University of Bremen

Master Thesis

Solving the Directed Feedback Vertex Set Problem in Theory and Practice

Enna Gerhard

November 25, 2024

Supervision and first reviewer
Prof. Dr. Sebastian Siebertz

Second reviewer
Prof. Dr. Sebastian Maneth

Abstract. A *directed feedback vertex set* (dfvs) is a subset of vertices of a graph, that, when removed, makes the graph acyclic. The DIRECTED FEEDBACK VERTEX SET problem (DFVS) is to find such a subset. It is $\mathbf{NP}$-complete, hence, we do not know how to solve it efficiently.

In this work we explore practical approaches to find a minimum dfvs on real word instances. This was also the goal of the Parameterized Algorithms and Computational Experiments (PACE) challenge 2022. Our submission obtained the second place in the exact track and was the best student team.

Our first step of finding a minimum dfvs is to apply a set of data reduction rules. These are applied in polynomial time and reduce the size of the instance. We give a detailed overview of existing and new data reduction rules.

A kernel can be seen as a collection of reduction rules with theoretical guarantees. It is a polynomial time algorithm that takes a problem instance and returns a smaller equivalent instance with a bound on its size. We show that a very simple reduction rule removes the input restriction for the kernel of Bergougnoux et al. (2021). After its application, we obtain an instance with at most $\mathcal{O}(f^4)$ vertices, with f being the size of a minimum feedback vertex set on its cycle preserving undirected graph.

We implemented several techniques to solve a reduced instance. We achieved best results when performing an iterative reduction to HITTING SET, which we then either reduce to INTEGER LINEAR PROGRAM, MAX SAT or VERTEX COVER. Solvers for these three problems already exist, which can solve large instances in practice.

Depending on the instance, we select the best suitable solving approach. For the first time, we explain the solver that we submitted in detail and present an improved version. We compare our results with approaches of other challenge participants, in particular *DAGer*, the winner of the exact track. Our improved solver is now able to solve the same number of instances within the time limit of the PACE challenge.

Keywords: Design and analysis of algorithms, DIRECTED FEEDBACK VERTEX SET, Graph theory, Parameterized complexity

Kurzfassung.

Ein *Directed Feedback Vertex Set* (Dfvs) ist eine Teilmenge der Knoten eines Graphen, sodass der restliche Graph wenn sie entfernt würde kreisfrei wird. Das Directed Feedback Vertex Set Problem (DFVS) beschreibt die Suche nach einer solchen Teilmenge. Es ist $\mathbf{NP}$-vollständig, wir kennen also keinen Weg, es effizient zu lösen.

In dieser Arbeit erkunden wir Ansätze, ein kleinstmögliches Dfvs auf realistischen Probleminstanzen finden zu können. Dies war auch das Ziel des Parameterized Algorithms and Computational Experiments (PACE) Wettbewerbs 2022. In der Kategorie »Exakt« hat unser Wettbewerbsbeitrag den zweiten Platz erreicht und wir waren das beste Studierendenteam.

Unser erster Schritt, ein Dfvs zu finden, war die Anwendung von verschiedenen Datenreduktionsregeln. Diese werden in polynomieller Zeit berechnet und verringern die Größe der Probleminstanz. Wir stellen bekannte und neue Reduktionsregeln detailliert vor.

Ein Kernel kann als Kombination von Reduktionsregeln verstanden werden. Es ist ein Polynomialzeitalgorithmus der eine kleinere, gleichwertige Instanz mit einer beschränkten Größe errechnet. Wir zeigen, dass eine sehr einfache Reduktionsregel die Eingabebeschränkung des Kernels von Bergougnoux et al. (2021) überflüssig macht. Nach seiner Anwendung erhalten wir eine Instanz mit maximal $\mathcal{O}(f^4)$ Knoten, wobei f die Größe des kleinstmöglichen Feedback Vertex Set auf dem zugrundeliegenden ungerichteten Graphen ist, wobei wir parallele Kanten erhalten.

Wir haben verschiedene Techniken zum Finden der Lösung einer Probleminstanz implementiert. Unsere besten Ergebnisse haben wir bei einer iterativen Reduktion auf Hitting Set erzielt, welches wir weiter auf Integer Linear Program, Max SAT oder Vertex Cover reduzieren. Sogenannte *Solver* können die drei genannten Probleme auch auf großen Eingaben aus praktischer Sicht ziemlich schnell lösen.

Abhängig von der Probleminstanz wählen wir den vielversprechendsten Lösungsansatz. Wir stellen sowohl den von uns eingereichen Solver erstmals detailliert vor, als auch eine verbesserte Fassung vor. Zudem vergleichen wir unsere Ergebnisse mit denen von anderen Wettbewerbsteilnehmenden, vor allem dem erstplatzierten *DAGer*. Unser verbesserter Solver ist jetzt in der Lage, die selbe Zahl von Instanzen unter der Zeitbeschränkung des Wettbewerbs zu lösen.

Contents

1. Introduction **8**
1.1. PACE Challenge 12
1.2. Objective of this thesis 13
1.3. Structure of this thesis 14

2. Preliminaries **15**
2.1. General graph theoretic notation 15
2.1.1. Undirected graphs 16
2.1.2. Structures within graphs 17
2.1.3. Algorithms 19
2.2. Complexity Theory 21
2.3. Important NP-Complete Problems 23
2.3.1. Directed Feedback Vertex Set 25
2.3.2. Vertex Cover 26
2.3.3. Feedback Vertex Set 27
2.3.4. Hitting Set 28
2.3.5. Satisfiability 31
2.3.6. Integer Linear Programs and Linear Programs 33

3. Data Reduction Rules **36**
3.1. Formal notation 38
3.2. Reduction log 38
3.3. Visual notation 40
3.4. Structure of rules entries 40
3.5. Trivial rules 41
3.6. Existing data reduction rules 43
3.6.1. No predecessor or successor 43
3.6.2. Single predecessor or successor 44
3.6.3. Dominating bi-directed edge 46
3.6.4. Contract isolated paths of length three 47
3.6.5. Contract neighbors of degree three vertices 49
3.6.6. Crowns 51
3.6.7. Single disjoint cycle 53
3.7. Remove edges not on induced cycles 53
3.7.1. Strongly connected components 54

3.7.2. Remove directed dead ends . . . 57
3.7.3. Remove if there is no directed cycle . . . 58
3.7.4. Remove if there is always a shorter cycle . . . 59
3.7.5. Remove edges while tracking cycles in predecessors . . . 61
3.8. Pick cycle dominating vertices . . . 64
3.8.1. Pick vertices weakly dominating bi-directed edge . . . 64
3.8.2. Strongly dominating cycle . . . 65
3.8.3. Weakly dominating cycle . . . 66
3.9. Other interesting data reduction rules . . . 67
3.9.1. Too many internally vertex disjoint paths . . . 67
3.9.2. Dominated cliques . . . 69
3.9.3. Tail-Biting Worms . . . 69
3.9.4. Tunnels . . . 70
3.10. Recursive application . . . 70

4. Kernelisation **72**
4.1. Existing kernels for DFVS . . . 72
4.2. A kernel requiring a Feedback Vertex Set as input . . . 73
4.2.1. Preparing the graph and creating a first bound . . . 73
4.2.2. Bounding vertices of degree zero . . . 74
4.2.3. Bounding vertices of degree one . . . 75
4.2.4. Bounding vertices of degree greater than three . . . 77
4.2.5. Bounding the number of paths . . . 78
4.2.6. Bounding the length of paths . . . 82
4.2.7. Completing the bound . . . 83
4.3. New Kernel based on new data reduction rules . . . 84

5. Solving reduced instances **85**
5.1. Adding cycles iteratively . . . 85
5.2. Linear and partial orders . . . 89
5.3. Hints . . . 92
5.3.1. Bi-directed edges . . . 93
5.3.2. Short cycles . . . 93
5.3.3. Edge on multiple three-cycles . . . 94
5.3.4. Lower bounds of subgraphs . . . 95
5.3.5. Cliques . . . 96
5.4. Combined formulation . . . 96
5.5. Reduction to Vertex Cover . . . 99
5.5.1. Replacing cycles with Hitting Set gadgets . . . 99
5.5.2. Optimized gadgets . . . 99
5.6. Branch and bound . . . 103
5.6.1. Upper bounds . . . 104
5.6.2. Lower bounds . . . 106
5.6.3. Branch and reduce . . . 107

5.7. Combining the approaches 109

6. Practical Evaluation **111**
6.1. Dataset overview 111
6.2. Evaluation of reduction rules 113
6.3. Evaluation of solving techniques 116
6.4. Comparison with other PACE submissions 119
6.4.1. DAGer 121
6.4.2. Mount Doom 121
6.4.3. G^2OAT 121
6.4.4. DVFS 122
6.4.5. DiVerSeS 122
6.5. Creating an improved solver 122

7. Conclusion **125**
7.1. Summary 125
7.2. Further research on Directed Feedback Vertex Set 127
7.3. Solving other NP-hard problems 128
7.4. Future practical applications 129

8. Bibliography **130**

A. Appendix **139**
A.1. Thanks 139
A.2. Implementation 139
A.3. Tables 139

List of Figures **144**

List of Tables **146**

List of Algorithms **147**

Statutory declaration **148**

1. Introduction

Suppose we want to develop a new antibiotic drug against a specific type of bacteria.

Instead of developing a specific agent to target the bacterium directly, we could use existing agents to inhibit reactions that produce its food supply (Tamura et al., 2010). This food is produced in a series of chemical reactions that turn some molecules into other molecules. Evolution has made living organisms very robust and there are usually several alternative paths how a specific molecule is produced within a cell (Deutscher et al., 2006). Overall, these reactions therefore create a so called metabolic network.

Very simplified, it could look similar to Figure 1.1, with the reactions labeled R_1 to R_9 and the exchanged molecules labeled M_1 to M_9.

A typical feature of metabolic networks are cycles, where the product of a reaction, though a chain of other reactions is used again as an ingredient (Kun et al., 2008). A well known example would be the *Lactic Acid Cycle* that becomes active during intensive physical work.

Let us assume that our existing agents are each able to inhibit a single reaction. However, as there are several alternative paths how a specific molecule may be produced, we need to inhibit multiple reactions. This comes with side effects, that are caused elsewhere and outside of our current scope. To keep them at a minimum, we would search for a smallest combination of reactions to inhibit.

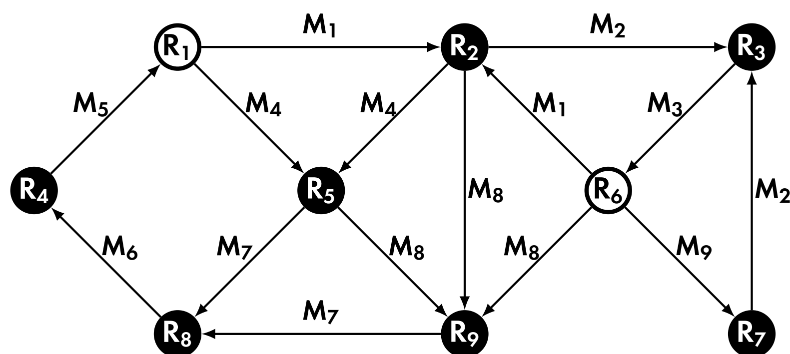


Figure 1.1.: Simplified fictive example of a metabolic network with an optimum solution

For the example in Figure 1.1, we could try to inhibit the reactions R_1 and R_6 and no cycle would remain.

Within computer science and especially parallel and distributed programming, we can run into problems because of concurrency. If two processes would read from the database at the same time and recompute the value they have just read, the second process might overwrite what the first process computed. A common technique to address this specific issue would be creating a lock on the database entries that we want to update first and only after that write the new value. When an entry is locked by a different process, we would simply wait until the computation of the other process has completed.

However, two processes might receive a lock on the first entry they want to update and then try to access the locked entry of the other process. They would now wait for each other, creating a deadlock. Furthermore, we could end up with much more complex structures of processes waiting for each other. A solution to recover from this situation would be to select a few processes that are part of the deadlock and terminate them, for example to recompute them afterwards.

We however only want to terminate as few of these processes as possible. We could try to analyze the structure of our deadlock to identify one such smallest set of processes (Großmann et al., 2022).

A possible deadlock situation is depicted in Figure 1.2. We could terminate processes P_1, P_4, P_5 and P_6 such that no deadlock would remain.

The careful reader has observed at this point, that we essentially solved the same problem twice, even though the two examples came from very different disciplines. If we are able to define our problem formally, we can try to find a general solution. Having solved the underlying principle, we can furthermore adapt it to further instances where we encounter the problem again.

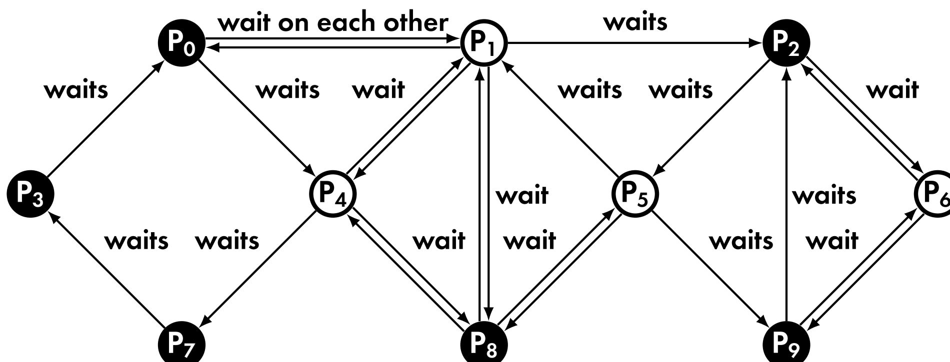


Figure 1.2.: Processes waiting on each other in a deadlock

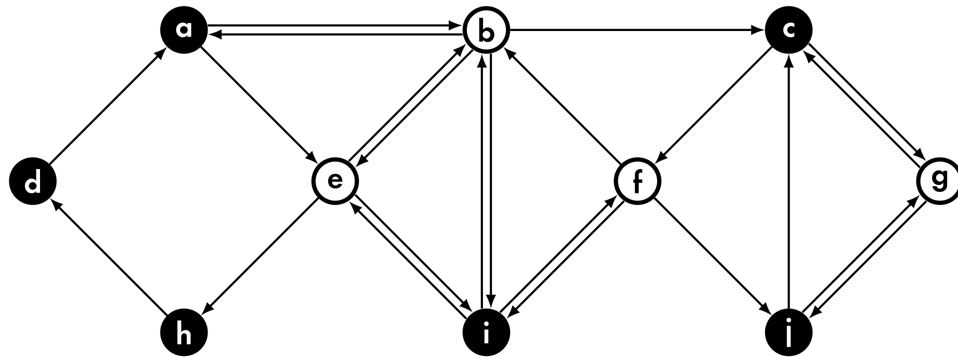


Figure 1.3.: Example graph with minimum DFVS

The natural concept for formalizing networks, relations and connectivity is that of graphs. A graph contains "vertices" to represent objects and, between these, "edges" to represent directed relations between the objects.

The graph in Figure 1.3 is equivalent to the situation within the deadlock from Figure 1.2. However, we have made it abstract and only kept the information relevant for solving the problem. This graph could also represent flow within a transportation network or resource and waste distribution in a circular economy. It could also represent articles of Wikipedia on a specific topic and how they reference each other or different buildings on a university campus and regular commutes between them. Or it could represent reactions within a cell as in the first example.

This thesis focuses on the Directed Feedback Vertex Set Problem (DFVS). DFVS is defined on directed graphs. It can be characterized as identifying a small set of vertices that, when removed, make the graph acyclic, preventing round walks in the remaining graph when only following the edges in their own direction. This set of vertices is called the directed feedback vertex set (dfvs). On the graph in Figure 1.3, the minimum dfvs would consist of vertices b, e, f and g. We will formally define DFVS in Section 2.3.1.

When we are able to find a minimum dfvs, we are able to find the smallest set of vertices that can be used to monitor or control internal circulation. We may therefore identify a set of crucial points within our network. When mapping these back to our real world, these could be the reactions we want to inhibit from our first example or the processes we need to terminate in our second example.

These vertices could also represent the inspection points for transportation services, species playing an important role in an ecosystem (Cozzens, 2015) or key processes in a circular economy. We could also measure how fail-safe a system is and how much of it could be out of order without it completely breaking down.

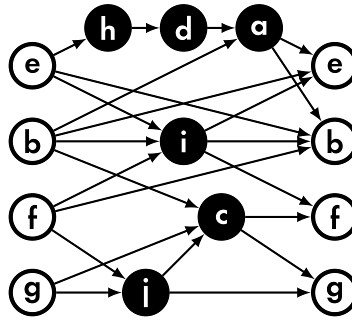


Figure 1.4.: Example of using DFVS for graph layouts

DFVS has been used for model checking in program verification (Lichtenstein and Pnueli, 1985), for very-large-scale integration (VLSI) (Leiserson and Saxe, 1991) and for circuit design, especially to improve testability using a technique called *partial scan* that benefits from a small DFVS (Chakradhar et al., 1994; Dey et al., 1994; Lin and Jou, 2000). There have been further applications in Bioinformatics such as detecting cancer genes (R. Li et al., 2021).

It is possible to use DFVS for graph layouts, especially with a small dfvs on a fairly large graph. The remaining vertices form an acyclic graph without any circular dependencies. We can thus use a hierarchical layout for the remaining vertices and need to handle special cases for the vertices that are part of a dfvs, for example using labels. This approach is depicted in Figure 1.4 on the example from Figure 1.3, repeating the vertices in the dfvs and placing the others such that all edges are drawn from left to right.

The power of graphs is that we have a very simple data structure that can be used universally, and are able to solve problems on them without being tied to a specific domain.

From a theoretical perspective, DFVS is one of the least understood classical $\mathbf{NP}$-complete problems as defined by Karp (1972). At the same time, it is among the most researched and has inspired several techniques within parameterized complexity (Lokshtanov, Ramanujan, Saurabh, et al., 2019).

A common practice for practically solving instances of $\mathbf{NP}$-complete problems is data reduction, which is a type of preprocessing. We usually perform several data reduction rules that simplify our problem instance. Afterwards, we can continue with other solving techniques such as branch-and-bound. The more reduction rules we have applied, the easier this actual solving becomes.

A kernel can be understood as a collection of data reduction rules that allow for a size guarantee of the remaining instance, depending on some measure. A common measure would be the size of the solution to the problem itself. We often search for a kernel that produces an instance of which the number of vertices is within a polynomial factor of the solution to the problem itself.

In contrast to many other problems, we do not know if such a polynomial kernel exists for DFVS. This is a major open question within theoretical computer science (Bang-Jensen et al., 2016) and for example listed first in the list of open problems in (Fomin, Lokshtanov, et al., 2019a).

At the same time, there have not been a lot of previous implementations solving DFVS in practice. It is therefore an ideal target for further research.

1.1. PACE Challenge

The *Parameterized Algorithms and Computational Experiments* (PACE) challenge is an international competition on efficiently solving theoretical problems in practice. In its 2022 iteration, its main topic was DFVS, aiming to address the aforementioned issues (Großmann et al., 2022). While its goal was to create *solvers*, efficient software systems combining several algorithms, for solving real world DFVS instances, it was also expected to inspire theoretical results.

In the 2022 iteration, there have been two tracks. On the "Exact track", exact solutions on a range of instances were computed each with a time limit of half an hour. The "Heuristic track" required finding solutions as small as possible on larger instances within five minutes.

Parts of the dataset of the PACE challenge that have been used in the heuristic track were generated using graphs from real world sources such as Social Networks, Purchase Networks and Wikipedia (Großmann et al., 2022). The other instances were generated artificially. A large portion was overall fairly sparse, but very dense locally. As such, the problem instances that our algorithms have been developed against are generally structured in ways that are realistic for real world applications (Barabási and Albert, 1999).

Parallelism was disallowed, solvers were restricted to a single processor core. It was possible to compete with almost any modern programming language. The entries had to be open source and were each accompanied by a solver description. In total, there were 26 different teams participating in the challenge with participants from 12 countries (Großmann et al., 2022).

We have participated in the PACE challenge as part of a student project (Bergenthal et al., 2022). We were very successful on the exact track, achieving the second place and completing as the best student team (Großmann et al., 2022). Among the main reasons for this success were several data reduction rules that we found and that we have explored and combined several different solving approaches. Our solver can be found in our public repository[1].

However, since the focus of this challenge was on achieving practical improvements, we did not discuss the details of our solving approach and its theoretical foundations. This can be seen as the main practical inspiration for this thesis.

[1] GRAPA, GitLab Department 3, University of Bremen
https://gitlab.informatik.uni-bremen.de/grapa/java/dfvs-solver

We continued working on DFVS from a theoretical point of view after the challenge. We were able to find kernels on several graph classes. One that is directly inspired by our practical work for the PACE challenge is presented in its own chapter in this thesis. We further obtained more results on the hardness of the Edge on Induced Cycle problem. The results have been presented at the *SOFSEM* conference (Dirks, Gerhard, et al., 2024).

We furthermore analyzed the properties of one data reduction rule that we discovered and were able to show that it can produce an exact solution to the Edge on Induced Cycle problem, if we exclude a specific graph as a directed minor. We have presented these results at the *SKILL* conference (Dirks and Gerhard, 2024).

1.2. Objective of this thesis

With this thesis, we want to give an overview how the Directed Feedback Vertex Set problem can be solved in practice. We want to support this both from a theoretical point of view and evaluate our results experimentally.

1. **We aim to collect and analyze data reduction rules that can be computed in polynomial time.**
   An important result is, that we only need to consider induced cycles. We introduce a new data reduction rule that heuristically solves the inverse Edge on Induced Cycle problem on most occasions. Furthermore, we are able to show that this rule allows us to achieve the kernel of Bergougnoux et al. (2021) without the need to supply a feedback vertex set of our instance.
2. **We implement and evaluate the data reduction rules.**
   The implementation of these rules can be found in the public repository.
3. **We want to explore practical approaches to solve the reduced instances.**
   This involved a direct reduction from DFVS to Integer Linear Program and an indirect reduction over Hitting Set to Integer Linear Program, Max SAT and Vertex Cover.
4. **We further want to compare our results with those of other participants of the PACE challenge.**
   Building upon their discoveries, we will create an improved version of our previous approach and the solver that we submitted to the PACE challenge.

1.3. Structure of this thesis

In Chapter 2, we introduce notation and the concepts used in this thesis and provide an overview of related optimization problems.

Chapter 3 provides an overview of existing, modified and novel data reduction rules. In Section 3.7, we have created a new data reduction rule that is both effective in practice and leads to an improved kernel for DFVS.

Chapter 4 gives an overview on DFVS kernelisation and explains the kernel of Bergougnoux et al. that we were able to improve – which we explain in Section 4.3.

In Chapter 5, we compare different approaches to practically solving DFVS. We were able to combine several different solving techniques towards a fairly unusual approach in Section 5.4 that lead to very substantial practical improvements. Another approach that has not been widely explored previously and was efficient on some instances is a reduction to Vertex Cover in Section 5.5.

We evaluate our solving approaches in Chapter 6. In Section 6.5, we explain how we were able to improve our existing solver.

We conclude in Chapter 7 and give a detailed overview on possible further research.

2. Preliminaries

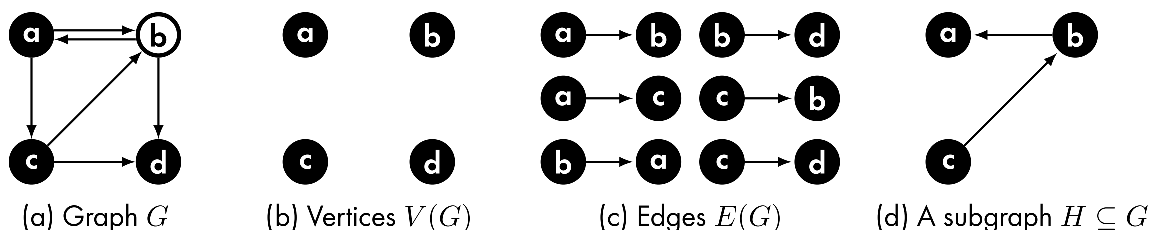
In the following, we will use standard notation from graph theory (for example Bang-Jensen and Gutin, 2009) and follow the definitions used in Bergougnoux et al. (2021) and Gerhard (2021) wherever possible. We will repeat most of the notation beyond basic mathematical notation. Definitions are *highlighted*.

2.1. General graph theoretic notation

A (directed) *graph* $G = (V, E)$ consists of a set of *vertices* $V(G) = V$ and a set of *edges* $E(G) = E \subseteq V(G)^2$, hence every edge $e \in E(G)$ is a pair of vertices $e = (u, v)$ with $u, v \in V(G)$. Unless otherwise noted, graphs are directed as we almost exclusively work on directed graphs. In the literature, directed graphs are often called *digraph*. As we defined them using sets, we will simply ignore adding vertices or edges if they have already been present.

We generally expect our graphs not to contain *loops* (w, w), as they are removed by our first reduction rule. When two vertices u, v are connected by edges (u, v) and (v, u), we refer to the vertices as connected by a *bi-directed edge* $\{u, v\}$. We write $\{u, v\} \in E(G)$ if $(u, v), (v, u) \in E(G)$. An edge (u, v) is *uni-directed* if $(v, u) \notin E(G)$. The number of vertices and edges is usually denoted as $n = |V(G)|$ and $m = |V(E)|$ respectively. If $(u, v) \notin E(G)$, we refer to (u, v) as a *non-edge*.

Graphs, for example $G = (\{a, b, c, d\}, \{(a, b), (a, c), (b, a), (b, d), (c, b), (c, d)\})$ may be represented visually, as depicted in Figure 2.1a. Vertices (2.1b) are represented as circles. They are usually labeled and, in this thesis, generally filled black if they do not have a special mean-



(a) Graph G (b) Vertices $V(G)$ (c) Edges $E(G)$ (d) A subgraph $H \subseteq G$

Figure 2.1.: Basic graph examples
(Adapted from Figure 2.1 in Gerhard, 2021)

ing. If a vertex has a special meaning or is referenced in the text, it will usually not be filled, like vertex b in the example. Edges are represented as arrows between two vertices (see 2.1c). In this example, a and b are connected by a bi-directed edge $\{a, b\}$. There are several non-edges such as (a, d) and (d, a), but also (d, c). We usually omit references to the current graph if it is clear from the context.

A *subgraph* $H \subseteq G$ is a graph that contains a subset of vertices and edges of a graph G with $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. A subset of vertices $V' \subseteq V(G)$ may be used to create an *induced subgraph* $G[V']$ with the set of vertices $V(G[V']) = V'$ and edges $E(G[V']) = \{(u, v) \in E(G) \mid u, v \in V'\}$, which is the maximal subgraph on these vertices. Figure 2.1d displays an example where both the vertex d and its incident edges, as well as two other edges $(a, b), (a, c)$, have been removed from G.

If an edge (u, v) exists, u is a *predecessor* of v and v a *successor* of u. The *predecessors* of v, denoted as ${}^{\rightarrow}N[v]_G = \{u \mid (u, v) \in E(G)\}$ are all predecessors of v in G. Analogously, the *successors* of v are $N^{\rightarrow}[v]_G = \{u \mid (v, u) \in E(G)\}$. We extend these definitions to sets of vertices W while ignoring possible edges between vertices within these sets. The *in-neighborhood* of W is defined as ${}^{\rightarrow}N[W]_G = \bigcup_{w \in W} {}^{\rightarrow}N[w]_G \setminus E(G[W])$, the *out-neighborhood* $N^{\rightarrow}[W]_G$ being defined analogously. The *in-degree* of a vertex v is ${}^{\rightarrow}\delta[v]_G = |{}^{\rightarrow}N[v]_G|$, its *out-degree* is $\delta^{\rightarrow}[v]_G = |N^{\rightarrow}[v]_G|$.

