Ukrainian Catholic University

Bachelor Thesis

Practical Aspects of Recognising Outer k-Planar Graphs

Author:
Ivan Shevchenko

Supervisor:
Prof. Dr. Alexander Wolff

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Department of Computer Sciences and Information Technologies
Faculty of Applied Sciences

Lviv 2025

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

Practical Aspects of Recognising Outer k-Planar Graphs

by Ivan Shevchenko

Abstract

Planar graphs have been thoroughly studied for decades as one of the simplest graph classes. The notion of planarity, however, imposes too harsh restrictions on the structure, significantly limiting the number of planar graphs. As a result, the research started shifting towards graphs that allow for a few crossings. One such class is the class of *outer k-planar* graphs, that is, the graphs that admit a circular drawing with straight-line edges and at most k crossings per edge. So far, their recognition has been discussed in a purely theoretical context, lacking practical validation. In this work, we focus on the practical aspect of the problem by implementing three different algorithms for recognising outer k-planar graphs. The first implementation is based on a formulation of the recognition problem as an integer linear program. The second implementation solves the problem by reducing it to the satisfiability of a Boolean formula. The third implementation follows a recently introduced algorithm based on dynamic programming. Comparing the resulting implementations, we found out that the one based on integer linear programming works best for most graphs due to the efficiency of the existing solvers and the simplicity of the encoding. The algorithm based on satisfiability is more complex, requiring an exponential number of clauses in terms of k. Finally, the algorithm based on dynamic programming is faster than the other two approaches only for small values of k (at most 3). Also we demonstrated the positive effect of biconnected decomposition on the performance of the algorithms. In the future, these implementations can be used to solve the recognition problem for small instances and to verify new implementations.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor, Prof. Dr. Alexander Wolf, who, together with Yuto Okada, guided me through every stage of writing this thesis. I am incredibly thankful for your invaluable support, commitment, and encouragement throughout our collaboration. This work would not have been possible without you. I want to thank Otfried Cheong for creating a wonderful drawing editor Ipe, that was used to create illustrations for this thesis. I am profoundly grateful to my family and closest friends for their unwavering love and support over the past years. You have shaped me into the person I am today. My sincere appreciation goes to the Faculty of Applied Sciences and the Ukrainian Catholic University as a whole for the growth I had the opportunity to experience – professional at the former and personal at the latter. I will always cherish the sense of community I felt within the walls of this university. Last but certainly not least, I want to express my deepest respect and gratitude to the Armed Forces of Ukraine. Your courage and sacrifice allow us to study, work, and simply live our lives. Without your heroism, none of this would be possible.

Contents

Chapter 1

Introduction

Nowadays, graphs are widely used across different domains and occupations. They prove to be an efficient tool for visualising relational systems. However, as the size of the graphs grows, so does their complexity, which makes understanding their internal structure increasingly difficult.

This problem was acknowledged by the community as early as in the 1980s. One of the key points discussed by the researchers in the field was the importance of reducing the edge crossings for improving visual clarity [2]. The suspicion that a drawing with fewer edge crossings was easier to comprehend was later confirmed by several experimental studies [25]. These studies showed that minimising the crossings in graph representations significantly improves the ability of humans to interpret the structure, particularly when dealing with complex or large graphs.

This led to the exploration of the ideal form for crossing minimising drawings – planar ones. However, requiring the drawing to be completely crossing free imposes severe limitations on an underlying graph. While providing a clean structure, these restrictions are often too constraining for many real-world graphs, especially large ones. As a consequence, many processes cannot be represented in a planar manner, so no insights acquired by the decades-long studies are applicable to them.

As a result, the research community expresses a growing interest in exploring graphs whose drawings are beyond planar; see the survey by Didimo et al. [10]. This led to an exploration of alternative approaches for eliminating edge crossings, such as drawings which include only parts of the edges [5, 4], or merge several edges to form a single track [9]. However, the most popular approach is to consider "almost" planar graphs – graphs which admit a drawing with a limited number of crossings. Such graphs offer a balance between visual recognisability and structural flexibility while still retaining some of the beneficial structural properties of planar graphs.

The most natural way to impose a limit on the number of crossings is to place an upper bound on the number of crossings per edge for some value k. This class of graphs is among those that have attracted the most interest from the research community lately [16]. In this work, however, we went a step further, additionally restricting the vertices to lie on a circle. Drawings that comply with both restrictions are called *outer k-planar*, and so are the graphs that admit such drawings.

1.1 Our contribution

However, despite all recent advances in this area, all work has been conducted in a theoretical context. Algorithms have not been implemented or tested empirically. Thus, they lack practical validation. In this thesis, we want to address this gap by implementing the most recent recognition algorithm and by introducing two alternative approaches based on Integer Linear Programming (ILP) and Satisfiability (SAT). While it is NP-hard to solve the general integer linear programming problem or to find a satisfying truth assignment for general Boolean formulas, there are very advanced solvers that can help us find the exact solutions for small- to medium-sized instances within an acceptable amount of time. We also evaluate the performance and efficiency of these methods, demonstrating their practical applicability and their limitations.

1.2 Structure of the thesis

- Chapter 2 discusses the state of the art by reviewing the main pieces of work in the context of planar and almost planar graph drawings. Here, we also discuss the complexity of the problem.
- Chapter 3 describes three algorithms for recognising outer k-planar graphs. Additionally, it presents the optimisations we used to improve the performance of the implemented methods.
- Chapter 4 describes the experiments that we conducted to evaluate the performance of the implemented algorithms and discusses the results that we obtained.
- Chapter 5 summarises the key accomplishments and overviews the results. It also discusses the limitations of our implementations and outlines potential improvements that can be made to overcome them.

Chapter 2

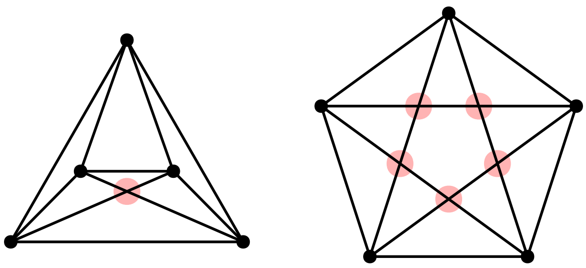
Related Work

As we discussed in Chapter 1, to get an easily interpretable graph drawing, one should minimise the number of crossings in it. The first algorithms that could do so were designed to recognise planar graphs and construct their drawing. The first linear-time algorithm for recognition of planar graphs was introduced as early as 1974 by Hopcroft and Tarjan [20]. In this chapter, we discuss studies that address the recognition problem for almost planar graphs.

2.1 Difficulty of dealing with beyond-planar graphs

Most relaxations of strict planarity dramatically increase the complexity of recognising such graphs. So, the general problem of minimising edge crossings in a graph drawing was known to be computationally intractable already in 1983 when Garey and Johnson [15] demonstrated that the Crossing Number problem, where the task is to check whether a given graph can be drawn with at most k crossings, is NP-hard. Their proof relies on a reduction from the Optimal Linear Arrangement problem, which is known to be NP-hard.

Minimising the number of local crossings is also hard. Korzhik and Mohar [23] showed that testing 1-planarity, that is, recognising whether a graph can be drawn with at most one crossing per edge, is NP-hard. Later, Cabello and Mohar [6] showed that testing 1-planarity is NP-hard even for near-planar graphs, that is, graphs that can be obtained from planar graphs by adding a single edge.



(a) 1-planar drawing (b) Outer 2-planar drawing

Figure 2.1: Drawing of K_5 (a) non-restricted and (b) restricted to a circular setting. Edge crossings are emphasised in red

Given the complexity of recognising k-planarity, researchers have considered exploring more restrictive settings, hoping that the imposed limitations could simplify the recognition. One of the considered restrictions is a circular setting that requires the vertices to be placed on a circle and the edges to be drawn as straight lines. This restriction gives rise to the *local circular crossing number* of a graph, which we study in this thesis. The graphs whose local circular crossing number is bounded by k are called *outer k-planar* graphs. For example, K_5, the complete graph with five vertices, is known to be the smallest non-planar graph. It can be drawn with a single crossing, hence it is 1-planar (see Figure 2.1a). When insisting on the circular setting, however, every second edge receives two crossings, hence K_5 is outer 2-planar (see Figure 2.1b).

2.2 Efficient recognition of some outer k-planar graphs

Recently Kobayashi et al. [22, Theorem 19] showed that the recognition problem for outer k-planar graphs is XNLP-hard with respect to the natural parameter k. In other words, it is unlikely that an algorithm that runs in $f(k) \cdot n^{O(1)}$ time for solving this problem exists. Here, f represents any computable function, and n represents the size of the input. Despite that, efficient algorithms for some constant values of k have been developed.

For $k = 0$, the recognition task simplifies to an outerplanarity test. Recognition can be accomplished by augmenting the graph with a new vertex connected to all original vertices and testing whether the resulting graph is planar. An alternative approach, described by Wiegers [27], introduces the concept of 2-reducible graphs, which are totally disconnected or can be made totally disconnected by repeatedly deleting edges adjacent to a vertex of degree at most two. The proposed outerplanarity test is based on an algorithm for testing 2-reducibility.

In the case of $k = 1$, two research groups independently presented linear-time algorithms [1, 18]. Both algorithms use the SPQR-decomposition of a graph for the test. Notably, the latter solution extends the graph to a maximal outer 1-planar configuration if such a drawing exists, unlike the former, which employs a bottom-up strategy which does not require any transformations of the original graph.

