Karlsruhe Institute of Technology

Pattern-Avoiding Vertex Orders with SAT

Master's Thesis of

Jonathan Benedikt Dransfeld

At the Department of Informatics
Institute of Theoretical Informatics (ITI)

Reviewer:	T.T.-Prof. Dr. Thomas Bläsius
Second reviewer:	Dr. Dominik Schreiber
Advisors:	T.T.-Prof. Dr. Thomas Bläsius
	Dr. Dominik Schreiber

9th January 2026 – 9th July 2026

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. I have followed the by-laws to implement scientific integrity at KIT.

Karlsruhe, 9th July 2026

. . . . . .

(Jonathan Benedikt Dransfeld)

Abstract

Modern SAT solvers are highly efficient and still very actively researched. Despite their efficiency, ordering problems remain difficult for SAT solvers. Recently, the SAT solver CaDiCaL has introduced user propagators, allowing for more and easier interaction with the SAT solver during the solving process. The success of this feature has already been demonstrated with speedups for the graph generation framework 'SAT modulo symmetries' and the state-of-the-art SMT solver cvc5. We use user propagators to improve the speed with which CaDiCaL can handle ordering problems. To keep this exploration focused, we experiment with one specific ordering problem, namely the Pattern-Avoiding Vertex Ordering problem. This problem has already been extensively studied for specific patterns, due to the fact that some graph classes like chordal and interval graphs can be defined by avoid some pattern.

We implement and evaluate multiple ways to solve this problem with SAT. In our evaluation, we see that user propagators perform well on large instances. Additionally, we show an application of our user propagator approach to One-Planarity Testing, where we achieve a notable speedup over the currently best known encoding.

Zusammenfassung

Moderne SAT solver sind höchst effizient und werden immer noch aktiv erforscht. Trotz ihrer Effizienz bleiben Ordnungsprobleme schwer für SAT solver. Vor kurzem hat der SAT solver CaDiCaL User Propagator eingeführt, welche für mehr und einfachere Interaktion mit dem SAT solver während des Löseprozesses erlauben. Der Erfolg dieser Funktion wurde schon mit Beschleunigungen des Graphgenerierungsprogramms 'SAT modulo symmetries' und dem modernen SMT solver cvc5 demonstriert. Wir benutzen User Propagator, um CaDiCaL im Umgang mit Ordnungsproblemen schneller zu machen. Um diese Erforschung fokussiert zu halten, experimentieren wir mit einem spezifischen Problem, nämlich dem Pattern-Avoiding Vertex Ordering Problem. Dieses Problem wurde schon ausgiebig für spezifische Muster erforscht, aufgrund der Tatsache, dass einige relevante Graphklassen wie chordale Graphen und Intervallgraphen darüber definiert werden können, dass sie ein bestimmtes Muster vermeiden können.

Wir implementieren und bewerten einige Methoden, um dieses Problem mit SAT zu lösen. In unserer Auswertung sehen wir, dass User Propagator gut auf schweren Instanzen funktionieren. Zusätzlich zeigen wir eine weitere Anwendung unseres User Propagators Ansatz zu dem One-Planarity Testing Problem, womit wir eine merkliche Beschleunigung gegenüber der besten bekannte Kodierung erzielen.

Contents

1 Introduction

Motivation Computational problems requiring orderings find numerous applications: In what order does a delivery driver visit customers? How do we serve requests for some shared resources, for example in cloud computing? Can the instructions of a program be reordered to allow for a faster execution without changing the program? How should we order data in a database to efficiently answer queries? What order of DNA sequence fragments achieves the most realistic reconstruction? How do we order the vertices in a layered graph drawing to maximize clarity? To solve these and other ordering problems, numerous abstractions for these applications have been studied, with many of them being NP-hard problems. Examples include Betweenness [Opa79] and generalizations [GM06], the Linear Ordering problem [CML15], the Bandwidth Minimization problem [Sax80], [CM69], [MUPG10], stack and queue number [HR92], [HMP25], and the Travelling Salesman Problem [Rei94] and generalizations [PCSS24].

One approach used to solve NP-hard problems is with SAT solvers. The propositional satisfiability (SAT) problem is one of, if not the single most thoroughly researched problem in computer science. With this research come efficient solvers, like CaDiCaL [Bie+24a] and Kissat [Bie+24b]. Examples where SAT solvers are used for ordering problems are the scheduling problems in [CB94], for certain graph parameters in [Cou16], and for the Hamilton Cycle problem in [VG09], [Soh+14]. There also exist more creative uses for encoding orderings into SAT: In [KSS23], the authors use SAT solvers to generate planar graphs fulfilling some other properties. They make use of a theorem from [Sch89] to ensure planarity: A graph is planar if there exist three orderings of the vertices of the graph such that for every edge with endpoints u and v and for all other vertices w, there is at least one order where both u and v come before w.

Although orderings are often natural ways to express some solution, they are mostly outperformed by other approaches in SAT solving. The Hamilton Cycle problem has better solving strategies in SAT not involving any ordering [Oha+25], and the generation of planar graphs in [KSS23] also presents a generally faster approach. A general bottleneck is the size of these encodings: Using ordering variables $o_{i,j}$ to represent the order of elements i and j, together with clauses representing the transitivity constraint $o_{i,j} \wedge o_{j,k} \rightarrow o_{i,k}$ for each triple of elements i, j, k, results in $\mathcal{O}(n^2)$ variables and $\mathcal{O}(n^3)$ clauses. Yet having faster SAT solving for orderings would be quite helpful, both for problems that still are best encoded by orderings and even to make progress on beating the state of the art where encodings are currently not optimal. The objective of this thesis is the search for faster SAT solving with orderings.

This goal is too general to approach, however. To make our goal focused, we only investigate one problem requiring orderings, namely the Pattern-Avoiding Vertex Ordering problem. The goal of the Pattern-Avoiding Vertex Ordering problem is to order the vertices of a graph such that no subsequence of length k exists fulfilling some conditions of present or not present edges. These conditions are summarized in the pattern graph. An example of a pattern graph and an ordered graph containing the pattern are shown in Figure 1.1. This thesis now provides a general approach for Pattern-Avoiding Vertex Ordering based on SAT solving.

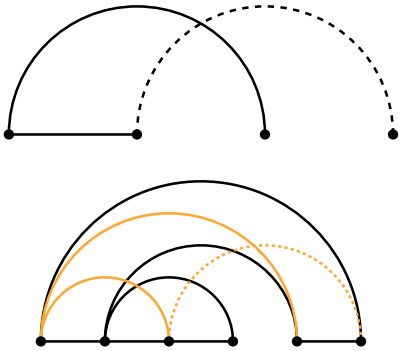

Figure 1.1: At the top is a pattern. A solid edge must be present to match the pattern, while a dashed edge must not be present. At the bottom is an ordered graph containing the pattern at the top. The occurrence of the pattern is highlighted in orange, with the dashed line representing that the edge is not in the graph.

Related work The Pattern-Avoiding Vertex Ordering problem is not new. There are three recent papers that serve both as a great introduction to the problem and as a basis for solving the Pattern-Avoiding Vertex Ordering problem in this thesis. The topics of these papers include, first, an overview of the patterns with three vertices and what graphs can be ordered to avoid these patterns in [FH21b], second, the pattern detection in ordered graphs [DFHP23], and third, the connection between small patterns and intersections of geometric objects in [FH21a]. In general, prior research on the Pattern-Avoiding Vertex Ordering problem has only considered the problem for a fixed pattern, with one interesting exception: For patterns with three vertices, there exists a generalized algorithm based on 2SAT [HMR14].

On the SAT solving side, one major advantage we have over previous encodings of orderings are user propagators [Faz+24], a newly introduced extension to the SAT solver CaDiCaL that allows users more ways to interact with the SAT solver during the solving process. We use them for efficient ordering constraints, but there are already other successes with user propagators. Two notable examples come from the paper introducing user propagators [Faz+24] itself. There, user propagators are used to speed up SAT modulo Symmetries [Sze25], a graph generation framework, and cvc5 [Bar+22], a state-of-the-art SMT solver.

We also note that there is previous work with similar goals. First is to ensure acyclicity in a graph using SAT [GJR14]. This approach also modifies the unit propagation internal to the SAT solver, similar to how a user propagator would. Second is the encoding of orderings, but in ILP [Cou16]. Although the encodings presented are for orderings in ILP, they can be reasonably translated into SAT.

Contribution For the main goal of this thesis, the advancement of solving ordering problems with SAT solvers, we implement and compare eight approaches. An approach based on performing custom unit propagation with user propagators solves the most instances and achieves an average speedup of 4.9595 over the naive encoding. We also present a separate approach based on counterexample-guided abstraction refinement [CGS04] solving fewer instances in total, but achieving an average speedup of 14.9299 over the naive encoding. Using incremental SAT solving, we improve this approach to a speedup of 34.4702.

Additionally, we apply the custom unit propagation to the Optimized One-Planarity Solver [Pup25], which uses partial orders for an encoding of planarity. In this case, we adapt our user propagator to handle partial orders and achieve a speedup of 2.5145 compared to the encoding used by the Optimized One-Planarity Solver.

On top of the practical considerations, we also advance the theory of Pattern-Avoiding Vertex Ordering and of ordering problems with SAT in general. We show in Theorem 3.12 that for every pattern P, there exists at least one graph G that cannot be ordered to avoid P. Further, for general complexity, we show that Pattern-Avoiding Vertex Ordering is NP-hard, coNP-hard and a member of Σ_2^{P} in Theorem 3.13.

For general ordering problems in SAT, we look at the meaning of clauses over ordering variables. We have three especially noteworthy insights in this regard: First is Lemma 4.1, showing that a clause over ordering variables prohibits all orderings that are a reverse topological ordering of a directed graph naturally arising from the clause. Second is Lemma 4.4, showing that any encoding not using additional variables must use $\mathcal{O}(n^3)$ clauses, thus making the naive encoding of transitivity optimal. This point also leads back to the reason why SAT solving with orderings is generally slow. Third is Theorem 4.14, showing that we need at least $\max(k!, n/(k-1))$ axioms in any resolution proof for unsatisfiable instances encoding some ordering.

Outline In Chapter 2, we introduce the necessary concepts in graph theory and SAT solving for this thesis. This includes the formal definition of the Pattern-Avoiding Vertex Ordering problem.

For the content of this thesis, we first look at the Pattern-Avoiding Vertex Ordering from a graph theoretic perspective in Chapter 3. Section 3.1 provides intuition behind the patterns and the graphs that can be ordered to avoid these patterns. We present known facts from previous studies of the Pattern-Avoiding Vertex Ordering problem for specific patterns and give new insight where we can, mostly with the focus of developing techniques that help us solve the Pattern-Avoiding Vertex Ordering problem. We also consider the theoretical complexity of this problem in Section 3.2. Although this does not directly help with solving the problem later on, it rounds out the picture on the problem we are working on.

With the graph theoretic perspective on the problem settled, we come to the strategies we use to solve the Pattern-Avoiding Vertex Ordering problem in Chapter 4. We split this solving process into four parts: First, we define the variable representation of orderings in the SAT solver in the introduction to Chapter 4. Second, we discuss different approaches to make sure that the representation of the ordering does in fact form an ordering in Section 4.1. Here, we implement eight different approaches to be tested later. Along the way, we establish a conditional lower bound on the efficiency of a direct encoding and how to push past this lower bound. Third, we discuss different approaches to prevent an occurrence of the pattern in Section 4.2. Here, we implement three different approaches to be tested later. Fourth, we list the steps we can take outside the SAT solver to speed up the solving process in Section 4.3. This includes filtering out simple instances using fast heuristics that do not significantly impact the time taken by the solver, preprocessing on the instance to simplify the problem given to the SAT solver, and incremental solving, with the aim to solve the instance before we need to fully describe it to the SAT solver. Last, like in Chapter 4, we end this section with another theoretical consideration, this time from a SAT perspective. We look at resolution proofs, which are connected to the best case performance of the SAT solver, specifically for ordering problems.

Chapter 4 provides the description of the approaches we take to solve the Pattern-Avoiding Vertex Ordering problem, and then in Chapter 5, we evaluate them. We present the experimental setup, including the benchmarks used, in Section 5.1. Then, we show the results achieved in Section 5.3.

Last, we consider the transfer of our techniques to another problem, namely One-Planarity Testing. This ties our previous progress back to the overarching goal of furthering the ways we deal with ordering problems in SAT. Chapter 6 introduces One-Planarity Testing and the current state-of-the-art solver presented in [Pup25]. We adapt our user propagator to deal with the partial orderings One-Planarity Testing requires (as opposed to the total orderings for the Pattern-Avoiding Vertex Ordering problem) and show that it can also achieve a faster solver in this case.

2 Preliminaries

This section introduces notation and concepts relevant throughout this thesis. For any $n \in \mathbb{N}$, we denote by $[n] = \{1, 2, \ldots, n\}$ the set of the first n natural numbers. We use $2^S = \{T \mid T \subseteq S\}$ for the set of all subsets of S, and the following notation for subsets of a set S with fixed size t:

$$\binom{S}{t} = \{T \subseteq S \mid |T| = t\}$$

There are two useful views on an ordering of a set S we use. First, as an irreflexive, antisymmetric, and transitive relation $\prec$. For $x, y \in S$, we use $x \prec y$ to denote x coming before y in the order. Second, as a mapping $\pi : S \to [|S|]$ from an element in S to its position in $[|S|]$. We can convert between these two views:

$$x \prec y \iff \pi(x) < \pi(y)$$
$$\pi(x) = |\{y \mid y \prec x\}|$$

The mapping π is bijective, and therefore allows for an inverse function $\pi^{-1} : [|S|] \to S$ mapping a position to the element at that position.

For matrix multiplication, we use the matrix multiplication exponent ω to denote the smallest constant such that it is possible to multiply two $n \times n$ matrices in $n^{\omega+o(1)}$ time. Similarly, we use $\omega(a, b, c)$ for the smallest constant to multiply an $n^a \times n^b$ and an $n^b \times n^b$ matrix in $n^{\omega(a,b,c)+o(1)}$ time. The most recent improvement on these constants comes from [Alm+25], giving $\omega < 2.371339$.

2.1 Graphs

An *undirected graph* $G = (V, E)$ is a set of vertices V together with a set of edges $E \subseteq \{\{u, v\} \mid u, v \in V\}$. A *directed graph* $G = (V, E)$ is a set of vertices V together with a set of edges $E \subseteq \{(u, v) \mid u, v \in V\}$. In both cases, we shorten the notation for an edge to uv. Note that for undirected graphs, $uv \in E$ is equivalent to $vu \in E$. For the directed case, the edge uv is directed from u to v. If the graph G of vertices V or edges E is not clear from context, we use $V(G)$ and $E(G)$ to explicitly denote these vertices and edges, respectively. For undirected graphs, we use $N(u) = \{v \mid uv \in E\}$ for the *open neighborhood* (or just *neighborhood*) of $u \in V$ and $N[u] = N(u) \cup \{u\}$ for the *closed neighborhood*. For directed graphs, we use $N^+(u) = \{v \mid uv \in E\}$ for the *outgoing neighborhood* of $u \in V$ and $N^-(u) = \{v \mid vu \in E\}$ for the *incoming neighborhood*.

We call two graphs G and G' *isomorphic* if there is a bijective mapping of vertices $\varphi : V(G) \to V(G')$ such that an edge between two vertices exists before the mapping if and only if it exists after the mapping.

$$\bigwedge_{u,v \in V} uv \in E(G) \leftrightarrow \varphi(u)\varphi(w) \in E(G')$$

We write $G \cong G'$ to denote that the two graphs G and G' are isomorphic. An *induced subgraph* of G over the vertices $V' \subseteq V$ is denoted by $G[V']$, and is the graph with exactly the vertices V' and all edges in G between any two vertices of V'.

$$V(G[V']) = V'$$
$$E(G[V']) = \{uv \mid u \in V' \wedge v \in V'\}$$

For two graphs G and G', if there is a subset of vertices $V' \subseteq V$ such that $G' = G[V']$, we write $G' \sqsubseteq G$.

Next, we define graphs with orderings, where we mainly follow the conventions of [DFHP23]. An *ordered graph* $(G, \prec_G)$ is an undirected graph $G = (V(G), E(G))$ together with a total order $\prec_G$ on its vertices $V(G)$. An *ordered pattern* $(P, \prec_P)$ is an undirected graph with two distinct sets of edges $P = (V(P), E_M(P) \cup E_F(P))$ together with a total order $\prec_P$ on its vertices $V(P)$. We call $E_M(P)$ *mandatory edges* and $E_F(P)$ *forbidden edges*, and use $E(P) = E_M(P) \cup E_F(P)$ to denote all edges of P. If $uv \notin E(P)$, we call uv an *undecided pair*. For any ordered graph or pattern G, we use $G[i]$ for the ith vertex of G, and $G[\ell..r]$ for the subgraph induced by the vertices from ℓ to r inclusive. Stated formally, we have $G[i] = v$ where $\pi(v) = i$, and $G[\ell..r] = G[\{\pi^{-1}(x) \mid \ell \leq x \leq r\}]$. If a pattern has no forbidden edges, we call it *positive*. Further, if we draw all vertices of P in a straight horizontal line and all edges of P above that line, and we can do so without intersecting two edges (except at their endpoints where they meet in a vertex), we call the pattern outerplanar. To simplify description of patterns, we call the vertices of a pattern $p_1, p_2, \ldots, p_k$, where $p_1 \prec_P p_2 \prec_P \cdots \prec_P p_k$.

We say that an ordered graph $(G, \prec_G)$ *contains* an ordered pattern $(P, \prec_P)$ if and only if there exists a subsequence of vertices in G in the same order as in P, where mandatory edges of P are present in G and forbidden edges in P are absent in G. No statement is made on the existence of edges in G if it does not exist in P. Formally, the graph G contains a pattern P if there exists both a subset of vertices $V'(G) \subseteq V(G)$ with $|V'(G)| = |V(P)|$ and a bijective mapping $f : V(P) \to V'(G)$ such that:

$$uv \in E_M(P) \wedge u \prec_P v \implies f(u)f(v) \in E(G) \wedge f(u) \prec_G f(v)$$
$$uv \in E_F(P) \wedge u \prec_P v \implies f(u)f(v) \notin E(G) \wedge f(u) \prec_G f(v)$$

Additionally, we call the vertices of G forming the pattern an *occurrence* of the pattern. In terms of the mapping f, the occurrence of the pattern is exactly the image $V'(G)$ of f.

In [DFHP23], the authors show that it is possible to construct any positive outerplanar pattern from the pattern with only a single vertex using three operations:

- (*Cover pattern*) Take a pattern P and connect its first vertex $P[1]$ with its last vertex $P[k]$ with a mandatory edge.
- (*Pattern chaining*) Take two patterns P_1 and P_2, and unify the last vertex $P_1[k_1]$ of P_1 with the first vertex $P_2[1]$ of P_2.
- (*Vertex creation*) Take a pattern P, and add a single isolated vertex in front of all other vertices.

We use a slightly modified definition. Our starting graphs are the two patterns with two vertices, once with and once without a connecting edge.

- (*Pattern chaining*) Take two patterns P_1 and P_2, and unify the last vertex $P_1[k_1]$ of P_1 with the first vertex $P_2[1]$ of P_2.

- (*Covered pattern chaining*) Take two patterns P_1 and P_2, unify the last vertex $P_1[k_1]$ of P_1 with the first vertex $P_2[1]$ of P_2, and also connect the outermost vertices $P_1[1]$ and $P_2[k_2]$ with a mandatory edge.

With this construction, we cannot achieve the pattern with exactly one vertex, but all other positive outerplanar patterns are possible. To see why, we note that chaining the pattern with two vertices and no edge followed by a pattern P is the same as adding an isolated vertex in front of P, so we can simulate the vertex creation operation. Additionally, applying the operation to cover a pattern multiple times in a row does nothing compared to doing it once, so we can integrate this directly after the pattern chaining. The exception to this is the pattern with two vertices that are connected by an edge, which has no possible preceding chaining step. This pattern is however directly part of our base case.

We now define the *Pattern-Avoiding Vertex Ordering* problem. Given are an instance graph G and an ordered pattern $(P, \prec_P)$. The goal is to find an ordering $\prec_G$ such that the ordered graph $(G, \prec_G)$ does not contain $(P, \prec_P)$. We use $\mathcal{G}_P$ to denote the set of all graphs that can be ordered to avoid the pattern P. The Pattern-Avoiding Vertex Ordering for a pattern P is equivalent to recognizing the graphs in $\mathcal{G}_P$.

We note that this problem is already studied extensively for specific patterns. Here, we again refer to [FH21b] for an overview over all patterns with three vertices, and [FH21a] for the connection to intersections of geometric objects.

2.2 SAT

We denote by $\mathbb{B} = \{\top, \bot\}$ the set of Booleans, where $\top$ stands for true and $\bot$ stands for false. A *SAT instance* $\mathcal{F} = (U, K)$ consists of a set of Boolean variables U and a set of clauses K. We generally denote the number of variables by $N = |U|$, and the number of clauses by $M = |K|$. The set of literals $\widetilde{U} = U \cup \{\overline{x} \mid x \in U\}$ of a SAT formula is the set of variables together with the set of negated variables. Each clause is a set of literals, so for the set of clauses C, we have $C \subseteq 2^{\widetilde{U}}$. We call a mapping from literals to Booleans $\phi : \widetilde{U} \to \mathbb{B}$ a *variable assignment*, which has to fulfill $\phi(\overline{x}) = \neg\phi(x)$. When we call a literal x assigned, it means $\phi(x) = \top$. A variable assignment ϕ *satisfies* a literal $x \in \widetilde{U}$ if and only $\phi(x) = \top$. Further, a clause $C \in K$ is satisfied if and only if any literal in it is satisfied, and a formula $\mathcal{F}$ is satisfied if and only if all of its clauses are satisfied. A formula $\mathcal{F}$ is called *satisfiable* if a variable assignment ϕ satisfying $\mathcal{F}$, and *unsatisfiable* otherwise. We call two formulas equi-satisfiable if the satisfiability of one implies the satisfiability of the other.

Let C and D be two clauses. When we have $C \subset D$, where C is specific smaller than D, then we call C the *stronger* clause and D the *weaker* clause. This naming is due to the fact that an assignment ϕ satisfying C implies that ϕ also satisfies D, but the reverse direction is not true. Therefore, the clause C imposes a stronger constraint on any satisfying assignment. If we have a clause D in our SAT instance, and replace D by $C \subset D$, we call this *strengthening*. Likewise, replacing C with $D \supset C$ is called *weakening*.

If a SAT instance is unsatisfiable, it is always possible to derive the empty clause using only *resolution*. Resolution is a binary operation on clauses, and we use $\otimes$ to denote the resolution operation. The resolution of two clauses $C \cup \{x\}$ and $D \cup \{\overline{x}\}$ results in $C \cup \{x\} \otimes D \cup \{\overline{x}\} = C \cup D$. The variable x is the variable we resolve over, and if we want to make x explicit, we write the resolution operation as $\otimes_x$. Note that in general, $C \otimes D$ is only a valid operation if there is a variable to resolve over. Let ϕ be a variable assignment and C, D, and X clauses with

$C \otimes D = X$. If ϕ satisfies C and D, then ϕ also satisfies X. Additionally, weakening any clause during a resolution proof maintains the correctness of the proof system, but is not required for completeness.

We use the SAT solver CaDiCaL [Bie+24a] for experimental evaluation. In general, the more performant solver Kissat [Bie+24b] exists, performing best on the sequential track in 2025 SAT Competition [CFHI25]. CaDiCaL does however offer the implementation of user propagators [Faz+24] needed in some cases, and to keep the solver consistent, we use CaDiCaL even when user propagators are not used.

3 Graph theory

Before we describe ways to solve the Pattern-Avoiding Vertex Ordering problem with SAT solvers, we make some observations purely from a graph theoretic perspective. Specifically, we investigate the graph classes $\mathcal{G}_P$ corresponding to the patterns P. In Section 3.1, we show properties of $\mathcal{G}_P$ beyond the characterization as the graphs that can be ordered to avoid P. This covers the meaning of the operations used to build the positive outerplanar patterns, complete alternative characterizations of $\mathcal{G}_P$ for some patterns P, and some additional properties. Then, in Section 3.2, we provide more insight into the structure of the graph class $\mathcal{G}_P$ itself, and the hardness of recognizing the graph class, or equivalently, solving the Pattern Avoiding Vertex Ordering problem.

3.1 Meaning of patterns

The Pattern-Avoiding Vertex Ordering problem is already studied for some specific patterns. Examples include chordal and interval graphs, which can be defined as graphs that can avoid the respective pattern in Figure 3.1 [FH21b].

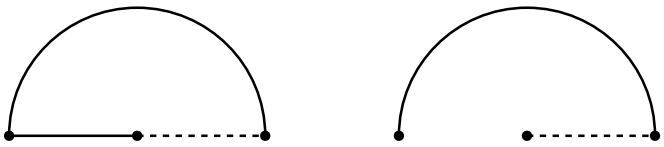

Figure 3.1: Some graph classes, like chordal and interval graphs, can be defined as the graphs that can be ordered to avoid some pattern. On the left is the pattern for chordal graphs. On the right is the pattern for interval graphs.

These graph classes can be defined in different, more intuitive ways as well. Chordal graphs are the graphs where every cycle of length at least 4 has a chord, that is, an edge connecting two vertices that are not adjacent in the cycle. Interval graphs can be represented as intersecting intervals on the real number line, that is, each vertex v is represented by an interval I_v and two vertices u and v are connected by an edge uv if their intervals I_u and I_v intersect. These alternative definitions help to visualize these graphs, find interesting properties about them and recognize when they appear in applications. For example, let each vertex represent a person working for a company, and two vertices are connected if the corresponding people are in the office at the same time. We assume that every person arrives at some fixed point of the day, and leaves at some fixed later point. In this case, it is straightforward to see that this graph is an interval graph with the intervals being the time the respective person is in the office, while it is less obvious when only looking at the pattern.

The goal of this section is to show more intuitive ways to describe the graph class $\mathcal{G}_P$ corresponding to the pattern P. In Section 3.1.1, we investigate various operations on patterns, and how they modify the meaning of the pattern. In Section 3.1.2, we look at the meanings of a collection of patterns. This includes both known patterns, like the chordal and interval

graphs given as an example, as well as new insights. A more relaxed approach to the meanings of patterns is discussed in Section 3.1.3: Instead of figuring out the precise meaning of a pattern P, we relate its graph class $\mathcal{G}_P$ to other graph classes. This still gives us an idea of the meaning of the pattern P, and shows properties of $\mathcal{G}_P$ that are useful when solving the Pattern-Avoiding Vertex Ordering problem later.

3.1.1 Operations on patterns

One of the two possible recursive steps in the definition of outerplanar positive patterns is to take two patterns P_1 and P_2, and identify the rightmost vertex of P_1 with the leftmost vertex of P_2. Avoiding the chained pattern P is now equivalent to partitioning the vertices $V(G)$ into two sets V_1 and V_2 where V_1 can be ordered to avoid P_1, and likewise with V_2. This is formally stated in Lemma 3.1.

Lemma 3.1: *For two ordered patterns P_1 and P_2, let $v_{1,r}$ be the rightmost vertex of P_1 and $v_{2,\ell}$ the leftmost vertex of P_2. Let P be the pattern made from identifying $v_{1,r}$ and $v_{2,\ell}$. A graph G can be ordered to avoid P if and only if the vertices of G can be partitioned into V_1 and V_2 such that the graph induced by V_i for $i \in \{1, 2\}$ can be ordered to avoid P_i.*

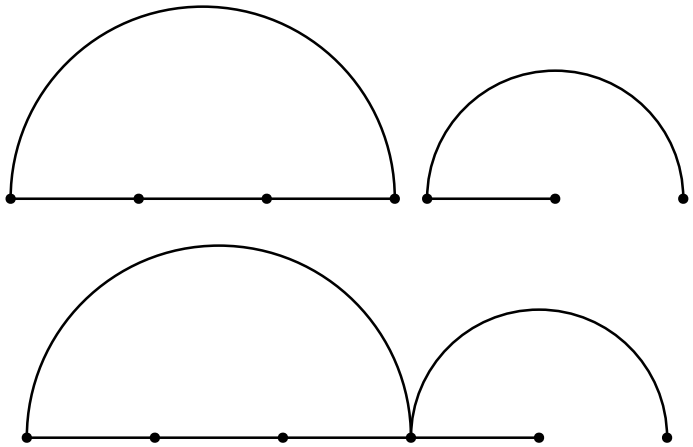

Figure 3.2: An example of pattern chaining without a covering edge. Chaining the patterns on the top results in the pattern on the bottom.

Proof. First, let $\prec$ be an ordering of G avoiding P. We want to show that there exists a partition of V into vertex sets V_1 and V_2 with orderings $\prec_1$ and $\prec_2$ avoiding P_1 and P_2, respectively. Let V_2 be the set of all vertices v such that there is an occurrence of P_1 in G ordered with $\prec$, and v is the last vertex in this occurrence. Further, let $V_1 = V \setminus V_2$. Then both V_1 and V_2 are already ordered to avoid P_1 and P_2, respectively. For V_1, any occurrence of P_1 would contain the last vertex of an occurrence of P_1, but these vertices are by definition not part of V_1, but V_2. For V_2, assume there is an occurrence of P_2. We also know that the first vertex in this occurrence is preceded by an occurrence of P_1 in the complete ordering of V by definition of V_2. This now gives an occurrence of P in V, which is not possible by the assumption that $\prec$ avoids P. An example of this direction is shown in Figure 3.3.

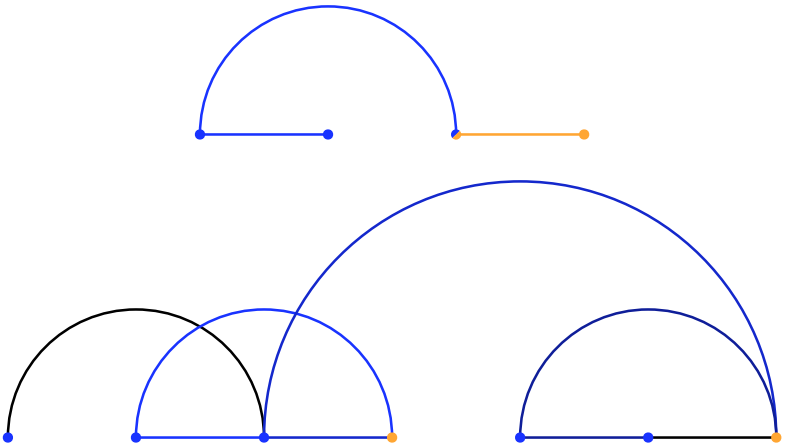

Figure 3.3: An example of the construction of V_1 and V_2 avoiding some pattern parts when the entire graph G is ordered to avoid some pattern P. The pattern P of this example is shown at the top. The two parts are P_1 from the first to third vertex and P_2 from the third to fourth vertex. At the bottom is an ordered graph G with the edges of the occurrences of P_1 highlighted in blue, using different shades for different occurrences. The vertices of V_1 are blue and the vertices of V_2 are orange. Every vertex at the end of an occurrence of P_1 is in V_2, while the remaining are in V_1.

For the other direction, we have a decomposition into V_1 and V_2 with orders $\prec_1$ and $\prec_2$ avoiding P_1 and P_2, respectively. From this, we show that it is possible to order G while avoiding P. The concrete ordering $\prec$ places all of V_1 before V_2, and has V_1 and V_2 ordered with $\prec_1$ and $\prec_2$, respectively. The following is an explicit construction of the ordering $\prec$ of G avoiding P:

$$u \prec v \iff \begin{cases} \top & u \in V_1 \wedge v \in V_2 \\ \bot & u \in V_2 \wedge v \in V_1 \\ u \prec_1 v & u, v \in V_1 \\ u \prec_2 v & u, v \in V_2 \end{cases}$$

To show the correctness, we assume there is an occurrence of the pattern P in the ordering $\prec$. Let x be the vertex of the pattern taking on the role of the identified vertex above. If $x \in V_1$, then there is an occurrence of P_1 in V_1. Likewise, if $x \in V_2$, then there is an occurrence of P_2 in V_2. One of these cases has to occur, but neither one can due to the assumption that both V_1 and V_2 are ordered to avoid P_1 and P_2, respectively. An example of this direction is shown in Figure 3.4.

■

Another insight of Lemma 3.1 is that when chaining without a covering edge, the order of the patterns is not relevant. This brings us to whether the same is true when chaining patterns with a covering edge. We can give an example where this is not the case. The graphs shown at the top of Figure 3.5 can be ordered to avoid the pattern at the bottom left, but not the one on the bottom right, although both can be created with the same steps up to ordering.

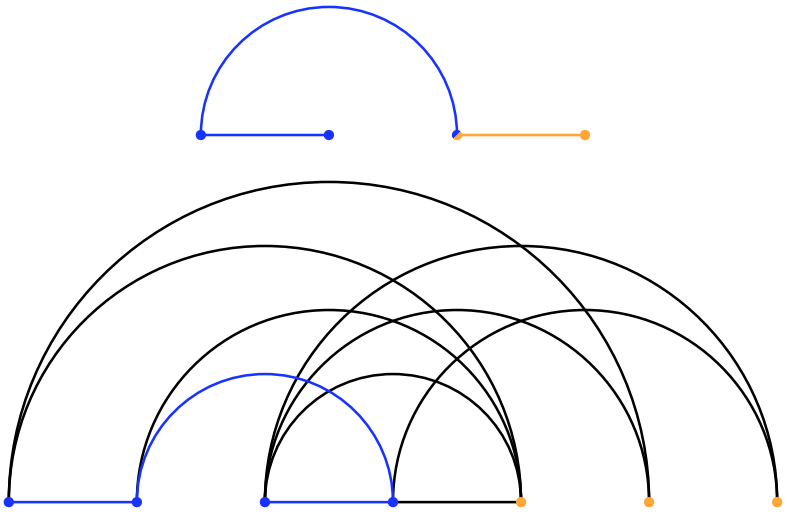

Figure 3.4: An example of the construction of $\prec_G$ avoiding some pattern parts when there are known parts V_1 and V_2 of the graph that are ordered to avoid P_1 and P_2, respectively. The pattern P of this example is shown at the top. The two parts are P_1 from the first to third vertex and P_2 from the third to fourth vertex. At the bottom is an ordered graph with the blue vertices ordered to avoid P_1 and the orange vertices ordered to avoid P_2. The edges between two blue vertices are highlighted in blue, and no edge between two orange vertices exists. Chaining the blue and orange vertices together avoids P.

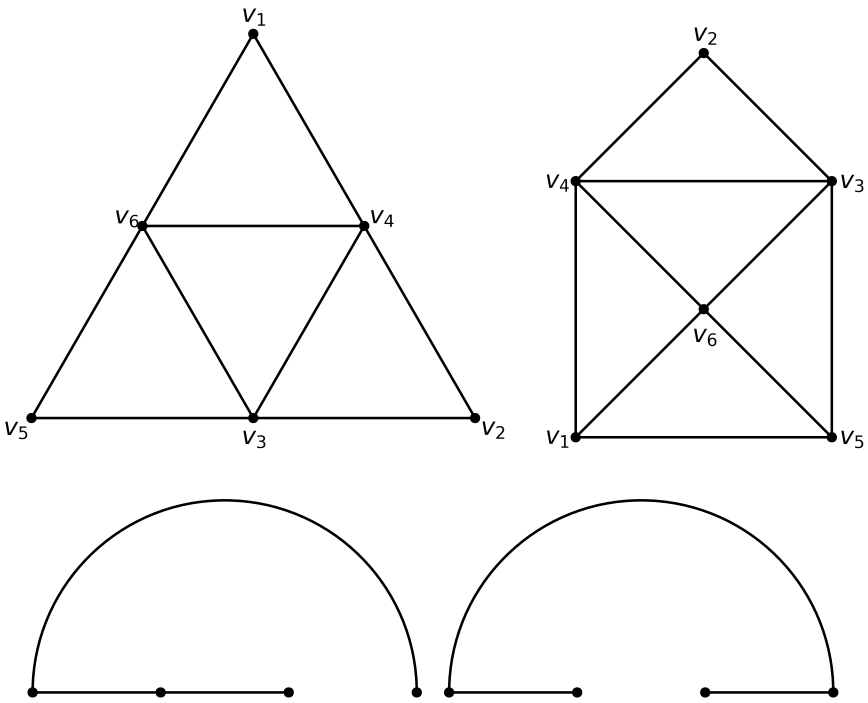

Figure 3.5: Counterexamples to covered pattern chaining having an interpretation similar to the one for uncovered pattern chaining given in Lemma 3.1. Both graphs at the top can be ordered to avoid the bottom left pattern with the order given in the vertex label, but cannot be ordered to avoid the bottom right pattern.

Apart from the construction operations for outerplanar positive patterns, there are two interesting operations on patterns in general. These are *reversing* and *complementing*. Both of these operations, together with the Lemmas we present here, are already discussed in [FH21b], although reversing is instead called mirroring.

First, we look at reversing. For a pattern P ordered with $\prec_P$, the reversed pattern P^R uses the same underlying graph P, but instead orders the vertices with $(\prec_{P^R}) = (\succ_P)$. The reversed pattern describes the same graph class and as the original pattern, and we state this result here as a lemma.

Lemma 3.2: *Let P be any pattern. Then $\mathcal{G}_P = \mathcal{G}_{P^R}$.*

Proof. Let $G \in \mathcal{G}_P$ be a graph, and $\prec_G$ an ordering that avoids P. Then the ordering $\succ_G$ avoids P^R. Therefore, we have $\mathcal{G}_P \subseteq \mathcal{G}_{P^R}$. Equality of these sets can be seen from the following:

$$\mathcal{G}_P \subseteq \mathcal{G}_{P^R} \subseteq \mathcal{G}_{(P^R)^R} = \mathcal{G}_P$$

■

Taking the complement of a pattern is different to its usual meaning in graph theory. Let $P = (V(P), E_M(P) \cup E_F(P))$ be a pattern, then the complement pattern $\overline{P}$ is created by swapping the mandatory and forbidden edges:

$$\begin{aligned} V(\overline{P}) &= V(P) \\ E_M(\overline{P}) &= E_F(P) \\ E_F(\overline{P}) &= E_M(P) \end{aligned}$$

Complementing a pattern P also complements the graph class $\mathcal{G}_P$ the pattern describes. Again, this is already noted in [FH21b]. Like with reversing, we formalize this in a lemma:

Lemma 3.3: *Let P be any pattern. Then $\mathcal{G}_{\overline{P}} = \overline{\mathcal{G}}_P$.*

Proof. Let G be a graph with an order $\prec_G$. We show that if G contains P, then $\overline{G}$ contains $\overline{P}$. For this, let the vertices $v_1, v_2, \ldots, v_k$ be an occurrence of P in G. Further, let $v_i v_j \in E(G)$ be an edge that must exist due to the corresponding vertices in the pattern being connected by a mandatory edge. In this case, $\overline{P}$ requires $v_i v_j$ to not be connected by an edge, which is the case in the complement graph $\overline{G}$. A similar argument shows that $\overline{G}$ has the mandatory edges required by $\overline{P}$.

Now, let $G \notin \mathcal{G}_P$ be a graph that cannot be ordered to avoid P. From the argument above, we know that any ordering of $\overline{G}$ must contain $\overline{P}$, and so it follows that $\overline{G} \notin \mathcal{G}_{\overline{P}}$. This shows the inclusion $\mathcal{G}_{\overline{P}} \subseteq \mathcal{G}_P$. Equality of these sets can be seen from the following:

$$\mathcal{G}_P = \mathcal{G}_{\overline{\overline{P}}} \subseteq \mathcal{G}_{\overline{P}} \subseteq \mathcal{G}_P$$

■

3.1.2 Meanings of specific patterns

For some patterns P, we can give interpretations of the graph class $\mathcal{G}_P$ they describe. The results of this section mainly come from already known patterns, and from problems we can model using the insight of Lemma 3.1.

First, we look at all three patterns with two vertices, once with no edge, once with a mandatory edge, and once with a forbidden edge. The pattern with no edge is found once the graph has at least two vertices, so it describes only the graphs with no or only one vertex. The pattern with a mandatory edge is found once there are two vertices connected by an edge, so it describes all empty graphs. On the other hand, the pattern with a forbidden edge is found once there are two vertices not connected by an edge, so it describes all complete graphs. Alternatively, this is the complement of the pattern with two vertices and a mandatory edge, using Lemma 3.3.

The paper [FH21b] gives an interpretation for all patterns with three vertices. For some of these, we can also find an interpretation using the result on pattern chaining from Lemma 3.1. Take, for example, the pattern on three vertices where the first two are connected by a mandatory edge, and last two by a forbidden edge. This pattern describes split graphs, which are graphs where the the set of vertices V can be partitioned into two sets V_1 and V_2 such that V_1 is an independent set and V_2 is a clique. This result is mentioned in [FH21b], but is also a direct consequence of Lemma 3.1 when chaining the pattern for an empty graph with the pattern for a complete graph. The graph classes of patterns which are left out by this are, up to complementing and reversing, linear forests, forests, intervals graphs, chordal graphs, triangle-free graphs, and comparability graphs. The patterns for these classes are shown in Figure 3.6.