Contracting two vertices $u, w \in E(G)$ into a vertex v_{uw} creates a new graph G', where u and w are merged into a single vertex v_{uw}, retaining the neighborhoods of both. We replace the two vertices with a single new vertex: $V(G') = (V(G) \setminus \{u, w\}) \cup \{v_{uw}\}$ and merge their neighborhoods:

$$\begin{aligned} E(G') = &\{(s, t) \in E(G) \mid s, t \notin \{u, w\}\} \\ &\cup \{(s, v_{uw}) \mid s \in {}^{\rightarrow}N[\{u, w\}]\} \\ &\cup \{(v_{uw}, t) \mid t \in N^{\rightarrow}[\{u, w\}]\} \end{aligned}$$

We contract (bi-directed) edges by contracting their endpoints.

Shortcutting a vertex $v \in V(G)$ is the operation of removing it, $V(G') = V(G) \setminus \{v\}$, and connecting all of its predecessors with all of its successors:

$$E(G') = \{(s, t) \in E(G) \mid s \neq v, t \neq v\} \cup \{(s, t) \mid s \in {}^{\rightarrow}N[v]_G, t \in N^{\rightarrow}[v]_G\}$$

2.1.1. Undirected graphs

When explicitly specified, graphs may be undirected. They, as well as their related concepts are defined in the same way as directed graphs with the following exceptions:

- There are only bi-directed edges $\{u, v\}$, referred to as edges. We do not draw arrows in both directions, rather connect them with a single line.

- Neighborhoods and degrees are defined just once for each vertex, $N[v]$, $\delta[v]$.
- We allow parallel edges. We treat them as labeled edges and add a mapping $\text{parallel} : E \mapsto n \in \mathbb{N}_0$ counting the number of parallel edges. We generally ignore this unless $\text{parallel}(e) > 1$.

Let G be an undirected graph. An undirected graph H is a *minor* of G obtainable from the subgraph of a second undirected graph G using only a sequence of edge contractions. We can picture the creation of such a minor as deleting vertices and edges from a graph and then applying a set of contractions. If a graph H can be obtained by such a procedure from, it has H as a minor.

2.1.2. Structures within graphs

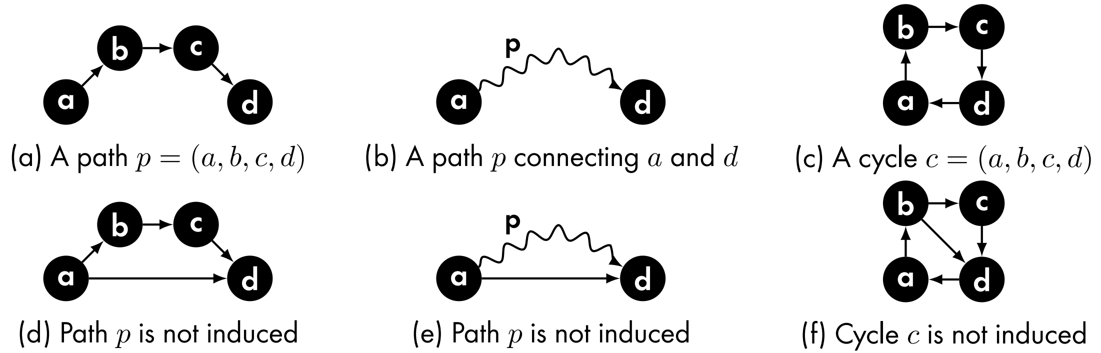
A *path* $v_1 \rightsquigarrow v_l \in G$ of length l is a sequence of distinct vertices $(v_1, \ldots, v_l)$ and connects vertex v_1 with vertex v_l by $l-1$ edges $e_1 = (v_1, v_2), e_2 = (v_2, v_3), \ldots, e_{l-1} = (v_{l-1}, v_l) \in E(G)$. In Figure 2.2a, the graph contains a path (a, b, c, d) of length four. We are sometimes only interested in the existence of a path without paying attention to the length or specific vertices. We use wavy lines to represent this visually, Figure 2.2b.

A *cycle* is a $v \rightsquigarrow w$-path closed by an edge (w, v). A cycle made up of k vertices is referred to as a k-cycle and has k edges. In Figure 2.2c, the graph contains a 4-cycle (a, b, c, d). In undirected graphs, cycles may use each bi-directed edge only once. Parallel edges form a two-cycle on their vertices.

The vertices of a path are denoted as $V(v_1 \rightsquigarrow v_n) = \{v_1, \ldots, v_n\}$. The *internal vertices* of a path are all vertices except for their end points $\text{internal}(v_1 \rightsquigarrow v_n) = V(v_1 \rightsquigarrow v_n) \setminus \{v_1, v_n\}$. A path is *induced* (sometimes called chordless) if and only if the subgraph induced by its vertices does not contain any edges other than the edges of the path itself. The same applies to induced cycles. As we do not allow loops, 2-cycles are always induced.

The path $p = (a, b, c, d)$ is induced in Figure 2.2a but not in Figure 2.2d, as there exists a shortcut from a to d. Neither is p in Figure 2.2e It would not be induced either if an edge in the other direction existed, for example (d, a), (c, b) or (c, a). The cycle on the vertices (a, b, c, d) is not induced on the graph in Figure 2.2f, but a, b, d would be.

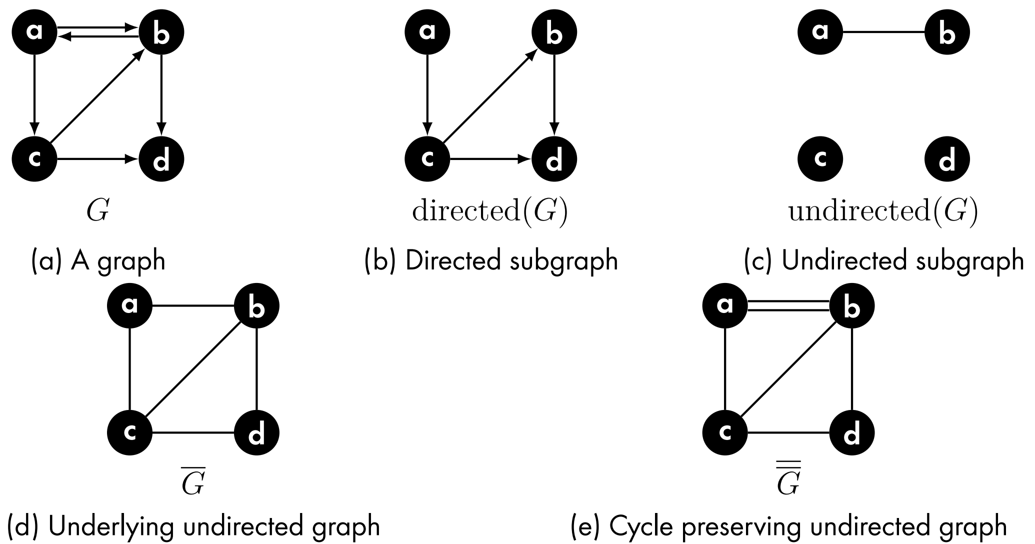
A *complete graph* K_n contains all possible edges between its $n \geq 1$ vertices, that is, for any two distinct vertices $u, v \in V(K_n), u \neq v : (u, v), (v, u) \in E(K_n)$. A *k-clique* is a set of k vertices that induce a complete graph. Contrarily, a *k-independent set* is a set of k vertices that induce an edgeless graph. We may omit the k in both cases if the size is not relevant or clear in its context.



(a) A path $p = (a, b, c, d)$ (b) A path p connecting a and d (c) A cycle $c = (a, b, c, d)$

(d) Path p is not induced (e) Path p is not induced (f) Cycle c is not induced

Figure 2.2.: Path examples



(a) A graph (b) Directed subgraph (c) Undirected subgraph

(d) Underlying undirected graph (e) Cycle preserving undirected graph

Figure 2.3.: A directed graph, split into its subgraphs and undirected structures

In some cases, only the bi-directed or uni-directed edges of a graph G are relevant. They form the *directed subgraph* $\mathrm{directed}(G) = \{V(G), \{(s,t) \in E(G) \mid (t,s) \notin E(G)\}\}$ and the *undirected subgraph* $\mathrm{undirected}(G) = \{V(G), \{\{s,t\} \mid (s,t), (t,s) \in E(G)\}\}$. Their union is the original graph. For a directed graph G, the *underlying undirected graph* is created by converting all edges to bi-directed ones, $\overline{G} = (V(G), \{\{u,v\} \mid (u,v) \in E(G)\})$. The *cycle preserving undirected graph* $\overline{\overline{G}}$ of a graph G has the same set of vertices and edges, however it counts bi-directed edges of the original graph as two parallel edges: $\mathrm{parallel}(\{u,v\}) = |\{(u,v),(v,u)\} \cap E(G)|$. An example is shown in Figure 2.3.

An *induced partial order* $\leq$ over a set A, assigns each element a *rank*. In our case, we usually create orders for vertices of a graph with $A = V(G)$. An element might share a rank with another element $\mathrm{rank}(a) = \mathrm{rank}(b)$ for $a \neq b$ for $a, b \in A$ if we do not restrict it with additional constraints. For any $a, b, c \in A$, the following conditions must hold:

1. If $\mathrm{rank}(a) \leq \mathrm{rank}(b)$ and $\mathrm{rank}(b) \leq \mathrm{rank}(c)$, then $\mathrm{rank}(a) \leq \mathrm{rank}(c)$.

2. If $\text{rank}(a) \leq \text{rank}(b)$ and $\text{rank}(b) < \text{rank}(c)$, then $\text{rank}(a) < \text{rank}(c)$.
3. If $\text{rank}(a) < \text{rank}(b)$ and $\text{rank}(b) \leq \text{rank}(c)$, then $\text{rank}(a) < \text{rank}(c)$.

A graph G is *acyclic* if it does not contain any cycles. For no $v, w \in V(G), v \neq w$ both $v \rightsquigarrow w$ and $w \rightsquigarrow v$. In the literature, it is sometimes called DAG, short for directed acyclic graph. We can impose a *linear order*, sometimes called *topological order*, on an acyclic graph, that is a special case of a partial order, such that:

1. For all pairs of vertices $v \neq w$ $\text{rank}(v) \neq \text{rank}(w)$.
2. For all edges $(s, t) \in E(G)$ the vertices adhere to the order, $\text{rank}(s) < \text{rank}(t)$.

A *strongly connected component* is a maximal subgraph $G' \subseteq G$ in which all vertices are pairwise reachable by a path. For each $v, w \in V(G')$, $v \rightsquigarrow w$ and $w \rightsquigarrow v$. All vertices of an acyclic graph are their own individual strongly connected components. An *s–t-cut* is a set of vertices $C \subseteq V(G)$ such that no $s \rightsquigarrow t$-path in $G[V(G) \setminus C]$ exists.

Similarly, a *forest* is an undirected graph that does contain any cycle. Vertices of a *connected component* $G'' \subseteq G$ are pairwise reachable on an undirected subgraph $H \subseteq \overline{G}$. A *tree* is a forest that only contains exactly one connected component.

Finally, a *class* of graphs $\mathcal{G}$ with a certain *property* is a set of graphs, such that each graph $G \in \mathcal{G}$ fulfills this property. An example could be *planar graphs* that, simply put, is the class of all graphs that can be drawn on the plane without crossing lines. There are also properties assigning a value to a graph based on a specific measure such as treewidth, where all graphs that are assigned the same value are in the same graph class. Another common definition of graph classes are based on excluding a specific graph as a minor or using one of the directed adaptations of this concept.

2.1.3. Algorithms

We use $\mathcal{O}$-notation to describe the worst case running time of algorithms while hiding constant factors. For an input size x, its actual function $g(x)$ will be in $\mathcal{O}(f(x))$ if there are fixed constants c, x_0 such that for all $x \geq x_o$, the value $g(x) \leq c \cdot f(x)$ (Knuth, 1976). We will mostly use functions over the number of vertices and edges, n and m of graphs.

Depending on the function $f(x)$ describing the worst case running time in $\mathcal{O}(f(x))$ of an algorithm, we say that it is *constant* for $f(x) = 1$, *linear* for $f(x) = x$, *quadratic* for $f(x) = x^2$ and *cubic* for $f(x) = x^3$. If there is a fixed c for $f(x) = x^c$, it is *polynomial*, for $f(x) = c^x$ it becomes exponential. Although worse dependencies exist, they are generally not relevant within our context.

There are some basic algorithms that are used and referenced throughout the thesis.

We can use a breadth first search (BFS) to efficiently find a set of vertices that a vertex v can reach, Algorithm 2.1. We start with the successors v and iteratively add successors that we can reach from the reachable vertices to the set of reachable vertices. We can furthermore modify this search to find a shortest $s \rightsquigarrow t$-path. For vertex, we track the source from which we first encountered it and afterwards trace back this shortest path. We write $s \rightsquigarrow t = \text{BFS}(s, t)$ for searching this shortest path. Since we inspect each edge at most once, this is possible in $\mathcal{O}(m)$.

Algorithm 2.1: Breadth First Search

Input: Graph G, source vertex s
Output: The set of all vertices reachable from s

```
1  Q := {s}                        // Initialize queue with single queued vertex
2  R := ∅                                 // Track vertices that we have reached
3  while Q is not empty do
4  |   v := poll(Q)                 // Select and remove first element in queue
5  |   for each w ∈ N→[v] do
6  |   |   if w ∉ R then
7  |   |   |   R ← R ∪ {w}                         // Remember not to add w again
8  |   |   |   if w ≠ s then
9  |   |   |   |   Q ← Q ∪ {w}              // Add w to the queue if it is not s
10 |   |   |   end
11 |   |   end
12 |   end
13 end
14 return R                        // All vertices we have visited in our search
```

We can efficiently find minimum s–t-cuts using the method of Ford and Fulkerson (1956) and its efficient implementation by Dinitz in 1970 (Dinitz, 2006). It applies an iterative search to augment a set of disjoint paths that have been found. As our graphs are not weighted, we assume a unit weight, allowing this search to complete in $\mathcal{O}(\sqrt{n} \cdot m)$ (S. Even and Tarjan, 1975).

We will subsequently use pseudo-code to specify algorithms in a formal way. We already used it in Algorithm 2.1 for BFS. It is closely resembling constructs found in programming languages. Input and output explain which data structures are provided to the algorithm and what it returns, sometimes also how this output can be interpreted and used. We initially assign values with $:=$ and reassign values with $\leftarrow$. We generally assume efficient data structures and that basic operations such as poll on queues are known and use notation otherwise used in the context of sets. We use `// comments` to additionally explain the working of the algorithm in natural language.

2.2. Complexity Theory

We already introduced several problems in the introduction informally. Formally, a *parameterized decision problem* is a language $L \subseteq \Sigma^* \times \mathbb{N}$ over a fixed, finite alphabet Σ. For an instance $(x, k) \in \Sigma^* \times \mathbb{N}$, the string x may encode some information, in our case often a graph, while the positive integer k is called the *parameter* (Fomin, Lokshtanov, et al., 2019c). An algorithm that receives (x, k) and correctly returns true if and only if $(x, k) \in L$ *decides* L, solving the decision problem.

In general, we want to solve problems fast and obtain an exact solution on all instances.

Complexity theory studies properties of problems especially linked to the running time and space in memory required for the execution of algorithms solving it. These are called *time complexity* and *space complexity*. We only consider time complexity in this thesis. For being able to execute algorithms in practice, there is a major difference between problems that can be solved in polyomial time and problems requiring exponentially or an even worse number of computation steps.

The class $\mathbf{P}$ contains all problems for which an algorithm deciding it within polynomial time exists. Problems in $\mathbf{NP}$ could be solved within polynomial time if we are able to use non-determinism within our algorithms, effectively being able to compute several paths in parallel. Obviously $\mathbf{P} \subseteq \mathbf{NP}$, while a major open problem within computer science is determining if $\mathbf{P} = \mathbf{NP}$. As a standard assumption in complexity theory, we assume that $\mathbf{P} \neq \mathbf{NP}$.

We can verify a solution to a problem in $\mathbf{NP}$ within polynomial time. This is called a *certificate*. Since we could perform this verification while trying out all exponentially many possible certificates, we can solve all problems in $\mathbf{NP}$ in exponential time using a *brute-force search*.

A *reduction* is an algorithm that takes an instance $(x, k) \in \Sigma^* \times \mathbb{N}$ of a problem $L \subseteq \Sigma^* \times \mathbb{N}$ and returns an instance (x', k') of another problem $L' \in \Sigma' \times \mathbb{N}$ such that $(x', k') \in L'$ if and only if $(x, k) \in L$. It *reduces* L to L'.

We are able to reduce all problems in $\mathbf{NP}$ within polynomial time to problems that are $\mathbf{NP}$-hard. While $\mathbf{NP}$-hard problems may be outside of $\mathbf{NP}$, problems that are both in $\mathbf{NP}$ and are $\mathbf{NP}$-hard are $\mathbf{NP}$-complete. That all these $\mathbf{NP}$-complete problems are essentially equally difficult to solve was first discovered by Karp in 1972.

DFVS is a classical $\mathbf{NP}$-complete problem as defined by (Karp, 1972). Examples of other classical $\mathbf{NP}$-complete problems that we will discuss in Section 2.3 are Vertex Cover, Hitting Set and Satisfiability.

Since DFVS is $\mathbf{NP}$-complete, if we want to obtain a correct solution and use an algorithm to solve it, we expect to need an exponential number of steps with respect to the input size $n + m$ in the worst case.

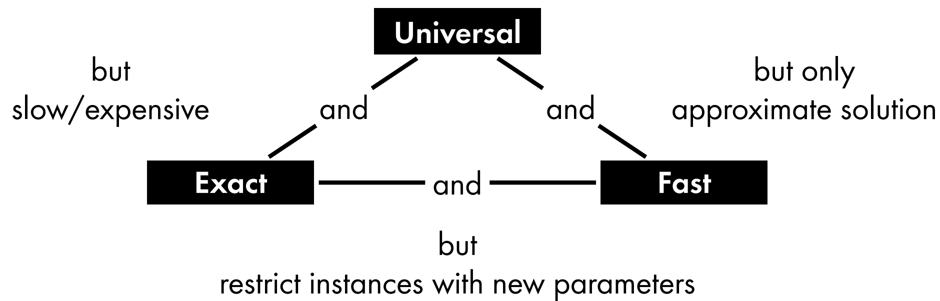


Figure 2.4.: Conflicting properties of algorithms
(Figure 1.2 from Gerhard, 2021)

We are, however, able to create efficient algorithms if we abandon one of our desired guarantees as depicted in Figure 2.4. If we did not require our solution to be exact, we might produce a much faster algorithm. We generally call this a heuristic.

If we have a guarantee with respect to the optimal solution size, we call this an approximation. For example, we could try to create a c-approximation, that is an algorithm that will run in polynomial time on all instances and will output a result at most c-times as large as the minimum solution. DFVS, however, is Approximable-Hard, so we cannot hope for such a constant factor approximation (Karp, 1972; Sun, 2024). The best approximation factor we know is $\mathcal{O}(\log(k)\log(\log(k)))$ (G. Even et al., 1998).

However, we can extract a function $f(k)$ of the parameter k to gain a more fine grained view on computation time. If extracting such a function allows us to create an algorithm with a running time $f(k) \cdot n^{\mathcal{O}(c)}$ for a fixed constant c, the problem is called *fixed parameter tractable*, $\mathbf{FPT}$. For all instances of the same parameter size, we are then able to solve the problem within polynomial time depending otherwise only on the input size n. In our case this would usually be the solution size k.

DFVS is $\mathbf{FPT}$ when parameterized by its solution size k, since the algorithm of Chen et al. (2008) completes within at most $k! \cdot 4^k \cdot n^{\mathcal{O}(1)+1}$ steps. We effectively obtain an efficient algorithm for the class of all graphs with a dfvs of size k. The parameter hides the actual hardness of the problem, such that $f(k)$ itself has to be exponential at best, since we assume $\mathbf{P} \neq \mathbf{NP}$. In our case using the algorithm by Chen et al., it is even factorial.

An important result within parameterized complexity is that if a problem is $\mathbf{FPT}$, there exists a kernel (Fomin, Lokshtanov, et al., 2019c, Theorem 1.4). A kernel is an algorithm that takes an instance of a problem, runs for limited time and returns an instance of the same problem that has a size with a guarantee dependent on the parameter. Using the approach explained by Fomin, Lokshtanov, et al., we can execute the algorithm by Chen et al. for $k! \cdot 4^k \cdot n^{\mathcal{O}(1)+1}$ steps. If it decided the problem, we effectively return its answer using a constant size yes-instace or no-instance, respectively. Otherwise, we return the instance itself, having ensured that it has a size smaller than $k! \cdot 4^k$ since we would have been able to solve it otherwise.

We are generally more interested in the existence of a polynomial size kernel, since it easily translate to practical applications. We could first kernelize the instance and after that apply non-polynomial time techniques directly solving the instance, for example branching algorithms that improve upon using regular brute-force search.

For DFVS such a polynomial kernel has not been found for general instances, nor has it been ruled out (Großmann et al., 2022). While making further standard assumptions, there are problems that are **FPT** for which no polynomial kernel exists (Bodlaender, Downey, et al., 2009).

2.3. Important NP-Complete Problems

While the focus of this thesis is on solving DFVS, there are three problems that are closely related: Vertex Cover, Feedback Vertex Set and Hitting Set.

Furthermore, Satisfiability, its extensions and Integer Linear Program are very problems that are well suitable for reduction. We will discuss them in further detail since there are well optimized programs, called SAT- and ILPsolvers that can solve them.

All of these problems are usually given as decision problems. For a graph problem, this would be given an integer k and a graph G, can we find a solution of size k for G?

Most problems also have optimization variants. This view is helpful for many practical applications in general and the PACE challenge in particular as it was about a minimization problem. What is a dfvs S on a graph G, such that if it has size $k = |S|$, there is no dfvs of size $k - 1$ on G. We will thus state all of the following problems in their optimization variant. For other problems such as Max SAT, a maximization may be the natural optimization.

They however are closely related. In our case, instead of deciding whether a graph contains a dfvs of a specific size, we will compute a minimum dfvs that can be found on such an instance. It is easy to solve the decision problem when being able to solve the optimization problem: Take the instance, compute a minimum dfvs and accept if and only if the computed solution is at most as large as the requested size. If, on the other hand we already decided that a graph contains a dfvs of a specific size, we will be able to find a minimum solution at least as small.

We already give an overview over the reductions that we will explain in the following sections in Figure 2.5. This figure also includes a selection of reductions that we will not cover but help to understand the context of practical solving.

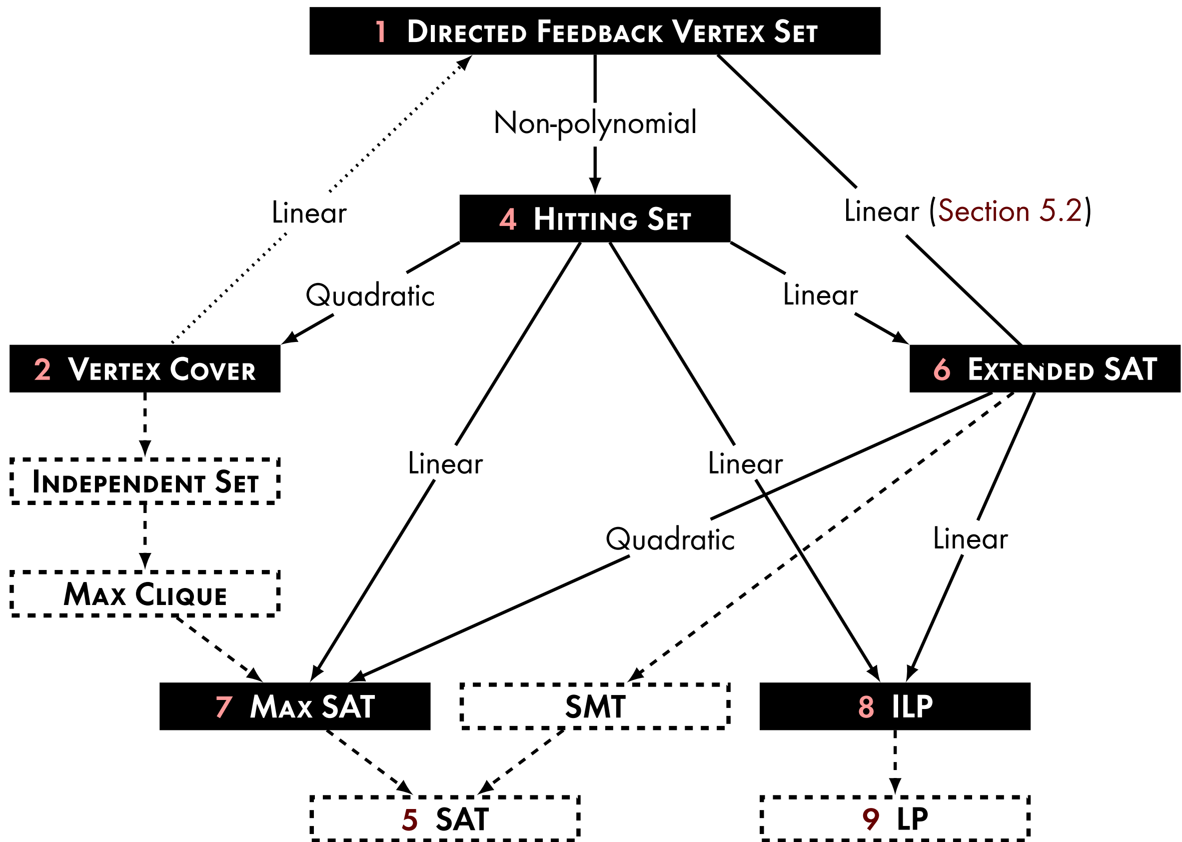


Figure 2.5.: Overview of reductions from DFVS to other problems

2.3.1. Directed Feedback Vertex Set

The Directed Feedback Vertex SetProblem DFVS, as already informally stated in the introduction, tries to cover all cycles with vertices S as possible (Fomin, Lokshtanov, et al., 2019b). A valid solution is called a *directed feedback vertex set* (dfvs) to distinguish it from the problem. Initially, it was called *Feedback Node Set* as one of Karp's classical $\mathbf{NP}$-complete problems (Problem 7, 1972). Its optimization variant is sometimes referred to as *Minimum Feedback Vertex Set* (MFVS). Sometimes, it is also called *Feedback Vertex Set on directed graphs*, however we use DFVS to distinguish it from Feedback Vertex Set, Problem 3, which is defined on undirected graphs.