Considering a special case of this problem, Hong and Nagamochi [19] proposed a linear-time algorithm for recognising full outer 2-planar graphs. An outer k-planar drawing is *full* if no crossings lie on the boundary of the outer face. Later, Chaplick et al. [7] extended the result of Hong and Nagamochi by introducing an algorithm for recognising full outer k-planar graphs for every k. Their algorithm runs in $f(k) \cdot n$ time, where f is a computable function.

For the general version of the problem and values of $k > 1$, no research has been conducted until recently when a group of researchers proposed an algorithm for the general case, which we discuss in Section 2.3.

2.3 Recognising general outer k-planar graphs

For a given outer k-planar drawing of a graph, Firman et al. [11] proposed a method for constructing a triangulation with the property that each edge of the triangulation is crossed by at most k edges of the drawing. Since the edges of the triangulation do not necessarily belong to the original graph, they are termed *links* to distinguish them from the original graph edges. The construction is done recursively.

Initially, the algorithm selects an edge on the outer face and labels it as the active link. At each recursive step, the active link partitions the graph into two regions: a left part already triangulated and a right part not yet explored. The objective of each step is to triangulate the right portion. To achieve this, a splitting vertex is chosen within the right region, dividing it into two smaller subregions. The splitting vertex is selected so that the two new links, connecting the split vertex with the endpoints of the active link, are each intersected by at most k edges, which allows including them into the triangulation. The algorithm then recurses, treating each of these newly formed links as the active one.

Later, Kobayashi et al. [22] exploited this approach to address the recognition problem for outer k-planar graphs. In contrast to the triangulation task, where the drawing is provided, the recognition problem requires determining whether a given graph admits an outer k-planar drawing. Although the core idea remains analogous, the absence of a drawing requires the exploration of all possible configurations. Here, each recursive step verifies whether the unexplored right portion of the graph can be drawn as an outer k-planar graph that is compatible with the left part.

Moreover, instead of relying on recursion, the method utilises a dynamic programming approach. This framework combines solutions of smaller subproblems retrieved from a table to solve larger ones. To populate this table, the algorithm iterates over all possible configurations corresponding to different recursion steps. Several parameters characterise each such configuration. The first parameter is the active link – a pair of vertices that divides the graph into a left and a right region. The second parameter is the set of vertices in the right part, which is not uniquely determined as in the triangulation case. Additionally, the configuration depends on the order in which edges intersect the active link and the number of intersections on the right side for each one of them. These parameters are used to ensure that the drawing of the right part is compatible with the left part. For each configuration, the algorithm considers all possible ways to split the right region further. For each of these splits, the method checks whether they are compatible with each other and with the left part of the drawing.

Using the restriction on the number of edges crossing each link, the authors demonstrated that, for a fixed k, the number of possible right subgraphs grows only polynomially with respect to the size of the graph. They proceeded by arguing that the overall time complexity of the algorithm is $2^{O(k \log k)} n^{3k+O(1)}$, showing that the algorithm is efficient for any fixed parameter k. This indicates that this problem belongs to the XP class, a class of problems that admit such "slicewise polynomial" algorithms.

Chapter 3

Proposed solution

As discussed in Chapter 2, the studies conducted in the field of beyond planar graphs lie purely in the theoretical domain, lacking practical validation. In this thesis, we aim to address this gap by implementing three different algorithms for computing the local circular crossing number of a given graph. Additionally, we aim to develop a command-line interface that would allow invoking the implemented algorithms, receiving the graph and a method as an input and returning the local circular crossing number k and a circular drawing of the provided graph with at most k intersections per edge as an output.

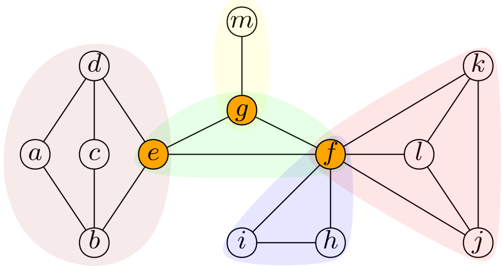
For operations with graphs, we use the C++ Boost Graph Library [26]. As a graph class, we use adjacency_list since, for all methods described below, we require both VertexList and EdgeList concepts to be able to iterate over both vertices and edges. We prefer this class to an adjacency_matrix as, according to the library documentation[1], it trades memory consumption and speed of graph traversal for the speed of edge insertion and deletion, and neither of these operations is used for algorithms described in this chapter. To represent a circular graph drawing, we simply use the sequence of vertices in the order in which they appear on the circle.

3.1 Bicomponent decomposition

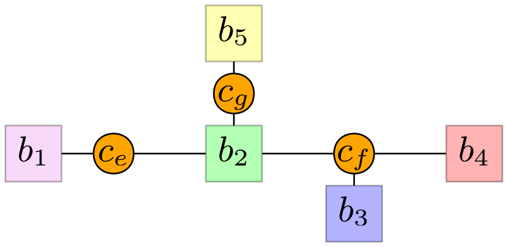
For complex problems, a decomposition into smaller subproblems often leads to a significant increase in performance. In our context of recognising outer k-planar graphs, an effective strategy to do so is to partition the graph into subgraphs in such a manner that allows us to process each part independently by the recognition algorithm. One of the plausible ways to accomplish this is to split the graph into biconnected components using block-cut decomposition, as shown in Figure 3.1a. It is worth noting that each edge of the graph belongs to a single biconnected component, referred to as a block. However, any two bicomponents may share a vertex, referred to as a cut vertex. Considering blocks and cut vertices as graph nodes, we can construct a so-called block-cut tree, wherein a block node is connected to a cut node if and only if the corresponding biconnected component contains a corresponding cut vertex, see Figure 3.1b.

Due to the nature of bi-connectedness, after getting outer k-planar drawings of all biconnected components separately, we can combine them easily into an outer k-planar drawing of the whole graph. To be more specific, if some component does not admit an outer k-planar drawing, neither does the whole graph. Otherwise, if for all of them such a drawing exists, they can be merged by combining duplicates of each cut vertex see Figure 3.1c and Figure 3.1d. This merging process does not introduce any additional edge crossings since both components are located on the outer face of each other. Moreover, as no new faces are created during this process (due to the

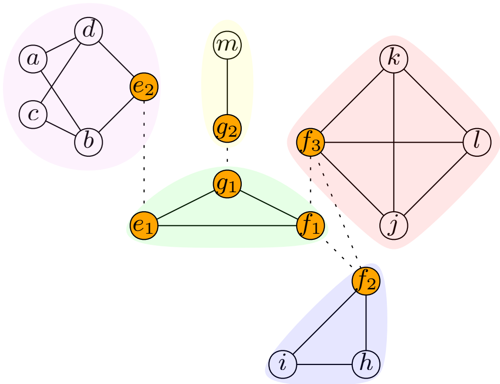
[1] https://www.boost.org/doc/libs/1_88_0/libs/graph/doc/adjacency_matrix.html



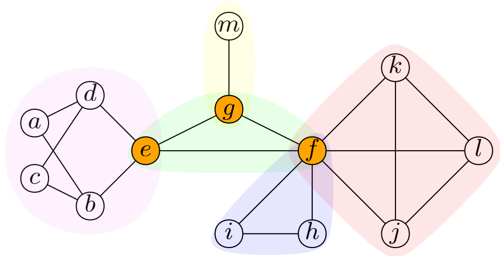
(a) An original graph with highlighted biconnected components and cut vertices



(b) Block-cut tree of a graph



(c) Outer k-planar drawing of each component



(d) Outer k-planar drawing of the original graph

Figure 3.1: An example of a bicomponent decomposition

acyclic structure of the block-cut tree), every vertex remains on the outer face of the graph during this process. Consequently, the resulting drawing of an original graph is outer k-planar, and it exists if and only if each biconnected component of the graph admits such a drawing.

In this work, we implemented this decomposition using the method biconnected_components[2] from the Boost Graph Library [26]. This function assigns an index of the bicomponent to each edge to which it belongs. Additionally, it provides a list of cut vertices. Afterwards, we copy each block as an independent graph and create mappings to translate new *local* vertices back to their original identifiers. Finally, we construct a supergraph representing the structure of a block-cut tree wherein each node references a copied block alongside corresponding mapping or a cut vertex.

To construct a drawing of the whole graph, after performing the decomposition, we perform a depth-first search on the block-cut tree, recording the predecessor for each node upon discovery. Additionally, each time a block vertex is discovered, we use one of the methods described in Sections 3.2, 3.3 and 3.5 to check whether the component admits an outer k-planar drawing and obtain it if so. Afterwards, we merge the new drawing with the already existing one by combining the common cut vertex if such exists. To be more specific, if the considered block is the first encountered one, its drawing is directly copied into a sequence that will form the final drawing. Otherwise, the block necessarily has a predecessor. Due to the structure of a tree, it is a cut node corresponding to a vertex that is shared with some other block. Due to how we traverse the tree, that other block has already been considered and thus added to a final drawing. As a result, the corresponding cut vertex is present in both global and local drawings. Since each drawing is represented as a cyclic sequence of

[2] https://www.boost.org/doc/libs/1_87_0/libs/graph/doc/biconnected_components.html

vertices, we can rotate the local one so that the corresponding cut vertex appears as the first one in a sequence. Finally, we insert the local drawing starting from the second element into the global one immediately after the corresponding cut vertex.

3.2 ILP-based algorithm

The problem of recognising the outer k-planar graphs is NP-hard. This justifies the use of exponential-time methods to approach our problem. By doing so, we benefit from decade-long studies conducted for these problems that resulted in the development of extremely optimised algorithms for their solving. One of them is an Integer Linear Programming problem (ILP). This problem asks to find a vector $\mathbf{x}$ that optimises the linear objective $\mathbf{c}^T\mathbf{x}$ subject to specific constraints $\mathbf{A}\mathbf{x} \leqslant \mathbf{b}$. Additionally, some variables in the ILP problem are restricted to integer values. Since researchers have extensively studied the problem and developed efficient solvers, we decided to use their results to build an algorithm for recognising outer k-planar graphs. This section discusses the process we use to encode the recognition task as the ILP problem.

