UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

Recognition of Outer k-planar Graphs

Author:
Ivan SHEVCHENKO

Supervisor:
Prof. Dr. Alexander WOLFF
Yuto OKADA

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

Recognition of Outer k-planar Graphs

by Ivan SHEVCHENKO

Abstract

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too…

Contents

Abstract **i**

1 Introduction **1**
- 1.1 Contributions . . . 1
- 1.2 Structure of the thesis . . . 1

2 Related Work **2**
- 2.1 Difficulty of dealing with beyond-planar graphs . . . 2
- 2.2 Efficient recognition of some outer k-planar graph . . . 3
- 2.3 Recognizing general outer k-planar graphs . . . 3
- 2.4 Our contribution . . . 4

3 Proposed Solution **5**
- 3.1 Bicomponent decomposition . . . 5
- 3.2 ILP-based algorithm . . . 6
- 3.3 SAT Formulation (unchanged) . . . 9
- 3.4 Dynamic algorithm . . . 10

4 Experiments and Results **13**

5 Conclusions **14**

Bibliography **15**

Chapter 1

Introduction

1.1 Contributions

1.2 Structure of the thesis

Chapter 2

Related Work

Already in the 1980s, researchers in the field of graph drawing acknowledged the importance of reducing the edge crossings for improving visualisation clarity [2]. The suspicion that a drawing with fewer edge crossings was easier to comprehend was later confirmed by several experimental studies [14]. 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.

The ideal form of crossing minimising drawings – planar ones, has been focused on by the research community for a long time, with the first of linear-time algorithms for recognising planar graphs presented already in 1974 [11]. 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. This has led to a growing interest in exploring graphs close to being planar; see the survey by Didimo et al. [5]. Such graphs allow a limited number of crossings while still retaining some of the beneficial structural properties of 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 [7] demonstrated that the Crossing Number problem determining 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 [13] showed that testing the 1-planarity, recognising whether a graph can be drawn with at most one crossing per edge, is NP-hard. Later, Cabello and Mohar [3] showed that testing 1-planarity is still NP-hard even for near-planar graphs, that is, graphs that can be obtained from planar graphs by adding a single edge.

Given the complexity of recognising k-planarity, researchers considered exploring more restrictive classes of graphs, hoping that imposed limitations could simplify the recognition. One of the considered classes is the class of outer k-planar graphs, a subclass of k-planar graphs that admit drawings where all vertices lie on the outer face.

2.2 Efficient recognition of some outer k-planar graph

Although the general recognition problem for outer k-planar graphs is NP-hard, efficient algorithms have been developed for specific values of k.

For $k = 0$, the recognition task simplifies to an outerplanarity test. Regocnition 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 in [16], introduces the concept of 2-reducible graphs, which are totally disconnected or can be made totally disconnected by repeated deletion of edges adjacent to a vertex with a 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, 9]. 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 [10] 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. [4] extended their result by introducing an algorithm for recognising full outer k-planar graphs for every k. Their algorithm runs in $O(f(k) \cdot n)$ time, where f is a computable function.

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

2.3 Recognizing general outer k-planar graphs

For a given outer k-planar drawing of a graph, Firman et al. [6] 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 graph 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. [12] extended 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 a 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 them, the method checks whether these splits are compatible with each other and 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.

2.4 Our contribution

A common drawback of the methods described above is the lack of practical validation. Although these algorithms have been analysed and discussed in a theoretical context, they have not been implemented or empirically tested. We address this gap by implementing the most recent recognition algorithm and introducing two alternative approaches based on Integer Linear Programming (ILP) and Satisfiability (SAT) formulations. While it is NP-hard to solve the general integer linear program or to find a satisfying truth assigment for general Boolean formulas, there are very advanced solvers for such formulations that could allow us to find exact solutions for small- to medium-sized instances within an acceptable amount of time. Then we evaluate the performance and efficiency of these methods, demonstrating their practical applicability and limitations.

Chapter 3

Proposed Solution

This chapter will discuss the methods this work considers to recognise the outer k-planar graphs. Besides recognition, these methods provide the outer k-planar drawings of the given graphs if possible.

