On Crossing Minimization Problems: Complexity and a Heuristic Approach

Bachelor Thesis

Simon Hol
s.hol@students.uu.nl

June 14th 2024

1st supervisor: dr. Marieke van der Wegen
2nd supervisor: dr. Erik Jan van Leeuwen
3rd supervisor: prof. dr. Hans Bodlaender

Faculty of Science
Department of Mathematics

Universiteit Utrecht

Contents

1 Introduction

This thesis is a but part of the whole project. The project as a whole is a submission for the PACE challenge 2024 [8]. As their about page states: "The Parameterized Algorithms and Computational Experiments Challenge (PACE) was conceived in Fall 2015 to deepen the relationship between parameterized algorithms and practice. Topics from multivariate algorithms, exact algorithms, fine-grained complexity, and related fields are in scope."

This year, in 2024, the problem is a so-called ONE-SIDED CROSSINGS MINIMIZATION PROBLEM (OCP). Informally, OCP wants to find an ordering of vertices of a layered graph such that the number of crossings of edges is minimal. A formal definition will be given later.
The problem features three different tracks with different rules and requirements for submissions: an exact track, the heuristic track, and a parameterised track. Our submission is for the heuristic track, which means that the result does *not* have to be optimal, but rather produce a *good* solution *quickly*.

For this thesis, after introducing preliminary notation and definitions, we will introduce the problem, including some of the variants and a proof of $\mathcal{NP}$-completeness. Then we will look at known heuristic methods, with our motivation for the methods we chose to implement. We will discuss some tiebreaking methods, methods that handle ties for certain heuristics.
After specifying what methods we intend to implement, we will give a general overview of the code [8], breaking down the datastructures, important functions, etc.
Finally, we end with a short discussion with some remarks about features that could be improved/optimised or added, as well as choices and ideas that could benefit from more extensive research.

1.1 Notation and definitions

Here, we will give some of the more general terminology and definitions we will use for this thesis.

For specifying vertices or edges of a graph G, we will use the notation $G = (V, E)$, where V are the vertices and E the edges. Furthermore, if the graph is a *(k-)layered graph*, it is denoted by $G = (L_0, L_1, ..., L_k, E)$, with L_i the k layers, and $E \subset L_0 \times L_1 \times \ldots \times L_k$ the edge set with $e \in E$ of the form $e = (v_0, v_1, \ldots v_k), v_i \in L_i \forall 0 \leq i \leq k$. The definition of a layer is a vertex set with the property that each edge has its endpoints in subsequent layers, and each each pair of layers is disjoint. The special case $k = 2$, which gives a bipartitite graph, will be given as $G = (L, U, E)$, with L, U for the "lower" and "upper" layer/side respectively instead of L_0, L_1. Furthermore, any

graph is complete if for each pair of vertices, there exists and edge between them. For a layered graph we specify this to edges between layers only, i.e. a layered graph is complete if for any pair of vertices in adjacent layers L_i and L_{i+1} there is an edge between them.
An *ordering* on a layer L_i is denoted by l_i, such that $\forall v \in L_i$, $l_i(v)$ gives the position of v. For comparison of vertices u, v in a layer L_i, we say $l_i(v) < l_i(u)$ if v comes before u in that ordering, and $l_i(v) < l_i(u)$ similarly defined. In particular, each vertex is assigned a *unique* position $\{1...|L_i|\}$.
For clarity, we use the notation l_0 and u_0 for bipartitite graphs $G = (L, U, E)$, even though L and U do not have subscripts themselves. This is to make distinction between vertices and orderings more clear. A *drawing* of a graph $(G, l_0, l_1, ..., l_k)$, is a visual representation of a graph G, given a specific order for each layer.

Lastly, we define $\text{cross}(G, l_0, l_1, ..., l_k)$ denoting the number of crossings for these orderings on the layers, and $\text{opt}(G)$ as the order such that $\text{cross}(G, l_0, l_1, ..., l_k)$ is minimal. This notation extends for (partially) given orderings, for example a one-sided fixed problem can be denoted as $\text{opt}(G, l_0)$ where l_0 is a known ordering on L. Important is that the *position* (i.e. coordinates) of the vertices does not matter for the total number of crossings, only the relative *order* or the vertices. The number of crossings can be calculated by checking each pair of edges and see if they cross or not. Note that a crossing between two edges $e = (u, v)$ and $e' = (u', v')$, with $e, e' \in E$, only occurs if the endpoints are in reverse order in their respective layers. More clearly, assume $u, u' \in L_i$ and $v, v' \in L_{i+1}$. We say e and e' cross if $l_i(u) < l_i(u') \wedge l_{i+1}(v) > l_{i+1}(v')$, or vice versa. Important notion is that, if we look at each pair of edges, we count both options e, e' and e', e. This essentially counts each crossing twice, hence we can restrict ourselves to one of the cases. Using $l_i(u) < l_i(u') \wedge l_{i+1}(v) > l_{i+1}(v')$ as the condition for crossing, define $c_{e,e'}$ as 1 if e and e' cross, and 0 otherwise. This gives us the following formula for the total number of crossings for a bipartite graph:

$$\text{cross}(G, l_0, u_0) = \sum_{e,e' \in E} c_{e,e'} \tag{1}$$

For k-layered graphs with $k > 2$, we can use the same method applied to each pair of subsequent layers, i.e. counting the crossings between L_0 and L_1, then L_1 and L_2, etc. and summing these $k - 1$ terms.

2 Problem definition

In this section we will look at the different variants of crossing minimization problems, with our main focus on the *one-sided crossing minimization problems of a 2-layered graph*, including a proof of $\mathcal{NP}$-completeness.

2.1 Variants of Crossing minimization problem

The first variant is whether or not one side of the graph is fixed. As the name suggests, the one-sides variant treats one of the layers as fixed, so a given ordering on that layer, whereas the two-sided does not.
The second variant is the number of independent layers of G. An important note is that for $k > 2$, this generally does not allow a two-sided variant. Instead, most known algorithms/approaches for k-layered graphs use a so-called "layer by layer sweep", for example the methods introduced by Matuszewski et al. [12], and Patarasuk [13]. The idea of these implementations is that they iterate over the layers, starting with an arbitrary order of the first layer. This means that it treats each pair of subsequent layers as a one-sided crossing minimization problem, finding a starting order for the next pair, and solving the next pair of layers. These algorithms go "up and down" so to speak. That is to say, once an ordering of the last layer L_k is found, the algorithm solves for L_k and L_{k-1} with the found order for L_k. Often, there is some sort of condition that determines when to stop, for example if there can no longer find an improvement on the ordering, or a certain number of sweeps has been done. In the case of the two-sided approach, most implementations aim to check all possible permutations, for example the "branch and bound method" [17]. This method uses a decision tree to check all permutations. A more detailed overview of the Branch-and-bound method will be given in Section 3.1.3.

2.2 Problem complexity (NP-completeness proof)

We will now give a proof of $\mathcal{NP}$-completeness for the One-sided crossing minimization problem, hereafter shortened to Crossings Problem, following Eades et al. [6]. First, we will give the definition of the Crossings problem, followed by the decision problem induced from a crossings problem. The decision version is used so that we can show that DCP is in $\mathcal{NP}$ by giving a method to *verify* a given solution in polynomial time, whereas we cannot do that with OCP.

Crossings Problem (OCP)
Instance: Given a bipartitite graph $G = (L, U, E)$, and ordering l_0 of L.
Problem: Find an ordering u_0 of U such that the total number of crossings $cross(G, l_0, u_0)$ is minimal.

The decision problem can then be defined as follows:

Decision Crossings Problem (DCP)
Instance: Given a bipartitite graph $G = (L, U, E)$, and an ordering l_0 of L and an integer M.
Problem: Is there an ordering u_0 of U such that $cross(G, l_0, u_0) \leq M$.

To prove $\mathcal{NP}$-completeness, we will use a reduction, in polynomial time, from a known $\mathcal{NP}$-complete problem to DCP, concluding that DCP must be $\mathcal{NP}$-complete as well. For this, we use the Feedback Arc Set (FAS) problem, which is $\mathcal{NP}$-complete as proven by Karp [10].

Let us first introduce the Feedback Arc Set problem we will use as our reduction problem:
Given a directed graph D with node set V and arc set B, a *feedback arc set* B' for D contains at least one arc of each directed cycle of D. Note that it follows directly that $B \setminus B'$ must be acyclic.

The formal definition of the Feedback Arc Set problem is:

Feedback Arc Set (FAS)
Instance: Given a directed graph D, and a positive integer k.
Problem: Does D have a feedback arc set of size at most k?

We will now proceed to prove the following theorem:

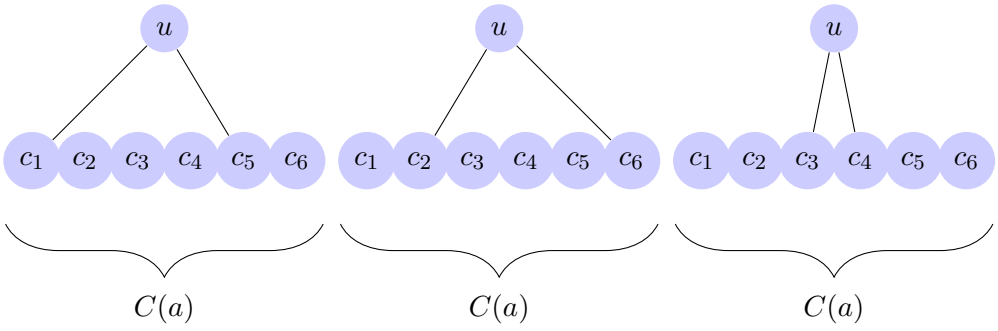
Theorem 2.1. DCP *is $\mathcal{NP}$-complete.*

Proof. It is easy to see that DCP is in $\mathcal{NP}$: as shown earlier in Equation (1), we can calculate the number of crossings of a graph with given orderings of each layer in $O(|E|^2)$. This means that we can verify a solution in polynomial time, hence it is in $\mathcal{NP}$.

For the reduction from FAS to DCP, we want to transform a directed graph $D = (V, B)$ into a two-layered graph $G = (L, U, E)$ from a given instance of FAS.
So, given a directed graph $D = (V, B)$, let $U = V$. Then, for each arc $a \in B$, we define a "clump" $C(a) = \{c_1(a), c_2(a), ..., c_6(a)\}$ of six nodes and let L be the union over B of all the clumps.

For the edges of G, we create two edges for each $u \in V$ and $a \in B$. If $a = (u, v)$ for some $v \in V$, then u is joined to $c_1(a)$ and $c_5(a)$; if $a = (v, u)$ for some $v \in V$, we create the edges between u and $c_2(a)$ and $c_6(a)$. If u is not incident with a, i.e. u is not an endpoint of a, then we join u to $c_3(a)$ and $c_4(a)$. See Figure 1 for a visualisation of the three options.

Now, let l_0 be an ordering on L that keeps the clumps together and



(a) $C(a)$ *where* $a = (u, v)$. *(b)* $C(a)$ *where* $a = (v, u)$. *(c)* $C(a)$ *where* a *is not incident with* u.

Figure 1. *The three possibilities visualised.*

ordered, i.e. for each clump we have $l_0(c_i(a)) < l_0(c_j(a))$ for $1 \leq i < j \leq 6$ and $a \in B$, as well as that for each $a, b \in B$ either $l_0(c_6(a)) < l_0(c_1(b))$ or $l_0(c_1(a)) > l_0(c_6(b))$ must hold. This enforces that no clumps intermingle, i.e. all $C(a)$ are subsequent and grouped together without any other between.