Problem 1: Directed Feedback Vertex Set

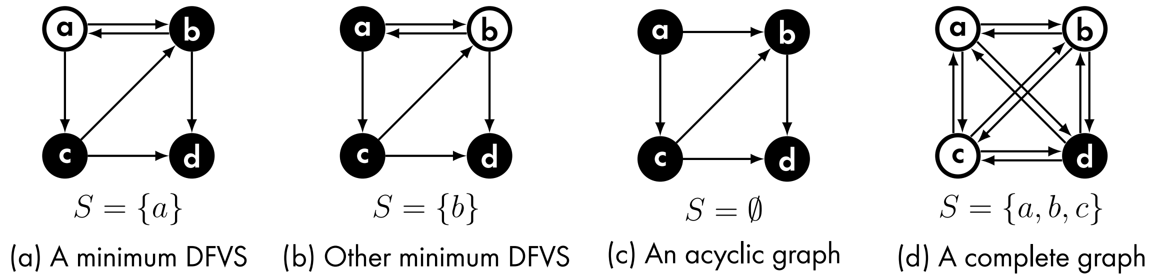
Notation: $S := \text{DFVS}(G)$
Input: A graph G
Output: A minimum size subset of vertices $S \subseteq V(G)$, such that $G[V(G) \setminus S]$ does not contain any cycles.

The decision variant of the problem is clearly in $\mathbf{NP}$. We can show this using a certificate. We take the input graph G, an existing solution S and then compute $G' = G[V(G) \setminus S]$. Since S was supposed to be a DFVS, G' now has to be an acyclic graph. We can now apply Kahn's algorithm (Kahn, 1962) on G' to verify that it is acyclic, Algorithm 2.2. We exhaustively remove vertices without predecessors or successors. If it indeed did not contain any cycles, no vertex remains. All of this is possible in quadratic time $\mathcal{O}(n \cdot m)$ since each vertex is evaluated at most twice, once having been added initially and then at most once for each of its incoming edges. In case of the decision problem, we furthermore need to verify that $|S| \leq k$ which is generally possible in linear time.

Algorithm 2.2: Verifying that a solution is a DFVS

Input: A graph G, a set of vertices S
Output: Is S a DFVS on G?

```
1  G' := G[V(G) \ S]
2  Q := V(G')                          // Create a queue from the vertices
3  while Q is not empty do
4  |  v := poll(Q)                     // Select and remove a member of Q
5  |  if →δ[v] = 0 then                // If v has no predecessor
6  |  |  Q := Q ∪ N→[v]                // Add successors of v back to the queue
7  |  |  G' := G'[V(G') \ v]           // Remove v from the graph
8  |  end
9  end
10 return V(G') = ∅                    // If the graph is empty, S was a DFVS
```



(a) A minimum DFVS (b) Other minimum DFVS (c) An acyclic graph (d) A complete graph

Figure 2.6.: DFVS examples

During the remainder of this thesis, we generally assume that we are looking for a minimum DFVS. There can be multiple minimum DFVS of the same size, as with Figures 2.6a and 2.6b. On an acyclic graph, the minimum DFVS is empty, see 2.6c. On a complete graph K_n, the size of the minimum DFVS is always $n-1$, three in case of the K_4 in 2.6d.

A parameterized algorithm that solves DFVS in $\mathcal{O}\left(k! \cdot 4^k \cdot n^{\mathcal{O}(1)}\right)$ was found by Chen et al. (2008). Internally, it uses iterative compression. It was further improved by Lokshtanov, Ramanujan, and Saurabh (2018) to $\mathcal{O}\left(k! \cdot 4^k \cdot k^5 \cdot (n+m)\right)$, such that it has a polynomial dependence on the number of vertices n as long as the solution size k remains constant. Recently, Xiong and Xiao (2024) proposed improvements to its iterative compression to obtain a running time of $\mathcal{O}\left(k! \cdot 2^{o(k)} \cdot (n+m)\right)$ while keeping its size. This is still infeasible for practical applications, especially with larger solution sizes.

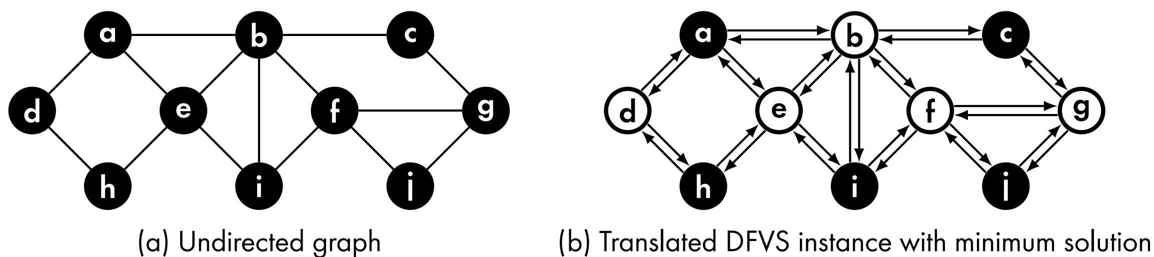
As already explained in Section 2.2, DFVS is $\mathbf{FPT}$ (Chen et al., 2008) while a polynomial kernel dependent only on the solution size has not yet been found.

2.3.2. Vertex Cover

The Vertex Cover problem tries to cover all edges with vertices W (Fomin, Lokshtanov, et al., 2019b). If W covers all edges of a graph, it is a vertex cover. It is one of the classical $\mathbf{NP}$-complete problems, initially called *Node Cover* (Problem 5 of Karp, 1972).

Problem 2: Vertex Cover

Notation: $S := \text{VC}(G)$
Input: An undirected graph G
Output: A minimum size subset of vertices $S \subseteq V(G)$, such that every edge is covered by at least one vertex that is a member of S: For all $\{v, w\} \in E(G) : v \in S$ or $w \in S$



(a) Undirected graph (b) Translated DFVS instance with minimum solution

Figure 2.7.: A reduction from Vertex Cover to DFVS

Solving Vertex Cover efficiently in practice was the objective of the PACE 2019 challenge (Dzulfikar et al., 2019). As a result, several efficient solvers exist. The most successful submission for exact solving is *WeGotYouCovered*[1]. It used several branching techniques and a reduction to Independent Set, a largest set of vertices that are not connected with edges in an undirected graph, and further to Max Clique, that attempts finding the largest possible clique in an undirected graph (Hespe et al., 2020). The solver they used for Max Clique internally used Max SAT, Problem 7, incrementally (C. M. Li, Jiang, et al., 2017).

DFVS is a natural directed translation of the Vertex Cover problem to directed graphs. A simple reduction of Vertex Cover to DFVS is as follows: For every edge $\{v, w\}$ in an undirected graph U, create edges (v, w) and (w, v). This will create a cycle between any two vertices that are adjacent in the undirected graph. Per definition, a DFVS will then need to contain either vertex in order to cover all cycles. Therefore, the vertices contained in a DFVS of such an instance will also be a valid solution for Vertex Cover. An example can be found in Figure 2.7.

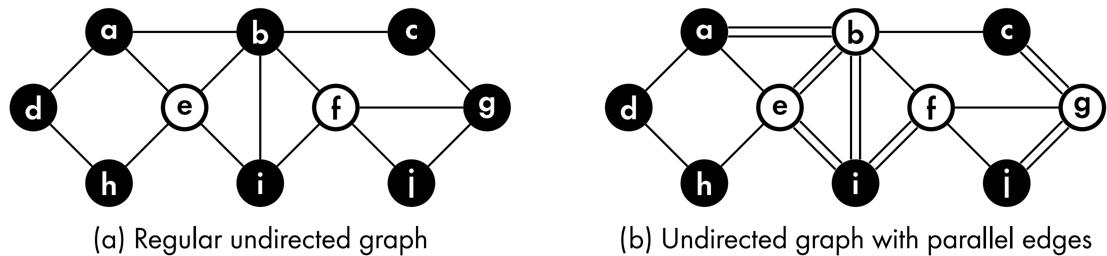
After having removed the vertices of the solution, longer cycles are not possible, since every bidirected edge is either covered by a vertex of its source or target. We would also be able to show that this reduction will always produce a minimum solution, this can be proven by contradiction.

A Vertex Cover on the undirected subgraph implies a lower bound for the DFVS on the whole graph. Since a smaller DFVS would imply the existence of a better Vertex Cover, such a better solution cannot exist: $|\text{DFVS}(G)| \geq |\text{VC}(\text{undirected}(G))|$.

2.3.3. Feedback Vertex Set

A problem on undirected graphs that is defined similar to DFVS is the Feedback Vertex Set problem (FVS), sometimes called Undirected FVS (UFVS) to distinguish it from DFVS. It is often attributed to Karp (1972), although only DFVS is directly specified in his paper.

[1] pace-2019, Karlsruhe Maximum Independent Sets, GitHub https://github.com/KarlsruheMIS/pace-2019



(a) Regular undirected graph (b) Undirected graph with parallel edges

Figure 2.8.: Examples of a minimum FEEDBACK VERTEX SET

Although it is similar defined as DFVS, it is completely different in its nature. For this problem, we want to remove vertices of an undirected graph, such that no undirected cycles remain (Fomin, Lokshtanov, et al., 2019b), creating a forest.

Problem 3: FEEDBACK VERTEX SET

Notation: $S := \text{FVS}(G)$
Input: An undirected graph G
Output: A minimum size subset of vertices $S \subseteq V(G)$, such that $G[V(G) \setminus S]$ does not contain any cycles.

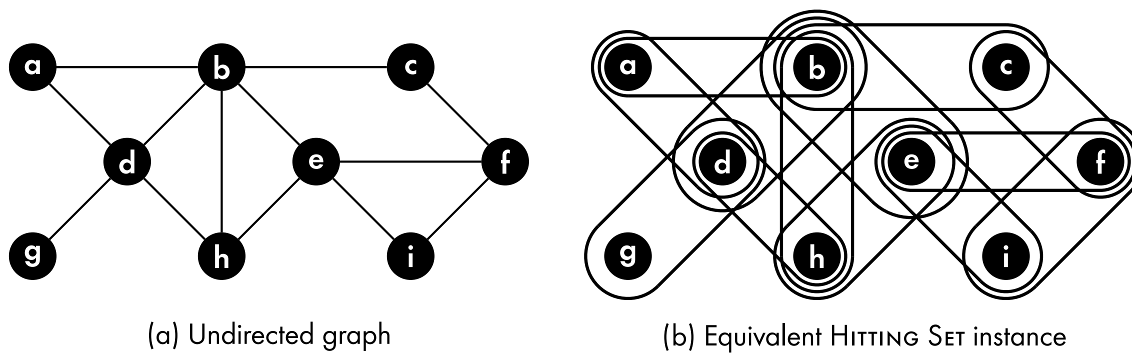
It is not trivial to reduce that problem to DFVS in polynomial time or vice-versa, as translating the undirected cycles to directed ones would create unwanted or unexpected paths that would also need to be accounted for and in the other direction, important directional information would be lost.

However, a FEEDBACK VERTEX SET solution on the cycle preserving undirected graph is immediately a valid, though not necessarily minimum dfvs, therefore $\left|\text{FVS}\left(\overline{\overline{G}}\right)\right| \geq |\text{DFVS}(G)|$.

In Figure 2.8a, we give a minimum feedback vertex set for the undirected graph from Figure 2.7a. In contrast to a minimum VERTEX COVER, this does not need to cover all edges. In Figure 2.8b, we show a minimum feedback vertex set on an undirected graph containing parallel edges. It is the cycle preserving undirected graph of the graph in Figure 1.3 on page 10 and in this case has the same size as a minimum DFVS on that graph.

2.3.4. Hitting Set

The HITTING SET problem is traditionally not defined on graphs. Instead, it is defined using a system of partially intersecting sets. These could also be represented with hypergraphs, i.e. undirected graphs containing edges with not exactly two endpoints. It is a classical $\mathbf{NP}$-complete problem (Problem 15 of Karp, 1972).



(a) Undirected graph (b) Equivalent Hitting Set instance

Figure 2.9.: A reduction from Vertex Cover to Hitting Set

Given a system of sets containing shared elements, we try to find a minimum set of vertices S such that each of the sets shares at least one member with S (Fomin, Lokshtanov, et al., 2019b).

Problem 4: Hitting Set

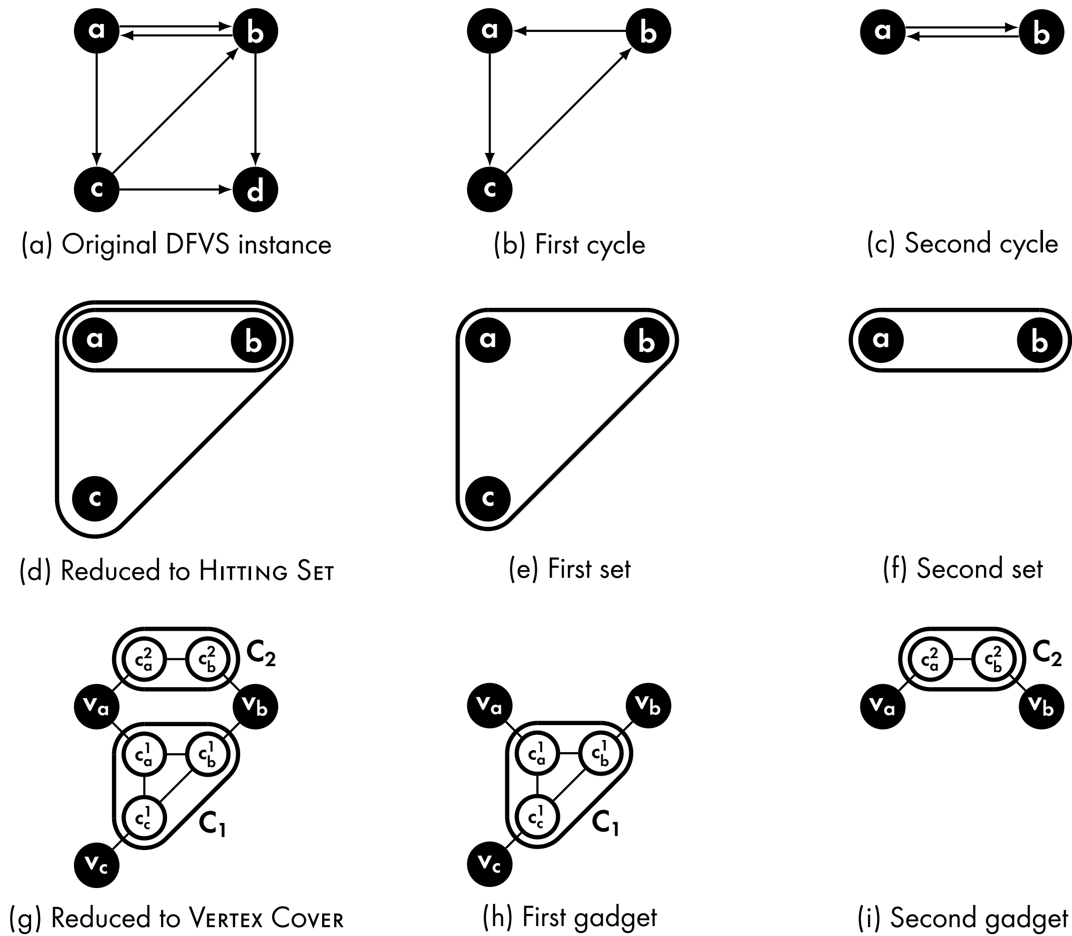
Notation: $S := \text{HS}(U, \mathcal{A})$
Input: Universe U, a family of sets $\mathcal{A}$, such that $\forall A \in \mathcal{A} : A \subseteq U$
Output: A minimum size subset of elements $S \subseteq U$, such that its intersections with all sets $A \in \mathcal{A}$ is not empty.

We will use the Hitting Set problem primarily as an intermediate step during reductions. It has a wide range of equivalent and related problems. Reducing it to other efficiently solvable problems is very simple (Ausiello et al., 1980).

A reduction from Vertex Cover to Hitting Set is straightforward. Create an element for every vertex. For every edge, add a set containing both of its endpoints. The Hitting Set solution is exactly a Vertex Cover on the respective graph. We omit the proof since it is obvious. An example for such a reduction is given in Figure 2.9.

We are able to extend the idea behind this simple reduction to a reduction from DFVS to Hitting Set. We represent the vertices of our graph as set elements. We then create a set for every cycle in the graph, containing all of its vertices. There may be exponentially many cycles (and there typically are), this reduction is therefore not polynomial. The Hitting Set is immediately a minimum DFVS.

Proof Suppose towards a contradiction that the returned DFVS S was not minimum. Then a smaller solution S' would exist. For every cycle, a vertex in S' exists. Since the Hitting Set contains sets directly corresponding to all cycles, every set contains a vertex from S'. In this case, S' would also be a smaller solution to the Hitting Set instance. This is a contradiction since S was already minimum. Therefore, we have also demonstrated that the returned DFVS is actually minimum. □



(a) Original DFVS instance (b) First cycle (c) Second cycle

(d) Reduced to Hitting Set (e) First set (f) Second set

(g) Reduced to Vertex Cover (h) First gadget (i) Second gadget

Figure 2.10.: Reduction from DFVS to Hitting Set to Vertex Cover

An example is shown in Figure 2.10. For all cycles (2.10b, 2.10c), we create sets (2.10e, 2.10f). The union of these sets is then the Hitting Set instance, 2.10d.

We are able to provide a further reduction from Hitting Set to Vertex Cover that is quadratic in size.

Given a set system $\mathcal{A}$ over our universe U we create a graph containing vertices V_{initial}. Each element $u \in U$ is associated with a different vertex $v_u \in V_{\text{initial}}$. For each set $A \in \mathcal{A}$, we add a clique C_A of the size of A. We then connect the vertex v_a of each set element $a \in A$ with a different member of the new clique $c_a^A \in C_A$ using an edge $\{v_a, c_a^A\}$.

For the resulting graph we can compute a minimum Vertex Cover S. We can translate the solution back to a Hitting Set solution. We can immediately add all set elements associated with vertices in the solution $S \cap V_{\text{initial}}$. In case the Vertex Cover selected all vertices of one of the created cliques, we need to push one of these out into our solution, taking any of the vertices of the original set. Therefore, we need to keep track of the cliques. This reduction finds a minimum solution for the Hitting Set instance.

We continue the example in Figure 2.10. We create gadgets (2.10h, 2.10i) representing the previous sets (2.10e, 2.10f). Together, these create a graph, 2.10g, of which the minimum Vertex Cover can be translated back to the Hitting Set solution, which in turn is directly a solution for DFVS.

2.3.5. Satisfiability

A formula of propositional logic in *Conjunctive Normal Form* (CNF) φ is a conjunction $\varphi = \bigwedge_{C \in \mathcal{C}} C$ of *clauses* $C \in \varphi$, which are disjunctions $C = \bigvee_{l \in L} l$ of *literals* $L \in C$, which are boolean variables $l = x$, or their negation $l = \neg x$ with $x \in X$ (Prestwich, 2021). An *assignment* ψ for variables X assigns each variable $x \in X$ either to true or false, $x \in \psi$ or $\neg x \in \psi$. An assignment ψ *satisfies* a literal if either $v \in \psi$ and $l = v$ or $\neg v \in \psi$ and $l = \neg v$. Similarly a clause C is satisfied if there is a literal $l \in C$ that is satisfied and a CNF φ is satisfied if all clauses $C \in \varphi$ are satisfied.

The Satisfiability problem (SAT) is to determine if a logic formula can be satisfied. In our case, we assume that it is already given in CNF, sometimes this problem is then called CNF-SAT (Fomin, Lokshtanov, et al., 2019b). It is usually given in its decision variant:

Problem 5: Satisfiability
Input: Variables X, a CNF formula φ over literals from X **Output**: Is there an assignment ψ that satisfies φ.

There are solvers capable of solving SAT very fast in practice, as it is an ideal target for reductions of several other problems with direct practical applications. Modern solvers use *Conflict-Driven Clause Learning* (CDCL) to iteratively discover new constraints for the problem. This makes iteratively adding constraints feasible, as already discovered constraints can be kept in such cases. Among the best performing and best documented solvers is *CaDiCal* which is using a CDCL based approach (Biere, Fazekas, et al., 2020).

While SAT as a problem is already very flexible, it does not help with our specific use case. We define Extended SAT with the following differences:

1. We also allow *integer variables* $y \in Y$. ψ assigns them an integer value $y \in [a, b]$ between and including $a \geq 1$ and $b \leq n$. They will be used in Section 5.2 on page 89.
2. We introduce *smaller than* literals $(y_1 < y_2)$ that compare two integer variables y_1 and y_2 and can be used in place of one literal per clause. They are satisfied by ψ if y_1 is assigned a smaller value than y_2. As a result, they are transitive.
3. The *size* of an assignment ψ is its count of boolean variables assigned true, $|\{x \in X \mid x \in \psi\}|$. We try to find a minimum size assignment.
4. For simplicity, we expect φ to be satisfiable.

Problem 6: Extended SAT

Input: Variables X, integer variables Y, a satisfiable CNF formula φ over literals from X and smaller than literals over Y and a size n such that $|X| = n = |Y|$.
Output: An minimum size assignment ψ that satisfies all clauses of φ

We can easily reduce Hitting Set to Extended SAT without even needing integer variables. We represent each element $a \in U$ using a boolean variable x_a, For every set in $\mathcal{A}$, we create a clause requiring any represented element to be selected, $\psi = \bigwedge_{A \in \mathcal{A}} \bigvee_{a \in A} x_a$. Using Hitting Set as an intermediate step during reduction, we can easily reduce from Vertex Cover and DFVS as well.

The DFVS instance from Figure 2.10d would be reduced to:

$$(a \vee b) \wedge (a \vee b \vee c)$$

Minimum satisfying assignments would be $\psi = \{a\}$ or $\psi = \{b\}$.

Our Extended SAT can be seen as a subset of the even more powerful Satisfiability Modulo Theories (SMT). We will not explain them in detail, but it is a long studied field of research and there are SMT solvers specifically optimized for similar use cases. They could for example encode integers with bit vectors and internally call existing SAT solvers (Barrett et al., 2021).

We attempted using Z3[2]. Although it was generally faster than ILP solvers explained in the next section, we decided against using it after it returned non-minimum solutions.

A more traditional optimization variant of Satisfiability that was successfully used by other PACE challenge participants is Max SAT, sometimes called Maximum Satisfiability. Given a logic formula in CNF that is generally not satisfiable, we try to find an assignment that satisfies as many clauses as possible (Fomin, Lokshtanov, et al., 2019b; C. M. Li and Manyà, 2021).

Problem 7: Max SAT

Input: Variables X, a CNF formula φ over literals from X
Output: An assignment ψ that satisfies as many clauses of φ as possible.

We can reduce Hitting Set to Max SAT in the same way we reduced it to Extended SAT above. However, a solution satisfying as many clauses of φ would assign all variables as true. To obtain a minimum solution, we add clauses containing each of our variables in negated form. The less variables we assign as true, the more clauses can be satisfied. However, now our solution may not satisfy one of our original clauses and instead assign several variables as false. We can prevent this by adding our original clauses a total of $n + 1$ times. Being able to satisfy such a clause now outweighs assigning a variable as false for our solution.

[2] Z3 Theorem Prover, z3, GitHub `https://github.com/Z3Prover/z3`

The example from Figure 2.10d would be reduced to:

$$\begin{aligned}&(a \vee b) \wedge (a \vee b) \wedge (a \vee b) \wedge (a \vee b) \wedge (a \vee b)\\ \wedge\, &(a \vee b \vee c) \wedge (a \vee b \vee c) \wedge (a \vee b \vee c) \wedge (a \vee b \vee c) \wedge (a \vee b \vee c)\\ \wedge\, &(\overline{a}) \wedge (\overline{b}) \wedge (\overline{c}) \wedge (\overline{d})\end{aligned}$$

We could reduce our EXTENDED SAT to MAX SAT using a similar approach. We however would need to ensure transitivity of smaller than literals which would create a quadratic number of additional constraints.

Several solvers performing well on a diverse set of instances exist, fostered by an annual challenge on solving it (Berg et al., 2024). Among the best current solvers is *EvalMaxSat*[3]. This solver performs a series of reductions to SAT (Avellaneda, 2020) following the approach of Morgado et al. (2014) and in turn uses a highly optimized solver internally.

Furthermore, they are already optimized for the operation we performed. They differentiate between hard and soft constraints, hard constraints needing to be fulfilled and optimizing the solution within the soft constraints.

2.3.6. Integer Linear Programs and Linear Programs

An INTEGER LINEAR PROGRAM (ILP) tries to find an assignment of integer values that optimizes an objective function given as a linear combination of variables while adhering to a set of constraints imposing bounds on linear combinations of their variables and only using integer values (Chong and Żak, 2008).

Its canonical form is to find a maximum solution for an objective function subject to constraints imposing an upper bound for the sum of the linear combination. In our context, we consider the dual problem. We try to find a solution that gives us a minimum on our objective function subject to a set of lower bounds. This variant could directly be translated into the canonical form (Vanderbei, 2020).

Problem 8: INTEGER LINEAR PROGRAM

Input: Variables X, an objective function f, a set of lower bound constraints C.
Output: Integer assignments S for X satisfying all constraints C with $f(s)$ being minimum.

We can generally expect ILP solvers to be optimized for our use case since since our instance can immediately be converted to its dual problem and takes on the standard form.

[3] EvalMaxSAT, Florent Avellaneda, GitHub `https://github.com/FlorentAvellaneda/EvalMaxSAT`

We can reduce EXTENDED SAT to ILP, since we required all its integer variables to contain values in a limited range. We represent boolean variables as 0 for false and 1 for true and directly reuse integer variables, keeping their bounds. For every clause C, we create a constraint with a lower bound of one. We represent a smaller than literal $(y_a < y_b) \in C$ as $y_b - y_a$. This value will be in the range $[1, n]$ if $y_b > y_a$ and between $[-n, -1]$ otherwise. Since there can be at most one smaller than literal in a clause, we can multiply the ILP variables representing EXTENDED SAT boolean variables with $n + 1$ and use the sum over the value of all represented literals.

Since we require this sum to be at least 1, our constraint is exactly fulfilled if the clause is satisfied. Any negative value occurring because of the smaller than literal not being fulfilled is outweighed if a 1 representing a boolean variable true is multiplied with $n + 1$. We obtain a positive value for a constraint if the integer variable is smaller as requested or any boolean variable is fulfilled.

The ILP for our example from Figure 2.10d would be:

Minimize

$$a + b + c + d$$

such that

$$\begin{aligned} a + b &\geq 1 \\ a + b + c &\geq 1 \end{aligned}$$

MIXED INTEGER PROGRAM (MIP), sometimes MIXED INTEGER LINEAR PROGRAM (MILP) is a superset of ILP that allows some variables to be assigned fractional values. We did not rely on these, however we used a MIP solver, *SCIP*[4] for solving ILP.

When completely removing the integrality requirement for constraints from an ILP, we refer to it as a LINEAR PROGRAM (LP) (Chong and Żak, 2008).

Problem 9: LINEAR PROGRAM
Input: Variables X, an objective function f, a set of lower bound constraints C. **Output**: Real number assignments S for X satisfying all constraints C with $f(s)$ being minimum.

In contrast to ILP, solving LP is possible in polynomial time when we limit the size, for example represented as bits, used to represent numbers handled, in effect determining the accuracy. We can solve it efficiently, for example using the simplex method, initially developed by Dantzig in 1963. While it has an exponential worst case running time, its average running time is comparable to later developed techniques that offer polynomial time guarantees (Chong and Żak, 2008; Vanderbei, 2020).

[4] SCIP Optimization Suite, Zuse Institute Berlin `https://www.scipopt.org/`

As a result, there are efficient Linear Program solvers. We used *GLOP*[5]. Solving ILPs often relies on iteratively adding further constraints to an internal LP formulation (Achterberg et al., 2008).

