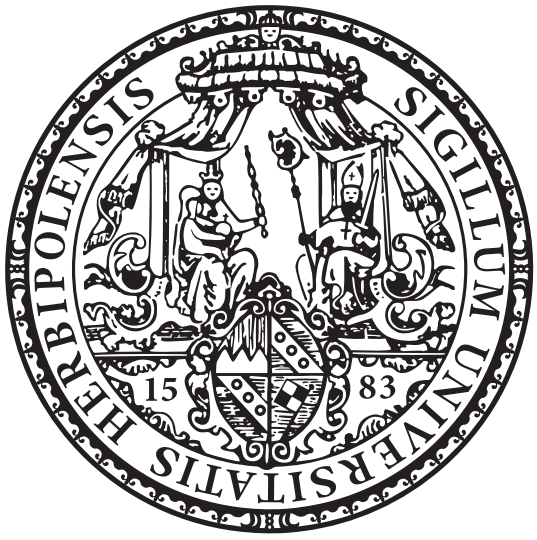
Practical Course Report

Exact and Heuristic Algorithms for Computing the Local Circular Crossing Number

Marcel Peters

Date of Submission: 23.05.2026
Advisors: Samuel Wolf
Prof. Dr. Alexander Wolff



Julius-Maximilians-Universität Würzburg
Lehrstuhl für Informatik I
Algorithmen und Komplexität

1 Introduction

A natural way to draw graphs is to embed the vertices onto some sort of convex outer curve and to have the edges all inside this curve. This can introduce clutter by having too many edges crossing each other. In this work, we look at minimizing the number of crossings each edge can have, also known as *local circular crossing minimization*.

To define the problem, we first define a convex map $f\colon S \to \mathbb{R}^2$ on a set S as an injective function, where no three points are colinear and for each quadruple $\{a, b, c, d\} \subseteq S$ the point $f(d)$ does not lie in the interior of the triangle spanned by $f(a)$, $f(b)$, and $f(c)$. A convex map f on S defines a cyclic order $\Omega_f \subseteq S^3$ where $(a, b, c) \in \Omega_f$ if and only if $f(a)$, $f(b)$ and $f(c)$ lie counterclockwise to each other for any three distinct $a, b, c \in S$. Notably, for any triple (a, b, c) of a cyclic order Ω, the permutations (b, c, a) and (c, a, b) lie also in Ω, whereas the inverses (c, b, a), (a, c, b) and (b, a, c) do not. Also, for any three distinct $a, b, c \in S$ where (a, b, c) does not lie in Ω, the inverse (c, b, a) lies in Ω.

Given a graph $G = (V, E)$, an outer drawing Γ of G is a tuple (f, g) where $f\colon V \to \mathbb{R}^2$ is a convex map, mapping each vertex to a distinct point in two dimensional space, and $g\colon E \to ([0,1]_{\mathbb{R}} \to \mathbb{R}^2)$ a function, mapping each edge to a continuous open curve, where for each $\{u, v\} \in E$ the endpoints correspond to the positions of the vertices, so $\{f(u), f(v)\} = \{g(\{u, v\})(0), g(\{u, v\})(1)\}$. A given outer drawing $\Gamma = (f, g)$ of a graph $G = (V, E)$ is an outer k-planar drawing for $k \geq 0$, if each edge $\{u, v\} \in E$, does not cross more than k other edges except at the endpoints, or more formally

$$\left(\sum_{e' \in E \setminus \{e\}} \varphi(e, e') \right) \leq k \quad \forall e \in E$$

where

$$\varphi(e, e') := \begin{cases} 1 & g(e)[(0,1)_{\mathbb{R}}] \cap g(e')[(0,1)_{\mathbb{R}}] \neq \emptyset \\ 0 & \textbf{else} \end{cases}.$$

If a graph G admits an outer k-planar drawing, we call G *outer k-planar*. The smallest number k for which a given graph is outer k-planar, is called the *local circular crossing number* of that graph. This problem is NP-hard [1]. Hong and Nagamochi [2] showed, that so called *full outer 2-planarity* can be recognized in linear time. A graph is *full* outer k-planar if it is outer k-planar and no crossings appear on the outer boundary. Chaplick et al. [3] generalized the linear time recognition of full outer k planarity to any fixed k and further showed that outer k-planarity can be recognized in quasi polynomial time for any fixed k. Kobayashi et al. [4] showed that outer k-planarity can be recognized in polynomial time for any fixed k, and also that for a fixed $c \geq 1$, there is no polynomial time approximation for finding the local circular crossing number unless $\mathsf{P} = \mathsf{NP}$.

Hong and Nagamochi [2] showed that we can assume the curves of the edges to be straight lines. Now crossings can be defined as two edges $\{u, v\}$ and $\{s, t\}$ crossing in a drawing (f, g) if and only if s and t lie on opposite sides of $\{u, v\}$, which in turn can be defined as $(u, v, s) \in \Omega_f \Leftrightarrow (u, v, t) \notin \Omega_f$. From this, it is easy to see that we can continuously map any two drawings with the same cyclic order into each other without introducing crossings. Thus we only need to look at the different possible cyclic orders of the vertices of a given graph. Since any given cyclic order has an inverse retaining all crossings, which we call the *mirror symmetry*, we can half the number of relevant orders.

Here, we present multiple approaches for finding the local circular crossing number, either exactly or heuristically. In most approaches, we formulate the cyclic order as a linear order with some additional handling to maintain the cyclic nature. An intuitive way to represent a linear order is by using absolute (one dimensional) positions, where an element precedes another if its position is before the other's. Another way is to represent the order relatively using a relation, where a pair (a, b) is in the relation, if a precedes b. This has the benefit, that it can be entirely encoded by binary values, but requiring to additionaly uphold the transitive property, that if (a, b) and (b, c) are in the relation,

(a, c) must also. In any way we represent the linear order, we can "rotate" the order, by moving the first element to the end, retaining all crossings, which we call the *rotational symmetry*. This allows us to fix a vertex to the front, reducing the number of relevant orders multiplicatively by $|V|$. Since we can trivially create a drawing from the drawings of two disconnected or 1-connected components without increasing the local circular crossing number, we only need to analyze 2-connected graphs. We furthermore assume that each graph has at least four vertices, since any drawing of graphs with less than four vertices is trivially outer (0-)planar.

This work is split in two main parts. In the first, we present heuristics for efficiently calculating a good solution. In the second, we present comparatively slow approaches for an exact solution, specifically ILP formulations and iterative approaches, based on brute force. After the main parts, we evaluate all approaches. For our data set, we will find the heuristics to give almost exact solutions and the iterative approaches to be faster than all of the ILP formulations, but no exact approach to be able to give a solution for larger instances.

2 Heuristics

2.1 Greedy Generation

The idea of a greedy algorithm is to calculate a solution by finding local optima. We start by creating an absolute linear order Ω prepopulated by the three vertices with highest degree. The concrete placement of these is irrelevant because of the mirror and rotational symmetries, but for our implementation, we ordered them decendingly. Next, iteratively, by selecting the vertex with next highest degree, we find the currently best possible placement for this vertex by trying every possible placement in and at the end of Ω. We do not need to try placing the vertex at the start of Ω, since this is equivalent to having the vertex at the end. After finding this best placement, we update Ω and handle the next vertex until we have placed all vertices, yielding the result.