As an implementation of ILP solver, we used Gurobi Optimizer [17] under the free academic licence. We chose it due to its outstanding performance demonstrated by Luppold et al. [24].

To encode a recognition problem to an ILP, we have to represent its structure using variables and constraints. We start with a graph drawing, which is represented, as described above, as a sequence of vertices. For the ILP, we can encode it using the so-called *ordering* variables, which indicate a relative order of two vertices. Specifically, for every pair of vertices u and v, we create a binary variable $a_{u,v}$ introducing Equation (3.18) as a constraint for the ILP problem. We interpret the value 1 as an indication of vertex u being located before vertex v and the value 0 as an indication of either v being located before u or u and v being the same vertex.

To ensure that these variables encode a valid sequence, we also have to enforce the transitivity. That is, for every ordered pair of distinct vertices u and w, and every other vertex v, if $a_{u,v} = 1$ and $a_{v,w} = 1$, meaning u is located before v and v is located before w, then u must be located before w, so the following should hold $a_{u,w} = 1$. Including also the implication for the reversed order, we get:

$$a_{u,v} = 1 \wedge a_{v,w} = 1 \Longrightarrow a_{u,w} = 1 \tag{3.1}$$
$$a_{u,v} = 0 \wedge a_{v,w} = 0 \Longrightarrow a_{u,w} = 0 \tag{3.2}$$

If we instead consider a pair w, u and the same vertex v, the constraints would look like follows:

$$a_{w,v} = 1 \wedge a_{v,u} = 1 \Longrightarrow a_{w,u} = 1 \tag{3.3}$$
$$a_{w,v} = 0 \wedge a_{v,u} = 0 \Longrightarrow a_{w,u} = 0 \tag{3.4}$$

Note that for any distinct vertices x and y the equality $a_{x,y} = 1 - a_{y,x}$ always holds, thus Equations (3.1) and (3.4) alike Equations (3.2) and (3.3) are equivalent. Consequently, it is enough to ensure only the first constraint as long as we do it for every ordered pair of vertices. Considering that the variables are binary, to limit $a_{u,w}$ to 1 it is enough to impose a constraint $a_{u,w} \geqslant \epsilon$ for any $\epsilon \in (0; 1]$. In a constraint for ILP, this ϵ must be represented as a linear function of $a_{u,v}$ and $a_{v,w}$. The values of this function must lie in the half-interval $(0; 1]$ if and only if both binary variables are 1. Using an expression $a_{u,v} + a_{v,w} - 1$ for this leads to Equation (3.7) in the encoded ILP formulation below.

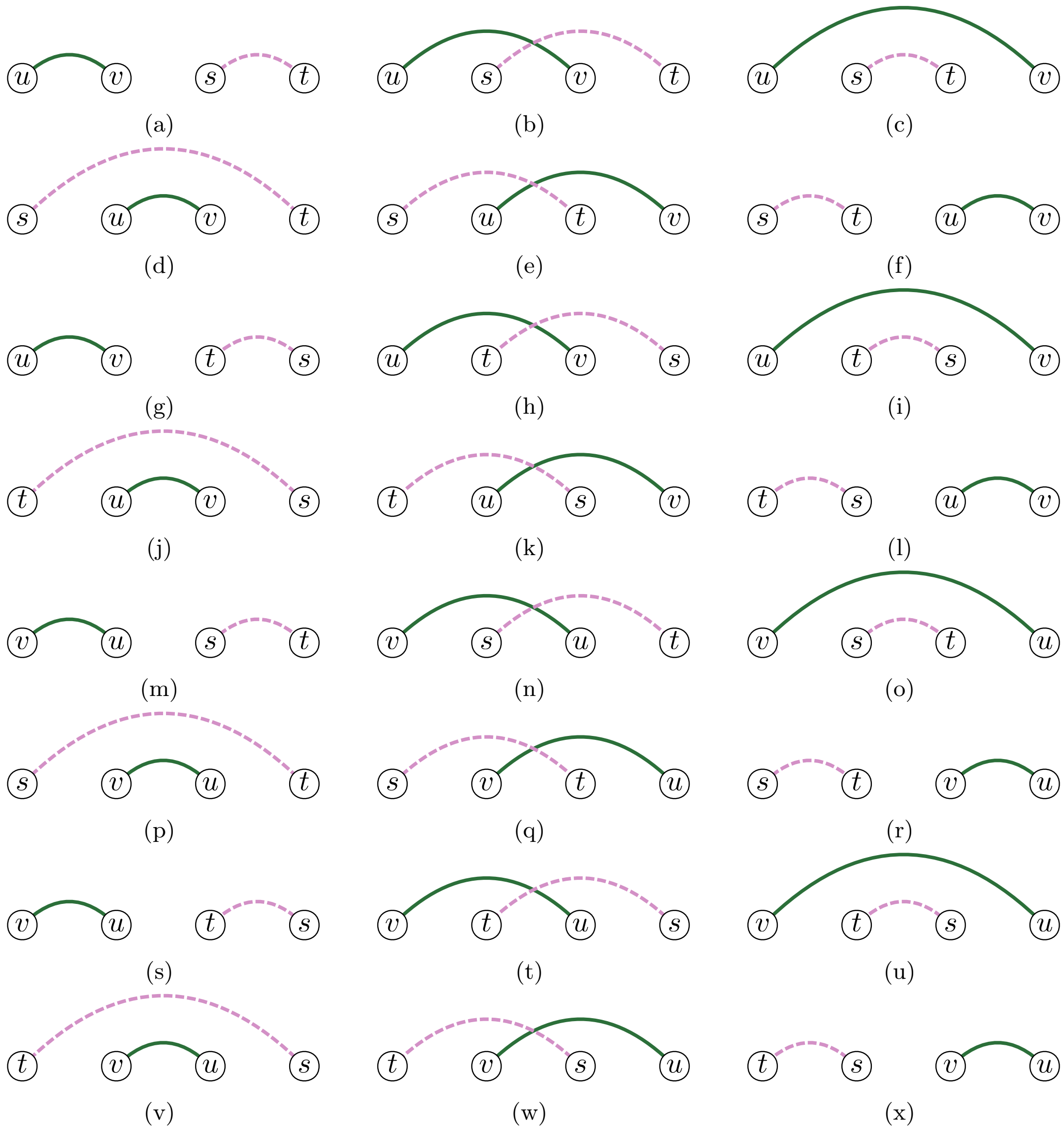


Figure 3.2: All 24 possible arrangements of the endpoints of two edges, only 8 of which result in an intersection; see the central column.

The next step of the algorithm is to encode the intersections. To represent them, for every unordered pair of edges, uv and st, we introduce a binary *crossing* variable $c_{uv,st}$, hence Equation (3.17) representing this constraint. The endpoints of the edges can be arranged in 24 different ways, among which only eight result in an intersection, as demonstrated in Figure 3.2. To encode this, we must ensure that the value of the variable $c_{uv,st}$ equals 1 if the corresponding ordering variables indicate one of these eight arrangements[3]. For example, considering the arrangement in Figure 3.2b, we have to limit the value of $c_{uv,st}$ to 1 if the endpoints are arranged in the order $usvt$. This order of vertices is implied by the model if and only if each equivalence out of $a_{u,s} = 1$, $a_{s,v} = 1$ and $a_{v,t} = 1$ holds. Thus, we can represent this limitation as follows:

$$a_{u,s} = 1 \land a_{s,v} = 1 \land a_{v,t} = 1 \implies c_{uv,st} = 1$$

[3]As the objective of the program is to minimise the number of crossings, we do not constraint $c_{uv,st}$ to 0 when uv and st do not cross, leaving this to the optimiser. Doing so, we simplify the problem by reducing the number of constraints for every pair of edges from 24 to 8.

To transform this into a constraint for an ILP formulation, we can use the same logic as for encoding Equation (3.1), getting as a result Equation (3.8). Equations (3.9) to (3.15) are constructed analogously for the other seven intersecting arrangements.

Lastly, the algorithm has to encode each edge's crossing number and minimise the maximal value out of them. The crossing number of each edge can be easily represented as a sum of the corresponding crossing variables:

$$cr_e \leqslant \sum_{e' \in E(G)} c_{e,e'} \tag{3.5}$$

However, as the maximum is not a linear function, constructing the objective function out of the per-edge crossing numbers is not as simple. To get around this limitation, we have to introduce a new continuous variable k, which represents the crossing number of the whole graph G. To ensure that, we have to bound k from below by the crossing number of each edge: $k \geqslant cr_e \forall e \in E(G)$. Combining this with Equation (3.5), we get Equation (3.16) as a constraint for the ILP problem. As a result, minimising for k would give the desired result.

