optimally. We chose to believe that it is the latter, as we have implemented the assignment heuristic by using a preexisting implementation of the Hungarian reduction from [12]. Furthermore, fig. 19b shows that it computes a permutation of V_b that, on average, produces around 5000 crossings more in the medium test set. Since the assignment heuristic algorithm runs in $O(n^3)$ time, we knew that it probably could not be used for all graphs in the competition due to the five minute time limit. But we were interested to see if it was possible to apply it to the smaller graphs. However, even in the medium test, the average duration of the assignment heuristic was around 250 seconds, as shown in figure 20. This is only 50 seconds away from the limit in the medium test set, whereas the smallest graphs in the public set contained the same amount of vertices as some of the bigger graphs in the medium test set. Therefore, we, conservatively, decided not to include the assignment heuristic in further tests

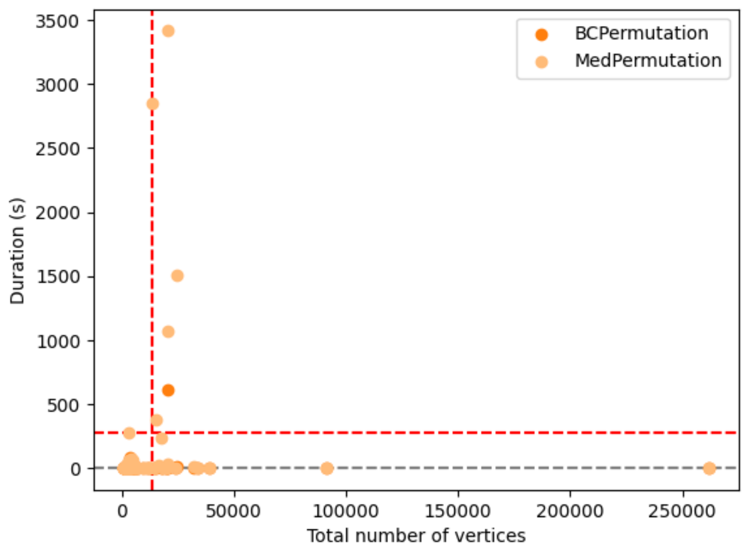


Figure 17: A scatter plot of the duration and total number of vertices for both permutation tie-breaking methods. The red horizontal line is set at 280 seconds, and indicates the time limit, whereas the red vertical line is sat at 13,500 vertices and acts as a suggested upper limit.

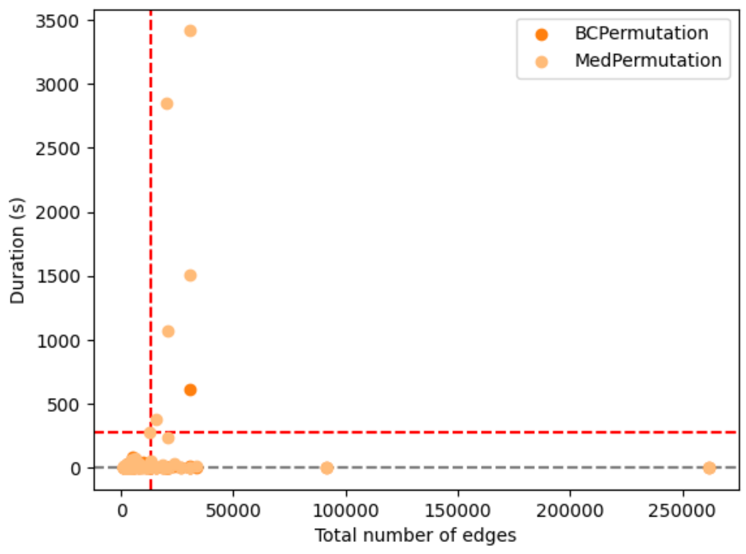
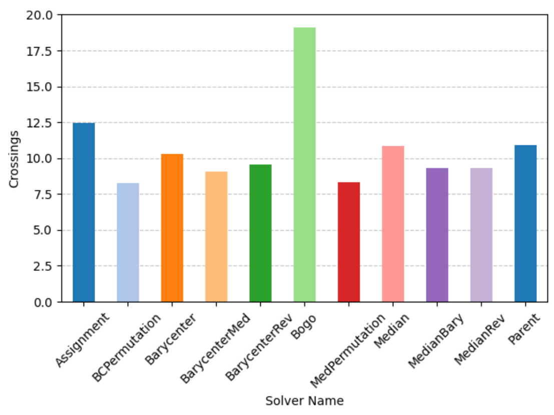
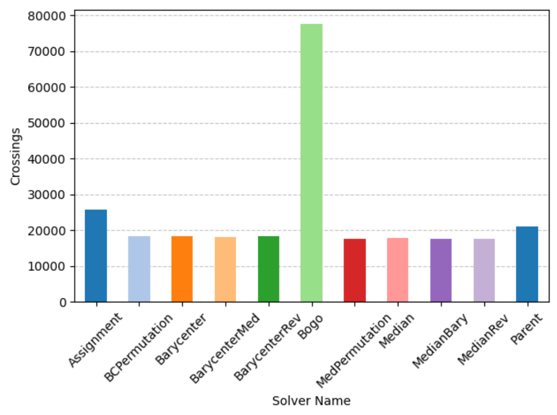