[5]Google's Linear Optimization Package, OR-Tools - Google Optimization Tools, Google, GitHub https://github.com/google/or-tools/tree/stable/ortools/glop

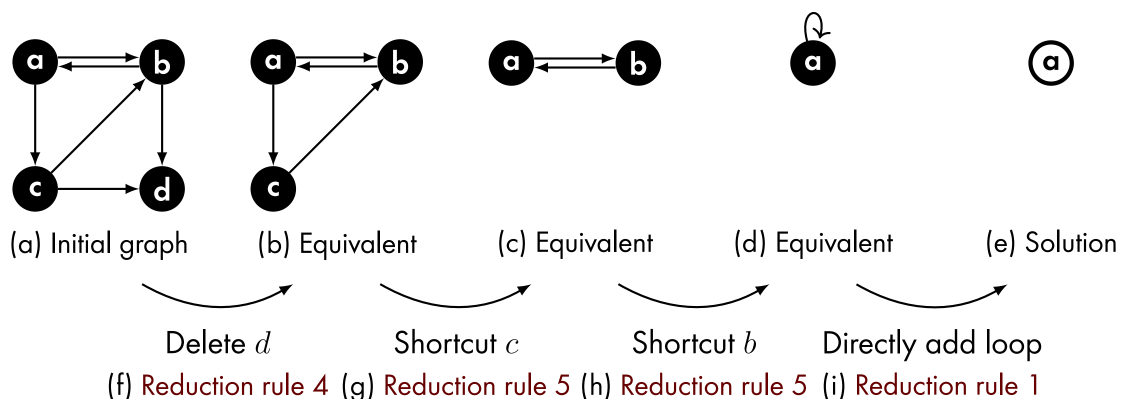
3. Data Reduction Rules

A data reduction rule, often simply called reduction rule, is a simple and efficiently computable algorithm. Given the instance of a problem, it returns an equivalent instance of the same problem that is typically smaller or in other ways more suitable for further solving. A kernel (see Section 2.2) can also be seen as a collection of reduction rules that imply a guarantee on the size of the obtained instance when applied exhaustively.

Data reduction has successfully been used as a preprocessing step in practically solving a variety of problems, such as Vertex Cover (Hespe et al., 2020), SAT (Biere, Järvisalo, et al., 2021) or MIP (Savelsbergh, 1994). It furthermore inspired several kernels, for example for Vertex Cover (Fellows, Jaffke, et al., 2018), but also for DFVS (Bergougnoux et al., 2021).

In this chapter, we collect existing reduction rules for DFVS and discuss new reduction rules that we created and implemented. We present overview in Table 3.1. It also includes reduction rules from the following chapter.

In the example in Figure 3.1, we immediately know that d does not lie on a cycle we would need to cover since, therefore we can remove it (3.1f). Instead of choosing c which can only cover cycles using exactly the edges (a, c) and (c, b), we can shortcut c, 3.1g, which would only create the already existing edge (b, a). The same would apply both for a and b now. We decide to shortcut b, 3.1h. The now created loop as a 1-cycle definitely needs to be added to the solution, 3.1i. In this case, our instance was solved only using reduction rules.



(f) Reduction rule 4 (g) Reduction rule 5 (h) Reduction rule 5 (i) Reduction rule 1

Figure 3.1.: Applying reduction rules to a DFVS instance and solving it

Table 3.1.: Overview of reduction rules

#	Rule	Match.	New n	New m	New k	Impl.
1	Shortcut loops	$\mathcal{O}(n)$	$n-1$	$\leq m-1$	k	$(\checkmark)$
2	Adjacent to $k+1$ other vertices[k]	$\mathcal{O}(m)$	$n-1$	$\leq m-2k$	$k-1$	$\times$
3	Adjacent to all other vertices	$\mathcal{O}(n)$	$n-1$	$m-2n+2$	$k-1$	$\checkmark$
4	No predecessor or successor	$\mathcal{O}(n)$	$n-1$	$\leq m$	k	$\checkmark$
5	Single predecessor or successor[†]	$\mathcal{O}(n)$	$n-1$	$\leq m-1$	k	$\checkmark$
6	Dominating bi-directed edge	$\mathcal{O}(m^2)$	$n-1$	$\leq m-2$	$k-1$	$\checkmark$
7	Remove lines	$\mathcal{O}(n)$	$n-1$	$\leq m-4$	$k-1$	$\checkmark$
8	Remove triangles	$\mathcal{O}(n)$	$n-1$	$\leq 2m-8$	$k-1$	$\checkmark$
9	Remove three independent	$\mathcal{O}(n)$	$n-1$	$\leq 2m-2$	k	👻
10	Crowns	$\mathcal{O}(n)$	$< n$	$< m$	$< k$	$\checkmark$
11	Strongly connected components[††]	$\mathcal{O}(n^2)$	n[†††]	$\leq m$[†††]	k[†††]	$\checkmark$
12	Directed dead ends	$\mathcal{O}(n^2)$	n	$\leq m-1$	k	$\checkmark$
13	No directed cycle	$\mathcal{O}(m^2)$	n	$m-1$	k	$(\checkmark)$
14	Remove ignoring shorter cycle	$\mathcal{O}(m^2)$	n	$m-1$	k	$(\checkmark)$
15	Shorter cycle in predecessors	$\mathcal{O}(m \cdot n^2)$	n	$m-1$	k	$\checkmark$
16	Weakly dominating 2-cycle	$\mathcal{O}(m^2)$	$n-1$	$\leq m-2$	$k-1$	$\checkmark$
17	Strongly dominating cycle	$\mathcal{O}(m \cdot n^2)$	$n-1$	$\leq m-2$	$k-1$	👻
18	Weakly dominating cycle	$\mathcal{O}(m \cdot n^2)$	$n-1$	$\leq m-2$	$k-1$	$\times$
19	More than $k+1$ disjoint paths[k†]	$\mathcal{O}(n^2 m \cdot \sqrt{n})$	n	$m+1$	k	$\times$
20	Floating vertex	$\mathcal{O}(n^3)$	$n-1$	$\leq m$	k	$(\checkmark)$
21	Covered directed neighborhoods	$\mathcal{O}(n^3)$	n	$\leq m$	k	$(\checkmark)$
22	Covered paths[f]	$\mathcal{O}(n^n)$	$\leq n$	$\leq m$	k	$(\checkmark)$
23	Non-contributing edge	$\mathcal{O}(n^3)$	n	$m-1$	k	$(\checkmark)$

$\checkmark$ Implemented $(\checkmark)$ ... implicitly 👻 ... but unused $\times$ Not implemented

[k] Requires a known upper bound for k.
[f] This rule becomes polynomial if the underlying FEEDBACK VERTEX SET is known
[†] May create loops, decreasing n, m and k when applying Reduction rule 1 immediately.
[††] Only a single application, resulting components can be solved individually.
[†††] Distributed across up to four created sub-instances.

We will state the reduction rules used in the kernel by Bergougnoux et al. (2021) in Section 4.2 on page 73 as they are mostly interesting in that context. We there follow the format that is used in this chapter.

3.1. Formal notation

A *data reduction rule* is an algorithm that takes a problem instance G as an input and produces another instance G' of the same problem, called *reduced instance*. It also provides instructions on how to obtain a solution for the original problem once the reduced instance has been solved. For a reduction rule to be *safe*, this solution needs to be valid and of the same (minimum) size. In our case, problem instances are always graphs.

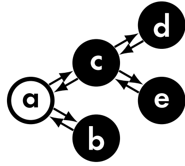
Usually, reduction rules will attempt to either reduce the number of vertices n of the graph or the number of edges m. They may also reduce the solution size k, such that the set of vertices in the solution of the reduced instance is smaller than that of the original one. We typically perform these operations directly on the current graph as creating copies would produce a large overhead.

Generally, when providing the computational complexity of rules, we will only consider the matching that the rule performs. The rewriting can usually affect the whole graph, for example if a vertex is removed by the rule, we will need to update up to $n-1$ other vertices that a vertex had been connected to, if we assume a data structure that tracks degrees of vertices. As a result, the application of a rule as a whole usually requires up to $\mathcal{O}\left(n^2\right)$ additional steps, which are often hidden by higher exponents required for matching. On sparse graphs, this becomes negligible, especially if we bound the maximum degree of vertices.

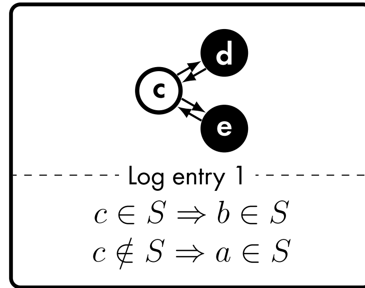
We assume an adjacency list based graph implementation. With an efficient set implementation, this allows for constant time testing for the presence of specific edges. We furthermore assume that basic counting, for example ${}^{\rightarrow}\delta\left[v\right], \delta^{\rightarrow}\left[v\right]$ is available in constant time.

3.2. Reduction log

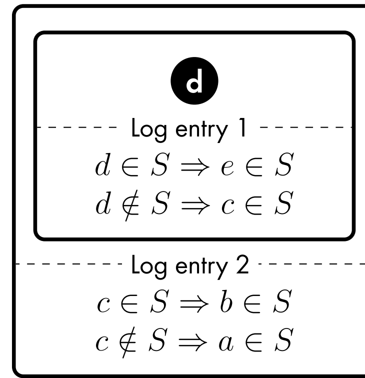
Several rules do not merely return a smaller equivalent instance but rather allow us to add vertices directly to the solution. In some cases, this depends on how the remaining instance was solved.



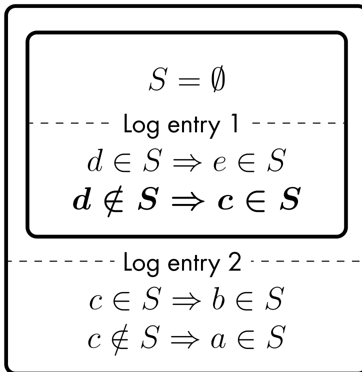
(a) Initial graph



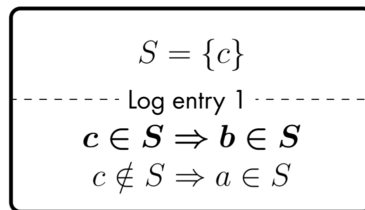
(b) Applied Reduction rule 7 on a



(c) Applied Rule 7 on c



(d) Solved instance manually



(e) Second log entry applied

$S = \{b, c\}$

(f) First log entry applied

Figure 3.2.: A log of changes to be applied after solving and reconstructing the solution

We rely on a log of instructions that will be applied backwards after the instance is solved. If a vertex v is *added to the solution*, we create a single log entry that will add it to a solution S' of the reduced instance, $S := S' \cup \{v\}$. For the reduced instance, we delete v and all edges that it was part of. Log entries can also be conditional depending on other vertices appearing in the solution of the reduced instance.

An example is shown in Figure 3.2. We apply Reduction rule 7 twice, each time creating a smaller instance and a log entry how the result should be interpreted afterwards (3.2b, 3.2c). After having solved the instance (3.2d), we apply the log in reverse order (3.2e, 3.2f). We always only consider the innermost pair of instance and log entry.

3.3. Visual notation

For many rules that are applied locally, we can separate the algorithm into two phases. During *matching*, we try to find a specific subgraph that this reduction rule can be applied to. In the *rewriting* phase, the graph is altered, vertices and edges may be added or removed.

These rules are illustrated similar to graph grammars or transformation units (Kreowski, Klempien-Hinrichs, et al., 2006; Kreowski, Kuske, et al., 2008).

They have a left hand that visualizes the state of a subgraph before the rule is applied and a right hand side that depicts the state after the rule is applied. With our visualization, we add labels to each vertex. If a label appears on both sides, it refers to the same vertex in the graph. If a label only appears on one side, it refers to a vertex being removed or added, respectively.

We may indicate that there are multiple vertices a vertex is connected with by using dots. This would not be possible with graph grammars, though we could usually define constructs of several rules that would achieve a similar behaviour. If a vertex is marked, we require it to be otherwise isolated, i.e. that all vertices it is connected with are the ones displayed. This would also require considerable amounts of work with regular graph grammars.

Usually, we allow different vertices on the right hand site of a rule to be matched on a single vertex in the actual graph. This is most common if a vertex would be both successor and predecessor of a single vertex or a vertex would be connected to two vertices that are in the focus of a rule.

Furthermore, there is a section containing the instructions on how to obtain the solution. Usually, we are able to compute the solution of the derived instance and add any vertices that are listed in this section. This section may also contain more complex instructions on how to obtain the solution of the original instance. These would be applied as described in Section 3.2.

3.4. Structure of rules entries

The reduction rules in this chapter are always stated and explained in the same schema.

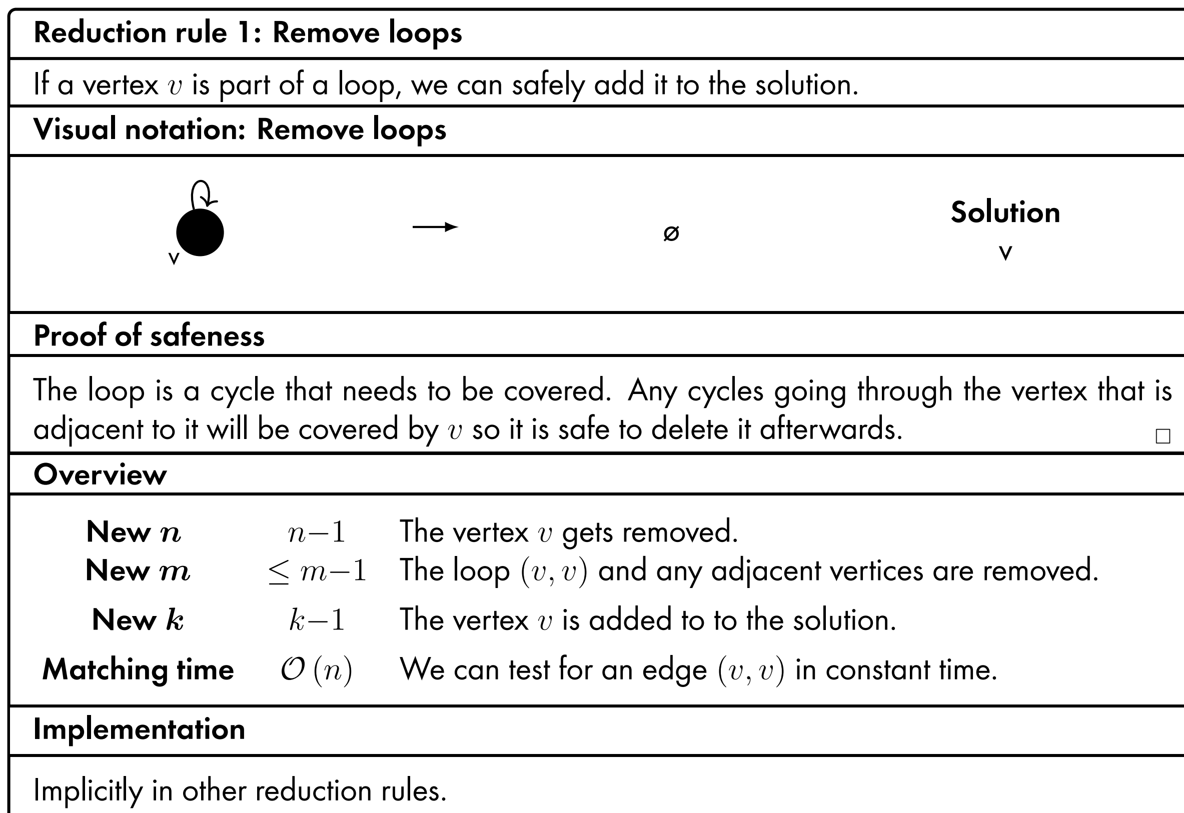
- **Rule**
  We start with a short description of the rule.
- **Visual notation**
  We use the visual notation from Section 3.3. In some cases, we use several rules for different variants.

- **Algorithm**
  If using the visual notation is infeasible, we rely on an algorithm that explains the matching for the rule.
- **Proof of safeness**
  Proof that the reduction rule is safe.
- **Overview**
  Table summarizing changes to k, n, m on a single successful application and the overall running time in $\mathcal{O}$-notation.
- **Implementation**
  A note where the implementation of the rule can be found.

We omit parts of it depending on context. We may furthermore give examples, comment on the origin of the rule or add other helpful information.

3.5. Trivial rules

The following rules are trivial and can be considered as folklore, we start with loops (Rule LOOP of Levy and Low, 1988, Lin and Jou, 2000, Swat, 2022b; Rule 1 of R. Fleischer et al., 2009).



Reduction rule 1: Remove loops

If a vertex v is part of a loop, we can safely add it to the solution.

Visual notation: Remove loops

Proof of safeness

The loop is a cycle that needs to be covered. Any cycles going through the vertex that is adjacent to it will be covered by v so it is safe to delete it afterwards. □

Overview

New n	$n-1$	The vertex v gets removed.
New m	$\leq m-1$	The loop (v, v) and any adjacent vertices are removed.
New k	$k-1$	The vertex v is added to to the solution.
Matching time	$\mathcal{O}(n)$	We can test for an edge (v, v) in constant time.

Implementation

Implicitly in other reduction rules.

The datasets are guaranteed not to contain loops. In the implementation, will immediately add vertices to S instead of allowing a loop to be created. This rule is therefore only applied implicitly within the other rules.

If we know an upper bound for k on our graph G, we can apply the following easily computable special case of Reduction rule 19 by Bergougnoux et al. (Rule 3, 2021). We explain upper bounds for DFVS in Section 5.6.1.

Reduction rule 2: Adjacent to more than k vertices

If we know a maximum solution size k and a vertex v has at least $k+1$ bi-directed edges, we can safely add v to the solution and delete it for the reduced instance.

Visual notation: Adjacent to more than k vertices

v 1 … 2 k+1 → 1 2 … k+1 **Solution** v

Proof of safeness

For all bi-directed edges, either vertex must be added to the solution. Suppose all other vertices were part of the solution. In that case, the solution would contain at least $k+1$ vertices, a contradiction to our original claim. A solution is therefore only possible when including v. □

Overview

New n	$n-1$	Vertex v is added to the solution.
New m	$\leq m-2k$	The adjacent k bi-directed edges and possible further edges connected to v are removed.
New k	$k-1$	We include v in the solution.
Matching time	$\mathcal{O}(m)$	For each vertex, we compute the cut of predecessors and successors. For this, we need to handle every edge at most twice.

Implementation

Not implemented

Although the matching is similar to Reduction rule 2, the following rule can be seen as a special and easily computable case of Reduction rule 6. Its matching can be implemented very efficiently. In practice, instead of searching through all vertices, we can take an arbitrary vertex and inspect its neighbors, since such a vertex needs to be adjacent to every vertex.

Reduction rule 3: Adjacent to all other vertices

If a vertex v lies on $n-1$ undirected edges, with $^{\rightarrow}\delta\left[v\right] = n-1$ and $\delta^{\rightarrow}\left[v\right] = n-1$, we can safely add v to the solution.

Visual notation: Adjacent to all other vertices

v 1 ... 2 n-1 → 1 2 ... n-1 **Solution** v

Proof of safeness

For all bi-directed edges, either vertex must be added to the solution. Suppose all other vertices were part of the solution. In that case, we could create a solution of equivalent size by removing any of the other vertices from the solution and including v. □

Overview

New n	$n-1$	Vertex v is added to the solution.
New m	$m-2n+2$	The adjacent n bi-directed edges are removed.
New k	$k-1$	We include v in the solution.
Matching time	$\mathcal{O}\left(n\right)$	For each vertex, we can immediately count the number of predecessors and successors.

Implementation

In `ReduceKBased`

3.6. Existing data reduction rules

We give an overview of the most important data reduction rules as found in the literature. Most of these were also implemented and used in our solver.

3.6.1. No predecessor or successor

This rule is commonly known (Rules IN0 and OUT0 of Levy and Low, 1988, Lin and Jou, 2000, Swat, 2022b; Rule 3 of R. Fleischer et al., 2009; Rule 1 of Bergougnoux et al., 2021). When applied recursively, see Section 3.10, it effectively becomes Kahn's Algorithm (1962).

Reduction rule 4: No predecessor or successor

If a vertex v has no predecessors or successors, resulting in an in- or out-degree of zero, we can remove v and any adjacent edges from the graph without affecting the solution.

Visual notation: No predecessors

Visual notation: No successors

Proof of safeness

The vertex v does not lie on a cycle. Suppose it was included in a minimum DFVS. In that case, we would be able to find another valid solution while not including this vertex as removing this vertex from the DFVS would not result in a cycle not being covered. □

Overview

New n	$n-1$	The vertex itself is removed.
New m	$\leq m$	Any remaining edges connected to the vertex are removed.
New k	k	No vertices are added to the solution.
Matching time	$\mathcal{O}(n)$	We can test the degree of each vertex in constant time. If applied successfully, we need to remove up to $n-1$ adjacent edges during each attempt, thus resulting in a total complexity of $\mathcal{O}(n^2)$.

Implementation

As part of `CombinedRecursiveReduction`, see Section 3.10.

3.6.2. Single predecessor or successor

This rule is also commonly known (Rules IN1 and OUT1 of Levy and Low, 1988, Lin and Jou, 2000, Swat, 2022b; Rule 4 of R. Fleischer et al., 2009; Rule 2 of Bergougnoux et al., 2021).

Reduction rule 5: Single predecessor or successor

If a vertex v has a single predecessor or successor u, we can shortcut v. As a special case to handle Reduction rule 1, if the single edge leads to a vertex w that is also connected in the opposite direction, we add w immediately to the solution instead of creating a loop.

Visual notation: Single predecessor

u v 1 … 2 i → u 1 … 2 i

Solution

∅

Visual notation: Single successor

1 2 … i v u → 1 2 … i u

Solution

∅

Proof of safeness

All cycles that involve v contain the edge (u, v) or (v, u), respectively.
This does not increase the solution size and still produces a valid solution. Assume that v was included in the solution. In this case, selecting u would lead to a solution of the same size and would cover all cycles through (u, v).
This rule also does not decrease the solution size. For any existing $v \rightsquigarrow u$-path that was closed to a cycle with the edge (u, v), there is now an equivalent $u \rightsquigarrow u$-path, since all successors of v are now successors of u. The reverse direction is analog. □

Overview

New n	$n-1$	The vertex v gets removed.
New m	$\leq m-1$	The edge (u, v) is removed. If u shared successors or predecessors, respectively, with v, these are "merged", decreasing the number of edges
New k	k	No vertices can be added to the solution unless the rule application creates a loop, in this case it would become $k-1$ because of immediately applying Reduction rule 1.
Matching time	$\mathcal{O}(n)$	We can test the degree of each vertex in constant time. If applied successfully, we need to remove up to $n-1$ adjacent edges during each attempt, thus resulting in a total complexity of $\mathcal{O}(n^2)$

Reduction rule 5: Single predecessor or successor (continued)
Implementation
As part of `CombinedRecursiveReduction`, see Section 3.10.

After having applied this rule and Reduction rule 4 exhaustively, all remaining vertices have at least two incoming and two outgoing edges.

3.6.3. Dominating bi-directed edge

A similar rule exists for Vertex Cover (Rule 6 of Fellows, Jaffke, et al., 2018). This is one way to generalize it for DFVS, also allowing the vertices to be adjacent to uni-directed edges.

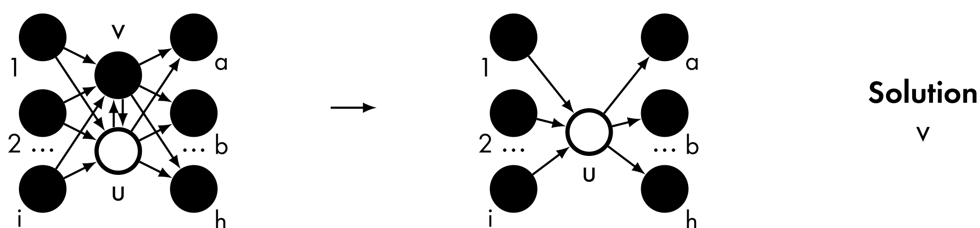
Reduction rule 6: Dominating bi-directed edge

If a vertex v dominates a vertex u that it shares a bi-directed edge with, we can immediately add v to the solution. It dominates such a vertex if both its predecessors and successors are a superset of the respective neighborhoods of u. This is exactly the case if all of these hold:

$$(u, v), (v,u) \in E$$
$$(a, u) \in E \Rightarrow (a, v) \in E$$
$$(u, b) \in E \Rightarrow (v, b) \in E$$

The predecessors and successors of u may intersect. In this case, v needs to be connected to these vertices via bi-directed edges too.

Visual notation: Dominating 2-cycle



Proof of safeness

Either u or v need to be part of the solution as $\{u, v\}$ forms a 2-cycle. Suppose v is not part of the solution. In that case, u only covers the 2-cycle $\{u, v\}$, as for any cycle that goes through u, another cycle exists that contains v instead. Thus, all of these cycles are already covered by other vertices that are part of such a minimum solution. If we include v, we will not increase solution size, but may also cover cycles that u would not cover. □

Reduction rule 6: Dominating bi-directed edge (continued)		
Overview		
New n	$n-1$	The vertex v gets removed.
New m	$\leq m-2$	The edges $(u, v), (v, u)$ are removed. If v has other neighbors, the connecting edges get removed as a result of deletion.
New k	$k-1$	We immediately include v in the solution.
Matching time	$\mathcal{O}(m^2)$	For every edge, we may test if the reverse edge resulting in a 2-cycle exists in constant time. Comparing the neighborhoods of up to m neighbors each for inclusion of one set within the other depends on the number of neighbors of either vertex.
Implementation		
Implemented in `reductions.AddMoreImportantSibling`.		

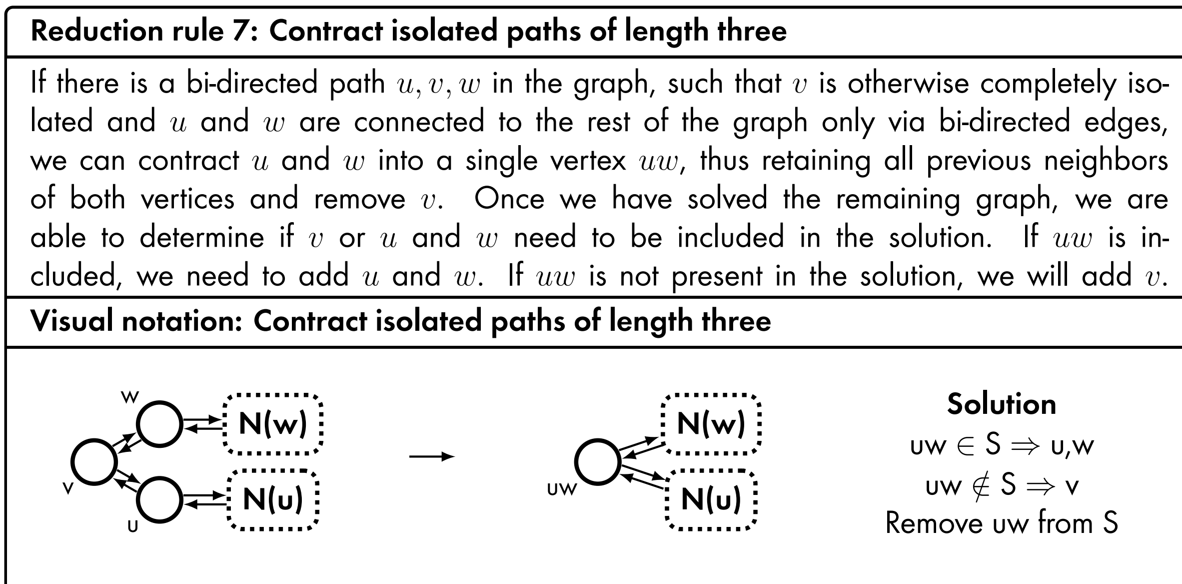
3.6.4. Contract isolated paths of length three

The following reduction rule is a fairly direct adaptation of a rule for Vertex Cover (Lemma 4.1 in Fomin, Grandoni, et al., 2009, Rule 7 of Fellows, Jaffke, et al., 2018). Since it uses the principle that it only relies on the local context, it can be seen as the blueprint for further adaptation of rules that have been defined for Vertex Cover.