We will now show that D has a feedback arc set of size at most k if and only if an ordering u_0 of U can be chosen such that $\mathit{cross}(G, l_0, u_0) \leq M$ with $|B| = \beta, |V| = \nu$ and

$$M = 4\binom{\beta}{2}\binom{\nu}{2} + \beta\binom{\nu-2}{2} + 4\beta(\nu-2) + \beta + 2k.$$

For this, we will first prove the following lemma:

Lemma 2.2. *If* u_0, *is an ordering of* U *and* B' *denotes* $\{(u, v) \in B : u_0(u) > u_0(v)\}$, *then*

$$\mathit{cross}(G, l_0, l_1) = 4\binom{\beta}{2}\binom{\nu}{2} + \beta\binom{\nu-2}{2} + 4\beta(\nu-2) + \beta + 2|B'|.$$

Proof. First, look at crossings between arcs incident to different clumps: Let a and b two distinct arcs in B. Then, for each $u \in V$, we have two arcs between u and $C(a)$ and two between u and $C(b)$. This means that, for any distinct pair $u, u' \in V$, we will always have four crossings between the arcs incident with u and $C(a)$ and arcs incident with u' and $C(b)$ (see Figure 2 for an example). Summed over each pair of clumps and pair of nodes, we get a total of $4\binom{\beta}{2}\binom{\nu}{2}$.

For the number of crossings between arcs from the same clump $C(a)$, let us look at the different cases.

First, the number of crossings between arcs incident with $c_3(a)$ and $c_4(a)$. The number of arcs to $c_3(a)$ and $c_4(a)$ is $\nu - 2$, since for all but the endpoints

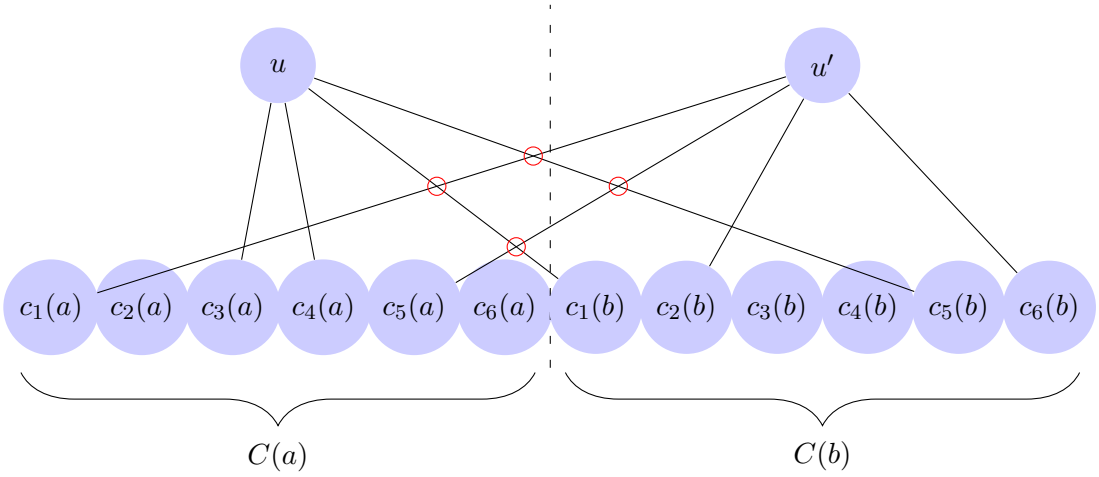
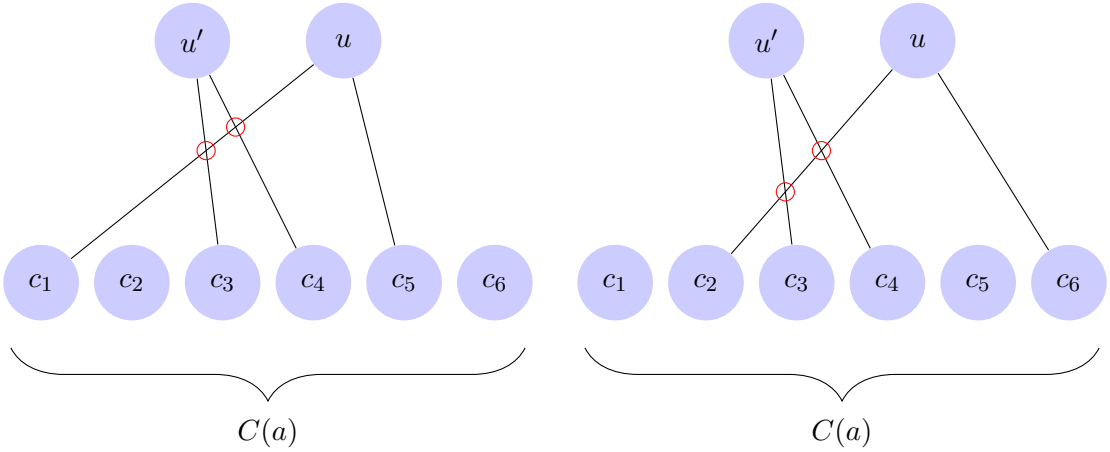


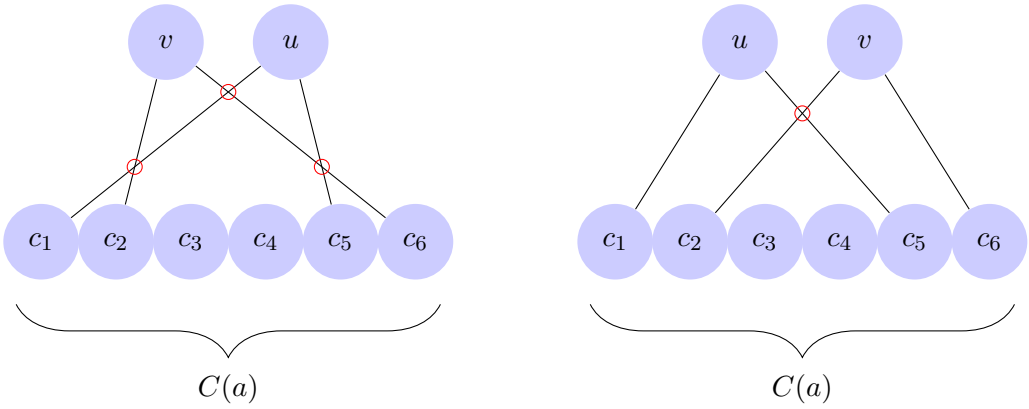
Figure 2. *Example with two arcs and two vertices.*

of the arc a we have that a is not incident with those vertices, so each falls in the category depicted in Figure 1c. This means that $|U|-2 = \nu-2$ nodes each give on edge incident with $c_3(a)$ and on with $c_4(a)$. Each distinct pair vertices $u, u' \in V$ results in a crossing, so this gives an additional $\binom{\nu-2}{2}$ crossings.
Furthermore, if $a = (u, v)$, then each arc out of $c_3(a)$ and $c_4(a)$ crosses one of $(u, c_1(a))$ and $(u, c_5(a))$ and one of $(v, c_2(a))$ and $(v, c_6(a))$ (see Figure 3). This gives a total of $4(\nu - 2)$ crossings, namely $2(\nu - 2)$ crossings between the $\nu - 2$ arcs out of c_3 and c_4 and the two edges between u and $C(a)$, and $2(\nu - 2)$ crossings with the arcs between $C(a)$ and v.



(a) *Case with $a = (u, v)$.* ***(b)*** *Case with $a = (v, u)$.*

Figure 3. *Example with two clumps and two arcs.*



(a) Case with $l_1(u) > l_1(v)$. *(b) Case with $l_1(u) < l_1(v)$.*

Figure 4. *Showing scenarios of ordering on arc $a = (u, v)$.*

The only crossings not counted yet, are crossings between arcs $(u, c_1(a))$, $(u, c_5(a))$, $(v, c_2(a))$ and $(v, c_6(a))$. We claim that this is 1 if $u_0(u) < u_0(v)$ and 3 if $u_0(u) > u_0(v)$. First, note that the exact position in U does not matter, only the position relative to each other. This can then easily be seen if we visualise the two scenarios, see Figure 4.
If we combine all this, we get that the total number of crossings between arcs from $C(a)$ is $\binom{\nu-2}{2}+4(\nu-2)+1$ if $u_0(u) < u_0(v)$ and $C(a)$ is $\binom{\nu-2}{2}+4(\nu-2)+3$ if $u_0(u) > u_0(v)$. If we sum this over all β clumps we get $\beta - |B'|$ terms with the 1, and $|B'|$ terms with the 3, since we defined B' as the set of elements that satisfy $u_0(u) > u_0(v)$. Now we can simply rewrite this to get

$$\beta\left(\binom{\nu-2}{2}+4(\nu-2)\right)+(\beta-|B'|)+3|B'| =$$
$$\beta\binom{\nu-2}{2}+4\beta(\nu-2)+\beta-|B'|+|B'|+2|B'| =$$
$$\beta\binom{\nu-2}{2}+4\beta(\nu-2)+\beta+2|B'|.$$

Combining this with the first term we found of the number of crossings between arcs incident with different clumps, we get the desired result from the lemma:

$$cross(G, l_0, l_1) = 4\binom{\beta}{2}\binom{\nu}{2}+\beta\binom{\nu-2}{2}+4\beta(\nu-2)+\beta+2|B'|.$$

■

Now to prove the theorem, we will use topological sort. A topological sort, or topological ordering, of a Directed Acyclic Graph (DAG) is a

linear ordering of vertices, such that for every directed edge (u, v), u comes before v. In our case, because D is directed and $B \setminus B'$ is acyclic, we can use topological sort.
Now suppose that D has a feedback arc set B' of size at most k. We can use topological sort on D to obtain an ordering u_0 on U such that $u_0(u) < u_0(v)$ if $(u, v) \in B \setminus B'$ since $B \setminus B'$ is acyclic.
Combined, since $|B'| \leq k$ and

$$B' = \{(u, v) \in B \ : \ u_0(u) > u_0(v)\}.$$

Theorem 2.2 implies that G has at most M crossings. This follows directly from filling in the lemma and using $|B'| \leq k$.

$$\begin{aligned} \mathit{cross}(G, l_0, u_0) &\leq 4\binom{\beta}{2}\binom{\nu}{2} + \beta\binom{\nu-2}{2} + 4\beta(\nu-2) + \beta + 2|B'| \\ &\leq 4\binom{\beta}{2}\binom{\nu}{2} + \beta\binom{\nu-2}{2} + 4\beta(\nu-2) + \beta + 2k = M. \end{aligned}$$

Conversely, suppose G has an ordering l_0 on L_0 such that G has at most M crossings. Then we know that

$$\mathit{cross}(G, l_0, u_0) \leq M = 4\binom{\beta}{2}\binom{\nu}{2} + \beta\binom{\nu-2}{2} + 4\beta(\nu-2) + \beta + 2k.$$

Additionally, it follows from Theorem 2.2 that, if $B' = \{(u, v) \in B : u_0(u) > u_0(v)\}$, then

$$\mathit{cross}(G, l_0, u_0) \leq 4\binom{\beta}{2}\binom{\nu}{2} + \beta\binom{\nu-2}{2} + 4\beta(\nu-2) + \beta + 2|B'|.$$

so $|B'| \leq k$, since M is defined using k. Additionally, the graph $D' = (U, B \setminus B')$ must be acyclic, which makes B' a feedback arc set. This can be seen since, if D' is not acyclic, there is some arc (u, v) such that $l_1(u) > l_1(v)$ (based on a topological sort), which is a contradiction, since we removed all those arcs in B', so we conclude that D' must be acyclic, so B' must be a feedback arc set, which concludes the proof of the theorem and we conclude that DCP is indeed $\mathcal{NP}$-complete. □

3 Known heuristic methods and their comparison

In this section we will go over some known heuristic methods, providing a short explanation. Then we will discuss the methods we chose to implement, with reasoning why that method was chosen.