To iterate through the positions efficiently, we first append the current vertex to the end of Ω and then only need to swap the current position with the one before, until the vertex is at the start of Ω. This also allows us to only store the index of the best possible placement and "reinserting" the element from the front to the best placement at the end, instead of needing to copy the entirety of Ω which reduces the amount of allocations and copies.

Algorithm 1: Greedy Generation

```
1:  function GREEDY(G = (V, E))
2:      ϒ ← SORT_DECENDING(V)
3:      Ω ← [ϒ_0, ϒ_1, ϒ_2]
4:      for v ∈ ϒ_{≥2} do
5:          i_min ← nil
6:          k_min ← ∞
7:          Ω ← APPEND(Ω, v)
8:          for i = |Ω| − 1, ..., 1 do
9:              k ← LOCAL_CIRCULAR_CROSSING_NUMBER(Ω, E)
10:             if k_min > k then
11:                 i_min ← i
12:                 k_min ← k
13:             end
14:             Ω ← SWAP(Ω, i, i − 1)
15:         end
16:         Ω ← REINSERT(Ω, 0, i_min)
17:     end
18:     return Ω
19: end
```

Algorithm 2: Calculating the local circular crossing number

```
1:  function LOCAL_CIRCULAR_CROSSING NUMBER(E, Ω)
2:      k_max ← 0
3:      for {u, v} ∈ E do
4:          k ← 0
5:          (f, l) ← if Ω_u < Ω_v then (Ω_u, Ω_v) else (Ω_v, Ω_u)
6:          for {s, t} ∈ E do
7:              if {u, v} ∩ {s, t} = ∅ ∧ ((f < Ω_s < l) ⊕ (f < Ω_t < l)) then
8:                  k ← k + 1
9:              end
10:         end
11:         if k_max < k then
12:             k_max ← k
13:         end
14:     end
15:     return k_max
16: end
```

While in each iteration, we start with nonsensical values **nil** and ∞, these get replaced in the first iteration with sensical values. Making the order Ω a bidirectional map instead of a simple map from $\mathbb{N}_{<|\Omega|} \to V$, encoded as an array, allows us to look up the positions of vertices in constant time at the cost of increasing the memory footprint, although this can be minimized by giving the vertices a canonical injective map $V \to \mathbb{N}_{<|V|}$ allowing us to use an array for the backwards direction $V \to \mathbb{N}_{<|\Omega|}$. For calculating the local circular crossing number (Algorithm 2), we assume we are given an edge list and an order represented by a map from $V \to \mathbb{N}_{<|V|}$.

In total, the algorithm has a time complexity of $\mathcal{O}(|V|^2|E|^2)$ where the $|E|^2$ component comes from calculating the local circular crossing number of the given order. There may be a way to reduce the time complexity for calculating the local circular crossing number by reusing parts of the previous calculation, but since moving a vertex can change the placement of multiple edges, we would need a better reasoning abount which edges change and which crossing are now able to be added, possibly requiring a larger memory footprint from the current $\mathcal{O}(|V| + |E|)$ for the entire greedy generation to probably $\mathcal{O}(|V| + |E|^2)$ or even larger.

2.2 Improving an Order

Another possibility is to improve a given order. Given an order with local circular crossing number k, we look at an edge e with k crossings and try to find the next best possible positioning for the endpoints of that edge. A possible positioning is one where for each edge the previous crossing number stays less than k if it was already less and stays less than or equal to k if it was equal to k. This ensures that the algorithm will terminate. The better of two possible positionings is the one where e has less crossings. If no positioning can be found where e has less than k crossings, the algorithm terminates. Else, we apply the positioning to the order and reapply the algorithm to the new order. We terminate directly for simplicity's sake. Instead, the other edges with crossing number k could be tried first.

To get the crossing numbers for all edges, we use a modification of Algorithm 2 which returns a map $E \to \mathbb{N}$ together with an edge of highest crossing number. We assume a remove function for lists that also returns the index of the element found.

Given an order with at most k crossings per edge, this algorithm can try to improve each edge at most $\mathcal{O}(k)$ times with $\mathcal{O}(|V|^2)$ different positionings, leading to $\mathcal{O}(k|V|^2|E|)$ total positionings. Only for each time the order is improved, it needs to check every other edge to determine if the positioning is possible, leading to $\mathcal{O}(k|E|^2)$. Since calculating the edge crossing numbers has a time complexity of $\mathcal{O}(|E|^2)$, the total time complexity of the algorithm is in

Algorithm 3: Improving an Order

```
1:  function IMPROVE(G = (V, E), Ω)
2:      K ← EDGE_CROSSING_NUMBERS(Ω, E)
3:      loop do
4:          {u, v} ← EDGE_WITH_MOST_CROSSINGS(K)
5:          k ← K({u, v})
6:          (Ω, b_v) ← REMOVE(Ω, v)
7:          (Ω, b_u) ← REMOVE(Ω, u)
8:          Ω ← APPEND(Ω, u)
9:          for i_u = |Ω| − 1, ..., 1 do
10:             Ω ← APPEND(Ω, v)
11:             for i_v = |Ω| − 1, ..., 1 do
12:                 K′ ← EDGE_CROSSING_NUMBERS(Ω, E)
13:                 if K′({u, v}) < K({u, v}) then
14:                     p ← ⊤
15:                     for e ∈ E do
16:                         p_1 ← K(e) < k ∧ K′(e) < k
17:                         p_2 ← K(e) = k ∧ K′(e) ≤ k
18:                         p ← p ∧ (p_1 ∨ p_2)
19:                     end
20:                     if p then
21:                         K ← K′
22:                         b_u ← i_u
23:                         b_v ← i_v
24:                     end
25:                 end
26:             end
27:             Ω ← REMOVE_AT(Ω, 0)
28:             Ω ← SWAP(Ω, i_v, i_v − 1)
29:         end
30:         Ω ← REMOVE_AT(Ω, 0)
31:         Ω ← INSERT(Ω, u, b_u)
32:         Ω ← INSERT(Ω, v, b_v)
33:         if K({u, v}) ≥ k then
34:             return Ω
35:         end
36:     end
37: end
```

$\mathcal{O}(k|V|^2|E|)\mathcal{O}(|E|^2) + \mathcal{O}(k|E|^2) = \mathcal{O}(k|V|^2|E|^3)$. In reality, when supplying an already good order, the algorithm will terminate much faster.

3 Exact Solutions

3.1 ILP Formulations

A common way to tackle NP-complete problems is to formulate them as an ILP and then let a solver try to find a solution. In our case, we formulate three different but related sets of constraints. Each of them includes the crossing variables $c_{\{e,e'\}} \in \{0, 1\}$ for each set of two edges $\{e, e'\} \subseteq E$, the local circular crossing number $k \in \mathbb{N}_0$ and the constraints $k \geq \sum_{e' \in E : e' \cap e = \emptyset} c_{\{e,e'\}}$, called k-constraints, for each $e \in E$. Edges that share a vertex can never cross, thus instead of creating crossing variables for them, they can be inlined as 0 in every constraint. Since sets are unordered, to make iteration through subsets of vertex and edge sets deterministic, we assume that vertices and edges have an implicit order. When specifying $\{v_1, ..., v_n\} \subseteq V$, the variables denote the vertices in the order $v_1 < ... < v_n$. The case for edges is analogous.