Combining everything together, we get the following formulation of the ILP problem:

$$\begin{aligned}
&\textbf{minimize} && k && && (3.6)\\
&\textbf{subject to} && a_{u,v} \geqslant a_{u,v} + a_{v,w} - 1, && \forall u, v, w \in V(G) && (3.7)\\
& && c_{uv,st} \geqslant a_{u,s} + a_{s,v} + a_{v,t} - 2, && \forall uv, st \in E(G) && (3.8)\\
& && c_{uv,st} \geqslant a_{u,t} + a_{t,v} + a_{v,s} - 2, && \forall uv, st \in E(G) && (3.9)\\
& && c_{uv,st} \geqslant a_{v,s} + a_{s,u} + a_{u,t} - 2, && \forall uv, st \in E(G) && (3.10)\\
& && c_{uv,st} \geqslant a_{v,t} + a_{t,u} + a_{u,s} - 2, && \forall uv, st \in E(G) && (3.11)\\
& && c_{uv,st} \geqslant a_{s,u} + a_{u,t} + a_{t,v} - 2, && \forall uv, st \in E(G) && (3.12)\\
& && c_{uv,st} \geqslant a_{t,u} + a_{u,s} + a_{s,v} - 2, && \forall uv, st \in E(G) && (3.13)\\
& && c_{uv,st} \geqslant a_{s,v} + a_{v,t} + a_{t,u} - 2, && \forall uv, st \in E(G) && (3.14)\\
& && c_{uv,st} \geqslant a_{t,v} + a_{v,s} + a_{s,u} - 2, && \forall uv, st \in E(G) && (3.15)\\
& && k \geqslant \sum_{st \in E(G)} c_{uv,st}, && \forall uv \in E(G) && (3.16)\\
& && c_{uv,st} \in \{0, 1\}, && \forall uv, st \in E(G) && (3.17)\\
& && a_{u,v} \in \{0, 1\}, && \forall u, v \in V(G) && (3.18)
\end{aligned}$$

To estimate the complexity of the formulation, we can calculate the number of constraints used in the encoding. So considering them in the order they are mentioned above, we have $O(n^3) + 8 \cdot O(m^2) + O(m) + O(m) + O(n) = O(n^3 + m^2)$ constraints, where n is the number of vertices and m is the number of edges in the input graph. Note that the size of the ILP formulation does not depend on the graph's local circular crossing number.

After encoding the problem as an ILP problem, as we described above, we run the solver. After optimisation, it returns assigned values for each variable used in the program. The variables of interest for us are ordering variables $a_{u,v}$ and k. The latter one indicates the minimal possible crossing number that is reported. The former one we use to reconstruct an outer k-planar drawing of the original graph. As the desired drawing is a sequence of vertices, we must order them. To do so, we can use the values of ordering variables to define the strict total order relation. We say that for two vertices u and v $u < v$ if and only if $a_{u,v}$ is assigned to 1. Due to the construction of these variables and transitivity constraints represented with Equation (3.7), this order satisfies all requirements of the strict total order. Thus, it can sort the vertices, resulting in an outer k-planar drawing.

3.3 SAT-based algorithm

Another example of a thoroughly studied problem with an exponential-time solver is a Boolean Satisfiability problem (SAT). As an input, the problem gets a Boolean formula in a conjunctive normal form (CNF). For the output, it asks for such an assignment of logic values TRUE and FALSE to the variables used in the input, such that the expression evaluates to TRUE. This section discusses our process of encoding the recognition task into the SAT problem. To do so, we represent our problem as a Boolean expression, which is satisfied if and only if the graph admits an outer k-planar drawing. As an implementation of SAT solver, we used kissat [3, 21]. We chose it for our work as this solver is one of the best-performing ones [12].

To encode the drawing, we use the ordering variables $a_{u,v}$ we used in the ILP-based algorithm described in the Section 3.2 interpreting the value 1 as TRUE and 0 as FALSE. To encode the transitivity constraint, we follow the same logic, ending up with the same Equation (3.1). In terms of Boolean algebra, this can be encoded as follows:

$$a_{u,v} \wedge a_{v,w} \rightarrow a_{u,w}$$

To transform this into CNF, we expand the implication and apply De Morgan's law, getting the first set of clauses in the SAT representation:

$$\begin{aligned} a_{u,v} \wedge a_{v,w} \rightarrow a_{u,w} &\equiv \overline{(a_{u,v} \wedge a_{v,w})} \vee a_{u,w} \\ &\equiv \overline{a_{u,v}} \vee \overline{a_{v,w}} \vee a_{u,w} \end{aligned} \tag{3.19}$$

To encode the edges' intersections, we also reuse crossing variables $c_{uv,st}$ from the ILP-based algorithm. Similarly, we ensure that the variable is set to TRUE if the endpoints of the corresponding edges are arranged in one of eight crossing patterns (see Figure 3.2). This leads to eight sets of clauses, each of which contains ones representing one of these arrangements for all crossing variables. For example, for the arrangement from Figure 3.2b the constraint for the variable $c_{uv,st}$ can be encoded as follows:

$$a_{u,s} \wedge a_{s,v} \wedge a_{v,t} \rightarrow c_{uv,st}$$

To get a clause out of it, we expand the implication and apply De Morgan's law:

$$\begin{aligned} a_{u,s} \wedge a_{s,v} \wedge a_{v,t} \rightarrow c_{uv,st} &\equiv \overline{a_{u,s} \wedge a_{s,v} \wedge a_{v,t}} \vee c_{uv,st} \\ &\equiv \overline{a_{u,s}} \vee \overline{a_{s,v}} \vee \overline{a_{v,t}} \vee c_{uv,st} \end{aligned} \tag{3.20}$$

The last thing to encode is the limit of k intersections per edge. Unlike the ILP-based algorithm, we cannot add the corresponding variable. To impose this restriction, we ensure that no edges are crossed at least $k+1$ times. For that, for each edge e_0 and every set of $k+1$ edges $E = \{e_1, e_2, \ldots, e_{k+1}\}$ we add a following clause to the Boolean formula:

$$\overline{c_{e_0,e_1}} \vee \overline{c_{e_0,e_2}} \vee \cdots \vee \overline{c_{e_0,e_{k+1}}} \tag{3.21}$$

which evaluates to TRUE if and only if at least one variable is set to FALSE. By inserting this clause for every possible set E, we ensure that no $k+1$ edges cross the same edge. Thus, if the resulting Boolean expression is satisfiable by some realisation of the variables, the values of ordering variables from this realisation would indicate an outer k-planar drawing.

Lastly we combine clauses from Equation (3.19) for all triplets of vertices u, v and w, with clauses from Equation (3.20) for all pairs of edges uv and st, and with clauses from Equation (3.21) for all edges e_0 and all sets E of $k+1$ edges. We combine the clauses using logical and operator ($\wedge$), as every single clause must be satisfied to graph to be outer k-planar. Then, we run the SAT solver using the resulting Boolean expression as an input. If the solver fails to find a satisfiable instance, the algorithm halts, indicating that the graph is not outer k-planar. Otherwise, the algorithm halts returning the desired graph drawing. To reconstruct an outer k-planar drawing, similar to the ILP approach, we sort the vertices using the values of the ordering variables assigned by the solver to specify the order.

Unfortunately, unlike the algorithm based on ILP, this one does not solve the optimisation problem of finding the minimal possible k. The resulting algorithm solves the decision problem – whether the graph admits an outer k-planar drawing or not. To transform this into an optimisation, we incrementally check each integer, starting from 0, until the Boolean expression becomes satisfiable.

Finally, to estimate the complexity of the described formulation, we calculate the number of clauses in the Boolean expression. So by considering them in the order they are mentioned above, we get $O(n^3)+O(m^2)+O(m^{k+2}) = O(n^3+m^{k+2})$ clauses, where n is the number of vertices and m is number of edges in the graph. Unlike in the ILP-based algorithm, here, the size of the encoded problem grows exponentially in terms of k, which makes this encoding more complex.

3.4 Optimisations for ILP- and SAT-based algorithms

In both ILP- and SAT-based algorithms, for encoding the intersection of two edges uv and st, we used binary variable $c_{uv,st}$. However, despite 24 possible arrangements of edges' endpoints (see Figure 3.2), we imposed only 8 constraints in each algorithm. Each of these constraints covers one of the arrangements that result in the intersection, leaving the rest 16 for the solver to optimise. This section discusses two ways we considered helping the solver optimise these variables.

The first optimisation is a simple extension of the same logic we applied for the intersecting arrangements. To ensure the correct values of crossing variables, we can impose all 24 constraints on each one. The first eight we described in the corresponding Sections 3.2 and 3.3. The other 16 we construct in a similar way. For example, to construct a constraint for an arrangement in Figure 3.2a, we have to encode the following:

$$a_{u,v} = 1 \wedge a_{v,s} = 1 \wedge a_{s,t} = 1 \Longrightarrow c_{uv,st} = 0$$

To represent it as a linear constraint for an ILP-based algorithm, we have to limit $c_{uv,st}$ from above if all three variables are 1. We can do so with an equation $3 - a_{u,v} - a_{v,s} - a_{s,t}$, resulting in a constraint:

$$c_{uv,st} \leqslant 3 - a_{u,v} - a_{v,s} - a_{s,t}, \quad \forall uv, st \in E(G)$$

For a SAT-based algorithm, we can transform the implication in the similar way we did it for the clause from Equation (3.20):

$$\begin{aligned} a_{u,v} \land a_{v,s} \land a_{s,t} \to \overline{c_{uv,st}} &\equiv \overline{a_{u,v} \land a_{v,s} \land a_{s,t}} \lor \overline{c_{uv,st}} \\ &\equiv \overline{a_{u,v}} \lor \overline{a_{v,s}} \lor \overline{a_{s,t}} \lor \overline{c_{uv,st}} \end{aligned}$$

By repeating these constraints for all non-crossing arrangements, we get the algorithms with exact crossing variables.