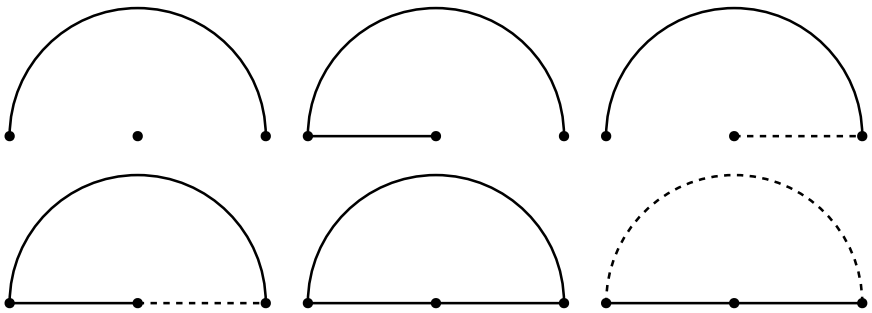

Figure 3.6: All patterns on three vertices that have a covering edge. On the top, from left to right, are the patterns for linear forests, forests, and interval graphs. On the bottom, from left to right, are the patterns for chordal graphs, triangle-free graphs, and comparability graphs.

A way that is often used to give meanings to graphs which can be ordered to avoid some pattern are intersections of objects, for example two-dimensional shapes. Specifically, we take the set of these objects as the vertices of our graph, and connect two of them by an edge if the corresponding objects intersect. An example of this we have already seen are interval graphs, where the objects are intervals in the real number line. More on intersection representations are presented in [FH21a]. Patterns with such a representation often contain both mandatory and forbidden edges. This is due to the patterns often giving insights in the form of an implication: If some objects intersect (meaning that the mandatory edges are present), then some other objects must also intersect (meaning that some forbidden edge may not be present).

We give an example of this with intersection graphs. Let G be an intersection graph, and $\mathcal{I} = \{[\ell, r] \subset \mathbb{R} \mid \ell \leq r\}$ be an interval representation of G. For a vertex $v \in V(G)$, we use $I_v \in \mathcal{I}$ to denote the interval it represents, where ℓ_v and r_v are the left and right endpoint of I_v.

We define the order $\prec_G$ of the vertices of G sorted by the right endpoint of the corresponding interval in the intersection representation. If the intervals of two vertices have the same endpoint, we order them arbitrarily. Let $u, v, w \in V$ be three vertices with $u \prec_G v \prec_G w$, where $uw \in E(G)$. From the order of u, v, and w, we know that $r_u \leq r_v \leq r_w$. Additionally, from $uw \in E(G)$, we know that I_u and I_w intersect, and it follows that $\ell_w \leq r_u$. Putting these two inequality chains together gives $\ell_w \leq r_u \leq r_v \leq r_w$, and it follows that $r_v \in I_w$. We also have $r_v \in I_v$, so I_v and I_w have at least r_v in common and the edge vw must exist in $E(G)$. Figure 3.7 visualizes this reasoning.

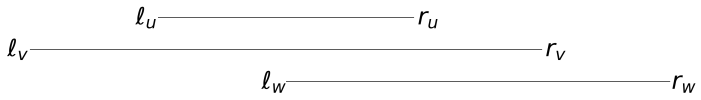

Figure 3.7: An example of three intervals, with the interval for u at the top, v in the middle, and w at the bottom. Here, $u \prec_G v \prec_G w$ due to $r_u \leq r_v \leq r_w$. The intervals for u and w intersect, so we have $\ell_w \leq r_u$. This means that the intervals for v and w must also intersect.

This gives us the kind of implication we were looking for: If two intervals intersect, then the right interval must also intersect everything in between.

Next, we look at some graph properties that are already formulated in terms of vertex orderings and have patterns that are equivalent to the constraint the graph property imposes on the vertex ordering in their definition. Here, we have two examples:

- **Degeneracy**: A graph G is called *k-degenerate* if is it is possible to delete a vertex with degree at most k until the graph is empty. An example of this is forests, which are exactly the 1-degenerate graphs: Every non-empty forest contains a leaf, and after deleting a leaf, we are still left with a forest. Therefore, when starting with a forest, we can repeatedly delete a leaf until the entire graph is empty. The pattern for forests does generalize for k-degenerate graphs: We construct the pattern with $k + 2$ vertices and connect the left-most vertex to all other vertices, that is, we use $E_M(P) = \{p_1 p_i \mid 2 \leq i \leq k + 2\}$ as our set of edges. The pattern for 3-degenerate graphs is shown in Figure 3.8. This prohibits any vertex from having $k+1$ or more neighbors to its right. An order avoiding this pattern is exactly a valid deletion order for showing that the graph is k-degenerate.

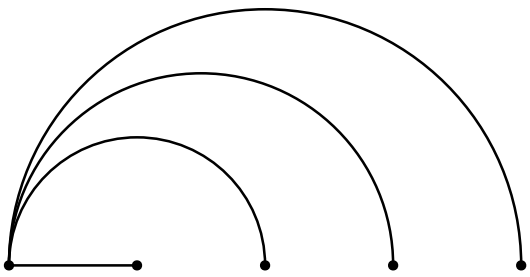

Figure 3.8: The pattern for the class of graphs that are 3-degenerate.

- **Bandwidth Minimization**: Another example is the Bandwidth Minimization problem (see for example [CM99]), which asks to order the vertices of an undirected graph G such that the maximum distance between two vertices connected by an edge is at most k. Formally, the Bandwidth Minimization problem requires the following:

$$\max_{uv \in E}(|\pi(u) - \pi(v)|) \leq k$$

  This translates rather directly into the Pattern-Avoiding Vertex Ordering problem: We take the pattern P with $k+2$ vertices, with only one mandatory edge $E_M(P) = \{p_1 p_{k+2}\}$. The pattern for graphs with bandwidth at most 3 is shown in Figure 3.9. This pattern is present in an ordering of a graph if there are two vertices that are at least $k+1$ apart, and therefore violate the constraint of the Bandwidth Minimization problem.

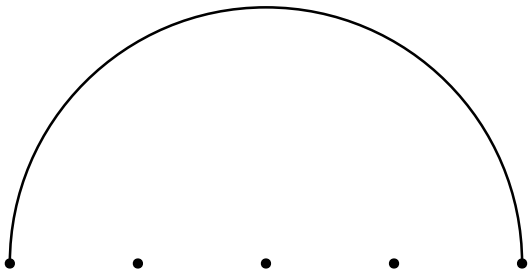

Figure 3.9: The pattern for the class of graphs that have bandwidth at most 3.

These two examples do not make use of the pattern chaining in Lemma 3.1. With Lemma 3.1, there are more known graph problems we can describe using Pattern-Avoiding Vertex Orderings. Specifically, we look at problems where we are asked to partition the vertices into sets where each set has to be in a graph class we have already seen before. Like the previous examples, this list is not exhaustive, but it gives an overview over known problems that can be expressed by Pattern-Avoiding Vertex Ordering using the insight of Lemma 3.1.

- **Graph Coloring**: In the Graph Coloring problem, we are given an undirected graph and a maximum number of colors k. The goal is to assign each vertex $v \in V$ a color (or rather integer) $c(v)$, where $c(v) \in [k]$, such that for each pair of vertices u and v connected by an edge, these two vertices have different colors.

$$c : V \to [k]$$
$$\bigwedge_{uv \in E} c(u) \neq c(v)$$

  Let $V_i = \{v \mid c(v) = i\}$ be the set of all vertices with color i. The graph induced by these vertices $G[V_i]$ has no edges, as if there were an edge $uv \in E(G[V_i])$, this edge uv would violate the condition that connected vertices have different colors. With this, we can alternatively define graph coloring as partitioning a graph into k independent sets. Therefore, the Graph Coloring problem is equivalent to the Pattern-Avoiding Vertex Ordering problem where the pattern has $k+1$ vertices and each pair of consecutive vertices is connected by a mandatory edge, that is, we have $E_M(P) = \{p_i p_{i+1} \mid 1 \leq i \leq k\}$. From an order $\prec_G$ avoiding P, we can even recover a valid k-coloring: We iterate over

the vertices $v \in V(G)$ from left to right. Let $N_{\text{left}}(v)$ be the left neighborhood of v. We set the color of v to one more than the largest color in the left neighborhood, or to 1 if the left neighborhood is empty.

$$c(v) = \begin{cases} 1 & N_{\text{left}}(v) = \emptyset \\ \max\{c(u) \mid u \in N_{\text{left}}(v)\} & N_{\text{left}}(v) \neq \emptyset \end{cases}$$

This coloring does not give the same color to two neighboring vertices x and y, because the coloring process implies $c(x) < c(y)$ when $x \prec_G y$. This reconstruction of a coloring from an order is known and used for example in [Cul92] for a local search algorithm for graph coloring. Additionally, this never uses the color $k+1$ or larger, as this would cause a path from left to right with at least $k+1$ vertices. This contradicts the assumption that G was ordered to avoid exactly such a path. An example of this pattern for 3-colorability is shown in Figure 3.10.

Figure 3.10: The pattern for the class of graphs that are 3-colorable. Equivalently, the graphs that can be decomposed into three independent sets.

- **Vertex Cover**: In the Vertex Cover problem, we are given an undirected graph and a maximum number of vertices k. The goal is to find a set of vertices $V' \subseteq V$ with the maximum size $|V'| \leq k$, such that for each edge $uv \in E$, at least one of its endpoints is chosen.

$$\bigwedge_{uv \in E} u \in V' \vee v \in V'$$

Note that $G[V \setminus V']$ contains no edges. If there were an edge $uv \in E(G[V \setminus V'])$, then both u and v would not be in V', even though they are connected by an edge. This would violate the constraint of the Vertex Covering problem, and so does not appear in a valid solution. The Vertex Cover problem is therefore equivalent to partitioning the graph into a set of size at most k (namely V') and an independent set (namely $V \setminus V'$). Therefore, the Vertex Cover problem is equivalent to the Pattern-Avoiding Vertex Ordering problem where the pattern has $k+2$ vertices, and the only one mandatory edge $E_M(P) = \{p_1 p_2\}$. This pattern P is the result of chaining the pattern for an independent set with k times the pattern for a graph with one vertex. An example of this pattern for a vertex cover of size 2 is shown in Figure 3.11.

Figure 3.11: The pattern for the class of graphs that have a vertex cover of size 2. Equivalently, the graphs that can be decomposed into one independent sets and two single vertices.

- **Feedback Vertex Set**: In the Feedback Vertex Set problem, we are given an undirected graph and a maximum number of vertices k. The goal is to find a set of vertices $V' \subseteq V$ with the maximum size $|V'| \leq k$, such that for each cycle in the graph G, at least one vertex is in V'. With this requirement, we know that $G[V \setminus V']$ contains no cycle, and is therefore a forest. This is similar to the Vertex Cover problem, but the remaining graph

after removing the k vertices is a forest, not an empty graph. Therefore, the Feedback Vertex Set problem is equivalent to the Pattern-Avoiding Vertex Ordering problem where the pattern has $k + 3$ vertices and the mandatory edges $E_M(P) = \{p_1p_2, p_1p_3\}$. An example of this pattern for a feedback vertex set of size 2 is shown in Figure 3.12.

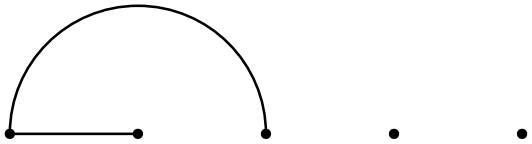

Figure 3.12: The pattern for the class of graphs that have a feedback vertex set of size 2. Equivalently, the graphs that can be decomposed into one forest and two single vertices.

- **Independent Feedback Vertex Set**: In the Feedback Vertex Set problem, we are given an undirected graph. The goal is to find an independent set of vertices $V' \subseteq V$, such that for each cycle in the graph G, at least one vertex is in V'. This is similar to the Feedback Vertex Set problem, where the size constraint on V' is replaced with the constraint that V' is independent. Therefore, the Independent Feedback Vertex Set is equivalent to the Pattern-Avoiding Vertex Cover problem where the pattern has 4 vertices and the mandatory edges $E_M(P) = \{p_1p_2, p_1p_3, p_3p_4\}$. The pattern for independent feedback vertex sets is shown in Figure 3.13.

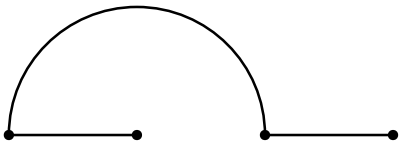

Figure 3.13: The pattern for the class of graphs that have an independent feedback vertex set. Equivalently, the graphs that can be decomposed into one forest and one independent set.

These are the examples of well-known problems that relate to chained patterns as shown in Lemma 3.1. Technically, we can derive other problems from this definition (for example, is it possible to partition the vertices of a graph into a tree, an independent set, and a bandwidth 4 graph?), but these problems are not generally interesting or meaningful.

Another set of well-known patterns relate to the *stack number* and *queue number* of a graph. The patterns we present next, together with the bounds they imply presented in the following, are discussed in [HR92]. For the stack number, we look at k-twists, a pattern with $2k$ vertices and the mandatory edge $E_M(P) = \{p_ip_{i+k} \mid i \in [k]\}$. For the queue number, we look at k-rainbows, a pattern with $2k$ vertices and the mandatory edge $E_M(P) = \{p_ip_{2k+1-i} \mid i \in [k]\}$. An example for both of these patterns is shown in Figure 3.14, with $k = 3$ for both.

If a graph G cannot be ordered to avoid a k-twist, then the stack number must be at least k. This bound is one-sided: If the graph can be ordered to avoid k-twist, then the stack number may still be larger. On the other hand, the queue number can fully be determined by solving the Pattern-Avoiding Vertex Ordering problem with the rainbow pattern: If a k-rainbow cannot be avoided, then the queue number is at least k, but if it can be avoided, the queue number is at most $k - 1$.

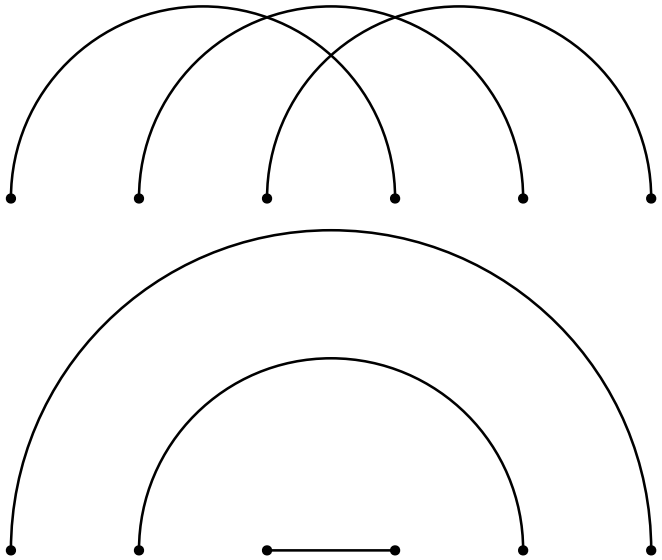

Figure 3.14: At the top is a 3-twist, which may not appear in a valid ordering for stack number 2. At the bottom is a 3-rainbow, which may not appear in a valid ordering for queue number 2.

Patterns like the ones for stack and queue number are positive, every vertex is only ever the left endpoint or the right endpoints of an edge, and the left endpoints all come before the right endpoints. These two sets of patterns are not the only ones with this property, as an example, the pattern for forests also has this property. These patterns can also be nicely represented as a collection of points on a two-dimensional grid. This idea is for example used in [HMP25] to investigate a mix of queue and stack numbers. For each edge $uv \in E(P)$, we represent it by the point $(\pi(u), \pi(v))$, where $u \prec_P v$. Likewise, when we have an ordered graph, for each edge $uv \in E(G)$ with $u \prec_G v$, we represent it by the point $(\pi(u), \pi(v))$. The ordered graph G now contains the pattern P if and only if we can find the geometric representation of P in the geometric representation of G, where we can arbitrarily and independently translate and scale both dimensions of the geometric representation of P. An example of this is shown in Figure 3.15. The reason for the earlier stated condition that each vertex of the pattern has to be only a left or a right endpoint, and that all left endpoints come before all right endpoints, is to make sure the two dimensions are fully independent. If a vertex were the left endpoint of some vertex, and the right endpoint of some other, there would be some pairs of points where the position of a point in one dimension had to match the position of the other point in the other dimension. Likewise, if there were a right endpoint before a left endpoint, it would impose a constraint that the horizontal position of one point must be higher than the vertical position of another point.

The geometric representation of patterns also helps in designing detection algorithms. For example, [HMP25] uses the insight that finding the k-rainbow pattern is equivalent to finding a strictly decreasing subsequence of length k in the set of points. This finds the edges from innermost to outermost. Another example we can give is for the pattern P shown in Figure 3.16, which has $k = 4$ and mandatory edges $E_M(P) = \{p_1p_4, p_2p_3, p_2p_4\}$. Detecting this pattern in an ordered graph is equivalent to finding an axis-aligned rectangle in a set of points where all corner points need to be part of the set. This has a simple $\mathcal{O}(m^2)$ algorithm: Choose two points that make up opposite corners, and then check if the other two corners also exist in

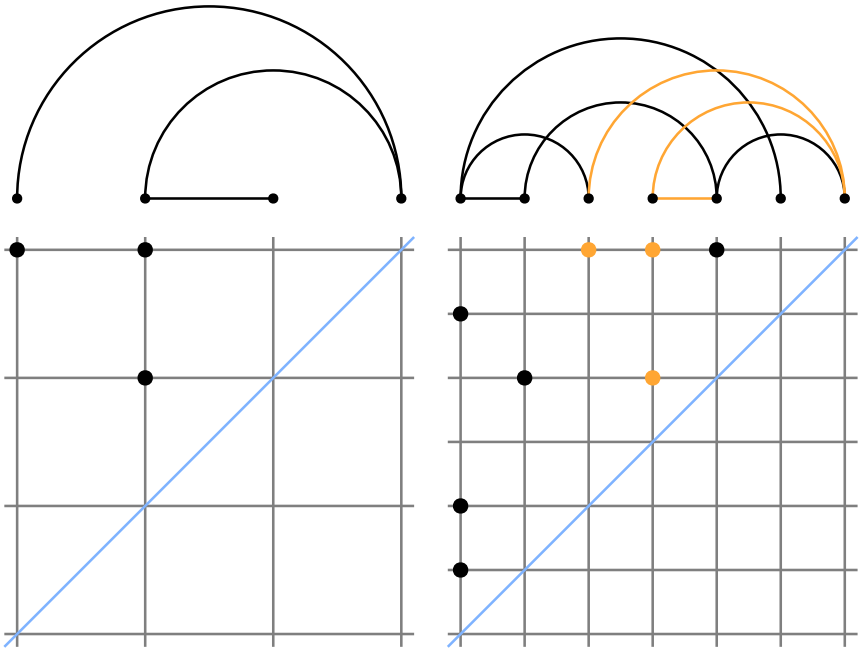

Figure 3.15: On the top are a pattern and an ordered graph containing the pattern, highlighted in orange. On the bottom are the geometric representations of the pattern and the ordered graph. The occurrence of the pattern in the geometric representation is also highlighted in orange. All points in the geometric representation are above the light blue line, due to the fact that the endpoint of the edge with smaller position determines the positioning along the vertical axis, and the other endpoint the positioning along the horizontal axis.

$\mathcal{O}(1)$ time per check. For this check to be in $\mathcal{O}(1)$, we also need to store the set of points in a two-dimensional Boolean array, which takes $\mathcal{O}(n^2)$. This can be sped up for dense graphs: Store each row of points in a bitset, where the ith position is 1 if there is a point at the ith position in the row. Now, there is a rectangle of points if and only if there are two bitsets which have at least two 1 in common. This can be checked by taking each pair B_i and B_j of bitsets and checking if there are at least two 1s in $B_i \& B_j$. This takes $\mathcal{O}(n^3/W)$, where W is the word size. We do note that this algorithm is only faster than the one in $\mathcal{O}(n^2)$ for a small part of possible instances, due to the fact that graphs with $m > n\sqrt{(n-1)/2}$ cannot be ordered to avoid P anyway. This is proven in Lemma 3.9 in Section 3.1.3.

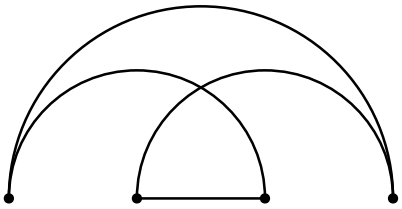

Figure 3.16: The pattern that has a rectangle as its geometric representation.

So far, every pattern we looked at fits into some category: A consequence of the pattern chaining presented Lemma 3.1, a nice geometric interpretation, and so on. There is one additional pattern for which we can give an intuition that does not fit in any category of patterns shown so far. This is the pattern P with $k = 4$ vertices and the mandatory edges $E_M(P) = \{p_1p_2, p_2p_3, p_3p_4, p_1p_4\}$. This pattern is shown in Figure 3.17. To see what this

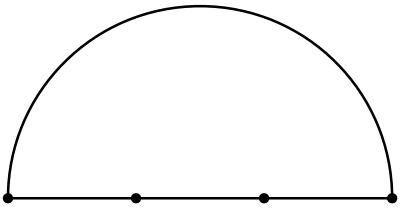

Figure 3.17: Another pattern for which we can give an intuition. If G is ordered to avoid this pattern, then for any two vertices of the graph G, their common neighborhood is either completely between them, or not one vertex of their common neighborhood is between them.

pattern means, let $x, y \in V(G)$ be two distinct vertices. Let $C = N(x) \cap N(y)$ be their common neighborhood. Consider an ordering $\prec_G$ that avoids P, and without loss of generality, let $x \prec_G y$. We partition C into two sets, namely the vertices C_{inside}, which are between x and y, and the vertices C_{outside}.

$$C_{\text{inside}} = \{v \in V(G) \mid x \prec_G v \prec_G y\}$$
$$C_{\text{outside}} = C \setminus C_{\text{inside}}$$

The claim is that either C_{inside} is empty, or C_{outside} is empty. To show this, let $u \in C_{\text{inside}}$ be a vertex between x and y, and $v \in C_{\text{outside}}$. There are two cases: If $v \prec_G x$, then the sequence v, x, u, y is an occurrence of the pattern P. Specifically, due to $u, v \in C$, the edges ux, uy, vx, vy must exist, so the sequence does form an occurrence of the pattern. Otherwise, if $v \not\prec_G x$, then $y \prec_G v$ must be true, due to $y \in C_{\text{outside}}$. In this case, the sequence x, u, y, v is an occurrence of the pattern. Therefore, for any two vertices $x, y \in V(G)$, we can conclude that the vertices of their common neighborhood $N(x) \cap N(y)$ are either all between x and y, or none of them are.

3.1.3 Graph class relations

In this section, we show inclusion relations between the graph class $\mathcal{G}_P$ for some patterns P to various other graph classes. This continues the research on known inclusion relations, and is later going to help us with filtering in Section 4.3.1.

Before we come to the first type of relation, we show the insight following from the definitions of patterns. When there is an undecided pair in the pattern P, and a graph G with some ordering contains P, then the vertices of that undecided pair in G are either connected by an edge, or they are not. Therefore, if we find an occurrence of the pattern P, then we also either found an occurrence of the pattern where the undecided pair is instead a mandatory edge, or we found an occurrence where the undecided pair is instead a forbidden edge. This idea is also already shown in [FH21b].

Lemma 3.4: *Let $P = (V(P), E_M(P) \cup E_F(P))$ be a pattern ordered with $\prec_P$, and $e \notin E(P)$ be an undecided pair. Let $E_M(P_1) = E_M(P) \cup \{e\}$ and $E_F(P_2) = E_F(P) \cup \{e\}$. There is an ordering of G avoiding P if and only if there is an ordering of G avoiding both $P_1 = (V(P), E_M(P_1) \cup E_F(P))$ and $P_2 = (V(P), E_M(P) \cup E_F(P_2))$, where both of these patterns are ordered with $\prec_P$.*

Proof. Let G be a graph ordered with $\prec_G$. We show that there is an occurrence of P if and only if there is an occurrence of either P_1 or P_2. Let $v_1, v_2, \ldots, v_k$ be an occurrence of P. Let $v_i v_j$ be the endpoints of the undecided pair e from the statement of the lemma. If $v_i v_j \in E(G)$, the same vertices are an occurrence of P_1. Otherwise, we have $v_i v_j \notin E(G)$, and the same vertices are an occurrence of P_2. On the other hand, assume that $\prec_G$ does not contain P. Then there is no subsequence of vertices which match up with all mandatory and forbidden edges of P, and therefore also not of P_1 and P_2, due to both of these patterns having all mandatory and forbidden edges of P.

With this, we can show the statement of the lemma. If G has an order $\prec_G$ avoiding P, then $\prec_G$ also avoids both P_1 and P_2. On the other hand, if every ordering of G contains P, then each order must also contain either P_1 or P_2. ∎

The important conclusion we can draw from Lemma 3.4 is that adding a mandatory or forbidden edge to a pattern P never removes a graph from the corresponding graph class $\mathcal{G}_P$. Informally, we can reason that adding more constraints to P makes it harder to find in an ordered graph, and so it is easier to find an ordering avoiding P. Formally, we state the following lemma:

Lemma 3.5: *Let $P = (V(P), E_M(P) \cup E_F(P))$ be some pattern, and $P' = (V(P), E_M(P') \cup E_F(P'))$ a pattern with more mandatory and forbidden edges, but the same set of vertices and same order $\prec_P$. That is, we require $E_M(P) \subseteq E_M(P')$ and $E_F(P) \subseteq E_F(P')$. Then $\mathcal{G}_P \subseteq \mathcal{G}_{P'}$.*

Proof. Let P' the pattern we get from adding a single edge uv to $E_M(P)$ or $E_F(P)$. We only show $G_P \subseteq G_{P'}$ for this case. The more general statement given in Lemma 3.5 follows by applying the single edge addition multiple times.

Let $G \in \mathcal{G}_P$ be an arbitrary graph that can ordered to avoid P. Let $\prec_G$ be an ordering on G that avoids P. From Lemma 3.4, we know that this ordering $\prec_G$ avoids both the pattern with the edge uv added to the mandatory edges, and with the same edge uv added to the forbidden edges. Knowing that the ordering $\prec_G$ avoids both these patterns together in particular also means that it avoids any one of them alone. These two patterns are exactly the two ways the pattern P' can look like, and so, we know that $G \in \mathcal{G}_{P'}$. ∎

Lemma 3.5 gives a notion of weaker and stronger patterns similar to what we have already seen for clauses. Let P and P' be patterns with the same vertices and order, and with P having a strict subset of mandatory and forbidden edges of P':

$$E_M(P) \subset E_M(P')$$
$$E_F(P) \subset E_F(P')$$

In this case, we say that the pattern P is *stronger* and P' is *weaker*. With this terminology, Lemma 3.5 states that stronger patterns have smaller graph classes. We have already seen an example of this: The pattern for interval graphs is a stronger version of the pattern for chordal graphs, and so, every interval graph must be a chordal graph. The difference between the patterns for interval and chordal graphs is highlighted in Figure 3.18.

There are two other relations between patterns we can establish. The first one is based on the pattern for the Bandwidth Minimization problem. These are the patterns that only contain the mandatory edge between the first and last edge, and an example for bandwidth 3 is shown in Figure 3.9 where this problem is introduced. We note in Lemma 3.6 that the pattern for higher bandwidths are weaker than the ones for lower bandwidths.

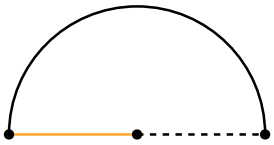

Figure 3.18: The difference between the pattern for chordal and interval graphs: If the mandatory edge highlighted in orange is present, the pattern represents the class of chordal graphs. Otherwise, the pattern represents the class of interval graphs.

Lemma 3.6: *Let P_k be the pattern with k vertices, and only one mandatory edge $E_M(P_k) = \{p_1p_k\}$. If $i < j$, then $\mathcal{G}_{P_i} \subseteq \mathcal{G}_{P_j}$.*

Proof. Let $G \in \mathcal{G}_{P_i}$ be a graph, and $\prec_G$ be an ordering of G avoiding P_i. Then $\prec_G$ is also an ordering avoiding P_j. To see this, assume there is an occurrence $v_1, v_2, \ldots, v_j$ of the pattern P_j. Then $v_1, v_2, \ldots, v_{i-1}, v_j$ would be an occurrence of the pattern P_i, which cannot be the case due to $\prec_G$ avoiding P_i. With this, it follows that $G \in \mathcal{G}_{P_i} \implies G \in \mathcal{G}_{P_j}$. ∎

Second, we note that the uncovered pattern chaining we talk about in Lemma 3.1 results in a weaker pattern than both its operands.

Lemma 3.7: *Let P_1 and P_2 be two patterns, and P the pattern resulting from unifying the last vertex of P_1 and the first vertex of P_2. Then $\mathcal{G}_P \subseteq \mathcal{G}_{P_1}$ and $\mathcal{G}_P \subseteq \mathcal{G}_{P_2}$.*

Proof. We only show $\mathcal{G}_P \subseteq \mathcal{G}_{P_1}$, the proof for $\mathcal{G}_P \subseteq \mathcal{G}_{P_2}$ is similar. From Lemma 3.1, we know that it is possible to order G avoiding P if the vertices can be partitioned into V_1 and V_2 such that $G[V_1]$ can be ordered to avoid P_1, and $G[V_2]$ can be ordered to avoid P_2. If $G \in \mathcal{G}_{P_1}$, then G can be ordered to avoid P_1. Therefore, with the partitioning of the vertices into $V_1 = V$ and $V_2 = \emptyset$, it is also possible to order G to avoid P. Therefore, we have $G \in \mathcal{G}_{P_1} \implies G \in \mathcal{G}_P$. ∎

This concludes the relations between graph classes where both classes are defined by a pattern. Next, we come to a local graph property. We call a property local if, for every vertex $v \in V(G)$, its neighborhood $N(v)$ has this property. An example of this is 2-colorability in a cycle of length 5: It is not possible to color a cycle of length 5 with two colors, but it is possible for the neighbors of every vertex. We show that there are certain patterns P where the corresponding graph class $\mathcal{G}_\mathcal{P}$ also has some local colorability.

Lemma 3.8: *Let P be a pattern with some number of vertices k, and the edges $E_M(P) = \{p_ip_{i+1} \mid i \in [k-1]\} \cup \{p_1p_k\}$ and $E_F(P) = \emptyset$. Let G be a graph. If G can be ordered to avoid P, then G is locally $(k-2)$-colorable.*

Proof. Let $x \in V(G)$ be an arbitrary vertex. To show the lemma, we take an order $\prec_G$ of G avoiding P and from it construct an order $\prec_{N(x)}$ avoiding the pattern P' that shows $(k-2)$-colorability. This pattern P' has $k-1$ vertices and the mandatory edges $\{p_ip_{i+1} \mid i \in [k-2]\}$. We construct the order $\prec_{N(x)}$ of $N(x)$ avoiding P' by first placing all vertices after x in the order, then all vertices before x. Pairs of vertices both to the left or both to the right of x maintain their respective ordering. The concrete definition for this order is:

$$u \prec_{N(x)} v = \begin{cases} u \prec_G v & u \prec_G x \wedge v \prec_G x \\ u \prec_G v & u \succ_G x \wedge v \succ_G x \\ \\ \bot & u \prec_G x \prec_G v \\ \top & v \prec_G x \prec_G u \end{cases}$$

Assume there exists a path of vertices $v_1, v_2, \ldots, v_{k-1}$ from left to right when ordered with $\prec_{N(x)}$, that is, we have $v_i \prec_{N(x)} v_{i+1}$ for every $i \in [k-2]$. In other words, we are assuming that we can find the pattern P' that $(k-2)$-colorable graphs can avoid in this order $\prec_{N(x)}$. We define a new sequence S of vertices, which rotates the sequence $v_1, v_2, \ldots, v_{k-1}$ to a valid sequence under $\prec_G$, and then insert x into it at the correct position. For the formal definition, let $t \in [k-1]$ be such that v_t is the last vertex of this path when instead ordered by $\prec_G$. The sequence S is defined as:

$$S = \begin{cases} v_{t+1}, v_{t+2}, \ldots, v_{k-1}, x, v_1, v_2, \ldots, v_t & t \neq k-1 \\ v_1, v_2, \ldots, v_{k-1}, x & t = k-1 \wedge v_{k-1} \prec_G x \\ x, v_1, v_2, \ldots, v_{k-1} & t = k-1 \wedge x \prec_G v_{k-1} \end{cases}$$

Then the sequence S is an occurrence of the pattern P in G when ordered with $\prec_G$, contradicting the requirement of $\prec_G$ to avoid the pattern P. To see that this sequence is an occurrence of the pattern P in G when ordered with $\prec_G$, we note that the sequence of v_i form a path of length $k-1$. Further, the vertex x is connected to all vertices v_i, so in particular to v_1 and v_{k-1}, closing the cycle of length k. This means that the vertices of S are an occurrence of the pattern P, a contradiction to $\prec_G$ avoiding P. Therefore, the assumption that the sequence $v_1, v_2, \ldots, v_{k-1}$ forming a path exists must be false, the order $\prec_{N(x)}$ does avoid the pattern for $(k-2)$-colorable graphs, and the graph G must be locally $(k-2)$-colorable. ∎

Lemma 3.8 is somewhat of an extension to Lemma 3.5. The main insight from 3.5 is that adding edges to a pattern results in a weaker pattern. We could, for example, interpret Lemma 3.8 with $k = 4$ as taking the four patterns shown in Figure 3.19, each of them weaker than the original pattern, and trying to avoid all of them.

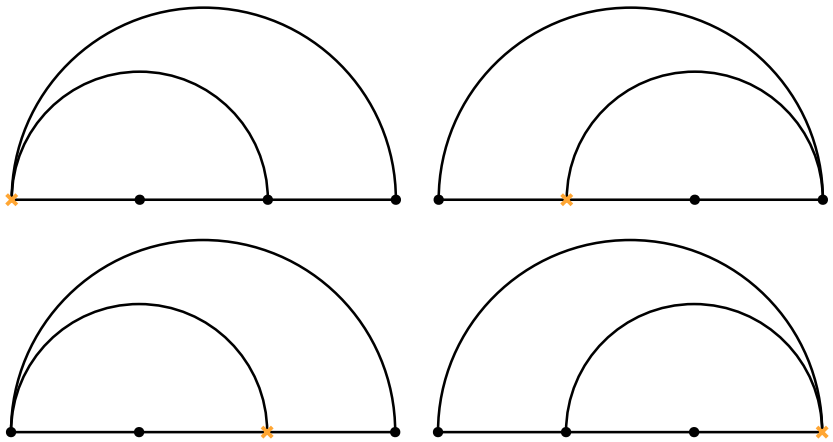

Figure 3.19: The patterns that are avoided when using Lemma 3.8 with $k = 4$. The vertex taking the role of x is marked in orange. In the case of $k = 4$, the actual number of patterns avoided by this method is only two, due to the left (and respectively right) two patterns looking the same, even with different vertices for x.

Next, we come to a combinatorial argument showing that dense graphs cannot be ordered to avoid the pattern P with $k = 4$ and $E_M(P) = \{p_1p_3, p_1p_4, p_2p_3, p_2p_4\}$. In Section 3.1.2, we have seen this pattern P visualized in Figure 3.16, where we have also seen an alternative interpretation of this pattern as points in a grid. We now use this interpretation in Lemma 3.9 to give an upper bound on the density of graphs that can be ordered to avoid P.

Lemma 3.9: *Let P be the pattern with $k = 4$ and $E_M(P) = \{p_1p_3, p_1p_4, p_2p_3, p_2p_4\}$. Let G be a graph. If $m > n\sqrt{(n-1)/2}$, then G cannot be ordered to avoid P.*

Proof. In Section 3.1.2, we noted that the problem of detecting P in an ordered graph can be reduced to finding an axis-aligned rectangle in a set of points. The point set obtained from this reduction has n rows and n columns, and m points. If the pattern is not present, then there are no two rows which contain a pair of points in the same positions, otherwise, these two pairs form the rectangle. Let X be the multiset of pairs of positions with points in any row, that is, for each row that has a point at position x and at position y with $x < y$, the multiset X contains one (x, y). We note that the number of points in row i is exactly the number of right neighbors $\deg_{\text{right}}(\pi^{-1}(i))$ of the vertex $\pi^{-1}(i)$. With that, we can estimate the number of pairs of positions $|X|$ that have to appear:

$$\begin{aligned}
|X| &= \sum_{v \in V(G)} (\deg_{\text{right}}(v))^2 \\
&= \left(\sum_{v \in V(G)} (\deg_{\text{right}}(v))^2 \right) \cdot \left(\sum_{v \in V(G)} 1^2 \right) \cdot \frac{1}{n} \\
&\geq \left(\sum_{v \in V(G)} \deg_{\text{right}}(v) \right)^2 \cdot \frac{1}{n} \\
&= m^2 \cdot \frac{1}{n}
\end{aligned}$$

The third step uses the Cauchy-Schwarz inequality. This shows that there are $|X| \geq m^2/n$ pairs of point positions. On the other hand, there are only $n(n-1)/2$ possible distinct pairs of positions. By the pigeonhole principle, we know that if $m^2/n > n(n-1)/2$, then there must be a pair of positions appearing twice in X and the pattern is guaranteed to be present. Therefore, the pattern P is present in a graph G, no matter the order, if the following inequality holds:

$$\begin{aligned}
& m^2/n > n(n-1)/2 \\
\implies & m^2 > n^2(n-1)/2 \\
\implies & m > n\sqrt{(n-1)/2}
\end{aligned}$$

∎

Last, we give a randomized argument that is somewhat of a counterpart to the previous one: In Lemma 3.10, we show that graphs with low maximum degree can always avoid certain patterns. For this, we first introduce the Lovász Local Lemma [Sze13]. The statement of this lemma is as follows: We choose an object randomly from a set of possible objects. Let B be a set of events, where a single event $\mathcal{E} \in B$ represents the chosen object having some bad property. Every event must happen with probability at most p and every event must depend on at most d other events. The Lovász Local Lemma states that, if $ep(d+1) \leq 1$, then there is at least one possible choice of object which has no bad property. In our case, we can use this by setting the orders of G as the objects we make a random choice from, and define the bad events to be an occurrence of the pattern P. With this, we come to the concrete lemma to show:

Lemma 3.10: *Let G be a graph with maximum degree Δ and P a pattern with a spanning tree in the mandatory edges $E_M(P)$. It is always possible to order G to avoid P if the following inequality holds:*

$$e \cdot \frac{1}{k!} \cdot (|Aut(P)| \cdot k \cdot \Delta(\Delta-1)^{k-2} + 1) \leq 1$$

Here, $Aut(P)$ is the set of automorphism of P, that is, the set of isomorphisms from P to itself.

Proof. We show this using the Lovász Local Lemma. We randomly choose an order for the vertices of G and define the set of bad events to be an occurrence of the pattern P in the order. Let $v_1, v_2, \ldots, v_k$ be a sequence of vertices that are an occurrence of the pattern if ordered $v_1 \prec v_2 \prec \cdots \prec v_k$. We define $\mathcal{E}_{v_1,v_2,\ldots,v_k}$ as the event that the vertices $v_1, v_2, \ldots, v_k$ are ordered in this order and with that form an occurrence of the pattern. For the probability p of any event $\mathcal{E}$ happening, we have

$$p = \frac{1}{k!}$$

This value for p follows from the fact that there are $k!$ equally likely ways the k vertices in V' can be ordered.

Next, we show a bound on d. Let X and Y be two sequences of k vertices that form an occurrence of the pattern in G. First, we argue that $\mathcal{E}_X$ and $\mathcal{E}_Y$ can only depend on each other if $X \cap Y \neq \emptyset$. The events $\mathcal{E}_X$ and $\mathcal{E}_Y$ being independent means $\mathbb{P}[\mathcal{E}_X \mid \mathcal{E}_Y] = p$.