Reduction rule 7: Contract isolated paths of length three

If there is a bi-directed path u, v, w in the graph, such that v is otherwise completely isolated and u and w are connected to the rest of the graph only via bi-directed edges, we can contract u and w into a single vertex uw, thus retaining all previous neighbors of both vertices and remove v. Once we have solved the remaining graph, we are able to determine if v or u and w need to be included in the solution. If uw is included, we need to add u and w. If uw is not present in the solution, we will add v.

Visual notation: Contract isolated paths of length three



Reduction rule 7: Contract isolated paths of length three (continued)

Proof of safeness

Suppose w is part of the solution. If v was part of the solution too, we would be able to obtain a solution of equivalent size by including u instead. If w was not part of the solution, v is because of the 2-cycle connecting both. If u was part of that solution, we would have been able to find an equivalent solution by replacing v with w. We can thus assume that w and u will always be part of the solution at the same time, if not v gets added, which is cheaper if neither is selected. □

Overview

New n	$n-2$	Vertex v is removed and u,w are contracted into a single vertex
New m	$\leq m-4$	Edges $\{v, w\}, \{v, u\}$ are removed. If u, w shared predecessors or successors, duplicates are removed.
New k	$k-1$	We add one vertex to the solution after having solved the reduced instance.
Matching time	$\mathcal{O}(n)$	For vertices connected with exactly two bi-directed edges, we test if there is a non-edge between the neighbors. These are constant time operations.

Implementation

In `reductions.ContractLines`.

To implement this behaviour, we need to rely on some form of recursion. Instead of manually calling the solver in this step, we implemented a log of pending changes, see Section 3.2. This is especially useful, as several other rules work similarly. An example is shown in Figure 3.2.

Actually, this rule would allow for u and w to be part of uni-directed edges. We however have to ensure that a contraction does not create new induced cycles. In, but not limited to, the following examples on a graph G, this is the case.

If $(u, w), (w, u)$ are non-edges:

1. If u and w are part of different components in $\mathrm{directed}(G)$.
2. If both ${}^{\rightarrow}N[u]_{\mathrm{directed}(G)} \subseteq {}^{\rightarrow}N[w]$ and $N^{\rightarrow}[u]_{\mathrm{directed}(G)} \subseteq N^{\rightarrow}[w]$.

If an edge (u, w) exists:

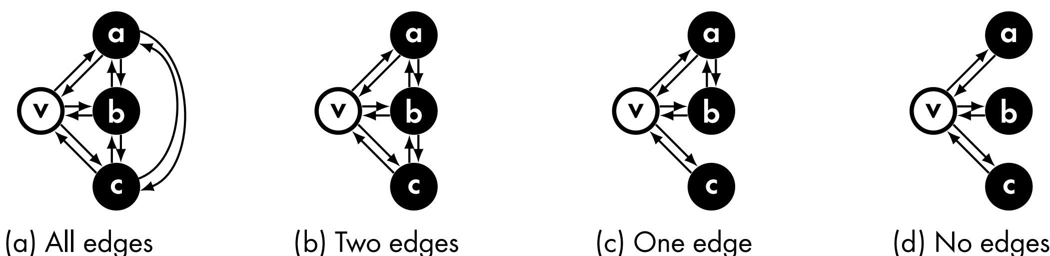
3. If excluding (u, w) from the graph no other induced $u{\leadsto}w$-path exists, we can delete (u, w) and immediately apply the rule, again closing any paths that used (u, w). Otherwise, contracting u and w into uw from the $u{\leadsto}w$-path would have created a $uw{\leadsto}uw$-cycle.

These structures have been uncommon in practice, so the implementation of the rule was kept simple to avoid errors. These could verified using search similar to Reduction rule 15.

3.6.5. Contract neighbors of degree three vertices

The rule above was able to eliminate vertices with only bi-directed edges of degree two. Expanding this to vertices of degree three naturally follows. There are now four cases of possible edges between the three adjacent vertices, depicted in Figure 3.3.

- **All edges**, 3.3a
  This case is already handled by Reduction rule 6, when consecutively applying it on a with the bi-directed edge $\{v, a\}$, b with $\{v, b\}$ and c with $\{v, c\}$, which adds all neighbors a, b, c into the solution, allowing v to be removed by Reduction rule 4.
- **Two other edges**, 3.3b
  The vertex b will be added to the solution by Reduction rule 6. v can now be processed by Reduction rule 7, contracting a and c.
- **One edge between a, b present**, 3.3c
  This case can be handled similarly to Reduction rule 7. We remove v and connect the neighborhood of c to a and b and add c into the solution if both a and b are in the solution of the remaining instance and v otherwise.
- **Vertices a, b, c are an independent set**, 3.3d
  We create three vertices ab, ac, bc. Each of these vertices will be connected to both neighborhoods of the respective vertices. Furthermore, edges $(ab, ac), (ac, ab), (ac, bc), (bc, ac)$ are added and v, a, b, c removed from the graph. We can now observe that we either need to include ac or at least two vertices into the solution. If ac is included, we include v since a, b, c did not need to cover their neighborhoods. If two were selected we include the vertex which has its only previous neighborhood completely covered by the two vertices now selected for the solution. We also include v to cover the edges to the other two remaining vertices. If all were selected, we include a, b, c as all their neighborhoods needed to be covered.



(a) All edges (b) Two edges (c) One edge (d) No edges

Figure 3.3.: Possible neighborhood layouts of vertices with degree three

This rule is adapted from a rule for Vertex Cover (Rule 8 of Fellows, Jaffke, et al., 2018). Its proof is similar to Reduction rule 7 and otherwise argues in the same way as for Vertex Cover, so we will leave it to Fellows, Jaffke, et al.

Reduction rule 8: Contract three connected neighbors

If a vertex is connected with bi-directed edges to neighbors a, b, c which contain exactly the bi-directed edge $\{a, b\}$ and are only adjacent to bi-directed edges, we can contract c into the other vertices thus decreasing n and k.

Visual notation: Contract three connected neighbors

N(a) N(b) N(c) a b c v → N(a) N(c) N(b) a b

Solution

a,b $\in$ S $\Rightarrow$ c

a $\notin$ S $\vee$ b $\notin$ S $\Rightarrow$ v

Overview

New n	$n-2$	Vertices v and c are removed.
New m	$\leq 2m-8$	Edges from the neighborhood $N(c)$ may be duplicated, four bi-directed edges are removed.
New k	$k-1$	Vertices v or c would have been additionally selected.
Matching time	$\mathcal{O}(n^2)$	We search for vertices with exactly three bi-directed edges. We then need to verify that exactly the edges $(a, b), (b, a)$ connect these three neigbours.

Implementation

In `reductions.DealWithDegreeThree`.

In fact, this is a special case of Fellows, Jaffke, et al. (Rule 8, 2018) where a vertex with a neighborhood being partitioned into exactly two cliques was removed. These structures however are fairly unlikely within our graphs and would create a large overhead while contributing fairly little to practical solving since a large number of edges is created.

This following rule is adapted from a Vertex Cover rule as well (Rule 9 of Fellows, Jaffke, et al., 2018) and we again refer to its proof for Vertex Cover.

Reduction rule 9: Contract independent degree three neighbors

If a vertex is connected with bi-directed edges to neighbors a, b, c which form an independent set and are only adjacent to bi-directed edges, we can contract them into three new vertices and remove v, thus decreasing n by one.

Reduction rule 9: Contract independent degree three neighbors (continued)

Visual notation: Contract independent neighbors of degree three vertices

Solution

I := {ab,ac,bc}
S ∩ I = {ac} ⇒ v
S ∩ I = {ab, ac} ⇒ v, a
S ∩ I = {ab, bc} ⇒ v, b
S ∩ I = {ac, bc} ⇒ v, c
I ⊆ S ⇒ a, b, c
Remove I from S

Overview

New n	$n-1$	Vertex v is removed, the other three are replaced by a counterpart.
New m	$\leq 2m-2$	Edges from the neighborhoods $N(a), N(b), N(c)$ may be duplicated, three bi-directed edges adjacent to v are replaced by two for the newly created combined vertices.
New k	k	Solutions where v would have been selected are pushed to ac, the value therefore does not change.
Matching time	$\mathcal{O}(n)$	We search for vertices with exactly three bi-directed edges. We then need to verify that the neigbours are independent. This is possible in constant time.

Implementation

In `reductions.DeleteThreeIndependent`, unused.

In practice, this rule increased running times as it creates a large number of additional edges, although it reduces the solution size by one.

3.6.6. Crowns

Crown rules have been successfully used for Vertex Cover kernelization (Abu-Khzam et al., 2007) and, with a few modifications, can be translated to DFVS.

Reduction rule 10: Crowns

A crown consists of two sets of vertices I, H

1. Vertices in I are an independent set.
2. H contains at least as many vertices, $|I| \geq |H|$.

Reduction rule 10: Crowns (continued)

3. Vertices in I are only connected to vertices in H.
4. There is a matching M on the bi-directed edges between I and H such that all vertices in H are matched.

As a result, I and H do not share any vertices, $I \cup H = \emptyset$. In that case, we can add all vertices in H directly into the solution and remove vertices in I.

Proof of safeness

By definition, there is a matching from all vertices in I to vertices in H. For all these edges, either adjacent vertex needs to be part of the solution. Since the matching size is $|H|$, adding all vertices in H is such a minimum solution while possibly covering other cycles as well. □

Overview

New n	$n - \|I\| - \|H\|$	All vertices in I, H, are removed.
New m	$\leq m - 2\|H\|$	All edges between I and H are removed, at least the required ones from the matching. Further edges connected to H are also removed.
New k	$k - \|H\|$	we add all vertices in H into our solution.
Matching time	$\mathcal{O}(\sqrt{n} \cdot m)$	In principle, crown rules are efficiently implementable by first computing a matching and then augmenting it, essentially based on flow algorithms

Implementation

In `CrownReduction`, partial

Vertices in H may be connected to any other vertices in the graph, including directed edges. In general, we disallow vertices in I to be connected to other vertices, as this would invalidate the argument we used.

However in some cases, we are still able to apply the rule. Essentially if all induced cycles vertices in I lie on at some point pass through H. Specifically, we do not allow cycles only within I.

1. We can allow any directed edges from or to H. All possible paths through these vertices would be covered by H.
2. We can furthermore allow paths containing edges from anywhere in the graph to enter I at vertices $I_{\text{in}} \subseteq I$ and leave I through $I_{\text{out}} \subseteq I$ if these do not intersect, $I_{\text{in}} \cap I_{\text{out}} = \emptyset$, and are not connected and there are no $v_i \in \text{in}, v_o \in \text{out}$ with $(v_i, v_o) \in E(G)$. Since they visit a vertex in H, they are covered for our minimum solution.
3. If every path leaving or entering I at some point passes through H or is closed by a shorter cycle. We could test this in a similar fashion as in Reduction rule 15.

We only implemented a search for easy to find small crowns. A sensible approach for finding all feasible crowns would be computing the crowns on the undirected subgraph and determining which of these are valid on the directed graph.

Crowns may also be identified with an LP-based approach (Abu-Khzam et al., 2007). If we already compute LPs for example for lower bounds, we might reuse the result of such computations.

3.6.7. Single disjoint cycle

If a vertex v lies on only one disjoint cycle, we can safely shortcut it (Rule 5 of R. Fleischer et al., 2009). There is at least one other vertex that dominates it and could be selected for the solution instead. This is a generalization of Reduction rule 5. We abandoned implementing this rule as such a case hardly ever occurred in our test data and further rules like Reduction rule 14 would in such a case likely remove edges of such constructs.

3.7. Remove edges not on induced cycles

Many of these reduction rules were derived from other rules, especially rules known to be working for Vertex Cover.

The most interesting principle however, was discovered by Lin and Jou (2000). Only induced cycles matter for solving DFVS. This is analogue for example to the Hitting Set edge domination reduction rule (Abu-Khzam, 2007; Weihe, 1998) that allows us to ignore sets that are supersets of other ones. If we form a Hitting Set of all cycles, only the sets that represent induced cycles will remain.

Thus, if an edge does not lie on an induced cycle, we can remove it immediately. As a consequence, we can remove vertices that do not lie on induced cycles as well. In order to be able to remove these edges, we need to solve the following Edge on Induced Cycle problem.

Problem 10: Edge on Induced Cycle
Input: Graph G, an edge $e \in E(G)$ **Output**: Does e lie on an induced cycle in G?

Unfortunately checking whether a vertex lies on an induced cycle is $\mathbf{NP}$-hard (Fellows, Kratochvil, et al., 1995; Lubiw, 1988). Similarly, Edge on Induced Cycle is $\mathbf{NP}$-hard as well – this even applies to graphs that become acyclic when removing a single edge (Dirks, Gerhard,

et al., 2024). We therefore cannot parameterize this by the size of a DFVS on the instance. We will formulate multiple smaller rules, that solve the inverse problem and can be implemented efficiently in practice.

3.7.1. Strongly connected components

A common approach to solving DFVS that builds upon this observation is splitting the graph into its strongly connected components and solving each component independently (Rule PIE of Lin and Jou, 2000, Swat, 2022b).

This is not exactly a reduction rule as that alone does not change the size of the instance. We however treat it as a reduction rule since edges between strongly connected components are removed in this step. It is best applied after the more simple reduction rules and before more expensive ones should be computed.

Most approaches found in the literature will apply some of the aforementioned reduction rules, then compute all strongly connected components and continue with computing on each component individually. Although algorithms for computing these components with running time linear in m exist (Tarjan, 1972), they produce a large overhead in discovering individual strongly connected components. An approach that we have found to be more efficient was relying on a recursive algorithm (L. K. Fleischer et al., 2000) and applying the more simple reduction rules ahead of each recursion step. This outweighed the additional logarithmic costs, especially since it can be computed in linear time when the graph already contains just a single component.

Reduction rule 11: Strongly connected components

We pick an arbitrary vertex v. We then compute the set of vertices $^{\rightarrow}V$ that can reach v with a path in G, and the set of vertices that v can reach, $V^{\rightarrow}$, as seen in Figure 3.4b. v is included in both of these sets. The intersection $^{\rightarrow}V \cap V^{\rightarrow}$ is a strongly connected component and can be solved independently. We can now recurse on $^{\rightarrow}V \setminus V^{\rightarrow}$, $V^{\rightarrow} \setminus^{\rightarrow} V$ and $V(G) \setminus \{^{\rightarrow}V \cup V^{\rightarrow}\}$ independently and start by applying the simple reduction rules. In case of Figure 3.4c, this would immediately remove three vertices that would otherwise be independent strongly connected components as in Figure 3.4d (assuming that shortcut rules are not applied).
In practice, it is best to find the largest component first. We therefore attempt to select a vertex of high in- and out-degree, as this is more likely to be placed in a larger component. We achieve this by picking the vertex with the largest product of in- and out-degree, as displayed in Figure 3.4a.

Reduction rule 11: Strongly connected components (continued)

Proof of safeness

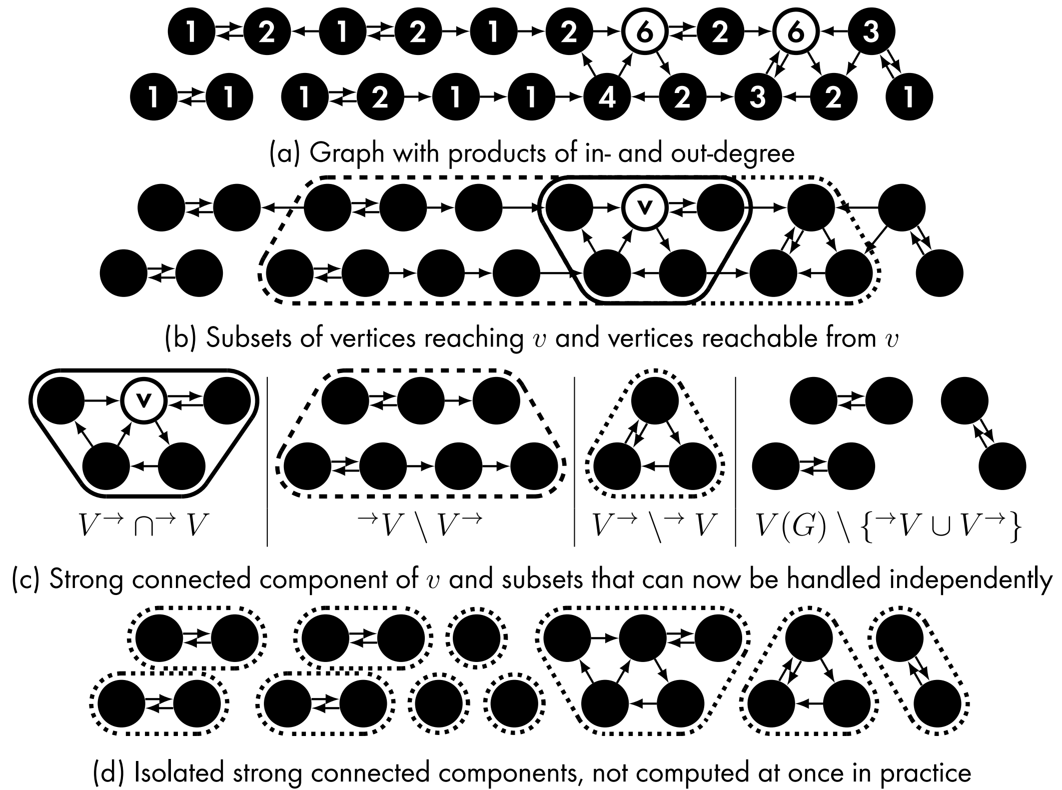
Per definition a cycle can only live within one strongly connected component. As such, there cannot exist a cycle that does not get hit by the solution within any strongly connected component solved independently. □

Overview

New n	n	No vertices are being removed.
New m	$\leq m$	No edges are added. If there were edges between different strongly connected components, these will not appear in any of the produced instances.
New k	$\leq k, \geq \frac{k}{4}$	In case we are able to separate several strongly connected components, their sizes will add up to k, thus reducing the k for individual instances. A single application creates up to four separate instances that divide that k between each other.
Matching time	$\mathcal{O}(n^2)$	We perform two BFS and compare the resulting sets afterwards. This is just for a single application of the rule.

Implementation

In `reductions.Fleischer`.



(a) Graph with products of in- and out-degree

(b) Subsets of vertices reaching v and vertices reachable from v

(c) Strong connected component of v and subsets that can now be handled independently

(d) Isolated strong connected components, not computed at once in practice

Figure 3.4.: Strong connected components

3.7.2. Remove directed dead ends

Similar to Reduction rule 4, we can remove edges that immediately lead to or come from a vertex that do not have a same directed predecessor or contain only bi-directed edges otherwise.

Reduction rule 12: Remove directed dead ends

If for an edge $e = (s, t)$ the vertex s does not have a predecessor that is not a successor at the same time, we can remove e. This works analogous for successors of t.

Visual notation: Remove without directed predecessors

a s 1
b … … 2
h i
→
a s 1
b … … 2
h i

Solution
∅

Visual notation: Remove without directed successors

1 t a
2 … … b
i h
→
1 t a
2 … … b
i h

Solution
∅

Proof of safeness

There can be no cycle going through e that would not be covered in the solution anyway, as either vertex of a 2-cycle needs to be included. □

Overview

New n	n	No vertices are being removed.
New m	$\leq m$	Any identified edges may be removed.
New k	k	Vertices are not touched.
Matching time	$\mathcal{O}(m)$	For each vertex, we need to test if all uni-directed edges of a vertex have the same direction. Each edge only gets tested twice. This would result in $\mathcal{O}(n+m)$, however the m takes precedence. A specialized graph implementation might track this and could thus only need to test n vertices.

Implementation

In `CombinedRecursiveReduction`, see Section 3.10.

3.7.3. Remove if there is no directed cycle

In the previous rule we already observed that we do not need to pay attention to 2-cycles. This remains true even if the 2-cycles occur further down the path. In effect, we can independently work on the directed subgraph (Rule PIE of Lin and Jou, 2000, Swat, 2022b).

This rule was reintroduced independently by Červený et al. (Rule 5, 2022).

Reduction rule 13: No directed cycle

If an edge does not lie on a cycle in the directed subgraph, we can remove it.

Algorithm

Input: Graph G, edge (s,t)
Output: Is (s,t) definitely not induced?

```
1  Q := {t}                         // Initialize queue with single queued vertex
2  R := {t}                         // Track vertices that have been queued
3  if (t, s) ∈ E(G) then
4  |   return false                 // s, t are a 2-cycle
5  end
6  while Q is not empty do
7  |   v := poll(Q)                 // Select and remove first element in queue
8  |   for each w ∈ N→[v] do
9  |   |   if (w, v) ∉ E(G) then    // The edge (v, w) was directed
10 |   |   |   if w = s then
11 |   |   |   |   return false     // We found a directed t⇝s-path
12 |   |   |   else if w ∉ R then   // We add vertices only once
13 |   |   |   |   Q ← Q ∪ {w}      // Add w to the queue
14 |   |   |   |   R ← R ∪ {w}      // Remember not to add w again
15 |   |   |   end
16 |   |   end
17 |   end
18 end
19 return true                      // We did not find a directed t⇝s-path
```

Proof of safeness

There is no induced $t \rightsquigarrow s$-path in the graph. The search followed all edges, except for edges that definitely lie on a 2-cycle, making all cycles that would use it non-induced. □

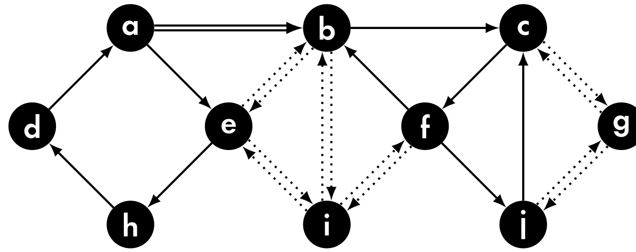


Figure 3.5.: Edge not on a directed cycle

Reduction rule 13: No directed cycle (continued)		
Overview		
New n	n	No vertices get removed directly.
New m	$m-1$	The edge is removed.
New k	k	Vertices are unchanged.
Matching time	$\mathcal{O}(m^2)$	We inspect each edge at most once and apply this rule for each edge.
Implementation		
Implicitly in `DeleteNonInducedEdge`		

An example is shown in Figure 3.5. The edge (a, b) does not lie on a cycle in the directed subgraph and could thus be removed.

After a successful application of the rule, Reduction rule 12 should be applied recursively to remove larger acyclic components at once. We can furthermore modify this search to remember the edge it used to first reach a vertex. In this case, we could directly reconstruct the cycle we have found and mark all edges on it as processed, removing the need to perform the BFS again.

3.7.4. Remove if there is always a shorter cycle

If the source vertex of our edge contains an edge to a vertex we would visit, any path back to it would close a shorter cycle. We can thus ignore these in our search. The same applies to predecessors of the target vertex.

This makes this rule a generalization of Lin and Jou (Rule DOME, 2000). It was introduced independently by Červený et al. (Rule 6, 2022).

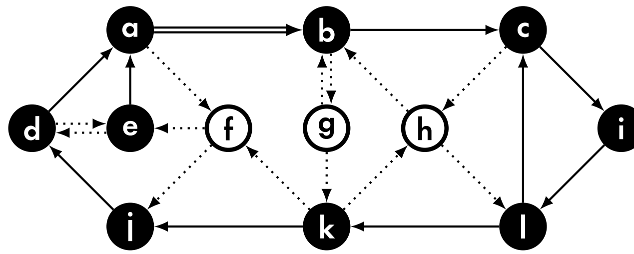


Figure 3.6.: Edge with edges allowed for search and forbidden vertices marked

Reduction rule 14: Remove ignoring shorter cycle

If for every cycle C an edge $e = (s, t)$ lies on, a vertex $w \in C$ such that a different edge (s, w) or (w, t) exists, we can remove e.
We can implement this rule by modifying our breadth-first-search from t to s. We can exclude all vertices that are either a direct predecessor of t or a direct successor of s. Note that this even applies to bi-directed edges, since this implies a 2-cycle which needs to be covered as well.

Modification for the algorithm of Reduction rule 13

```
     ...  ...  ...  ...
8    |    |    |    |  return false
9*   |    |    |    else if
     |    |    |        w ∉ R                         // We add vertices only once
     |    |    |    and
     |    |    |        (s,w), (w,t) ∉ E(G)           // Ignore if shortcut exists
     |    |    |    then
10   |    |    |    |  Q ← Q ∪ {w}
     ...  ...  ...  ...  ...
```

Proof of safeness

If we look at the Hitting Set set systems that our DFVS instance would produce, every set of a cycle going through e would be a subset of a different set via the alternative edge directly leading to w. The instance without the respective sets is therefore equivalent. □

Overview

New n	n	No vertices get removed directly.
New m	$m-1$	The edge is removed.
New k	k	Vertices are unchanged.
Matching time	$\mathcal{O}(m^2)$	We additionally test for the existence of two edges which is possible in constant time.

Implementation

Implicitly in `DeleteNonInducedEdge`

An example for allowed vertices and edges for a BFS for the edge (a, b) is shown in Figure 3.6. We may not visit f since it is a successor of a. Likewise, we can ignore g, h as predecessors of b. Bi-directed edges like $\{d, e\}$ remain irrelevant for (a, b). The BFS finds a path in the end, we are thus not allowed to remove (a, b).

3.7.5. Remove edges while tracking cycles in predecessors

The example in Figure 3.6 from the previous section illustrates a case where Reduction rule 14 does not remove (a, b) even though it did not lie on an induced cycle. The path c, i, l was already a cycle. In fact, all other edges in the example lie on an induced cycle and would thus not be removed either.

We can further modify our search from Reduction rule 14 to rule out cases where we would follow a path during search that contains an induced cycle on its own. Our approach will not find all occasions of edges not on induced cycles, but usually finds a lot of useful results.

The non-induced edge (a, b) of the example in Figure 3.6 is now discovered as depicted in Figure 3.7. We do not visit h as b is a forbidden successor of c. We stop our search in l as c is a forbidden successor of l.

Reduction rule 15: Shorter cycle in predecessors

We perform a BFS. For every vertex, we track a set of disallowed successors. We do not visit vertices if they have an edge to a vertex on our previous path. In case we find a different path to a vertex, the disallowed vertices will be the cut of both. In case this leads to vertices being allowed whereas they were not before, we need to search from these vertices again. This implicitly includes disallowing edges $(s, w), (w, t)$ for a visited vertex w as in Reduction rule 14. If we cannot find a path back to s using these and the previous rules within the BFS, we can safely delete our edge.