Another optimisation that we considered can be implemented only for an ILP-based algorithm. The primary problem caused by this inaccuracy in ILP formulation is that the variables' influence on the objective function is not direct. The crossing variables constrain another variable k, which in turn affects the objective function, so it might be hard for an optimiser to estimate the effect of crossing variables accurately. With this optimisation, we help the solver by introducing a small direct influence on an objective function bypassing the intermediate variable k. To do so, we include an extra term in the objective function from Equation (3.6): $\frac{\sum c_{e_1,e_2}}{|E|^2}$. By using $|E|^2$ as a dominator in the fraction, we ensure that the value of the inserted term never exceeds 1 so that the optimiser would always prioritise decreasing k over this term. As a result, the optimised objective function for the ILP problem is:

$$k + \frac{\sum_{e_1,e_2 \in E(G)} c_{e_1,e_2}}{|E|^2}$$

3.5 DP algorithm

The last algorithm we considered was introduced by Kobayashi et al. [22]. Unlike previously discussed ones, this algorithm was explicitly designed to solve the recognition problem. This method uses the approach of dynamic programming, where the solution for a problem is built based on solutions of similar but smaller problems. Thus, the final drawing of the graph is built incrementally each time for a larger part of the original graph.

The whole process can be divided into steps. Each one of them can be parameterised by three parameters. The first is a pair of vertices u and v that split a graph into two parts, denoted as a link. The next is a set R_{uv} of vertices lying to the right of the link. And lastly the set $E_{uv} = \{e_1, e_2, \ldots, e_l\}$ of l edges crossing the uv link from the right to the left side. To represent each step separately, we introduce a graph $G_{uv,R_{uv}}$ which consists of vertices $\{u, v\} \cup R_{uv}$ alongside all connecting edges from an original graph G with inserted vertices $t_1, t_2, \ldots, t_l$ connected to corresponding vertices by edges $e_1, e_2, \ldots, e_l$. We call a configuration on each step *drawable* if exists an outer k-planar drawing of a corresponding graph $G_{uv,R_{uv}}$ which cyclic order contains $(u, t_{\tau(1)}, t_{\tau(2)}, \ldots, t_{\tau(l)}, v)$ as a consecutive subsequence for some permutation τ. On each step, the algorithm finds all possible permutations for which the configuration is drawable and stores them in the lookup table.

In the implementation of the algorithm, we first construct an index of all possible configurations. It allows us to simply iterate through it later without searching for the next configuration. Since, on each step, we try to draw a configuration using two smaller drawable ones, we group them by the size of the right part, guaranteeing that all smaller drawable configurations are already discovered at any point of the process. Unfortunately, it is unfeasible to consider all possible configurations due to the sheer number of them. For a graph with n vertices, there are $\frac{n(n-1)}{2}$ links with 2^{n-2} possible right sides each. As the link together with the right side uniquely determines the edges that cross the link, totally there are $n(n-1) \cdot 2^{n-3}$ possible configurations.

To significantly reduce the search space, the authors used the result of Firman et al. They showed [11] that there is a vertex w from the right side of any drawable configuration that splits it into two smaller *narrow* configurations for which $|E_{uw}| \leqslant k$ and $|E_{vw}| \leqslant k$. By reversing their argument, we get that $G_{uv,R_{uv}}$ admits an outer k-planar drawing if and only if we can combine it from two narrow drawable configurations. This allows us to limit the considered configurations only to narrow once, reducing their number to $2^{O(k)}m^{k+O(1)}$ [22, Lemma 15] instances.

Despite this optimisation, there is still a massive number of configurations. To minimise the memory consumption and make it feasible, we represent the right sides as binary masks stored as 64-bit integers. This decision limits the current implementation to graphs with at most 64 vertices. However, considering the complexity of the algorithm, we believe the graphs of bigger sizes would require an unreasonable amount of resources anyway[4].

To populate this index, we start by iterating over possible values for l[5]. For each choice of l and each link uv, we consider an augmented graph H obtained by removing u and v from the original graph G alongside all connected edges. Then, we select exactly l edges from H to cross the link uv. These edges further subdivide some connected components of H into connected subcomponents. Crucially, as these subcomponents do not contain edges that cross the link, each one of them must be located entirely on one side. Thus, finding all valid right sides for a given link means finding all valid black-white colourings of subcomponents, where white indicates belonging to the right and black to the left side. Consequently, each selected edge has to connect subcomponents of different colours, or in other words, the metagraph of H with subcomponents as vertices connected by selected edges has to be bipartite. After ensuring this holds, we construct all possible right sides for the selected link. As each connected bipartite graph can be coloured in exactly two ways, there are exactly 2^d possible right sides, where d is the number of connected components in H.

After constructing an index, we proceed to fill the lookup table. In there, for each configuration, we record all discovered sets of arrangements of E_{uv} that can appear in an outer k-planar drawing of $G_{uv,R_{uv}}$ grouped by the link uv and the right side R_{uv}. Each arrangement A_{uv} apart from a permutation τ of edges in E_{uv} also contains a map $f_{uv} : E_{uv} \to \mathbb{N}_+$ that matches each edge from E_{uv} with its number of intersections in the drawing of $G_{uv,R_{uv}}$.

To fill the corresponding cell of the lookup table, we have to find all arrangements for which a specific configuration is drawable. We start by selecting a split vertex w that belongs to the right side R_{uv}. For each w, we iterate over all configurations with a link uw and a right side R_{uw} that is a subset of R_{uv}. Additionally, we also consider a complementary configuration with a link vw and right side $R_{vw} = R_{uv} \setminus (R_{uw} \cup \{w\})$. For each such pair of configurations, we iterate over all pairs of drawable arrangements A_{uw} and A_{vw} saved in the lookup table and search for all possible ways to combine them into an outer k-planar drawing of $G_{uv,R_{uv}}$.

There is only one way to glue drawings of two configurations together. However, to form a valid drawing of $G_{uv,R_{uv}}$, we also have to decide on the order of edges crossing the link uv represented by a permutation τ_{uv}. To ensure the correctness of the solution, we go through all possible ones. For each permutation, we check whether

[4]Potentially it is possible to develop a specified bitmask object which could handle any number of vertices by using multiple integers stored in an array.

[5]By iterating over this first, we ensure that it is easy to extend the index for $k+1$ edges if the check for outer k-planarity is unsuccessful.

the resulting drawing is valid – each edge is crossed at most k times – and construct a mapping f_{uv} if so. To do that, we focus on an inner triangle consisting of three vertices u, v and w, and all the edges crossing at least one of the links uv, uw and vw. Additionally, we include edge (u, v) if such exists. Crucially, to calculate all the intersections apart from the edges, we also need the order in which they enter the triangle. As sides of the triangle are exactly the links, this order is represented in corresponding permutations: τ_{uw} from A_{uw}, τ_{vw} from A_{vw} and τ_{uv} which is considered one by one. To represent the order in the triangle, we insert l_{uv} helper vertices between u and v, given that $l_{uv} := |E_{uv}|$. We treat them as endpoints of corresponding edges that enter the triangle by crossing the link uv. Similarly, we insert vertices along the links uw and vw. Importantly, each inserted vertex is an endpoint only for one edge, and along each link, they are ordered according to the corresponding permutations. By considering these vertices as edges' endpoints, we limit the view to the intersections created by the combination of two parts, ignoring those in R_{uw} and R_{vw}. So, to get the crossing number for each edge, apart from the ones we count in the triangle, we also have to add those accounted by f_{uw} and f_{vw}. If the crossing number of any edge exceeds k, we discard the permutation τ_{uv} and proceed to the next one. Otherwise, we extract the mapping f_{uv} and, together with the current permutation τ_{uv}, add it as a new arrangement to the lookup table.

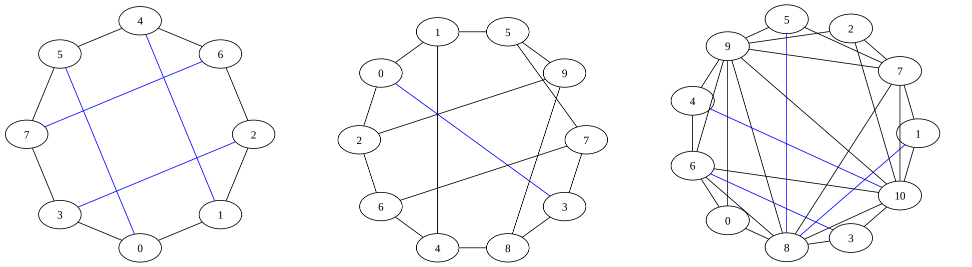
If at any moment, the algorithm finds a drawable configuration with the link uv and right side $R_{uv} = V(G) \setminus \{u, v\}$, it halts indicating that G is outer k-planar. In this case, the graph $G_{uv,R_{uv}}$ is equivalent to the original graph G and admits an outer k-planar drawing. If, on the other hand, we reach the end of the index and do not find such a configuration, the algorithm halts, indicating that the graph is not outer k-planar.

In case of success, apart from a positive answer, we also have to return the graph drawing. There are multiple ways to accomplish this. One is to use a backtracking algorithm after finding the configuration to rediscover all smaller configurations with which the last one was built. Another approach is to store information on how each configuration was constructed alongside each arrangement. The latter trades the memory consumption required for additional information in each arrangement for the time required to perform backtracking. We used the second option in our implementation as it is much easier. Moreover, the bottleneck of the current implementation is running time and not memory. As additional information, we opted to store the drawing of the right side. To get this, we combine the drawings of two parts on each step, inserting the split vertex w between them. As a result, to get the drawing of G having the drawable configuration $G_{uv,R_{uv}}$, it is enough to add vertex u at the start and vertex v at the end of the drawing stored alongside the arrangement A_{uv} in the lookup table.

Similar to the SAT-based algorithm from the Section 3.3, this method only tests whether the graph admits an outer k-planar drawing or not for a fixed k, so to find the minimal possible crossing number, we have to check each value incrementally.