Assume that $X \cap Y = \emptyset$. We already know the probability $\mathbb{P}[\mathcal{E}_X]$ to be p. To find $\mathbb{P}[\mathcal{E}_X \cap \mathcal{E}_Y]$, we show a random way to construct a permutation containing both the sequence X and Y. This happens in three steps: Randomly choose the k out of the n positions where the sequence X is, then randomly choose the k positions out of the remaining $n-k$ positions where the sequence Y is, and then randomly permute the remaining $n-2k$ elements. Here, we used $X \cap Y = \emptyset$ in the second step: After choosing the k positions for X, none of the k positions for Y are already chosen. This process of building a permutation gives the following chance for $\mathcal{E}_X \cap \mathcal{E}_Y$ happening:

$$\begin{aligned}
\mathbb{P}[\mathcal{E}_X \cap \mathcal{E}_Y] &= \frac{\binom{n}{k} \cdot \binom{n-k}{k} \cdot (n-2k)!}{n!} \\
&= \frac{\frac{n!}{k!(n-k)!} \cdot \frac{(n-k)!}{k!(n-2k)!} \cdot (n-2k)!}{n!} \\
&= \frac{1}{(k!)^2}
\end{aligned}$$

From this, we get the probability of $\mathbb{P}[\mathcal{E}_X \mid \mathcal{E}_Y]$:

$$\begin{aligned}
\mathbb{P}[\mathcal{E}_X \mid \mathcal{E}_Y] &= \frac{\mathbb{P}[\mathcal{E}_X \cap \mathcal{E}_Y]}{\mathbb{P}[\mathcal{E}_Y]} \\
&= \frac{1}{(k!)^2} / \frac{1}{k!} \\
&= \frac{1}{k!} \\
&= p
\end{aligned}$$

Therefore, for two sequences X and Y sharing no vertex, the events $\mathcal{E}_X$ and $\mathcal{E}_Y$ are independent.

We are now looking to upper bound the number of events that are not independent of $\mathcal{E}_X$ for any event X. The number of these events is the number of sequences Y that form an occurrence of the pattern and share a vertex with X. Due to the pattern P containing a spanning tree in the mandatory edges, we know that for any sequence Y, the vertices of Y also have this spanning tree in G. In the other direction, for each occurrence of the spanning tree in G, there are at most $|Aut(P)|$ sequences on these vertices forming an occurrence of the pattern.

Now, let $\mathcal{E}_X$ be an arbitrary event. We build the spanning tree vertex by vertex: First, choose one of the k vertices in X. Then, build the tree by repeatedly choosing a neighbor of a vertex already in the tree, with Δ options in the first step and $\Delta - 1$ options in all following steps. Last, we need to determine the order of these vertices. We map the vertices of the tree just constructed to the spanning tree of P, and then note that we can order it in at most $|Aut(P)|$ ways that maintain the pattern. This bounds d as follows:

$$d \leq |Aut(P)| \cdot k \cdot \Delta(\Delta - 1)^{k-2}$$

From here, we substitute p and d with the bounds we have found. Then, making use of the inequality required in the statement of Lemma 3.10, we can show that the requirement of $ep(d+1) \leq 1$ of the Lovász Local Lemma is fulfilled.

$$\begin{aligned} ep(d+1) &\leq e \cdot \frac{1}{k!} \cdot (|Aut(P)| \cdot k \cdot \Delta(\Delta-1)^{k-2} + 1) \\ &\leq 1 \end{aligned}$$

■

With the bound of Lemma 3.10 established, we would like to know the minimum value of Δ depending on a given k. Figure 3.20 shows a plot of $e \cdot \frac{1}{k!} \cdot (k \cdot \Delta(\Delta-1)^{k-2} + 1) = 1$, giving the maximum value Δ can take for a given k when $|Aut(P) = 1|$. Table 3.1 shows the inverse direction: For each Δ up to 9, the minimum value of k is given such that graphs with minimum degree Δ can always be ordered to avoid a pattern with k vertices and a spanning tree in the mandatory edges.

Δ	1	2	3	4	5	6	7	8	9
k	3	5	7	10	12	15	17	20	23

Table 3.1: For each maximum degree Δ from 1 to 9, the minimum pattern size k such that $e \cdot \frac{1}{k!} \cdot (k \cdot \Delta(\Delta-1)^{k-2} + 1) = 1$ holds. The full plot is shown in Figure 3.20.

While the specific bound of Lemma 3.10 might not be practical, it gives a general strategy to find graph class relations. Specifically, to show $\mathcal{G}_P \subseteq \mathcal{G}'$ for some pattern P, we need to show that the number of possible pattern sequences including some fixed vertex v in G' is small enough. From there, we can adapt the argument of Lemma 3.10.

3.2 Structure of pattern-avoiding graph classes

In Section 3.1.2, we focused on the interpretations of patterns, and the relation between them. This section instead focuses on the graph class $\mathcal{G}_P$ for a fixed pattern P. First, in Section 3.2.1, we investigate the structure of $\mathcal{G}_P$ and show that $\mathcal{G}_P$ can never contain all graphs. Second, Section 3.2.2 gives an overview over the hardness of solving the Pattern-Avoiding Vertex Ordering problem, or equivalently, solving the recognition problem for $\mathcal{G}_P$.

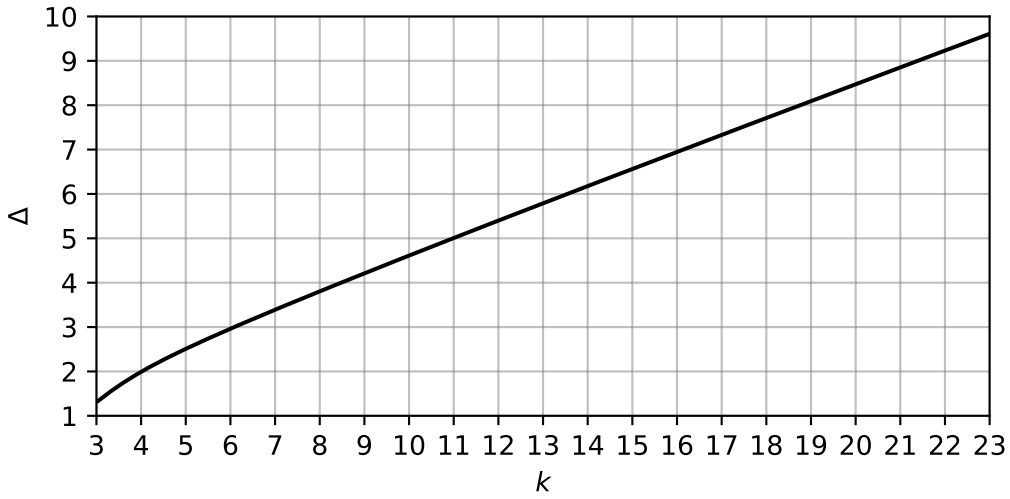

Figure 3.20: Plot of $e \cdot \frac{1}{k!} \cdot (k \cdot \Delta(\Delta - 1)^{k-2} + 1) = 1$. This is the upper bound on the maximum degree Δ for the graph G by the pattern size k so that the condition of Lemma 3.10 is fulfilled when $|Aut(P)| = 1$. Note that k starts at 3 and Δ starts at 1.

3.2.1 Obstructions

A graph class $\mathcal{G}$ is called *hereditary* if, for $G \in \mathcal{G}$ and G' is an induced subgraph of G, then $G' \in \mathcal{G}$. In Lemma 3.11, we show that the graph class $\mathcal{G}_P$ is hereditary for every pattern P. This is already known, see for example [FH21b].

Lemma 3.11: *For every pattern P, the graph class $\mathcal{G}_P$ is hereditary.*

Proof. We want to show that if $G \in \mathcal{G}_P$ and G' is an induced subgraph of G, then $G' \in \mathcal{G}$. Because $G \in \mathcal{G}_P$, we know that there is an ordering $\prec_G$ avoiding P. We define $\prec_{G'}$ as $\prec_G$ limited to the vertices of G'. Assume $(G', \prec_{G'})$ contains P, and let $v_1, v_2, \ldots, v_k$ be the subsequence of vertices which form an occurrence of the pattern P. Then the same set of vertices $v_1, v_2, \ldots, v_k$ are an occurrence the pattern in G, due to all vertices in G' also existing in G and the fact that $v_i \prec_{G'} v_{i+1}$ implies $v_i \prec_G v_{i+1}$. Therefore, the assumption that $(G', \prec_{G'})$ contains P must be false and G' is part of $\mathcal{G}_P$. ∎

A consequence of Lemma 3.11 is that is not possible enforce graph properties that are not hereditary using a pattern. These properties include, for example, that the graph is connected. This has implications for the kind of ordering problems that can be solved using Pattern-Avoiding Vertex Ordering problem: For example, the Hamilton Cycle problem cannot be represented by any forbidden pattern.

With the insight of Lemma 3.11, we now focus on *obstructions*. A graph G is an obstruction for a hereditary graph class $\mathcal{G}$ if G is not part of $\mathcal{G}$, but the graph $G - v$ after removing any vertex $v \in V(G)$ is in $\mathcal{G}$. Alternatively, obstructions are the graphs that may not appear as an induced subgraph in any graph of $\mathcal{G}$. Next, in Theorem 3.12, we show that every graph class $\mathcal{G}_P$ based on some pattern P has at least one obstruction.

Theorem 3.12: *For every ordered pattern P, there is at least one graph G such that for every ordering $\prec_G$ of $V(G)$, the ordered graph $(G, \prec_G)$ contains P.*

Proof. To show Theorem 3.12, we construct a graph G for a fixed pattern P such that for every ordering of the vertices of G, we can find an occurrence of P. From Lemma 3.5, we know that if the pattern contains undecided pairs, we can replace them arbitrarily by mandatory or forbidden edges and get a stronger pattern. Hence, from now on, we assume that the pattern contains no undecided pairs.

Let $P[1..\ell]$ be the graph induced by the first ℓ vertices in the ordering of P. We show the claim via induction over the prefixes of P, that is, by induction over ℓ. For the base case, we have $P[1..1]$, which is always the pattern with a single vertex. This forbids every graph except the one with no vertices. We choose the graph with one vertex as our construction in this case.

Now, assume that we have a graph G_ℓ such that for any ordering of its vertices, we can find an occurrence of $P[1..\ell]$. In the induction step, we construct $G_{\ell+1}$. The rough plan for the remainder of the construction is as follows: We take multiple copies of G_ℓ in $G_{\ell+1}$ connected in such a way that the occurrence of $P[1..\ell]$ in one of these copies can be extended to an occurrence of $P[1..\ell+1]$. To show that this extension is always possible, we consider an arbitrary ordering $\prec_{G_\ell}$ of this graph. We then order the copies of G_ℓ by the position of their respective last vertex in the ordering $\prec_G$ and look at the first copy C_1 of G_ℓ in this order. There is at least one vertex of each of the other copies after all vertices of C_1. We connect the copies of G_ℓ in such a way that, no matter what vertices of C_1 are the occurrence of the pattern $P[1..\ell]$ and no matter in what order, there is another copy C' of G_ℓ such that every vertex of C' can extend the $P[1..\ell]$ in C to $P[1..\ell+1]$. Then, because one vertex of C' is behind all vertices of C_1, this extension to $P[1..\ell+1]$ appears. An example of this step from G_ℓ to $G_{\ell+1}$ is shown in Figure 3.21. The exact way to connect these copies of G_ℓ comes below in the formal description.

Before we formally build $G_{\ell+1}$, we need some mapping from subsets of a set S with a fixed size t to the first natural numbers. More specifically, we need some injective function $f_{S,t}$ with the following signature:

$$f_{S,t} : \binom{S}{t} \to [\binom{|S|}{t}]$$

The exact choice of this function does not matter. One possible option is to define some order $<$ on S, and map each set in $\binom{S}{t}$ to its position in lexicographic the order.

Further, there are three values we define for convenience. First is $d_{\ell+1}$, the degree of vertex $\ell+1$ in $P[1..\ell+1]$. Second is n_p, the number of ways in which we can select $d_{\ell+1}$ vertices from G_ℓ. Third is n_c, which is the number of copies of G_ℓ are going to make up $G_{\ell+1}$.

$$\begin{aligned} d_{\ell+1} &= \deg_{P[1..\ell+1]}(v_{\ell+1}) \\ n_p &= \binom{n_\ell}{d_{\ell+1}} \\ n_c &= 2n_p + 1 \end{aligned}$$

Now, we formally define the vertices $V(G_{\ell+1})$ and edges $E(G_{\ell+1})$. The edges are in defined in two parts: The edges $E_1(G_{\ell+1})$ within the copies of G_ℓ and the edges $E_2(G_{\ell+1})$ between copies. While the definitions of $V(G_{\ell+1})$ and $E_1(G_{\ell+1})$ are straightforward, the second set of edges $E_2(G_{\ell+1})$ needs a bit more explanation. The meaning of $E_2(G_{\ell+1})$ is that for each copy C of G_ℓ and each subset of vertices $X \subseteq V(C)$ with exactly $d_{\ell+1}$ vertices, we take another copy

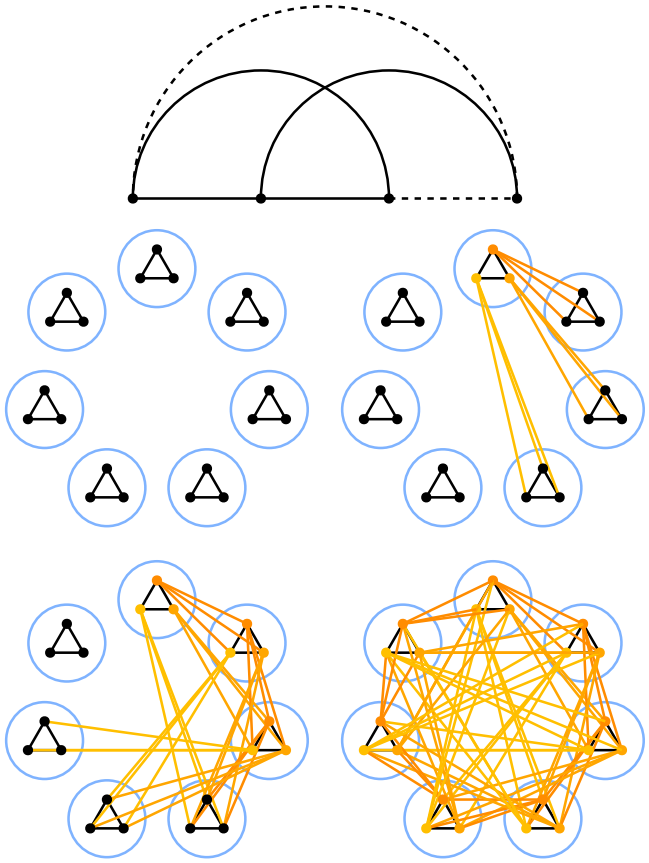

Figure 3.21: An example of the construction in the induction step of the proof in Theorem 3.12. At the top is the pattern P. The bottom shows the construction of the last induction step for this pattern, with the obstruction for the first three vertices being a triangle. This is not the obstruction constructed when following the steps of Theorem 3.12, but is chosen for more clarity in the last step, which is described in the following. The top left shows just the seven copies of the made in this induction step. The top right shows how one copy is connected to the following. The bottom left shows how this pattern is continued for the next copies. The bottom right shows the full construction.

C' and connect all vertices of X with all vertices of C'. We are also making sure that for each combination of two copies C and C' of G_ℓ, we only choose it for at most one such set X in total.

$$V(G_{\ell+1}) = V(G_\ell) \times [n_c]$$
$$E_1(G_{\ell+1}) = \{(u,i)(v,i) \mid u,v \in V(G_\ell), i \in [n_c]\}$$
$$E_2(G_{\ell+1}) = \{(u,i)(v,j) \mid u \in X, v \in V(G_\ell), X \in \binom{V(G_\ell)}{d_{\ell+1}}, j = (i + f_{V(G_\ell),n_c}(X)) \bmod n_c\}$$
$$E(G_{\ell+1}) = E_1(G_{\ell+1}) \cup E_2(G_{\ell+1})$$

For a vertex $(v, i) \in V(G_{\ell+1})$, we refer to the first component v as the original vertex and to the second component i as the index of the vertex. For the subgraph induced by vertices with the same index i in $G_{\ell+1}$, we use the notation $G_{\ell+1}[*, i]$. We note that these are exactly the copies of G_ℓ due to $\{(u, i)(v, i) \mid u, v \in V(G_\ell)\} \subseteq E(G_{\ell+1})$, so we know that $G_{\ell+1}[*, i] \cong G_\ell$.

The claim to prove is that, given that G_ℓ contains $P[1..\ell]$, no matter how G_ℓ is ordered, then $G_{\ell+1}$ contains $P[1..\ell+1]$, no matter how $G_{\ell+1}$ is ordered. To show this, we fix an arbitrary ordering π on the vertices $V(G_{\ell+1})$. We define the last position $p(i)$ of the subset of vertices in $V(G_\ell)$ where the index is i as the maximum position in π as the maximum such position:

$$p(i) = \max\{\pi(v) \mid v \in V(G_{l+1}[*, i])\}$$

Further, we choose i_0 minimizing the value p among all choices out of $[n_c]$:

$$\forall i \in [n_c] : p(i_0) \leq p(i)$$

This means that the last vertex with index being i_0 appears before the last vertex of any other index. Because $G_{\ell+1}[*, i_0] \cong G_\ell$, we know that it contains $P[1..\ell]$, no matter what the order π is, by the induction hypothesis.

Let φ be a specific mapping from the vertices of $P[1..\ell]$ to the vertices of $G_{\ell+1}$ showing an ordered occurrence of $P[1..\ell]$ in $G_{\ell+1}[*, i_0]$ when ordered with π. We further define T as the vertices in the neighborhood of $v_{\ell+1} \in P[1..\ell+1]$ in G_ℓ:

$$T = \varphi(N_{P[1..\ell+1]}(v_{\ell+1}))$$

We know that $|T| = d_{\ell+1}$ and that all vertices of T are contained within one copy of G_ℓ, namely in $G_{\ell+1}[*, i_0]$. This means that there exists another copy G_ℓ, specifically the one with index $j = (i_0 + f_{V(G_\ell),n_c}(T)) \bmod n_c$, so that out of all vertices in $G_{\ell+1}[*, i_0]$, exactly the vertices of T are connected to all vertices of $G_{\ell+1}[*, j]$. The last vertex of $G_{\ell+1}[*, j]$ now extends the occurrence of $P[1..\ell]$ in $G[*, i_0]$ to an occurrence of $P[1..\ell+1]$.

To finalize this proof, we note that this last vertex of $G_{\ell+1}[*, j]$ does in fact come after all vertices of the pattern occurrence $P[1..\ell]$ in $G_{\ell+1}[*, i_0]$:

$$\begin{aligned} \max\{\pi(v) \mid v \in \varphi(P[1..\ell])\} &\leq p(i_0) \\ &< p(j) \end{aligned}$$

The first inequality holds because all vertices in the occurrence of the pattern $P[1..\ell]$ in $G_{\ell+1}[*, i_0]$ cannot be after the last vertex of $G_{\ell+1}[*, i_0]$. The second inequality comes from the definition of i_0. ∎

From Theorem 3.12, we can also conclude that there is an infinite number of graphs that are not part of $\mathcal{G}_P$ for any pattern P: As an example, we have all graphs which contain the constructed obstruction as an induced subgraph. A more interesting question is for the number of obstructions the graph class $\mathcal{G}_P$ has. Here, we cannot give a complete answer, but two examples. If the pattern P has k vertices, but no edges, then the set of obstructions for $\mathcal{G}_P$ is the set of graphs with exactly k vertices. On the other hand, for the class of forests, the obstructions are all graphs which form a cycle, of which there is an infinite amount.

3.2.2 Theoretical complexity

In this section, we consider the theoretic complexity of the Pattern-Avoiding Vertex Ordering problem. This is motivated by two things: First, to give an overview over the theoretical complexity large number of different graph classes and properties we have seen represented in $\mathcal{G}_P$ for different patterns P. Second, to investigate the complexity of Pattern-Avoiding Vertex Ordering relative to the SAT problem. We are going to make use of various concepts in complexity theory in this section, see [AB09] for an overview.

We first consider the effect of fixing the pattern P on the complexity of Pattern-Avoiding Vertex Ordering. For a fixed pattern P, the problem of finding an order which avoids the pattern P is in NP because we can verify an ordering in the time it takes to detect an ordered pattern, which is possible in $\mathcal{O}(n^{f(P)})$ time, with the algorithm from [DFHP23]. For a positive outerplanar pattern, we always have $f(P) = \omega$, also shown in [DFHP23]. However, for general patterns, the function $f(P)$ can become arbitrarily large: The case where the pattern is a complete graph is exactly the problem of finding a clique of size k, which is conjectured to be impossible to solve in $\mathcal{O}(n^{(\omega k)/3-\varepsilon})$ time for any $\varepsilon > 0$, see for example [Wil18]. Still, if P is fixed, then the value of $f(P)$ is constant and the Pattern-Avoiding Vertex Ordering is in NP.

Next, we consider the case when the pattern P is not fixed and given in the input. We note that Pattern-Avoiding Vertex Ordering is both NP-hard and coNP-hard. With the precise reasoning presented in the proof of Theorem 3.13, we only note here that both the 3-Coloring and coClique problem can represented by using an appropriate pattern.

With the hardness of Pattern-Avoiding Vertex Ordering given, we focus on an upper bound on the complexity. Here, we show that Pattern-Avoiding Vertex Ordering is in Σ_2^{P}, a complexity class in the polynomial hierarchy containing both NP and coNP. We give the definition Σ_2^{P} from [AB09] here. A problem L is in Σ_2^{P} if and only if there is a property $R : (x, y_1, y_2) \to \mathbb{B}$ checkable in polynomial time and a polynomial p such that the following holds:

$$x \in L \iff \exists y_1 \in \{0,1\}^{p(|x|)} \forall y_2 \in \{0,1\}^{p(|x|)} : R(x, y_1, y_2) \tag{3.1}$$

With the definition of Σ_2^{P} established, we show that the Pattern-Avoiding Vertex Ordering problem is in Σ_2^{P} in Theorem 3.13. The proof of this theorem is already quite apparent when reformulating the definition of the Pattern-Avoiding Vertex Ordering problem a bit: We have a satisfiable instance if and only if there exists an ordering of the vertices such that for all subsets of vertices, the subset having size k implies that its vertices do not form an occurrence of the pattern.

Theorem 3.13: *The Pattern-Avoiding Vertex Ordering problem with P given in the input is NP-hard, coNP-hard, and in Σ_2^P.*

Proof. NP-hardness follows via reduction from 3-Coloring: Let G be a graph to be colored with three colors. Let P be the pattern with $k = 4$ vertices and the edges $E_M(P) = \{p_1p_2, p_2p_3, p_3p_4\}$. It is known that the graphs that can be ordered to avoid P are exactly the graphs the 3-colorable, see the explanation given for this example at Figure 3.10.

Next, coNP-hardness follows via reduction from coClique: Let G be a graph, and we want to test if there is no clique with k vertices present. Let P be the pattern with $k = 4$ vertices and the edges $E_M(P) = \{p_ip_j \mid i, j \in [n] \wedge i \neq j\}$. If a clique of size k is present, no matter the ordering, the k vertices of the clique match exactly the pattern P. On the other hand, if no clique of size k exists, we can use an arbitrary order for the vertices $V(G)$ to avoid P. Any

occurrence of the pattern requires k pairwise connected vertices, which are never present due to G not containing a clique of size k in this case. This solves the coClique problem and not Clique itself because Pattern-Avoiding Vertex Ordering cannot avoid the pattern if a clique is present, and so gives the opposite answer to an instance than Clique would.

With this, we know that Pattern-Avoiding Vertex Ordering is NP-hard and coNP-hard. To show membership in Σ_2^{P}, we need to show that some $R : (x, y_1, y_2) \to \mathbb{B}$ exists such that the defining condition in Equation 3.1 holds. Here, x is the problem instance consisting of a graph G and a pattern P. We interpret y_1 as an order $\prec_G$ and y_2 as a subset of vertices $S \subseteq V(G)$. We now define the property R as the following: If $|S| = k$, then the vertices of S ordered with $\prec_G$ must not be an occurrence of the pattern P. This gives a condition equivalent to the definition of the Pattern-Avoiding Vertex Ordering problem: If a graph G has an order $\prec_G$ avoiding the pattern P, then no sequence of k vertices is an occurrence of the pattern, and vice versa. ∎

We can also give another lower bound using Theorem 3.13 in Corollary 3.14. This corollary uses the hypothesis in complexity theory that NP ≠ coNP. With this, it follows that Pattern-Avoiding Vertex Ordering is not in NP or coNP.

Corollary 3.14: *Assume NP ≠ coNP. Then Pattern-Avoiding Vertex Ordering is not in NP and not in coNP.*

Proof. For this proof, we use the statement of Theorem 3.13 that Pattern-Avoiding Vertex Ordering is NP-hard and coNP-hard, together with the common conjecture that NP ≠ coNP. From these insights, we prove that Pattern-Avoiding Vertex Ordering is neither in NP nor in coNP. Assume that Pattern-Avoiding Vertex Ordering is in coNP, then we could take any problem X in NP, reduce it to Pattern-Avoiding Vertex Ordering due to it being NP-hard, and solve its complement non-deterministically in polynomial due to Pattern-Avoiding Vertex Ordering being in coNP. With this, we have NP ⊆ coNP. From this, complementing each problem in both classes directly gives coNP ⊆ cocoNP = NP, which would show NP = coNP contradicting the NP ≠ coNP conjecture. With this, our assumption that Pattern-Avoiding Vertex Ordering is in NP must be wrong. The proof for Pattern-Avoiding Vertex Ordering not being in coNP is analogous. ∎

Using a SAT solver to solve a problem that is not in NP seems contradictory, but is in our case explained by the reduction: The size of the SAT instance produced by a reduction has a size in $\mathcal{O}(n^{f(P)})$. This is not a polynomial reduction and with that, the larger size of the instance makes up for the fact that we are reducing a problem not in NP to a problem in NP.

For this last part of the discussion on complexity, we go back to the Pattern-Avoiding Vertex Ordering problem where the pattern P is fixed and not given in the input. Mainly, this comes down to patterns inheriting the complexity of recognizing the graph class they represent. More interesting is the broad spectrum of complexity classes covered by this. For this, we make use of the pattern interpretations shown in Section 3.1.2. Starting with problems in P, we have for example all patterns on three vertices, with this already shown in [FH21b] and [HMR14]. Next, there is a pattern representing the Vertex Cover problem (with an example shown in Figure 3.11), which is a problem in FPT [HN24]. Note that we parameterize Vertex Cover by solution size, while the corresponding parameter for Pattern-Avoiding Vertex Ordering is the size of the pattern. Unless P = NP, there is no polynomial time algorithm for Vertex Cover. After that, we come to the Bandwidth Minimization problem (for example Figure 3.9) and Clique problem, with both of them being in XP. For Clique, this can be seen using a simple algorithm that checks every set of k vertices for being a clique, with there being $\mathcal{O}(n^k)$ such

sets. For Bandwidth Minimization, this is harder to see, with an algorithm given in [Sax80]. Clique is also known to be W[1]-hard, and so there is no FPT algorithm for Clique unless ETH fails [LMS11]. This brings us to the NP-hard problems, where we have 3-Coloring and calculating the queue number. The 3-Coloring problem is known to NP-hard [Kar72], but is at its core more a problem about partitioning vertices than about ordering them. On the other hand, deciding if the queue number is 1 or not is already NP-hard [HR92] and has a pattern where the leftmost and rightmost vertex are connected, which makes it impossible to apply the partitioning idea of Lemma 3.1.

4 Solving Pattern-Avoiding Vertex Ordering with SAT

In this section, we describe the different approaches we take to solve the Pattern-Avoiding Vertex Ordering problem using SAT. We split this problem into two parts: First, we address the way we translate the ordering into SAT. To encode the ordering $\prec$, we define the *ordering variables* $o_{u,v}$ for each $u, v \in V$, where $o_{u,v}$ is assigned true if and only if $u \prec v$. We give some additional notes on ordering variables later in this introductory part. Making sure that the ordering variables form a coherent ordering is the subject of Section 4.1. Second, we address the way we ensure that the ordering does not contain an occurrence of the pattern P. This is discussed in Section 4.2.

There are two more parts to this section on solving the Pattern-Avoiding Vertex Ordering problem. In Section 4.3, we discuss parts of the solving process outside the SAT solver. This includes incremental solving, preprocessing the instances, and trying to filter out easy instances before using the SAT solver. After that, in Section 4.4, we consider the theoretical performance of SAT solving by examining resolution proofs.

For the remainder of the introduction to Chapter 4, we give preliminary insights needed for the approaches presented in Section 4.1 and 4.2. First are the ordering variables $o_{u,v}$. To reduce the number of variables created in the SAT solver, we note that $o_{u,v} = \overline{o}_{v,u}$ and only one of these two needs to be created for each $u, v \in V$. We do this by fixing some order $<$ on V and only creating the variable $o_{u,v}$ when $u < v$. The variable $o_{u,v}$ is assigned to true in the assignment ϕ if and only if u comes before v in the ordering:

$$\phi(o_{u,v}) = \top \iff u \prec v$$

Additionally, we define $o_{u,v} = \overline{o}_{v,u}$ as a shorthand notation. This makes sure that we do not need to correctly order the indices every time we use an ordering variable.

This definition inherently ensures that $\prec$ is irreflexive and antisymmetric. Because $o_{v,v}$ does not exist as a variable, we know that $v \not\prec v$. Antisymmetry follows from $o_{u,v} = \overline{o}_{v,u}$:

$$\begin{aligned} u \prec v &\iff \phi(o_{u,v}) = \top \\ &\iff \phi(\overline{o}_{v,u}) = \top \\ &\iff \neg(\phi(o_{v,u}) = \top) \\ &\iff \neg(v \prec u) \end{aligned}$$

Transitivity, on the other hand, does not directly follow from the definition of the variables. We start the Section 4.1 with approaches ensuring transitivity.

Next, we define three sets of graph for convenience. All of them use the same vertex set as the instance graph G, even if in some cases, these vertices may have no neighbors. First is the *variable graph* G_U, based on the set of variables U of a SAT instance $\mathcal{F}$, where two vertices u and v are connected by an undirected edge if U contains the variable $o_{u,v}$ or $o_{v,u}$. The reason

for this 'or' is that, between the two variables $o_{u,v}$ and $o_{v,u}$, we only ever create one in the SAT instance. This set of graphs is motivated mainly by Lemma 4.4, where we show that we are allowed to remove some ordering variables under certain circumstances.

$$G_U = (V(G), E_U)$$
$$uv \in E_U \iff o_{u,v} \in U \vee o_{v,u} \in U$$

We note that similar graphs over SAT instances are defined in the paper [BV02], a paper investigating transitive symmetric relations. In that paper, the variables $e_{i,j}$ are defined if the elements i and j are in relation, and a similar undirected graph can be defined in that case.

Second is the *clause graph* G_C, based on a clause C, which has a directed edge from u to v if and only if the literal $o_{u,v}$ is in the clause C. Note that by the convention $o_{u,v} = \overline{o}_{v,u}$, if the negated literal $\overline{o}_{v,u}$ appears in the clause, we add the edge uv to the graph.

$$G_C = (V(G), E_C)$$
$$uv \in E_C \iff o_{u,v} \in C$$

A clause C is now satisfied by an ordering $\prec$ on V if and only if there is an edge $uv \in E_C$ such that $u \prec v$.

Third, we define the *assignment graph* G_ϕ based on a (partial) variable assignment ϕ, which has a directed edge from u to v if and only if $o_{u,v}$ is assigned to true in ϕ. If $o_{u,v}$ is assigned false in ϕ, then the edge vu exists. If $o_{u,v}$ is unassigned in ϕ, then neither of the edges uv and vu exist.

$$G_\phi = (V(G), E_\phi)$$
$$uv \in E_\phi \iff \phi(o_{u,v}) = \top$$

This correlation between variable sets, clauses or partial assignments on the one side and graphs on the other also allows us to intuitively use the language of graphs for clauses and partial assignments. For example, we call a clause C a *cycle clause* if the edges of the clause graph G_C form a single cycle, and a *cyclic clause* if it contains at least one cycle. Additionally, we define *linear clauses* to be clauses that are a single directed path. Formally, a linear clause C is of the form $C = \{o_{v_1,v_2}, o_{v_2,v_3}, \dot{;} o_{v_{k-1},v_k}\}$.

Another advantage of these graphs is a simple visualization of different aspects of clauses and assignments. First is the visualization of resolution. When resolving C and D over the variable $o_{u,v}$, we must have the edge uv in one of G_C or G_D, and the edge vu for the negated literal $\overline{o}_{u,v} = o_{v,u}$ in the other. Figure 4.1 shows an example of such a resolution.

Visualizing if a clause C is satisfied by an assignment ϕ is possible in a similar way. If C is satisfied by ϕ, there is some directed edge uv that is present in both graphs. This follows directly from the fact that, if uv is in both graphs, then G_C must contain the literal $o_{u,v}$ and the assignment ϕ has assigned $o_{u,v}$ to true by definition of these graphs. An example of this is shown in Figure 4.2.

Last, we inspect the orderings that are prohibited by some clause C. For a clause C, the orderings it prohibits are exactly the reversed topological orderings of G_C. We show the visual argument before we come to the formal definition in Lemma 4.1. This argument uses the clause graph G_C and assignment graph G_ϕ. We order the vertices of G from left to right according to the order $\prec$. In this drawing, all edges of G_ϕ point right. If there is at least one edge of G_C also points right, then the graphs G_C and G_ϕ have this edge in common. Otherwise, there is no edge shared by these two graphs. The orderings where all edges of G_C point left are exactly the reversed topological orderings of G_C. An example of this is shown in Figure 4.3.

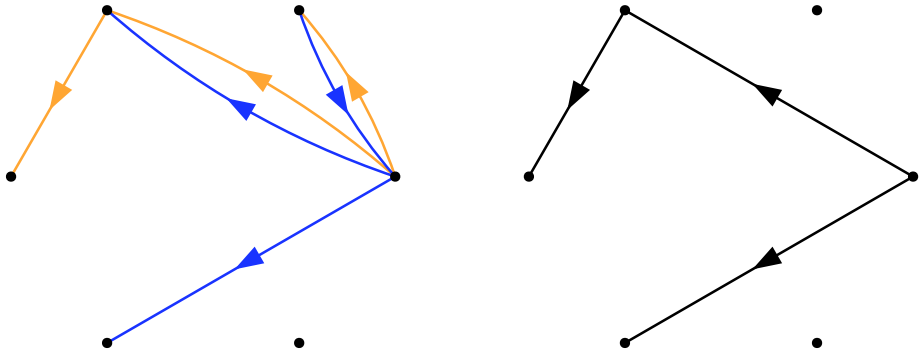

Figure 4.1: An example visualization of the resolution of two clauses on ordering variables for six elements. On the left are two clauses, represented by orange and blue edges, respectively. On the right is the clause resulting from resolving these two clauses.

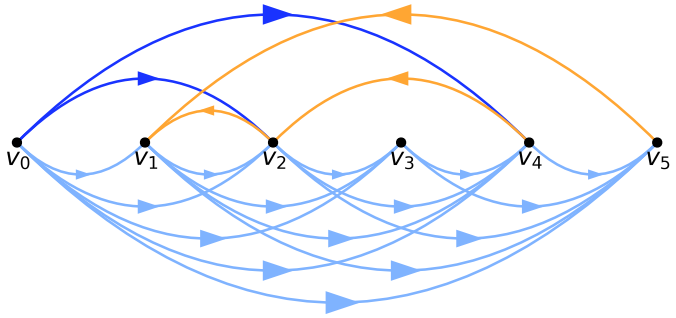

Figure 4.2: An example visualization for the satisfied literals of a clause. The upper edges belong to G_C and the lower edges belong to G_ϕ. The clause is $C = \{o_{v_0,v_2}, o_{v_0,v_4}, o_{v_2,v_1}, o_{v_4,v_2}, o_{v_5,v_1}\}$ and the variable assignment is consistent with order $v_0 \prec v_1 \prec v_2 \prec v_3 \prec v_4 \prec v_5$. The blue edges of G_C correspond to literals which are assigned true under ϕ, while the orange edges correspond to literals which are assigned false under ϕ.

Lemma 4.1: *Let $T^R(G)$ be the set of reversed topological orderings of a graph, that is, the set of orderings $\prec$ of $V(G)$ such that for every directed edge uv, we have $v \prec u$. We take the set of all ordering variables, and assign the ordering variables according to this order, meaning that $\phi(o_{u,v}) = u \prec v$. For a clause C over order variables, the set $T^R(G_C)$ is exactly the set of orderings that do not fulfill C.*

Proof. Let $\prec$ be an ordering of V. If $(\prec) \notin T^R(G_C)$, then there exists an edge $uv \in E(G_C)$ with $u \prec v$. It follows that $\phi(o_{u,v}) = \top$ and because $o_{u,v} \in C$, the variable assignment satisfies C. Otherwise, we have $v \prec u$ for every edge $uv \in E(G_C)$, giving $\phi(o_{u,v}) = \bot$ for all $uv \in E(G_C)$ and the clause C is not satisfied. ∎

From here, we approach the Pattern-Avoiding Vertex Ordering problem in two parts: Section 4.1 deals with the condition that the ordering variables need to adhere to some ordering. Then, in Section 4.2, we show how to ensure the condition that a pattern needs to be avoided in an ordering.

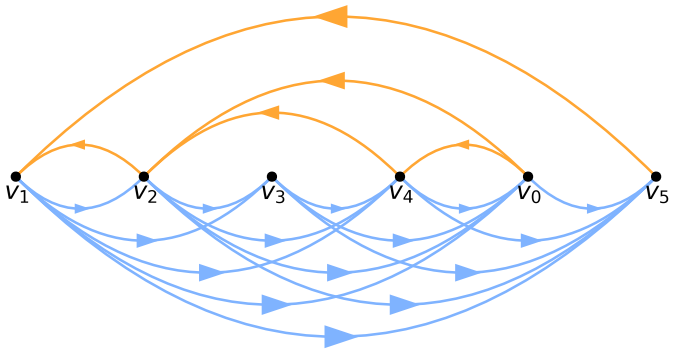

Figure 4.3: An example visualization for an unsatisfied clause. The upper edges belong to G_C and the lower edges belong to G_ϕ. The clause is $C = \{o_{v_0,v_2}, o_{v_0,v_4}, o_{v_2,v_1}, o_{v_4,v_2}, o_{v_5,v_1}\}$ and the variable assignment is consistent with the order $v_1 \prec v_2 \prec v_3 \prec v_4 \prec v_0 \prec v_5$. This is the same clause as in Figure 4.2, but with a different order so that the clause is not satisfied. There is no edge that is both in G_C and G_ϕ, and due to all edges of G_ϕ pointing right in this visualization, all edges of G_C are pointing left.

4.1 Ordering constraint

In this section, we discuss how to ensure that a valid ordering follows from the assignment of the ordering variables. The simplest way to ensure this is to enforce transitivity on the ordering variables. We can encode transitivity by definition:

$$\bigwedge_{\substack{u,v,w \in V \\ u,v,w \text{ distinct}}} o_{u,v} \wedge o_{v,w} \rightarrow o_{u,w} \tag{4.1}$$

Writing the transitivity term in the conjunction of Equation 4.1 as a disjunction, the literals contained in the clause are $\{\overline{o}_{u,v}, \overline{o}_{v,w}, o_{w,u}\}$. We call a clause C of this form, where G_C is a single triangle, a *transitivity clause* from now on. In particular, this means that every transitivity clause is a cycle clause. Although the clause $\{\overline{o}_{u,v}, \overline{o}_{v,w}, o_{w,u}\}$ corresponds to the notation above, we note that $\{o_{u,v}, o_{v,w}, o_{w,u}\}$ and $\{\overline{o}_{u,v}, \overline{o}_{v,w}, \overline{o}_{w,u}\}$ are also transitivity clauses and, in most cases, more elegant examples.

With the transitivity clauses defined, we first investigate what other clauses can be derived from just transitivity clauses. We show in Lemma 4.2 that we can derive all clauses C which contain a cycle in G_C.

Lemma 4.2: *Let $\mathcal{F}$ be a SAT instance with some set of ordering variables U. Let C be an arbitrary cyclic clause. If G_U is chordal, then we can deduce C from the set of all transitivity clauses using only the resolution rule and the weakening rule.*

Proof. First, assume that C is a cycle clause. We show that we can deduce C via resolution from transitivity clauses by induction on the length of the cycle. The base case is that C has length three, and is already a transitivity clause. Now, we assume that we can deduce any cycle clause up to length ℓ. Let $v_1, v_2, \ldots, v_{\ell+1}$ be the vertices of a cycle of length ℓ. We take two arbitrary non-adjacent vertices v_i and v_j such that o_{v_i,v_j} exists. Due to G_U being chordal, a choice for these variables must exist. Without loss of generality, we assume $i < j$. Split the

cycle along this chord into $v_1, v_2, \ldots, v_i, v_j, \ldots, v_\ell, v_{\ell+1}$ and $v_i, v_{i+1}, \ldots, v_{j-1}, v_j$. By assumption, we can deduce both of the clauses for these cycles from transitivity clauses. Resolving over the variable o_{v_i,v_j} gives the complete cycle.