3.1.1 Definitions

We abbreviate the set $\{0, 1\}$ as $\mathbb{B}$. Furthermore, to simplify the notation for the formulations, we make use of common logic operations that can be reduced to arithmetic operations. For this, we introduce the concept of *polarity*, i.e. negative and positive occurrance. An expression occurs positively (negatively) in a constraint if it has a positive (negative) sign on the smaller side of the comparison. Further, we say that a constraint is positive (negative) for a variable if that variable occurs negatively (positively) in the normalized constraint. A constraint is *normalized* when it has the form $c \geq \sum_{i \in I} a_i x_i$ for an index set I, constants c and a_i and distinct variables x_i for each $i \in I$. Notably the constraint $x \geq t$ where x does not occur in t is positive for variable x and term t occurs in it positively.

Some operations can be directly replaced irrespective of their polarity; namely inversion $\neg A$, implication $A \Leftarrow B$, exclusive disjunction $A \oplus B$, and equivalence $A \Leftrightarrow B$ reduce to $1 - A$, $A \vee \neg B$, $(A \wedge \neg B) \vee (\neg A \wedge B)$ and $(A \Leftarrow B) \wedge (\neg A \Leftarrow \neg B)$, respectively. Other operations behave differently depending on their polarity; namely conjunction $A \wedge B$ and disjunction $A \vee B$. These operations get replaced in a negative context via their De Morgan dual: $-(A \wedge B)$ to $(\neg A \vee \neg B) - 1$ and $-(A \vee B)$ to $(\neg A \wedge \neg B) - 1$. Conjunction in a positive context $A \wedge B$ reduces to $A + B - 1$. Disjunction in a positive context for contraint c reduces to two constraints $c[A/(A \vee B)]$ and $c[B/(A \vee B)]$, where for $A \vee B$, we substitute A or B, respectively.

We assume that every term t used as a constraint is implicitly true, i.e. $t \geq 1$. Notably, implication $A \Leftarrow B$ and equivalence $A \Leftrightarrow B$ as constraints then have the following equivalences:

$$A \Leftarrow B \geq 1 \equiv A \geq B$$

$$A \Leftrightarrow B \geq 1 \equiv \begin{cases} A \geq B \\ \neg A \geq \neg B \end{cases}$$

3.1.2 Order Fomulation

The order formulation is based on the bachelor's thesis of Ivan Shevchenko [5] and serves as a base to compare against. It only includes one additional kind of variables: the order variables a_{uv} for each ordered pair of $u, v \in V$ where $u \neq v$, which are to be interpreted that $a_{uv} = 1$ iff u lies before v. By themselves, these variables are constrained by antisymmetry $a_{vu} = \neg a_{uv}$ and transitivity $a_{uw} \Leftarrow a_{uv} \wedge a_{vw}$ for $u, v, w \in V$. Furthermore, these variables constrain the crossing variables $c_{\{uv,st\}} \Leftarrow a_{us} \wedge a_{sv} \wedge a_{vt}$ in all eight different possible ways, for each pair of edges $\{uv, st\} \subseteq E$ where uv and st don't share any vertices. Since the variables a_{uv} and a_{vu} can be defined by the other, one of them can be inlined as the inversion of the other.

As currently defined, for any triple $\{u, v, w\} \subseteq V$, we have six transitivity contraints: one for each direction of each pair. This can be reduced to two in total, since in these six, three are isomorphic. A given contraint $a_{uw} \Leftarrow a_{uv} \wedge a_{vw}$ can be rewritten to $0 \Leftarrow a_{uv} \wedge a_{vw} \wedge \neg a_{uw}$. We replace the six old transitivity contraints by $0 \Leftarrow a_{uv} \wedge a_{vw} \wedge a_{wu}$ and $0 \Leftarrow a_{wv} \wedge a_{vu} \wedge a_{uw}$, one for each direction, for each triple $\{u, v, w\} \subseteq V$.

One way to optimize this formulation is to fix one vertex $f \in V$ to the start (or end) of the order where then $a_{fv} = 1$ (or $a_{vf} = 1$ respectively) for any other vertex $v \in V$, removing the rotational symmetry. Further, we can remove the mirror symmetry by fixing the order of two other vertices $\{l, h\} \subseteq V \setminus \{f\}$ via either $a_{lh} = 0$ or $a_{lh} = 1$.

3.1.2.1 Valid Inequalities

Valid inequalities are constraints that do not change what solutions are feasable, but can guide the solver to find the solutions faster. Since the k-constraints are long sums of crossing variables, knowing a smaller set of edges with known crossing number can help break up the k-constraints, giving a quicker lower bound for k. In our case, we look for two kinds of subgraph, both being a chorded cycle, as seen in Figure 2. The first kind is a cycle with at least two crossing chords, having a crossing number

$$
\begin{aligned}
a_{uv} &\in \mathbb{B} && \forall \{u,v\} \subseteq V \setminus \{f\} : \{u,v\} \neq \{l,h\} \\
c_{\{e,e'\}} &\in \mathbb{B} && \forall \{e,e'\} \subseteq E \\
k &\in \mathbb{B} \\
a_{uf} &:= 1 && \forall u \in V \setminus \{f\} \\
a_{lh} &:= 1 \\
a_{vu} &:= 1 - a_{uv} && \forall \{u,v\} \subseteq V \\
0 &\geq a_{uv} + a_{vw} + a_{wu} - 2 && \forall \{u,v,w\} \subseteq V \\
0 &\geq a_{wv} + a_{vu} + a_{uw} - 2 && \forall \{u,v,w\} \subseteq V \\
c_{\{uv,st\}} &\geq a_{us} + a_{sv} + a_{vt} - 2 && \forall uv, st \in E : \{u,v\} \cap \{s,t\} = \emptyset \\
c_{\{uv,st\}} &\geq a_{ut} + a_{tv} + a_{vs} - 2 && \forall uv, st \in E : \{u,v\} \cap \{s,t\} = \emptyset \\
c_{\{uv,st\}} &\geq a_{vs} + a_{su} + a_{ut} - 2 && \forall uv, st \in E : \{u,v\} \cap \{s,t\} = \emptyset \\
c_{\{uv,st\}} &\geq a_{vt} + a_{tu} + a_{us} - 2 && \forall uv, st \in E : \{u,v\} \cap \{s,t\} = \emptyset \\
k &\geq \sum_{e' \in E, e' \cap e = \emptyset} c_{\{e,e'\}} && \forall e \in E
\end{aligned}
$$

Figure 1: Order formulation with fixed vertex $f = \max(V)$ and fixed pair $\{l,h\} \subseteq V \setminus \{f\}$

of one. The second kind is of the first kind, but with an additional chord crossing at least one of the other two crossing chords, thus having a crossing number of two. In the ILP formulation, they are implemented as the sum of a set of edges $E' \subseteq E$ having at least k' crossings where k' is the known crossing number of the subgraph, or more formally, as the constraint $k' \leq \sum_{\{e,e'\} \subseteq E' : e \cap e' = \emptyset} c_{\{e,e'\}}$.

The problem is finding these subgraphs faster than the solver would additionally take to solve without the constraints. Finding all cycles of a given graph to find our configurations is itself NP-hard, so we have to limit us to a smaller set of cycles. We chose fundamental cycles as this smaller set, being a base to generate all other cycles and being able to be found efficiently by a modified breadth or depth first search. Instead, one could also use a heuristic specifically tailored to this problem. Since a depth first search tends to generate large cycles, we use a breadth first search. Having found the fundamental cycles, we only need to check if two chords cross for the configuration to be of the first kind and if an additional chord crosses one of these to be of the second kind.