We represent the drawing as a sequence of vertices in the circular order in which they appear on the boundary of the outer face. Considering only straight-line drawings, this order uniquely determines all the edge crossings and, thus, also the number of crossings per edge. Indeed, consider two edges, uv and st. Without loss of generality, we can assume that u is located before v in the arrangement, s before t, and u before s. Under these assumptions, the edges uv and st cross if and only if s is located before v and t after v.

3.1 Bicomponent decomposition

For complex problems, decomposing 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 them independently by the recognition algorithm. One of the plausible ways to accomplish this is to split the graph into biconnected components as shown in the figure 3.1a. This process is implemented using block-cut decomposition. It is worth noting that each graph edge 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 vertex, see figure 3.1b.

TODO

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

TODO

(B) Block-cut tree of a graph

TODO

(C) Outer k-planar drawing of each component

TODO

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

FIGURE 3.1: An example of bicomponent decomposition

How to show that this boosts performance more formally in the next sentence?

Due to the nature of bi-connectedness, after getting outer k-planar drawings of biconnected components separately, we can combine them easily into an outer k-planar drawing of the whole graph, hence, the increase in performance. To be more specific, if some component does not admit an outer k-planar drawing, neither does the whole graph. Otherwise, if all components admit such a drawing, they can be merged by combining duplicates of each cut vertex see figures 3.1c and 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 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[1] from the Boost Graph Library [15]. 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 other sections of this chapter 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 as described before. 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 that has already been considered and thus added to a final drawing. As a result, we can find a corresponding cut vertex in both global and local drawings. Since each drawing is represented as a cyclic sequence of vertices, we can rotate the local drawing so that the corresponding cut vertex appears as the first item 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

As the problem of recognising the outer k-planar graphs is NP-hard, it can be reduced to another NP-hard problem. Some of them have already been studied for decades. Thus, extremely optimised algorithms for their solving have emerged during this period. One of them is an Integer Linear Programming problem (ILP). This problem asks to find a realisation of a variable vector $\mathbf{x}$ that optimises the objective represented as a linear combination of variables $\mathbf{c}^T\mathbf{x}$ 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

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

k-planar graphs. In this section, we discuss the reduction of our problem to the ILP. To implement the described reduction, we used Gurobi Optimizer [8] under the free academic licence.

To reduce 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 a sequence using the so-called "ordering variables", which defines a relative order of two vertices in a sequence. Specifically, for every pair of vertices u and v, we create a binary variable $a_{u,v}$ introducing a constraint (3.18). The value 1 indicates that vertex u is located before vertex v and value 0 indicates that either v is located before u or u and v is 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 v, and every other vertex w, if $a_{u,w} \equiv 1$ and $a_{w,v} \equiv 1$, meaning u is located before w and w is located before v, then u must be located before v, so the following should hold $a_{u,v} \equiv 1$. Including also the implication for the value 0, we get:

$$a_{u,w} \equiv 1 \wedge a_{w,v} \equiv 1 \longrightarrow a_{u,v} \equiv 1 \tag{3.1}$$

$$a_{u,w} \equiv 0 \wedge a_{w,v} \equiv 0 \longrightarrow a_{u,v} \equiv 0 \tag{3.2}$$

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

$$a_{v,w} \equiv 1 \wedge a_{w,u} \equiv 1 \longrightarrow a_{v,u} \equiv 1 \tag{3.3}$$

$$a_{v,w} \equiv 0 \wedge a_{w,u} \equiv 0 \longrightarrow a_{v,u} \equiv 0 \tag{3.4}$$

Note that for any distinct vertices x and y the following $a_{x,y} \equiv 1 - a_{y,x}$ 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 $a_{u,v}$ to a value 1 it is enough to impose a constraint $a_{u,v} \geqslant \epsilon$ for any $\epsilon \in (0;1]$. In a constraint for ILP, this ϵmust be represented as a linear function of $a_{u,w}$ and $a_{w,v}$, whose value lies in the half-interval $(0;1]$ if and only if both binary variables are 1. An example of such an equation is $a_{u,v} + a_{v,w} - 1$, leading to a constraint (3.7) in the ILP formulation.