3.1 Comparison between different considered heuristic methods

Mainly considering the time constraint on the thesis, we had to limit ourselves as to what we could implement. As a result, we looked at known methods and their performance relative to each other on One-sided crossing minimization problems to determine which methods would be valid options.

The methods we considered for One-sided crossing problems where the Assignment Method, the Stochastic Method, the Branch-and-bound Method, the Barycenter Method and Median Method. We will give a short overview of each of these methods, followed by argumentation on our final choice.

3.1.1 The Assignment Method

This method, first introduced by Catarci [1] considers OCP as an assignment problem. Generally, an assignment problem can be seen as a problem to find an optimal (i.e. highest rating/reward, or lowest cost) *assignment* of *jobs/tasks* to *applicants/workers*, with some sort of rating/reward or cost for each assignment.

This method specifically transforms OCP to an Assignment Problem, then applies a known algorithm to solve that Assignment Problem. An example of an algorithm used for the assignment problem is the so-called Hungarian Method [11], which solves an assignment problem with m *tasks* and m *jobs* in $O(m^3)$ time.
A formal definition of the assignment problem, with m jobs and applicants, uses a *cost* $c_{i,j}$ for each combination of job i with applicant j.
Furthermore, let x be a $m \times m$ matrix such that x_{ij} denotes whether applicant i is assigned to job j. Specifically, $x_{i,j}$ is 1 if assigned, 0 otherwise. We also add the constraint that each worker is assigned one job only, and that each job is done by exactly one person. This gives the (formal) conditions $\sum_{i=1}^{m} x_{i,j} = 1$ for all m applicants, and similarly, $\sum_{j=1}^{m} x_{i,j} = 1$ for all m jobs.
Then, the formal definition becomes:

Definition 3.1. For a general assignment problem of size m, we have

$$\text{Minimize } Z(x) = \sum_{i=1}^{m}\sum_{j=1}^{m} x_{ij}c_{i,j}$$

$$\sum_{i=1}^{m} x_{i,j} \text{ for } j = 1 \text{ to } m$$

$$\sum_{j=1}^{m} x_{i,j} \text{ for } i = 1 \text{ to } m$$

$$x_{ij} \in \{0, 1\} \text{ for } i, j = 1 \text{ to } m$$

Here, x is called a *base feasible solution*, or assignment matrix. Then x_{ij} is said to be allocated if $x_{ij} = 1$ and not allocated if it is 0. This also implies the conditions that each row (and column) can have only one allocation each.

In particular, a solution x specifically for an assignment problem results in a permutation of U, denoted by the $m \times m$ assignment matrix such that $x_{ij} = 1$ if and only if vertex i is placed at position j. The minimization of $Z(x)$ then uses this matrix to determine which c_{ij} count towards the total cost, denoted by the multiplication with x_{ij}.

Now, to apply this to a crossings problem, we do the following:

Given an instance of a crossings problem with $G = (L, U, E)$, $|L| = n, |U| = m$, G a bipartite graph and l_0 a fixed ordering on L. We denote A as the $n \times m$ adjacency matrix of G such that $a_{ij} = 1$ if and only if i and j are adjacent.

Let B a $m \times n \times m \times n$ matrix, such that its elements $b_{a,b,c,d}$ are 1 if and only if ($c > a$ and $d < b$) or ($c < a$ and $d > b$) holds, and 0 otherwise for all $a, c \in U, b, d \in L$. We see that this is precisely the condition to check if the pair of edges (a, b) and (c, d) cross, for $a, c \in U, b, d \in L$. Note that this matrix B gives all *possible* crossings, if G is a complete graph. That is, if G is not complete, and not all pair of edges (a, b) and (c, d) exist, even though they *would* cross, $B_{a,b,c,d}$ is still defined as if these edges exist.

With this, can define the cost-function for any given $i, j \leq m$

$$c_{ij} = \sum_{h=1}^{n}\left(a_{hi}\left(\sum_{c=1}^{m}\sum_{d=1}^{n} b_{j,h,c,d}\right)\right)$$

where, $a_{h,i}$ is an element of A, the adjacency matrix, and $b_{j,h,c,d}$ an element of B, the crossings matrix.

The idea is that $c_{i,j}$ is the *maximum* number of crossings resulting from placing the i'th vertex at position j. Again, here we treat the remainder of G like a complete graph (i.e. the graph consisting of all vertices except i is complete), so this gives an upper bound of the number of crossings caused by placing vertex i at position j.

Now that we have our cost-function, we can treat this as an assignment problem and solve it using the given formulas, with the important note that, since B treated G like a complete graph, we are given an upper bound of number of crossings, thus there is no guarantee that the resulting order is optimal.

3.1.2 The Stochastic Method

Dresbach [3] introduced the stochastic method for directed acyclic graphs. Because the graph was acyclic, one can transform that graph into a k-layered graph since the layers can be determined based on the direction of the edges, for example using topological sort. Once the layers are established, the directions of the edges are no longer of importance. So in the case of a OCP, where a bipartitite graph G is given, we can still use this method even though it is not a directed graph, since the layers are already established.

For the remainder of this section, let $G = (L, U, E)$ a bipartite graph with an ordering l_0 on L, and $n = |L|, m = |U|$. Specific points will be referred to as $l_i \in L$ and $u_j \in U$ with $1 \leq i \leq n$ and $1 \leq j \leq m$. For a specific drawing of G, define M as the adjacency matrix, with the rows and columns given by L and U respectively. Note that L and U must be in the same order as the order in which they appear in that particular drawing of G. Then $m_{i,j}$ is 1 if the vertices in position i and j are adjacent, and 0 if not.

The total number of crossings is then, using the following calculation and M, as follows:

$$\text{cross}(G, M) = \sum_{a=1}^{n-1} \sum_{b=a+1}^{n} \sum_{c=1}^{m-1} \sum_{d=b+1}^{m} m_{a,d} m_{b,c}. \tag{2}$$

Now, define a *frequency matrix* F, with F an $m \times n$-matrix. Here $f_{i,j}$ is the number of times an edge crosses the edge between l_i and u_j for a complete graph. Note that this is the number of times that $m_{i,j}$ is counted in Equation (2), which gives rise to the following formula:

$$f_{i,j} = (m - j)(i - 1) + (n - i)(j - 1) \; \forall i = 1..n, j = 1..m.$$

Since this method uses a geometrical mean, some values must be redefined: $f_{1,1} = f_{n,m} = 1$, and not 0 (as it would otherwise be). Similarly to the Assignment Method, because $f_{i,j}$ treats G as complete, it gives us an estimation of the number of crossings caused by the edge between positions l_i and u_j.

We will use a so-called "assessment number" $a_{i,k}$ to denote the number of crossings from placing the i'th vertex (w.r.t. the original, that is the given, order) in L, call this vertex l_i, at the k'th position. We do this by taking the geometric mean of the elements in the row vector of l_i (i.e. all adjacent

vertices of l_i). The same can be done for elements in $u_j \in U$ by defining $b_{j,l}$ in a similar manner.

For $i = 1..n$ and $k = 1..n$, the mean is given by:

$$a_{i,k} := \sqrt[\delta(l_i)]{\prod_{j=1}^{m} (f_{k,j})^{m_{i,j}}}.$$

where $\delta(l_i)$ is the degree of vertex l_i.
Again, $b_{j,l}$ can be defined similarly for $j = 1..m$ and $l = 1..m$.

This can be done for each pair (i, k) giving a $n \times n$-matrix for L and a $m \times m$-matrix for U, where each element is an estimation how many crossings are caused by placing vertex l_i and l_k. The idea now is to use greedy insert to determine which vertex should be put where based on the lowest assessment number for all open spots. If all assessment numbers are equal for all remaining vertices, try to fill in other spots first. After filling as many spots as possible, order the remaining vertices in arbitrary order.

3.1.3 The Branch-and-bound Method

This method for bipartite graphs, introduced by Valls et al. [17], differs quite a bit from the others in that it works with a decision tree as structure: the aim is to represent each possible permutation (and thus ordering) as a node in the graph. Each node is then given a value based on the number of crossings in the ordering. The optimal result is then the node with the lowest crossings associated.

Let $G = (L, U, E)$ a bipartite graph. The general structure of the tree is an empty root, splitting into $|L|!$ branches, one for each permutation of L. Note that this method does not require one side fixed, but it is possible, eliminating all other branches.
Then, each child node becomes associated with one fixed node, starting at the first position, then the second, etc. This gives a number of branches equal to the number of unfixed vertices. We assume that these branches are all ordered, such that two branches are ordered based on the fixed node. For example, if one branch fixed vertex u in the i'th place, and the second branch vertex v, the order of the branches must be equal to the order of u and v w.r.t. each other *in the original order of* U. This recursively defines the whole tree. See Figure 5 for a partial example with $L = \{1, 2, 3\}, U = \{4, 5, \ldots\}$. Again, note that if we already have a fixed order of L, we can ignore the other branches, they are included for clarity of the construction.

As for the cost of a node S, it depends on three values:

- Crossings between two edges incident with vertices in U that are already fixed (in the current node), call this k_1.

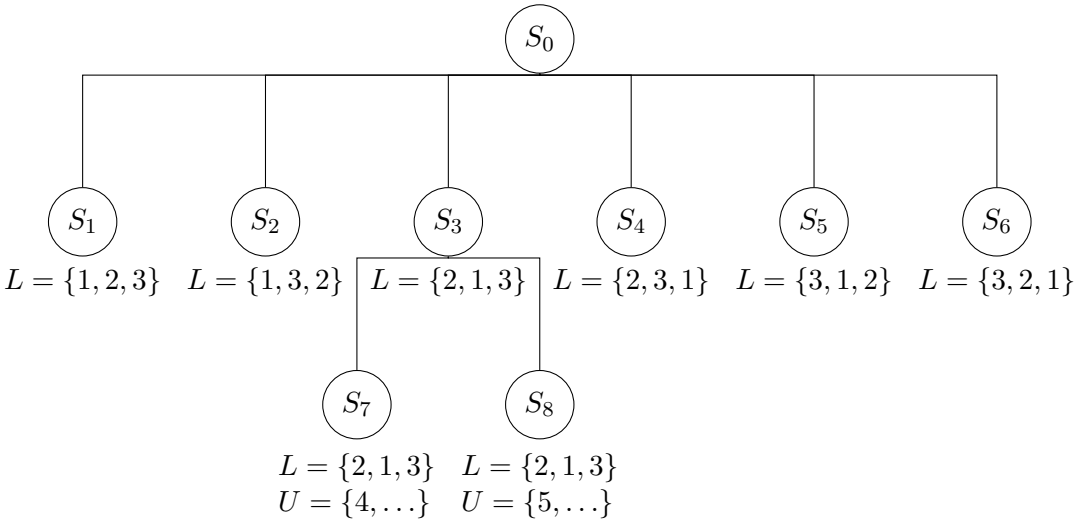


Figure 5. *An example of Branch-and-bound.*

- Crossings between edges incident with one fixed and one unfixed vertex in U, call this k_2.
- Crossings between edges incident with two unfixed vertices in U, call this k_3.

Giving the result $\text{cost}(S) = k_1 + k_2 + k_3$. Note that only k_3 is variant under the order of the remaining vertices, k_1 and k_2 not. The proposed algorithm then calculates a lower bound of k_3, that we will call k_4, resulting in a *lower bound* of the cost.