Now, let C be a cyclic clause. We first take an arbitrary cycle of C, and construct it with the procedure above. From there, we can add the remaining literals using weakening. ∎

An important consequence of Lemma 4.2 is, if G_U is chordal, there may not be a cycle in G_ϕ for any assignment ϕ satisfying all transitivity clauses. If G_ϕ contains a cycle on the vertices $v_1, v_2, \ldots, v_\ell$, we can use the construction of Lemma 4.2 to construct a clause C where G_C is the cycle in the opposite direction $v_\ell, v_{\ell-1}, \ldots, v_1$. The assignment ϕ now does not satisfy the clause C.

So far, the introduction of this section focuses heavily on ensuring transitivity. This approach is examined in detail in Section 4.1.1, 4.1.2, and 4.1.3. All we need, however, is that we can reconstruct a valid ordering from the ordering variables $o_{u,v}$. For this, it would be sufficient to ensure that the assignment graph G_ϕ is acyclic, so that we can take a topological ordering of G_ϕ as the order. This idea is discussed more in Section 4.1.4.

4.1.1 Direct encoding of transitivity

We already noted that each term of the conjunction in Equation 4.1 can be transformed into a clause. This section investigates directly using this set of clause to encode transitivity into SAT. We show in Lemma 4.3 that this encoding permits exactly the assignments to the variables consistent with some ordering. Further, in Lemma 4.4, we show that this encoding is minimal in some sense.

Lemma 4.3: *A variable assignment ϕ fulfills the transitivity condition of Equation 4.1 if and only if there exists an ordering $\prec$ of the vertices V such that $\phi(o_{u,v}) = \top \iff u \prec v$ for every $u, v \in V$.*

Proof. First, we argue that the variable assignment ϕ derived from an ordering $\prec$ satisfies every transitivity clause. Because transitivity clauses are cyclic and there are no topological orderings of G_C, this follows directly from Lemma 4.1.

To show that $\prec$ is an ordering if ϕ is a variable assignment respecting all clauses, we need to show that $\prec$ irreflexive, antisymmetric and transitive. Irreflexivity and antisymmetry are already covered in the definition of the variables. For transitivity, if we have $u \prec v$ and $v \prec w$, we know that $\phi(o_{u,v})$ and $\phi(o_{v,w})$ are both true. Because of the transitivity clause $\{\overline{o}_{u,v}, \overline{o}_{v,w}, \overline{o}_{w,u}\}$, we know that $\overline{o}_{w,u}$ must be true. This dictates that $u \prec w$ is also the case and that the relation is transitive. ∎

With Lemma 4.3 proven, we already have one way to ensure transitivity in the SAT solver, namely by adding all transitivity clauses. We call this approach `encode-all`. This encoding is rather large, however: There are $n(n-1)(n-2)/3$ transitivity clauses when the instance graph has n vertices. This number is less than the $n(n-1)(n-2)$ terms in the conjunction of Equation 4.1, but every term is in a group of three which produce the same clause. For fixed

vertices u, v, and w, the three terms in Equation 4.2 result in the same clause $\{\overline{o}_{u,v}, \overline{o}_{v,w}, \overline{o}_{w,u}\}$. Due to this, the number of transitivity clauses is smaller than the number of terms in the conjunction of Equation 4.1 by a factor of 3.

$$\begin{aligned} o_{u,v} \wedge o_{v,w} &\longrightarrow o_{u,w} \\ o_{u,v} \wedge o_{v,w} &\longrightarrow o_{u,w} \\ o_{u,v} \wedge o_{v,w} &\longrightarrow o_{u,w} \end{aligned} \tag{4.2}$$

The approach `encode-all` uses $\Theta(n^3)$ clauses. Using an encoding this large comes with two problems: Even giving the encoding to the SAT solver is already in $\Theta(n^3)$ time, and the SAT solver potentially needs to do a lot of work for unit propagation. We show in Lemma 4.4 that there is no way to improve the number of clauses in the encoding unless we add additional variables, or remove some existing ordering variables.

Lemma 4.4: *Let U be the set of all ordering variables. Unless the set of variables U is modified, using exactly the set of all transitivity clauses is the smallest set ensuring transitivity.*

Proof. We first examine what clauses C are consistent with every ordering. The first claim is that a clause C must be cyclic. This follows from the fact that every acyclic graph has at least one topological ordering, and following Lemma 4.1, an acyclic clause C would prohibit at least one ordering. Building on this, assume that G_C contains a cycle and an edge $uv \in E_C$ such that after removing it, $G_C - uv$ still contains a cycle. This clause still permits all orderings and corresponds to a new clause $C \setminus \{o_{u,v}\}$, which is stronger than C itself. We can conclude that we can strengthen any clause C until the corresponding graph G_C is only a single directed cycle, so every clause in the minimal encoding is a cycle clause.

Let $T = \{o_{u,v}, o_{v,w}, o_{w,u}\}$ be an arbitrary transitivity clause. We now show the minimality of the encoding in Equation 4.1 by constructing a variable assignment ϕ with two properties:

1. It is not consistent with any ordering $\prec$ and should therefore be forbidden.
2. It fulfills every possible clause C where G_C is only a single directed cycle, except when $C = T$.

A variable assignment with these two properties shows that T is necessary. As T is an arbitrary transitivity clause, not one of them can be omitted. To construct this variable assignment ϕ, we need some arbitrary linear order $<$ on V. First, we order all vertices except u, v, and w according to $<$, which is labelled [order] in the formal definition below. Next, we set u, v, and w to be in front of all vertices, labelled [front] below. Last, we set u, v, and w to form the cycle $u \longrightarrow v \longrightarrow w \longrightarrow u$, labelled [cycle] below. The assignment graph for the variable assignment ϕ following from this is shown in Figure 4.4. The formal definition of ϕ is as follows:

$$\phi(o_{x,y}) = \begin{cases} x < y & x \notin \{u,v,w\} \wedge y \notin \{u,v,w\} & \text{[order]} \\ \top & x \in \{u,v,w\} \wedge y \notin \{u,v,w\} & \text{[front]} \\ \bot & x \notin \{u,v,w\} \wedge y \in \{u,v,w\} & \\ \top & (x,y) \in \{(u,v),(v,w),(w,u)\} & \text{[cycle]} \\ \bot & (x,y) \in \{(v,u),(w,v),(u,w)\} & \end{cases} \tag{4.3}$$

The resulting relation $\prec$ asserts $u \prec v \prec w \prec u$, implying that $u \prec u$. This conflicts with the irreflexivity of any ordering and confirms that ϕ has the first property. For the second property, let C be an arbitrary clause where G_C is a single cycle. We now distinguish three cases based on the vertices of the cycle. If the cycle contains both vertices in $\{u, v, w\}$ and

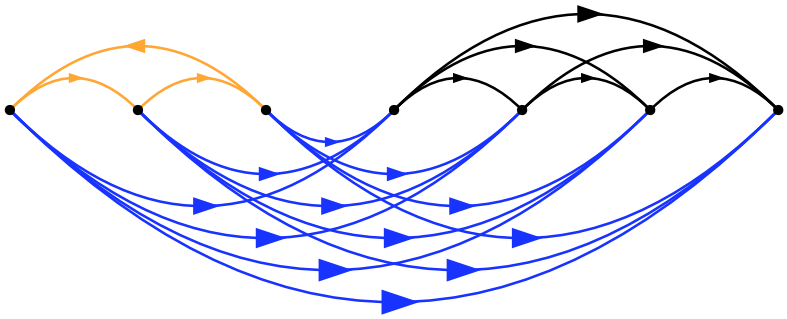

Figure 4.4: The assignment graph G_ϕ of the assignment constructed in Equation 4.3 to show that every transitivity clause is necessary. The black edges correspond to the [order] part of Equation 4.3, the blue edges to the [front] part, and the orange edges to the [cycle] part.

$V \setminus \{u, v, w\}$, it also has an edge from $\{u, v, w\}$ to $V \setminus \{u, v, w\}$ and so, the clause is fulfilled by a variable set in [front]. The clauses C with the vertices in the cycle of G_C exclusively in $V \setminus \{u, v, w\}$ are fulfilled by [order]. Last, there is exactly one relevant clause C on $\{u, v, w\}$ which is not T, namely $\{o_{u,v}, o_{v,w}, o_{w,u}\}$. This clause is fulfilled on all three literals by [cycle].

Last, we note that due to requiring that no new variables are added, it is sufficient to only look at clauses over the ordering variable $o_{u,v}$. Therefore, every transitivity clause is necessary, and we can conclude that the encoding consisting of all transitivity clauses has the minimal number of clauses. ∎

4.1.2 Limiting transitivity

From Lemma 4.4, we know that it is impossible to reduce the number of clauses encoding transitivity if the set of variables U is not modified. In this section, we focus on the idea that for some ordered patterns P and graphs G, the order of some vertices $u, v \in V(G)$ might not be relevant. In this case, we can remove the variable $o_{u,v}$ ordering these two vertices, together with all clauses this variable appears in. We formalize this insight in the following more general lemma:

Lemma 4.5: *Consider a SAT instance $\mathcal{F}$ that only has ordering variables, but not necessarily all possible ordering variables. Fix one ordering variable $o_{u,v}$ in $\mathcal{F}$. Let $\mathcal{F}'$ be the instance resulting from removing $o_{u,v}$ and all clauses $o_{u,v}$ appears in from $\mathcal{F}$. If $o_{u,v}$ appears in no acyclic clause in $\mathcal{F}$, and it is possible to deduce every cycle clause in $\mathcal{F}'$ over the remaining ordering variables, then $\mathcal{F}$ and $\mathcal{F}'$ are equi-satisfiable.*

Proof. Let $\mathcal{F}$ be the SAT-instance with $o_{u,v}$, and $\mathcal{F}'$ the SAT-instance without $o_{u,v}$. To prove equi-satisfiability between $\mathcal{F}$ and $\mathcal{F}'$, we show that we can construct a satisfying variable assignment ϕ for $\mathcal{F}$ from a satisfying assignment ϕ' for $\mathcal{F}'$, and vice versa. Showing the direction from ϕ to ϕ' is as simple as restricting the assignment: Because $\mathcal{F}'$ is only a subset in variables and clauses of $\mathcal{F}$, it follows that $\phi' = \phi_{|\mathcal{F}'}$ is a satisfying assignment for $\mathcal{F}'$.

Now, we look at the direction from ϕ' to ϕ. Let $o_{u,v}$ be the variable removed when transforming $\mathcal{F}$ to $\mathcal{F}'$. To extend ϕ' to a satisfying assignment ϕ for $\mathcal{F}$, we choose an assignment for $o_{u,v}$ and leave all other variables unmodified. Note that by the condition of Lemma 4.5, the variable $o_{u,v}$ only appears in cyclic clauses in $\mathcal{F}$. We only focus on a stronger subset of cyclic clauses in $\mathcal{F}$, namely the cycle clauses, and show that we can extend ϕ' to fulfill all cycle

clauses in $\mathcal{F}$. A cycle clause is not fulfilled if the cycle in the opposite direction appears in G_ϕ. It is therefore sufficient to assign $o_{u,v}$ in such a way that adding the corresponding edge does not close any cycle in the assignment graph $G_{\phi'}$. Because all cycle clauses are deducible in $\mathcal{F}'$, and ϕ' satisfies $\mathcal{F}'$, we know that $G_{\phi'}$ does not contain a cycle. It follows that $u \rightsquigarrow v$ and $v \rightsquigarrow u$ cannot both be true in $G_{\phi'}$, as these two paths together would form a cycle. Therefore, if $u \rightsquigarrow v$, we can set $\phi(o_{u,v}) = \top$ and if $u \not\rightsquigarrow v$, we can set $\phi(o_{u,v}) = \bot$:

$$\phi(x) = \begin{cases} \top & x = o_{u,v} \wedge u \rightsquigarrow v \\ \bot & x = \overline{o}_{u,v} \wedge u \rightsquigarrow v \\ \bot & x = o_{u,v} \wedge u \not\rightsquigarrow v \\ \top & x = \overline{o}_{u,v} \wedge u \not\rightsquigarrow v \\ \phi'(x) & \text{otherwise} \end{cases}$$

■

Lemma 4.5 requires that every cycle clause needs to be deducible using the given clauses in the reduced instance $\mathcal{F}'$. One way would be to just add all cycle clauses, but this might result in a large number of clauses. However, we already know from Lemma 4.2 that we can deduce all cycle clauses from transitivity clauses, as long as G_U is chordal. This means we can repeatedly apply Lemma 4.5 and remove variables as long as they do not appear in an acyclic clause and G_U remains chordal. In the context of Pattern-Avoiding Vertex Ordering, these acyclic clauses are pattern-avoidance clauses.

The idea to remove unnecessary variables, but to leave a chordal graph of related vertices is already used [VG09] for the Hamilton Cycle problem. In that paper, it is also noted that the condition of having a chordal graph of variables can be approached by first adding all necessary variables, and then adding variables until the graph is chordal. Another, even earlier use of this idea is presented in [BV02]. That paper investigates symmetric transitive relations, where $\phi(e_{i,j}) = \top$ if the elements i and j are in relation.

We make use of Lemma 4.5 by finding a chordal graph containing all variables needed by acyclic clauses. To find this chordal graph, start with a variable graph containing all needed variables. The fill-in of a vertex v is the number of missing edges in its neighborhood $|\{xy \mid x, y \in N(v) \wedge xy \notin E\}|$. Choose the vertex v with the smallest fill-in and add all missing edges to the neighborhood, then remove v. Repeat this process until the graph is empty, then add all vertices back in, together with the original and added edges. We use this graph as the variable graph G_U. Add a transitivity clause for each triangle in the variable graph G_U. We call this approach `encode-needed`.

4.1.3 User-defined unit propagators

User propagators allow for interaction with the SAT solver during the solving process. A deeper explanation of the features of user propagation, general motivations and experiments with CaDiCaL are presented in [Faz+24]. We give a quick overview over user propagator functionality useful for us:

- **Notification of variables assignments:** These notifications provide the user propagator with information about the current state of variables in the SAT solver. This includes notifications when a literal becomes unassigned. These notifications are 'stack-like': When a literal becomes unassigned, it is the last literal that has been assigned and has not yet been unassigned.

- **Unit propagation**: CaDiCaL, like most modern SAT solvers, performs unit propagation [IB21]. When the solver has fixed the assignment of all but one literal in a clause to false, the last literal must be true and can be immediately fixed as well. These variable assignment deductions can cause more unit propagations in other clauses. Instead of giving the solver the clauses, the user propagator allows us to give the SAT solver deduced variable assignments. For each of these variable assignments, the solver may ask the use propagator for a reason clause.

- **Clause addition**: Clauses can be added during the search. This method, as opposed to incremental SAT, does not need to restart the search and can react to conflicts caused by the new clause immediately.

- **Model checking**: Once the SAT solver found a satisfying assignment, the user propagator can check if it is valid under constraints not given to the solver. If not, the SAT solver expects a clause to be added. After that, the solver can then continue searching for an assignment.

We use different parts of user propagators in the upcoming sections. For now, we just focus on the unit propagation part: Instead of adding transitivity clauses to the SAT solver, we deduce the variables following from transitivity and give them to the solver. We call this approach `unit-propagator-all` when using all ordering variables, and `unit-propagator-needed` when using the reduced variable set we can get when using Lemma 4.5.

When the literal $o_{u,v}$ is assigned by the SAT solver, we iterate over all $x \in V$ such that $o_{x,u}$ is already assigned, and over all $y \in V$ such that $o_{v,y}$ is already assigned. This iteration can be done efficiently by maintaining the list of all such x and y for each vertex in a stack. In other words, we maintain the graph G_ϕ, which is modified through edge additions and stack-like edge deletions. This accurately reflects the current state because the SAT solver announces variable assignments and backtracks on them in a stack-like manner. The user propagator then deduces that $o_{x,v}$ and $o_{u,y}$ must be true. These are not all units that could be derived from this assignment, but are sufficient to ensure that the solver eventually receives all units that can be derived. Take for example the case where $o_{x',u}$ and $o_{v,y'}$ are already assigned, and the unit propagator now gets a notification for the assignment of $o_{u,v}$. It is now correct to derive $o_{x',y'}$, but only $o_{x',v}$ and $o_{u,y'}$ are given back to the solver. The unit $o_{x',y'}$ is derived in the next step, for example when $o_{x',v}$ is given to the unit propagator, together with the already known unit $o_{v,y'}$. When a unit $o_{u,w}$ is derived from $o_{u,v}$ and $o_{v,w}$, we also need to remember these reasons. If the SAT solver then requests a justification for $o_{u,w}$, we can give the reason clause $\{\overline{o}_{u,v}, \overline{o}_{v,w}, o_{u,w}\}$.

For the analysis of the unit propagator, we take a look at a single branch of the solver from an empty to a complete assignment. In particular, we consider no backtracks for now. In this case, the unit propagator might still do up to $\Theta(n^3)$ steps. This happens, for example, when ordering all ordered pairs of vertices (u, v) in lexicographic order and then assigning the ordering variable $o_{u,v}$ corresponding to each pair in that order. On the other hand, the best case is now $\mathcal{O}(n^2)$ steps to assign all variables, which matches the number of variables and is therefore best possible. This occurs, for example, when the literals $o_{v_i,v_{i+1}}$ are assigned for each i from 1 to $n-1$, in that order. All variables not explicitly assigned in this order are assigned by unit propagation.

Given that the worst case has no improvement and the best case is the best possible, a look at the average case seems interesting. What 'average' means is however difficult to pin down, due to these variable assignments coming from the SAT solver. The heuristics used by

SAT solvers for choosing variables to assign and what assignment to choose first in a branch make the probability distribution over all possible cases very unpredictable. We can make one observation however, again ignoring backtracks. Let $o_{u,v}$ be the literal to be assigned. Then there will be $\deg^+_{G_\phi}(u) + \deg^-_{G_\phi}(v)$ unit propagation of already assigned literals. We can see this by considering some vertex $x \in N^+_{G_\phi}(u)$ and look at the moment the variable $o_{v,x}$ is assigned. When the literal $o_{v,x}$ is assigned, the unit propagator finds $o_{u,x}$, which is already assigned. On the other hand, if $o_{x,v}$ is assigned, the unit propagator finds $o_{u,v}$, which again is already assigned. A similar argument can be made to explain the $\deg^-_{G_\phi}(v)$ term.

4.1.4 CEGAR

Counterexample-guided abstraction refinement (CEGAR) is a general strategy in problem-solving with SAT that initially uses fewer constraints than necessary and adds additional constraints when a solution is found that does not satisfy the full set of constraints [CGS04]. The solution that violates the full constraint set is the counterexample that guides the refinement. An example for the application of CEGAR is presented in [Oha+25]. The authors use an encoding for Hamilton Cycle into SAT which ensures that for every vertex, exactly two incident edges are part of the cycle, and that only one cycle exists globally. While the first set of constraints is directly encoded, the second is done via CEGAR: Once a solution is found, a check is performed to validate that only one cycle exists. If that is the case, a solution is found. Otherwise, a cut between some cycle and the rest of the graph is found, and then the additional constraint is added that at least one of these edges must be part of the solution.

At this point, we need to reevaluate what ensuring an ordering means. Already in Section 4.1.2, we have departed slightly from the typical notion of an ordering with Lemma 4.5. This lemma allows us to remove some ordering variables and get an equi-satisfiable instance after the variable removal. However, this means that we need to fill in the relative order of vertices for which we do not have an ordering variable, and in the proof of Lemma 4.5, we show that this is always possible if G_U is chordal. In this section, we go a step further and only make sure that any ordering $\prec_G$ exists over the set of ordering variables U, no matter what this set looks like. We present four ways of ensuring this with CEGAR. Two are based on adding clauses when assigning a literal causes a cycle in the assignment graph G_ϕ, and we maintain that any topological ordering of G_ϕ is also a valid order of G avoiding the pattern P. The next approach is similar, but only adds additional clauses preventing cycles in the assignment graph G_ϕ during model checking. Last, we use an approach that tries to reconstruct a valid order avoiding the pattern P matching the assignment ϕ as well as possible during model checking, and only adds clauses preventing some cycle if this fails.

First, the two approaches that prohibit cycles in G_ϕ upon their creation. Specifically, we want to detect when the literal $o_{u,v}$ is added, but some path from v to u already exists in G_ϕ. Let $(u, x_1, x_2, \ldots, x_\ell, v)$ be this path, with $\ell \geq 1$. In this case, we want to add the following clause C to prevent this cycle:

$$C = \{o_{u,x_1}, o_{x_1,x_2}, \ldots, o_{x_\ell,v}, \overline{o}_{u,v}\} \tag{4.4}$$

To achieve this, we need a data structure which can handle the updates made to G_ϕ and answer the queries needed to add the clause described in Equation 4.4. The exact list of queries and updates needed is:

- Checking if a path from a vertex u to another vertex v exists. This is needed to determine if a clause needs to be added in response to a variable assignment.

- Adding a directed edge uv, with the guarantee that no path from v to u exists. This corresponds to a variable assignment given by the SAT solver.
- Stack-like undoing of edge additions. This corresponds to backtracking done by the SAT solver.
- Giving an example of a path from a vertex u to another vertex v, with the guarantee that such a path does exist. This is needed to determine the variables of the clause C to be added.

Reachability in directed graphs is extensively studied [He+25]. Our setting is however unique due to the last requirement on the data structure: Usually, reachability data structures only need to be able to report whether a path from u to v exists, but not to provide a concrete example. We use two different approaches to implement this. First, when $o_{u,v}$ is assigned, we start a simple BFS starting from v to find a path to u. We call this approach `cycle-prevention`.

The second approach, which we call `reachability-cycle-prevention`, adapts the dynamic reachability from [HHS20]. For this, we choose one random vertex x upon starting the solver. We maintain all vertices that can reach x and all vertices reachable from x. When checking if the path $v \rightsquigarrow u$ exists, we use the following three insights listed in [HHS20]:

- If the paths $v \rightsquigarrow x$ and $x \rightsquigarrow u$ both exist, then a path $v \rightsquigarrow u$ exists.
- If a path $v \rightsquigarrow x$ does not exist, but $u \rightsquigarrow x$ exists, then no path $v \rightsquigarrow u$ exists.
- If a path $x \rightsquigarrow u$ does not exist, but $x \rightsquigarrow v$ exists, then no path $v \rightsquigarrow u$ exists.

We first check if one of these three cases apply. If the paths $v \rightsquigarrow x$ and $x \rightsquigarrow u$ exist, we can chain them together to immediately get an example of a path $v \rightsquigarrow u$. Otherwise, if one of the two cases applies that ensure no path from v to u exists, we immediately go to adding the edge uv to the data structure. Last, if none of these three cases applies, we fall back to searching for a path with a BFS. This concludes the two approaches that add new clauses to prohibit cycles once they appear in the partial assignment.

These two approaches prevent cycles in G_ϕ upon their creation. On the other hand are approaches that only look for cycles once a satisfying assignment has been found. These approaches have the advantage that of speeding up unsatisfiable instances where little to no transitivity clauses are needed, but the disadvantage of giving the solver less direct unit propagation information. The first of these approaches, which we call `cycle-refinement`, implements a simple check if a cycle is present in the assignment via DFS, and if this is the case, adds a clause to prohibit this cycle.

The second approach that adds cycle preventing clauses during model checking, called `path-sampling-refinement`, is a bit more involved: Instead of just prohibiting any cycle, we first reconstruct an order. This order might contain an occurrence of the pattern that the solver does not prohibit due to the assignment graph G_ϕ containing a cycle. We check the order for such a pattern occurrence, and prohibit the cycle that made the order possible.

For the reconstructed order, to make sure there is a cycle we can prohibit if an occurrence of the pattern exists, we require that if an edge uv in the assignment graph is directed backward, meaning that if $v \prec u$, then there is a path $v \rightsquigarrow u$ entirely of edges directed forward. Such an order can be found by reversing the post-order of a DFS. The post-order of a DFS orders vertices by the time they are backtracked out of. A single DFS might not visit all vertices, but we can restart the DFS at another vertex until all vertices are visited. Looking at a tree-,

forward-, or cross-edge uv, we note that the DFS always backtracks out of v before u, and so we have $u \prec v$ in the reversed post-order. The only edge type where the edge uv is ordered as $v \prec u$ is back edges. Here, we have a path of tree edges from u to v by definition of a back edge. This shows that the reversed post-order has exactly the properties we want from an order.

Now, we have the order $\prec$ found by the DFS. If we find an occurrence of the pattern on this order $\prec$, we know that there is at least one consecutive pair of vertices (u, v) in the pattern with the edge directed backwards. When all of these edges are directed forward, the SAT solver would already detect this pattern and not give this assignment to the model check. We also note that the ordering variable $o_{u,v}$ must exist, even after doing the variable removals allowed by Lemma 4.5, because the removal of an ordering variable is not allowed when it appears in a pattern avoidance clause. Let $(v, x_1, x_2, \ldots, x_\ell, u)$ be the forward directed path from v to u, with $\ell \geq 1$. This path can be found by starting at v and then going to the parent in the DFS tree we used to find the order of vertices, due to vu being a back edge. Then, similar to the one in Equation 4.4, we add the following clause C to prevent this cycle:

$$C = \{o_{v,x_1}, o_{x_1,x_2}, \ldots, o_{x_\ell,u}, \overline{o}_{v,u}\}$$

This concludes the approaches we have to ensure that the ordering variables do in fact form an ordering. We note that in the worst case, these approaches might still add $\Omega(n^3)$ clauses. This is due to the fact that none of these approaches add new variables, and the reduction of Lemma 4.5 might not be applicable. With this, it follows from Lemma 4.4 that enforcing transitivity with certainty requires $\Omega(n^3)$ clauses. Even though these approaches are not dealing with transitivity directly, they do add cyclic clauses at some point without using new variables and are therefore subject to the same lower bound.

4.1.5 No encoding

So far in this section, we discussed different ways to ensure that the ordering variables do form some valid ordering. Surprisingly, this is not necessary for all patterns: In the case of the pattern representing the coloring (as seen for example in Figure 3.10 in Section 3.1), adding only the pattern avoidance clauses is already sufficient to encode the Pattern-Avoiding Vertex Ordering problem into SAT. Intuitively, this is the case because we can always find these patterns P when walking along a cycle long enough. The formal statement is given in Lemma 4.6.

Before we come to formally stating this, we need to clarify what an occurrence of pattern P is when the variable assignment ϕ of the ordering variables $o_{u,v}$ do not follow transitivity anymore. In this case, we say that the sequence of vertices $v_1, v_2, \ldots, v_k$ is an occurrence of the pattern if the subgraph induced by these vertices in this order matches the mandatory and forbidden edges of the pattern P, and if $\phi(o_{v_i,v_{i+1}}) = \top$ for all $i \in [k-1]$. That is, we are only requiring the ordering variables between consecutive vertices of the pattern occurrence to order the pattern correctly. This is similar to the actual pattern avoidance clauses used in Section 4.2.

Lemma 4.6: *Let P be a pattern with k vertices and only the edges $E_M(P) = \{p_i p_{i+1} \mid i \in [k-1]\}$. Let ϕ be a variable assignment fulfilling all pattern avoidance clauses. Then ϕ also fulfills all transitivity clauses, even when they are not part of the encoding.*

Proof. Let P be a pattern with the restrictions given in Lemma 4.6. For the variables, we are exclusively using the ordering variables $U = \{o_{u,v} \mid uv \in E(G)\}$ for each edge. Due to every pair of consecutive vertices being connected in P, we know that U contains all variables needed for the pattern avoidance clauses. Let ϕ be a variable assignment, and assume there is a cycle $v_1, v_2, \ldots, v_\ell$ in G_ϕ. We know that for each consecutive pair of vertices $v_i v_{i \bmod \ell+1}$, they are connected by an edge in G, otherwise the variable would not even exist. Consider the following infinite sequence of vertices:

$$v_1, v_2, \ldots, v_\ell, v_1, v_2, \ldots, v_\ell, v_1, \ldots$$

We know that for any two consecutive elements x and y of this sequence that $\phi(o_{x,y}) = \top$ and $xy \in E(G)$. Therefore, the first k vertices of this sequence are an occurrence of the pattern P, and because the sequence is infinite, the pattern P occurs no matter what k is used. With this, we know that any cycle in G_ϕ would violate the pattern avoidance clauses and G_ϕ must be cycle free. ∎

The result of Lemma 4.6 raises the question of when we actually need to ensure the ordering constraint for any pattern. While we cannot answer the question if the patterns shown in this lemma are the only ones not requiring the ordering constraint, we can give an example when it is required: Let P be the pattern for forests, that is, the pattern with only the mandatory edges $E_M(P) = \{p_1p_2, p_1p_3\}$. Further, take the cycle graph with the five vertices v_1, v_2, v_3, v_4, v_5. To avoid handling long indices due to moduli, we implicitly understand all indices in this example to cycle after 5 to 1, for example $v_1 = v_6 = v_{11} = \ldots$ for the first vertex. The edges of G are $\{v_1v_2, v_2v_3, v_3v_4, v_4v_5, v_5v_1\}$. We now define the variable assignment ϕ for the ordering variables o_{v_i,v_j}. For each vertex v_i, we set the variables the two variables pointing to the next two vertices $o_{v_i,v_{i+1}}$ and $o_{v_i,v_{i+2}}$ to true, see Equation 4.5. The assignment graph G_ϕ is shown in Figure 4.5.

$$\phi(o_{v_i,v_j}) = \begin{cases} \top & i+1 = j \mod 5 \\ \top & i+2 = j \mod 5 \\ \bot & \text{otherwise} \end{cases} \tag{4.5}$$

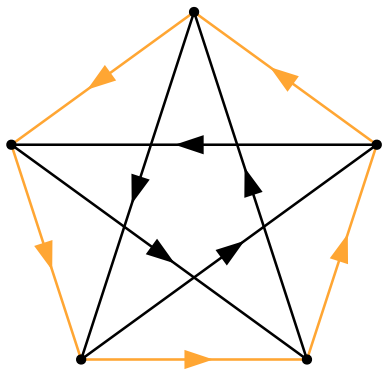

Figure 4.5: The assignment graph G_ϕ for the assignment ϕ of Equation 4.5. The five edges that are part of the cycle in the graph G are the outer ones highlighted in orange. This assignment avoids the pattern for forests, even though a cycle is not a forest.

For the pattern to occur, we would need that any vertex v_i comes first, the one of its neighbors v_{i+1} or v_{i+4}, and then the other neighbor. That is, we would either need $\phi(o_{v_i,v_{i+1}}) = \phi(o_{v_{i+1},v_{i+4}}) = \top$, or $\phi(o_{v_i,v_{i+4}}) = \phi(o_{v_{i+4},v_{i+1}}) = \top$. This is never the case in the given assignment ϕ, and so, the pattern is avoided. An example of how this partial assignment needs to look like is shown in Figure 4.6.

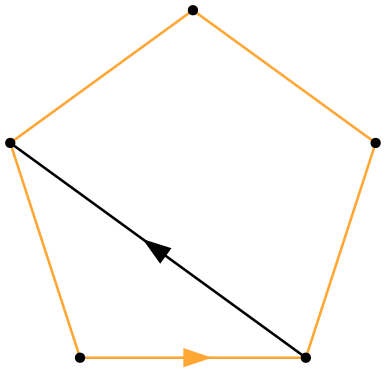

Figure 4.6: The partial assignment that needs to be present in a cycle of length 5 so that the pattern for forests is detected. Only the edges marked with an arrow are part of the assignment graph, the five edges that are part of the cycle in the graph G are the outer ones highlighted in orange. If the assignment graph followed transitivity, this would be present.

4.2 Pattern avoidance constraint

In this section, we discuss ways to prevent the pattern P from occurring in the order $\prec_G$ the SAT solver finds. We look at three ways to do this: First is a direct encoding in Section 4.2.1, which simply adds one prohibiting clause for every sequence of vertices that form an occurrence of the pattern. This encoding is going to turn out prohibitively large, but gives the groundwork for the way we prohibit pattern occurrences. Next is a recursive encoding in Section 4.2.2, which is based on the recursive construction of positive outerplanar patterns discussed in the preliminaries Section 2.1. Last, in Section 4.2.3, we look at CEGAR approaches like the ideas used for the ordering constraint in Section 4.1.4.

4.2.1 Direct encoding

Let $v_1, v_2, \ldots, v_k$ be a set of vertices that, if ordered $v_1 \prec v_2 \prec \cdots \prec v_k$, would form the pattern P. To encode pattern avoidance, the most straight-forward idea is to find every possible sequence of vertices forming a pattern and prohibit it. The direct translation of this idea into a disjunction would be as follows:

$$\bigvee_{i=0}^{k} \bigvee_{j=i+1}^{k} \overline{o}_{i,j}$$

Using this translation yields a clause with $k(k-1)/2$ literals. This clause is however highly redundant: If, for example, the clause is satisfied on the literal $\overline{o}_{v_1,v_3}$, then it must also be satisfied on at least one of $\overline{o}_{v_1,v_2}$ or $\overline{o}_{v_2,v_3}$. If this were not the case, and both of $\overline{o}_{v_1,v_2}$ and $\overline{o}_{v_2,v_3}$ are false, then transitivity would also force $\overline{o}_{v_1,v_3}$ to be false. This argument can be made for every literal $\overline{o}_{v_i,v_j}$ if $j-i>1$. After cutting all these literals, the following remains:

$$\bigvee_{i=0}^{k-1} \overline{o}_{i,i+1} \tag{4.6}$$

This gives the clause $C = \{\overline{o}_{v_1,v_2}, \dots, \overline{o}_{v_{k-1},v_k}\}$. We note that G_C is a single path from v_k to v_1. From Lemma 4.1, we know that a clause C prohibits the reversed topological orderings of G_C. This is consistent with the goal we are trying to achieve: Any reversed topological ordering of G_C has the vertices $v_1, v_2, \dots, v_k$ in this order, which would form the pattern.

Before we continue with the discussion of the direct encoding, we can give an additional note on a known algorithm from [HMR14] solving the Pattern-Avoiding Vertex Ordering for patterns (and sets of patterns) with up to three vertices in polynomial time. This algorithm is based on 2SAT, which makes sense due to pattern avoidance clauses for patterns of order 3 always being of the form $\{\overline{o}_{u,v}, \overline{o}_{v,w}\}$. We thus ask whether a SAT instance over ordering variables encoding transitivity and clauses of the form $\{\overline{o}_{u,v}, \overline{o}_{v,w}\}$ can always be solved efficiently, or if it is necessary to incorporate the knowledge that these constraints come from avoiding a pattern in a graph. We call this problem *Ordered Linear 2SAT*. In Lemma 4.7, we show that knowledge of the constraint structure is actually necessary, by showing that the Ordered Linear 2SAT problem is NP-complete. This separates Ordered Linear 2SAT from Pattern-Avoiding Vertex Ordering with patterns of order 3, assuming that $P \neq NP$.

Lemma 4.7: *Let $\mathcal{F}$ be an Ordered Linear 2SAT instance, meaning a SAT instance where every clause is either of the form $\{o_{i,j}, o_{j,k}\}$ or $\{\overline{o}_{i,j}, \overline{o}_{j,k}, o_{i,k}\}$ with $i,j,k \in [n]$. Ordered Linear 2SAT is NP-complete.*

Proof. The problem Ordered Linear 2SAT is in NP due to SAT being in NP. Any polynomial time verifying algorithm for general SAT also verifies a solution for this problem.

It remains to show that the problem is NP-hard. We show this by reduction from Betweenness, which is shown to NP-hard in [Opa79]. The Betweenness problem gives a size n and constraints of the form (x,y,z), where $x,y,z \in [n]$. The goal is to find an ordering $\prec$ of the elements of $[n]$ such that for each constraint (x,y,z), the ordering either fulfills $x \prec y \prec z$ or $z \prec y \prec x$. In other words, the element y needs to be between x and z.

In the reduction from Betweenness to Ordered Linear 2SAT, we first add the transitivity clause $\{\overline{o}_{i,j}, \overline{o}_{j,k}, o_{i,k}\}$ for every possible $i,j,k \in [n]$. Then, for each constraint (x,y,z) in the Betweenness instance, add the following four clauses:

$$\begin{aligned} C_{\text{out,left}} &= \{o_{y,x}, o_{x,z}\} \\ C_{\text{out,right}} &= \{o_{y,z}, o_{z,x}\} \\ C_{\text{in,left}} &= \{o_{z,x}, o_{x,y}\} \\ C_{\text{in,right}} &= \{o_{x,z}, o_{z,y}\} \end{aligned} \tag{4.7}$$

All of these clauses are of the form $\{o_{u,v}, o_{v,w}\}$. We now show the equivalence between the Betweenness instance and the instance we just constructed. In the first step, we show that we can turn a solution $\prec$ for the Betweenness instance into a solution ϕ for the Ordered Linear 2SAT instance. The second step shows the converse direction.

First, let $\prec$ be an ordering that is a solution to the Betweenness instance. As usual, we define the variable assignment ϕ by $\phi(o_{i,j}) = i \prec j$. Because $\prec$ is an order and fulfills transitivity, all clauses of the form $\{\overline{o}_{i,j}, \overline{o}_{j,k}, o_{i,k}\}$ are satisfied. This is already formally proven in Lemma 4.1 in the introduction to Chapter 4. It remains to show that the clauses of Equation 4.7 are satisfied. This can be done by simply verifying that for both possible ways the constraint (x, y, z) is fulfilled, each of these clauses has a satisfied literal. Table 4.1 lists all clauses of Equation 4.7 and on which literal they are satisfied on in both cases.

Clause	$x \prec y \prec z$	$z \prec y \prec x$
$C_{\text{out,left}}$	$o_{x,z}$	$o_{y,x}$
$C_{\text{out,right}}$	$o_{y,z}$	$o_{z,x}$
$C_{\text{in,left}}$	$o_{x,y}$	$o_{z,x}$
$C_{\text{in,right}}$	$o_{x,z}$	$o_{z,y}$

Table 4.1: The list of clauses added for each constraint (x, y, z) of Betweenness problem, and the literal they are satisfied on. The constraint (x, y, z) is fulfilled by either $x \prec y \prec z$ or $z \prec y \prec x$, and both of these two cases satisfies each clause of Equation 4.7 on one of its two literals.

On the other hand, assume ϕ satisfies all clauses. Again, we use $\phi(o_{i,j}) = i \prec j$. This defines a valid ordering due to the SAT instance containing all clauses ensuring transitivity. Again, this also follows from Lemma 4.3 in Section 4.1. We can deduce two more clauses using resolution:

$$\begin{aligned} C_{\text{out}} &= C_{\text{out,left}} \otimes_{o_{x,z}} C_{\text{out,right}} \\ &= \{o_{y,x}, o_{z,x}\} \\ C_{\text{in}} &= C_{\text{in,left}} \otimes_{o_{x,z}} C_{\text{in,right}} \\ &= \{o_{x,y}, o_{z,y}\} \end{aligned}$$

These two clauses define $o_{x,y} \leftrightarrow o_{y,z}$. This means that if we have $\phi(o_{x,y}) = \top$, we must also have $\phi(o_{y,z}) = \top$ and so $x \prec y \prec z$ fulfilling the (x, y, z) constraint. Likewise, $\phi(o_{x,y}) = \bot$ implies $\phi(o_{y,z}) = \bot$ which gives $z \prec y \prec x$, again fulfilling the (x, y, z) constraint of the Betweenness problem. ∎

This concludes the theoretical consideration on the difference between Pattern-Avoiding Vertex Ordering and the more general Ordered Linear 2SAT. We now return to the direct, exhaustive encoding of pattern avoidance following from Equation 4.6. Here, we have another, bigger problem with this encoding approach, namely that there are potentially many vertex sequences that could form the pattern. For a positive pattern, a complete graph on n vertices has $\Theta(n^k)$ possible pattern occurrences. We note in Lemma 4.8 that this problem also appears with satisfiable instances.

Lemma 4.8: *Let $k \geq 4$ and $n \geq k$ be positive even integers. There exists a pattern P with k vertices and a graph with n vertices such that G can be ordered to avoid P, but there are $\Theta(n^k)$ possible occurrences of P when ordering G.*

Proof. We use the pattern P with k vertices and with the edges $E_M(P) = \{p_i p_{i+1} \mid i \in [k-1]\} \cup \{p_1 p_k\}$. For the graph G, we use the complete bipartite graph where both partitions contain $n/2$ vertices. An example with $k = 4$ and $n = 8$ is shown in Figure 4.7. This graph G can be ordered to avoid P by first placing all vertices of one partition, and then the vertices

of the other partition. This ordering already avoids the pattern P' with $k' = 3$ vertices and edges $E_M(P') = \{p_1p_2, p_2p_3\}$, which is the pattern defining bipartite graphs. We also note that P' is weaker than P: First, we remove all edges from P but p_1p_2 and p_2p_3, resulting in a weaker pattern following from Lemma 3.5. Then, we remove all vertices except the first three, resulting in a weaker pattern following from Lemma 3.7. This results in P'. Therefore, G can be ordered to avoid P.