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 variable $c_{uv,st}$, hence the constraint (3.17). The endpoints of the edges can be arranged in 24 different ways, as demonstrated in the figure 3.2. Among them, there are only eight in which the edges intersect, so the variable $c_{uv,st}$ must be equal to 1 if the corresponding "ordering variables" indicate one of these eight arrangements[2]. 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 $usvt$. If the vertices are arranged so, each equation out of $a_{u,s} \equiv 1$, $a_{s,v} \equiv 1$ and $a_{v,t} \equiv 1$ must hold. Moreover, if these equations hold, the vertices must be arranged as $usvt$. Thus, we have to encode the following equation:

$$a_{u,s} \equiv 1 \wedge a_{s,v} \equiv 1 \wedge a_{v,t} \equiv 1 \longrightarrow c_{uv,st} \equiv 1$$

[2] 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 it to the optimiser. Doing so, we simplify the problem by reducing the number of constraints for every pair of edges from 24 to 8.

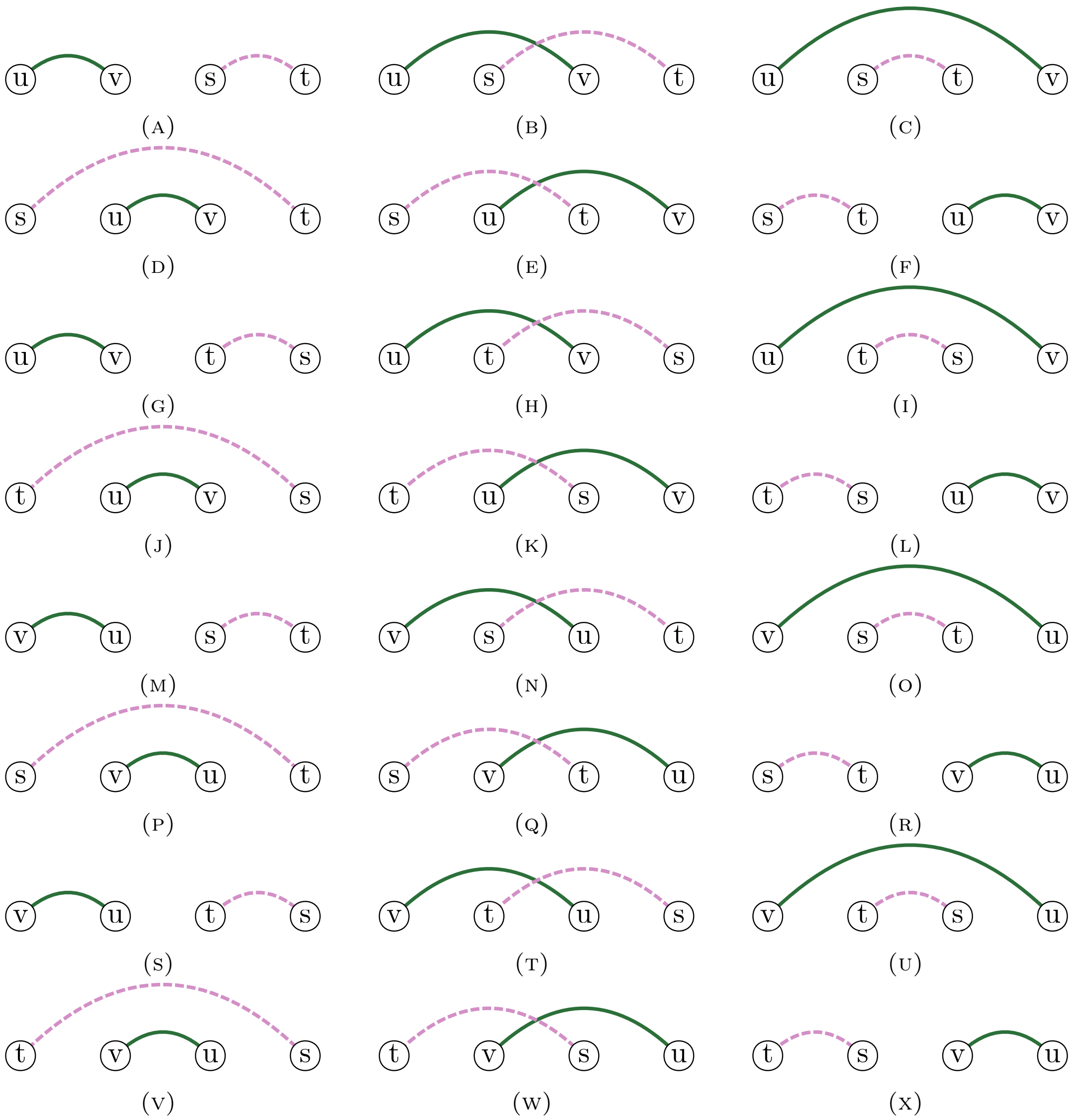

FIGURE 3.2: All 24 possible arrangements of two edges' endpoints, only 8 of which result in intersection. *I do not think it is really necessary. It is TOO big*