3.1.3 Betweenness Formulation

Looking at the fact that in the order formulation, the crossing variables each need eight different constraints that force them to be positive, we can add additional variables to reduce the number of contraints per crossing variable at the cost of adding contraints for these variables. One way to formulate an edge crossing in an ordered context is to say that two edges $uv, st \in E$ cross iff s lies between u and v iff t does not lie between u and v. We add a variable $b^v_{\{s,t\}} \in \mathbb{B}$ with the defining constraints $b^v_{\{s,t\}} :\Leftrightarrow a_{sv} \oplus a_{tv}$ for each edge $\{s,t\} \in E$ and vertex $v \in V \setminus \{s,t\}$. Now, the crossing

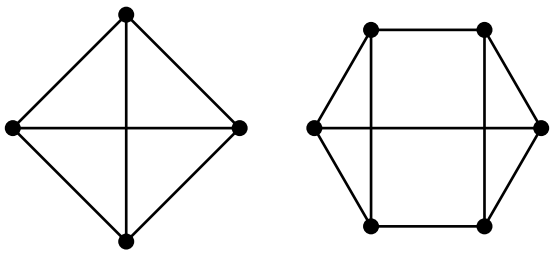


Figure 2: The smallest graph of the first kind (left) and second kind (right) of chorded cycle.

$$
\begin{aligned}
a_{uv} &\in \mathbb{B} && \forall \{u,v\} \subseteq V \setminus \{f\} : \{u,v\} \neq \{l,h\} \\
b_e^v &\in \mathbb{B} && \forall e \in E : \forall v \in V \setminus e \\
c_{\{e,e'\}} &\in \mathbb{B} && \forall \{e,e'\} \subseteq E \\
k &\in \mathbb{B} \\
a_{uf} &\coloneqq 1 && \forall u \in V \setminus \{f\} \\
a_{lh} &\coloneqq 1 \\
a_{vu} &\coloneqq 1 - a_{uv} && \forall \{u,v\} \subseteq V \\
0 &\geq a_{uv} + a_{vw} + a_{wu} - 2 && \forall \{u,v,w\} \subseteq V \\
0 &\geq a_{wv} + a_{vu} + a_{uw} - 2 && \forall \{u,v,w\} \subseteq V \\
b_{\{st\}}^v &\geq a_{sv} - a_{tv} && \forall e \in E : \forall v \in V \setminus e \\
b_{\{st\}}^v &\geq a_{tv} - a_{sv} && \forall e \in E : \forall v \in V \setminus e \\
b_{\{st\}}^v &\leq a_{vs} + a_{vt} && \forall e \in E : \forall v \in V \setminus e \\
b_{\{st\}}^v &\leq a_{tv} + a_{sv} && \forall e \in E : \forall v \in V \setminus e \\
c_{\{uv,e\}} &\geq b_e^u - b_e^v && \forall uv, e \in E : \{u,v\} \cap e = \emptyset \\
c_{\{uv,e\}} &\geq b_e^v - b_e^u && \forall uv, e \in E : \{u,v\} \cap e = \emptyset \\
k &\geq \sum_{e' \in E, e' \cap e = \emptyset} c_{\{e,e'\}} && \forall e \in E
\end{aligned}
$$

Figure 3: Betweenness formulation with fixed vertex $f = \max(V)$ and fixed pair $\{l,h\} \subseteq V \setminus \{f\}$

variables can be defined as $c_{\{uv,e\}} :\Leftrightarrow b_e^u \oplus b_e^v$. Since crossing variables only occur positively in the constrains for the objective that is to be minimized, the positive contraints $c_{\{uv,e\}} \Leftarrow b_e^u \oplus b_e^v$ suffice.

3.1.4 Triangle Formulation

The previous formulations all encode the cyclic order as a linear order, but we can also formulate the cyclic order directly. As for variables, we only add triangle variables $t_{uvw} \in \mathbb{B}$ for each triple $\{u,v,w\} \subseteq V$. These represent inclusion in the cyclic order relation. To remove the mirror symmetry, we fix one t_{uvw} to be always included by $t_{uvw} = 1$. Where there are six different ways for three vertices to be ordered with a linear order, three each correspond to the same of two permutations in a cyclic order, namely the clockwise and counterclockwise directions, which is why we only need a single binary variable for all of these. These triangle variables need to be constrained internally since they are transitively dependent on each other. Similar to the transitive order constraints, we add the constraints $t_{buw} \Leftarrow t_{buv} \wedge t_{bvw}$ and $t_{bwu} \Leftarrow t_{bwv} \wedge t_{bvu}$ for each $b \in V$ and $\{u,v,w\} \subseteq V \setminus \{b\}$. These constraints can also be optimized the same way the transitive order constraints were optimized. For constraining the crossing variables, we add $c_{\{uv,st\}} \Leftrightarrow t_{uvs} \oplus t_{uvt}$ for each disjunct $uv, st \in E$, meaning that s and t must lie on opposite sides of uv, similar to the betweenness formulation.

The triangle formulation also has an order-based formulation, by defining $t_{uvw} \Leftarrow a_{uv} \wedge a_{vw}$ for every distinct $u,v,w \in V$, yielding six constraints for each triple $\{u,v,w\} \subseteq V$.

While the triangle and betweenness formulations look similar, they are different in a fundamental way: in the way they encode the six linear orders for a triple $\{u,v,w\} \subseteq V$. Representing the order as a permutation, the triangle formulation merges the even and the odd permutations into a state each, whereas the betweenness formulation merges each two permutations with the same middle element into a state.

$$
\begin{aligned}
t_{uvw} &\in \mathbb{B} && \forall \{u,v,w\} \subseteq V : \{u,v,w\} \neq \{x,y,z\} \\
c_{\{e,e'\}} &\in \mathbb{B} && \forall \{e,e'\} \subseteq E \\
k &\in \mathbb{B} \\
t_{xyz} &\coloneqq 1 \\
t_{vwu}, t_{wuv} &\coloneqq t_{uvw} && \forall \{u,v,w\} \subseteq V : \{u,v,w\} \neq \{x,y,z\} \\
t_{wvu}, t_{vuw}, t_{uwv} &\coloneqq 1 - t_{uvw} && \forall \{u,v,w\} \subseteq V : \{u,v,w\} \neq \{x,y,z\} \\
0 &\geq t_{buv} + t_{bvw} + t_{bwu} - 2 && \forall b \in V : \forall \{u,v,w\} \subseteq V \setminus \{b\} \\
0 &\geq t_{bwv} + t_{bvu} + t_{buw} - 2 && \forall b \in V : \forall \{u,v,w\} \subseteq V \setminus \{b\} \\
c_{\{uv,st\}} &\geq t_{uvs} - t_{uvt} && \forall uv, st \in E : \{u,v\} \cap \{s,t\} = \emptyset \\
c_{\{uv,st\}} &\geq t_{uvt} - t_{uvs} && \forall uv, st \in E : \{u,v\} \cap \{s,t\} = \emptyset \\
k &\geq \sum_{e' \in E, e' \cap e = \emptyset} c_{\{e,e'\}} && \forall e \in E
\end{aligned}
$$