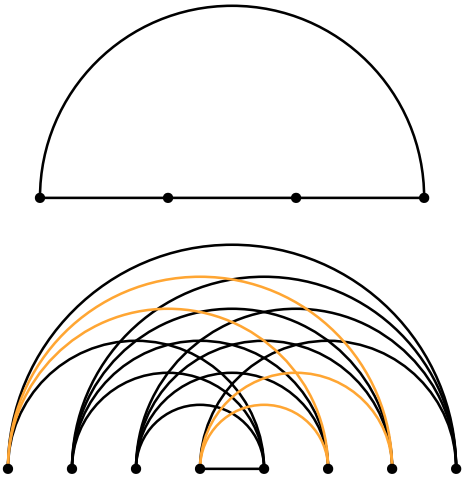

Figure 4.7: At the top is an example for the pattern P with $k = 4$. At the bottom is a graph G ordered to avoid the pattern P, but containing many cycles of length 4. One such cycle is highlighted in orange. Exhaustively adding a pattern avoidance clause for each possible pattern occurrence requires $\Theta(n^4)$ clauses.

We now only need to argue that the number of possible pattern occurrences is in $\Theta(n^k)$. We give this number precisely in the following:

$$2\left(\frac{(n/2)!}{(n/2-k)!}\right)^2$$

To see this, we construct one vertex sequence that would form the pattern P. First, we choose one of the two partitions of G to start in, giving us the factor 2 in front. Then, we alternate between the two partitions and choose one new vertex each time. With this, we have $n - i + 1$ choices for the ith vertex chosen in a partition. The total number of choices in one partition is given by:

$$n/2 \cdot (n/2-1) \cdot \dots \cdot (n/2-k+1) = \frac{(n/2)!}{(n/2-k)!}$$

There are two partitions, so we get this factor twice. Together with the choice of starting partition, we have the claimed number of possible occurrences. In general, this is in $\Theta(n^k)$. ∎

We note that the number of clauses we need to add in the proof of Lemma 4.8 is more than the number of cycles of length k present in the bipartite graph because a single cycle needs multiple clauses. When only fixing the vertices, changing the starting point or reversing the order of vertices results in different possible patterns.

Coming back to the issues with this encoding, even finding all possible patterns is a difficult problem in its own right, and has been tackled for example by the Glasgow Subgraph Solver [MPT20]. All these factors motivate the search for other encoding methods.

4.2.2 Recursive encoding

In Section 2.1, we note that we can build any positive outerplanar pattern from two base patterns and using two combining operations. The base patterns are the two patterns on two vertices, once with and once with mandatory edge. The two combining operations both take two patterns and unify the last vertex of the first pattern with first vertex of the second pattern. One of these operations does nothing further, while the other one also adds a mandatory edge between the two outermost vertices, which we call a covering edge. We build an encoding for pattern avoidance from this construction.

For this encoding, let P_0 be the pattern with two vertices and no edge and P_1 the pattern with two vertices and one mandatory edge connecting them. Construct the pattern $P_h = P$ as a chain of patterns $P_2, P_3, \ldots, P_h$, where each pattern can be built by applying (covered) pattern chaining to two previous patterns. For each $i \in \{0, 1, 2, \ldots, h\}$ and each pair of vertices $u, v \in V(G)$, we add the variable $p_{i,u,v}$. Intuitively, the variable $p_{i,u,v}$ means that the pattern P_i can be found starting at u and ending at v. This already means that $p_{i,u,v}$ can only be assigned true if $o_{u,v}$ is also assigned true.

Next, we come to constraints these variables need to fulfill. We first handle the two base cases. The pattern with two vertices and no edges can be found from u to v if and only if $u \prec v$. If the pattern also has a mandatory edge between the two vertices, we also need uv to be an edge in G.

$$\bigwedge_{u,v \in V} p_{0,u,v} \leftrightarrow o_{u,v} \tag{4.8}$$

$$\bigwedge_{uv \in E} p_{1,u,v} \leftrightarrow o_{u,v} \tag{4.9}$$

For the remaining subpatterns $P_2, P_3, \ldots, P_h$, we need to translate the pattern chaining operations into clauses. First, we take all subpatterns that were created using the pattern chaining operation without adding a covering edge. We can describe each of these operations by a triple (i, l, r), meaning that the subpattern P_i is created by chaining P_l followed by P_r. Let I be the set of all these triples. When the ordered graph contains the pattern P_l from u to w, and the following pattern P_r from w to v, then there is the chained pattern P_i from u to v.

$$\bigwedge_{(i,l,r) \in I} \bigwedge_{u,v,w \in V} p_{l,u,w} \wedge p_{r,w,v} \rightarrow p_{i,u,v} \tag{4.10}$$

The logic for covered patterns is similar. Let $\tilde{I}$ be the set of all triples describing covered pattern chaining operations.

$$\bigwedge_{(i,l,r) \in \tilde{I}} \bigwedge_{\substack{uv \in E \\ w \in V}} p_{l,u,w} \wedge p_{r,w,v} \rightarrow p_{i,u,v} \tag{4.11}$$

Last, we need to make sure that the complete pattern P_h cannot appear. To do so, we simply require all variables $p_{h,u,v}$, which represent the complete pattern P_h being between u and v, to be false.

$$\bigwedge_{u,v \in V} \neg p_{h,u,v} \tag{4.12}$$

There are some optimizations we can do when choosing the variables, and when encoding these constraints into clauses. First, the variables $p_{0,u,v}$ and $p_{1,u,v}$ are always equivalent to the variable $o_{u,v}$ for all $u, v \in V$ by Equation 4.8 and 4.9, and can therefore be replaced by $o_{u,v}$. Additionally, if the variable $p_{i,u,v}$ for $i \geq 2$ and $u, v \in V$ is never on the right side of the

implication of Equation 4.10 or 4.11, then it is a pure variable and can be eliminated. Such eliminations might cause more pure variable eliminations, because there might be variables that are only on the left side of implications where the right side is an eliminated variable. Intuitively, a variable $p_{i,u,v}$ is eliminated if the subpattern P_i from u to v can never be part of a complete pattern. We can also replace all occurrences of $p_{t,u,v}$ for all $u, v \in V$ with $\bot$, due to the requirement of Equation 4.12. Last, we can generally restrict the variables $P_{i,u,v}$ to those with $u \neq v$, because only the pattern with one vertex could occur from one vertex to itself, and our base case patterns already have two vertices. The important exception to this is when we do not use a transitivity encoding, like discussed in Section 4.1.5.

The worst case for the number of clauses created from Equation 4.10 and 4.11 is up to $\mathcal{O}(n^3 \cdot k)$. This bound comes due to the conjunction of up to t operations, where we note that $t \leq k - 1$, and then a conjunction over triples of vertices of G. This is still problematic for large graphs, but not as bad as the up to $\mathcal{O}(n^k)$ clauses of the direct encoding.

Last, we note that this encoding can be extended for patterns that are not outerplanar or positive. The construction of positive outerplanar patterns shown in Section 2.1 is based on the ordered pattern detection algorithm of [DFHP23]. We could, in a similar manner to the current definitions, define the variable $p_{i,v_1,v_2,\ldots,v_{mw(P)}}$ to mean that the subpattern i exists with the anchors $v_1, v_2, \ldots, v_{\text{mw(P)}}$. Here, $mw(P)$ is the merge-width of the pattern as defined in [DFHP23].

4.2.3 CEGAR

Like the approach for transitivity in Section 4.1.4, we can start with not encoding pattern avoidance and only add clauses when needed. This simplifies the problem of finding patterns, due to the fact that we are already given a (partial) order.

We first look at model checking approaches, where we are already given a complete order. Here, we need to identify at least one occurrences $v_1, v_2, \ldots, v_k$ of a pattern if one exists, and add the clause of Equation 4.6 to prevent this occurrence. An approach that works in all cases is to run the known pattern detection algorithm from [DFHP23], which runs in $\mathcal{O}(n^\omega)$ for positive outerplanar patterns (or $\mathcal{O}(n^3)$ in our case due to us not using fast matrix multiplication).

Some patterns do however have faster detection algorithms. We do note, however, that our evaluation uses the $\mathcal{O}(n^3)$ approach. An example for a pattern with faster detection is the pattern P with $k = 4$ vertices, and only the mandatory edges $E_M(P) = \{p_1p_2, p_1p_4, p_3p_4\}$. The pattern P is shown in Figure 4.8. For this pattern, an $\mathcal{O}(n + m)$ time detection algorithm

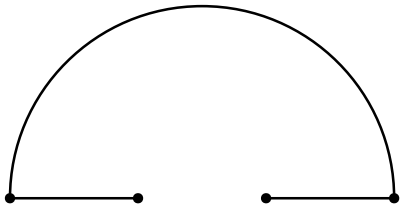

Figure 4.8: An example pattern with an $\mathcal{O}(n + m)$ detection algorithm.

exists. To detect this pattern in an ordered graph $(G, \prec_G)$, calculate for each vertex $v \in V(G)$ the closest vertex to the left $\ell(v)$ and the closest vertex to the right $r(v)$. Then, iterate over all edges $uv \in E(G)$ (where $u \prec_G v$ without loss of generality) and check if $r(u) \prec \ell(v)$. If this is the case, then $u, r(u), \ell(v), v$ is an occurrence of the pattern: The edge uv exists in G

due to the algorithm only considering pairs u, v if $uv \in E(G)$, and the edges $ur(u)$ and $v\ell(v)$ exists due to $r(u)$ and $\ell(v)$ being the closest right and left neighbor of u and v, respectively. Therefore, the algorithm finds an occurrence of P if it exists.

The first part of this algorithm takes $\mathcal{O}(\sum_{v \in V(G)} \deg(v)) = \mathcal{O}(n + m)$ time, due to each vertex needing to iterate over all neighbors. The second part takes $\mathcal{O}(m)$, due to the $\mathcal{O}(1)$ time check for each edge $uv \in E(G)$. This fast detection algorithm can now be used in the CEGAR approach to speed up the time needed by the user propagator to detect the pattern P and add a clause to prevent it.

The second idea for using pattern detection with CEGAR is to check partial assignments. Instead of using this approach to completely ensure pattern avoidance, we only use it to prune branches in the SAT solvers search. We again take as an example the pattern with four vertices and the set of mandatory edges $E_M(P) = \{p_1p_2, p_1p_4, p_3p_4\}$ shown in Figure 4.8. From Lemma 3.5, we know that adding the edge p_1p_3 results in a weaker pattern P'. If the instance graph G can be ordered to avoid P, then G can also be ordered to avoid P'. Inspecting the pattern P' gives us two insights: The vertex p_1 is connected to all remaining vertices, and the remaining vertices $P[2..4]$ are the pattern for a star. Therefore, if $\prec_G$ is an order of G avoiding P', then the right neighborhood of any vertex must be a star. Because P' is a weaker pattern than P, the same rule applies to P as well. This rule can be implemented using user propagators: When a literal $o_{u,v}$ is assigned, add v to the right neighborhood of u. Then, check if the right neighborhood of u is still a star. To be more specific, let $N_{\text{right}}(u)$ be the right neighborhood of u, then we need to check if there is more than one vertex with degree at least 2 in $N_{\text{right}}(u)$. If so, we can add a new clause that at least one of these vertices with degree at least 2 must be to the left of u.

4.3 External strategies

Section 4.1 and 4.2 present the theory needed to solve Pattern-Avoiding Vertex Ordering inside the SAT solver. This section shows additional strategies that happen outside the solving process of the SAT solver. The main motivation for using external strategies are the large encodings produced when solving the Pattern-Avoiding Vertex Ordering problem: From Section 4.1 and 4.2, we know that ensuring transitivity and pattern avoidance may both result in general in $\Omega(n^3)$ clauses, no matter the actual strategy used. With this, even if the SAT solver can show unsatisfiability quickly, the time needed to encode the instance is already large.

This section shows three approaches to prevent this problem. In Section 4.3.1, we first try to solve the problem without the use of the SAT solver. This hopefully especially filters out instances that are large but easy to solve. In Section 4.3.2, we give reduction rules for the Pattern-Avoiding Vertex Ordering problem to decrease the size of the instance graph. Last, in Section 4.3.3, we make use of incremental SAT solving. In incremental SAT solving, we do not give the entire instance to the SAT solver at once, but let it run multiple times, where we can add new clauses between runs. We use the result given by the SAT solver, both the satisfiability and the variable assignment, to find a result without needing to give the solver the entire input.

4.3.1 Filtering

In Section 3.1.3, we discussed the relation of pattern graph classes $\mathcal{G}_P$ to other graph classes. We make use of these relations in a step before starting the main solving process. Assume we have $\mathcal{G}_P \subseteq \mathcal{G}'$, and we can easily check if G is in $\mathcal{G}'$. If we have $G \notin \mathcal{G}'$, we can conclude that $G \notin \mathcal{G}_P$ must also hold. We also note that this check may have a one-sided error: Because $G \in \mathcal{G}'$ makes no statement on G being in $\mathcal{G}_P$ or not, there is problem with the correctness of the filtering approach when falsely determining $G \in \mathcal{G}'$. The other direction of this works as well: If $\mathcal{G}' \subseteq \mathcal{G}_P$ and $G \in \mathcal{G}'$, then $G \in \mathcal{G}_P$ must also be true. These ideas are already used for other problems, for example graph coloring: If a graph has a clique with at least $k+1$ vertices, it cannot be colored using k colors.

In this section, we present five filtering approaches. The first two can identify unsatisfiable instances early, and the latter three can identify satisfiable instances early. We start with the first insight: If $\mathcal{G}_P \subseteq \mathcal{G}'$ and $G \notin \mathcal{G}'$, then $G \notin \mathcal{G}_P$. Here, we have two filters:

- **Clique filter**: Let P be a positive pattern on k vertices, and P_{Clique} the pattern on k vertices where each pair of vertices is connected by a mandatory edge. Note that $G \in \mathcal{G}_{P_{\text{Clique}}}$ if and only if there is no clique of size k in G. Then P_{Clique} always has a superset of the edges of P, and from Lemma 3.5, we know that $\mathcal{G}_P \subseteq \mathcal{G}_{P_{\text{Clique}}}$. Testing if a graph G is in $\mathcal{G}_{P_{\text{Clique}}}$ is generally not fast, because this is exactly solving co-Clique problem. We can however use a heuristic to solve this, and in our case, that is deleting the vertex with the smallest degree until only a clique remains.

- **Local bipartiteness filter**: In Section 3.1.3, we mention a class of patterns where the corresponding graph classes are locally $(k-2)$-colorable in Lemma 3.8. We only make use of this for patterns of size $k = 4$, where we can check for local bipartiteness in $\mathcal{O}(nm)$ time. This might still be in $\mathcal{O}(n^3)$ in very dense cases, but this can prevented by letting the algorithm only run up to, for example, $\mathcal{O}(n^2)$ steps and then terminating. Again, this is only a filtering step, it is not a problem for the correctness of the entire process if this step incorrectly reports that the graph is locally bipartite. Note that the requirement for local bipartiteness does not only apply directly to the patterns used in Lemma 3.8, but also patterns with a subset of edges due to Lemma 3.5. The original pattern of Lemma 3.8 for local bipartiteness and another pattern that follows local bipartiteness are shown in Figure 4.9.

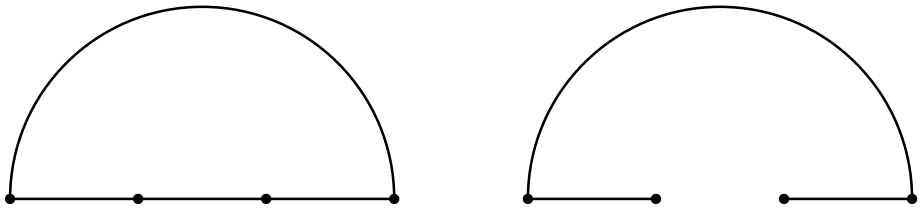

Figure 4.9: On the left is the pattern P where the corresponding graph class $\mathcal{G}_P$ must be locally bipartite, according to Lemma 3.8. On the right is the pattern P' where the corresponding graph class $\mathcal{G}_{P'}$ must also be locally bipartite, due to the left pattern being locally bipartite and Lemma 3.5 giving $\mathcal{G}_{P'} \subseteq \mathcal{G}_P$.

Next, we look at the other direction: If $\mathcal{G}' \subseteq \mathcal{G}_P$ and $G \in \mathcal{G}'$, then $G \in \mathcal{G}_P$. In all three filtering approaches we present here, we use ideas from Section 3.1.3 to transform the input pattern P to a stronger pattern P', where $\mathcal{G}_{P'}$ is known from Section 3.1.2.

- **Bandwidth filter**: Let P be a pattern, and $t = \max\{|i - j| \mid p_i p_j \in E_M(P)\} - 1$ be one less than the length of its longest mandatory edge. We build a new pattern which only contains one such longest edge: First, remove all edges from P except one longest mandatory edge, which gives a stronger pattern following from Lemma 3.5. Let $p_i p_j$ be the remaining mandatory edge. Then, remove all vertices from the pattern that are not between p_i and p_j, which gives a stronger a pattern following from Lemma 3.7. This new pattern P' is exactly the pattern enforcing that the graph has bandwidth at most t. An example of this construction process is shown in Figure 4.10. Therefore, if we can show that the graph has bandwidth at most bt, then it can be ordered to avoid P.

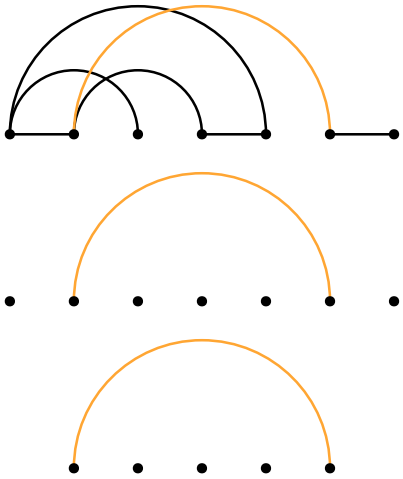

Figure 4.10: An example of the way the bandwidth filter modifies the pattern. At the top is the original pattern. The first step is the removal of all edges except one longest edge, highlighted in orange in all steps. The second step is the removal of all vertices outside the remaining edge. If a graph has bandwidth at most 3, it can be reordered to avoid the pattern at the top.

Now, we want to figure out if a given graph has low bandwidth. While extensive research on the Bandwidth Minimization problem exists, it is most often treated as an optimization problem, for example in [CM69] and [MUPG10], while we have the decision variant. We can make explicit use of the fact that the target bandwidth is given to us: First, use the Cuthill-McKee algorithm [CM69] to minimize the bandwidth. If this heuristic already gives a bandwidth of at most t, we are done. Otherwise, we use the following local search: Scan over all vertices, and for each vertex v, find the longest edge to the left, where we call the other endpoint of this edge v_ℓ, and to the right, where we call the other endpoint of this edge v_r. If $\pi(v_r) - \pi(v_\ell) > 2t$, then the outer vertices are so far apart that the moving v cannot make all incident edges short enough. In this case, we randomly move either v_ℓ to the right or v_r to the left by swapping it with the neighboring vertex. Otherwise, we move v and one of the outer vertices closer together is needed: For $\pi(v) - \pi(v_\ell) > t$, we randomly move either v_ℓ to the right or v to the left, and for $\pi(v_r) - \pi(v) > t$, we randomly move either v to the right or v_r to the left. To keep this local search short, only $\mathcal{O}(\log(n))$ iterations are done, with each iteration taking $\mathcal{O}(n)$ time.

- **Graph Coloring filter**: Let P be a pattern, and let $X = \{p_{i_1}p_{i_2}, p_{i_3}p_{i_4}, \ldots, p_{i_{2t-1}}p_{i_{2t}}\}$ be a sequence of mandatory edges with two properties: For all $j \in [2t-1]$, we have $i_j \leq i_{j+1}$, and second, t is as large as possible. Remove all edges from P except for those X, which gives a stronger pattern following from Lemma 3.5. Then, remove all vertices p_x if an edge p_ip_j still exists with $i < x < j$. This gives a stronger pattern according to Lemma 3.6. After these two steps, the pattern is a chain of patterns with two vertices, some with and some without connecting edge. In the last step, we reorder this pattern chain so that all patterns with a connecting edge come first, which describes the same graph class following Lemma 3.1, and then delete all the isolated vertices at the end of the pattern, giving us a stronger pattern according to Lemma 3.7. An example of this construction process is shown in Figure 4.11. The final pattern is exactly the one for the problem t-Coloring.

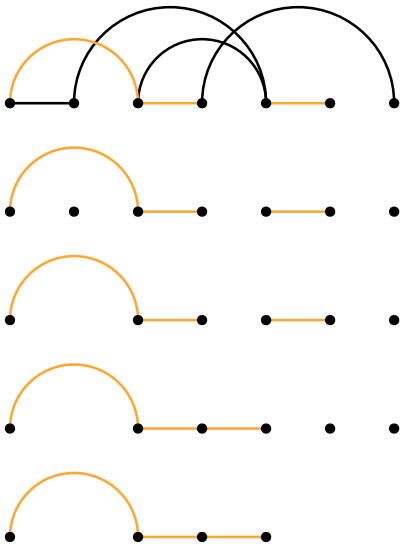

Figure 4.11: An example of the way the coloring filter modifies the pattern. At the top is the original pattern. The first step is the removal of all edges except the ones in X, highlighted in orange in all steps. The second step is the removal of all vertices under any remaining edge. The third step is the reordering of the chained edges. The fourth step is the removal of all isolated vertices at the right. If a graph can be colored with three colors, it can be reordered to avoid the pattern at the top.

The size t of the set X can be calculated using dynamic programming. Let x_i be the size of the largest set X of the suffix of the pattern $P[i..k]$. The value we are looking for is $t = x_1$. For simplicity, we define $x_{k+1} = 0$. Now, for each vertex p_i, there are two options: First is taking an outgoing edge p_ip_j to the right into the set X, giving $x_i = x_j + 1$ because it is optimal to extend the solution for x_i starting from p_j with the optimal solution for x_j. Second is not taking any edge outgoing from p_i, in which case we take the solution x_{i+1}. This gives the following recurrence:

$$x_i = \max(\max\{x_j + 1 \mid i < j \wedge p_ip_j \in E_M(P)\}, x_{i+1})$$

This can be evaluated in $\mathcal{O}(n+m)$ time by iterating i from k down to 1, and calculating x_i in each step. For each x_i, we need $\mathcal{O}(\deg(p_i))$ time, summing to $\mathcal{O}(m)$ steps in the maximum.

With t calculated, we now only need to figure out if the graph G is t-colorable. For this, we use DSATUR [Bré79]. If DSATUR manages to find a t-coloring, then an ordering avoiding P exists. The DSATUR algorithm runs in $\mathcal{O}((n+m)\log(m))$.

- **Vertex Cover filter**: This filter uses the shortest edge, as opposed to the longest used in the bandwidth filter. From there, we modify the pattern to the pattern for the Vertex Cover problem. Let P be a pattern, and let $t = k - (\min\{|i-j| \mid p_i p_j \in E_M(P)\} + 1)$. Note that $\min\{|i-j| \mid p_i p_j \in E_M(P)\}$ is the length of the shortest mandatory edge of P. First, remove all edges from P except one shortest mandatory edge, which gives a stronger pattern following from Lemma 3.5. Then, remove all vertices from the pattern that are between p_i and p_j, which gives a stronger a pattern following from Lemma 3.6. The remaining pattern is a chain of patterns with two vertices, once with a connecting mandatory edge and t times without an edge. An example of this construction process is shown in Figure 4.12. This is the pattern for a Vertex Cover of size t.

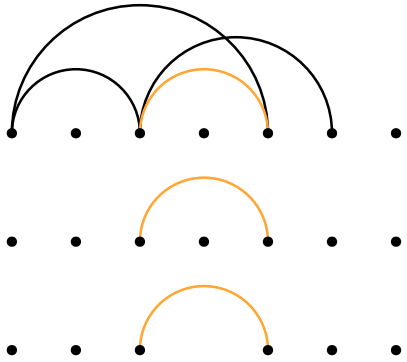

Figure 4.12: An example of the way the vertex cover filter modifies the pattern. At the top is the original pattern. The first step is the removal of all edges except the shortest one, highlighted in orange in all steps. The second step is the removal of all vertices under the shortest edge. If a graph has a vertex cover of size 4, it can be reordered to avoid the pattern at the top.

Fast FPT algorithms are known for Vertex Cover, with the most recent one running in $\mathcal{O}(1.25284^t \cdot \text{poly}(n))$ time given in [HN24]. Due to k already being very small in our case, we simply use an $\mathcal{O}(2^t \cdot m)$ time algorithm: For each edge $uv \in E$ of the graph, branch over taking u or taking v into the vertex cover. Then, after taking x, remove x and all incident edges from the graph. This results in at most 2^t possible choices, and removing a vertex and incident edges takes at most $\mathcal{O}(m)$ time.

4.3.2 Preprocessing

In this section, we focus on steps to simplify the instance given to the SAT solver. In particular, we note that for some patterns P, there are some vertices of the instance graph G that can be removed without changing whether G can be ordered to avoid P or not. This kind of reduction rule is already known for other problems, an example is part of the preprocessing shown in [HN24] for Vertex Cover. In the remainder of this section, we state the two Lemmas 4.9 and 4.10, and use them to give two reduction rules.

Lemma 4.9: *Let P be a positive pattern where $p_i p_{i+1} \in E_M(P)$ exists for each $i \in [k-1]$. Let G be a graph, and $u, v \in V(G)$ two vertices with $uv \notin E(G)$ and $N(u) \subseteq N(v)$. Then G can be ordered to avoid P if and only if $G' = G[V \setminus \{u\}]$ can be ordered to avoid P.*

Proof. If there exists an ordering $\prec_G$ of G avoiding P, then G' can be ordered to avoid P using the same order restricted to the vertices of G'.

In the other direction, let $\prec_{G'}$ be an order of G' avoiding P. We extend this order to an order $\prec_G$ of G by placing u directly before v. Any two vertices already present in G' retain their relative order. Any element before v is also before u, and any element after v is also after u. Last, u comes before v. This gives the following formal definition of $\prec_G$:

$$x \prec_G y \iff \begin{cases} x \prec_{G'} y & x \neq u \wedge y \neq u \\ v \prec_{G'} y & x = u \wedge y \neq v \\ x \prec_{G'} v & y = u \\ \top & x = u \wedge y = v \\ \bot & x = v \wedge y = u \end{cases}$$

Assume G ordered with $\prec_G$ has an occurrence of the pattern P. Because G' ordered with $\prec_{G'}$ does not have an occurrence of P, the added vertex u must be contained in the pattern. This also means that v is not contained in the occurrence of P. If it were contained in the occurrence, it would have to come directly after u, but u and v are not connected by an edge, while two consecutive vertices in the pattern are always connected by an edge. Let $v_1, v_2, \ldots, v_{i-1}, u, v_{i+1}, \ldots, v_k$ be the sequence of vertices forming the occurrence of the pattern. The sequence where u is replaced by v, namely $v_1, v_2, \ldots, v_{i-1}, v, v_{i+1}, \ldots, v_k$, must now also be an occurrence of the pattern: By the requirement of Lemma 4.9, the pattern P is positive and we have $N(u) \subseteq N(v)$. Therefore, substituting u with v only adds more edges, which never removes an occurrence of a positive pattern. This would mean that $\prec_{G'}$ also has an occurrence of the pattern P, contradicting the requirement for $\prec_{G'}$. ∎

Lemma 4.9 gives us our first preprocessing rule: If P is a positive pattern where $p_i p_{i+1} \in E_M(P)$ exists for each $i \in [k-1]$, remove all vertices $u \in V(G)$ if another vertex $v \in V(G)$ with $uv \notin E(G)$ and $N(u) \subseteq N(v)$ exists. To improve the speed of this check, we use signatures: For each vertex v, we use a bitset B_v of some fixed length ℓ. Then, for every neighbor $x \in N(v)$, the bit at position $x \bmod \ell$ in B_v is set to 1. If $N(u) \subseteq N(v)$, then the 1-bits in B_u must also be a subset of the 1-bits in B_v. A fast check for this with bitsets is testing for $B_u \& B_v = B_u$. With this, we have the complete description of the domination rule.

The second rule we are looking at removes vertices of small degree when possible. Intuitively, if the first vertex is a positive pattern P has degree $\deg(p_1)$, and a vertex $v \in V(G)$ has a lower degree $\deg(v) < \deg(p_1)$, then v can never be the first vertex in an occurrence of P. Simply placing v at the very front now makes sure it never can be part of an occurrence of P at all.

Lemma 4.10: *Let P be a pattern where the first vertex p_1 is incident to $\deg_M(p_1)$ mandatory edges, and the last vertex p_k is incident to $\deg_M(p_k)$ mandatory edges. Let G be a graph, and u a vertex with $\deg(u) < \max(\deg_M(p_1), \deg_M(p_k))$. Then G can be ordered to avoid P if and only if $G' = G[V \setminus \{u\}]$ can be ordered to avoid P.*

Proof. The first direction is the same as in Lemma 4.9: If there exists an ordering $\prec_G$ of G avoiding P, then G' can be ordered to avoid P using the same order restricted to the vertices of G'.

For the other direction, we first show this for $\deg(u) < \deg_M(p_1)$. Let $\prec_{G'}$ be an order avoiding P in G'. We extend this to an ordering of G by placing the removed vertex u at the very front:

$$x \prec_G y \iff \begin{cases} x \prec_{G'} y & x \neq u \wedge y \neq u \\ \top & x = u \\ \bot & y = u \end{cases}$$

Assume G ordered with $\prec_G$ has an occurrence of the pattern P. Because G' ordered with $\prec_{G'}$ does not have an occurrence of P, the added vertex u must be contained in the pattern. Because u is at the very front, it must be the first vertex in the occurrence of P. This is not possible due to $\deg(u) < \deg_M(p_1)$, because u cannot have all the vertices as neighbors that take on the roles of the pattern vertices $N(p_1)$.

The proof in the case where $\deg(u) < \deg_M(p_k)$ is similar. The only difference is that we place u at the very back instead of the front when extending the ordering. ∎

Lemma 4.9 gives us our second preprocessing rule: While there exists a vertex $v \in V(G)$ with $\deg(u) < \max(\deg_M(p_1), \deg_M(p_k))$, remove it. For the following explanation, let $t = \max(\deg_M(p_1), \deg_M(p_k))$. For this removal process, initially all vertices with degree below t are put into a removal queue. While this queue is not empty, a vertex v is taken from the queue and removed from the graph, decrementing the degrees of the neighbors $N(v)$. If the degree of a neighbor $u \in N(v)$ also falls below t, then u is also added to the removal queue. This is repeated until the removal queue is empty, which takes at most $\mathcal{O}(n + m)$ time.

4.3.3 Incremental solving

In Section 4.3.1, we tried to avoid the usage of the SAT solver at all. In this section, we instead attempt to only give partial instances to the SAT solver. For this, we make use of incremental SAT solving. Incremental SAT solving allows us to solve one SAT instance, then add additional variables and clauses, and solve this larger SAT instance using for example learned clauses from the previous run.

Lemma 3.11 states that every graph class $\mathcal{G}_P$ based on a pattern P is hereditary. Therefore, finding any induced subgraph $G' \sqsubseteq G$ that cannot be ordered to avoid P implies that all of G cannot be ordered to avoid P. This gives us a first idea of how to apply incremental solving: Choose a chain of induced subgraphs $G'_1 \sqsubseteq G'_2 \sqsubseteq \cdots \sqsubseteq G'_t = G$. Then, encode the Pattern-Avoiding Vertex Ordering problem for G'_1 only. If the SAT solver gives unsatisfiable as the result, then we know that result for G is also unsatisfiable and we can stop. Otherwise, if we just solved the problem G'_i for $i < t$, we add additional encoding to solve the problem on G'_{i+1} next. This process is repeated until either a step returns unsatisfiable and we know the result to be unsatisfiable on G, or we add all of G and get satisfiable as the result. We now have a strategy which still needs to encode the entire problem for satisfiable instances, but can potentially exit a lot earlier on unsatisfiable instances. We call this approach `subgraph-incremental`.

The description of this strategy does leave one question open: How exactly do we find this chain of induced subgraphs $G'_1 \sqsubseteq G'_1 \sqsubseteq \cdots \sqsubseteq G'_t = G$? In the optimal case, we have G'_1 as the smallest obstruction in G if the graph is an unsatisfiable instance, and $t = 1$ if the graph is a satisfiable instance. This approach, however, is not practical, because it comes down to finding an obstruction, which would already solve the Pattern-Avoiding Vertex Ordering problem. Instead, we want to heuristically determine the chain in such a way that, if an obstruction

exists, it is likely to be part of the earlier induced subgraphs in the chain. We split this task into two parts: First, we determine a priority order $<_G$ on the vertices of G, and then split the vertices in this order into multiple chunks. Note that this priority order $<_G$ is separate from the order $<_G$ avoiding P we are trying to find in Pattern-Avoiding Vertex Ordering problem. Formally, let $<_G$ be the priority order, where $\rho(i)$ is the vertex in the ith position of this order. Let $(c_1, c_2, \ldots, c_t)$ with $c_i < c_{i+1}$ and $c_t = n$ be the chunk sizes. Then the vertices of the ith subgraph are:

$$V'_i = \{\rho(j) \mid j \leq c_i\}$$

For determining the priority order, we have four strategies. First is `bfs`, which simply uses a breadth-first search. If G has multiple components, one is fully explored before the BFS is started again at the next. The vertices are placed in the priority order in the same order the BFS discovers them. Second is `degenerate`, which takes a vertex $v \in V(G)$ with the lowest degree, adds v to the front of the priority order, and then deletes v from G. This process is repeated until no vertex is left. Importantly, note that the last vertex deleted in the process comes first in the priority order, and is therefore always part of G'_0. Third is `random`, which takes a random priority order. Fourth is `sampling`. In this last strategy, we first randomly shuffle the graph multiple times, and start the pattern detection algorithm each time. For each vertex v, we record how many times it occurs in a pattern, which is its weight $w(v)$. This does not need to be exact, the only goal is to identify which vertices are likely difficult to place in an actual order. Then, we take the vertex v with the highest weight, place it at the back of the priority order $<_G$, delete it from G, and increase the weight of each neighbor by $w(v)/\deg(v)$. The redistribution of weight helps to increase connectivity: Having a lot of vertices that had a high initial weight, but are all disconnected from each other, is not going to result in a difficult subgraph. The choice of dividing by $\deg(v)$ is to maintain the sum of weight in the graph.

Next, we come to determining the chunk sizes $(c_1, c_2, \ldots, c_t)$, where we use three different strategies. First is `chunks`, where we divide the n vertices into 16 chunks of equal size. The concrete chunks sizes are $c_i = \lfloor n/16 \rfloor$ and $t = 16$. Second is `k-exponential`, where we start with $2k$ vertices, and double the amount of vertices in each step, giving us $c_i = \min(k \cdot 2^i, n)$. This starts out with very small encodings to quickly solve simple unsatisfiable instances, but then scales up quickly when it becomes more likely that there is an ordering avoiding P. Last is a more extreme version, called `small-to-all`, which uses $c_1 = 4k$ and $c_2 = n$.

So far, the incremental solver can only exit early when it finds a part of the instance graph G that cannot be ordered to avoid P, and that can only happen if such a part even exists. Additionally, we need to make sure that the heuristics used for finding the chain of induced subgraphs actually manages to find difficult parts of the graph. We give a high-level description of another strategy for incremental SAT solving, called `heuristic-incremental`, that addresses both of these points: Instead of starting with the SAT solver, try to find an ordering $\prec$ using some heuristic. When the heuristic ordering fails, find a set of vertices $V'_1 \subseteq V$ such that, when V'_1 is removed, we have a valid ordering $\prec$ on $V \setminus V'_1$. Then, use the SAT solver to find a valid order on the subgraph induced by these removed vertices $G[V'_1]$. From there, restart the heuristic solver, but fix the relative order of the vertices V'_1 to the one the SAT solver found. Should the heuristic solver again fail to find a valid order, extend the set V'_1 to a new set of removed vertices $V'_2 \supset V'_1$ such that the found order $\prec$ is valid on $V \setminus V'_2$. Restart the SAT solving process with $G[V'_2]$. Due to $V'_2 \supset V'_1$, we know that $G[V'_1] \sqsubseteq G[V'_2]$ and that we can use incremental solving. From here, we repeat the process of extending the

set of removed vertices V_i' until either the heuristic solver finds a valid solution, in which case we have a satisfiable instance, or the SAT solver tells us that we have an unsatisfiable instance.

We present a heuristic for `heuristic-incremental` that works for any positive pattern P if $p_1p_2, p_1p_k, p_{k-1}p_k \in E_M(P)$. Let $I \subseteq V$ be an independent set. Let $G' = G[V \setminus I]$, and $\prec_{G'}$ an order on G' avoiding P. Now, we partition the independent set into $I = I_\ell \cup I_r$, and place all of I_ℓ left of the vertices G' and all of I_r right of the vertices of G'. Note that, if there is an occurrence of the pattern P, exactly one vertex of I is part of it. At least one vertex of v must be part of this occurrence due to the fact that G' is ordered to avoid P. Assuming that at least two vertices I_ℓ would mean that the first two vertices of the occurrence come from I_ℓ, but these vertices are not connected by an edge as required by $p_1p_2 \in E_M(P)$. Likewise, $p_1p_k \in E_M(P)$ prevents at least one vertex from I_ℓ and one from I_r being part of an occurrence of P, and $p_{k-1}p_k \in E_M(P)$ prevents two vertices from I_r being part of an occurrence of P. We can therefore attempt to place each vertex $v \in I$ independently to the left and to the right, and if both fail, we add v to the set of vertices the SAT solver places.

This approach can even be applied recursively: We take out multiple independent sets $I_1, I_2, \ldots, I_t$ from G, and let the SAT solver order $G[V \setminus \bigcup_{i\in[t]} I_i]$. Then, we use the idea above to first place the vertices of I_t back into the graph, then I_{t-1}, and so on. If, at any point, we fail to place all vertices in a layer, the unplaced vertices are added to the SAT solver, we clear all vertex placements in all layers, and restart with a SAT solving run. Otherwise, after managing to place the vertices of I_1, we are done and have a satisfiable instance. Note that the idea of `subgraph-incremental` also still applies: If the SAT solver reports in any iteration that the instance is unsatisfiable, even when only given a subset of the vertices, we know that the entire graph is also an unsatisfiable instance.

With this, it remains to decide on the number of layers in the independent set removal process. Here, we use three strategies. First is `k-minus-1`, which uses $t = k - 1$ layers. Second is `half-graph`, which removes an independent set until at least $n/2$ vertices are removed. Third is `full-graph`, which removes an independent set until the graph is empty.

This wraps up the incremental approaches used, and also the entirety of the approaches we attempted to solve the Pattern-Avoiding Vertex Ordering problem. The upcoming Section 4.4 is another section on theory, but this time from a SAT solving perspective. There, we look into the complexity of resolution proofs, which are connected to the best-case performance of SAT solvers.

4.4 Notes on resolution proofs

In this section, we take a step back from the practical implementations of different ideas and evaluate the underlying ideas in a more theoretical way. It is known that conflict-driven clause learning, as used by CaDiCaL, is as powerful as general resolution [PD11]. This means that the number of decisions the solver needs to have until finding out that an instance is unsatisfiable is at least the length of the shortest resolution proof. This connection between the SAT solver performance and the length of resolution proofs gives us reason to look into resolution proofs. We start by showing that is never necessary to resolve two cyclic clauses in a resolution proof in Lemma 4.11.

Lemma 4.11: *Consider an unsatisfiable instance with an encoding which only uses ordering variables. There exists a resolution proof which does not resolve any pair of transitivity clauses, and is at most a factor $n - 1$ longer than the shortest resolution proof.*

Proof. The last resolution step of a resolution proof results in the empty clause, which is acyclic. On the other hand, resolving two cyclic clauses does result in a cyclic clause. Transitivity clauses are cyclic. This means, if we at any point resolve two transitivity clauses, there must be a step between this resolution and the final step where resolve a cyclic clause with an acyclic clause.

Let A be an acyclic clause and C_1 and C_2 be cyclic clauses such that the shortest resolution proof somewhere first resolves C_1 and C_2 over y, and the result with A over x. We call the result of these resolutions $R_1 = A \otimes_x (C_1 \otimes_y C_2)$. For the second resolution to be possible, we need to have $\overline{x} \in C_1 \otimes_y C_2$, so $\overline{x}$ needs to be already present in C_1 or C_2. Without loss of generality, let $\overline{x} \in C_1$. We now show that we can change these two resolutions in the resolution proof to first resolving A with C_1, and the result with C_2. We call the result of these resolution $R_2 = (A \otimes_x C_1) \otimes_y C_2$. This has two parts: First, we need to make sure that this modified sequence of resolutions is still possible. Second, we argue that the resulting clause of the changed steps is the same or a stronger clause than the one resulting of the original steps, so the rest of the proof is still possible. The second requirement is equivalent to $R_2 \subseteq R_1$.