To find the lower bound, we use an auxiliary directed graph H on the remaining, unfixed vertices. For H, define the edges (u, v) with weights $K(u, v)$ for each pair $u, v \in H$. We then introduce the notation $K(i, j), i, j \in U$ as the number of crossings between arcs incident with i and arcs incident with j if vertex i comes before vertex j in the (proposed) ordering of U. Note that this also includes (v, u), so for each pair of nodes, we get two connecting edges with weights $K(u, v)$ and $K(v, u)$ respectively. For each such pair, we take the minimum of the two, giving an estimated lower bound of crossings between the vertices in H (i.e. the unfixed vertices in the original graph G), call this k_4. Formally, we say for $H = (V, E)$, that

$$k_4 = \sum_{(u,v),(v,u) \in E} \min(K(u, v), K(v, u))$$

Then we can define the lower bound of the node S as $k_1 + k_2 + k_4$. The result of the algorithm would then be, for each layer in the tree, the node with the lowest lower bound. In particular, this method finds the optimal result, if certain conditions apply (see [17, section 5]).

3.1.4 The Barycenter Method

The barycenter method, first introduced by Sugiyama et al. [16], uses the *barycenter* of a graph. Intuitively, a barycenter is the center of mass of a body. In the case of a graph the center of mass can be seen as the average *weight* of a vertex.

Given a bipartite graph $G = (L, U, E)$ and an ordering l_0 on L, this method finds the barycenter of a vertex in the unfixed side by taking the average of the x-coordinates of the neighbours. So for a vertex u in U, we take the average of the x-coordinates of all neighbouring vertices $v \in L$. For this, we need to first define the x-coordinate of v. This can be done by taking the order of v in the ordering l_0. Let N_u be all the neighbours of u, so $N_u = \{v \in L \ : \ (u, v) \in E\}$ for all $u \in U$, ordered w.r.t. l_0. We can then define the barycenter of u as follows:

$$\text{Barycenter}(u) = BC(u) = \frac{1}{|N_u|} \sum_{v \in N_u} l_0(v) \tag{3}$$

Now that we have the barycenter of each vertex, we can find an ordering u_0 of U by sorting all vertices in ascending order of barycenters. In case of ties (equal barycenter), there are several options, which we will discuss in Section 3.2;

3.1.5 The Median Method

The median method, first given by Eades and Wormald [5], is very similarly to the barycenter method: rather than the average of the neighbours, take the median. Do note that this ensures that the median is always the same as one of its neighbours, except for vertices without neighbours. Again, let N_u be the (ordered) neighbours. Assume these are v_i with $1 \leq i \leq n$ with $n = |N_u|$. Then we define:

$$\text{Median}(u) = med(u) = l_0\left(v_{\lfloor \frac{n}{2} \rfloor}\right) \tag{4}$$

For empty N_u, i.e. U has no neighbours, we choose $med(u) = 0$.
Sort this on median, with the caveat that vertices with *odd* degree come before vertices with an *even* degree.
Similarly to the barycenter method, this can also result in ties between vertices with the same median, for different methods, see Section 3.2.

3.1.6 Conclusion on the heuristic methods.

Now that we have outlined our choices, we will give our reasoning for choosing from the methods we explained above. The reasoning for the heuristics is largely based on Jünger and Mutzel [9], which showcases all the methods

above with their respective performances, combined with the want/need for a guarantee for the upper bound.

First, based on from their "relative optimality" [9, Figures 2 and 4], and somewhat of their runtime [9, Figures 3 and 5], we decided to at least implement the barycenter method. The problem with the barycenter method is that their is no known *constant* approximation of the result, although there is a $O(\sqrt{n})$ approximation as proven by Eades and Wormald [5, Section 4.2, Theorem 4]. However, we do not think this is a good enough guarantee, so we opted to include the median heuristic as well, since we thought it had a upper bound of three times the optimum, as proven by Eades and Wormald [5, Section 4.1, Theorem 3]. However, after being stuck on replicating the proof for a while, one of my advisors actually found that the original proof had a mistake. We will still give this proof here for completeness' sake, and where we found a mistake.

Theorem 3.1. *For all two-layered networks $G = (L, U, E)$ and orderings l_0 on L,* $\operatorname{cross}(G, l_0) \leq 3 \operatorname{opt}(G, l_0)$.

Proof. (Proposed proof[1]) First, let us introduce terminology: Let $\chi(t) = \frac{t(t-1)}{2}$; if t is a nonnegative integer, $\chi(t)$ is the number of subsets of cardinality 2 of a set of cardinality t.
The "crossings array" $C = |U| \times |U|$ is a matrix where the uv-th entry denote the number of crossings between arcs incident with u with arcs incident with v when $u_0(u) < u_0(v)$, with u_0 an ordering on U. Formally, for $u, v \in U$, $u \neq v$,

$$c_{uv} = |\{(t, u), (w, v) \in E : l_0(u) < l_0(v)\}|$$

and $c_{uu} = 0$.
Additionally, define $\tau_{uv} = 1$ if $med(u) = med(v)$, and $\tau_{uv} = 0$ otherwise. Finally, remember that N_u denotes all neighbours of vertex u. The degree of u is denoted with $d_u = |N_u|$. We then define ι_{uv} as $|N_u \cap N_v|$.
Before proving the theorem above, we will first prove the next lemma.

Lemma 3.2. *Suppose that $u, v \in U$ and $med(u) \leq med(v)$.*

(a) If d_u and d_v are both even, then

$$4c_{uv} \leq 3d_u d_v - 2d_u - 8\chi\left(\frac{\iota_{uv}}{2}\right),$$
$$4c_{vu} \geq d_u d_v + 2d_u + 8\chi\left(\frac{\iota_{uv}}{2}\right) - (1 + 2\iota_{uv})\tau_{uv}.$$

[1]For lack of better terminology, it is still called "Theorem", which implies something that can be proven, even though this proof is incorrect. Hence this annotation.

(b) If d_u is even and d_v is odd, then

$$4c_{uv} \leq 3d_u d_v - d_u - 8\chi\left(\frac{\iota_{uv}}{2}\right),$$
$$4c_{vu} \geq d_u d_v + d_u + 8\chi\left(\frac{\iota_{uv}}{2}\right) - (1 + 2\iota_{uv})\tau_{uv}.$$

(c) If d_u is odd and d_v is even, then

$$4c_{uv} \leq 3d_u d_v - 2d_u - d_v - 2 - 8\chi\left(\frac{\iota_{uv}}{2}\right),$$
$$4c_{vu} \geq d_u d_v + 2d_u + d_v + 2 + 8\chi\left(\frac{\iota_{uv}}{2}\right) - (1 + 2\iota_{uv})\tau_{uv}.$$

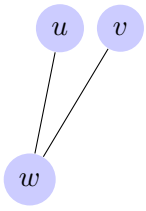
(d) If d_u and d_v are both odd, then

$$4c_{uv} \leq 3d_u d_v - d_u - d_v - 1 - 8\chi\left(\frac{\iota_{uv}}{2}\right),$$
$$4c_{vu} \geq d_u d_v + d_u + d_v + 1 + 8\chi\left(\frac{\iota_{uv}}{2}\right) - (1 + 2\iota_{uv})\tau_{uv}.$$

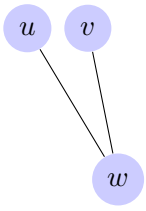
Proof. (Proposed proof) Divide the edges incident with two vertices u and v in four groups:

$$\begin{aligned}
\alpha &= \{uw : l_0(w) \leq med(u)\},\\
\beta &= \{vw : l_0(w) \geq med(v)\},\\
\gamma &= \{vw : l_0(w) < med(v)\},\\
\delta &= \{uw : l_0(w) > med(u)\},\\
E &= \{uw \in \alpha \ : \ vw \in \gamma\},\\
F &= \{vw \in \beta \ : \ uw \in \delta\},\\
G &= \{vw \in \gamma \ : \ uw \in \delta\}.
\end{aligned}$$

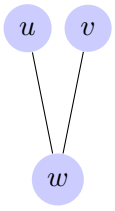
and denote $a = |\alpha|, b = |\beta|, c = |\gamma|, d = |\delta|, e = |E|, f = |F|$ and $g = |G|$. See Figure 6 for examples for cases in e, f and g.



(a) *Drawing of an element for e.*



(b) *Drawing of an element for f.*



(c) *Drawing of an element for g.*

Figure 6. *Showing examples for given sets e, f, g.*

We see that, if $med(u) = u_0(u) \leq med(v) = u_0(v)$, then edges in α cannot cross edges in β, based on the condition of the groups. For crossings between edges in α and γ is at most $a(c - g) - \chi(e + 1)$.

Intuitively, this can be seen as follows: ac is the maximum number of crossings possible between α and γ. Then ag is subtracted, since g denotes the number of edges that cannot cross with an edge in γ, giving a total of ag less possible crossings. That these g edges cannot cause crossings can be seen by taking Figure 6c and adding an edge from α. Per definition, w' lies left of u, hence cannot cross the edge (w, u). Lastly, note that elements that count towards e are vertices w such that both (u, w) and (v, w) are existing edges. In particular, note that, if $med(w) < med(w')$, the pairs (u, w) and (v, w') do not cross. If $med(w) > med(w')$, the pair (u, w') and (v, w) do not cross. This holds for each distinct pair of w, w'. (Again, this can be seen by adding a vertex w' to Figure 6a and checking the pairs.) This means that each pair reduces the total number of possible crossings by one, giving $\chi(e)$. Additionally, for $w = w'$, the edges (u, w) and (v, w), giving an additional $2e$ less crossings. This gives the total of $\chi(e) + e$. We then note that $\chi(e + 1) = \frac{(e+1)e}{2} = \frac{e(e-1)}{2} + \frac{2e}{2} = \chi(e) + e$, which gives the desired term, so we can agree that the total number of crossings between elements of α and γ is at most $a(c - g) - \chi(e + 1)$, between edges in α and γ.

A similar bound for crossings between edges in β and edges in δ can be found, namely $b(d - g) - \chi(f + 1)$. Remains crossings between edges in γ with edges in δ. This is at most $cd - \chi(g + 1)$.

Combining all these terms combined then give the following:

$$c_{uv} \leq a(c - g) - \chi(e + 1) + b(d - g) - \chi(f + 1) + cd - \chi(g + 1) \quad (5)$$

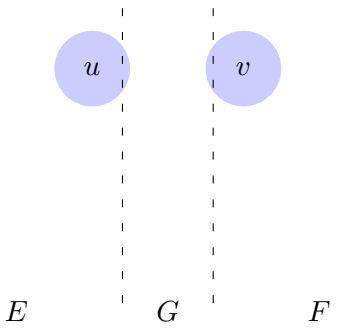


Figure 7. *Visualisation of* $\iota_{uv} = e + f + g + \tau_{uv}$.

For the following steps we will use that $\iota_{uv} = e + f + g + \tau_{uv}$. This follows from the fact that elements counted towards e, f and g have edges incident with u and v, hence they are neighbour of both u and v. Finally, if u and v have the same median, $\tau_{uv} = 1$, else 0. See Figure 7 for a visualization. First of, it is clear that e, f and g give the desired outcome if the medians of u and v are different.