Figure 18: A scatter plot of the duration and total number of edges for both permutation tie-breaking methods. The red horizontal line is set at 280 seconds, and indicates the time limit, whereas the red vertical line is sat at 13,000 vertices and acts as a suggested upper limit.



(a) Mean crossings tiny for all our heuristics.



(b) Mean crossings medium for all our heuristics.

Figure 19: Two bar plots depicting the mean number of crossings for each heuristic.

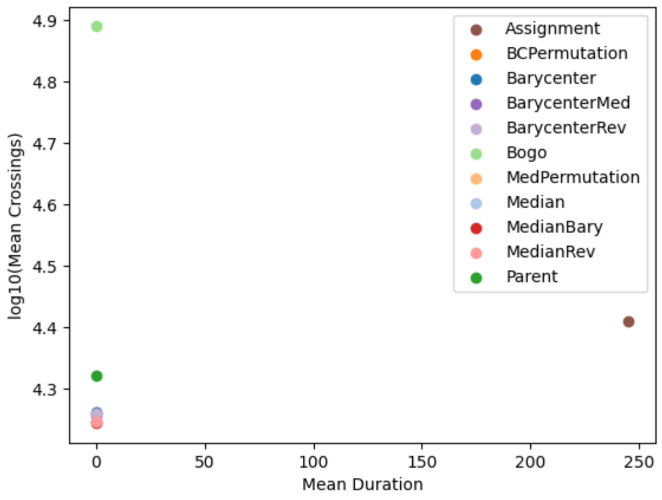
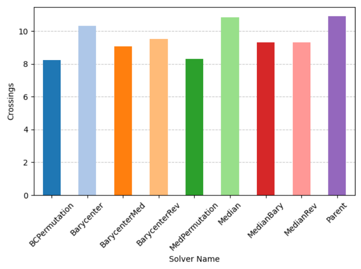


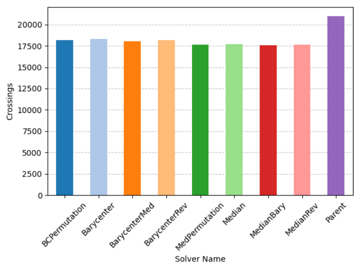
Figure 20: Performance for all heuristics on the medium test set.

and thus not in our hybrid algorithms, as the graphs simply were too large and the results were not competitive enough.

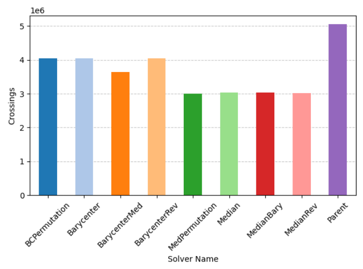
The bar plots in figure 21 show the mean number of crossings for all the heuristics when applied to the `tiny`, `medium`, and `public` test sets, that was described in section 8.1. We have filtered out the bogo heuristic and the assignment heuristic, due to the above reasons. In fig. 21c, we can see that the parent heuristic is significantly worse than the other heuristics, and as such we have decided not to include it further. The bar plot for the public test set is the most promising since it provides the



(a) Mean crossings tiny



(b) Mean crossings medium



(c) Mean crossings public

Figure 21: Mean crossings of our main minimizers in the `tiny`, `medium`, and `public` test sets.

most relevant comparison as it represents the competition's test environment. It is clearly visible which heuristics can find the lowest number of crossings and which do not. The median variants are the most effective in terms of finding fewer crossings with the permutation variant in the lead, while the barycenter variants had a slightly worse performance than the median heuristic. It is, however, worth remembering that even if it looks like the barycenter variants have worse performance than the median variants in terms of mean crossings, the barycenter variants still win in some cases as seen in table 4.