Figure 4: Triangle formulation with a fixed triple $\{x,y,z\} \subseteq V$

$$
\begin{aligned}
a_{uv} &\in \mathbb{B} && \forall \{u,v\} \subseteq V \setminus \{f\} : \{u,v\} \neq \{l,h\} \\
t_{uvw} &\in \mathbb{B} && \forall \{u,v,w\} \subseteq V : \{u,v,w\} \neq \{x,y,z\} \\
c_{\{e,e'\}} &\in \mathbb{B} && \forall \{e,e'\} \subseteq E \\
k &\in \mathbb{B} \\
a_{uf} &\coloneqq 1 && \forall u \in V \setminus \{f\} \\
a_{lh} &\coloneqq 1 \\
a_{vu} &\coloneqq 1 - a_{uv} && \forall \{u,v\} \subseteq V \\
0 &\geq a_{uv} + a_{vw} + a_{wu} - 2 && \forall \{u,v,w\} \subseteq V \\
0 &\geq a_{wv} + a_{vu} + a_{uw} - 2 && \forall \{u,v,w\} \subseteq V \\
t_{uvw} &\geq a_{uv} \wedge a_{vw} && \forall u,v,w \in V : u \neq v \neq w \\
c_{\{uv,st\}} &\geq t_{uvs} - t_{uvt} && \forall uv, st \in E : \{u,v\} \cap \{s,t\} = \emptyset \\
c_{\{uv,st\}} &\geq t_{uvt} - t_{uvs} && \forall uv, st \in E : \{u,v\} \cap \{s,t\} = \emptyset \\
k &\geq \sum_{e' \in E, e' \cap e = \emptyset} c_{\{e,e'\}} && \forall e \in E
\end{aligned}
$$

Figure 5: Ordered triangle formulation with fixed vertex $f = \max(V)$ and fixed pair $\{l,h\} \subseteq V \setminus \{f\}$

3.1.5 Index Formulation

When looking at ILP formulations, the values of some variables are completely defined by other variables. In order based formulations only the order variables, of which there are $\mathcal{O}(|V|^2)$, are not completely defined by other kinds of variables. In the triangle formulation the triangle variables, of which there are $\mathcal{O}(|V|^3)$, are the only not completely defined variables. The idea is that if we can reduce the number of free variables, we may be able to find a solution faster. The absolute minimum of free variables is in $\mathcal{O}(|V|)$.

We have the index variables $x_v \in [0, |V|-1]_{\mathbb{N}}$ for each $v \in V$, and order variables $a_{uv} \in \mathbb{B}$ for each $\{u,v\} \subseteq V$. Since no two vertices can inhabit the same position, they need to be constrained to be at least one apart via $x_v - x_u \geq 1 - |V| a_{vu}$ for each $u,v \in V$. The summand of $|V| a_{vu}$ is required as

$$
\begin{aligned}
x_v &\in [0, |V|-2]_{\mathbb{N}} && \forall v \in V \setminus \{f\} \\
a_{uv} &\in \mathbb{B} && \forall \{u,v\} \subseteq V \setminus \{f\} : \{u,v\} \neq \{l,h\} \\
c_{\{e,e'\}} &\in \mathbb{B} && \forall \{e,e'\} \subseteq E \\
k &\in \mathbb{B} \\
a_{uf} &\coloneqq 1 && \forall u \in V \setminus \{f\} \\
a_{lh} &\coloneqq 1 \\
a_{vu} &\coloneqq 1 - a_{uv} && \forall \{u,v\} \subseteq V \\
x_v - x_u &\geq 1 - (|V|-1)a_{vu} && \forall u,v \in V \\
c_{\{uv,st\}} &\geq a_{us} + a_{sv} + a_{vt} - 2 && \forall uv, st \in E : \{u,v\} \cap \{s,t\} = \emptyset \\
c_{\{uv,st\}} &\geq a_{ut} + a_{tv} + a_{vs} - 2 && \forall uv, st \in E : \{u,v\} \cap \{s,t\} = \emptyset \\
c_{\{uv,st\}} &\geq a_{vs} + a_{su} + a_{ut} - 2 && \forall uv, st \in E : \{u,v\} \cap \{s,t\} = \emptyset \\
c_{\{uv,st\}} &\geq a_{vt} + a_{tu} + a_{us} - 2 && \forall uv, st \in E : \{u,v\} \cap \{s,t\} = \emptyset \\
k &\geq \sum_{e' \in E, e' \cap e = \emptyset} c_{\{e,e'\}} && \forall e \in E
\end{aligned}
$$

Figure 6: Index formulation with fixed vertex $f = \max(V)$ and fixed pair $\{l,h\} \subseteq V \setminus \{f\}$

to allow u to have a larger index than v. This also yields all required defining constraints for the order variables. The crossing variables are constrained from the order variables like in the order formulation.

To remove the rotational symmetry, we fix one vertex $f \in V$ to the end of the order, setting $a_{vf} = 1$ for all $v \in V \setminus \{f\}$, allowing us to remove variable x_f, redefine the other index variables by $x_v \in [0, |V|-2]_{\mathbb{N}}$, and tighten the constraint $x_v - x_u \geq 1 - (|V|-1)a_{vu}$ for each $u,v \in V$. For the mirror symmetry, we fix the order of two vertices $\{l,h\} \subseteq V \setminus \{f\}$ via $a_{lh} = 1$, like in the order formulation.

3.1.6 Feasability

All above mentioned formulations can be converted into a feasability check where we fix k to a specific value. Using lower and upper bounds, we can perform a binary search in combination with this feasability check to find the minimum k. The upper bound can be found by a heuristic or by the local circular crossing number of the complete graph with the same number of vertices. The lower bound simply be 0 or calculated via a formula. In our case we used $|E| \leq \sqrt{16.875k}|V|$ by Pach and Tóth [6] to conclude $k \geq \frac{1}{16.875}\frac{|E|^2}{|V|^2}$. When checking for a fixed k, we only need to replace the k-constraints, since they are the only constraints that change, leaving the rest untouched. This also allows the solver to use its past state to build a possible solution upon.

3.2 Iterative Approaches

One of the simplest ways to implement NP-complete problems is to iterate through all possible solutions and selecting the best one. We look at two ways to represent the order of the vertices, both based on embedding them onto a line segment. A solution is represented in both cases by a map, in one from vertices to their positions in the other the reverse. The maps are generated as permutations that have a fixed element (in our case, the first) and where two other elements have a fixed order (in our case, the second element must be less than the third) to remove all symmetries. Iterating through these permutations can be done efficiently using Heap's algorithm [7]. To fix the order of two vertices, we just skip every permutation that does not have the two vertices in that order. While it sounds inefficient, the slow part is not iterating though the permutations, but determining their local circular

crossing number. As an additional optimization, we end the search if we have found a solution with a crossing number of zero.

3.2.1 Vertices to Positions

For a map from vertices to their positions, to calculate the local circular crossing number, we handle each edge uv seperately by counting the crossings with every other disjunct edge $st \in E$ by checking if exactly one vertex of s and t is between v and w to detect a crossing. We can precalculate for each edge the set of edges disjunct to it, enabling the check to only handle disjunct edges.

3.2.2 Positions to Vertices

For a map f from positions to their vertices, to calculate the local circular crossing number, we handle each pair of positions $\{a, b\}$ seperately by first checking if they make up an actual edge $\{f(a), f(b)\} \in E$ and then counting the number of actual edges between the outer positions $\{1...a-1, b+1...|V|\}$ and inner positions $\{a+1...b-1\}$. Using an adjecency matrix represented under the hood by a bitmap allows the extensive edge checks to be done quickly, with a constant time access and keeping the entire matrix in cache for small enough graphs.