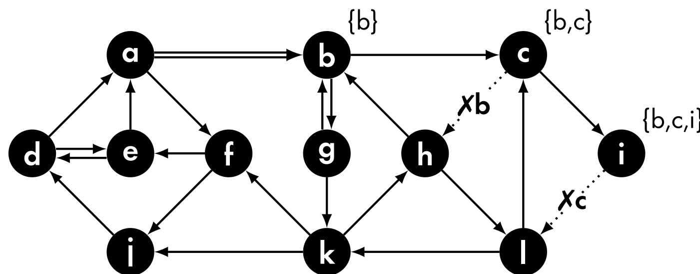


Figure 3.7.: Sets of disallowed successors on the example from Figure 3.6

Reduction rule 15: Shorter cycle in predecessors (continued)

Algorithm

Input: Graph G, edge (s,t)

Output: Is (s,t) definitely not induced?

```
1  Q := {t}                      // Initialize queue with single queued vertex
2  R := {t}                      // Track vertices that have been queued
3  F_t := {t}                    // Track forbidden successors, disallow t for t
4  while Q is not empty do
5      v := poll(Q)              // Select and remove first element in queue
6      for each w ∈ N→[v] do
7          w_allowed ← true      // w is allowed in the current search
8          for each f ∈ F_v do   // Successors forbidden for t⇝v-paths
9              if (w, f) ∈ E(G) then        // w had a forbidden successor
10                 if w ≠ s or f ≠ t then   // Ignore if edge is (s,t)
11                     w_allowed ← false    // Disallow visiting w from v
12                 end
13             end
14         end
15         if w_allowed then     // Edge (v,w) did not close a shorter cycle
16             if w = s then
17                 return false             // We found a directed t⇝s-path
18             else if w ∉ R then           // First encounter of w
19                 Q ← Q ∪ {w}              // Add w to the queue
20                 F_w := F_v ∪ {w}         // All disallowed of v and w
21                 R ← R ∪ {w}              // We now have encountered w
22             else                         // Subsequent encounter of w
23                 N := F_w ∩ (F_v ∪ {w})   // New disallowed successors
24                 if N ⊊ F_w then          // Less disallowed
25                     F_w ← N              // Update disallowed successors
26                     if w ∉ Q then
27                         Q ← Q ∪ {w}      // Add w back into the queue
28                     end
29                 end
30             end
31         end
32     end
33 end
34 return true                   // We did not find a directed t⇝s-path
```

Reduction rule 15: Shorter cycle in predecessors (continued)		
Proof of safeness		
If we look at the HITTING SET set systems that our DFVS instance would produce, every set of a cycle going through e would be a subset of a different set because of a cycle closed within the searched path. The instance without the respective sets is therefore equivalent. □		
Overview		
New n	n	No vertices get removed directly.
New m	$m-1$	The edge is removed.
New k	k	Vertices are unchanged.
Matching time	$\mathcal{O}(m \cdot n^2)$	A removal of the set elements may trigger another update of successors of this vertex. Every vertex is initially assigned a set containing at most all n vertices of the graph. Each of its up to n removals of vertices might trigger another search. The search itself is possible in $\mathcal{O}(n^2)$ and occurs at most n times. We apply this rule for all m edges.
Implementation		
In `DeleteNonInducedEdge`		

This does not cover specific cases where a combination of several cycles would in theory block a path but no individual one does it alone. In such cases, we could possibly look into tracking relying on logical expressions instead, but their evaluation would be **NP**-hard since we could otherwise solve the initial problem.

We were able to show that this algorithm becomes exact on the class of all graphs that exclude the graph in Figure 3.8 as a directed minor (Dirks and Gerhard, 2024). A directed minor is very similar to its undirected counterpart, however takes directions of edges into account, while preserving paths. Although this is fairly restrictive, it allows arbitrarily long cycles and DFVS solution sizes.

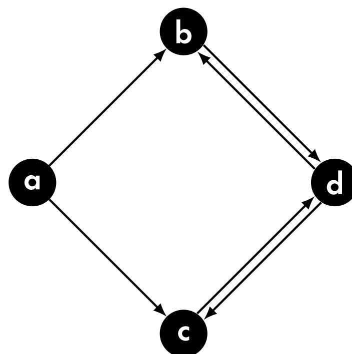


Figure 3.8.: Directed minor to exclude for our EDGE ON INDUCED CYCLE heuristic

3.8. Pick cycle dominating vertices

Since Reduction rule 6 was very effective, we have searched for rules that follow the same principle. The two main ways to relax its very strict requirements are extending the scope of the neighborhood that is taken into account and to generalize it to cycle lengths greater than two.

3.8.1. Pick vertices weakly dominating bi-directed edge

We were able to construct a new rule related to Reduction rule 6, although it does not subsume it. Instead of requiring v to be adjacent to a superset of neighbors of w, we only consider either the incoming or outgoing neighbors.

Reduction rule 16: Pick cycle dominating vertices on bi-directed edges

Let us look at a vertex v that it lies on a 2-cycle with a vertex w. If w does not lie on a cycle such that we do not visit vertices that v has a 2-cycle with, we can add v to the solution.

Proof of safeness

Suppose that w was part of the solution and v was not. In that case, all neighbors of v would need to be part of the solution as well, to cover the 2-cycles that v lies on. In this case, we can find a solution of equivalent size by removing w and including v. □

Overview

New n	$n-1$	The dominating vertex v is added to the solution.
New m	$\leq m-2$	The bi-directed edge $\{v, w\}$ is removed. Any further adjacent edges of v are removed.
New k	$k-1$	The dominating vertex v is added to the solution.
Matching time	$\mathcal{O}(m^2)$	We perform a search from w to w.

Implementation

In `DeleteObsoleteOnTwoCycle`

The application of this rule usually allowed us to shortcut w with Reduction rule 5, since our graphs were most likely to contain these structures if v has a bi-directed edge to a single successor of w.

In principle, we are able to adapt our search approaches from Reduction rules 14 and 15 for this rule. We can use all vertices that v has a bi-directed edge with as forbidden vertices and will thereby only respect induced cycles. Depending on the desired trade-off between quality and performance, we can chose to track forbidden successors.

We could also further restrict our rule to either cover all successors or predecessors of w with bi-directed edges of v to improve its performance while omitting possible vertices to directly include in the solution. Based upon our evaluation, the decrease in solution size however outweighs this additional computation time.

3.8.2. Strongly dominating cycle

This is a generalization of Reduction rule 6 to allow longer cycles. It can be implemented very easily, since there is always exactly one suitable successor if the rule can be applied.

Reduction rule 17: Strongly dominating cycle

If a vertex v dominates a cycle C, such that for all vertices $w \in C, w \neq v$ the predecessors and successors are a superset of their counterparts of w except for vertices in C itself, we can immediately add v to the solution.

Algorithm

Input: Graph G, an edge (v, w)
Output: Does v dominate a cycle through (v, w)?

1. $v_{\text{previous}} := v$ // Start with v as the previous vertex
2. $v_{\text{current}} := w$ // Use w as the next vertex
3. **while** $v_{\text{current}} \neq w$ **do**
4. **if** ${}^{\rightarrow}N[v_{\text{current}}] \setminus \{v_{\text{previous}}\} \not\subseteq {}^{\rightarrow}N[v]$ **then**
5. **return** *false* // v does not have a counterpart to a predecessor
6. **end**
7. $F := N^{\rightarrow}[v_{\text{current}}] \setminus N^{\rightarrow}[v]$ // Find successor on cycle
8. **if** $|F| \neq 1$ **then**
9. **return** *false* // v does not dominate v_{current} on cycle
10. **end**
11. $v_{\text{previous}} \leftarrow v_{\text{current}}$ // Move on to the next vertex
12. $\{v_{\text{current}}\} \leftarrow F$ // Unwrap single vertex in F to inspect next
13. **end**
14. **return** *true* // We can immediately add v

Proof of safeness

Suppose v was not part of the minimum solution. Any vertex of C needs to be part of a DFVS. Thus another vertex $w \in C$ is included in the solution. If w was covering a cycle other than C, this has so far not been covered, even though there are the same edges leading to v. However, now we could obtain a solution of equivalent size by including v instead of w, contradicting our claim. □

Reduction rule 17: Strongly dominating cycle (continued)		
Overview		
New n	$n-1$	The vertex v gets removed.
New m	$\leq m-2$	The edges adjacent to v are removed. At least two edges were part of C
New k	$k-1$	We immediately include v in the solution.
Matching time	$\mathcal{O}(m \cdot n^2)$	We start a search for each of the m edges. We compare the predecessors and successors for each vertex during search and there is always a single possible successor.
Implementation		
Implemented but removed		

We implemented this rule, however decided against using it, since it was only able to match very few vertices.

3.8.3. Weakly dominating cycle

We are able to generalize Reduction rule 17 further. We only need to ensure that we dominate with respect to all cycles at some point. This increases the implementation difficulty as there now might be multiple viable successors during search.

Reduction rule 18: Weakly dominating cycle
If a vertex v dominates a cycle C, such that there is an edge (u', w') on it, possibly with $u = v$ or $w = v$ such that for all vertices u on C until and including u' the predecessors of u are a subset of their counterparts of v except for their predecessor in C itself and similarly all successors starting with w' otherwise only contain other successors of v apart from the next vertex in C, we can immediately add v to the solution.
Proof of safeness
Suppose v was not part of the minimum solution. Any vertex of C needs to be part of a DFVS. Thus another vertex $v' \in C$ is included in the solution. Because of our construction, C does not contain induced cycles that v is not a member of. If v' was covering a cycle other than C, this has so far not been covered, even though for all of its incoming paths and outgoing paths through C a direct edge to or from v exists which would contradict the claim of a solution. However, now we could obtain a solution of equivalent size by including v instead of v', contradicting the claim of v not being part of the minimum solution. $\square$

Reduction rule 18: Weakly dominating cycle (continued)		
Overview		
New n	$n-1$	The vertex v gets removed.
New m	$\leq m-2$	The edges adjacent to v are removed. At least two edges were part of C
New k	$k-1$	We immediately include v in the solution.
Matching time	$\mathcal{O}(m \cdot n^2)$	We can build two trees using a search, containing possible successors until u' and possible predecessors before w', and test for all possible edges between the two if they do not intersect already. The successive test is hidden in the $\mathcal{O}$-notation.
Implementation		
Not implemented		

3.9. Other interesting data reduction rules

The following rules were not implemented or we abandoned their implementation quickly when we did not find instances where the rules could be applied and would produce a notable effect. Some have appeared in the literature while we found several other reduction rules.

3.9.1. Too many internally vertex disjoint paths

This rule has been included as Rule 3 in Bergougnoux et al., 2021. It is a special case of the sunflower rule for Hitting Set (Erdős and Rado, 1960; Van Bevern, 2013). It is a generalization of Rule 6 of (R. Fleischer et al., 2009) to non-edges instead of requiring the start and end vertex of the paths to be identical.

Reduction rule 19: Too many internally vertex disjoint paths
If there are at least $k+1$ internally vertex-disjoint paths from vertex u to v in G, we may introduce a new edge (u, v) for G'. If $u = v$, since this would create a loop, we can immediately add v to the solution.

Reduction rule 19: Too many internally vertex disjoint paths (continued)

Proof of safeness

A solution for G' is obviously a solution of the same size for G since G is a subgraph.
In the other direction, suppose a solution of size k exists for G. Since there are at least $k+1$ internally vertex-disjoint u, v paths, at least one of these u, v paths is not covered by a vertex in the solution.
For every path $v \rightsquigarrow u$, a vertex within this path, possibly v or u must hence be included in the solution. This would also cover all cycles shortcut by the newly introduced edge (u, v). If no path $v \rightsquigarrow u$ exists, the edge (u, v) does not contribute to a cycle, thus not increasing the solution size. □

Overview

New n	n	No vertices get removed.
New m	$m+1$	An edge is introduced.
New k	k	Vertices are unchanged.
Matching time	$\mathcal{O}\left(n^2 \cdot m \cdot \sqrt{n}\right)$	There are at most n^2 pairs of vertices to check. Each application is possible with a standard (s, t)-cut possible in $\mathcal{O}\left(\sqrt{n}m\right)$, we would be able to stop after confirming a flow greater than k.

Implementation

Not implemented

This rule is visualized in Figure 3.9. It will likely allow further applications of rules such as Reduction rule 14, although it increases m by one.

Practical implementations should only inspect non-edges since adding an edge does not make sense if it is already present. Furthermore, only vertices with out-degree $k+1$ and in-degree $k+1$ are suitable as u and v, respectively. Since even this was barely the case on the PACE challenge instances, this rule was not implemented.

This rule could be further extended. If we remove the set of disjoint $v \rightsquigarrow u$-paths P, any lower bound l for DFVS on the remaining graph $G[V(G) \setminus \bigcup_{p \in P} V(p)]$ can be subtracted from $k+1$, allowing us to introduce an edge if there are more than $k+1-l$ disjoint cycles. Computing lower bounds for DFVS will be discussed in detail in Section 5.6.2 on page 106.

Unfortunately, we are not aware of techniques that would allow us to obtain lower bounds on this remaining graph with theoretical guarantees. A good candidate would have been a Cycle Packing, however this problem itself is NP-hard (Nutov and Yuster, 2004), preventing theoretical results from this approach.

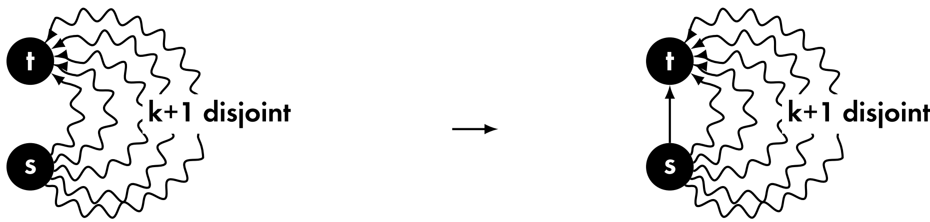


Figure 3.9.: Introduce edge if more disjoint paths than vertices in solution exist

3.9.2. Dominated cliques

This rule can be understood as an attempt to generalize crowns, Reduction rule 10, and could also be adapted for Vertex Cover.

Let us look at two independent vertices v, u that have neighborhoods $N(v), N(u)$ that each contain exactly two cliques. A vertex cannot be included in both cliques since it would otherwise have been included by Reduction rule 6. Further bi-directed or uni-directed edges between the cliques are allowed. Now, if $N(u) \subseteq N(v)$, we can add $N(u)$ into the solution and remove u.

Proof Suppose u was part of the solution. In that case, two vertices of $N(u)$, one of each clique, would not be selected. As a result, v would need be part of the solution too. We could, however, create a solution of the same size by not including u and v and selecting all of $N(v)$ instead. □

We cannot immediately include $N(v)$ though, as a minimum solution may include v but not u.

3.9.3. Tail-Biting Worms

The previous rule allows for creating several rules following a similar argument. Let us look at an independent set of vertices $H = v_1, v_2, \dots, v_l$ that have neighborhoods $N(v_1), \dots, N(v_i)$ that each contain exactly two cliques. Further edges between the cliques are allowed. Now, if $\forall i \in [1, l-1] : N(v_i) \subseteq N(v_{i+1})$ and $N(v_l) \subseteq N(v_1)$, we can add $\bigcup_{i \in [1,l]} N(v_i)$ into the solution and remove H.

Proof Suppose any v_i was part of the solution. In that case, a vertex of both cliques would not be selected, thus both neighbors would need to be part of the solution too, chaining to all other vertices in H. We could, however, create a solution of the same size by not including H and selecting all the neighborhoods instead. □

3.9.4. Tunnels

A tunnel is a segment between two vertex cuts C_1, C_2 with $|C_1| \leq |C_2|$ that has no cuts smaller than $|C_2|$ in between nor other edges entering or leaving it and does not contain cycles. Vertices in the incoming cut may only have predecessors outside of the tunnel, the ones of the outgoing cut only successors, possibly between the cuts. The cuts may also share vertices. We can remove all internal vertices of a tunnel and add an edge between each pair of vertices if they were connected by a path in the original segment between the cuts. This reduction is safe.

Proof Suppose any internal vertex v and possible further internal vertices $S' \supseteq \{v\}$ of the tunnel were part of the solution $S \supseteq S'$.

If $|S'| \geq |C_2|$, we could obtain a solution of equivalent size by including all of C_2 and possibly even a smaller solution when including C_1.

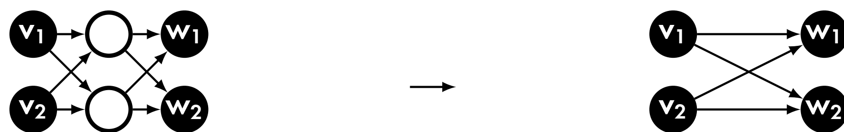
If there were less than $|C_2|$ internal vertices S' in the solution, then for each $c_1 \rightsquigarrow c_2$ with $c_1 \in C_1, c_2 \in C_2$ that contains v, there is at least one $p = c_1 \rightsquigarrow c_2$ that does not contain any vertex in S', otherwise we would have obtained a smaller cut. Each cycle closed by a $p' = c_2 \rightsquigarrow c_1$ however needs to be covered and since no vertices in p are in S, it must be covered by a vertex in p'. Now, each cycle covered by v is also covered by another vertex v' in some other p', leading to a smaller solution size when excluding v. □

The smallest tunnel not removed by other reduction rules is an independent pair of vertices v_1, v_2 pairwise connected to another pair of vertices and subsequently to vertices w_1, w_2. We can remove the intermediate vertices and introduce shortcuts as depicted in Figure 3.10a. The two framing cuts do not need to have the same size, as long as there is no cut between them that is smaller than the larger of the two, 3.10b.

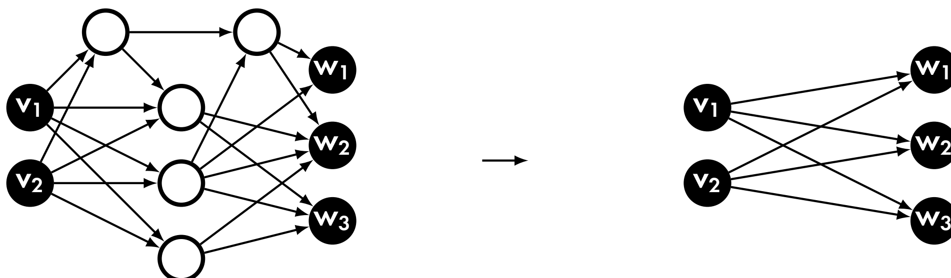
3.10. Recursive application

For an implementation that is efficient in practice, the most simple and easy to compute rules can be merged into a single combined rule. The best suitable candidates are Reduction rules 1, 4, 5 and 12. These rules are not context dependent and only depend on a vertex and its most immediate neighbors. They are applied recursively, an application on a vertex causes its neighbors to be marked for re-inspection.

We designed the algorithm to only work on a predefined set of vertices, supplied in a queue Q. Initially, we call it populated with all vertices $V(G)$ to process the whole graph. After an application of one of the other reduction rules, we may call it only on the subset of changed vertices and its neighbors, eliminating the inspection of vertices where no change could have possibly occurred. The contained rules are simple and therefore do not require a log, we simply collect a set of vertices S to add to a solution of the reduced instance.



(a) Minimum tunnel example



(b) Wider and unequal cuts

Figure 3.10.: Tunnel examples

This approach extends the recursive application of Kahn's algorithm (1962).

This has been implemented in `reductions.CombinedRecursiveReduction`.

4. Kernelisation

In this chapter, we will construct a $|F|^4$-kernel for DFVS, which depends on the size of an underlying minimum Feedback Vertex Set F. It mostly follows the construction of Bergougnoux et al. (2021). The main difference is, that it does not require the Feedback Vertex Set as an input, although it could be approximated.

4.1. Existing kernels for DFVS

As mentioned in Section 2.2 on page 21, finding a polynomial size kernel for DFVS based on the solution size k is an open problem.

There are two other main approaches within kernelization. The first is choosing a parameter that is generally smaller than the solution size to further "refine" a kernel compared to one parameterized by its solution size (Jansen and Bodlaender, 2013). We do not even know a polynomial size kernel for k, but we can do the opposite by choosing a parameter that is generally larger than the solution size and thereby restricting the input.

Kernels have been found if the graph is a tournament (Abu-Khzam, 2007; Dom et al., 2010), that is a graph where all pairs of vertices are connected by an edge, but not in the other direction. This has been transferred to generalizations of tournaments (Bang-Jensen et al., 2016).

Lokshtanov, Ramanujan, Saurabh, et al. (2019) were able to find a kernel of size $(k \cdot l)^{\mathcal{O}(\eta^2)}$ using the solution size k and the size l of a Treewidth-η Modulator of the graph. The Treewidth-η Modulator is sometimes referred to as "Wannabe Treewidth", as it implies a fixed Treewidth of η after the removal of l vertices. Furthermore, on graphs with bounded expansion and nowhere dense graphs, a kernel of size of size $\mathcal{O}(k)$ exists if they do not have cycles longer than a fixed d (Dirks, Gerhard, et al., 2024).

There have been further kernels requiring the remaining graph $G[V(G) \setminus S]$ after having deleted the DFVS S to contain specific structures (Mnich and van Leeuwen, 2017).

4.2. A kernel requiring a Feedback Vertex Set as input

Bergougnoux et al. (2021) introduced a kernel for DFVS if a solution F to the FEEDBACK VERTEX SET of the cycle preserving undirected graph $\overline{\overline{G}}$ of G is provided. It is inspired by the kernel of Bodlaender and Dijk (2010) for FEEDBACK VERTEX SET.

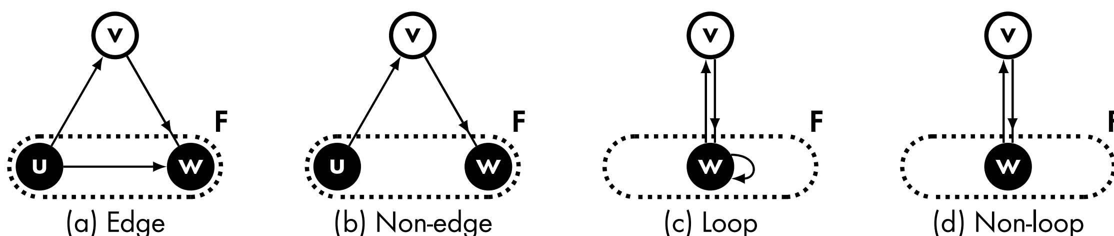
We will provide a slight modification to this kernel in the subsequent section, but restate its relevant points here. We will use slightly different notation and especially names for consistency with the rest of the thesis. Contrary to Bergougnoux et al., we continue to keep track of k as the DFVS solution size while they employed $|F|$ as its upper bound instead. The kernel depends on $|F|$ and k.

Bergougnoux et al. first compute an FVS F of $\overline{\overline{G}}$ using a 2-approximation, referring to Bafna et al. (1999) for an algorithm that computes this in $\mathcal{O}\left(\min\left(m \cdot \log\left(n\right), n^2\right)\right)$. While also handling weighted instances, it first greedily picks vertices on an undirected cycle for the solution and removes them afterwards in reverse order if they have become redundant.

4.2.1. Preparing the graph and creating a first bound

Bergougnoux et al. first apply Reduction rules 4 and 5, after which every vertex has at least two predecessors and two successors. They then apply Reduction rule 19, but only taking pairs of vertices in F into account. This rule merely adds edges without increasing k. Our size guarantee will only depend on the number of vertices, this addition therefore does not interfere with our parameter.

They introduce two additional notions. A *potential edge* is a pair (u, v) of vertices from F regardless of an edge $(u, v) \in E(G)$ being present. We allow $u = v$. Since there are $|F|(|F|-1) = |F|^2 - |F|$ ordered pairs of vertices within F because of loops not being reversed, there are exactly as many potential edges and at most as many non-edges. A vertex v *contributes* to a potential edge (u, w) if $(u, v), (v, w) \in E(G)$. Examples of a vertex v contributing to edges, non edges, loops and non-loops in F is shown in Figure 4.1.



(a) Edge (b) Non-edge (c) Loop (d) Non-loop

Figure 4.1.: A vertex contributing to different potential edges

After the application of Reduction rule 19, for each non-loop (v, v) and non-edge $(u, v) \notin E(G[F])$, there are at most k vertex-disjoint $u \rightsquigarrow v$ paths in G for any vertex or non-edge. Therefore, for each vertex pair u, v in F, there are at most k vertices contributing to non-edges or non-loops (u, v) which may also include vertices in $V(G) \setminus F$. This implies that there are at most $k\,|F|^2 - k\,|F|$ vertices contributing to non-edges in $G[F]$ and at most $k\,|F|$ vertices contributing to loops in $G[F]$ in the original graph G.

Taking $A = V(G) \setminus F$, Bergougnoux et al. continued by bounding the number of vertices in $G[A]$, with regard to k. They bound the number of vertices in sets $A_0, A_1, A_2, A_{\geq 3}$ of vertices with total degree 0, 1, 2 and at least 3, respectively, in $G[A]$.

4.2.2. Bounding vertices of degree zero

Bergougnoux et al. first apply the following Reduction rule 20 on vertices in A_0 (2021). Bounding the number of vertices in A_0 becomes simple after that. We give it in its generalized form in which it was also introduced by Červený et al. (Rule 7, 2022).

Reduction rule 20: Floating vertex

If the shortcutting of a vertex v would not introduce any new edges, we can safely remove v. This is the case if all predecessors of v contain an edge to all of its successors, $\forall w_{\text{in}} \in {}^{\rightarrow}N[v] : \exists w_{\text{out}} \in N^{\rightarrow}[v] : (w_{\text{in}}, w_{\text{out}}) \in E(G)$.

Visual notation: Floating vertex

v
1 a
2 … … b
i h
→
1 a
2 … … b
i h

Solution
∅

Proof of safeness

The vertex v can not lie on an induced cycle. Suppose v was part of a minimum solution. Each cycle that v lies on leads through both one of its predecessors and successors. Let us denote them as u_{in} and u_{out}. Because of the rules condition, an edge $(u_{\text{in}}, u_{\text{out}})$ must be present too. Therefore, a path $u_{\text{out}} \rightsquigarrow u_{\text{in}}$ must exist, which gets closed by $(u_{\text{in}}, u_{\text{out}})$ and therefore would need to be covered by the solution, contradicting the initial claim. □

Reduction rule 20: Floating vertex (continued)		
Overview		
New n	$n-1$	The vertex v gets removed.
New m	$\leq m$	Any edges adjacent to v are removed.
New k	k	Since shorter cycles always exist, v cannot be part of any minimal solution.
Matching time	$\mathcal{O}(n^3)$	For every pair of predecessors and successors of v, we test if an edge connecting them exists.
Implementation		
Indirectly in `DeleteNonInducedEdge`		

This rule is superseded by Reduction rule 14 in combination with Reduction rule 4.

Proof For any edge (u_{in}, v) leading to v, any successor u_{out} of v will be disallowed during search of Reduction rule 14, since an edge $(u_{\text{in}}, u_{\text{out}})$ exists. This leads to the deletion of all incoming edges of v. Reduction rule 4 will remove v since it is not adjacent to incoming edges after that. □

After having applied Reduction rule 20 on vertices in A_0, for a vertex $a_0 \in A_0$, all of its predecessors and successors must have been in F. Because of Reduction rule 4, at least one of either must have existed, otherwise it would have been reduced. If a_0 was not removed by Reduction rule 20, at least one non-edge in $G[F]$ must have existed. As a result, a_0 has contributed to at least one non-edge or non-loop of $G[F]$ and is one of the $k|F|^2 - k|F|$ vertices already accounted for above.