We have also managed to identify a condition where the barycenter heuristic outperforms the median heuristic. This is seen in figure 22 which shows a scatter plot of the median and barycenter heuristics, where each point represents a victory for either the median or barycenter heuristic. This was done to reduce the noise the graph otherwise would have. Because of this, we were then able to identify a density interval in which only the barycenter heuristic is present. This interval is $0.003 < \rho < 0.0048$, where $\rho =$ density, and this means that the barycenter heuristic has outperformed the median heuristic for all graphs in the public test set with a density in this interval. Thus we were able to conclude that the barycenter heuristic must be better for graphs with a density in the interval.

Based on table 4a, we can see that the barycenter heuristics and its tie-breaking methods create a permutation of V_b that results in fewer edge crossings, than the median heuristic and its tie-breaking methods 33 times. However, when we narrow the sample size down to only contain graphs from the public test set, as shown in table 4b, we see that this discrepancy is lower. This table is more representative of the graphs that will be present in the competition and thus serves as a better indicator. But this also tells us that the barycenter heuristic is better when dealing with smaller graphs, such as the ones present in the tiny - and medium test sets. The best performing heuristic is actually the median heuristic using the permutation tie-breaking method with 28 victories. This is closely followed by the barycenter heuristic using the permutation tie-breaking method with 26 victories. Thus these two methods are the two best performing heuristics. How-

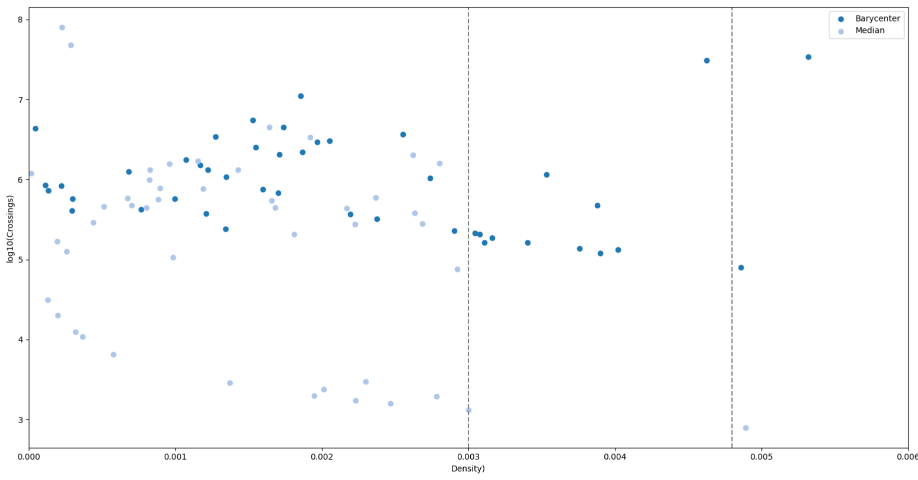


Figure 22: Density interval for where the barycenter is more efficient than the median minimizer.

Heuristic	Victories
Barycenter variants	99
Barycenter	40
Permutations	40
BarycenterMed	13
BarycenterRev	6
Median variants	66
Permutations	35
MedianBary	23
Median	7
MedianRev	1
Assignment	5
Parent	2

(a) Overall

Heuristic	Victories
Barycenter variants	50
Permutations	26
Barycenter	12
BarycenterMed	8
BarycenterRev	4
Median variants	48
Permutations	28
MedianBary	14
Median	5
MedianRev	1
Parent	1

(b) Only looking at public test set

Table 4: The combined table of victories achieved from different median and barycenter variants across all the test sets(a) and the public set(b). A victory is achieved when the heuristic produces a permutation of V_b that results in the lowest amount of edge crossings compared to the other heuristics.

ever, these two methods also have the highest duration, as explained in section 8.2.6. Thus we should also identify heuristics to use when the graph indicates that the duration of the permutation tie-breaking method would be too great. The two heuristics that perform the third and fourth best are respectively the median heuristic using the barycenter heuristic as tie tie-breaking method, with 14 victories, and the barycenter heuristic with 12 victories. These two heuristics should therefore be used when the permutation tie-breaking method's expected duration is too great. Furthermore, because of the density interval we have identified, we know that the barycenter heuristic performs better when the density of the graph is $0.003 < \rho < 0.0048$. This information was used in forming our hybrid algorithm which we will discuss in section 9.