4 Evaluation

We benchmarked all algorithms on the dataset Shevchenko used in their bachelor's thesis [5], which they obtained from the House of Graphs database [8]. Using the same dataset gives us a way to compare our approaches to the other ones they presented. This dataset consists of 2007 connected graphs with at most 10 vertices, of which 1326 graphs are biconnected. We evaluate the full set without first decomposing the graphs into their biconnected components.

For the heuristics, we look at the quality of the solution, since the run time is negligible for graphs of that size. For the exact solutions, we look at the time the algorithm takes to give an optimal solution. The machine used for benchmarking uses an Intel Core i9-10900K CPU, has 32GiB of DDR4-2133 RAM and runs on an operating system using the Linux kernel, version 6.17. To solve the ILP formulations, we used Gurobi [9] version 12.0.3. The iterative approaches are written in Rust, version 1.92.0, and compiled using the default release optimizations. We only used a single core, but the algorithms can easily be run on multiple cores with a few tweaks.

4.1 Heuristics

We compare the greedy generation against random permutations both before and after they have been improved using the improvement heuristic. The quality of the solution as shown in Figure 7 is the additive difference between the result of the heuristic and the true local circular crossing number of the graph. In most cases, the greedy generation and the improvement of a random permutation are

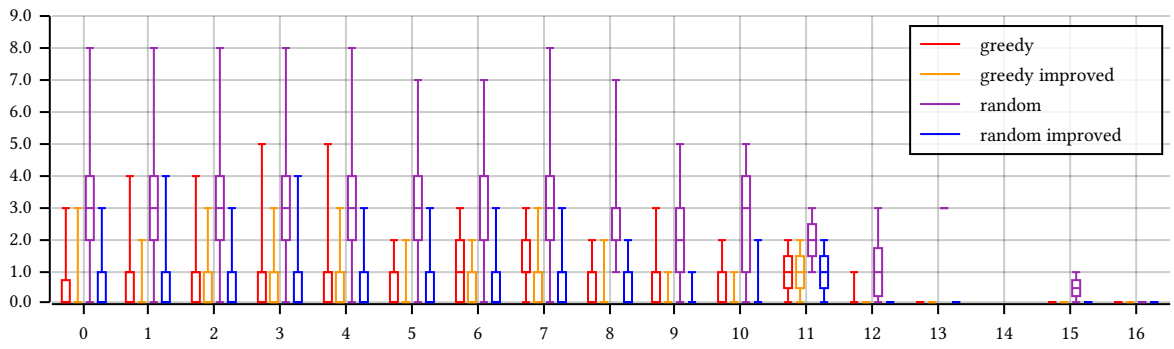


Figure 7: Quality of the solutions of the heuristics (y-axis) against local circular crossing number (x-axis)

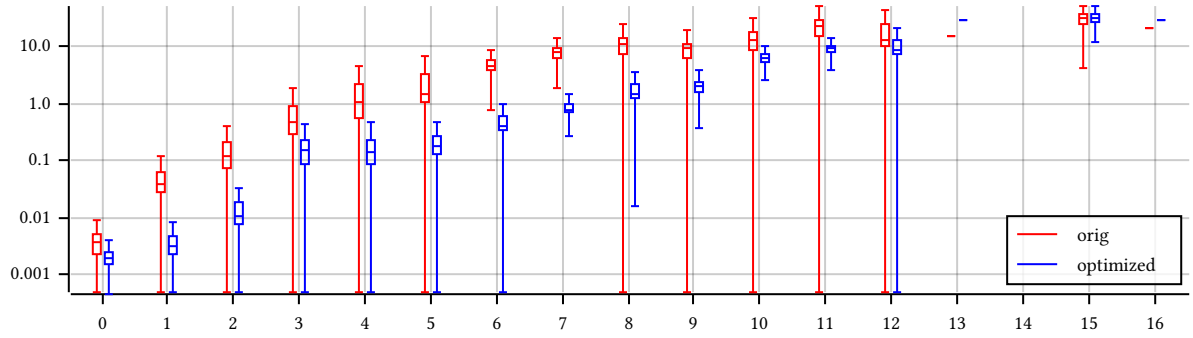


Figure 8: Time of the original and optimized ILP formulations in seconds

only off by one. In many cases, the greedy generation can be further improved using the improvement heuristic leading to an exact solution. The solutions for higher numbers must be interpreted with care, since each graph has at most 10 vertices, leaving not much room for a large differrence with almost complete graphs.

4.2 Exact Solutions

As seen in Figure 8, where we compare Shevchenko's original ILP formulation against our optimized one, removing the symmetries yields faster results in most cases, although interestingly the difference gets smaller the more complete a given graph is.

When comparing our other ILP formulations (Figure 9), we can see that the order and betweenness formulations equally fast. The ordered triangle formulation starts out relatively equal to the other order based formulations, but gets worse for more complete graphs. The triangle formulation is worse than all other three in most cases. Using valid inequalities improves the time only for graphs with a crossing number of one and only slightly at that, so we chose not to provide a figure for that.

For benchmarking, we used the improved greedy generation for determining the upper bound and once zero and once the above mentioned formula as a lower bound. In the current benchmarks both lower bounds are equally fast, so only the data from the formula approach is used in the figures. The feasability formulations compare to each other similar to their respective ILP formulations, as seen in Figure 10.

Comparing the feasability formulations to their respective ILP formulations, as seen in Figure 11, we see that the feasability formulations are much faster for graphs with small crossing number while getting slower than their ILP counterpart for larger numbers. The relative speed of the feasability formulations compared to the ILP formulations, even though they need to run almost the same formu-

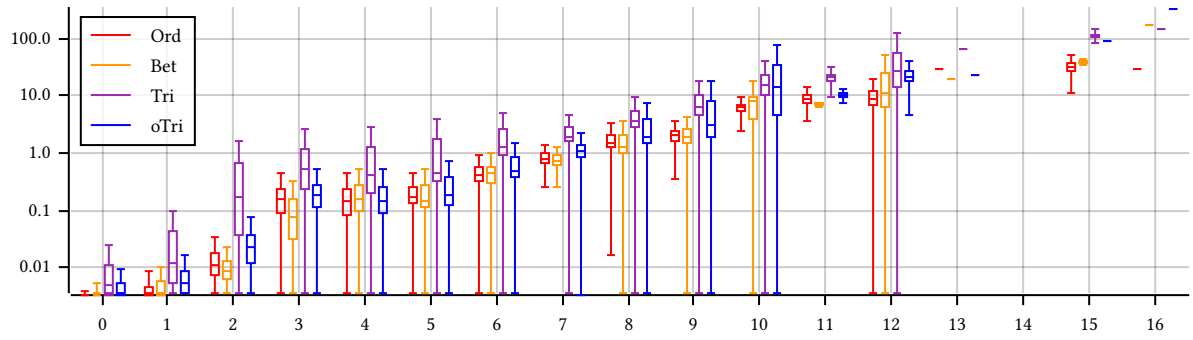


Figure 9: Time of our presented ILP formulations Order (Ord), Betweenness (Bet), Triangle (Tri) and Ordered Triangle (oTri) in seconds