If the medians of u and v are the same, the first we see is that $g = 0$ since

there can be no values in-between. Furthermore, per the definition of the median it is a neighbour of both u and v. This means that the number of vertices not counted in e and f is exactly 1, namely the shared neighbour. This one vertex is then counted with τ_{uv}, proving that $\iota_{uv} = e + f + g + \tau_{uv}$. The original proof then takes the following steps:

$$c_{uv} \leq ac + bd + cd - eg - fg - \frac{e^2 + f^2 + g^2}{2} \tag{6}$$

$$\leq ac + bd + cd - \frac{\iota_{uv}^2 + \iota_{uv} - \tau_{uv} - 2ef}{2} \tag{7}$$

$$\leq ac + bd + cd - \frac{(\iota_{uv} - \tau_{uv}^2) + 2\iota_{uv} - 2\tau_{uv}}{4} \tag{8}$$

concluding that

$$c_{uv} \leq ac + bd + cd - 2\chi\left(\frac{\iota_{uv}}{2}\right) \tag{9}$$

This is where we were unable to reproduce the step from Equation (6) to Equation (7) that the original proof did, later realizing that it contained a mistake. We will show why it cannot be true, starting from Equation (5) and using that $a \geq e, b \geq f$ and $\chi(t+1) \geq \frac{t^2}{2}$.

$$\begin{aligned} c_{uv} &\leq a(c-g) - \chi(e+1) + b(d-g) - \chi(f+1) + cd - \chi(g+1) \\ &\leq ac - eg - \frac{e^2}{2} + bd - fg - \frac{f^2}{2} + cd - \frac{g^2}{2} \end{aligned}$$

This gives us Equation (6). We will now prove by contradiction that the second step cannot be obtained from the first step. Assume

$$ac + bd + cd - eg - fg - \frac{e^2 + f^2 + g^2}{2} \leq ac + bd + cd - \frac{\iota_{uv}^2 + \iota_{uv} - \tau_{uv} - 2ef}{2}$$

Now rewrite, and use $\iota_{uv} = e + f + g + \tau_{uv}$, so $\iota_{uv}^2 = (e + f + g + \tau_{uv})^2$. This gives

$$\begin{aligned} \frac{e^2 + f^2 + g^2 + 2eg + 2fg}{2} &\geq \frac{\iota_{uv}^2 + \iota_{uv} - \tau_{uv} - 2ef}{2} \\ \frac{\iota_{uv}^2 - 2ef - 2\iota_{uv}\tau_{uv} + \tau_{uv}^2}{2} &\geq \frac{\iota_{uv}^2 + \iota_{uv} - \tau_{uv} - 2ef}{2} \\ \tau_{uv}^2 - 2\iota_{uv}\tau_{uv} &\geq \iota_{uv} - \tau_{uv} \\ \tau_{uv}^2 + \tau_{uv} &\geq \iota_{uv} + 2\iota_{uv}\tau_{uv}. \end{aligned}$$

Note that $\iota_{uv} = e + f + g + \tau_{uv}$, hence $\iota_{uv} \geq \tau_{uv}$. From this, we know that $\tau_{uv}^2 < 2\iota_{uv}\tau_{uv}$, again using that $\iota_{uv} \geq \tau_{uv}$. Based on this, the steps are not true,

For now assume that the steps above are correct, then we will complete the original proof.
Now, after looking at $u_0(u) \leq u_0(v)$, we will now look at the case if u and v are placed such that $u_0(u) > u_0(v)$, with u_0 the ordering induced by the median method. Then, the edges in α must cross edges in β, except for one crossing if $med(u) = med(v)$. This gives:

$$c_{vu} \geq ab + \chi(e) + \chi(f) + \chi(g) + ef + fg - \tau_{uv}$$

and, analyzing as for Equation (9)[2]

$$c_{vu} \geq ab + 2\chi\left(\frac{\iota_{uv}}{2}\right) - \frac{(1 + 2\iota_{uv})\tau_{uv}}{4}. \tag{10}$$

We will now prove part (a) of Lemma 3.2, the case with both d_u and d_v even. That d_u and d_v are both even implies that $a = d = \frac{d_u}{2}$ and $b = c+2 = \frac{d_v+2}{2}$. This can be seen using the definition of $med(u)$ and $med(v)$. Since these are even, the median is at position $\frac{d_u}{2}$. We can then see that half of the total is smaller or equal, and the the other half larger then the median, giving $a = d = \frac{d_u}{2}$. Similarly, $\frac{d_v}{2} - 1$ vertices are *strictly* smaller in terms of median, and $\frac{d_u}{2} + 1$ larger of equal, giving $b = c + 2 = \frac{d_v+2}{2}$ as claimed.
Applying these claims to Equation (9),

$$\begin{aligned}
c_{uv} &\leq ac + bd + cd - 2\chi\left(\frac{\iota_{uv}}{2}\right) \\
&\leq cd + ab + cd - 2\chi\left(\frac{\iota_{uv}}{2}\right) \\
&\leq ab + 2(b-2)a - 2\chi\left(\frac{\iota_{uv}}{2}\right) \\
&\leq ab + 2bd - 4a - 2\chi\left(\frac{\iota_{uv}}{2}\right) \\
&\leq ab + 2ab - 4a - 2\chi\left(\frac{\iota_{uv}}{2}\right) \\
&\leq 3ab - 4a - 2\chi\left(\frac{\iota_{uv}}{2}\right)
\end{aligned}$$

resulting in (a) as in Lemma 3.2:

$$c_{uv} \leq 3ab - 4a - 2\chi\left(\frac{\iota_{uv}}{2}\right) \leq \frac{3}{4}d_u(d_v + 2) - 2d_u - \chi\left(\frac{\iota_{uv}}{2}\right).$$

[2]There are no intermediate steps in the original proof, to support this, and we have not had time to work out the steps and check this claim, since it (supposedly) is similar to the steps in **??**.

Similar for Equation (10)

$$
\begin{aligned}
c_{vu} &\geq ab + 2\chi\left(\frac{\iota_{uv}}{2}\right) - \frac{(1+2\iota_{uv})\tau_{uv}}{4} \\
c_{vu} &\geq \frac{d_u}{2}\frac{d_v+2}{2} + 2\chi\left(\frac{\iota_{uv}}{2}\right) - \frac{(1+2\iota_{uv})\tau_{uv}}{4} \\
c_{vu} &\geq \frac{d_u(d_v+2)}{4} + 2\chi\left(\frac{\iota_{uv}}{2}\right) - \frac{(1+2\iota_{uv})\tau_{uv}}{4},
\end{aligned}
$$

which is equal to the claim in Lemma 3.2, divided by four.
These two statements combined prove part (a) of the lemma, parts (b)-(d) can be proven in a similar manner. ■

To prove Theorem 3.1, we use two lemmas from Eades and Kelly [4]:

Lemma 3.3. *[4, Lemma 2]* $\text{cross}(G, l_0, u_0) = \sum_{u_0(u)<u_0(v)} c_{uv}$.

Lemma 3.4. *[4, Lemma 3]* $\text{opt}(G, l_0) \geq \sum_{u,v} \min(c_{uv}, c_{vu})$*, where the sum is over all unordered pairs* $\{u, v\}$.

These lemmas imply that, if we can establish that there is a uniform bound B such that whenever $u_0(u) < u_0(v)$ we have

$$c_{uv} \leq B c_{vu}$$

then it follows that

$$\mathit{cross}(G, l_0, u_0) \leq B\ \mathit{opt}(G, l_0)$$

by summing the inequality over all pairs u, v with $u_0(u) < u_0(v)$.
The left side follows directly, the right side is a bit more work.

$$
\begin{aligned}
\text{cross}(G, l_0, u_0) &= \sum_{u_0(u)<u_0(v)} c_{uv} && (11)\\
&\leq \sum_{u_0(u)<u_0(v)} \min(c_{uv}, B c_{vu}) && (12)\\
&\leq B \sum_{u_0(u)<u_0(v)} \min(c_{uv}, c_{vu}) && (13)\\
&\leq B\,\text{opt}(G, l_0). && (14)
\end{aligned}
$$

Here we used that each term in Equation (12) is larger or equal then the same terms in : for terms in Equation (12), we know that $c_{uv} \leq B c_{vu}$. Then, for Section 3.1.6, if $c_{uv} \leq c_{vu}$, the term becomes $B c_{uv}$, which is larger then in Equation (12). Similarly, if $c_{uv} > c_{vu}$ the term becomes $B c_{vu}$, which is equal to the term in Equation (12).

We see that, for the parity of the degree of u odd and any parity of the degree of v, using

$$\iota \leq \min\{d_u, d_v\},$$
$$8\chi\left(\frac{\iota_{uv}}{2}\right) \geq -1,$$

that Lemma 3.2 directly implies the bound $c_{uv} \leq 3c_{vu}$, by taking the respective inequalities and multiplying c_{vu} by 3.
For d_u even and d_v odd, we will use Lemma 3.2 and the following inequalities

$$\iota \leq \min\{d_u, d_v\},$$
$$8\chi\left(\frac{\iota_{uv}}{2}\right) \geq -1,$$

to show that $c_{uv} \leq 3c_{vu}$.
Then, Theorem 3.1 follows directly from Lemmas 3.3 and 3.4 with $B = 3$.
It is clear that all cases with $u_0(u) < u_0(v)$ fall under Lemma 3.2, but we will mention a somewhat special case, with d_u even and d_v odd.

we can assume that $\tau_{uv} = 0$, because if not, the median heuristic enforces order based on parity of the degree, odd before even, which would give $u_0(u) > u_0(v)$. This is a contradiction since we are only looking at pairs u, v such that $u_0(u) < u_0(v)$. So we conclude that u and v cannot have the same median, i.e. $\tau_{uv} = 0$. Then (b) of Lemma 3.2 and the proof follows directly, using $\tau_{uv} = 0$.

We will write out the case with d_u, d_v both even, all other cases are similar.

$$4c_{uv} \leq 3d_u d_v - 2d_u - 8\chi\left(\frac{\iota_{uv}}{2}\right) \leq 3d_u d_v - 2d_u + 1 \leq 3d_u d_v - 3$$

Here we used that $8\chi\left(\frac{\iota_{uv}}{2}\right) \geq -1$ and that $d_u \geq 2$ (since it is even), resulting in $-2d_u + 1 \leq -3$.
We also get:

$$4c_{vu} \geq d_u d_v + 2d_u + 8\chi\left(\frac{\iota_{uv}}{2}\right) - (1 + 2\iota_{uv})\tau_{uv} \geq d_u d_v + 2_u - 1$$

Here we used that $2d_u + 8\chi\left(\frac{\iota_{uv}}{2}\right) - \iota_{uv} \geq 0$ because $\iota \leq \min\{d_u, d_v\}$, giving $d_u - \iota_{_{uv \geq 0}}$. Similarly, we know $d_u \geq 2$, resulting in the statement above.
Now we see the desired inequality, concluding the proof:

$$4c_{uv} \leq 3d_u d_v - 3 \leq 3d_u d_v + 6d_u - 3 \leq 12c_{vu}$$
$$c_{uv} \leq 3c_{vu}$$

So we have found a bound $B = 3$. Now, using Lemma 3.3 and **??**, we can directly conclude that $\text{cross}(G, l_0, u_0) \leq 3\,\text{opt}(G, l_0)$, as desired. □

Again, because of the mistake we found in Equations (6) to (7), this does not guarantee the constant approximation of the median heuristic that was claimed and we would have wanted. However, it might be possible to find bounds similar to Equations (9) and (10), resulting in a theorem with a different factor than 3 as in Theorem 3.1.

3.2 Tiebreaking methods for Barycenter and Median methods

This section will showcase different ways to handle the aformentioned ties occuring in the Barycenter and Median methods. This is based on a test by Poranen and Mäkinen [14], complemented by the results of our implemented methods ran on the public instances of the PACE challenge, which can be seen in Appendix A.