9 Hybrid algorithm

Based on the results of the testing, see section 8.3, of the barycenter and median heuristics and their variants, we developed a hybrid algorithm that utilizes the strengths of both heuristics. We have decided to have our hybrid algorithm utilize the time limit constraint of five minutes in the competition, and thus our hybrid algorithm can be seen as a priority queue. Before we dequeue a heuristic from the queue we ensure that there is a minimum of 20 seconds remaining, to ensure it does not exceed the time limit. The priority in which we have added the heuristics to the queue can be seen below and is based on our findings in section 8.3.

1. Median - Permutations
2. Barycenter - Permutations
3. Median - Barycenter
4. Barycenter
5. Barycenter median
6. Median

However, this priority can changed based on the graph's properties. We identified two limits for when we could use the permutation tie-breaking method, in section 8.2.6. Thus if the graph exceeds these limits, $|V_a| + |V_b| \geq 13500$ or $|E| \geq 13000$, we do not enqueue the permutation tie-breaking methods for both the median and barycenter heuristics. Furthermore, if the density of the graph, ρ, $0.003 < \rho < 0.0048$ then the barycenter heuristics have a higher priority than the median heuristics.

Once we dequeue a heuristic, it is then used to compute a permutation, π_{V_b} of V_b, and count the edge crossings produced by this permutation. The number of crossings and π_{V_b} is stored as a pair in an array. Ultimately, when the time limit is reached or the queue is empty, the permutation that produces the fewest crossings is chosen and π_{V_b} is returned.

References

[1] *Pace 2024*, https://pacechallenge.org/2024/, Accessed on 13th of may 2024, 2023.

[2] P. Eades and N. C. Wormald, "Edge crossings in drawings of bipartite graphs," *Algorithmica*, vol. 11, no. 4, pp. 379–403, Apr. 1994, ISSN: 1432-0541. DOI: 10.1007/BF01187020. [Online]. Available: https://doi.org/10.1007/BF01187020.

[3] T. Poranen and E. Mäkinen, "Tie-breaking heuristics for the barycenter and median algorithms," Apr. 2006.

[4] H. Gruber, M. Holzer, and O. Ruepp, "Sorting the slow way: An analysis of perversely awful randomized sorting algorithms," in *Fun with Algorithms*, P. Crescenzi, G. Prencipe, and G. Pucci, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 183–197.

[5] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computional Geometry, Algorithms and Applications*, 3rd. Springer Berlin, Heidelberg, 2008, ISBN: 978-3-540-77973-5.

[6] G. Sander, "Graph layout through the vcg tool," in *Graph Drawing*, R. Tamassia and I. G. Tollis, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 194–205, ISBN: 978-3-540-49155-2.

[7] T. Catarci, "The assignment heuristic for crossing reduction," *Systems, Man and Cybernetics, IEEE Transactions on*, vol. SMC-25, pp. 515–521, Apr. 1995. DOI: 10.1109/21.364865.

[8] *Munkres assignment algorithm*, Accessed: 2024-05-17, 2024. [Online]. Available: https://users.cs.duke.edu/~brd/Teaching/Bio/asmb/current/Handouts/munkres.html.

[9] E. Mäkinen, "Remarks on the assignment heuristic for drawing bipartite graphs," *Acta Polytechnica Scandinavica, Mathematics and Computing Series No. 46*, 1990. [Online]. Available: https://trepo.tuni.fi/handle/10024/65543.

[10] X. Yu and M. Stallmann, "New bounds on the barycenter heuristic for bipartite graph drawing," Jul. 2001.

[11] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis, *Graph Drawing: Algorithms for the Visualization of Graphs*, 1st. Prentice Hall PTR, 1998, ISBN: 0133016153.

[12] *Hungarian algorithm — Implementation in C++*, Accessed: 2024-05-17, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Hungarian_algorithm#Implementation_in_C++.