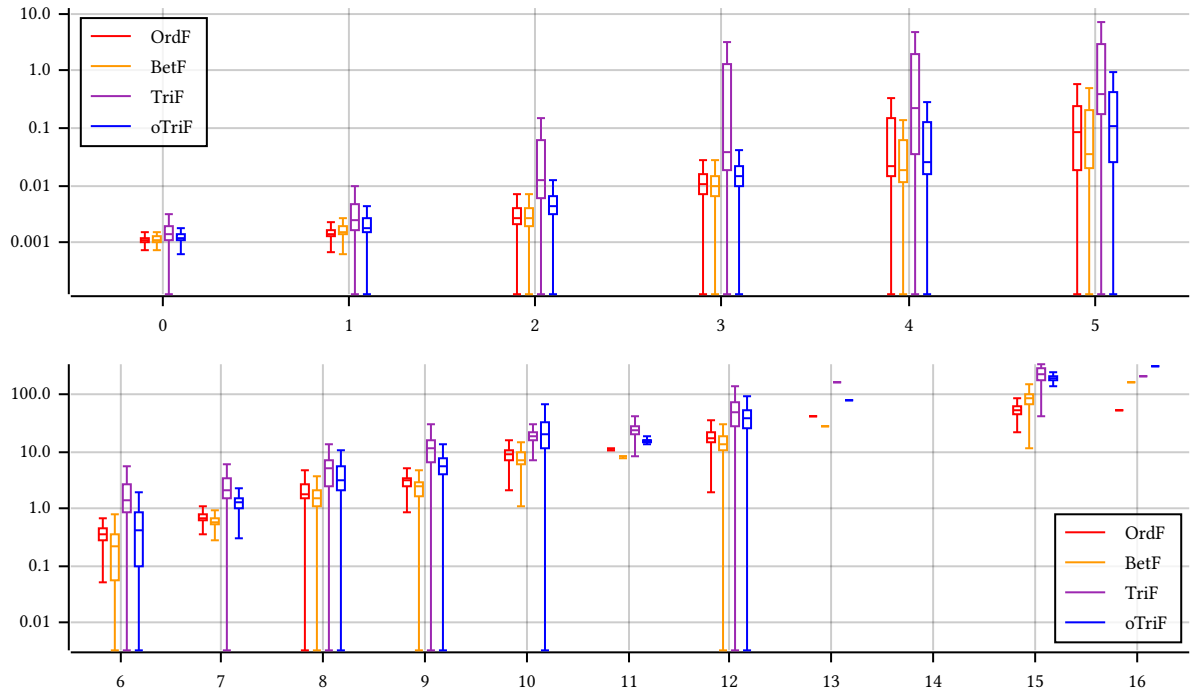


Figure 10: Time of the feasability formulations in seconds

lation multiple times, hints at the k-constraints being a large factor for slowing the ILP formulations down.

Against the hopes above, the index formulation fared worse than the order formulation, as seen in Figure 12. This may come from the unconstrained variables being no longer binary, still leaving, with optimizations, $(|V|-1)$ different values for each vertex, resulting in $(|V|-1)^2$ different states

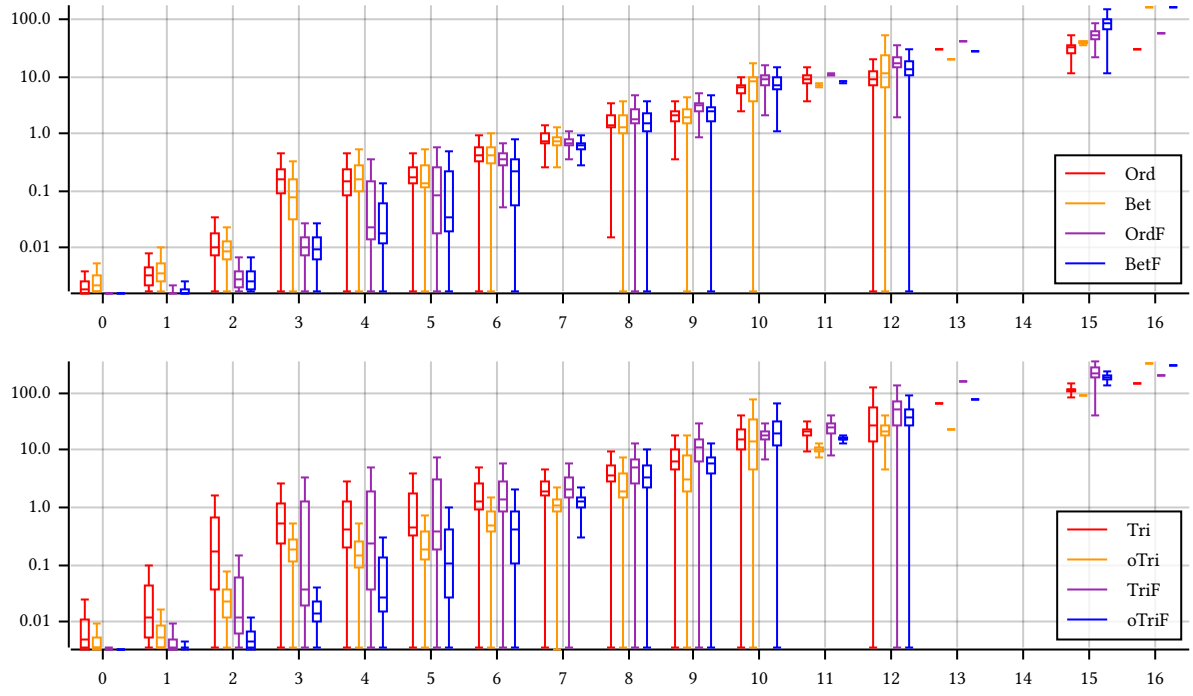


Figure 11: Time of the ILP and feasability in seconds

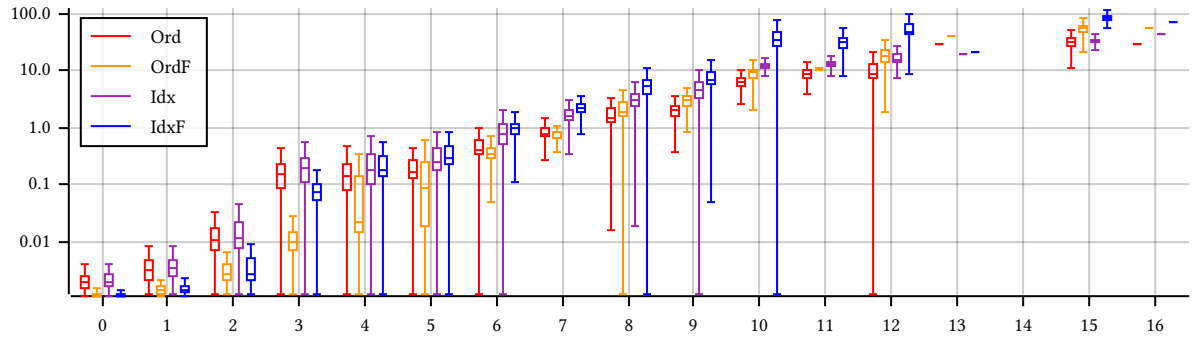


Figure 12: Time of the Order and Index formulation with their respective feasability formulation