4.2.3. Bounding vertices of degree one

Bergougnoux et al. continue with vertices in A_1. As before, there can only be a limited number of vertices A_1 contributing to a loop in $G[F]$. The other vertices of degree one will be ensured to contribute to a non-edge in $G[F]$ and thus bounding the size of A_1 by applying another reduction rule. Let $A'_1 \subseteq A_1$ be the vertices not contributing to a non-edge or loop in $G[F]$.

Because of Reduction rule 5, every vertex has at least two predecessors and successors. Since vertices $a_1 \in A'_1$ do not contribute to a loop, at least one of its predecessors and two successors or two of its predecessors and one successor are in F. Consequently, there is a pair of distinct successors and predecessors in F and therefore a_1 contributes to a potential edge in $G[F]$.

Bergougnoux et al. apply the following Reduction rule 21 on vertices in A_1 to rule out the existence of vertices A'_1 not contributing to non-edges (2021). We again give it in its generalized form and show that it is subsumed by our rule.

Reduction rule 21: Covered predecessors and successors

Given a vertex v, if all predecessors but one predecessor p contain edges to all successors, remove edges from all but this predecessor p. If all predecessors contain edges to all but one successor s, remove edges to all but this successor.

Visual notation: Covered predecessors

Visual notation: Covered successors

Proof of safeness

Suppose v was part of a minimum solution. We could then create a solution of the same size using p or s, respectively. All cycles not leading through p or s are closed immediately and will thus be covered by some other vertex in the solution. Cycles initially leading through v are kept intact and thus do not allow for a smaller solution. □

Overview

New n	n	No vertex gets removed.
New m	$\leq m$	Adjacent covered edges are removed.
New k	k	Vertices are unchanged.
Matching time	$\mathcal{O}(n^3)$	For every pair of predecessors and successors of v, we test if an edge connecting them exists.

Implementation

Indirectly in `DeleteNonInducedEdge`

Reduction rule 21 is subsumed by Reduction rule 14.

Proof In case predecessors of v with one exception p are covered, Reduction rule 14 will remove all edges (u_{in}, v) with $u_{\text{in}} \in ({}^{\to}N[v] \setminus \{p\})$, since for all $u_{\text{out}} \in N^{\to}[v]$, an edge $(u_{\text{in}}, u_{\text{out}})$ exists, preventing the search from completing. □

After the application of this rule, Reduction rule 5 will shortcut v. All vertices in A_1 are therefore part of the $k|F|^2 - k|F|$ vertices already accounted for above.

4.2.4. Bounding vertices of degree greater than three

Since $\overline{G}[A]$ is a forest, there are less vertices of degree greater than or equal three than leaves minus one. In Section 4.2.3, we were able to bound the number of leaves in $G[A]$ to $k|F|^2 - k|F|$. As a result, there are at most $k|F|^2 - k|F| - 2$ vertices in $A_{\geq 3}$.

In the example in Figure 4.2, there are three vertices e, f, i with a degree greater than or equal three in $G[A]$. We omitted edges adjacent to vertices in F.

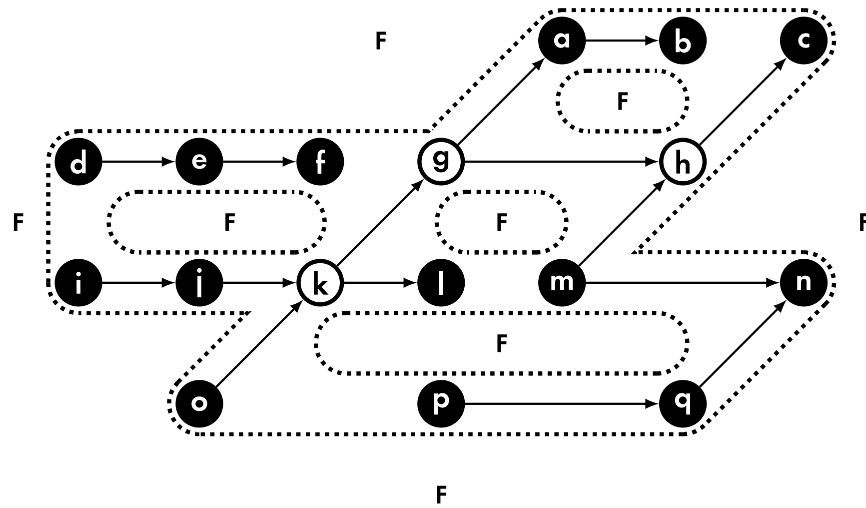


Figure 4.2.: Undirected forest outside of the underlying Feedback Vertex Set

4.2.5. Bounding the number of paths

Because of Reduction rules 4 and 5, each vertex in A_2 has at least two neighbors in F. Bergougnoux et al. continue to distinguish three types of vertices in A_2:

1. Vertices with two incoming edges in $G[A]$, called *source vertices* $A_{2,\mathrm{in}}$.
2. Vertices with two outgoing edges in $G[A]$, *sink vertices* $A_{2,\mathrm{out}}$.
3. Vertices with one predecessor and one successor in $G[A]$, *balanced vertices* $A_{2,\mathrm{both}}$.

In the example in Figure 4.2, m is a source vertex and n is a sink vertex. The vertices a, e, j, q are balanced vertices.

Bergougnoux et al. now continue by obtaining a bound of the number and length of inclusion-wise maximal directed paths in $G[A]$ with all internal vertices in A_2. The endpoints of such paths $s \rightsquigarrow t$ are either in A_2 if all predecessors of s or all successors of t have been in F or can lie in A_1 or $A_{\geq 3}$. These are depicted in Figure 4.3 on the graph from Figure 4.2 with vertices in A_1 or $A_{\geq 3}$ highlighted.

If at least one endpoint lies in A_1 or $A_{\geq 3}$, the inclusion-wise maximum path is called *outer path*, otherwise it is an *inner path*. We take the underlying undirected graph $\overline{G}[A]$ and exhaustively shortcut degree 2 vertices. After this, no degree 2 vertices remain. Every edge in the resulting forest can represent at most two outer paths while possibly representing a long chain of inner paths.

This is shown in Figure 4.4. Every edge in the graph represents exactly one outer path with the exception of (p, h), which represents two outer paths and one inner path.

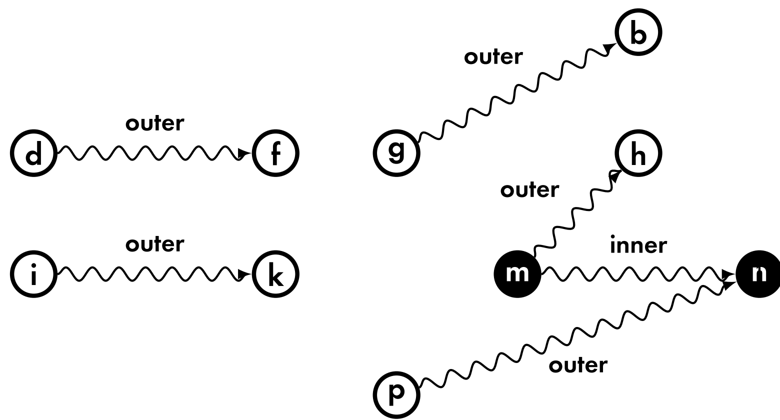


Figure 4.3.: Maximal inner and outer paths with internal vertices of degree two

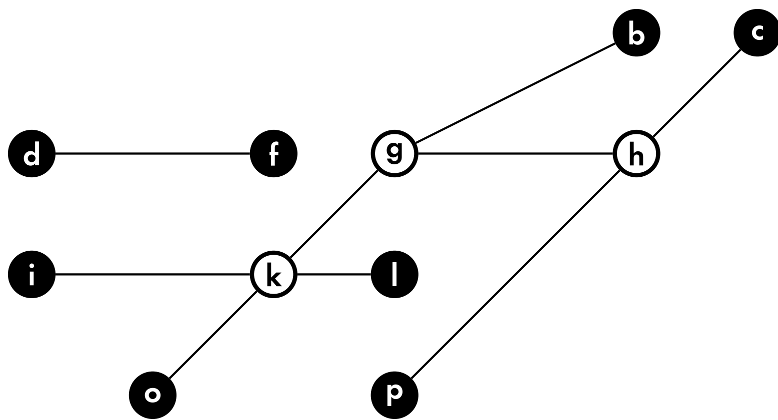


Figure 4.4.: The structure of the forest after contracting degree two vertices exhaustively

The number of outer paths is thus bounded to be at most the number of vertices of the forest minus one. This would be the vertices in A_1 and $A_{\geq 3}$, in total $2((k|F|^2 - k|F| + k|F|^2 - k|F| - 2) - 1) = 4k|F|^2 - 4k|F| - 6$.

Bergougnoux et al. introduce a new rule (2021) to limit the number of inner paths. This rule cannot be applied effectively in general since there are too many possible paths. Instead, they apply this rule only to paths with inclusion wise maximal inner vertices in A_2.

Reduction rule 22: Covered paths

We are allowed to remove vertices of a path $P = v_1 \rightsquigarrow v_r \in G$ that is not a cycle if there are no i, j with $1 \leq i \leq j \leq r$ and $u, w \in V(G) \setminus V(P)$ such that there are edges $(u, v_i), (v_j, w) \in E(G)$ coming from the endpoints of a non-edge (u, w) or a loop with $u = w$.

Proof of safeness

Suppose that any vertex v_l of P was part of the minimum solution. All cycles leading through it must have entered the path somewhere before or at the vertex and leave it at the vertex or further along the path.
For each edge (u, v_i) entering the path by leading to a vertex v_i with $1 \leq i \leq l$, there are direct edges to all vertices w that are at the end of edges (v_j, w) leaving the path at vertex v_j with $l \leq j \leq r$. Since there is a cycle, there is now at least one path $w \rightsquigarrow u$. However, since the edge (u, w) is closing that path to form a cycle as well, any path $w \rightsquigarrow u$ must be covered by at least one vertex in the solution. The same holds for loops of which the vertices need to be part of the solution anyways.
If we removed v_l from the solution, all of these paths would still be covered and thus leading to a solution of smaller size, contradicting our assumption. □

Overview

New n	$n - \|P\|$	The vertices of the path are removed.
New m	$\leq m - \|P\| + 1$	Edges adjacent to P are removed as well.
New k	k	The vertices are not part of a minimum solution.
Matching time	$\mathcal{O}(n^n)$	There are exponentially many possible paths that we would need to check, making this general version of the rule infeasible in practice. If the FEEDBACK VERTEX SET of the underlying undirected graph is known, we are able to apply this rule efficiently.

Implementation

Indirectly in `DeleteNonInducedEdge`

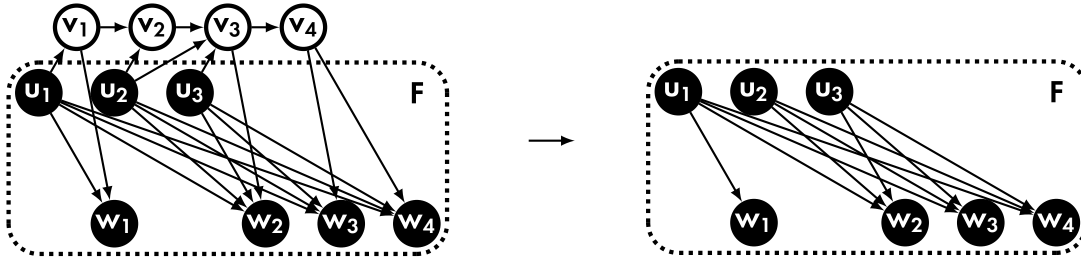


Figure 4.5.: Example of an application of Reduction rule 22

We give an example in Figure 4.5. For each predecessor of a vertex of the path $v_1 \rightsquigarrow v_4$, an edge to all successors of vertices of the vertex itself and further down the path is present. If a vertex would have a successor of an earlier vertex of the path as a predecessor, this vertex would have needed to be part of a loop, which would already have been removed by Reduction rule 1.

This rule is subsumed by Reduction rule 14 in combination with Reduction rule 4 without requiring to compute a Feedback Vertex Set beforehand.

Proof Each incoming edge of vertices of $v_1 \rightsquigarrow v_r$ will be removed by Reduction rule 14, since an edge of its source to any successor of vertices further down the path exists. After this step, the remaining edges would either be removed by the same rule or, with the first vertices having no remaining predecessors, would be removed by Reduction rule 4 alongside the vertices of the path itself. □

After applying Reduction rule 22, every inner path contributes to at least one non-edge or loop (u, w) of $G[F]$, paths contributing to a non-edge or loop (u, w) being defined in the same way as vertices but with a path $v_1 \rightsquigarrow v_r = (v_1, \ldots, v_r)$ and edges $(u, v_i), (v_j, w) \in E(G)$ such that $1 \geq i \geq j \geq r$. The number of inner paths now becomes limited, since Reduction rule 19 did not introduce an edge (u, w) as a shortcut. Two of these paths may share the first or last edge connected to the non-edge or loop they contribute to. As a result, every two of these paths increase the number of disjoint $u \rightsquigarrow w$ paths between non-edges or loops (u, w) by one. Therefore there are at most $2k$ of these inner paths contributing to (u, w). Since there are at most $|F|$ such vertices and $|F|(|F| - 1)$ possible non-edges, there are at most $2k|F|^2$ inner paths.

The maximum total number of paths in $G[A]$ that contain vertices in A_2 is therefore $4k|F|^2 + 2k|F|^2 - 4k|F| - 6$.

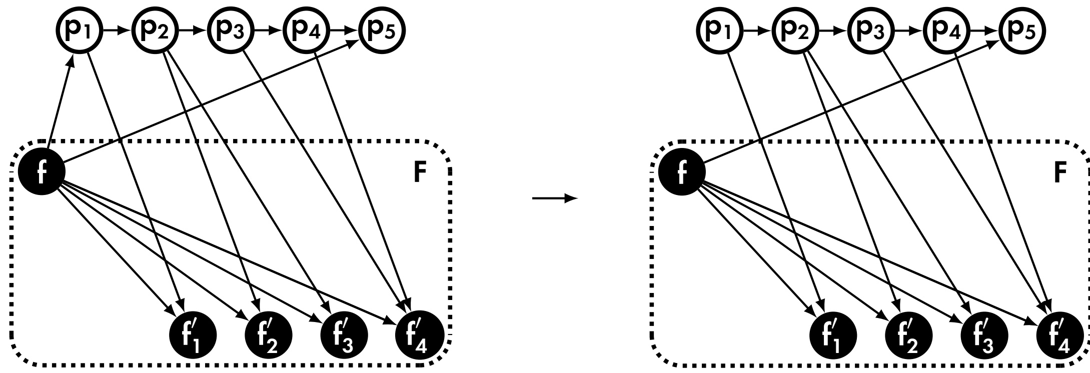


Figure 4.6.: Example of an application of Reduction rule 23

4.2.6. Bounding the length of paths

Bergougnoux et al. continue by providing a bound on the length of both outer and inner paths to bound the size of A_2. They first bound the number of edges from vertices in F entering these paths on their inner vertices.

For a vertex $f \in F$, with two successors $p_1, p_r \in N^{\rightarrow}[f] \cup A$ connected by an induced path $P = p_1 \rightsquigarrow p_r$ with p_1 and internal vertices being balanced vertices in A_2, and without an edge $(f, p_i) \in E(G), p_i \in \mathrm{internal}(P)$, we call P an *out-segment* for f.

Bergougnoux et al. now bound the number of possible out-segments for each vertex $f \in F$. They apply the following Reduction rule 23 on all out-segments of f. We again give the rule of Bergougnoux et al. (Rule 7, 2021) in a generalized form, which is subsumed by Reduction rule 14. Figure 4.6 gives an example for such a rule application.

Reduction rule 23: Non-contributing edge

Given edges (f, p_1) and (f, p_r), if for an induced path between their endpoints $P = p_1 \rightsquigarrow p_r$ with $f \notin {}^{\rightarrow}N[\mathrm{internal}(P)]$ for all successors $f' \in N^{\rightarrow}[\mathrm{internal}(P)]$ an edge (f, f') exists, we can delete (f, p_1). If $f = f'$, it would require (f, f') to be a loop.

Proof of safeness

Each cycle that (f, p_1) lies on leads through both f and either p_r for which an edge (f, p_r) exists or some f' that is reached by an edge (f, f'). Since for all cycles through f and $p_1 \rightsquigarrow p_r$ or $p_1 \rightsquigarrow f'$ that are closed by a path $p_r \rightsquigarrow f$ or $f' \rightsquigarrow f$, the cycle using the edge (f, p_r) or (f, f') needs to be covered as well. Removing (f, p_1) therefore does not change any possible minimum solution. □

Reduction rule 23: Non-contributing edge (continued)		
Overview		
New n	n	No vertices are removed.
New m	$\leq m-1$	The edges is removed.
New k	k	Vertices are unchanged.
Matching time	$\mathcal{O}(n^3)$	For each vertex f, $\delta^{\rightarrow}[f]$ is at most n, with paths of length at most n leading to p_r, with each vertex on this path required to be tested with n successors of f.
Implementation		
Indirectly in `DeleteNonInducedEdge`		

Reduction rule 23 is subsumed by Reduction rule 14.

Proof Suppose we perform our Reduction rule 14 on (f, p_1). It would then perform its search, however it would not be allowed to reach any of the successors $f' \in N^{\rightarrow}[p_i]$, nor would it be allowed to pass through p_r since in both cases an edge (f, t) or (f, f') exists. □

For each $f \in F$, there are at most $k\,|F|$ out-segments for f.

Proof Each out-segment of a vertex $f \in F$ contributes to at least one non-edge in F or f itself. There can be at most $|F|-1$ such non-edges to vertices in F that such paths would contribute to. For each such non-edge and f itself, there can be at most k paths contributing to it, as otherwise Reduction rule 19 would have introduced an edge or loop, respectively. □

As a result, each vertex $f \in F$ can only have at most $8k\,|F|^2 + 4k\,|F|^2 - 8k\,|F| - 12$ neighbors in A_2. We obtain this by multiplying the number of possible path segments for f which is one both for incoming and outgoing path segments with the number of both inner and outer path-segments, $4k\,|F|^2 + 2k\,|F|^2 - 4k\,|F| - 6$ which we obtained in Section 4.2.5. Since there are at most $|F|$ such vertices, we obtain at most $8k\,|F|^3 + 4k\,|F|^3 - 8k\,|F|^2 - 12\,|F|$ vertices in A_2.

4.2.7. Completing the bound

We can compute the sum of the previously collected bounds to obtain the complete size:

1. There are at most $|F|^2$ vertices contributing to loops, see Section 4.2.1.
2. There are both at most $k\,|F|^2 - k\,|F|$ vertices in A_0 and A_1, see Sections 4.2.2 and 4.2.3.
3. There are at most $8k\,|F|^3 + 4k\,|F|^3 - 8k\,|F|^2 - 12\,|F|$ vertices in A_2, see Section 4.2.6.
4. There are at most $k\,|F|^2 - k\,|F| - 2$ vertices in $A_{\geq 3}$, see Section 4.2.4.

In total, there are at most $8k\,|F|^3 + 4k\,|F|^3 - 5k\,|F|^2 + |F|^2 - 3k\,|F| - 14\,|F|$ vertices remaining. If we use its upper bound $|F|$ for k, this is clearly within $\mathcal{O}\left(|F|^4\right)$.

4.3. New Kernel based on new data reduction rules

Following the same arguments as Bergougnoux et al. (2021), we are able to simplify and slightly improve their kernel. We only rely on Reduction rules 4, 5, 14 and 19.

For each of the individual Reduction rules 20 to 23 used by Bergougnoux et al., we were able to prove Reduction rule 14 subsuming it, usually even in our much more general version. In some cases, we had removed remaining vertices with Reduction rule 4.

As a result, we are able to repeat all the claims, however instead of using a specifically computed Feedback Vertex Set or an approximation of it, we can rely on an hypothetically minimum FVS of our graph. All of the size bounds used after having applied the respective reduction rules still apply for our hypothetical case – without the specific FVS being used in computation.

We first apply Reduction rules 4 and 5 exhaustively. Using the result of S. Even and Tarjan (1975) as described in Section 2.1.3, we are able to complete the search for disjoint paths for Reduction rule 19 on each non edge. We then apply Reduction rule 14. We repeat this execution four times to ensure all preconditions of already applied rules are fulfilled for all of the respective rules of Bergougnoux et al. Afterwards, we apply Reduction rule 4 a single time to remove the vertices that now do not have adjacent edges since they were all removed by Reduction rule 14.

On instances that do not have a fixed size Feedback Vertex Set for the underlying undirected graph, the algorithm is not a kernel.

An obvious and easily constructed example would be the graph in Figure 4.7, constructed using the tunnels from Section 3.9.4 on page 70. They have a fixed solution size, in this example two with $S = \{v_1, v_2\}$ and can be extended indefinitely. A bound on the number of vertices in the graph for a given solution size therefore cannot exist. Furthermore, all edges in this example lie on induced cycles as all cycles in this graph have the exact same length.

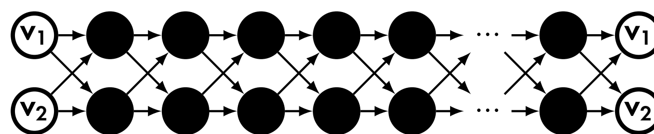


Figure 4.7.: Counterexample with an arbitrarily large graph of solution size two

5. Solving reduced instances

For real world instances as introduced for the PACE challenge, using the well known **FPT** algorithm ofChen et al. (2008) and its improved versions was not feasible since there were very large factors hidden in the $\mathcal{O}$-notation. Furthermore, the instances in part required very large solution sizes, the smallest non-trivial to solve containing fifty, the largest several thousands of vertices. We will discuss this further in Section 6.1. These sizes would be hidden in the high exponent in the parameter.

We further realized that instead of using regular branch and bound algorithms, reducing to other problems such as ILP and SAT and using existing, highly optimized solvers for solving was most efficient. We will first look into the most naive approach of effectively using a non-polynomial reduction to Hitting Set, make it feasible by applying it iteratively, then use induced partial orders which address the main problem of the iterative approach and finally combine both approaches.

5.1. Adding constraints for cycles iteratively

A naive approach to solve DFVS on a graph G is a reduction to Hitting Set. Take the vertices $V(G)$ as elements used in the sets of the universe of the Hitting Set instance. For every cycle of the graph, add a set containing all of its vertices. We can then take the resulting Hitting Set instance and solve it - either using a solver specialized on Hitting Set or by reducing Hitting Set in turn to SAT or ILP as in Algorithm 5.2. A minimum Hitting Set solution directly translates to a minimum DFVS.

This reduction however is not polynomial as there can be exponentially many cycles in G. Similar to our approaches used in several reduction rules in Chapter 3, a Hitting Set instance consisting only of induced cycles would be sufficient. The number of induced cycles however can still be exponential for a size n, especially when allowing for long cycles as in the counterexample depicted in Figure 4.7 on the facing page.

Instead of computing and solving a complete Hitting Set instance, we can modify our approach to become iterative, Algorithm 5.1. We keep track of a set of cycles that need to be covered for the solution. We then create a Hitting Set instance from the current subset of cycles and solve it. With the remaining graph after removing the solution, we search for a cycle using a BFS

after having searched for components and applying Kahn's algorithm (Kahn, 1962), essentially Reduction rule 4, add it to our subset of cycles and go back to recomputing the HITTING SET. As soon as the remaining graph does not contain any cycles, the solution to our HITTING SET instance is also a minimum DFVS of the graph.

Algorithm 5.1: Iterative HITTING SET generation

```
Input: A graph G
Output: Minimum DFVS S on G
1  H := ∅                          // The set of cycles to hit is initially empty
2  while true do
3  |  S := HS(V, H)                        // Solve HittingSet for current cycles
4  |  G' := G[V(G) \ S]      // Copy the original graph and remove solution
5  |  kahn(G')                       // Ensure only relevant vertices remain
6  |  if G' is empty then
7  |  |  return S                                   // S is already a DFVS for G
8  |  end
9  |  v := pick(V(G'))                        // Chose an arbitrary vertex in G'
10 |  C := BFS(v, v)                        // Search for the shortest v⇝v path
11 |  H ← H ∪ {V(C)}                 // Add C to the system of tracked cycles
12 end
```

Algorithm 5.2: HITTING SET ILP Formulation

```
Input: A set of vertices V, sets of cycles H
1  for each v ∈ V do
2  |  add variable x_v ∈ {0, 1}       // Represents vertices being part of the
   |    solution
3  end
4  for each C ∈ H do
5  |  add constraint ∑_{v∈C} x_v ≥ 1            // One of the vertices needs to be
   |    selected
6  end
Objective: Minimize ∑_{v∈V} x_v
Result: {v ∈ V | x_v = 1}
```

This has been implemented in `exact.IterativeCycleSolver`.

The iterative algorithm Algorithm 5.1 always finds a minimum DFVS.

Proof Clearly, the algorithm returns a DFVS, as it only terminates if it has obtained one. We thus need to confirm two things:

1. The algorithm terminates.

2. The returned DFVS is actually minimum.

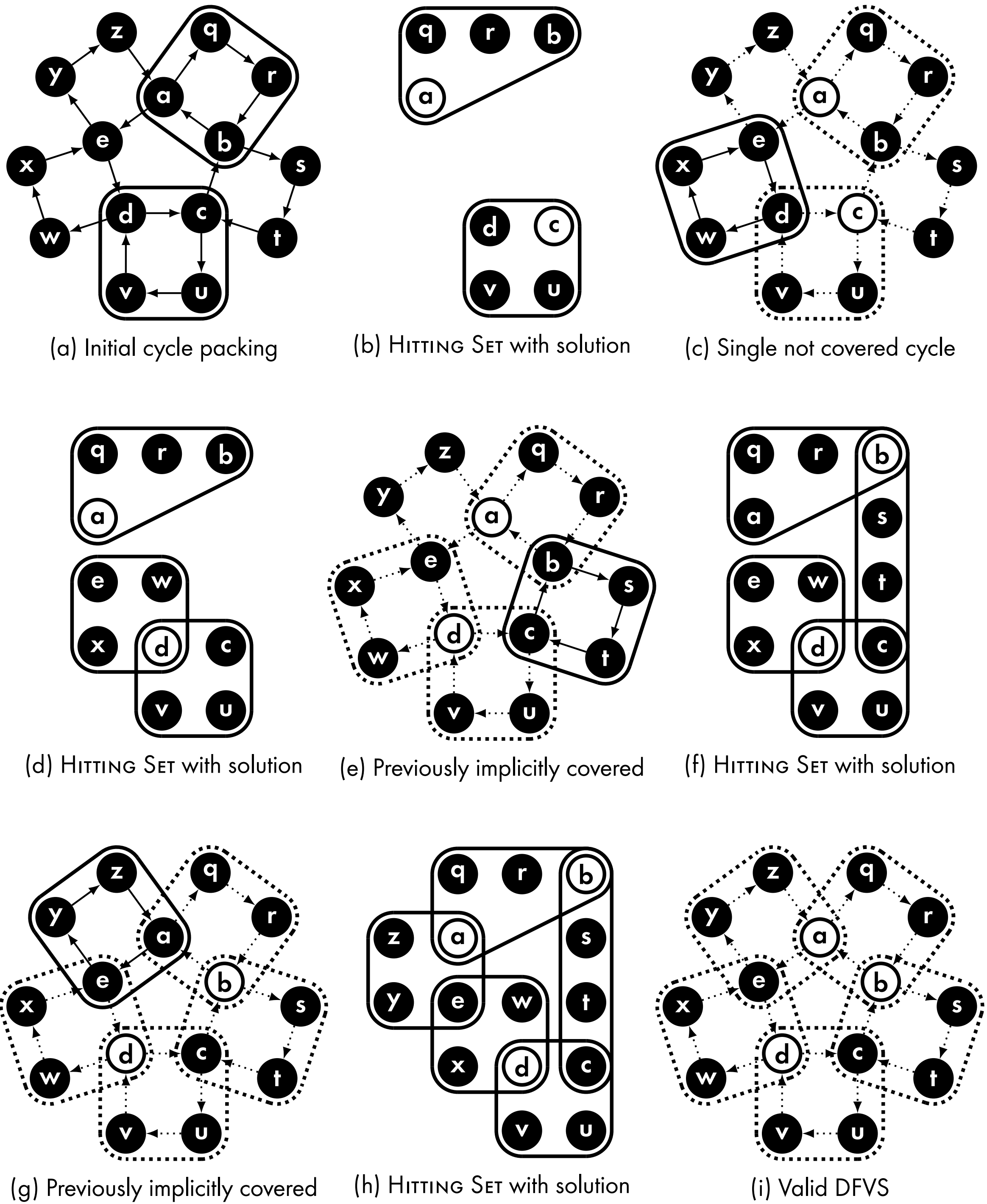
We can directly show that the algorithm terminates. The Hitting Set, per definition, must hit each set it is provided with, hence every current cycle in the Hitting Set set system H. Every iteration of the algorithm identifies at least one additional cycle to add to H – otherwise $\textsc{hs}(H)$ was already a DFVS for the graph. The number of such cycles, though possibly exponential, is finite. As such, the algorithm terminates at the latest when all cycles have been added to the Hitting Set instance.