For the following, with the "original order" we mean the order of the vertices as they originally were given. So if we are to use the median method to determine the order of a list of vertices $\{v_0, v_1, ..., v_k\}$, this is the original order.
Poranen and Mäkinen tested several methods and combinations of methods. The "building blocks" are the following six methods:

- **Standard:** the standard procedure, i.e. no tiebreaking, preserve the order in the original.
- **Reverse:** this reverses the order of the tied vertices w.r.t. the original order.
- **Random:** this chooses a random permutation of all tied vertices.
- **Optimal:** a procedure that find the optimal arrangement of the tied vertices, i.e. the permutation that minimizes the number of crossings between edges incident with tied vertices.
- **Variance:** a formula to determine the "orientation" of a vertex, either left- or right-oriented with the idea that it is better, in terms of crossings, to place a left oriented vertex to the left and a right one to the right. Assume that the vertices $u_0, u_1, .., u_i$ are the vertices to the left of $BC(u)$ under ordering l_0, and $u_{i+1}, u_{i+2}, ..., u_k$ the ones to the right. Then the left orientation $v_l(u)$ and right orientation $v_r(u)$ of a vertex u are defined as follows

$$v_l(u) = (l_0(u_1) - BC(l_0(u)))^2 + (l_0(u_2) - BC(l_0(u)))^2 + \dots + (l_0(u_i) - BC(l_0(u)))^2$$

and

$$v_r(u) = (l_0(u_{i+1}) - BC(l_0(u)))^2 + (l_0(u_{i+2}) - BC(l_0(u)))^2 + \ldots + (l_0(u_k) - BC(l_0(u)))^2.$$

If $v_l(u) > v_r(u)$, then u is said to be left oriented with value $v_l(u) - v_r(u)$. Otherwise, u is said to be right oriented with value $v_r(u) - v_l(u)$. For the median heuristic, v_l and v_r are obtained by using med's instead of BC's. In the variance heuristic the vertices with identical barycenters (resp. medians) are ordered according to their orientation values. If orientation values are identical, then their relative order is kept unchanged.

A combination of these methods will be denoted with something like "standard-optimal" (on either barycenter of median), meaning that we apply the "optimal" tiebreaking method to the "standard" (barycenter/median) heuristic method. The authors also used a combination of the tiebreaking methods above with the other heuristic, i.e. "bc+std" means that the barycenter heuristic is applied as a tiebreaking method to the "standard" median heuristic, sorting tied vertices w.r.t. to the median method on their barycenter.

Poranen and Mäkinen conclude that the following tiebreaking methods are the most optimal: for the barycenter heuristic, the reverse, random, and reverse with optimal method. In the case of the median heuristic, the best tie-breaking method was the reversed barycenter method.

Because of this, we wanted to implement all tiebreaking methods and try to verify those findings. In the end, we did not manage to implement the reverse optimal tiebreaking for the barycenter heuristic due to performance issues (calculating the score for all possible permutation was simply too slow).
Additionally, we realised, after submitting the project for the PACE challenge, that we misinterpreted the "Reverse + Barycenter on Median": rather than running the barycenter method on the median and reversing ties *w.r.t. the barycenter method*, we implemented it to sort on barycenter in descending order. The latter option reverses *all* rather than just the tied vertices. Because we already submitted the project, we could not really change it, although we did do a cursory test to see if it differs (results are included in the `.csv` file in the repository [8]). We tested by running both implementations on the same 113 test cases, comparing the crossingcount per problem. The result was only a handful of cases where the result was different, although, as expected, the "fixed" implementation is the favored version.
As for the rest, to verify the findings of Poranen and Mäkinen, we ran all these methods on the public instances of the PACE challenge, which are the 100 public instances and the tiny set, found on their website, to see if any of these methods excel on specific test cases, which would make these

methods more useful in this specific use-case. The results can be seen in Appendix A. We made a more grouped and compact overview which can be seen in Table 1. We can clearly see that the barycenter method outperforms the median by a large margin in terms of number of times those methods give the best answer, although there does not seem to be a substantial difference between the different tiebreaking methods for the barycenter heuristic themselves. Looking at and comparing the problems where different tiebreaking methods are the best does not give any easily identifiable result: no visible relation w.r.t. density, size or any other problems property give any reason as to why that method is best for those cases. Because of this, for the final program we chose to simply run all methods and chose the best answer.

Method Name	Barycenter:	Median:	Random on BC:	Reverse + BC on Median:	Reverse on BC:
#Best answer.	64	9	56	6	64
Avg density when correct.	0.097469156	0.311314174	0.110542286	0.233637927	0.097557675
#Best answer (large set).	51	4	43	4	51
Avg density (large set).	0.002641032	0.000456891	0.002024062	0.000456891	0.002752114
#Best answer (tiny set).	13	5	13	2	13
Avg density (tiny set).	0.469487179	0.56	0.469487179	0.7	0.469487179

Table 1. *A table with results of comparing the used methods.*

4 The implemented algorithm

Now that we have gone over the different options we considered, this section will go over the process of writing the actual program, found at [8], starting from the given input and ending with the desired result. This includes motivation for the chosen programming language, parsing of the input, explanation of the datastructure, heuristic methods, tiebreaking methods, formatting the output, etc.

4.1 Programming Language

Even though the tester/verifier library provided by PACE is given in PYTHON, we opted for C# instead. The main reason is familiarity, but it also helps that C# is a more performance-oriented language. Additionally, it turned out that LINQ (built-in query language for C#, similarly structured as SQL) is quite helpful for applying functions, like group, filter and sort, to large data sets.

4.2 Parsing the input

The input, as given on io-page on the PACE-site contains a *problem descriptor line* containing "n_0, n_1, and M", where

- n_0 is the number of vertices in bipartition L (the fixed side);
- n_1 is the number of vertices in bipartition U (the free partition);
- M the number of edges to follow.

This represents a bipartite graph $G = (L, U, E)$ with $|L| = n_0, |U| = n_1, |E| = m$.
The vertices in L are numbered 1 to n_0 and the vertices in U from $n_0 + 1$ to $n_0 + n_1$. Each edge is given as a pair of integers (u, v) with $1 \leq u \leq n_0$ and $n_0 + 1 \leq v \leq n_1$. We will refer to this given number as `u.order` for a given vertex u. Keep in mind that this is/can be different from the *assigned order* in the final result.

These lines are given from the console, so they are read using a readline command, and parsed into the datastructure as described below.

4.3 Datastructures

There are basically two datastructures that are consistently used. One to store the original problem and one to store the solution.

First, to store the original problem, we store n_0 and n_1 and an array of size m containing all edges. We also store the neighbours of nodes in U as

an array of lists of nodes `Neighbours` such that the u-th entry corresponds to the list of neighbours N_u of the vertex u for all $u \in \{1...n_1\}$. We make sure that

- `Neighbours` is ordered such that `Neighbours[i]` contains the neighbours of node i in U;
- the neighbours are ordered w.r.t. u. So for all $v, w \in N_u$, we have that $l_0(v) < l_0(w)$.

This way, given a vertex $u \in U$, we can use the order of u to find its neighbours. This is done by taking `u.order` and subtracting $n_0 + 1$. This gives an index i such that $0 \leq i \leq n_1 - 1$ such that `Neighbours[`i`]` gives the neighbours of u. This list can be made in conjunction with storing the edges by storing the first point of the edge to the list for the second. For example, the edge (u, v) with $u \in L, v \in U$ is stored in `Neighbours[v.order-|L|-1]`, as described above.
Each edge is stored as a tuple of integers corresponding to the labels of the vertices as read from the input, so edge `u.order,v.order`. An edge is stored as a pair of nodes. A node is given a label `order` which corresponds to the given number in the original problem. This way, we are able to identify which vertex is placed at what position.
The solution is stored as an array of size n_1, containing a permutation of the nodes numbered $n_0 + 1$ to $n_0 + n_1$. The label on the vertex is used to output the found order. For this, loop over the result and output the label of each entry, which corresponds to the "original vertex".

4.4 Implemented Heuristic Methods

As we already discussed in Section 3.1.6, we opted for the median heuristic and the barycenter. These can be found in the files with the same name. The implementation of these two is pretty straightforward. Using the list of neighbours, one can calculate the median and barycenter easily using the definition give in Equation (4) and Equation (3) respectively. Once we have the median (or barycenter) of all vertices, we can use LINQ to sort them, giving the resulting order.

4.5 Implemented Tiebreaking Methods

For this section, note that *each tiebreaking method* works on *groups of ties*, that is, all vertices that would be tied under the standard method (same barycenter or same median & degree), leaving the rest of the vertices in the same order.
The implemented tiebreaking methods are, for the median method Reverse + Barycenter and the standard median methods.

The Reverse + Barycenter calculates the barycenter for each vertex, and sorts them in reverse order (i.e. descending) w.r.t. the original order (using `u.order`).
The Standard does not do any additional tiebreaking. This is the bare-bones median method as discussed in 3.1.5.

For the barycenter method we have the standard, Random and Reverse tiebreaking methods.
Again, Standard dos not do any tiebreaking, it follows the method as discussed in 3.1.4.
The Reverse method simply reverses (w.r.t. the original order, using `u.order`) each group of ties and returns the concatenated result.
The Random method uses the so-called Fisher-Yates shuffle [7]. This algorithm shuffles by elements by swapping them following this psuedocode, given an array `A` of n elements:

```
for i from n−1 down to 1 do
    j ← random integer such that 0 ≤ j ≤ i
    exchange A[j] and A[i]
```

4.6 General helperfunctions

Here we will touch upon some of the things we implemented that do not influence the solution, like functions for outputting, logging and tests.

There are two types of output accepted, either via `stdin/stdout` or a file. Because of this, there are two functions that output the result to either console or file.

For logging, considering the program has to be called from a shell script, we cannot use the console for debugging purposes because that would be considered "giving output". So in order to have some information (like the value of a variable, number of loops, etc.) while the program is running, we implemented a simple logging system that outputs a value to a specified file. This was the main method used to gather the data represented in Appendix A.

4.7 Testing Verification

At the start we wanted to be able to make our own tests, so we implemented a *graph generator*. This generator constructs a bipartite graph $G = (L, U, E)$. For this, it takes two bounds for L and U and a percentage, which we see as 100 times the density. We can then take random numbers between the bounds for L and U, giving the size of L and U. Then, for each possible edge $(u, v) \in L \times U$, take a random number and see if is smaller then the percentage given. This way, we construct a graph with density

times a hundred roughly equal to the given percentage.

The idea was to use tests to check functions and assumptions on the generated tests, for example whether the neighbours are sorted, the edges are given in ascending order, etc. but we ended up using the provided instances instead, mostly because it felt that the investment for making tests could not be justified if we could be doing other things, like implementing tiebreaking methods and (more) heuristic methods.

4.8 Crossing counters

One of the biggest challenges was implementing the crossing counter function in a way that is was fast enough to be usable while running. This is a requirement in order to be able to compare the optimal method mid-run. So, after running the median method and the barycenter method, we compare the solutions and take the best one of the two. Here, "best" of course refers to the solution that has the least amount of crossings. This requires a method to efficiently and quickly calculate these crossings.

The most straightforward implementation is in $O(|E|^2)$, which checks for each pair of edges if they crossed or not, but his turned out to be too slow when running graphs containing hundreds of thousands of edges.

A more efficient approach is to use the list of neighbours to find edges which can cause crossings, rather than checking all edges. We know that the crossings are only between edges with the endpoints reversed. This also means that, given an edge $e = (u, v)$ with $u \in U, v \in L$, we can limit our search to the edges $e' = (u', v'), u' \in U, v' \in L$ such that $u' < u \vee v' > v$. We can do this by filtering the neighbours of all vertices $u' < u$ with the condition $v' > v$. This enforces that, for each filtered neighbour v' of u', (u, v) crosses (u', v') The number of crossings is then the size of that list. resulting in the following psuedocode:

```
crossingTotal = 0
for each u ∈ U
    for each v ∈ N_u
        for each u' < u do
        result ← filter N_u' on v' > v
        crossingTotal += |result|
```

Listing 1. *Psuedocode of the crossingcounter using filtering of vertices.*

Still, this had to run around five times (once for each method used), which took more then 10 minutes combined on some of the larger test cases, so this was still not practical.