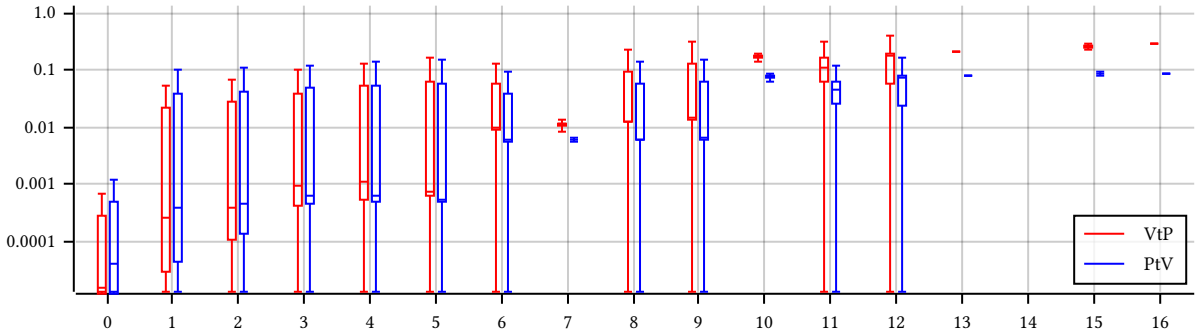


Figure 13: Time of the Vertex to Position (VtP) and Position to Vertex (PtV) approaches in seconds

before applying further constraints, even more than the $(|V|-1)(|V|-2)$ different states for the order variables.

Looking at the iterative approaches, while slightly slower for graphs with small crossing number, the position to vertex approach is quickly slightly faster for graphs with a larger crossing number than the vertex to position approach, as seen in Figure 13.

The more interesting part is when we compare the ILP formulations to the iterative approaches, as seen in Figure 14. The figure is split in two as to better see the differences for small values. The iterative approaches are on average orders of magnitude faster and get faster relatively to the ILP formulations the more complete a given graph is, not to mention the memory footprint being much smaller, which might contribute to the speed, allowing more or even all of the computational data to be stored in cache.

For graphs with at least 13 vertices, no approach shown here was able to find a solution in under four hours, after which we terminated the search.

5 Conclusion

In this work, we presented two heuristics, four ILP formulations and two iterative algorithms for finding the local circular crossing number. We improved Shevchenko's original ILP formulation mainly by removing two kinds of symmetries. Still, almost brute-forcing the solution is faster by far.

Every exact solution approach shown here still cannot handle larger graphs. The ILP formulations could be improved, possibly by improving the current k-constraints that possibly play a part in the slow run time. For improving the iterative approaches, one could make use of the fact that when iterating through the permutations only two vertices are swapped, instead of recalculating everything for each permutation.

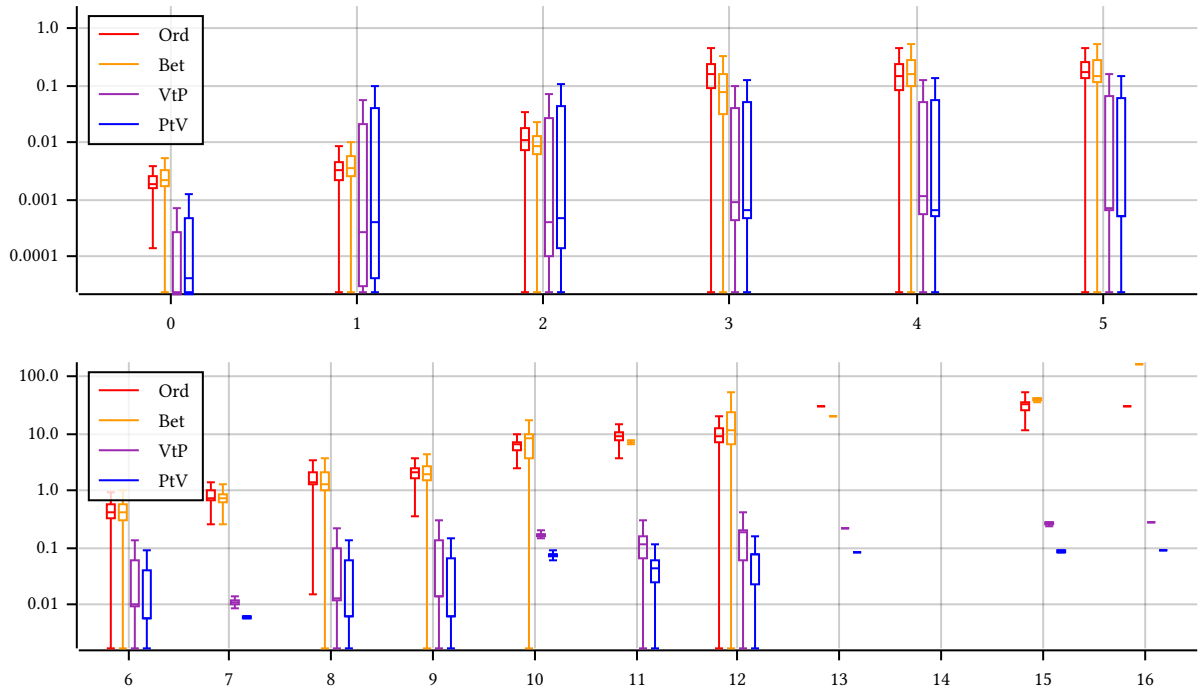


Figure 14: Time of the Order (Ord) and Betweenness (Bet) formulations, compared to the Vertex to Position (VtP) and Position to Vertex (PtV) approaches in seconds

Bibliography

[1] S. Masuda, T. Kashiwabara, K. Nakajima, and T. Fujisawa, "On the NP-completeness of a computer network layout problem.," *Proceedings of the 1987 IEEE international symposium on circuits and systems.*, pp. 292–295, 1987.

[2] S.-H. Hong and H. Nagamochi, "A linear-time algorithm for testing full outer-2-planarity," *Discrete Applied Mathematics*, vol. 255, pp. 234–257, 2019, doi: 10.1016/j.dam.2018.08.018.

[3] S. Chaplick, M. Kryven, G. Liotta, A. Löffler, and A. Wolff, "Beyond Outerplanarity," in *Graph Drawing and Network Visualization*, F. Frati and K.-L. Ma, Eds., Cham: Springer International Publishing, 2018, pp. 546–559. doi: 10.1007/978-3-319-73915-1_42.

[4] Y. Kobayashi, Y. Okada, and A. Wolff, "Recognizing 2-Layer and Outer k-Planar Graphs," in *41st International Symposium on Computational Geometry (SoCG 2025)*, O. Aichholzer and H. Wang, Eds., in Leibniz International Proceedings in Informatics (LIPIcs), vol. 332. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, pp. 65:1–65:16. doi: 10.4230/LIPIcs.SoCG.2025.65.

[5] I. Shevchenko, "Practical Aspects of Recognising Outer k-Planar Graphs," Bachelor's Thesis, 2025.

[6] J. Pach and G. Tóth, "Graphs drawn with few crossings per edge," *Combinatorica*, vol. 17, no. 3, pp. 427–439, Sept. 1997, doi: 10.1007/BF01215922.

[7] B. R. Heap, "Permutations by Interchanges," *The Computer Journal*, vol. 6, no. 3, pp. 293–298, 1963, doi: 10.1093/comjnl/6.3.293.

[8] K. Coolsaet, S. D'hondt, and J. Goedgebeur, "House of Graphs 2.0: A Database of Interesting Graphs and More," *Discrete Applied Mathematics*, vol. 325, no. C, pp. 97–107, 2023, doi: 10.1016/j.dam.2022.10.013.

[9] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual." [Online]. Available: https://www.gurobi.com/