3.6 Interface of the implementation

To have the ability to invoke the implemented algorithms for some specific graph, we designed a command-line interface. As an input, it receives a graph and a name of the algorithm the user wants to invoke. As an output, the user receives a local circular crossing number k of the passed graph alongside its circular drawing, where each edge is intersected at most k times. To communicate with a user, we use the Graphviz [14]



(a) Cubical graph ($k = 2$) (b) Peterson graph ($k = 4$) (c) Goldner-Harary Graph ($k = 6$)

Figure 3.3: An example of the resulting drawing for some well-know graphs.

DOT language for both the graph and its drawing. We use this representation as it is easily interpretable by humans and computers.

Speaking more precise, the command-line interface has 1 required parameter – an undirected graph, represented as a string in Graphviz DOT language – and 2 optional arguments: method specifying one of the three implemented algorithms and output file specifying a path to file in which the graph's drawing should be saved. By default, the former holds the value corresponding to the ILP-based algorithm, and the latter holds an empty string, indicating that the drawing should not be saved anywhere. After running the algorithm, the tool will display the local circular crossing number and/or potential error messages.

To transform the returned drawing into a picture, we use the `neato` layout engine [13]. In Figure 3.3 we demonstrate the drawings returned by the ILP-based algorithm for some well-known graphs, depicted by `neato` engine. In the returned drawings, we use blue to indicate the edges that cross exactly k other edges, where k is the local circular crossing number.

Chapter 4

Experiments and Results

This chapter describes our experiments that evaluate and compare the performance of the methods described above and reports the results. For conciseness, here we refer to the implementations of the ILP-, SAT- and DP-based algorithm as ILP, SAT, and DP respectively.

4.1 Data

To evaluate the implementation, we require a set of graphs which will be provided as input to the algorithm. To acquire them, we used the online database The House of Graphs [8] that contains interesting graphs. We decided to conduct experiments on these graphs, as they are most likely to be typical targets of the algorithms. In this work, we experimented with a small subset of all those graphs.

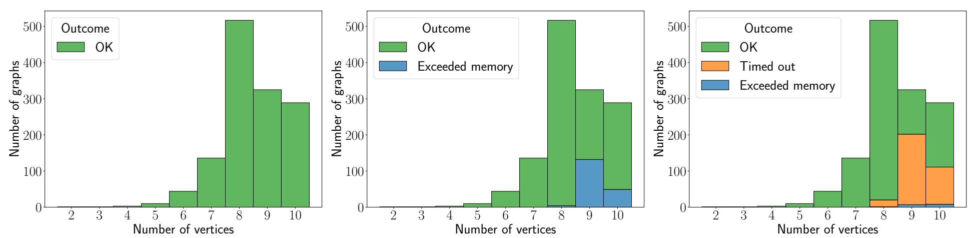
The problem of computing the local circular crossing number is NP-hard. Consequently, the resources required by all implemented algorithms grow exponentially with the input size. To be able to perform the experiments, we had to limit the size of the graphs. We have done this by picking only graphs with at most 10 vertices. This constraint leaves plenty of graphs to experiment with while significantly limiting the computational resources. Also, as we designed the implementation only for connected graphs, we filtered out unconnected graphs.

Why the plots whould be placed here? We obtained this data from the results of Section 4.3

This query resulted in 2007 different graphs. Among them 1326 are biconnected and 681 are not.

4.2 Experiment setup

All experiments described below were conducted on a virtual cloud server provided by Amazon Web Services. The hardware available was limited to a single core of an AWS Graviton4 Processor and 2GiB of random access memory. As an operating system, we used Linux.



(a) ILP (b) SAT (c) DP

Figure 4.1

For each experiment, we chose a set of graphs, a set of methods, and a set of configurations. We constructed all possible triplets of these parameters. For each one of them, we compute the local circular crossing number of the given graph using the specified method in a given configuration. Apart from recording the number itself, we also track the time required by the corresponding algorithm, which does not include the time required to start up and parse the graph, as it is part of any algorithm. Unfortunately, some methods require an unreasonable amount of time to calculate the result for some graphs. To mitigate this, we limit the execution time to 10 minutes for each triplet, indicating the outcome in the results.

We save the results of each experiment in CSV format. For each entry, we specify the Graphviz representation of the graph, the configuration, the method, and the execution results. The latter includes the local circular crossing number, the time required to compute it and whether the algorithm succeeded or not.

Apart from reporting, we also use the results obtained by all experiments to cross-validate the correctness of the implementations. Since no algorithm has been previously implemented for recognising outer k-planar graph, we do not have a reliable source to provide the correct local circular crossing numbers for test graphs. This cross-validation acts as a primary method for testing the implemented algorithm, with the only other one being manual verification of the returned drawing for small instances.

It is worth noting that the Gurobi optimiser supports multithreading. However, to make the comparison of the algorithms fair, we limited it to a single thread.

4.3 Biconnected decomposition

The first experiment we considered was comparing the performance of the algorithms with and without bicomponent decomposition. Here, we used only non-biconnected graphs from the dataset to show the difference between the two configurations. We ran this for all three methods. The results are presented in the Figure 4.2.

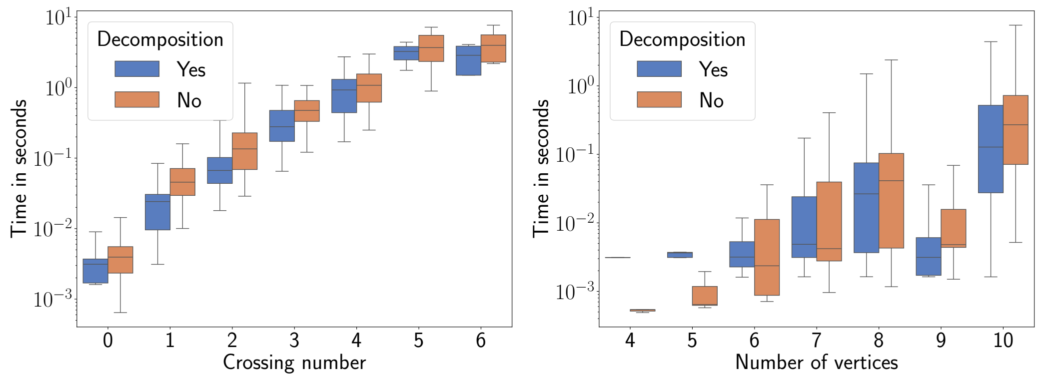
The results prove that using decomposition indeed boosts performance for almost all methods and graphs. The only exceptions are small graphs for ILP, for which the cost of initialising multiple environments outweighs the cost of decoupling the problem.

As we demonstrated in this experiment, the non-biconnectivity of the graphs artificially decreases the complexity of the recognition task, as each one of them requires multiple times fewer resources compared to equally sized biconnected graphs. Thus, in the following experiments, we consider only biconnected graphs to exclude this source of noise from the results.

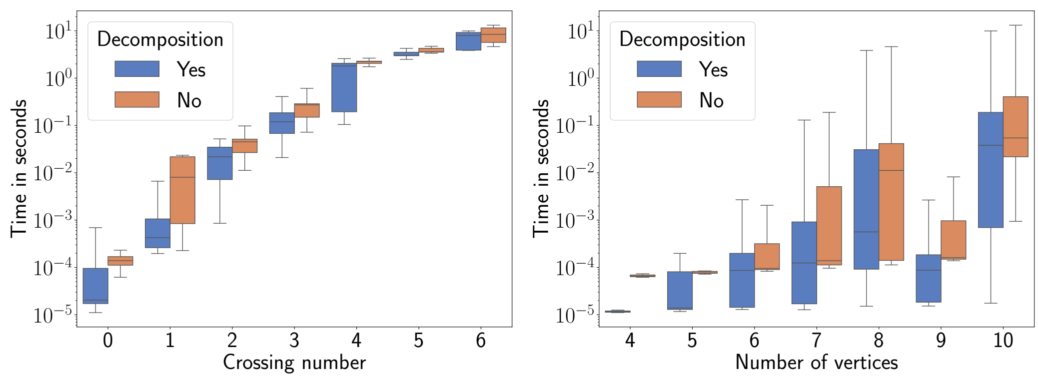
4.4 Comparison of the algorithms

In this experiment, we compared the performance of the algorithms. We grouped results by crossing numbers and the algorithm used for solving and displayed in Figure 4.3. Due to the complexity of the problem, some runs ran out of allocated resources. So, to display the results, we used only the measurements from runs that successfully found the minimal crossing number. As a result, starting from $k = 6$, boxes for SAT and DP depict fewer runs compared to ILP as they required more resources for some graphs than were available.

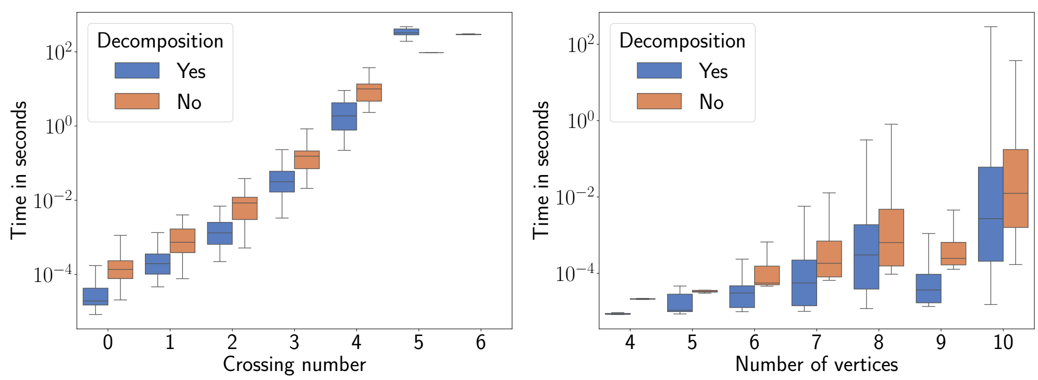
SAT and DP, however, were limited primarily by different factors. Unlike ILP, encoding in SAT contains an exponential number of clauses in terms of local circular



(a) ILP



(b) SAT



(c) DP

Figure 4.2: Results of the experiment, demonstrating the influence of biconnected decomposition on the running time of ILP , SAT and DP.