We show $R_2 \subseteq R_1$ first. We know that none of the clauses contain opposite literals x and $\overline{x}$, as such a clause would be a tautology that cannot be part of a resolution proof. This means that for any two clauses X and Y resolved over the variable t, we have $X \otimes_t Y = (X \cup Y) \setminus \{t, \overline{t}\}$. With this, the subset relation between these two orders of operations is explained by the following:

$$\begin{aligned} R_2 &= (A \otimes_x C_1) \otimes_y C_2 \\ &= (A \cup C_1 \cup C_2) \setminus \{x, \overline{x}, y, \overline{y}\} \\ &\subseteq (A \cup ((C_1 \cup C_2) \setminus \{y, \overline{y}\})) \setminus \{x, \overline{x}\} \\ &= A \otimes_x (C_1 \otimes_y C_2) \\ &= R_1 \end{aligned}$$

In a complete resolution proof, we can now replace the resolution steps $A \otimes_x (C_1 \otimes_y C_2)$ producing R_1 with the steps $(A \otimes_x C_1) \otimes_y C_2$ producing R_2. From there, we can weaken R_2 back to R_1, and the rest of the resolution proof does not need to be modified.

This leaves to argue that the changed order still gives valid resolutions. Due to $\overline{x} \in C_1$, the first step is valid. We also know that $y \in C_1$ and $y \notin \{x, \overline{x}\}$, which means that the literal y is still present in $A \otimes C_1$ and we can resolve this with C_2.

Doing this changed order of operations might change the length of the proof, because the clause resulting from $C_1 \otimes C_2$ might be reused for multiple resolutions. The last claim to show is that this increase is by at most a factor $n - 1$. To see this, let $C \otimes A$ be a resolution of a cyclic clause C and an acyclic clause A in the shortest proof. We now ignore all steps required to construct C and temporarily assume C to be an axiom, which shortens the proof. If C is used in multiple axioms, we provide a copy of C for each of them. Next, we reduce C to a clause containing a single cycle C' to make sure it does not contain additional literals apart from the cycle. This strengthening of C to C' only shortens the proof further. We now have a resolution proof that might not be valid from the original axioms anymore, but is in any case shorter. From this shortened proof, we now reconstruct a valid proof which is at most $n - 1$ times as long, does not resolve two cycle clauses, and is valid from the original axioms.

The first step of this reconstruction is to add steps creating each cycle clause C' from only transitivity clauses, which is shown in Lemma 4.2. This makes sure that we are now only using the original set of axiom clauses. Because this construction requires up to $n - 2$ steps,

the single resolution of $C \otimes A$ might have turned into up to $n-1$ resolutions, counting both the construction of C and its final resolution with A. The second step is to apply the cycle resolution prevention shown above, which does not modify the proof length. ∎

Knowing that we never need to resolve two transitivity clauses leads us to investigate resolution of transitivity and acyclic clauses. Let $T = \{o_{u,v}, o_{v,w}, \overline{o}_{u,w}\}$ be a transitivity and C an acyclic clause, where we can resolve T and C. Without loss of generality, let $o_{u,w} \in C$, so we can resolve over $o_{u,w}$. We also require that $\overline{o}_{u,v}, \overline{o}_{v,w} \notin C$, so we do not produce a tautology. This resolution gives the following resulting clause:

$$T \otimes_{o_{u,w}} C = (C \cup \{o_{u,v}, o_{v,w}\}) \setminus \{o_{u,w}\} \tag{4.13}$$

Before we come to the ways this resolution can look in detail, we give a general insight for the set of clauses prohibited by $T \otimes C$. From Lemma 4.1, we know that the set of orderings prohibited by any clause C is exactly the reversed topological orderings $T^R(G_C)$ of its clause graph G_C. We first show that the set of prohibited orderings of C cannot gain new elements when resolving with a transitivity clause in Lemma 4.12.

Before we show the formal proof, we note that we can also informally argue $T^R(G_{T\otimes C}) \subseteq T^R(G_C)$ by using a property of resolution: Resolving two clauses gives a clause which is a logical consequence of the first two clauses. We know T does not prohibit any orderings. Therefore, resolving with T should not make it possible to add more prohibited orderings to C, only remove them.

Lemma 4.12: *Let $T = \{o_{u,v}, o_{v,w}, \overline{o}_{u,w}\}$ be a transitivity clause and C an acyclic clause that can be resolved with T. Without loss of generality, let $o_{u,w} \in C$. Then we have $T^R(G_{T\otimes C}) \subseteq T^R(G_C)$.*

Proof. We give a proof by contradiction. Assume there is an ordering $\prec$ which is in $T^R(G_{T\otimes C})$, but not in $T^R(G_C)$. For $(\prec) \notin T^R(G_C)$, there must be some edge $xy \in E(G_C)$ such that $x \prec y$. From Equation 4.13, we know that all edges except uw are also present in T^R. Because $(\prec) \in T^R(G_{T\otimes C})$, it follows that $y \prec x$ if $xy \neq uw$. This leaves to show that $w \prec u$. Again from Equation 4.13, we know that $uv, vw \in E(G_{T\otimes C})$. Because $(\prec) \in T^R(G_{T\otimes C})$, it follows that $v \prec u$ and $w \prec v$, which gives us $w \prec u$. ∎

Next, we come to a detailed look at how these resolutions can look like. There are four different cases for $T \otimes C$, depending on whether $o_{u,v}$ and $o_{v,w}$ are in C. We formally list these cases in Table 4.2, and all four cases are shown in Figure 4.13.

Case	$o_{u,v} \in E$	$o_{v,w} \in E$
1	yes	yes
2	yes	no
3	no	yes
4	no	no

Table 4.2: The definition for the four different ways a resolution between a transitivity clause and an acyclic clause can look like.

In the first case, we have $o_{u,v}, o_{v,w} \in C$, and so $T \otimes C = C \setminus \{o_{u,w}\}$. Here, we also have $T^R(G_C) \subseteq T^R(G_{T\otimes C})$, due to the fact that removing an edge only removes a constraint from the topological orderings. Together with Lemma 4.12, this gives us $T^R(G_C) = T^R(G_{T\otimes C})$ in this case.

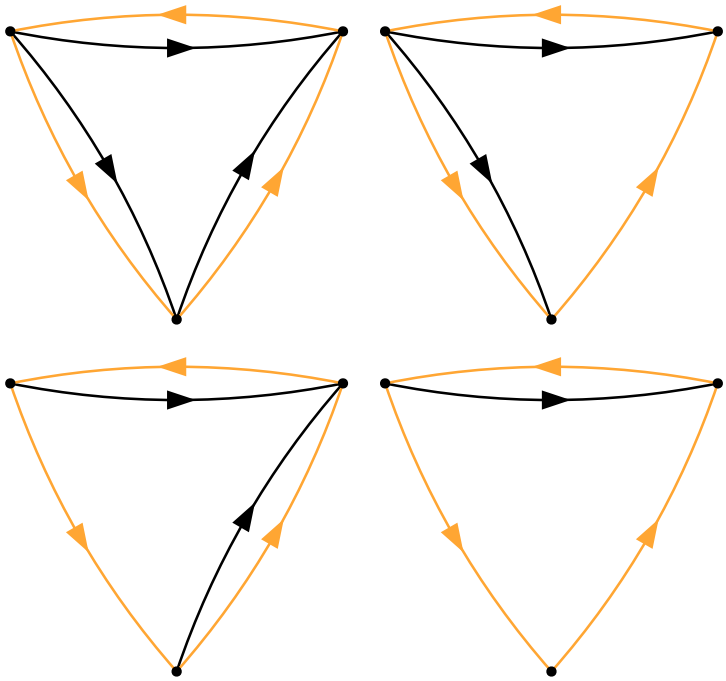

Figure 4.13: The four different ways a resolution between a transitivity clause and an acyclic clause can look like. The edges of the acyclic clause G_C are in black, the edges of the transitivity clause graph G_T are in orange. The graph G_C may contain more edges than depicted here, as long as one endpoint is not u, v, or w.

The first case produces a clause with one less variable than C, while the second and third keep the number of variables the same. Only the fourth case increases the number of variables, which raises the question whether we even need to perform resolutions of the fourth case. For general ordering problems, the answer is yes. To show this, we make use of Ordered Linear 2SAT, which is NP-complete as proven in Lemma 4.7. This SAT problem only has transitivity clauses and clauses with at most two variables. Resolving two clauses with at most two variables results in another clause with at most two variables. Additionally, when the resolution with transitivity clauses is not allowed to increase the number of variables, then any resolution with a transitivity clause also results in a clause with at most two variables. Last, Lemma 4.11 already shows that no resolution between two transitivity clauses is needed. With these insights, we can simply exhaustively resolve clauses with each other. There are $\mathcal{O}(n^2)$ ordering variables, and with that, at most $\mathcal{O}(n^4)$ clauses with at most two variables. When a new clause is created via resolution, we resolve it with all previous clauses, giving a final $\mathcal{O}(n^8)$ time for this exhaustive resolution. This is a polynomial time algorithm for a problem we have already shown to be NP-complete, and is not possible assuming P $\neq$ NP. The only assumption that led us to this algorithm is that the fourth resolution case is never needed, and is therefore false unless P = NP.

From Lemma 4.12, we know that resolving an acyclic clause with a transitivity clause cannot add to the set of prohibited orderings by the acyclic clause, only remove from it. The informal argument given for this is that resolution cannot add new prohibited orderings out of nowhere, they need to be a consequence of resolved clauses. We can extend this argument to the resolution of two acyclic clauses.

Lemma 4.13: *Let C and D be two acyclic clauses that can be resolved over the ordering variable $o_{u,v}$. Without loss of generality, let $o_{u,v} \in C$ and $\overline{o}_{u,v} \in D$. Then we have $T^R(G_{C\otimes D}) \subseteq T^R(G_C) \cup T^R(G_D)$.*

Proof. Let $(\prec) \in T^R(G_{C\otimes D})$ be an order. The graph $G_{C\otimes D}$ contains all edges of G_C and G_D, except for uv and vu. Due to this, the only possible reason for $(\prec) \notin T^R(G_C)$ or $(\prec) \notin T^R(G_D)$ is that the vertices u and v are incorrectly ordered. If $v \prec u$, then $(\prec) \in T^R(G_C)$. Otherwise, $(\prec) \in T^R(G_D)$. Therefore, $(\prec) \in T^R(G_{C\otimes D})$ implies $(\prec) \in T^R(G_C)$ or $(\prec) \in T^R(G_D)$, and so we have $T^R(G_{C\otimes D}) \subseteq T^R(G_C) \cup T^R(G_D)$. ∎

With Lemma 4.13, we conclude our investigation into the effect of resolution on the set of prohibited orderings. Next, we look at the minimum size of the resolution proof when we have an unsatisfiable instance. Our strategy to tackle this is by giving a lower bound on the number of axioms needed. In Theorem 4.14, we show that for any pattern with k vertices, at least $\max(k!, n/(k-1))$ axioms are needed. The phrasing of Theorem 4.14 is more general, however, applying this lower bound for all ordering problems.

Theorem 4.14: *Let $\mathcal{F}$ be an unsatisfiable SAT instance with only ordering variables for n elements, and consisting of all transitivity and some linear clauses, where the linear clauses have at most $k-1$ literals. Further, assume that after removing any element x of the n elements together with all ordering variables x appears in results in a satisfiable instance. Let K_L be the set of linear clauses in $\mathcal{F}$. Then $|K_L| \geq \max(k!, n/(k-1))$.*

Proof. First, we show that there must be at least $k!$ linear clauses. Every linear clause prohibits exactly $n!/k!$ orderings. To show this, we first note that every linear clause with at most $k-1$ literals touches at most k elements to order. These up to k elements have a fixed order. We can therefore construct a forbidden ordering by choosing the positions of these k elements, and then choosing any permutation for the remaining $n-k$ elements. The total number of forbidden orderings is counted by the following:

$$\begin{aligned}\binom{n}{k} \cdot (n-k)! &= \frac{n!}{k! \cdot (n-k)!} \cdot (n-k)! \\ &= \frac{n!}{k!}\end{aligned}$$

For resolution of two acyclic clauses, we know from Lemma 4.13 that for any two acyclic clauses C and D, we have $T^R(G_{C\otimes D}) \subseteq T^R(G_C) \cup T^R(G_D)$. For resolution involving transitivity clauses, we know from Lemma 4.11 that we never need to resolve two transitivity clauses, and when resolving a transitivity clause T with an acyclic clause C, we have $T^R(G_{T\otimes C}) \subseteq T^R(G_C)$ from Lemma 4.12. From all these insights, we know that the orderings prohibited by any acyclic clause C derived during a resolution proof is at most the union of the orderings prohibited by the clauses used to derive C. A resolution proof ends with the empty clause, which prohibits all $n!$ orderings, and this derivation may use all clauses given in K_L. This gives us the following bound on the size of K_L:

$$\begin{aligned} n! &= |T^R(G_\emptyset)| \\ &= |\bigcup_{C\in K_L} T^R(G_C)| \\ &\leq |K_L| \cdot \frac{n!}{k!} \\ \implies |K_L| &\geq k! \end{aligned}$$

In the second part, we show that there must be at least $n/(k-1)$ clauses in K_L. For this, we look at the union of all clause graphs of all linear clauses G_{K_L}:

$$V(G_{K_L}) = [n]$$
$$E(G_{K_L}) = \bigcup_{C \in K_L} E(G_C)$$

Assume there is a vertex x of this union of clause graphs with no outgoing edge. We want to show that such a vertex x cannot exist. Let $\mathcal{F}'$ be the instance $\mathcal{F}$ with all ordering variables containing x and all clauses containing one of these ordering variables removed, and let ϕ' be a satisfying assignment for $\mathcal{F}'$. This assignment ϕ' exists due to the requirement set in Theorem 4.14, and it can be extended to a satisfying assignment ϕ for all of $\mathcal{F}$ by placing the element x last:

$$\phi(o_{u,v}) = \begin{cases} \bot & u = x \\ \top & v = x \\ \phi'(o_{u,v}) & \text{otherwise} \end{cases}$$

To show that this satisfies $\mathcal{F}$, we need to show that all clauses $K \setminus K'$ removed when transforming the instance to $\mathcal{F}'$ are satisfied. All remaining clauses K' are part of $\mathcal{F}'$ and therefore already satisfied by ϕ'. Let $C \in K \setminus K'$ be any such clause. If C is a transitivity clause, then it is satisfied due to the fact that ϕ gives a valid ordering on all n elements. Otherwise, C is a linear clause, and the variable $o_{v,x}$ is in the clause for some $v \in [n]$. We know the clause must contain $o_{v,x}$ or $o_{x,v}$, otherwise it would have been removed, and it cannot be $o_{x,v}$, as this would result in an outgoing edge from x, which does not exist by assumption. This assumption now leads to a contradiction, because there cannot be a satisfying assignment ϕ for $\mathcal{F}$.

From this, we know that there may not be a vertex without an outgoing edge in G_{K_L}. A similar argument shows that there may not be a vertex with no incoming edge, with the most noteworthy difference that we place the vertex at the very start when transforming ϕ' to ϕ. This means that every vertex must have at least two incident edges. On the other hand, every edge is incident to two vertices. With this, we know that there must be at least n edges in G_{K_L}. Every linear clause has at most $k-1$ literals, and so can contribute at most $k-1$ edges. We must therefore have at least $n/(k-1)$ linear clauses in an unsatisfiable instance. ∎

Theorem 4.14 gives a lower bound on the number of axioms and therefore implicitly a lower bound of $\max(k!, n/(k-1)) - 1$ on the number of steps needed by the shortest resolution proof. Next, we show a resolution proof for bipartiteness. The minimal graphs that are not bipartite are odd cycles. Therefore, let n be odd and G a cycle of length n with vertices $v_1, v_2, \ldots, v_n$. To prevent clutter with moduli in the vertex indices, the index of the vertices implicitly wraps to 1 after n. For example $v_1 = v_{n+1} = v_{2n+1} = \ldots$, and similar for other indices. The pattern avoidance clauses are $C_{i,\text{fwd}} = \{o_{v_i,v_{i+1}}, o_{v_{i+1},v_{i+2}}\}$, which we are calling forward clauses, and $C_{i,\text{bwd}} = \{o_{v_{i+2},v_{i+1}}, o_{v_{i+1},v_i}\}$, the backward clauses, for each $i \in [n]$. We note that we do not need transitivity clauses in this example, because we are looking at a pattern for a coloring problem,

and Lemma 4.6 states that any assignments for these patterns implicitly fulfill transitivity. We now do a chain of resolutions around the entire cycle, alternating between forward and backward clauses. An example of this resolution chain is visualized in Figure 4.14.

$$\begin{aligned} C_{1,\text{fwd}} &= \{o_{v_1,v_2}, o_{v_2,v_3}\} \\ C_{1,\text{fwd}} \otimes C_{2,\text{bwd}} &= \{o_{v_1,v_2}, o_{v_4,v_3}\} \\ C_{1,\text{fwd}} \otimes C_{2,\text{bwd}} \otimes C_{3,\text{fwd}} &= \{o_{v_1,v_2}, o_{v_4,v_5}\} \\ &\dots \\ C_{1,\text{fwd}} \otimes C_{2,\text{bwd}} \otimes C_{3,\text{fwd}} \otimes \dots \otimes C_{n,\text{fwd}} &= \{o_{v_1,v_2}\} \end{aligned} \tag{4.14}$$

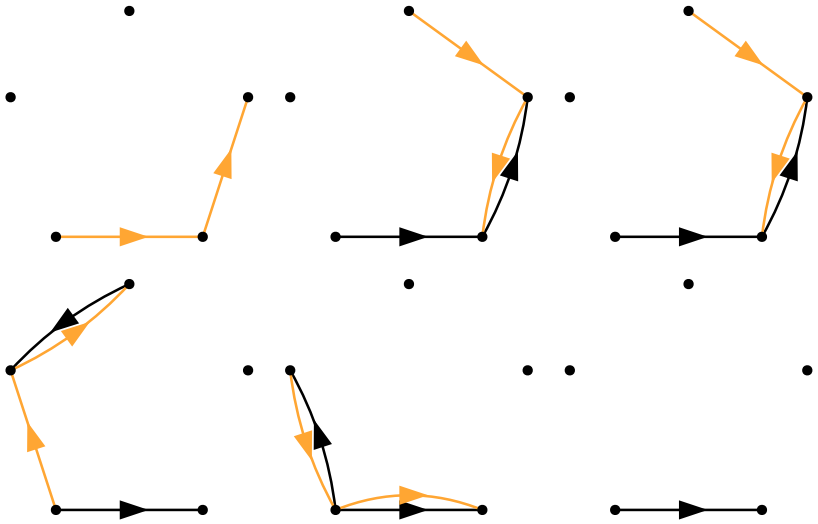

Figure 4.14: The resolution chain of Equation 4.14 visualized. The first graph shown is $G_{C_{1,\text{fwd}}}$. From then on, every graph has the result of the previous resolution in black, and the next clause to resolve with highlighted in orange. The final result is $\{o_{v_1,v_2}\}$.

When reversing all clauses, meaning that forward clauses are replaced with their backward counterpart and all backward with forward, the resolution chain stays valid. Now, the result is o_{v_1,v_2}:

$$C_{1,\text{bwd}} \otimes C_{2,\text{fwd}} \otimes C_{3,\text{bwd}} \otimes \dots \otimes C_{n,\text{bwd}} = \{o_{v_2,v_1}\}$$

Now, it takes only the resolution $\{o_{v_1,v_2}\} \otimes \{o_{v_2,v_1}\} = \emptyset$ to show that an odd cycle is not bipartite. This resolution proof requires $2n - 1$ steps. For $n = 3$, the lower bound of $k! - 1 = 5$ of Theorem 4.14 already matches the number of steps in this resolution. The more general bound of $n/2 - 1$ steps shows that the lower bound already is optimal up to constant factor in general.

5 Evaluation

This section presents the experimental evaluation of the approaches presented in Chapter 4. We first detail the experimental setup in Section 5.1, including relevant details of our implementation. The benchmarks used are discussed in Section 5.2. Then, we show the results of the experiments in Section 5.3, where we investigate the effects of various parts of the solving process on the speed of the solver.

5.1 Experimental setup

This section describes the setup for the experimental evaluation of the approaches presented in Chapter 4. The experiments were run on an AMD Ryzen 9 3950X 16-Core running with 3.5GHz and 64228MB RAM. For each approach and instance, a 60 second timeout was used unless noted differently. When mentioning the speedup of one approach over another, we treat unsolved instances as being solved in 60 seconds.

The approaches discussed in Chapter 4 are implemented with C++20 and using version 2.2.0 of CaDiCaL. The implementation is compiled with g++ version 11.4.0 using only the optimizations of the -O2 flag.

Time measurement is done by the program implementation itself. The time needed for loading the instance into memory, outputting results and freeing memory after the solving process is not measured. Any time used is attributed to one of four categories: pre- and inprocessing, adding the ordering constraint, adding the pattern avoidance constraint, and the SAT solving process. Calls to the user propagators are attributed to the SAT solving process, while the construction of the user propagators is attributed to adding the respective constraint. Time is measured with the C++ std::steady_clock and stored in microseconds. If milliseconds are used, rounding becomes a notable issue when using the incremental approaches, where sometimes only little work is done within some single measurement.

5.2 Benchmarks

For benchmarking, we use nine types of benchmarks generated in different ways. From each type, 50 benchmarks were generated. In the following description of the benchmark generation, we make use of the random() function, which generates a random number between 0 and 1 uniformly at random, and the randint(l, r) function, which generates a random integer between l and r uniformly at random, including both endpoints. We use python version 3.10.12 for the generation of all benchmarks, and use the python functions of the same name to generate these random values. The seeds used in the benchmark generation are the integers from 0 to 49, inclusive on both ends, in all cases.

The nine types of benchmarks can be summarized into three categories, which we label `structure`, `geometry`, and `property`. Across all benchmarks, the size of graphs are chosen to make for benchmarks that are reasonable for the SAT solver to solve in the given timeout. For most benchmark types, that means we use $n = \text{randint}(50, 2000)$. Additionally, for most

benchmark types, a random set of edges is deleted after generating a graph. This is done to obtain graphs with differing densities, but similar underlying structure. An example of this is the Delaunay instance type, which would be close to the upper bound of $m \leq 3n - 6$ for planar graphs in most cases. We generally use $p = \min(\text{random}() \cdot 0.8 + 0.4, 1.0)$ as the chance that an edge is *not* deleted. This results in $p = 1.0$ in one fourth of the benchmarks in expectation, and $p \geq 0.4$ in all benchmarks. In general, we use $\tilde{E}$ to denote the set of edges generated before the random deletion.

First, we look at the `structure` benchmarks. These are graphs with highly regular structure, where only the generation parameters and edge deletion are randomized.

- **Cycle product**: For these instances, we take the Cartesian product of multiple cycles. This results in a graph with no triangles as long as no initial cycle is a triangle, and therefore a graph with no large cliques. Additionally, the cycles provide an additional challenge for the SAT solver: Rotating the order of the vertices in a cycle does not remove any pattern occurrence, forcing the solver to branch into different orders.

  We generate a cycle product instance with the following process: Let $r = \text{randint}(3, 6)$ and $p = \min(\text{random}() \cdot 0.8 + 0.4, 1.0)$. Let L be an r-tuple where the ith element L_i is chosen as $L_i = \text{randint}(3, 11 - i)$. The graph G has vertices $V(G) = [L_1] \times [L_2] \times \cdots \times [L_r]$. The edge set $\tilde{E}$ contains an edge if the vertices differ by one in exactly one position i, modulo L_i:

  $$(u_1, u_2, \ldots, u_r)(v_1, v_2, \ldots, v_r) \in \tilde{E}$$
  $$\Longleftrightarrow$$
  $$\exists i \in [r] : (u_i - v_i) \bmod L_i \in \{1, L_i - 1\} \wedge j \neq i \rightarrow u_j = v_j$$

  The resulting graph is the Cartesian product of cycles of length $L_1, L_2, \ldots, L_r$. For each edge $uv \in \tilde{E}$, it is added to $E(G)$ with probability p.

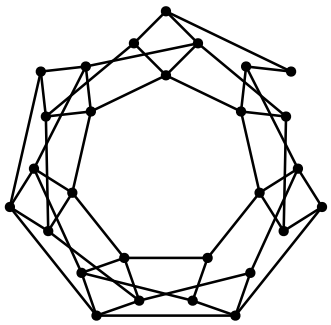

Figure 5.1: An example of a cycle product instance. This visualization is generated with the same rules as the benchmarks, but with the fixed parameters $r = 2, L = (4, 7)$ and $p = 0.9$.

- **Mod**: For the mod instances, we connect pairs of vertices a certain distance apart, modulo the number of vertices. The motivation for these instances is similar to the cycle product instances, although with more chances for large cliques.

We generate a mod instance with the following process: Let $n = \text{randint}(50, 2000)$, $d = \text{randint}(3, 7)$, and $p = \min(\text{random}() \cdot 0.8 + 0.4, 1.0)$. Let $L = \{1\}$, and then add an element $\text{randint}(2, n-1)$ until $|L| = d$. The graph G has vertices $V(G) = \{v_i \mid i \in \{0, 1, 2, \ldots, n-1\}\}$. The edge set $\tilde{E}$ is $\{v_i v_{(i+\ell) \bmod n} \mid i \in \{0, 1, 2, \ldots, n-1\}, \ell \in L\}$. For each edge $uv \in \tilde{E}$, it is added to $E(G)$ with probability p.

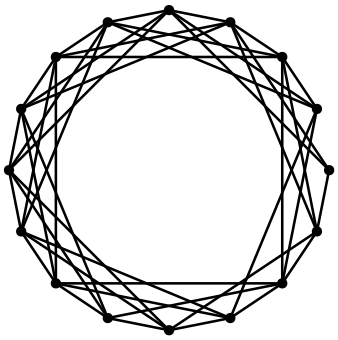

Figure 5.2: An example of a mod instance. This visualization is generated with the same rules as the benchmarks, but with the fixed parameters $d = 3$, $L = (1, 3, 4)$ and $p = 0.9$.

- **Multipartite**: Multipartite instances partition the vertices into multiple sets, and then connect pairs vertices that are not in the same set. This give graphs with a fixed upper bound on the chromatic number.

  We generate a multipartite instance with the following process: Let $n = \text{randint}(50, 2000)$, $k = \text{randint}(2, 7)$, and $p = \max(\text{random}() \cdot 0.8 + 0.4, 1.0)$. The graph G has the vertices $V(G) = \{v_i \mid i \in [n]\}$. Every vertex $v \in V(G)$ is assigned a random color $c(v) = \text{randint}(1, k)$. For each pair of vertices $u, v \in V(G)$, we have $uv \in \tilde{E}$ if and only if $c(u) \neq c(v)$. For each edge $uv \in \tilde{E}$, it is added to $E(G)$ with probability p.

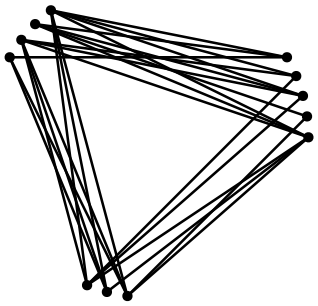

Figure 5.3: An example of a multipartite instance. This visualization is generated with the same rules as the benchmarks, but with the fixed parameters $n = 12$, $k = 3$ and $p = 0.5$.

Second are the `geometry` benchmarks. All of these benchmarks have some underlying geometric interpretation, and are in all cases generated from this geometric interpretation.

- **Delaunay**: A Delaunay triangulation [KL10] of a point set in $\mathbb{R}^2$ has the property that for any triangle T, there is no point inside the circumcircle of T. We use Delaunay triangulation to generate random planar graphs by triangulating random point sets.

The vertices of the graph are the points, and two vertices are connected by an edge if the respective points are part of the same triangle. Any graph that can be created from this process is called a *Delaunay graph*. An example is shown in Figure 5.4.

We generate a Delaunay instance with the following process: Let $n = \text{randint}(50, 2000)$ and $p = \min(\text{random}() \cdot 0.8 + 0.4, 1.0)$. Let $P \subseteq \mathbb{R}^2$ be a set of points with $|P| = n$, where the ith point is located at (x_i, y_i) with $x_i = \text{randint}(-10^9, 10^9)$ and $y_i = \text{randint}(-10^9, 10^9)$. The graph G has the vertices $V(G) = P$. Two vertices u and v are connected by an edge uv in $\tilde{E}$ if and only if the points are connected by a line in a Delaunay triangulation of P. For each edge $uv \in \tilde{E}$, it is added to $E(G)$ with probability p.

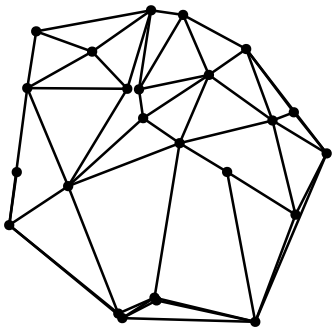

Figure 5.4: An example of a Delaunay instance. This visualization is generated with the same rules as the benchmarks, but with the fixed parameters $n = 24$ and $p = 0.9$.

- **Gabriel**: Let G be a Delaunay graph, with $P \subset \mathbb{R}^2$ being the point set used to create G. For any $uv \in E(G)$, delete the edge if there is a point within the circle whose diameter is the line segment connecting u and v. Any graph that can be created from this process is called a *Gabriel graph* [KL10]. We randomly create Gabriel graphs by using the random process to create Delaunay graphs, and then deleting the edges that are not allowed in a Gabriel graph. An example is shown in Figure 5.5.

  We generate a Gabriel instance with the following process: Let $n = \text{randint}(50, 2000)$ and $p = \min(\text{random}() \cdot 0.8 + 0.4, 1.0)$. Let $P \subseteq \mathbb{R}^2$ be a set of points with $|P| = n$, where the ith point is located at (x_i, y_i) with $x_i = \text{randint}(-10^9, 10^9)$ and $y_i = \text{randint}(-10^9, 10^9)$. The graph G has vertices $V(G) = P$. Two vertices u and v are connected by an edge uv in $\tilde{E}$ if and only if the points are connected by a line in a Delaunay triangulation of P and the circle with center $(u + v)/2$ and diameter $|u - v|$ does not contain any point. For each edge $uv \in \tilde{E}$, it is added to $E(G)$ with probability p.

- **Hyperbolic**: Hyperbolic graphs appear as a natural structure for analyzing networks [Kri+10]. We add a simple generator for random hyperbolic graphs to our benchmarks.

  We generate a hyperbolic instance with the following process: Let $n = \text{randint}(50, 500)$, $t = \text{random}() \cdot 0.4 + 0.9$, and $p = \max(\text{random}() \cdot 0.8 + 0.4, 1.0)$. Choose a set P of n points uniformly at random inside a unit disk. The graph G has vertices $V(G) = P$. Two vertices u and v are connected by an edge uv in $\tilde{E}$ if and only if the points have hyperbolic distance under t when interpreting the disk as a Poincaré disk:

$$uv \in \tilde{E} \iff \operatorname{arcosh}\left(1 + 2\frac{|u - v|^2}{(1 - |u|^2) \cdot (1 - |v|^2)}\right) < t$$

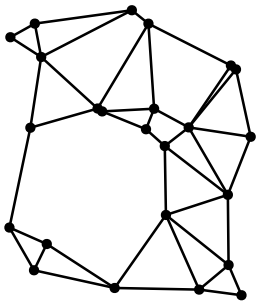

Figure 5.5: An example of a Gabriel instance. This visualization is generated with the same rules as the benchmarks, but with the fixed parameters $n = 24$ and $p = 0.9$.

For each edge $uv \in \tilde{E}$, it is added to $E(G)$ with probability p.

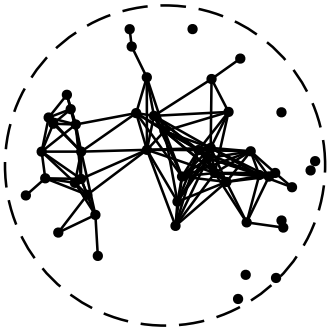

Figure 5.6: An example of a hyperbolic instance. This visualization is generated with the same rules as the benchmarks, but with the fixed parameters $n = 48$, $t = 1.1$ and $p = 0.9$. The dashed circle is the border of the Poincaré disk.

- k-**closest**: Last for the geometric instances, we generate graphs by randomly placing points in a space, then connecting each point to the k closest ones. This gives the graph a form of locality: The more neighbors two vertices share, the closer their respective points likely are, resulting in a higher chance that they are connected themselves.

  We generate a k-closest instance with the following process: Let $n = \text{randint}(50, 2000)$, $k = \text{randint}(3, 7)$, $d = \text{randint}(2, 4)$, and $p = \max(\text{random}() \cdot 0.8 + 0.4, 1.0)$. Let $P \subseteq \mathbb{R}^d$ be a set of points with $|P| = n$, where every point has its position in each dimension independently chosen by $\text{randint}(-10^9, 10^9)$. The graph G has vertices $V(G) = P$. Two vertices u and v are connected by an edge uv in $\tilde{E}$ if and only if u is among the k closest point to v, or v is among the k closest points to u. For each edge $uv \in \tilde{E}$, it is added to $E(G)$ with probability p.

Third are the `property` benchmarks. These benchmarks are generated with some graph theoretic property in mind.

- **Degenerate**: Degenerate graphs are already introduced in Section 3.1.2, due to the possibility to describe them using a pattern.

  We generate a degenerate instance with the following process: Let $n = \text{randint}(50, 2000)$, $d = \text{randint}(2, 6)$, and $p = \max(\text{random}() \cdot 0.8 + 0.4, 1.0)$. The graph G has the vertices $V(G) = \{v_i \mid i \in [n]\}$. To generate the edges, we iterate over all j from 1 to n, and

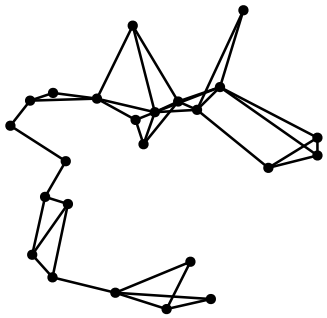

Figure 5.7: An example of a k-closest instance. This visualization is generated with the same rules as the benchmarks, but with the fixed parameters $n = 24$, $k = 3$, $d = 2$ and $p = 0.9$.

select d vertices from $\{v_i \mid i \in [j-1]\}$ uniformly at random with replacement. If $j \leq d$, all vertices are chosen. The vertex labels are shuffled after this step. Then, let $v_i v_j$ with $i < j$ be in $\tilde{E}$ if and only if v_i was chosen in the jth step. For each edge $uv \in \tilde{E}$, it is added to $E(G)$ with probability p.

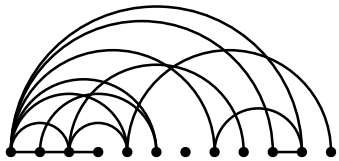

Figure 5.8: An example of a degenerate instance. This visualization is generated with the same rules as the benchmarks, but with the fixed parameters $n = 12$, $d = 3$ and $p = 0.9$, and without shuffling the vertices. The vertices are ordered left to right so that every vertex has at most d neighbors to its left.

- **Random Mycielski**: The Mycielski construction introduced in [Myc55] is used to show that there are graphs with no triangles that still need arbitrarily many colors for a proper coloring. This construction takes a graph G_i and constructs G_{i+1} in three steps: Create a copy v' for every vertex $v \in V(G_i)$, connect each copy to the neighbors of the originals, and connect every copy v' to a new vertex x.

$$V(G_{i+1}) = V(G_i) \cup \{v' \mid v \in V(G_i)\} \cup \{x\}$$
$$E(G_{i+1}) = E(G_i) \cup \{uv' \mid uv \in E(G_i)\} \cup \{v'x \mid v \in V(G_i)\}$$

  In this construction step, the number needed to color the graph increases by one, while the size of the largest clique stays the same. We generate benchmarks of this type by generating a random graph, and then applying this construction a few times.

  We generate a random Mycielski instance with the following process: Let $n = \text{randint}(25, 100)$, $k = \text{randint}(1, 4)$, and $p = \text{random}() \cdot 0.2 + 0.1$. Start with the graph G_0 with n vertices where each edge exists independently with chance p. Then, apply the Mycielski construction k times to get the instance graph $G = G_k$. Note that the actual number of vertices after these steps ranges from 51 to 1615.

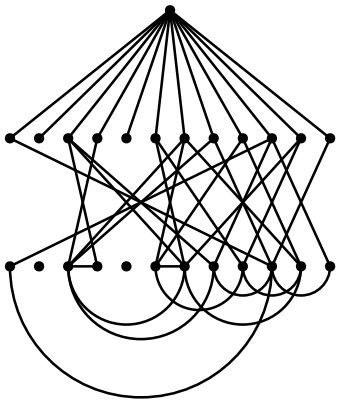

Figure 5.9: An example of a random Mycielski instance. This visualization is generated with the same rules as the benchmarks, but with the fixed parameters $n = 12$, $k = 1$ and $p = 0.1$. The bottom row is the original graph, the middle row are the vertex copies, and the top row contains only the vertex x that is connected to all vertex copies.

5.3 Results

This section discusses the experimental results from testing the approaches introduced in Chapter 4. Most of the experiments use the preprocessing discussed in Section 4.3.2, but do not use the filtering of Section 4.3.1. The pattern P used for the benchmarks has four vertices and the edges $E_M(P) = \{p_1p_2, p_1p_4, p_3p_4\}$, and is shown in Figure 5.10.

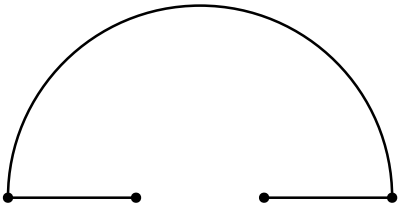

Figure 5.10: The pattern used for the benchmarks.

In the upcoming evaluation, first and foremost is the comparison the speed of the different approaches for the ordering constraint introduced in Section 4.1. The comparison of these approaches is presented in Section 5.3.1. Next, in Section 5.3.2, we compare the different incremental approaches against each other and the non-incremental approach from the previous section. These two sections give the main results that can be transferred to other applications. For the next three sections, we look at the constants of the previous evaluations: Section 5.3.3 considers the effect of using CEGAR instead of the recursive encoding to ensure pattern avoidance. Section 5.3.4 tests the effect of using filtering and not using preprocessing. Section 5.3.5 looks at the difference in solver speed when using different patterns. Then, we make one last jump in perspective from the solver to the benchmark instances in Section 5.3.6.

Before we do get to the actual evaluation, we do need to note conventions for the results presented in this section. First, when talking about the average speedup between two approaches, we take the geometric mean of the speedups of all instances solved by at least one approach. If an instance is solved by only one approach, we take the timeout (commonly 60 seconds) as the time taken for the other approach. This generally results in somewhat

lower speedups than would be achieved if all approaches could run to completion, due to the fact that in most cases, the approach in question would probably not solve the instance right after the timeout. For any decimal numbers, we round them to four digits. If an axis in a plot represents time, then it uses linear scale from 0 to $10^0 (= 1)$, and logarithmic scale from 10^0 onwards.

5.3.1 Ordering constraint

First, we present the experimental results for the different ways to ensure the ordering constraint. The results from this experiment are relevant to more general problems than the Pattern-Avoiding Vertex Ordering problem, as these techniques can be used to solve other problems with ordering constraints. We compare the eight approaches presented in Section 4.1, namely `encode-all`, `encode-needed`, `unit-propagator-all`, `unit-propagator-needed`, `cycle-prevention`, `reachability-cycle-prevention`, `cycle-refinement`, and `path-sampling-refinement`. The results of this experiment are given in Figure 5.11. Due to the two best approaches, namely `unit-propagator-all` and `cycle-prevention`, being very close after the 60 second timeout, they instead have a 120 second timeout for a better comparison. Additionally, we show the number of solved instances and their speedups compared to `encode-all` in Table 5.1. In this table, we use the timeout of 60 seconds for all instances.