We can do so in the same way as we encoded the equation (3.1), getting the constraint (3.8). The constraints (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. The crossing number of each edge can be easily represented using "crossing variables" by summing the corresponding ones:

$$cr_{e_1} \leqslant \sum_{e_2 \in E(G)} c_{e1,e2} \tag{3.5}$$

However, as taking the maximal value is not a linear function, to represent the objective, we have to introduce a new variable k, which represents the crossing number of the whole graph G. To ensure that we have to bound k from below by crossing number if each edge: $k \geqslant cr_{e_1} \forall e_1 \in E(G)$. Combining this with equation (3.5), we get the constraint (3.16). As a result, minimising for k would give the desired result.

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

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

In addition to the described reduction, we also considered a small optimisation of an objective function for the ILP formulation. In the implementation we discussed, we only ensure that each "crossing variable" equals 1 if two edges actually cross, so for any non-crossing edges, the variable might take on both 0 and 1. As the variables' influence on the objective is not direct, but through a constraint on variable k, it might be hard for an optimiser to estimate the influence of each variable accurately. To help it, we included an extra term in the objective function (3.6): $\frac{\sum c_{e_1,e_2}}{|E|^2}$. By using $|E|^2$ as a dominator of 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.

3.3 SAT Formulation (unchanged)

Another approach to solving this problem is to check for a specific k whether the given graph is outer-k-planar. This check can be encoded as a boolean satisfiability problem. This problem asks whether it is possible to assign logic values TRUE or FALSE so that all disjunctive clauses are satisfied. A disjunctive clause is a single literal or a disjunction of several. Literal is either a variable or a negation of a variable, with the former being the positive and the latter the negative literal.

Similarly to the ILP algorithm described in 3.2, this algorithm uses the same "ordering variables" $a_{u,v}$ for each pair of vertices u and v that represent the arrangement of the vertices. If the boolean variable $a_{u,v}$ is TRUE, the vertex u is located before the vertex v and vice versa otherwise.

Similarly, these variables must account for transitivity, which means that for every triple of vertices u, v, and w $a_{u,v} \equiv \text{TRUE}$ and $a_{v,w} \equiv \text{TRUE}$ implies $a_{u,w} \equiv \text{TRUE}$. This can be written as follows: $a_{u,v} \wedge a_{v,w} \rightarrow a_{u,w}$. Expanding the implication, this transforms into $\overline{a_{u,v} \wedge a_{v,w}} \vee a_{u,w}$. After applying De Morgan's law, we receive $\overline{a_{u,v}} \vee \overline{a_{v,w}} \vee a_{u,w}$, which represents a clause in the SAT problem.

The next step is to represent the crossing variables $c_{uv,st}$ in terms of the ordering ones for each pair of edges uv and st. Similarly to the ILP algorithm, we can restrict $c_{uv,st}$ to TRUE if uv and st cross by adding new clauses to the problem. The clauses are constructed by making the implications for each of the eight intersecting cases shown in figure 3.2, expanding them, and applying De Morgan's law. For example, for the case $a_{u,s} = 1$, $a_{s,v} = 1$, and $a_{v,t} = 1$, we start with the logical equation as follows: $a_{u,s} \wedge a_{s,v} \wedge a_{v,t} \rightarrow c_{uv,st}$. Afterwards, we expand the implication: $\overline{a_{u,s} \wedge a_{s,v} \wedge a_{v,t}} \vee c_{uv,st}$. Finally, we apply De Morgan's law: $\overline{a_{u,s}} \vee \overline{a_{s,v}} \vee \overline{a_{v,t}} \vee c_{uv,st}$ receiving one of the eight clauses for $c_{uv,st}$.