For the next improvement, we had the idea to use binary search on N_u to find the first vertex v' that was smaller then v. Note that "larger then v" also works, but then the number of neighbours would be $|N_u| - 1 - \texttt{indexOf}(v')$ rather then $\texttt{indexOf}(v')$. (Note that this works because

the vertex list N_u is sorted.) Furthermore, we can also tighten the lower bound for the binary search for each subsequent neighbour. We can do this because each subsequent neighbour is further to the right than the last, so the first vertex smaller then that v must also be further to the right then the previous result. In other words, say we find index i for neighbour w. The resulting index i' for the next neighbour w' satisfies $i \leq i'$, so we can tighten the lower bound of our search to i, or the previous result. If we move on the the next vertex u, we reset the lower bound.
So, what we do is use binary search to find the first index that is smaller then the given vertex. Because the neighbours are sorted, we can tighten the lower bound after each search.
See the psuedocode below for a more structured overview.

```
crossingTotal = 0
for each u in U
    N_u ← Neighbours[u.order-1-|L|]
    for each u' > u do
        N_u' ← Neighbours[u'.order-1-|L|]
        lowerBound = 1
        upperBound = |N_u'|
        for each v in N_u
            result ← BinarySearch(lowerBound,
                                  upperBound, v, N_u')
            lowerBound = result
            crossingTotal += result
```

Listing 2. *Psuedocode of crossingcounter using BinarySearch.*

After this, we were able to run the 100 public instances in about half an hour, so the performance had definitely improved enough that it was acceptable.

4.9 Running the program

Some of the other things that made this project fairly difficult are (1) that it has to run in a Linux environment and (2) if the program is not in python, it contain a executable, for example a shell script. In the end, we included several script files, for Windows and Linux, for manual testing and for testing using the dedicated tester from PACE[3]. The manual testers use a local instance of the test cases, where as the supplied tester calls the solver with each test case separately.

To run the program yourself, using the manual solver, make sure you have the test cases included in the IOFiles folder (more detailed instructions in the included README). If you use the tester from PACE, simply

[3]A python package, found on their website and in the default pip package manager.

run that tester and supply it with the SOLVER.SH or .BAT, and it should run automatically.

4.10 Scrapped/partially implemented features

There were some things that we did not manage to finish in time. We will quickly go over those things, and the reasoning why.

First, one of the initial plans was to find the optimal permutation of ties. This was before looking into serious tiebreaking methods, just a personal idea to attempt some form of optimization of the found order. This was done using a function to calculate each permutation of each group of ties and taking the Cartesian product of all these groups. This was immediately scrapped when we tried to run this method on the larger instances given the way the complexity scales.
Quite similar, one of the tiebreaking methods that was considered the best option in Section 3.2 was the OPTIMAL method, which finds the optimal permutation of tied vertices. Here, "optimal" means that the permutation of tied vertices produces the least number of crossing with each other. This is similar to the original idea, except for the fact that this looks *only* at the tied vertices themselves, without calculating it for the total order.
Even though we did manage to implement this, given the nature of the method in that it check every possible permutation, there was no way to implement a crossing counter that was fast enough to run this millions of times. The implemented functions include a partial crossing counter, following the same idea as for the "complete" crossing counter, as well as a functions to calculate all possible permutations.

Additionally, one of the final attempts at optimizing the function for counting crossings was also a suggestion from a supervisor, to look at the sweepline algorithm. Sadly, there was not enough time to fully implement this algorithm, but we will still give a quick and general idea of the algorithm. For a more detailed overview and formal proof, see [15].
This algorithm uses a line called the sweepline that moves over all *event points* (all vertices and crossings). Additionally, it uses a *sweepline status* (all events "currently happening", in this cases all edges that cross the sweepline) and an *event queue*, which contains all possible events in the future. Important is that both the sweepline status is sorted on the order in which the edges cross the sweepline, and the event queue sorted in the order that the events happen.
See 8 for an example. The dashed line is the sweepline, going from top to bottom. The event points are the u_i, v_i and the four points indicated. The current sweepline status (from left to right) is $\{(v_1, u_1), (v_2, u_2), (v_3, u_1), (v_3, u_3)\}$. The event queue is $\{\text{Red circle 1}, \text{Red circle 2}, u_1, u_2, u_3, u_4\}$. The current

event is the point at the blue square.
We see that the points v_i are passed, so they are in the queue. The point at the green diamond is passed as well, so at that point, the order of (v_1, u_4) and (v_2, u_2) was swapped. Also, after swapping those two, as said above, we check the swapped pair with their respective neighbours for possible event points. So because of that, the blue square and the first red circle were added to the event queue.

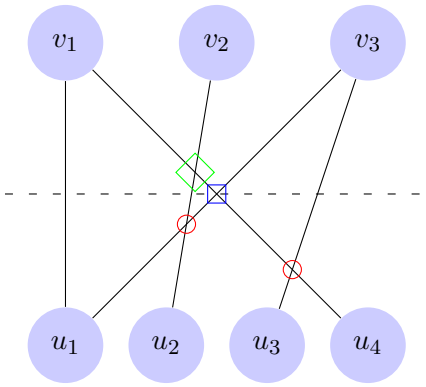


Figure 8. *Example with two arcs and two vertices.*

Each event point triggers certain changes in the sweepline status and the event queue, either adding or removing event points. In particular, each time the sweepline status changes, the changed event is checked with its neighbours (the previous and next event in the queue) to see if they crossed or not, changing the order of events. The point of intersection is added to the event queue, since this would impact the ordering of the sweepline.
In the case of finding all crossings between edges, the event points are the vertices which are either a "starting point" or "end point" of an edge. If the sweepline passes a starting point, we can easily see that the corresponding edge now crosses the sweepline, so it is removed from the sweepline status. Similarly, passing an endpoint means that edge no longer crosses the sweepline.
If the sweepline changes, the algorithm checks if any future events change, i.e. if the changed order/added edge introduces a intersection in the future. If so, this point is added to the event queue.
Obviously, since each intersection is passed by the sweepline, we know the total number of crossings based on the total number of events (minus the vertices of edges).

Lastly, we would have liked to expand the test cases, just to make sure that the implemented features work as intended, but it felt a bit "too much" for the rewards, compared to some other things we could implement instead.

5 Discussion & Future work

In this section we will touch on some points that still have and future work.

First, as already mentioned, we wanted to improve the crossings function by using the sweepline algorithm, but due to the time constraint we did not manage to implement the working function before the thesis deadline. It definitely looks like an interesting approach for a crossings algorithm, compared to the "default" crossing count.

Secondly, we misinterpreted the reverse + barycenter tiebreaking for the median method, so our data does not cover all tiebreaking methods we wanted. Furthermore, with the data we did obtain for our tiebreaking comparisons, as shown in Appendix A,we did not find any clear reason or idea as to what caused the difference in performance of the respective methods. We would have liked to look into it further, especially to see if there is some sort of property or logic for the cases, for example whether certain methods perform better on dense or sparse graphs, or that size plays a role, etc.

Additionally, at the start of the project, we had ideas to implement some sort of improvement system. For the submission to the PACE challenge, there are 8 minutes to find the best result of a given problem, so it would be a good investment to see if a iterative improving algorithm, for example would be 2-opt[4] would have improved the performance, or changing up the starting order, different combinations of tiebreaking methods, etc.

Obviously, there is also the point of adding more diverse heuristic methods, besides the median and barycenter methods. As shown in [9], there are at least a dozen heuristic methods that could have their uses. This also ties into the lack of "logic" w.r.t. the different problems: right now the implementation simply runs all implemented methods and chooses the method that returns the best result, without any prior choosing or optimization. For example, there are several mentions that the barycenter and median methods perform better on denser graphs [5, 6]. This would be an interesting test to see how well differentiating on certain properties of the problems impacts the results and performance of the program.

As for the mistake in the proof, without the upper bound of the median method we no longer have a proof for a constant approximation of the upper bound like we wanted. Even though it does not impact the submission for the PACE-challenge directly, we would have liked to search for a valid

[4]For clarification: in short, 2-opt swaps two elements such that the remaining solution is better. First introduced as a possible solution to the Travelling salesman problem by Croes [2].

upper bound and write a correct proof.

In conclusion, we can say that even though the program is fully functional *as is*, there is more then enough room for optimization and expansion on the existing framework.

References

[1] T. Catarci. The assignment heuristic for crossing reduction. *IEEE Transactions on Systems, Man, and Cybernetics*, 25(3):515–521, 1995. doi: 10.1109/21.364865.

[2] G. A. Croes. A method for solving traveling-salesman problems. *Operations Research*, 6(6):791–812, 1958.

[3] S. Dresbach. A new heuristic layout algorithm for dags. In *Operations Research Proceedings 1994: Selected Papers of the International Conference on Operations Research, Berlin, August 30–September 2, 1994*, pages 121–126. Springer, 1994.

[4] P. Eades and D. Kelly. Heuristics for reducing crossings in 2-layered networks. *Ars Combinatoria*, 21(A):89–98, 1986.

[5] P. Eades and N. C. Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11:379–403, 1994. doi: 10.1007/BF01187020.

[6] P. Eades, B. McKay, and N. Wormald. On an edge crossing problem. In *Proceedings of the Ninth Australian Computer Science Conference*, pages 327–334. Varies, 1986.

[7] R. A. Fisher and F. Yates. *Statistical tables for biological, agricultural and medical research*. London: Oliver Boyd, third edition, 1948.

[8] S. Hol. Pace2024 heuristic submission. A link to the repository. URL `https://git.science.uu.nl/s.hol/pace2024-heuristic-submission`.

[9] M. Jünger and P. Mutzel. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. *Journal of Graph Algorithms and Applications*, 1:1–25, 01 1997. doi: 10.1142/9789812777638_0001.

[10] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. ISBN 0-306-30707-3.

[11] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics (NRL)*, 52, 1955. doi: 10.1002/nav.3800020109.

[12] C. Matuszewski, R. Schönfeld, and P. Molitor. Using sifting for k-layer straightline crossing minimization. volume 1731 of *Lecture Notes in Computer Science (LNCS)*, pages 217–224. Springer, Berlin, Heidelberg, 12 1998. doi: 10.1007/3-540-46648-7_22.

[13] P. Patarasuk. Crossing reduction for layered hierarchical graph drawing, 2004. URL `https://diginole.lib.fsu.edu/islandora/object/fsu:180382/datastream/PDF/view`. This is a master thesis at Florida State University (FSU).

[14] T. Poranen and E. Mäkinen. Tie-breaking heuristics for the barycenter and median algorithms. This is a self-published manuscript. URL `https://www.researchgate.net/publication/254412140_Tie-Breaking_Heuristics_for_the_Barycenter_and_Median_Algorithms`.

[15] F. P. Preparata and M. I. Shamos. *Intersections*, pages 278–287. Springer-Verlag, 1985.

[16] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981. doi: 10.1109/TSMC.1981.4308636.

[17] V. Valls, R. Martí, and P. Lino. A branch and bound algorithm for minimizing the number of crossing arcs in bipartite graphs. *European Journal of Operational Research*, 90(2):303–319, 1996. doi: 10.1016/0377-2217(95)00356-8.