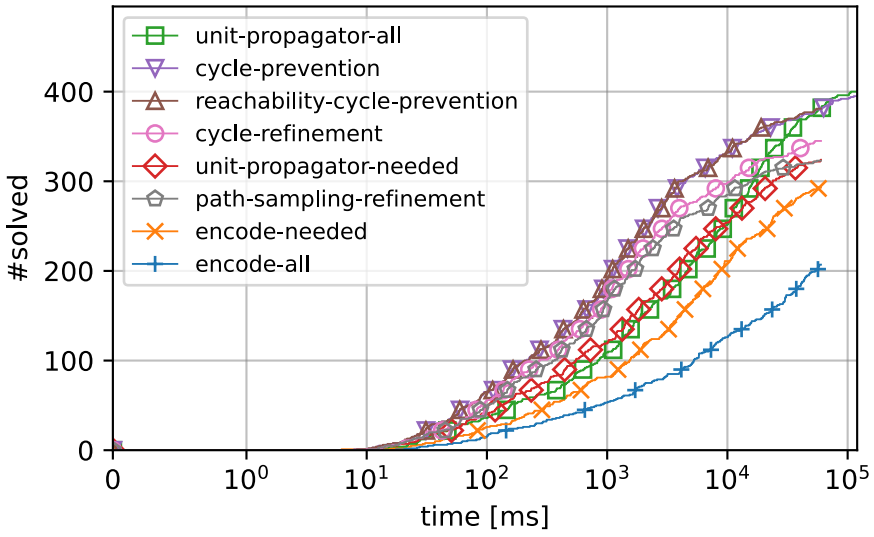

Figure 5.11: The number of instances solved by each approach for the ordering constraint, compared to the time needed. All benchmarks use a 60 seconds timeout, except for `unit-propagator-all` and `cycle-prevention` with a 120 seconds timeout.

First, we compare `encode-all` and `encode-needed`. This comparison shows us the effectiveness of the reduction in the number of variables we get from applying Lemma 4.5. The respective plot lines are highlighted in Figure 5.12.

The first question we want to answer is the relation between the number of variables eliminated in `encode-needed` and the speedup we get. The plot 5.13 shows the expected relation: The less dense the variable graph G_U is, and equivalently the less variables exist, the higher the speedup is. We do note however that there are instances where `encode-needed` is slower than `encode-all`. There are two explanations for this: Calculating the chordal

	#solved			Speedup vs `encode-all`
	SAT	UNSAT	total	
`encode-all`	50	157	207	1.0000
`encode-needed`	53	240	293	3.1641
`unit-propagator-all`	81	301	382	4.9595
`unit-propagator-needed`	60	264	324	6.5556
`cycle-prevention`	75	305	380	14.9299
`reachability-cycle-prevention`	76	305	381	14.9089
`cycle-refinement`	41	304	345	11.1429
`path-sampling-refinement`	20	303	323	9.2757

Table 5.1: The number of solved instances and average speedup compared to `encode-all` for all approaches to ensure the ordering constraint. All results in this table are after a 60 seconds timeout.

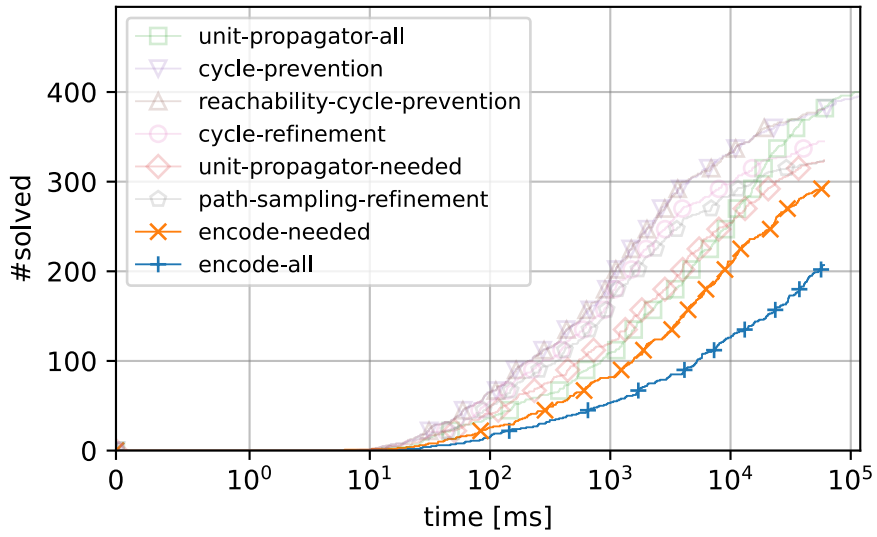

Figure 5.12: The number of instances solved by each approach for the ordering constraint, compared to the time needed. The approaches `encode-all` and `encode-needed` are highlighted.

completion of the needed variables takes time, and the SAT solver may have problems with the smaller encoding, for example having fewer results from unit propagation. To test for this, we also compare the time only spent by the SAT solver, ignoring the time needed for any preprocessing or encoding. The result of both comparisons are shown in Figure 5.14. Here, we can see that there are instances that are solved faster by `encode-all`, even when ignoring the time spent outside the SAT solver. In total, of the 200 instances solved by both approaches, `encode-needed` is faster on 133 of them than `encode-all`. When only looking at the time spent by the SAT solver, `encode-needed` is faster on 142 of them than `encode-all`. Of the remaining 58 instances, the type of instance where `encode-all` is faster than `encode-needed` is also overwhelmingly of the random Mycielski type. A complete list of the number of instance

by type where `encode-all` is faster than `encode-needed` is shown in Table 5.2. In total, the average speedup of `encode-needed` over `encode-all` is 3.1641, and an even average higher speedup of 4.4012 when only looking at the time spent by the SAT solver.

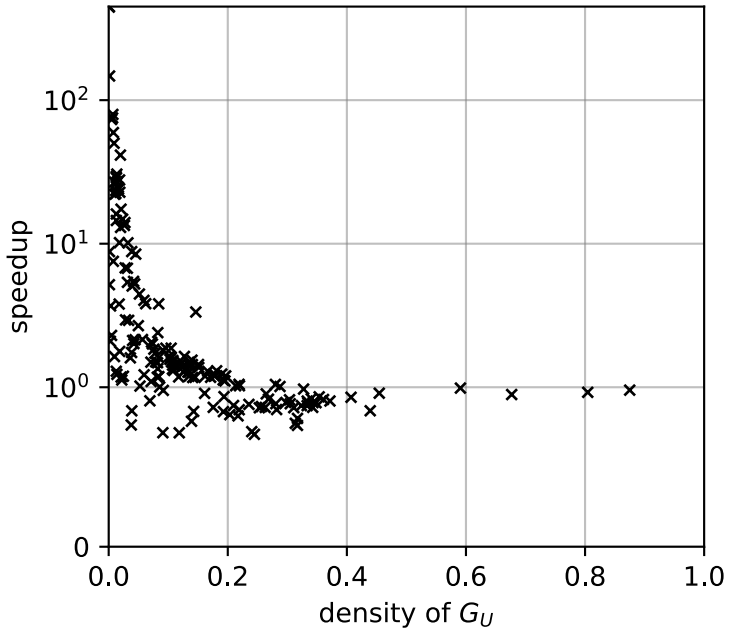

Figure 5.13: The speedup of `encode-needed` over `encode-all`, depending on the density of G_U. Calculating the chordal completion is possible in one second timout for all instances in this plot.

Instance type	`encode-all` faster
Cycle product	6
Degenerate	5
Delaunay	0
Gabriel	0
Hyperbolic	0
k-closest	0
Mod	8
Multipartite	7
Random Mycielski	32

Table 5.2: The number of instances per type where `encode-all` is faster than `encode-needed` when only looking at the time taken by the SAT solver. Only instances solved by both `encode-all` and `encode-needed` are considered.

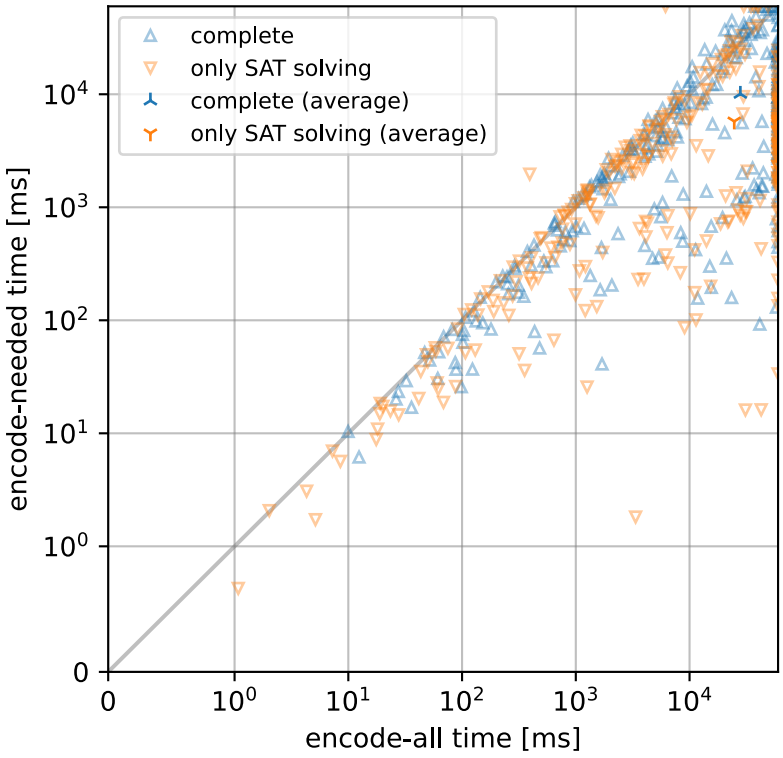

Figure 5.14: A comparison of the solving times for instances solved by at least `encode-all` or `encode-needed`. The blue upward-pointing triangles give the complete time needed to solve the instance, including the time spent for preprocessing and encoding. The orange downward-point triangles give the time needed to solve the instance by the SAT solver. The average for both are also marked.

On the other hand, when using a unit propagator, using all ordering variables instead of only the needed ones results in a faster solver. Figure 5.15 shows the plot shown in Figure 5.11 with the approaches `unit-propagator-all` and `unit-propagator-needed` highlighted. Figure 5.16 shows the speedup compared to the density of G_U, like Figure 5.13 shows this comparison for the speedup of `encode-needed` against `encode-all`.

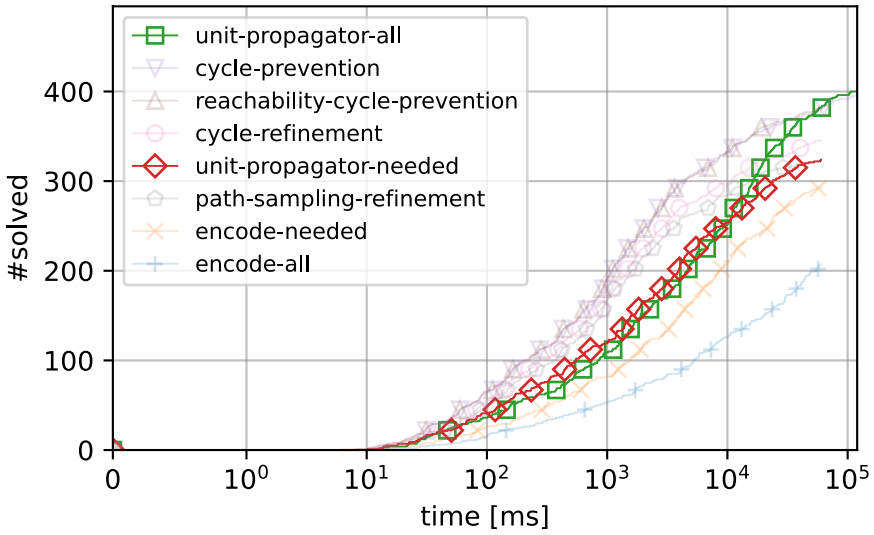

Figure 5.15: The number of instances solved by each approach for the ordering constraint, compared to the time needed. The approaches `unit-propagator-all` and `unit-propagator-needed` are highlighted.

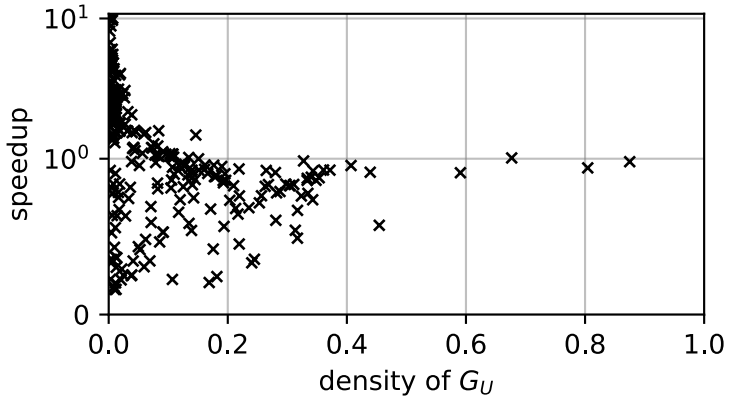

Figure 5.16: The speedup of `unit-propagator-needed` over `unit-propagator-all`, depending on the density of G_U. Calculating the chordal completion is possible in one second timout for all instances in this plot.

Last, we take a look at the three best approaches, namely `unit-propagator-all`, `cycle-prevention`, and `reachability-cycle-prevention`. For further implementation and experiments, we use `cycle-prevention`. Although `unit-propagator-all` performs better on the hard benchmarks, our implementation choice is motivated by preliminary results on easier instances, where `cycle-prevention` is still better. There is little difference between `cycle-prevention` and `reachability-cycle-prevention`. We choose `cycle-prevention` due to it being simpler to adapt.

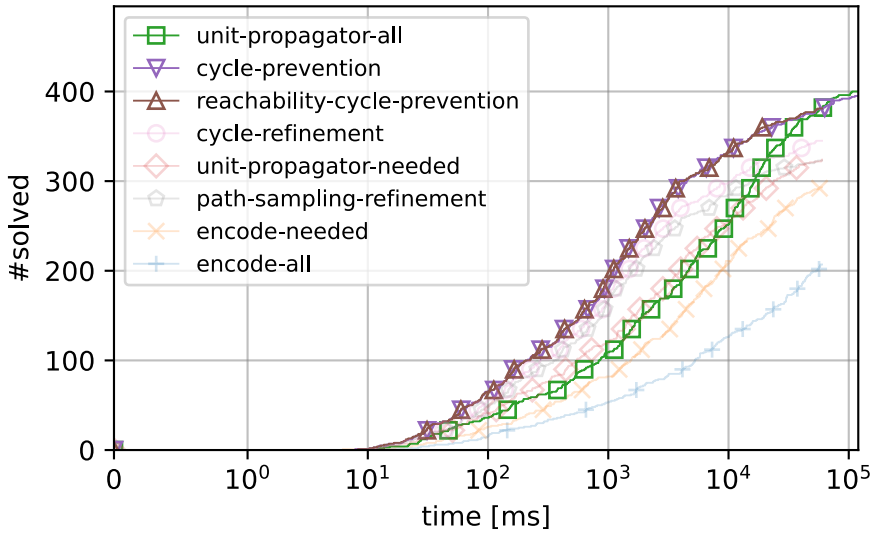

Figure 5.17: The number of instances solved by each approach for the ordering constraint, compared to the time needed. The best approaches, `unit-propagator-all`, `cycle-prevention`, and `reachability-cycle-prevention`, are highlighted.

5.3.2 Incremental solving

Section 4.3.3 describes two incremental approaches to solve the Pattern-Avoiding Vertex Ordering problem. All incremental approaches are based on `cycle-prevention`. First, we look at `subgraph-incremental`, where we give the SAT solver the encoding for only an induced subgraph of the instance graph G, and then use incremental solving to add more and more parts of the graph until we have the entire graph. First, we fix the strategy for the sizes of these subgraphs to `chunks` and test the different ways to choose the subgraphs: `bfs`, `degeneration`, `random`, and `sampling`. The result of this comparison is shown in Figure 5.18.

Here, we can see that the `degeneration` approach performs best, although the number of solved instances is close at the timeout. This is common among all incremental approaches, and is most likely due to the fact that incremental solving is only helpful when the solver can exit early at some point, which is less likely for difficult instances. We use `degeneration` for the next test, where we determine the best size scaling strategy for the subgraphs given to the SAT solver. The result of this comparison is shown in Figure 5.19.

In this comparison, `small-to-all` performs better overall in the number of instances solved, while `k-exponential` is faster on easier instances. Again, the number of instances are largely the same for all incremental approaches. The approach `k-exponential` outperforming both other approaches is reasonable: First, the success of `k-exponential` over `small-to-all` is explained by it managing to give a second chance to find an obstruction if the initial small graph does not contain one. Second, `k-exponential` outperforming `chunks` comes from `k-exponential` avoiding large, non-complete graphs, which take time and are likely to arise for satisfiable instances.

Next, we look at `heuristic-incremental`. Here, we only have one parameter to test for: The number of layers chosen for the heuristic solver. The result of this comparison is shown in Figure 5.20.

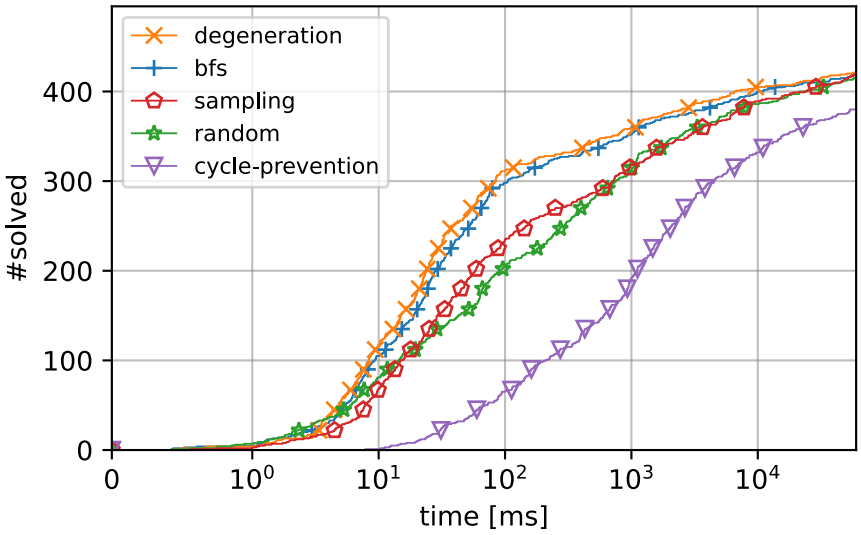

Figure 5.18: The number of solved instances compared to the time taken for the four different ways to choose the subgraphs with the `subgraph-incremental` approach. Additionally, the non-incremental `cycle-prevention` approach for comparison.

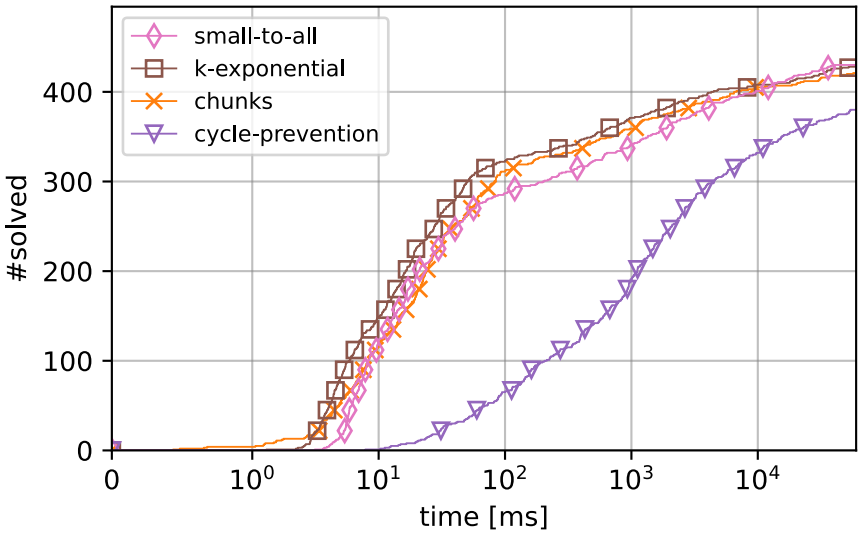

Figure 5.19: The number of solved instances compared to the time taken for the three different ways to scale the size of the subgraphs with the `subgraph-incremental` approach. Additionally, the non-incremental `cycle-prevention` approach for comparison.

Last, we give a comparison of the best `subgraph-incremental` solver, using `degeneration` and `k-exponential`, with the best `heuristic-incremental` solver, using `full-graph`. The result of this comparison is shown in Figure 5.21. For this plot, we split the instances by satisfiable and unsatisfiable to understand the effect of the two incremental solvers. Similar to Table 5.1, we give Table 5.3 showing the number of solved instances and speedup over `cycle-prevention` for these three approaches.

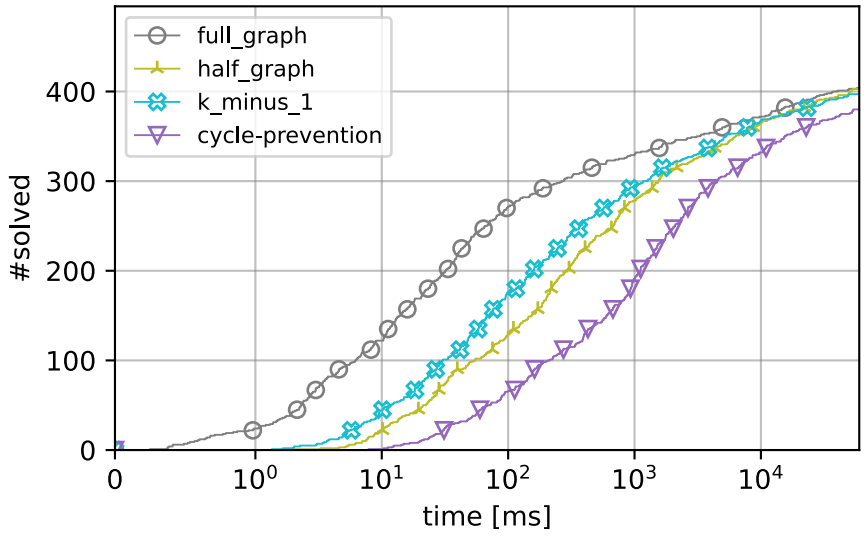

Figure 5.20: The number of solved instances compared to the time taken for the three different ways to choose the number of layers with the `heuristic-incremental` approach. Additionally, the non-incremental `cycle-prevention` approach for comparison.

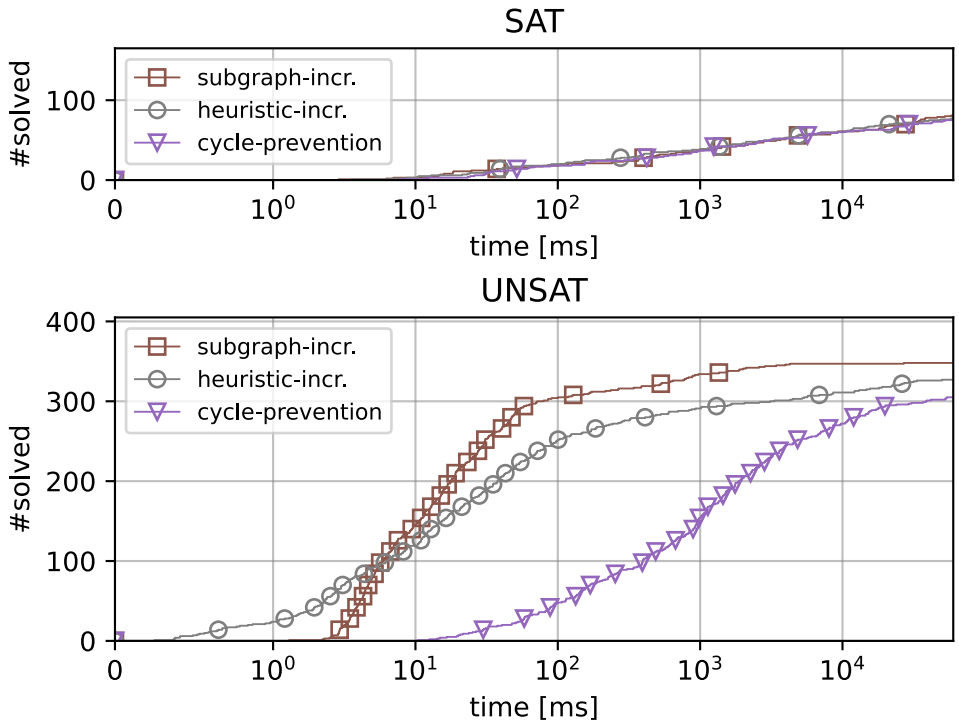

Figure 5.21: The number of solved instances compared to the time taken for the fastest incremental subgraph and incremental approach. Additionally, the non-incremental `cycle-prevention` approach for comparison. The results are split by satisfiable and unsatisfiable instances.

	#solved			Speedup vs `cycle-prevention`
	SAT	UNSAT	total	
`cycle-prevention`	75	305	380	1.0000
`subgraph-incr.`	81	348	429	34.4702
`heuristic-incr.`	77	327	404	19.2646

Table 5.3: The number of solved instances and average speedup compared to `cycle-prevention` for the fastest subgraph and heuristic incremental approach. All results in this table are after a 60 seconds timeout.

From this plot, we conclude that there is no major difference for satisfiable instances. For unsatisfiable instances, we already know that `subgraph-incremental` performs better than `cycle-prevention` from the experiments shown in Figure 5.18 and 5.19. It is however interesting that `subgraph-incremental` also solves more satisfiable instance than `cycle-prevention`, even though `subgraph-incremental` can only exit early on unsatisfiable. This means that the incremental addition of vertices guides the SAT solver better than the standard heuristics.

5.3.3 CEGAR for Pattern Avoidance

In this section, we discuss the two approaches discussed in Section 4.2.3 for using CEGAR to ensure pattern avoidance. First is `pattern-CEGAR`, which detects a pattern during model checking, and adds a pattern avoidance clause as described in Equation 4.6 for each occurrence of the pattern found. The detection does not need to find all occurrences of the pattern, as long as at least one is found if one is present. For this, we use the faster detection of the benchmark pattern (see Figure 5.10) also presented in Section 4.2.3. Second is the idea that the right neighbors $N_{\text{right}}(v)$ of any vertex v must be a star for the benchmark pattern, called `consistency`. This still requires an encoding for pattern avoidance, where we use the recursive encoding presented in Section 4.2.2. We implement both approaches on top of and compare both approaches with `encode-all`, which also uses the recursive encoding of the pattern avoidance. The result of this comparison is shown in Figure 5.22.

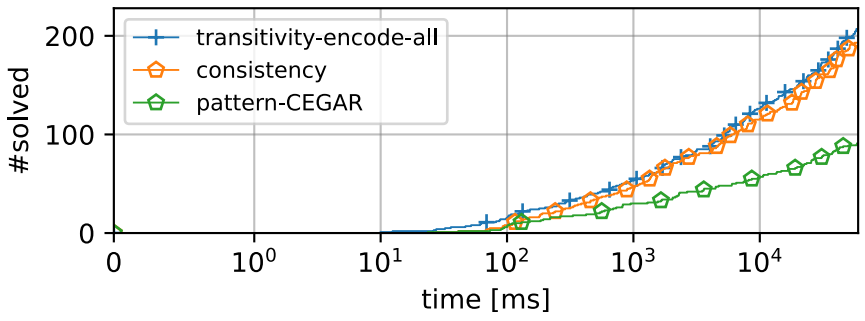

Figure 5.22: The number of solved instances compared to the time taken for the two CEGAR approaches for pattern avoidance. All runs use the non-incremental solver with `encode-all`.

Both approaches turn out to be slower than `encode-all`. The `consistency` approach is comparable to `encode-all`, which is most likely due to the work done by the user propagator per literal assignment is low. On the other hand, `pattern-CEGAR` takes considerably more time.

5.3.4 Filtering and preprocessing

All experiments in Section 5.3 except for those in this subsection use no filtering, but do use preprocessing. In this section, we investigate the effect of preprocessing and filtering on the speed of the SAT solver. For this, we look at the two fastest approaches, namely `unit-propagator-all` and `cycle-prevention`.

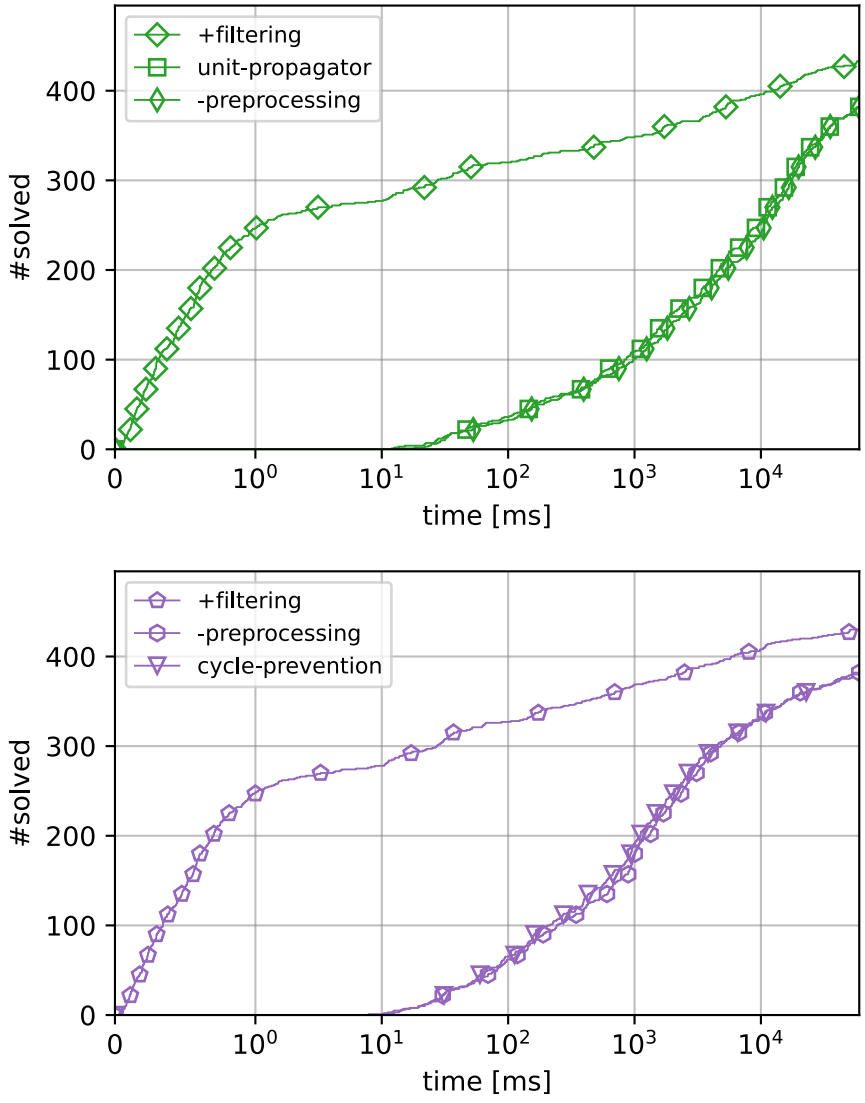

Figure 5.23: The number of solved instances compared to the time taken when turning on filtering or turning off preprocessing for the approaches `unit-propagator` and `cycle-prevention`.

From this, we can see that preprocessing has little effect on the time spent by the solver. This is probably because for this pattern, only vertices with degree 0 or 1 can be removed. On the other hand, filtering quickly eliminates about half the instances, including some that remain otherwise unsolved. This shows the reason for not using filtering in other experiments: Doing so would reduce the gap between approaches, making it harder to distinguish the best ones. Additionally, the strength of filtering is more specific to the Pattern-Avoiding Vertex Ordering problem. Compare this to results achieved without filtering, for example those on the ordering constraint seen in Section 5.3.1, where it is more reasonable to expect that the speedup behaves similarly when these techniques are applied to other ordering problems.

Last, we give a comparison between non-incremental solving with filtering, and incremental solving without filtering. This comparison is shown in Figure 5.24, together with the non-filtering non-incremental approach `cycle-prevention` as a baseline.

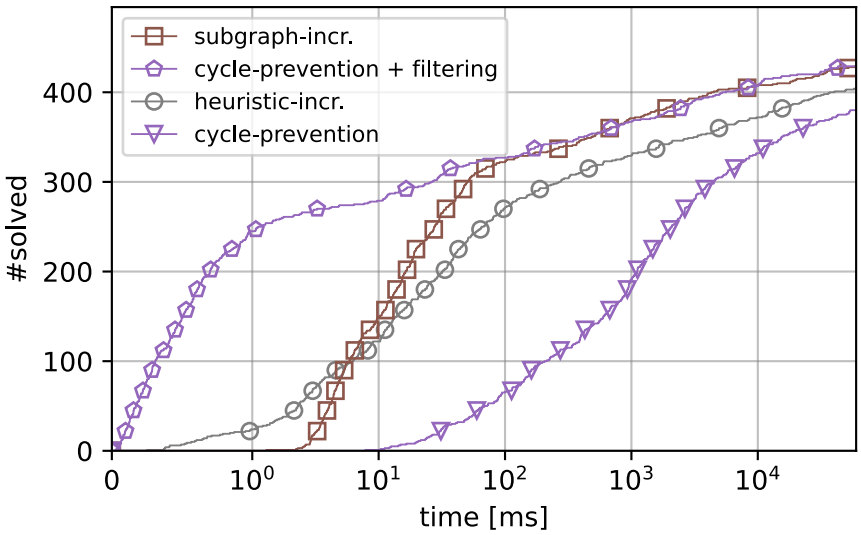

Figure 5.24: The number of solved instances compared to the time taken, comparing a non-incremental filtering to an incremental non-filtering approach.

Here, we can see that the `subgraph-incremental` approach is competitive with the non-incremental filtering approach. We attribute this to both of them being able to exit early out of unsatisfiable instances, which make up the majority of instances. The major difference between these two approaches is that filtering solves the easy instances faster, using simple heuristics, while `subgraph-incremental` does needs to do some work deciding on the chain of subgraphs before solving it can solve its first instances.

5.3.5 Comparing patterns

In this section, we compare different patterns with the same approach. Specifically, we run the non-incremental solver with using `cycle-prevention`, preprocessing, and no filtering. First, we look at the patterns P that are *ordered cycles*: The ordered cycle with k vertices has the edges $E_M(P) = \{p_i p_{i+1} \mid i \in [k-1]\} \cup \{p_1 p_k\}$. We compare the ordered cycles with k ranging from 4 to 7. There are two different ways to describe an ordered cycle with seven vertices with the recursive definition shown in Chapter 2 that is used for the encoding presented in Section 4.2.2. These two different ways are:

- **(A)** Chain two edges for a path with $k = 3$. Chain an edge and a path with $k = 3$ for a path with $k = 4$. Chain two paths with $k = 4$ with a covering edge for an ordered cycle with $k = 7$.
- **(B)** Chain two edges for a path with $k = 3$. Chain two paths with $k = 3$ for a path with $k = 5$. Chain a path with $k = 3$ and one with $k = 5$ with a covering edge for an ordered cycle with $k = 7$.

The comparison for these five patterns is shown in Figure 5.25. With this comparison, we investigate two aspects of Pattern-Avoiding Vertex Ordering: How does the speed of the solver scale with the size of the pattern, and how do different encodings effect the speed of the solver?

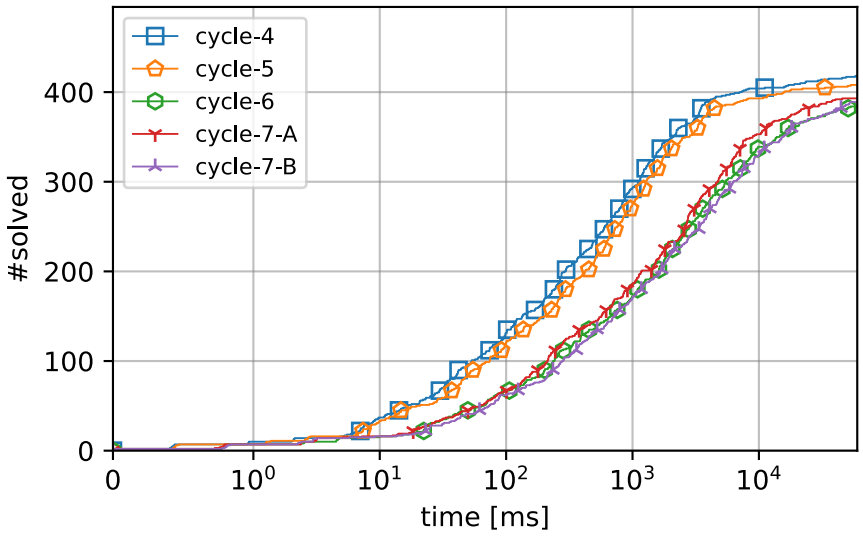

Figure 5.25: The number of solved instances compared to the time taken for five different cycles. All runs use the non-incremental solver with `cycle-prevention`.

The only notable gap in this plot is the cycles with 5 and 6 vertices. This is most likely due to the cycle with $k = 5$ requiring only two steps to create with the recursive definition, while $k = 6$ requires at least three steps. The number of variables for the recursive definition of pattern avoidance defined in Section 4.2.2 scales linearly with the number of steps in the construction of the pattern.

Another aspect we look at is the direction of the pattern. From Lemma 3.2, we know that reversing a pattern does not change whether a graph is a satisfiable or unsatisfiable instance. The patterns we use for this comparison are P and P^R, where the pattern P has size $k = 4$ and mandatory edges $E_M(P) = \{p_1p_4, p_2p_3, p_2p_4\}$. The comparison is shown 5.26.

From this comparison, we see that for the vast majority of instances, it does not make a difference. The remaining instances can be explained by the SAT solver getting (un)lucky with the chosen variable branching or other heuristics.

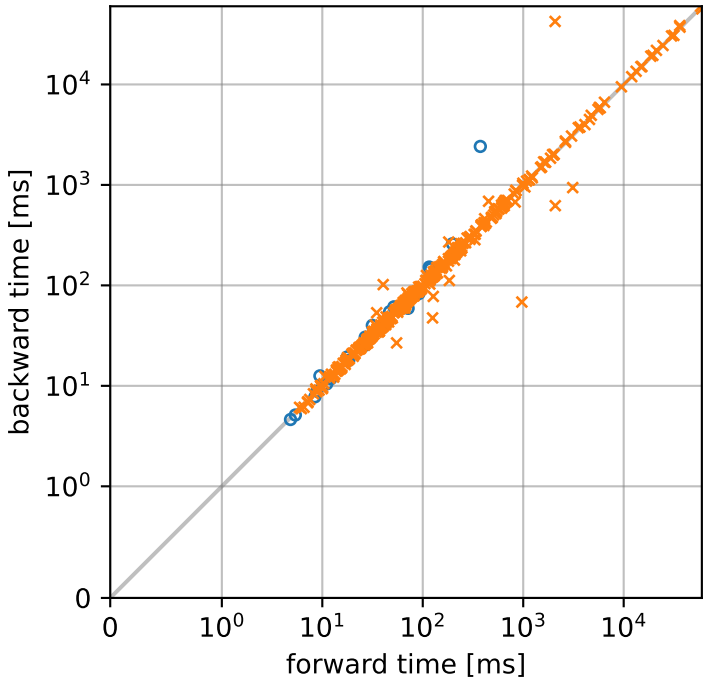

Figure 5.26: A comparison of the time needed for each instance for the two different directions of a pattern. Both runs use the non-incremental solver with `cycle-prevention`.

5.3.6 Comparing benchmark types

So far, Section 5.3 compared different approaches to solving the Pattern-Avoiding Vertex Ordering problem. This section instead compares the nine types of instances presented in Section 5.2 used to run these experiments. To compare these instance types, we show comparison plots between two approaches, and use different markers for different instance types. This way, we can see what instance types are favorable for what approaches.

We start with the first two encodings discussed, `encode-all` and `encode-needed`. This comparison gives us the effect of reducing the variable set per instance type. The plot showing this comparison is given in Figure 5.27.

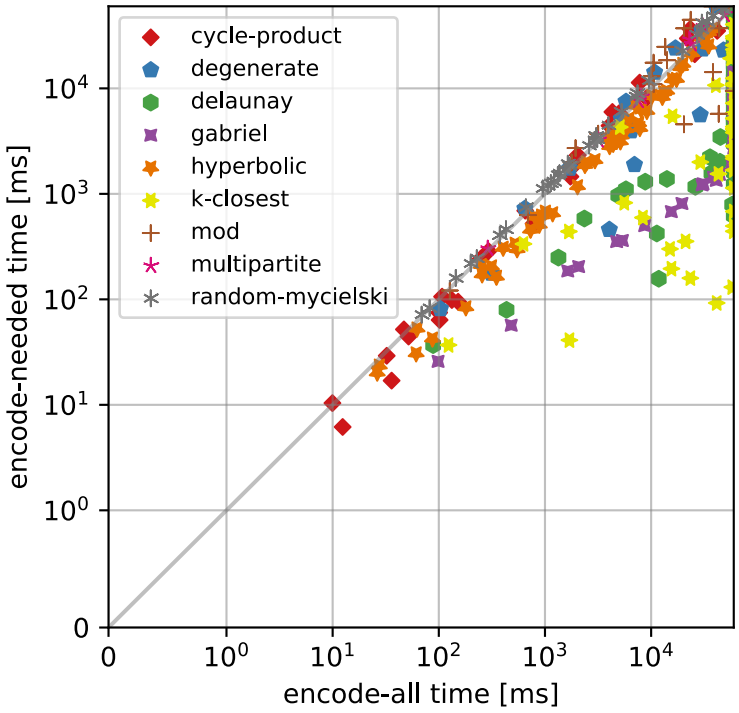

Figure 5.27: A comparison of the time needed for each instance using `encode-all` and `encode-needed` by instance type.