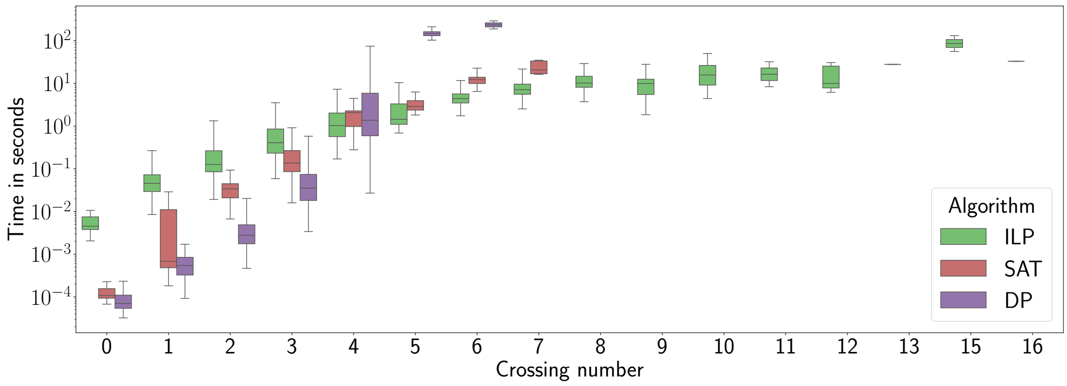


Figure 4.3: Comparison of running times of different algorithms for graphs with different crossing numbers.

crossing number. As a result, the solver ran out of allocated memory for 185 out of 1326 biconnected graphs. Similarly, DP ran out of memory 16 times. However, the bigger limiting factor for this algorithm is time, as 317 runs exceeded the 10-minute time limit imposed on each one. All other runs finished successfully.

The results show that the time required by both SAT and DP grows much faster than the time required by ILP. The latter requires each instance to set up an environment for the graphs. With smaller crossing numbers, these costs outweigh the solver's speed. However, for instances with bigger k, the time required for setup is negligible.

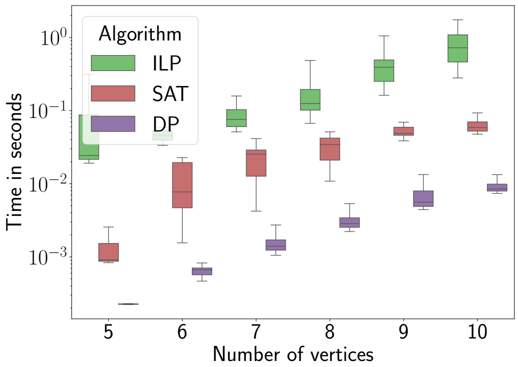
As the algorithm's running time depends not only on crossing numbers but also on the size of the graph, we can represent the results more accurately by grouping them by both the crossing number and the number of vertices. To demonstrate this dependency, in Figure 4.4, we created a plot for each crossing number represented by at least 150 graphs. These plots show that, for $k \in \{2, 3\}$, DP and SAT are consistently the two fastest methods, with the former being faster than the latter. However, starting from $k = 4$ and graphs with at least 9 vertices, ILP outperforms the other two methods. Most importantly, these results agree with the ones displayed in Figure 4.3, which aggregate them for each value k.

4.5 Optimisation benchmark

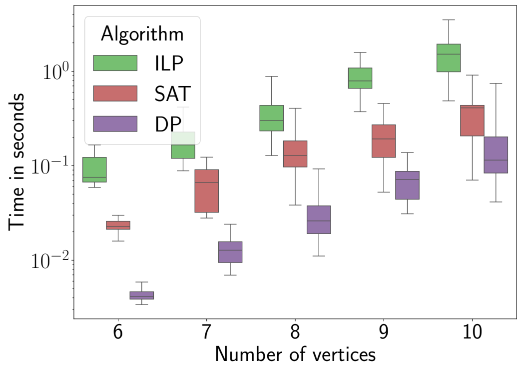
The last experiment we considered is the comparison of optimisations we discussed for ILP and SAT in Section 3.4. For ILP, we ran four configurations for each biconnected graph using none, one, or both optimisations. For SAT, we used two configurations with and without optimisation. The results are presented in Figure 4.5 and Figure 4.6 respectively.

In the first plot, we can clearly see that adding additional constraints to enforce the exact value for each crossing variable degrades the performance. For SAT, however, this change does not influence this much. We can see a slight rise in execution time, but the difference is within the margin of error. Here, the difference may be caused by the algorithm writing down the required additional constraints, not the SAT solver.

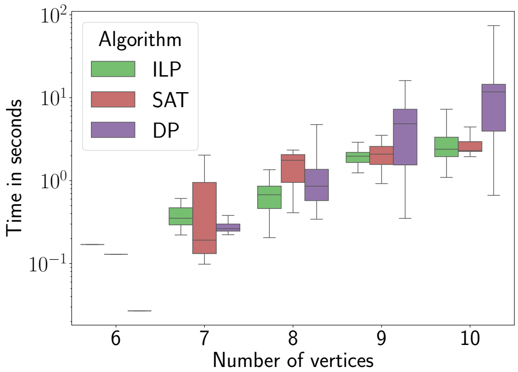
The objective optimisation for ILP, which includes an extra term in the objective function, makes small but still an improvement. As a result, we consider the algorithm that includes this optimisation to be the most efficient ILP. Thus, we have used this configuration for all other experiments. For SAT, we used a configuration that did not include optimisation.



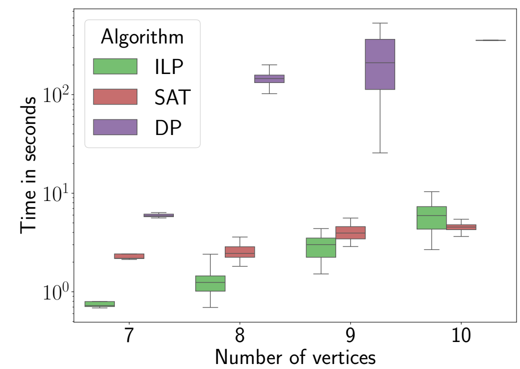
(a) $k = 2$



(b) $k = 3$



(c) $k = 4$



(d) $k = 5$

Figure 4.4: Comparison of running times required by algorithms to recognise outer k-planar graphs for various values of k.

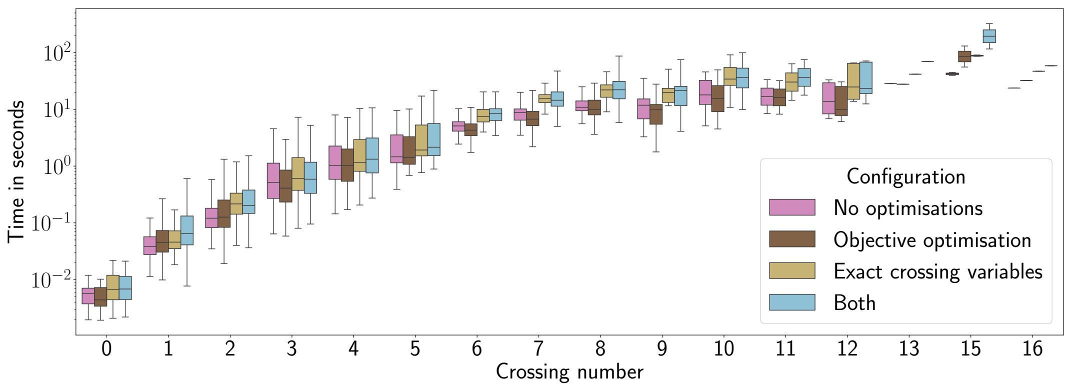


Figure 4.5: Comparison of different configurations for ILP.

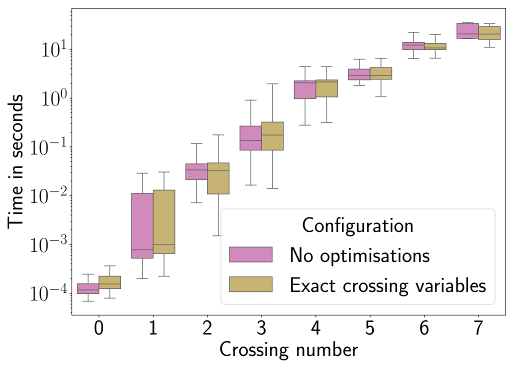


Figure 4.6: Comparison of different configurations for SAT.

Chapter 5

Conclusions

In this bachelor thesis, we have provided implementations of three algorithms for recognising outer k-planar graphs. Two of them are based on integer linear programming and satisfiability formulation and have been introduced in this thesis. The last one uses a dynamic programming approach, recently presented by Kobayashi et al. [22]. Additionally, we provide a command line tool to invoke the desired method for a specific graph G. The tool returns the local circular crossing number k of G together with a circular drawing of G where each edge is intersected at most k times.