A Tiebreaking comparison data

Problem	$\|L\|$	$\|U\|$	$\|E\|$	Graph Density	Number of Medians	Number of Barycenters
1	6328	6218	12545	0.000318826	3	2
2	10153	10013	20165	0.000198353	3	2
3	1485	1433	2917	0.001370767	3	2
4	1078	991	2148	0.002010675	18	17
5	3638	3426	7191	0.000576951	27	26
6	949	870	1898	0.002298851	18	17
7	5580	5356	10973	0.000367156	18	24
8	791	718	1581	0.002783755	17	22
9	45740	45444	91183	4.38673E-05	6	7
10	131315	130809	262123	0.00001526	6	7
11	1467	1483	3723	0.001711283	623	816
12	1608	1647	4099	0.001547741	724	943
13	1920	1998	4890	0.001274712	858	1116
14	847	839	15362	0.021617347	689	714
15	982	973	30672	0.032100941	831	931
16	1098	1109	2206	0.001811639	562	779
17	2245	2224	4468	0.000894874	1107	1570
18	2502	2371	4872	0.000821275	1211	1691
19	2125	2096	8257	0.001853839	1382	1755
20	639	626	1264	0.003159889	323	432
21	794	895	1688	0.002375357	407	554
22	1070	1070	3210	0.002803738	797	814

23	1564	1564	4692	0.001918159	1174	1202
24	1828	1828	5484	0.001641138	1376	1410
25	6889	6682	20273	0.000440408	2408	4440
26	10343	10092	30536	0.000292542	3667	6694
27	7511	7804	15237	0.000259947	2420	4523
28	10040	10453	20447	0.00019483	3201	5997
29	15829	2461	18289	0.000469488	2461	2461
30	17219	2657	19875	0.000434418	2657	2657
31	657	657	1314	0.00304414	436	483
32	745	745	1490	0.002684564	494	555
33	922	922	1844	0.002169197	608	690
34	13122	1747	13122	0.00057241	1747	1747
35	21362	2847	21362	0.000351247	2847	2847
36	1599	1548	1991	0.000804363	446	827
37	1597	1564	13283	0.005318076	909	1435
38	1629	1692	12744	0.004623643	998	1518
39	2484	2596	9919	0.001538196	1917	2251
40	2860	2943	11305	0.001343118	2213	2592
41	4800	4800	19004	0.000824826	4798	4800
42	2345	2345	9243	0.001680843	2343	2344
43	3321	3321	13121	0.001189677	3319	3319
44	65536	65536	1114112	0.000259399	29476	58471
45	256	256	2304	0.03515625	144	225
46	16077	16077	33433	0.00012935	16077	16076
47	794	794	1891	0.002999511	794	794

48	1234	1234	2967	0.001948441	1234	1234
49	493	493	1189	0.004892018	493	492
50	975	975	2344	0.002465746	975	974
51	1086	1086	2631	0.002230803	1086	1085
52	2095	2074	4168	0.000959257	835	1505
53	912	886	1797	0.002223922	403	680
54	1183	1230	2412	0.001657629	480	888
55	1644	1634	3277	0.001219895	704	1121
56	1152	1201	2352	0.001699972	503	812
57	2006	2006	4010	0.000996512	3	4
58	2606	2606	5210	0.000767165	3	4
59	2934	2934	5866	0.000681431	3	4
60	1442	1472	3028	0.001426536	355	783
61	804	801	1696	0.002633524	188	463
62	943	907	2024	0.002366418	213	510
63	1186	1200	1719	0.001207841	258	624
64	1580	1535	2146	0.000884839	340	777
65	518	504	763	0.002922565	128	259
66	1013	1077	1074	0.000984417	257	566
67	1719	1786	2158	0.000702901	532	1041
68	1840	1866	2310	0.000672795	566	1112
69	26294	7532	26294	0.000132767	7532	7216
70	30514	8706	30514	0.000114863	8706	8385
71	11678	6020	20924	0.000297632	4765	5451
72	15818	8772	30611	0.000220611	6722	7880

73	1492	1445	4427	0.002053397	406	961
74	1517	1543	4610	0.00196947	415	982
75	1980	1989	6024	0.001529625	558	1283
76	498	528	1025	0.003898168	236	365
77	654	645	1298	0.003077069	325	478
78	1508	1466	2973	0.001344806	707	1072
79	525	539	1063	0.003756516	255	403
80	581	593	1173	0.003404609	279	448
81	1230	1277	2506	0.001595457	606	921
82	1700	1667	3313	0.00116906	500	1011
83	1738	1731	3466	0.001152078	488	996
84	1804	1832	3549	0.001073851	498	1055
85	674	673	1759	0.003877849	399	543
86	865	884	2700	0.003530981	536	736
87	971	989	2630	0.002738673	577	796
88	1179	1184	4051	0.002901996	727	989
89	1531	1537	5157	0.002191533	967	1293
90	720	724	2532	0.004857274	455	592
91	1033	1038	3335	0.003110271	674	883
92	519	522	1705	0.006293417	334	435
93	769	773	2389	0.004018929	509	648
94	1352	1377	4749	0.002550889	951	1198
95	1463	1429	3906	0.001868339	888	1202
96	2005	2097	2161	0.000513975	520	1088
97	1180	1110	3431	0.002619484	158	578

98	1736	1719	5182	0.001736488	221	856
99	7006	9340	18680	0.00028547	2336	7004
100	8794	11724	23448	0.000227428	2932	8792
complete_4_5	4	5	20	1	1	1
cycle_8_shuffled	4	4	8	0.5	3	3
cycle_8_sorted	4	4	8	0.5	3	4
grid_9_shuffled	4	5	12	0.6	2	3
ladder_4_4_shuffled	4	4	10	0.625	4	3
ladder_4_4_sorted	4	4	10	0.625	4	4
matching_4_4	4	4	4	0.25	1	4
path_9_shuffled	5	4	8	0.4	3	4
path_9_sorted	5	4	8	0.4	4	4
plane_5_6	5	6	10	0.333333333	4	5
star_6	2	6	6	0.5	1	2
tree_6_10	6	10	15	0.25	3	5
website_20	10	10	12	0.12	2	9

Table 2. *A table containing basic details on that problems graph.*

Problem	Barycenter Method	Median Method	Random on BC	Reverse + BC on Median:	Reverse on BC
1	259822	38651088	**170062**	38651088	448802
2	997222	100240143	622582	100240143	**444362**
3	**18774**	2050623	25846	2050623	61362
4	18425	637127	11551	1011527	**2408**
5	99495	7141427	63501	12097107	**7536**
6	15642	518080	5844	797762	**2166**
7	10847	23583253	**10839**	28881982	**10839**
8	**1962**	435118	1968	518780	1974
9	12885182	2037978558	10526358	2064930216	**568622**
10	60616276	16979783549	29633896	17110340946	**5577138**
11	2062625	2504627	2062641	2524668	**2062614**
12	**2502967**	3069589	2502968	3100757	2502973
13	3417751	4123336	**3417746**	4162249	**3417746**
14	**1442511**	1542654	**1442511**	1542752	**1442511**
15	**10857054**	10864422	10857057	10865550	10857057
16	238632	716626	238617	790577	**238616**
17	**923744**	2905158	**923744**	3206388	923766
18	**1155200**	3403053	**1155200**	3737092	1155205
19	11130815	11983613	11130794	11984723	**11130758**
20	186929	325197	186927	348173	**186920**
21	324562	601066	324567	657995	**324559**
22	**1596236**	1915982	1596266	1916307	1596267
23	3431064	4121780	3431068	4122207	**3431047**
24	4543996	5407714	4543964	5408212	**4543951**
25	**272176**	14978072	272235	16152883	272234

26	409532	32573193	409520	35195712	**409498**
27	**120290**	17861434	120313	21941780	120347
28	**160556**	31520183	160579	38880075	160622
29	**0**	**0**	**0**	**0**	**0**
30	**0**	**0**	**0**	**0**	**0**
31	**213510**	213626	**213510**	213731	**213510**
32	**280916**	281036	**280916**	281167	**280916**
33	**440799**	440958	**440799**	441113	**440799**
34	**0**	**0**	**0**	**0**	**0**
35	**0**	**0**	**0**	**0**	**0**
36	443607	807671	443595	886056	**443590**
37	**33673667**	35003979	33673668	35006691	33673671
38	30684426	32184204	30684400	32186542	**30684397**
39	**790778**	1477204	790845	1479240	790820
40	**239105**	834102	239128	835978	239152
41	**1333704**	1353842	**1333704**	1353842	**1333704**
42	**447719**	457372	447720	457377	447720
43	**773096**	786960	**773096**	786960	773098
44	2.60871E+11	2.9798E+11	**2.60871E+11**	2.97982E+11	2.60871E+11
45	1023445	1164406	1023465	1165227	**1023435**
46	**39567**	59228	**39567**	59228	**39567**
47	**1453**	2018	**1453**	2018	**1453**
48	**2204**	3067	**2204**	3067	**2204**
49	**876**	1209	**876**	1209	**876**
50	**1751**	2423	**1751**	2423	**1751**

51	**1915**	2628	**1915**	2628	**1915**
52	1732172	3370922	**1732171**	3643781	1732194
53	308164	620751	**308158**	664574	308164
54	590291	1194290	590302	1294025	**590288**
55	**1310790**	2253576	1310794	2419583	1310799
56	680115	1200845	680094	1294758	**680081**
57	**252004**	2759132	**252004**	3263012	**252004**
58	**425104**	4675704	**425104**	5509412	**425104**
59	**538756**	5915166	**538756**	6984756	**538756**
60	**1270945**	1601862	1270954	1675667	1270956
61	**371612**	488253	371615	510316	371619
62	566769	702349	**566741**	724131	566745
63	371601	570599	**371600**	622785	371604
64	561984	890179	561982	985404	**561978**
65	74807	110663	74799	120674	**74794**
66	**106127**	246979	106131	286453	106131
67	475994	896415	476000	998726	**475992**
68	582986	1055552	**582985**	1172618	582988
69	**728697**	775578	728698	775578	728708
70	**847368**	902428	847393	902428	847377
71	**571161**	611451	571171	613362	571185
72	**841172**	901888	841215	905111	841237
73	2905493	3333146	**2905486**	3372364	2905508
74	**3167271**	3572724	3167276	3610360	3167285
75	5573430	6317943	**5573403**	6380944	5573425

76	119586	213127	**119577**	227657	119585
77	207462	351425	207468	368444	**207456**
78	**1088145**	1823360	1088149	1921914	1088175
79	**138520**	240134	138527	253816	138523
80	163537	278501	163537	295159	**163531**
81	757438	1263088	757435	1348983	**757425**
82	1519623	2031491	**1519619**	2124104	1519624
83	1698309	2191631	**1698306**	2298899	1698313
84	**1760965**	2345993	1760980	2462376	1760988
85	475933	564959	**475930**	568776	475931
86	1131716	1285076	**1131709**	1289046	1131727
87	1033705	1249618	1033700	1259218	**1033697**
88	227173	545195	**227143**	556038	227153
89	370934	866434	**370933**	881404	370935
90	**79614**	202677	79616	206926	79623
91	162159	326431	162164	334157	**162154**
92	**60178**	117773	60191	120729	60193
93	133386	266589	133378	272646	**133373**
94	**3634300**	3993653	3634320	3995600	3634340
95	2215289	2667330	**2215281**	2689079	2215299
96	463951	948301	**463950**	1092502	463951
97	1961889	2407475	1961876	2435212	**1961867**
98	**4504928**	5543296	4504954	5613304	4504971
99	**51792631**	53175502	**51792631**	54524581	**51792631**
100	**81607829**	83743892	**81607829**	85910537	**81607829**

complete_4_5	**60**	**60**	**60**	**60**	**60**
cycle_8_shuffled	**4**	**4**	**4**	5	**4**
cycle_8_sorted	**3**	**3**	**3**	4	**3**
grid_9_shuffled	**17**	21	**17**	21	**17**
ladder_4_4_shuffled	**11**	13	**11**	13	**11**
ladder_4_4_sorted	**3**	5	**3**	5	**3**
matching_4_4	**0**	2	**0**	6	**0**
path_9_shuffled	**6**	**6**	**6**	7	**6**
path_9_sorted	**0**	**0**	**0**	**0**	**0**
plane_5_6	**0**	6	**0**	6	**0**
star_6	**0**	3	**0**	9	**0**
tree_6_10	**13**	38	**13**	59	**13**
website_20	**17**	33	**17**	45	**17**

Table 3. *A table containing results for the used methods per problem with the best result highlighted.*