The last step in the construction of the problem is to count the number of crossings for each edge. The goal of this solver is to check whether the number of crossings can be smaller or equal to some constant k for each edge. To ensure this, we can build a set of clauses that prevent the problem from being satisfiable if the value k is too small. To do so, for every edge e_0, we consider all combinations of $e_1, e_2, \ldots, e_{k+1}$ for each of which we construct the following clause: $\overline{c_{e_0,e_1}} \vee \overline{c_{e_0,e_2}} \vee \cdots \vee \overline{c_{e_0,e_{k+1}}}$. Doing so, we ensure that for each edge e_0, no $k+1$ different edges intersect e_0, which effectively means that each edge has at most k crossings if all clauses are satisfied.

implementation details (iterate over all permutations, finding minimal k)

possible optimizations (2k + 2, equivalence instead of implication for c_{...}, eliminating c_{...} at all)

3.4 Dynamic algorithm

The last algorithm we considered was introduced by Kobayashi et al. [12]. Its core idea is to build the drawing incrementally. Each step of this process can be parameterised by three parameters. The first is a pair of vertices u and v that split a potential drawing into two parts, denoted as a link. The next is a set R_{uv} of vertices lying to the right of the split. 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. Let $G_{uv,R_{uv}}$ be a graph consisting 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.

To significantly reduce the search space, the authors used the result of Firman et al. In their work [6], authors showed that the right side of any *drawable* configuration can be split into two smaller parts by selecting a vertex w from the right side in such a way that both links uw and vw are crossed by at most k edges. This allows us to limit the search only to *narrow* configurations with $l \leqslant k$ as by reversing the argument, we get that outer k-planar drawing for $G_{uv,R_{uv}}$ is possible if and only if we can combine it from two *narrow drawable* configurations for active links uw and vw for some vertex w from the right side.

In the implementation of the algorithm, we first construct an index of all possible *narrow* configurations. Since, on each step, we try to draw a configuration using two smaller *drawable* ones, we grouped them by the size of the right part, guaranteeing that all smaller *drawable* configurations are already discovered at any point. Since the amount of crossing edges is bounded by k, the number of entries in this index can be bounded by the following amount: $2^{O(k)}m^{k+O(1)}$ [12, Lemma 15].

To populate this index, we start by iterating over possible values for l[3]. 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 each connected component 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 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 graph H. As a result, there might be a massive number of entries in the index and, consequently, the lookup table itself. 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 this size would require an unreasonable amount of resources anyway[4].

In the lookup table itself, 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}[5] consist of a permutation τ of edges in E_{uv} and 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 $G_{uv,R_{uv}}$.

Next, iterating over the index, we fill the corresponding cells of the lookup table. For that, 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 for a link uw with a right side R_{uw} being a subset of R_{uv}. At the same time, 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

[3] 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.

[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] I think I should use another letter

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 the resulting drawing is valid – each edge is crossed at most k times. To do so, 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 are looked through one by one. To represent this in the triangle, we insert l_{uv} helper vertices between u and v, given that l_{uv} is a size of E_{uv}, and treat them as endpoints of edges that cross the link uv. Similarly, we insert vertices corresponding to edges crossing uw and vw. Importantly, each inserted vertex is an endpoint only for one edge and is 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 add a new arrangement to the lookup table.

If at any moment, the algorithm finds a *drawable* configuration for vertices u and v with $R_{uv} = V(G) \setminus \{u, v\}$, it halts indicating that G is outer k-planar, as in such a case $G_{uv,R_{uv}}$ is an original graph G and admits an outer k-planar drawing. If, on the other hand, the lookup table is filled, and no such configuration is found, the algorithm halts, indicating that the graph is not outer k-planar. Similar to the SAT-based algorithm from the section 3.3, this method only tests whether the graph admits outer k-planar drawing or not for a fixed k, so to find the minimal possible crossing number we have to check the values incrementally.

Chapter 4

Experiments and Results

Chapter 5

Conclusions

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] 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.

[4] 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.

[5] 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.

[6] 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. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 14:1–14:17. DOI: 10.4230/LIPIcs.GD.2024.14.

[7] 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.

[8] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. 2024. URL: https://www.gurobi.com.

[9] 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.

[10] 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.

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

[12] 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. Leibniz International Proceedings in Informatics (LIPIcs). To appear. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. DOI: 10.48550/arXiv.2412.04042.

[13] 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.

[14] 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.

[15] *The boost graph library: user guide and reference manual*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[16] Manfred Wiegers. "Recognizing outerplanar graphs in linear time". In: *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.