We have also demonstrated an example of the program's output for a sample graph and the results of the experiments designed to test, evaluate the performance and compare the algorithms with each other. After having analysed the results of the experiments, we conclude as follows:

- Despite the overhead required to perform the biconnected decomposition, doing so significantly improves the performance of all methods.
- In the ILP-based algorithm, including constraints for all arrangements of two edges' endpoints significantly worsens its performance. On the other hand, including an extra symetry-breaking term in the objective function slightly improves the performance.
- In the SAT-based algorithm, including clauses for all arrangements does not influence the performance.
- The computational resources required for executing the ILP-based algorithm grow more slowly compared to both the SAT- and DP-based algorithms with increasing local circular crossing number.
- The running times of the SAT- and DP-based algorithm grow exponentially in terms of the local circular crossing number of the graph.

5.1 Limitations

The first and most major limitation of our implementations is the complexity of the underlying algorithm. The exponential dependency of the running time on local circular crossing number of the input graph significantly limits the number of graphs for which using these implementations is practically reasonable.

The current version also limits the number of vertices to 64 in the input graph for the DP-based algorithm, as it uses a bitmask for storing subsets of vertices. We implemented this using an integer as a bitmask to lower the memory consumption; hence, this limit may be different for other systems.

5.2 Future work

There are many directions to explore as future work. First of all, we could combine our implementations into a library. By doing so, the algorithms could more easily be called by other programs.

Secondly, we could optimise the underlying algorithms discussed in this work. In particular, it would be desirable to simplify the algorithm for very small values of k. Can outer 2-planar graphs be recognised in, say, quadratic time?

Another direction of improvement might be developing an algorithm that, for the given embedding of an outer k-planar graph, draws it using Bézier curves for edges instead of straight lines. Using this approach, we can improve the readability of the drawings.

Finally, we can explore the options for further optimisations of the implemented algorithms. One of the highly promising directions to do so would be to make the implementation of the DP-based algorithm multithreaded. By ensuring that all threads check configurations with the right sides of the same size, we eliminate the requirement of memory synchronisation. This makes this algorithm embarrassingly parallel, as it requires synchronising all threads less than $|V(G)|$ times throughout the whole execution.

Bibliography

[1] Christopher Auer, Christian Bachmaier, Franz J. Brandenburg, Andreas Gleißner, Kathrin Hanauer, Daniel Neuwirth, and Josef Reislhuber. "Outer 1-Planar Graphs". In: *Algorithmica* 74.4 (2016), pp. 1293–1320. DOI: 10.1007/s00453-015-0002-1.

[2] Carlo Batini. "A layout algorithm for data flow diagrams". In: *IEEE Transactions on Software Engineering* 12.1 (1986), pp. 538–546. DOI: 10.1109/TSE.1986.6312901.

[3] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximilian Heisinger. "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020". In: *Proceedings of SAT Competition 2020 – Solver and Benchmark Descriptions*. Ed. by Tomáš Balyo, Nils Froleyks, Marijn J. H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Vol. B-2020-1. Department of Computer Science Report Series B. University of Helsinki, 2020, pp. 50–53. URL: https://tuhat.helsinki.fi/ws/files/142452772/sc2020\%5Fproceedings.pdf.

[4] Till Bruckdorfer, Sabine Cornelsen, Carsten Gutwenger, Michael Kaufmann, Fabrizio Montecchiani, Martin Nöllenburg, and Alexander Wolff. "Progress on Partial Edge Drawings". In: *Journal of Graph Algorithms and Applications* 21.4 (2017), pp. 757–786. DOI: 10.7155/jgaa.00438.

[5] Till Bruckdorfer and Michael Kaufmann. "Mad at Edge Crossings? Break the Edges!" In: *6th International Conference Fun with Algorithms (FUN)*. Ed. by Evangelos Kranakis, Danny Krizanc, and Flaminia Luccio. Vol. 7288. LNCS. Springer, 2012, pp. 40–50.

[6] Sergio Cabello and Bojan Mohar. "Adding One Edge to Planar Graphs Makes Crossing Number and 1-Planarity Hard". In: *SIAM Journal on Computing* 42.5 (2013), pp. 1803–1829. DOI: 10.1137/120872310.

[7] Steven Chaplick, Myroslav Kryven, Giuseppe Liotta, Andre Löffler, and Alexander Wolff. "Beyond Outerplanarity". In: *25th International Symposium on Graph Drawing and Network Visualization (GD)*. Ed. by Fabrizio Frati and Kwan-Liu Ma. Vol. 10692. LNCS. Springer, 2018, pp. 546–559. DOI: 10.1007/978-3-319-73915-1_42.

[8] Kris Coolsaet, Sven D'hondt, and Jan Goedgebeur. "House of Graphs 2.0: A Database of Interesting Graphs and More". In: *Discrete Applied Mathematics* 325.C (2023), pp. 97–107. DOI: 10.1016/j.dam.2022.10.013. URL: https://houseofgraphs.org.

[9] Matthew Dickerson, David Eppstein, Michael Goodrich, and Jeremy Meng. "Confluent Drawings: Visualizing Non-planar Diagrams in a Planar Way". In: *Journal of Graph Algorithms and Applications* 9.1 (2005), pp. 31–52. DOI: 10.7155/jgaa.00099.

[10] Walter Didimo, Giuseppe Liotta, and Fabrizio Montecchiani. "A Survey on Graph Drawing Beyond Planarity". In: *ACM Computing Surveys* 52.1 (2019), pp. 1–37. DOI: 10.1145/3301281.

[11] Oksana Firman, Grzegorz Gutowski, Myroslav Kryven, Yuto Okada, and Alexander Wolff. "Bounding the Treewidth of Outer k-Planar Graphs via Triangulations". In: *32nd International Symposium on Graph Drawing and Network Visualization (GD)*. Ed. by Stefan Felsner and Karsten Klein. Vol. 320. LIPIcs. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, pp. 14:1–14:17. DOI: 10.4230/LIPIcs.GD.2024.14.

[12] Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. "SAT Competition 2020". In: *Artificial Intelligence* 301.103572 (2021). DOI: 10.1016/j.artint.2021.103572.

[13] Emden R. Gansner, Yehuda Koren, and Stephen North. "Graph drawing by stress majorization". In: *12th International Conference on Graph Drawing and Network Visualization (GD)*. GD'04. Springer-Verlag, 2004, pp. 239–250. DOI: 10.1007/978-3-540-31843-9_25.

[14] Emden R. Gansner and Stephen C. North. "An open graph visualization system and its applications to software engineering". In: *Software: Practice and Experience* 30.11 (2000), pp. 1203–1233.

[15] Michael R. Garey and David S. Johnson. "Crossing Number is NP-Complete". In: *SIAM Journal on Algebraic Discrete Methods* 4.3 (1983), pp. 312–316. DOI: 10.1137/0604033.

[16] *Graph Drawing Contest 2025: Live Challenge. Minimizing the k in k-Plane Drawings*. Website. Accessed 27-04-2025. URL: https://mozart.diei.unipg.it/gdcontest/2025/live/.

[17] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. Website. Accessed 20-04-2025. 2024. URL: https://www.gurobi.com.

[18] Seok-Hee Hong, Peter Eades, Naoki Katoh, Giuseppe Liotta, Pascal Schweitzer, and Yusuke Suzuki. "A Linear-Time Algorithm for Testing Outer-1-Planarity". In: *Algorithmica* 72.4 (2015), pp. 1033–1054. DOI: 10.1007/s00453-014-9890-8.

[19] Seok-Hee Hong and Hiroshi Nagamochi. "A linear-time algorithm for testing full outer-2-planarity". In: *Discrete Applied Mathematics* 255.C (2019), pp. 234–257. DOI: 10.1016/j.dam.2018.08.018.

[20] John Hopcroft and Robert Tarjan. "Efficient Planarity Testing". In: *Journal of the ACM* 21.4 (1974), pp. 549–568. DOI: 10.1145/321850.321852.

[21] *Kissat SAT Solver*. Website. Accessed 25-04-2025. URL: https://fmv.jku.at/kissat/.

[22] Yasuaki Kobayashi, Yuto Okada, and Alexander Wolff. "Recognizing 2-Layer and Outer k-Planar Graphs". In: *41st Annual Symposium on Computational Geometry (SoCG)*. Ed. by Oswin Aichholzer and Haitao Wang. Vol. 332. LIPIcs. To appear. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, pp. 63:1–63:17. DOI: 10.48550/arXiv.2412.04042.

[23] Vladimir P. Korzhik and Bojan Mohar. "Minimal Obstructions for 1-Immersions and Hardness of 1-Planarity Testing". In: *Journal of Graph Theory* 72.1 (2013), pp. 30–71. DOI: 10.1002/jgt.21630.

[24] Arno Luppold, Dominic Oehlert, and Heiko Falk. *Evaluating the Performance of Solvers for Integer-Linear Programming*. Technical Report. Hamburg University of Technology, 2018. DOI: 10.15480/882.1839.

[25] Helen Purchase. "Which aesthetic has the greatest effect on human understanding?" In: *5th International Symposium on Graph Drawing and Network Visualization (GD)*. Ed. by Giuseppe Di Battista. Vol. 1353. LNCS. Springer, 1997, pp. 248–261. DOI: 10.1007/3-540-63938-1_67.

[26] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., 2002. URL: https://www.boost.org/doc/libs/1\%5F88\%5F0/libs/graph/doc/index.html.

[27] Manfred Wiegers. "Recognizing outerplanar graphs in linear time". In: *13th International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*. Ed. by Gottfried Tinhofer and Gunther Schmidt. Vol. 246. LNCS. Springer, 1987, pp. 165–176. DOI: 10.1007/3-540-17218-1_57.