Here, we see that the instance types cycle product, hyperbolic, multipartite and random Mycielski instances do not improve a lot, mod and degenerate instances have a few improvements, and the Delaunay, Gabriel and k-closest instances generally improve notably. Because the only improvement of `encode-all` over `encode-needed` is the removal of variables, it is a reasonable assumption that the instance types with the highest improvement are those where, on average, a lot of variables can be removed. We can verify this by taking the data already used for comparing the density of the variable graph after removing as many variables as possible for and the speedup of `encode-needed` over `encode-all` in Figure 5.13. Table 5.4 shows exactly this average density of the variable graph, ordered from lowest to highest. We exactly what we expect: The Delaunay, Gabriel and k-closest instances have the fewest variables, and we can even see that the degenerate and mod instances that have a few improvements are still considerable better than the next highest entry, the cycle product instances. It also appears that the improvement from removing variables is only worthwhile if a lot of variables can be removed, as already instances with variable graph density 0.0384, where about 97% of variables are removed, show very little improvement to removing none at all.

Instance type	Average density
k-closest	0.0088
Delaunay	0.0090
Gabriel	0.0094
degenerate	0.0166
mod	0.0384
cycle product	0.0926
hyperbolic	0.1148
random Mycielski	0.2550
multipartite	0.7284

Table 5.4: The average density of G_U by instance type. The entries are ordered from low to high average density.

Next, we compare the two fastest non-incremental approaches, `cycle-product` and `unit-propagator-all`. The plot for this comparison is shown in Figure 5.28. From the comparison shown in Figure 5.17, we note that `cycle-product` is generally faster, but `unit-propagator-all` solves more instances. We therefore look out for the instance types where `cycle-product` performs better.

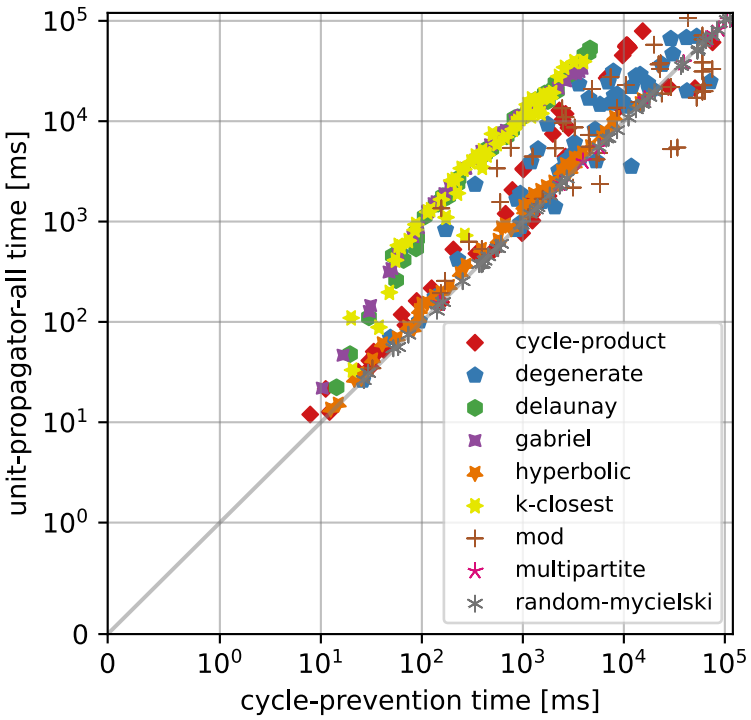

Figure 5.28: A comparison of the time needed for each instance using `cycle-prevention` and `unit-propagator-all` by instance type.

Again, the Delaunay, Gabriel and k-closest instances split away from the other instances types. This is reasonable, due to `cycle-product` using a reduced variable set, even less than `encode-all`. It is however noteworthy that they seem to have a constant offset, yet a similar

slope, meaning that this reduction in variables for these instances likely some only has a constant effect during the solving process for `unit-propagator-all` when compared to `cycle-prevention`.

Next, we look at the effect of filtering on each instance type. This comparison is shown in Figure 5.29. We already noted in Section 5.3.4 that `cycle-prevention` with filtering manages to solve significantly more instances than without. It is now interesting to see what instance types filtering manages to solve best compared to `cycle-prevention`.

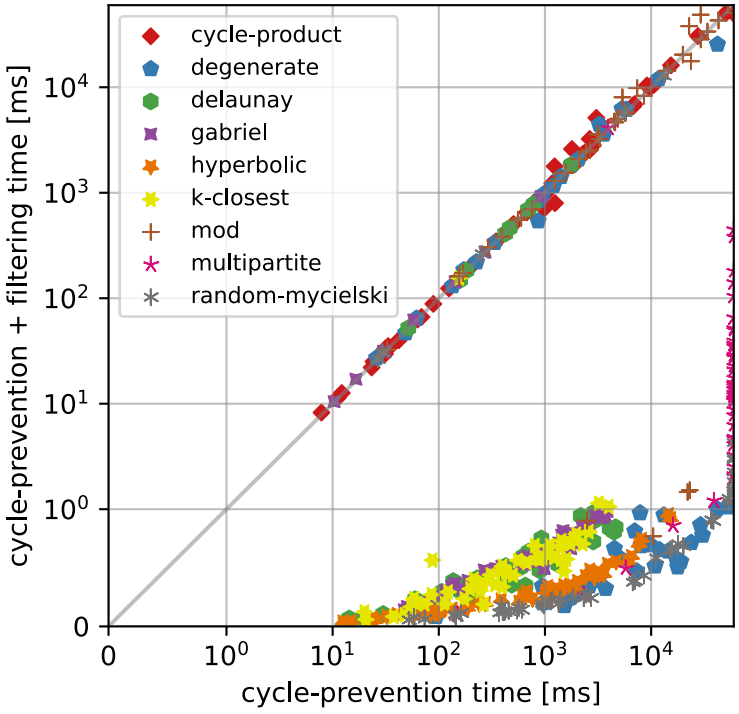

Figure 5.29: A comparison of the time needed for each instance using `cycle-prevention` with and without filtering by instance type.

In this case, we see a different split between instance types. The Delaunay, Gabriel and k-closest instances are still the type of instance where the time taken by the two approaches is rather different, but this time, the random Mycielski, multipartite, and some of the mod and degenerate instances are also there. Most interesting are the multipartite instances that are grouped up along the right edge. These instance all time out using `cycle-prevention`, but get solved by the filtering. This shows that we have an instance type hard for the SAT solver to solve, but simple when using heuristics.

We can make a similar comparison between the incremental solver and `cycle-prevention`. This comparison is shown in Figure 5.30. Again, we want to see what instance types the incremental manages to solve best compared to the non-incremental approach.

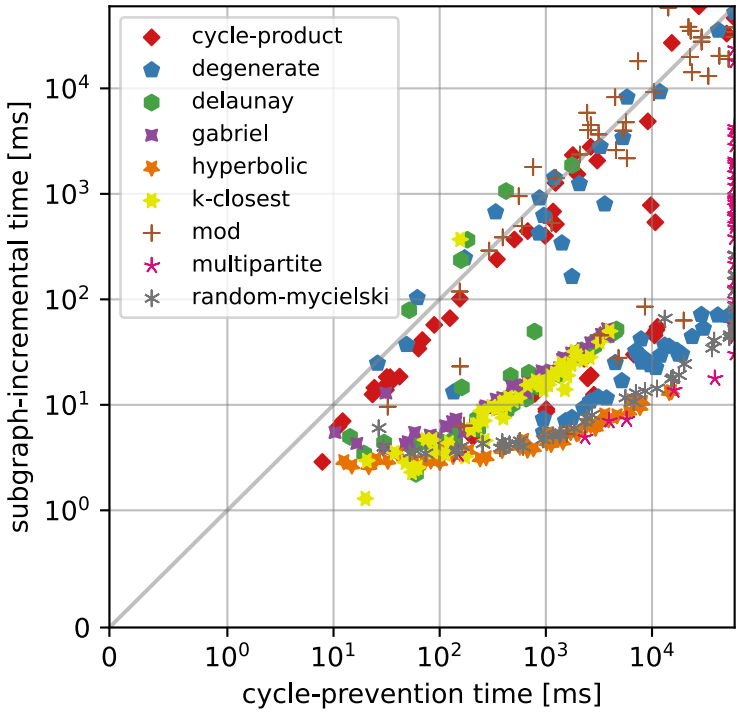

Figure 5.30: A comparison of the time needed for each instance using `cycle-prevention` and `subgraph-incremental` by instance type.

In this comparison, we see mostly the same effects as the comparison between `cycle-prevention` with and without filtering in Figure 5.29. There are two main differences: First is the effect already noted in the analysis of the plot in Figure 5.24 comparing the incremental approach to filtering, namely the incremental approach has to do some more work before actually starting the solving process. Second is a large increase in outliers, both in the instances that are solved a lot faster and those close to even. For instances where the two approaches are almost even, but the incremental approach solved it a bit faster, the reason is likely that the incremental solver found a subgraph given an unsatisfiable SAT instance, but only later in the solving process. For instances types where the incremental solver is generally a lot faster but has a few slower instances, the reason is likely that the heuristic did not find the subgraph showing that the instance is unsatisfiable.

6 Application to One-Planarity Testing

This section is meant to show the application of the ordering constraints in Section 4.1 beyond the Pattern-Avoiding Vertex Ordering problem. The specific application we look at is *one-planarity*. A graph is one-planar if it can be drawn in the plane such that every edge crosses at most one other edge. The paper [Pup25] introduces the *Optimized One-Planarity Solver* (OPS), an efficient solver for one-planarity based on SAT. OOPS encodes a partial order as part of the solving process. This section compares the approach taken in [Pup25], which is equivalent to the `encode-all` approach, to our `unit-propagator-all` approach.

6.1 Encoding of One-Planarity

Before we come to the experimental comparison, we discuss the implementation used in these experiments. OOPS uses the glucose SAT solver [AS18] instead of CaDiCaL. To make use of our user propagator implementation, we reimplement the encoding of OOPS. Our implementation does not use any of the filtering or preprocessing presented in [Pup25], however. Our implementation is therefore not directly competitive with OOPS, but is still a proof of concept for the use of our techniques in other applications.

We give a quick overview over the SAT encoding used by OOPS in this section. The intuition behind the encoding is to subdivide each edge, allow pairs of subdivision vertices to be merged, and then check the graph for planarity. If there is a one-planar drawing of the graph, then there is a planar drawing where the crossing edges have their subdivision vertices merged. We use e_{ij} to denote the newly added vertex between v_i and v_j. The new graph G' has vertices $V(G') = V(G) \cup V_S(G)$, where $V_S(G) = \{e_{ij} \mid v_i v_j \in E(G)\}$ are the vertices created by subdividing each edge. For each pair of subdivision vertices, we use the variable $m_{e_{ij},e_{kl}}$ to denote whether e_{ij} and e_{kl} are merged. We only ever need to merge subdivision vertices where the corresponding original edges do not share a vertex, so the variable $m_{e_{ij},e_{kl}}$ is only used in the encoding if $\{i, j\} \cap \{k, l\} = \emptyset$. We denote by M the pairs of vertices in G' that may be merged. Encoding that a vertex can be merged with at most one other vertex can be done using the following constraints:

$$\bigwedge_{\substack{e,f,g \in V_S(G) \\ (e,f),(e,g) \in M}} \overline{m}_{e,f} \vee \overline{m}_{e,g}$$

We now want to check the graph G' for planarity in our SAT encoding. Note that G' is bipartite with the original vertices $V(G)$ in one set and the subdivision vertices $V_S(G)$ in the other. For this planarity check, the encoding of OOPS makes use of a characterization of bipartite planar graphs presented in [FFNO11]: Let $G = (V_1 \cup V_2, E)$ be a bipartite graph with partite sets V_1 and V_2. Draw all vertices on a horizontal line, called the *spine*, in some order $\prec_G$. Draw all edges $uv \in E$ with $u \in V_1$, $v \in V_2$ and $u \prec_G v$ as a semicircle above the spine, and all other edges as a semicircle below the spine. If and only if G is planar, it is possible to choose $\prec_G$ such that the edges never cross. An example of this idea is visualized in Figure 6.1.

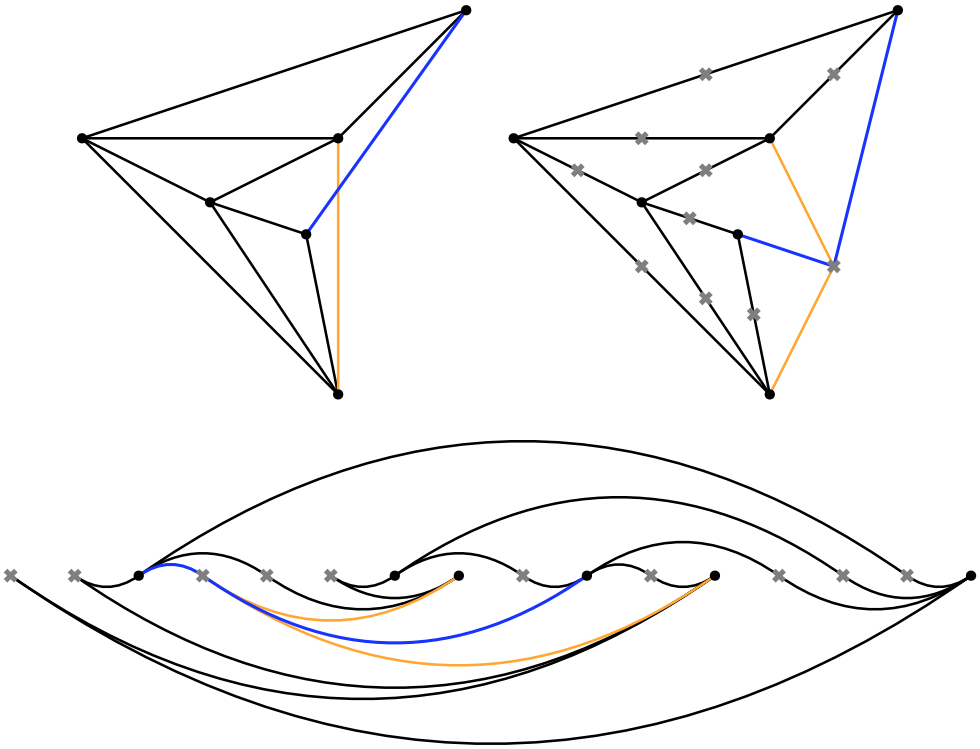

Figure 6.1: At the top left, we have a one-planar graph G. Only the two edges highlighted in blue and orange cross. At the top right, the graph G' resulting from subdividing each edge in G. Each subdivision vertex is marked by a gray x. The subdivision vertices for the crossing edges are merged, resulting in a planar graph. On the bottom is a possible ordering of G'. If the left endpoint is an original vertex, marked by a black dot, the edge is above the spine. Otherwise, the edge is below.

With this, we encode the ordering similar to how we did before: The variable $o_{u,v}$ is assigned true if the vertex u coming before the vertex v. Due to vertex merging, there is now the possibility of two vertices being in the same position if they can be merged, so $o_{u,v} = \overline{o}_{v,u}$ is not necessarily always true anymore. We now give our constraints in four parts: First, if two vertices cannot be merged, then one must come before the other (Equation 6.1). Second, for any two vertices u and v, it is impossible that u comes before v and v comes before u (Equation 6.2). Third, the ordering of vertices follows transitivity (Equation 6.3). Fourth, if two vertices are in the same position, then they are merged (Equation 6.4).

$$\bigwedge_{\substack{u,v \in V(G') \\ (u,v) \notin M}} o_{u,v} \leftrightarrow o_{v,u} \tag{6.1}$$

$$\bigwedge_{\substack{u,v \in V(G') \\ (u,v) \in M}} \overline{o}_{u,v} \vee \overline{o}_{v,u} \tag{6.2}$$

$$\bigwedge_{u,v,w \in V(G')} (o_{u,v} \wedge o_{v,w}) \rightarrow o_{u,w} \tag{6.3}$$

$$\bigwedge_{u,v,w \in V(G')} (o_{u,v} \wedge o_{v,u}) \rightarrow m_{u,v} \tag{6.4}$$

This ensures that the ordering variables give a valid ordering with the possibility to merge vertices. Next, we come to the encoding of planarity for bipartite graphs. For each edge $e \in E(G')$, let $v(e) \in V(G)$ be the endpoint in the original graph, and let $s(e) \in V_S(G)$ be the endpoint that is a subdivision vertex. With this, we encode that there are no crossings above and below the spine in Equation 6.5 and 6.6, respectively.

$$\bigwedge_{e,f \in E(G')} \overline{o}_{v(e),v(f)} \vee \overline{o}_{v(f),s(e)} \vee \overline{o}_{s(e),s(f)} \tag{6.5}$$

$$\bigwedge_{e,f \in E(G')} \overline{o}_{s(e),s(f)} \vee \overline{o}_{s(f),v(e)} \vee \overline{o}_{v(e),v(f)} \tag{6.6}$$

Last, we adapt the symmetry breaking from [Pup25]. If a valid order exists, then rotating or reversing the order yields another valid order. First, we can require a vertex v_1 to be fixed at the first position (see Equation 6.7). If there exists a valid order where the first vertex is not v_1, then we can rotate the order until v_1 is the first. Second, we can fix the order of two vertices v_2 and v_3 (see Equation 6.8). If there exists a valid order where v_3 comes before v_2, then it can reverse it.

$$\bigwedge_{2 \leq i \leq n} o_{v_1,v_i} \tag{6.7}$$

$$o_{v_2,v_3} \tag{6.8}$$

6.2 Evaluation

The experiments with our implementation use the filtered list of Wikipedia graphs already used in [Pup25]. The benchmarks described in 5.2 are generally too large for this solver to handle in a reasonable time. An added advantage of reusing this benchmark set from [Pup25] is that we can use the results from [Pup25] to verify our results, which increases the trust in our implementation. We test `encode-all` versus an adapted version of `unit-propagator-all`. We note that the implementation for the user propagator needs to be adapted from the implementation for Pattern-Avoiding Vertex Ordering due to $o_{u,v}$ and $\overline{o}_{v,u}$ being separate variables in some cases in One-Planarity Testing, that is, we are now dealing with a partial order.

The experiments were run with a 150 second timeout on the same hardware as the ones presented in Section 5.3. Figure 6.2 shows the result of these experiments, comparing `encode-all` to `unit-propagator-all` for all instances solved. In this case, there is no instance that is only solved by one of the two approaches.

For most test instances, `unit-propagator-all` performs better than `encode-all`. The average speedup of `unit-propagator-all` over `encode-all` is 2.5145. This demonstrates the viability of applying `unit-propagator-all` to problems other than Pattern-Avoiding Vertex Ordering.

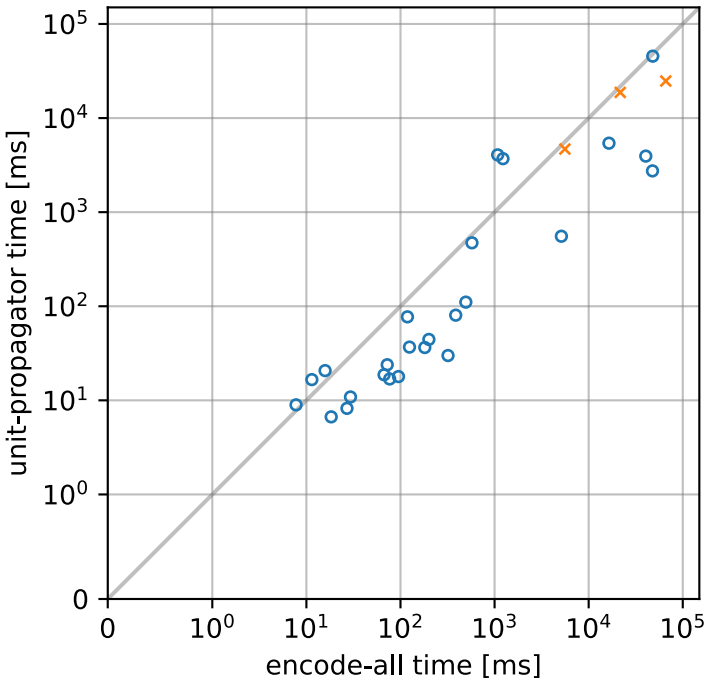

Figure 6.2: A comparison of the time needed for each instance using `encode-all` and `unit-propagator-all` for One-Planarity Testing. Satisfiable instances are marked with a blue o and unsatisfiable instances with an orange x.

7 Conclusion

Summary In this thesis, we examined the Pattern-Avoiding Vertex Ordering problem. This problem is already well examined for specific patterns, with recent overviews given by [DFHP23], [FH21b], and [FH21a]. We introduced multiple ways to solve this problem for a class of patterns using the state-of-the-art SAT solver CaDiCaL [Bie+24a], including with the newly introduced user propagators [Faz+24]. Of these solving strategies, most successful were `cycle-prevention`, a solution based on CEGAR, and `unit-propagator-all`, a new approach using user propagators. Further, we discussed incremental solving, preprocessing, filtering, different ideas for encoding the pattern avoidance, and evaluated their practical use.

In addition to practical experiments with current SAT solvers, we have investigated the problem from a theoretical perspective. Chapter 3 provides new general insight into Pattern-Avoiding Vertex Ordering, like the meaning of Pattern Chaining in Lemma 3.1 and the fact that every pattern P has at least one graph that cannot be ordered to avoid P, proven in Theorem 3.12. Further, for the complexity of ordering problems, we have established the Pattern-Avoiding Vertex Ordering problem's membership in Σ_2^{P} as shown in Theorem 3.13, the NP-hardness of the Ordered Linear 2SAT problem as proven in Lemma 4.7, and the notes on resolution proofs for ordering problems in Section 4.4.

Future work On a broader level, this thesis makes progress on the long-standing problem to efficiently encode ordering problems into SAT. To make this problem reasonably approachable, we only focused on the Pattern-Avoiding Vertex Ordering problem. There are, however, results applicable to other problems, as we have seen in Chapter 6. Here, we suggest five possible directions for further research:

- **Position-based encoding**: Another paper investigating the encoding of acyclicity in graphs is [ZWY23]. Here, an approach is discussed based on assigning every vertex a position. We discarded this approach based on preliminary experiments. There are however three potential advantages to this approach. First is the potential for additional search space pruning: When each vertex is assigned a position, it is possible to add constraints that limit how close or far apart two vertices are, not just how they are ordered. Second, we obtain smaller encodings, which can beat the lower bound of the complete encoding presented in Lemma 4.4. Third, user propagators may prove useful in enforcing such constraints.

- **Further application of the ordering constraint**: Chapter 6 demonstrates the application of `unit-propagator-all` to One-Planarity Testing. It is now natural to look for and evaluate other applications of this user propagator implementation. This ranges from very natural ordering problems, like the Hamilton Cycle problem, to more creative uses of orderings, like the planarity test we have seen in Section 6.1.

- **Data structures for user propagators**: The `cycle-prevention` and `reachability-cycle-prevention` performed similarly well. With additional engineering of the reachability data structure, `reachability-cycle-prevention` might be able to beat `cycle-`

`prevention` decisively. On a broader note, SAT solving with user propagators requires data structures with incremental addition and stack-like undoing, which sits between conventional incremental and fully dynamic data structures. Further, it might be interesting to consider the effect of branching heuristics (for example the commonly used VSIDS [Mos+01]) to create data structures with a better average case performance.

- **Pattern-avoidance clauses**: The pattern avoidance discussed in Section 4.2.1 and CEGAR approach in Section 4.2.3 only prohibits one occurrence of the pattern per clause. An interesting question is the possibility to prohibit multiple occurrences of a pattern with fewer clauses. This is particularly interesting when a pattern has a lot of automorphisms: If some vertices form an occurrence of a pattern P, then permuting them according to an automorphism of P maintains the pattern occurrence. With this, the `pattern-CEGAR` approach might become viable.
- **Improved pattern detection**: In general, our implementation uses the recursive encoding for pattern avoidance presented in Section 4.2.2 based on the pattern detection algorithm of [DFHP23]. There are, however, potentially ways to give improved encodings for specific patterns, for example with the pattern splitting idea of Lemma 3.1. Likewise, the `path-sampling-refinement` approach might become competitive using better, specialized pattern detection.

Bibliography

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009. ISBN: 978-0-521-42426-4.

[Alm+25] Josh Alman, Ran Duan, Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. "More Asymmetry Yields Faster Matrix Multiplication". In: *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2025, New Orleans, LA, USA, January 12-15, 2025*. Edited by Yossi Azar and Debmalya Panigrahi. SIAM, 2025, pp. 2005–2039. DOI: *10.1137/1.9781611978322.63*.

[AS18] Gilles Audemard and Laurent Simon. "On the Glucose SAT Solver". In: *Int. J. Artif. Intell. Tools* Volume 27 (2018), 1840001:1–1840001:25. DOI: *10.1142/S0218213018400018*.

[Bar+22] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. "cvc5: A Versatile and Industrial-Strength SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*. Edited by Dana Fisman and Grigore Rosu. Vol. 13243. Springer, 2022, pp. 415–442. DOI: *10.1007/978-3-030-99524-9_24*.

[Bie+24a] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. "CaDiCaL 2.0". In: *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*. Edited by Arie Gurfinkel and Vijay Ganesh. Vol. 14681. Springer, 2024, pp. 133–152. DOI: *10.1007/978-3-031-65627-9_7*.

[Bie+24b] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. "CaDiCaL, Gimsatul, IsaSAT and Kissat Entering the SAT Competition 2024". In: *Proc. of SAT Competition 2024 – Solver, Benchmark and Proof Checker Descriptions*. Edited by Marijn Heule, Ashlin Iser, Matti Järvisalo, and Martin Suda. Vol. B-2024-1. University of Helsinki, 2024, pp. 8–10.

[Bré79] Daniel Brélaz. "New Methods to Color Vertices of a Graph". In: *Commun. ACM* Volume 22 (1979), pp. 251–256. DOI: *10.1145/359094.359101*.

[BV02] Randal E. Bryant and Miroslav N. Velev. "Boolean satisfiability with transitivity constraints". In: *ACM Trans. Comput. Log.* Volume 3 (2002), pp. 604–627. DOI: *10.1145/566385.566390*.

[CB94] James M. Crawford and Andrew B. Baker. “Experimental Results on the Application of Satisfiability Algorithms to Scheduling Problems”. In: *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 2*. Edited by Barbara Hayes-Roth and Richard E. Korf. AAAI Press / The MIT Press, 1994, pp. 1092–1097.

[CFHI25] Cayden Codel, Katalin Fazekas, Marijn Heule, and Ashlin Iser. *The Results of SAT Competition 2025*. 2025.

[CGS04] Edmund M. Clarke, Anubhav Gupta, and Ofer Strichman. “SAT-based counterexample-guided abstraction refinement”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* Volume 23 (2004), pp. 1113–1123. DOI: *10.1109/TCAD.2004.829807*.

[CM69] E. Cuthill and J. McKee. “Reducing the bandwidth of sparse symmetric matrices”. In: *Proceedings of the 1969 24th National Conference*. New York, NY, USA: Association for Computing Machinery, 1969, pp. 157–172. ISBN: 9781450374934. DOI: *10.1145/800195.805928*.

[CM99] Gianna M. Del Corso and Giovanni Manzini. “Finding Exact Solutions to the Bandwidth Minimization Problem”. In: *Computing* Volume 62 (1999), pp. 189–203. DOI: *10.1007/S006070050002*.

[CML15] Josu Ceberio, Alexander Mendiburu, and José Antonio Lozano. “The linear ordering problem revisited”. In: *Eur. J. Oper. Res.* Volume 241 (2015), pp. 686–696. DOI: *10.1016/J.EJOR.2014.09.041*.

[Cou16] David Coudert. *A note on Integer Linear Programming formulations for linear ordering problems on graphs*. Inria ; I3S ; Universite Nice Sophia Antipolis ; CNRS, Feb. 2016, p. 33.

[Cul92] Joseph Culberson. “Iterated Greedy Graph Coloring and the Difficulty Landscape”. In: (1992). DOI: *10.7939/R3M32NH6Q*.

[DFHP23] Guillaume Ducoffe, Laurent Feuilloley, Michel Habib, and François Pitois. “Pattern detection in ordered graphs”. In: *CoRR* Volume abs/2302.11619 (2023). arXiv: *2302.11619*.

[Faz+24] Katalin Fazekas, Aina Niemetz, Mathias Preiner, Markus Kirchweger, Stefan Szeider, and Armin Biere. “Satisfiability Modulo User Propagators”. In: *J. Artif. Intell. Res.* Volume 81 (2024), pp. 989–1017. DOI: *10.1613/JAIR.1.16163*.

[FFNO11] Stefan Felsner, Éric Fusy, Marc Noy, and David Orden. “Bijections for Baxter families and related objects”. In: *J. Comb. Theory A* Volume 118 (2011), pp. 993–1020. DOI: *10.1016/J.JCTA.2010.03.017*.

[FH21a] Laurent Feuilloley and Michel Habib. *Classifying grounded intersection graphs via ordered forbidden patterns*. 2021. DOI: *10.48550/ARXIV.2112.00629*.

[FH21b] Laurent Feuilloley and Michel Habib. “Graph Classes and Forbidden Patterns on Three Vertices”. In: *SIAM Journal on Discrete Mathematics* Volume 35 (2021), pp. 55–90. DOI: *10.1137/19M1280399*.

[GJR14] Martin Gebser, Tomi Janhunen, and Jussi Rintanen. “SAT Modulo Graphs: Acyclicity”. In: *Logics in Artificial Intelligence*. Edited by Eduardo Fermé and João Leite. Cham: Springer International Publishing, 2014, pp. 137–151. ISBN: 978-3-319-11558-0.

[GM06] Walter Guttmann and Markus Maucher. “Variations on an Ordering Theme with Constraints”. In: *Fourth IFIP International Conference on Theoretical Computer Science (TCS 2006), IFIP 19th World Computer Congress, TC-1 Foundations of Computer Science, August 23-24, 2006, Santiago, Chile*. Edited by Gonzalo Navarro, Leopoldo E. Bertossi, and Yoshiharu Kohayakawa. Vol. 209. Springer, 2006, pp. 77–90. DOI: *10.1007/978-0-387-34735-6_10*.

[He+25] Huangleshuai He, Zhengyi Yang, Dong Wen, Wenqian Zhang, Michael Yu, Wenke Yang, and Wenjie Zhang. “A Survey on Efficient Graph Reachability Queries”. In: *Data Science: Foundations and Applications*. Edited by Xintao Wu, Myra Spiliopoulou, Can Wang, Vipin Kumar, Longbing Cao, Xiangmin Zhou, Guansong Pang, and Joao Gama. Singapore: Springer Nature Singapore, 2025, pp. 58–77. ISBN: 978-981-96-8295-9.

[HHS20] Kathrin Hanauer, Monika Henzinger, and Christian Schulz. “Faster Fully Dynamic Transitive Closure in Practice”. In: *18th International Symposium on Experimental Algorithms, SEA 2020, Catania, Italy, June 16-18, 2020*. Edited by Simone Faro and Domenico Cantone. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 14:1–14:14. DOI: *10.4230/LIPICS.SEA.2020.14*.

[HMP25] Deborah Haun, Laura Merker, and Sergey Pupyrev. “Forbidden Patterns in Mixed Linear Layouts”. In: *42nd International Symposium on Theoretical Aspects of Computer Science, STACS 2025, Jena, Germany, March 4-7, 2025*. Edited by Olaf Beyersdorff, Michal Pilipczuk, Elaine Pimentel, and Kim Thang Nguyen. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025, 45:1–45:21. DOI: *10.4230/LIPICS.STACS.2025.45*.

[HMR14] Pavol Hell, Bojan Mohar, and Arash Rafiey. “Ordering without Forbidden Patterns”. In: *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*. Edited by Andreas S. Schulz and Dorothea Wagner. Springer, 2014, pp. 554–565. DOI: *10.1007/978-3-662-44777-2_46*.

[HN24] David G. Harris and N. S. Narayanaswamy. “A Faster Algorithm for Vertex Cover Parameterized by Solution Size”. In: *41st International Symposium on Theoretical Aspects of Computer Science, STACS 2024, Clermont-Ferrand, France, March 12-14, 2024*. Edited by Olaf Beyersdorff, Mamadou Moustapha Kanté, Orna Kupferman, and Daniel Lokshtanov. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, 40:1–40:18. DOI: *10.4230/LIPICS.STACS.2024.40*.

[HR92] Lenwood S. Heath and Arnold L. Rosenberg. “Laying Out Graphs Using Queues”. In: *SIAM Journal on Computing* Volume 21 (1992), pp. 927–958. eprint: *https://doi.org/10.1137/0221055*.

[IB21] Ashlin Iser and Tomás Balyo. “Unit Propagation with Stable Watches (Short Paper)”. In: *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*. Edited by Laurent D. Michel. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 6:1–6:8. DOI: *10.4230/LIPICS.CP.2021.6*.

[Kar72] Richard M. Karp. "Reducibility Among Combinatorial Problems". In: *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*. Edited by Raymond E. Miller and James W. Thatcher. Plenum Press, New York, 1972, pp. 85–103. DOI: *10.1007/978-1-4684-2001-2_9*.

[KL10] Sanjiv Kapoor and Xiang-Yang Li. "Proximity Structures for Geometric Graphs". In: *Int. J. Comput. Geom. Appl.* Volume 20 (2010), pp. 415–429. DOI: *10.1142/S0218195910003360*.

[Kri+10] Dmitri V. Krioukov, Fragkiskos Papadopoulos, Maksim Kitsak, Amin Vahdat, and Marián Boguñá. "Hyperbolic Geometry of Complex Networks". In: *CoRR* Volume abs/1006.5169 (2010). arXiv: *1006.5169*.

[KSS23] Markus Kirchweger, Manfred Scheucher, and Stefan Szeider. "SAT-Based Generation of Planar Graphs". In: *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, Alghero, Italy, July 4-8, 2023*. Edited by Meena Mahajan and Friedrich Slivovsky. Vol. 271. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 14:1–14:18. DOI: *10.4230/LIPICS.SAT.2023.14*.

[LMS11] Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. "Lower bounds based on the Exponential Time Hypothesis". In: *Bull. EATCS* Volume 105 (2011), pp. 41–72.

[Mos+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. "Chaff: Engineering an Efficient SAT Solver". In: *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*. ACM, 2001, pp. 530–535. DOI: *10.1145/378239.379017*.

[MPT20] Ciaran McCreesh, Patrick Prosser, and James Trimble. "The Glasgow Subgraph Solver: Using Constraint Programming to Tackle Hard Subgraph Isomorphism Problem Variants". In: *Graph Transformation - 13th International Conference, ICGT 2020, Held as Part of STAF 2020, Bergen, Norway, June 25-26, 2020, Proceedings*. Edited by Fabio Gadducci and Timo Kehrer. Springer, 2020, pp. 316–324. DOI: *10.1007/978-3-030-51372-6_19*.

[MUPG10] Nenad Mladenovic, Dragan Urosevic, Dionisio Pérez-Brito, and Carlos G. García-González. "Variable neighbourhood search for bandwidth reduction". In: *Eur. J. Oper. Res.* Volume 200 (2010), pp. 14–27. DOI: *10.1016/J.EJOR.2008.12.015*.

[Myc55] Jan Mycielski. "Sur le coloriage des graphs". In: *Colloquium Mathematicum* Volume 3 (1955), pp. 161–162. ISSN: 1730-6302. DOI: *10.4064/cm-3-2-161-162*.

[Oha+25] Ryoga Ohashi, Takehide Soh, Daniel Le Berre, Hidetomo Nabeshima, Mutsunori Banbara, Katsumi Inoue, and Naoyuki Tamura. "SAT-Based CEGAR Method for the Hamiltonian Cycle Problem Enhanced by Cut-Set Constraints". In: *28th International Conference on Theory and Applications of Satisfiability Testing (SAT 2025)*. Edited by Jeremias Berg and Jakob Nordström. Vol. 341. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, 24:1–24:10. ISBN: 978-3-95977-381-2. DOI: *10.4230/LIPIcs.SAT.2025.24*.

[Opa79] J. Opatrny. "Total Ordering Problem". In: *SIAM Journal on Computing* Volume 8 (1979), pp. 111–114. DOI: *10.1137/0208008*.

[PCSS24] Petrica C. Pop, Ovidiu Cosma, Cosmin Sabo, and Corina Pop Sitar. "A comprehensive survey on the generalized traveling salesman problem". In: *Eur. J. Oper. Res.* Volume 314 (2024), pp. 819–835. DOI: *10.1016/J.EJOR.2023.07.022*.

[PD11] Knot Pipatsrisawat and Adnan Darwiche. “On the power of clause-learning SAT solvers as resolution engines”. In: *Artif. Intell.* Volume 175 (2011), pp. 512–525. DOI: *10.1016/J.ARTINT.2010.10.002*.

[Pup25] Sergey Pupyrev. “OOPS: Optimized One-Planarity Solver via SAT”. In: *33rd International Symposium on Graph Drawing and Network Visualization, GD 2025, Norrköping, Sweden, September 24-26, 2025*. Edited by Vida Dujmovic and Fabrizio Montecchiani. Vol. 357. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025, 14:1–14:19. DOI: *10.4230/LIPICS.GD.2025.14*.

[Rei94] Gerhard Reinelt. “The Traveling Salesman, Computational Solutions for TSP Applications”. In: vol. 840. Springer, 1994. ISBN: 3-540-58334-3. DOI: *10.1007/3-540-48661-5*.

[Sax80] James B. Saxe. “Dynamic-Programming Algorithms for Recognizing Small-Bandwidth Graphs in Polynomial Time”. In: *SIAM Journal on Algebraic Discrete Methods* Volume 1 (1980), pp. 363–369. DOI: *10.1137/0601042*.

[Sch89] Walter Schnyder. “Planar graphs and poset dimension”. In: *Order* Volume 5 (Dec. 1989), pp. 323–343. ISSN: 1572-9273. DOI: *10.1007/BF00353652*.

[Soh+14] Takehide Soh, Daniel Le Berre, Stéphanie Roussel, Mutsunori Banbara, and Naoyuki Tamura. “Incremental SAT-Based Method with Native Boolean Cardinality Handling for the Hamiltonian Cycle Problem”. In: *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings*. Edited by Eduardo Fermé and João Leite. Vol. 8761. Springer, 2014, pp. 684–693. DOI: *10.1007/978-3-319-11558-0_52*.

[Sze13] Mario Szegedy. “The Lovász Local Lemma - A Survey”. In: *Computer Science - Theory and Applications - 8th International Computer Science Symposium in Russia, CSR 2013, Ekaterinburg, Russia, June 25-29, 2013. Proceedings*. Edited by Andrei A. Bulatov and Arseny M. Shur. Vol. 7913. Springer, 2013, pp. 1–11. DOI: *10.1007/978-3-642-38536-0_1*.

[Sze25] Stefan Szeider. “SAT Modulo Symmetries: A Survey”. In: *Proceedings of the 10th International Workshop on Satisfiability Checking and Symbolic Computation (SC-Square 2025) Collocated with The 30th International Conference on Automated Deduction (CADE 2025), Stuttgart, Germany, August 2, 2025*. Edited by Madalina Erascu and Mikolás Janota. Vol. 4116. CEUR-WS.org, 2025, pp. 1–11.

[VG09] Miroslav N. Velev and Ping Gao. “Efficient SAT Techniques for Relative Encoding of Permutations with Constraints”. In: *AI 2009: Advances in Artificial Intelligence, 22nd Australasian Joint Conference, Melbourne, Australia, December 1-4, 2009. Proceedings*. Edited by Ann E. Nicholson and Xiaodong Li. Vol. 5866. Springer, 2009, pp. 517–527. DOI: *10.1007/978-3-642-10439-8_52*.

[Wil18] Virginia Vassilevska Williams. “On some fine-grained questions in algorithms and complexity”. In: *Proceedings of the international congress of mathematicians: Rio de janeiro 2018*. World Scientific. 2018, pp. 3447–3487.

[ZWY23] Neng-Fa Zhou, Ruiwei Wang, and Roland H. C. Yap. “A Comparison of SAT Encodings for Acyclicity of Directed Graphs”. In: *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*. Edited by Meena Mahajan and Friedrich Slivovsky. Vol. 271. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 30:1–30:9. ISBN: 978-3-95977-286-0. DOI: *10.4230/LIPIcs.SAT.2023.30*.