We can argue in the same way that the solution must be minimum. We have demonstrated above in Section 2.3.4 on page 29 that the Hitting Set over all cycles corresponds to a minimum DFVS. Within each iteration, our Hitting Set instance contains only subset of all cycles of the graph and therefore an exact subset of sets of the Hitting Set instance over these cycles. The solution for such a subset of a Hitting Set instance can only be smaller than the solution of the complete instance, as we could in any case use the solution of the complete instance to hit all sets. Since we only stop if the solution is a DFVS, it implies that it has the exact same minimum size. □

As solving in itself takes the largest amount of time, we aim for as few required iterations as possible. We can precompute a system of short induced cycles as these are most likely relevant for the solution. In each step, we can search for a packing of short cycles in the remaining graph and add all these at once. We have to rely on a heuristic, especially since it is not yet known whether packing shortest cycles is **FPT** on general graphs (Bentert et al., 2024). Alternatively, we can use strategies that add even more cycles in such an iteration. A very feasible approach was computing a cycle cover, such that each edge of the remaining graph is contained within at least one cycle. We add all cycles up to a specific length and for the remaining edges perform a BFS to find a cycle they lie on. We do not perform an additional search for edges on a cycle that have been found in the second step.

> This has been implemented in `tools.EdgeCycleFinder`.

This however is not always an improvement when compared with initially computing the set of all cycles. In the worst case, we actually need all cycles in order for the Hitting Set solution to be a DFVS. Figure 5.1 illustrates these issues. It depicts the progression of iteratively collecting cycles on a graph consisting of five induced cycles, each overlapping with two others in exactly one vertex. We alternate between the graph with vertices from the intermediate solution having been removed and the corresponding Hitting Set instance with its solution for the next iteration. We could continue this construction further for any uneven number of intersecting cycles. Graphs constructed in this fashion contain exactly two additional cycles, an induced cycle in the center and a non-induced cycle on the outside, that are not relevant for our argument.



(a) Initial cycle packing
(b) Hitting Set with solution
(c) Single not covered cycle
(d) Hitting Set with solution
(e) Previously implicitly covered
(f) Hitting Set with solution
(g) Previously implicitly covered
(h) Hitting Set with solution
(i) Valid DFVS

Figure 5.1.: Iterative solving adding cycle packings having to add almost all cycles

We add maximal cycle packings for the remaining graph on every step. Apart from the initial graph in Figure 5.1a, these however only yield the single cycle not covered by the intermediate solution. We select a different element for the Hitting Set that results in uncovering a previously implicitly covered cycle twice, Figure 5.1e. Only after using all these cycles as constraints, we receive a valid dfvs.

The problem of this approach is, that the algorithm has a diminishing return. Especially during later iterations, the solutions of the Hitting Set instances are already close to the actual solution, but a few cycles that would have been "accidentally" covered before now come up and need to be taken into account on every round.

Figure 5.1 demonstrated this. Most of the iterations did not raise the solution size. We mitigate this to a degree by immediately adding several short cycles before starting the first iteration, which would immediately have creating a feasible instance for our example.

5.2. Linear and partial orders

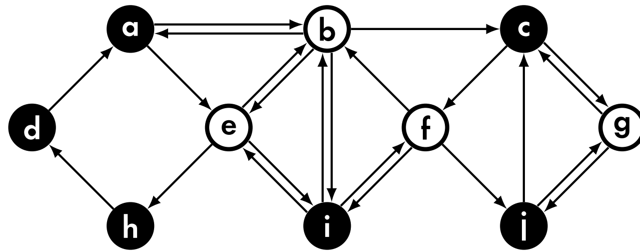
The solution to this main problem of the iterative approach is directly creating an instance for the SAT or ILP solver to solve, with a result that directly translates to a valid minimum dfvs.

As a fairly simple approach, we could impose a linear order on the vertices and then track which vertices are part of the solution. Then, for every edge, we require that either its source comes before its target with respect to the order, its source is part of the solution or its target is part of the solution. The minimum set of vertices required to find a solution for this problem is exactly a minimum dfvs.[1]

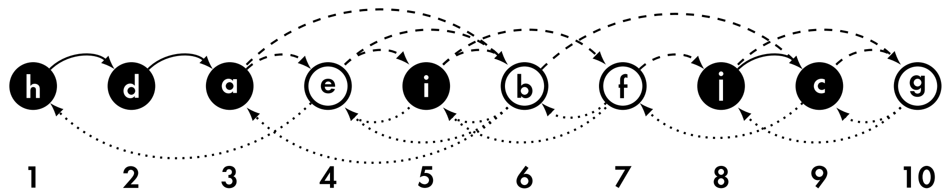
We show an example of a linear order in Figure 5.2b that allows the minimum solution $S = \{b, e, f, g\}$ on the graph from 5.2a. Covered allowed edges are dashed, covered disallowed edges are dotted.

The order can actually be partial, we can ignore vertices having the same rank with respect to the order as long as they are not connected by an edge or are part of the solution. We give such an induced partial order of minimum width, that is the least difference in ranks for the previous example in Figure 5.2c. As previously, covered allowed edges are dashed, disallowed edges all of which are covered are dotted.

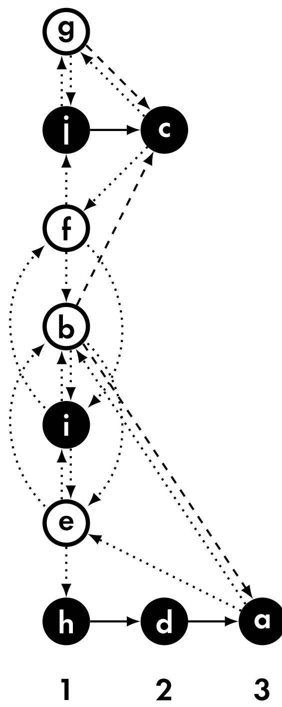
[1] In this regard, DFVS is related to Kelly Width, which can also be defined with linear orders (Hunter and Kreutzer, 2008). We can take the k vertices of our DFVS and use the same linear order of vertices so that each vertex only has these k vertices as strongly reachable vertices. The number of such vertices and thereby Kelly Width of our graph is therefore at most k.



(a) Original graph with the unique minimum DFVS



(b) A linear order with the minimum solution



(c) A minimum width induced partial order for the minimum solution

Figure 5.2.: Example of a linear and an induced partial order

Encoding such an order within SAT requires a table like structure which would lead to a quadratic number of required variables. This can be lowered to $n \cdot \log(n)$ by assigning a binary encoded rank to each vertex. Creating constraints to ensure this would be tedious, which is why we use EXTENDED SAT, Problem 6 on page 32. We give an encoding for EXTENDED SAT in Algorithm 5.3. Overall, the solver needs to find a solution with the least vertices selected for the solution.

Algorithm 5.3: Partial order EXTENDED SAT formulation

Input: A graph G
Variables:
1 **for each** $v \in V(G)$ **do**
2   **add** $x_v \in \{true, false\}$ `// Represents vertices being part of the solution`
3   **add** $y_v \in [1, n]$ `// Represents the rank with respect to the order`
4 **end**
Constraints:
1 **for each** $(s, t) \in E(G)$ **do**
2   **add** $(y_s < y_t) \vee x_s \vee x_t$ `// Either adhere to the order or select` s `or` t `for the solution`
3 **end**
Result: $S = \{v \in V \mid x_v\}$

Similarly, we can create an ILP formulation, Algorithm 5.4. We create two variables for each vertex, one representing a vertex being part of the solution using integer values $\{0, 1\}$ and a variable containing its rank with respect to the order $[1, n]$. Comparing vertices with respect to the order becomes simple as we can simply subtract ranks and ask for a positive solution size. Requiring for an edge against the order to have either source or target within the solution is less obvious, we however can simply add the value of the source and the target being part of the solution scaled up beyond the range allowed for the order by multiplying it with $n + 1$. Again, we optimize for as few vertices in the solution as possible.

This has been implemented in `exact.LinearOrderedSolver`.

Proof First, we show that the such a solution is actually a DFVS. Clearly, if none of its vertices is included in the solution S, the source of every edge has a lower rank than the target with respect to the order. This obviously translates to all pairs of predecessors and successors within paths that do not lead through S. A cycle is therefore impossible, as it would need to contain an edge that would violate this property with respect to its path.

Secondly, we need to show that given a DFVS, a solution of the formulation exists. Suppose a set of vertices S is a minimum DFVS for a graph G. The subgraph $G' = G[V(G) \setminus S]$ is a forest containing no cycles. We take all vertices $V(G')$ that have no incoming edge and give them the rank 1 within the order. We continue by assigning rank 2 to all vertices in $V(G')$ that

Algorithm 5.4: Partial order ILP formulation

Input: A graph G
Variables:
```
1 for each v ∈ V(G) do
2   add x_v ∈ {0,1}    // Represents vertices being part of the solution
3   add r_v ∈ [1,n]    // Represents the rank with respect to the order
4 end
```
Constraints:
```
1 for each (s,t) ∈ E(G) do
2   add r_s − r_t + (n+1)·x_s + (n+1)·x_t ≥ 1 // Either adhere to the order
      or select s or t for the solution
3 end
```
Objective: Minimize $\sum_{v \in V} x_v$
Result: $S = \{v \in V \mid x_v = 1\}$

have no incoming edge from all except the previously assigned vertices and continue until we have covered all vertices. This finishes as G' is a forest and thus does not contain cycles. Each edge in G now either respects the order as previously constructed or either its source or target is contained in S.

The problem with this approach is, that the SAT and ILP solvers tend to get stuck on such instances, compared to fairly efficient solving on instances resembling Hitting Set instances.

5.3. Hints

In effect, we are looking for an approach that combines the best of both worlds. Luckily, we are able to combine the techniques given above even on the SAT/ILP-formulation level.

The ILP and SAT formulations above can be extended by further constraints which provide hints to the solvers. As a result, the solver does not need to infer these rules on its own using less efficient general rules. We provide the rules both in SAT and ILP formulations and assume that variables represent vertices as $x_v, v \in V(G)$ with x_v satisfied or assigned 1, respectively, to indicate that they are part of the solution.

5.3.1. Bi-directed edges

The most obvious improvement is to find all two-cycle $D = \{u, v \mid (u, v), (v, u) \in E(G)\}$. We can now add a constraint that ensures that either of its vertices needs to be part of the solution. For each two-cycle $\{u, v\} \in D$, we add a constraint:

For SAT

$$x_u \vee x_v$$

For ILP

$$x_u + x_v \geq 1$$

This constraint obviously does not alter the solution size, as it could have been directly inferred from existing constraints.

This is especially beneficial for modern SAT solvers as they can immediately use these constraints for *unit resolution*, sometimes called *unit propagation*, to propagate the effects of a decision as soon as either vertex has been picked for the solution. This results in increased efficiency of branch-and-bound algorithms (Darwiche and Pipatsrisawat, 2021).

Furthermore, we are able to remove the edge constraints from our partial order ILP formulation. At least one vertex of this cycle is already selected, thus eliminating any possible longer cycles that would go through the two-cycle.

5.3.2. Short cycles

We can similarly add hints for short cycles H of length greater than two. Since at least one vertex on such a cycle needs to be included in the solution, we can add a constraint providing that lower bound to the solver.

For each short cycle $C \in H$, we add a constraint:

For SAT

$$\bigvee_{v \in C} x_v$$

For ILP

$$\sum_{v \in C} x_v \geq 1$$

Constraints of this form do not alter the solution size either. A further constraint cannot reduce the possible solution size. If adding such a constraint would increase the solution size, then a previous solution would not have included at least one vertex of the cycle and would thus not have been a valid DFVS.

As a consequence, we are allowed to add such constraints for any set of cycles present in the graph. If the iterative cycle solving from Section 5.1 has effectively plateaued at a solution size and does find the necessary constraints, we are effectively able to add the constraints enforcing an induced order from Section 5.2 to ensure all cycles are respected.

In practice, even when starting partial order based solving, we will first compute all induced cycles of small length and start with these as hints right from the beginning.

5.3.3. Edge on multiple three-cycles

For two or more induced cycles of length three that share an edge (u, v) and with non shared vertices $w_1, \ldots, w_i \in W$, as in Figure 5.3, we can add a constraint that forces us to either add u, v or all vertices in W:

For SAT

$$x_u \vee x_v \vee \bigwedge_{w \in W} x_w$$

For ILP

$$|W| \cdot x_u + |W| \cdot x_v + \sum_{w \in W} x_w \geq 1$$

Again this obviously does not increase solution size as it directly follows from the other constraints.

To find such constructs, we can iterate over all edges $(s, t) \in E(G)$ and simply search for predecessors and successors of its source and target and take the cut ${}^{\rightarrow}N[s] \cap N^{\rightarrow}[t]$ of these sets.

> This has been implemented in `tools.ThreeCycleFinder#findLogicalThreeCycles`.

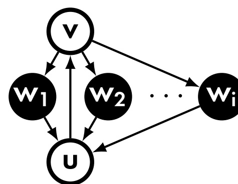


Figure 5.3.: Shared edge on three-cycles

5.3.4. Lower bounds of subgraphs

If we know the minimum solution size k' of a subgraph $H \subseteq G$, we can add this minimum solution size as a constraint for the respective vertices of the supergraph. This is easily done for ILPs.

Using a similar approach as in Section 5.2 on page 89, we could introduce counting for Extended SAT by adding numeral variables $y_v \in \{0, 1\}$ for each $v \in V(G)$ and increase its value if the respective vertex is in the solution with

$$\bigwedge_{v \in V(G)} (x_v \Leftrightarrow y_v = 1)$$

although in our specific case, $x_v \Leftarrow y_v = 1$ would have been sufficient. However, if we target SAT or via their intermediate Max SAT solvers, we generally do not benefit that much from this sort of constraint, although we could use optimized counting strategies effectively adding a logarithmic number of variables (Bittner et al., 2019).

A regular SAT constraint

$$\bigvee_{H' \subseteq \{V(H) || H'| = k'\}} \bigwedge_{v \in H'} x_v$$

For ILP

$$\sum_{v \in V(H)} x_v \geq k'$$

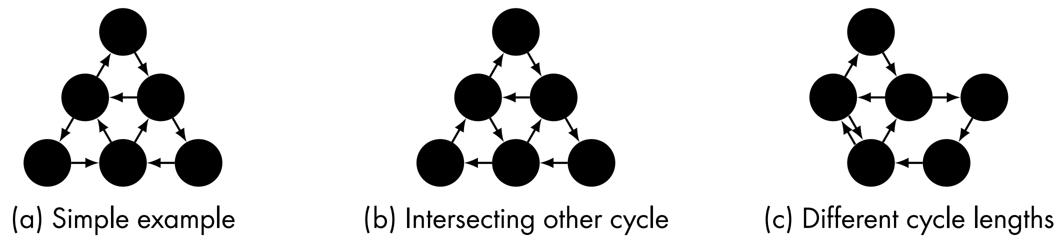
can be simplified to

$$\sum_{v \in V(H)} y_v \geq k'$$

when using extended SAT with counting.

Proof Suppose a lower bound of k' for $V(H)$ was incorrect. Then there would be a solution S for G that contains fewer vertices in H than for the minimum solution size of H. However, then $S \cup V(H)$ would have to be a solution for H, since $G[V(G) \setminus S]$ was a forest. This would contradict k' being the minimum size of a solution on H. □

This is especially beneficial for a graph of which we know the solution size, and which we can quickly identify as a subgraph. As such, it is a generalization of the short cycles hint from Section 5.3.2, since these can be represented as their vertex set with a lower bound of one. Another structure that fulfills this condition is the Triforce, a set of three cycles pairwise intersecting in single, different vertices, such as in Figure 5.4a. The Triforce may intersect other induced cycles as in Figure 5.4b. The cycles may have any length, , Figure 5.4c, however computing them for cycles longer than four was ineffective.



(a) Simple example (b) Intersecting other cycle (c) Different cycle lengths

Figure 5.4.: Examples of Triforces

5.3.5. Cliques

A special case of lower bounds and a further generalization of the two-cycle hint from Section 5.3.1 are cliques. For an l-clique, at least $l-1$ vertices need to be selected for the solution.

We only need to keep track of cliques that are maximal, as this still only allows for one non-selected vertex in a sub-clique. This also applies to bi-directed edges if they are included in a larger clique. There are solving techniques to quickly enumerate maximum cliques (Eppstein, Löffler, et al., 2013; Eppstein and Strash, 2011) and solvers implementing them such as *Quick-Cliques*[2].

> This is implemented in the library: `algorithms.NativeMaximumCliqueEnumeration`.

The observation that we are able to remove constraints for imposing the induced partial order still holds. We can remove the bi-directed edge constraints, as long as we regard the bi-directed edges not part of larger cliques as two-cliques and keep them as constraints.

This does not make that much sense for Max SAT, as modern solvers will usually detect cliques as essentially encoding $l-1$-constraints, which they can handle in an optimized way (Biere, Le Berre, et al., 2014), although *CaDiCal* which we used indirectly via *EvalMaxSAT* did not include this feature (Biere, Fazekas, et al., 2020). At the same time, they are still well suitable for unit resolution as explained in Section 5.3.1.

5.4. Combined formulation

Taking the formulations for Hitting Set, induced partial orders and hints, we obtain a formulation that combines the different approaches in Algorithm 5.5. We extend the approach for induced partial orders from Section 5.2.

[2]QuickCliques, Darren Strash, GitHub https://github.com/darrenstrash/quick-cliques

We create the induced partial order only on the directed subgraph. In order to obtain it, we first compute the bi-directed edges D and ignore them when creating constraints for the partial order. We also identify edges that lie on multiple three-cycles and add these with the set of their other vertices W as T.

If constraints for cycles were previously added iteratively, we can use their set systems as lower bounds of 1 on the respective subgraphs. Otherwise, we specifically compute short induced cycles to be used as lower bounds. We can then enumerate the maximal cliques in the graph. Finally, we compute Triforces for their lower bounds of 2. We supply all of these constraints as a map $(W, k') \in B$.

> This has been implemented in `solver.LinearOrderedSolver`.

Such a combined formulation with its minimum solution is shown in Figure 5.5. It first shows the hard constraints for the directed subgraph (5.5b) with an already valid partial order in 5.5c and the undirected subgraph (5.5d) with the constraints derived from its maximum cliques, 5.5e.

Algorithm 5.5: Combined ILP formulation

Input: A graph G, its bi-directed edges D, shared three-cycle edges T and lower bounds for subgraphs B

Variables:

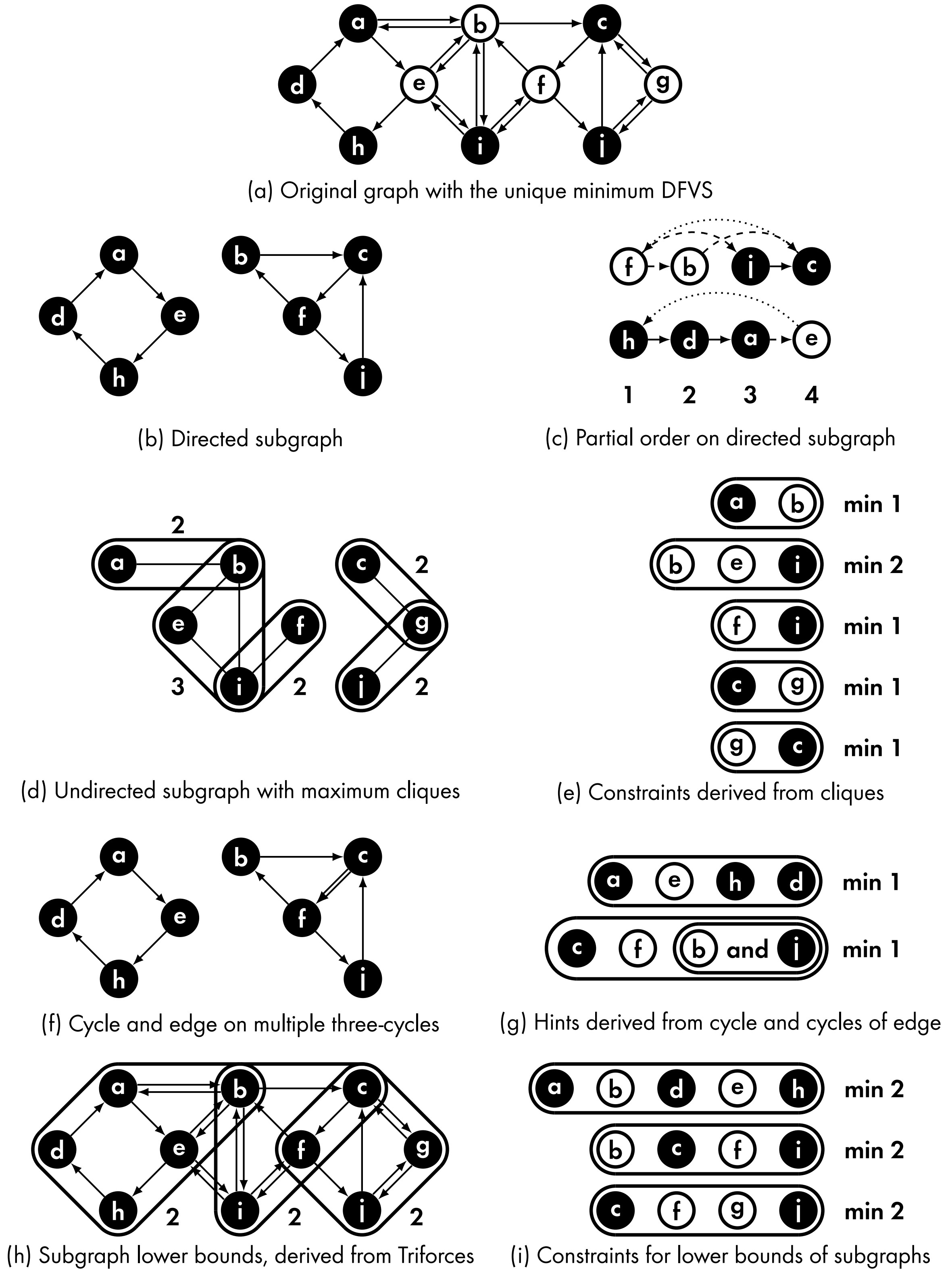
1 **for each** $v \in V(G)$ **do**
2 **add** $x_v \in \{0, 1\}$ `// Represents variables being part of the solution`
3 **add** $r_v \in [1, n]$ `// Represents the rank with respect to the order`
4 **end**

Constraints:

1 **for each** $(s, t) \in E(G)$ **do**
2 **if** $\{s, t\} \notin D$ **then**
3   **add** $r_s - r_t + (n+1) \cdot x_s + (n+1) \cdot x_t \geq 1$ `// Either adhere to the order or select` s `or` t `for the solution`
4 **end**
5 **end**
6 **for each** $(u, v, W) \in T$ **do**
7 **add** $|W| \cdot x_u + |W| \cdot x_v + \sum_{w \in W} x_w \geq |W|$ `// Select` u`,` v `or all in` W
8 **end**
9 **for each** $(W, k') \in B$ **do**
10 **add** $\sum_{w \in W} x_w \geq k'$ `// Apply precomputed lower bounds on subgraphs`
11 **end**

Objective: Minimize $\sum_{v \in V} x_v$

Result: $\{v \in V \mid x_v = 1\}$



(a) Original graph with the unique minimum DFVS

(b) Directed subgraph

(c) Partial order on directed subgraph

(d) Undirected subgraph with maximum cliques

(e) Constraints derived from cliques

(f) Cycle and edge on multiple three-cycles

(g) Hints derived from cycle and cycles of edge

(h) Subgraph lower bounds, derived from Triforces

(i) Constraints for lower bounds of subgraphs

Figure 5.5.: Structure of constraints of the combined formulation

5.5. Reduction to Vertex Cover

For graphs that predominantly consist of bi-directed edges, as did some of the PACE challenge, we are able to use a different approach. We perform a reduction to Vertex Cover with the goal of using one of the efficient existing solvers for this problem. We expand upon the idea of the non-polynomial Hitting Set to Vertex Cover reduction from Section 2.3.4.

If the graph consist only of bi-directed edges, we can immediately take its underlying undirected graph as a Vertex Cover instance. We have effectively undone a Vertex Cover to DFVS reduction as in Figure 2.7. The minimum Vertex Cover is also a minimum DFVS on the original graph.

> This has been implemented in `exact.VertexCoverSolver`.

5.5.1. Replacing cycles with Hitting Set gadgets

If some directed edges remain, we can perform a DFVS to Hitting Set to Vertex Cover reduction. Neither of the reductions is polynomial, however both may be suitable on instances with a low directed edge density. As we only need to take care of induced cycles, we can treat the bi-directed edges independent from the directed ones and directly add them to our set system and thus as edges for our Vertex Cover instance.

We then need to enumerate all other induced cycles. We can perform a recursive DFS on all remaining vertices without continuing branches on vertices that have an edge leading back to a member of the current path, similar to our reduction rule in Section 3.7.

For each cycle of length l in our Hitting Set instance, we create a Vertex Cover gadget, an l-clique with each vertex of the cycle connected to a different vertex of the clique. Examples for individual cycles are shown in Figures 5.6a and 5.6b.

> This has been implemented in `reductions.AnyHittingSet`.

5.5.2. Optimized gadgets

In a few cases, we are able to reduce the number of needed gadgets. If the directed graph contains two or more cycles share an arbitrary long path segment $s \rightsquigarrow t$ and split for individual path segments $(t, w_1, s), \dots, (t, w_i, s)$ we can use a single gadget, but instead connect one vertex

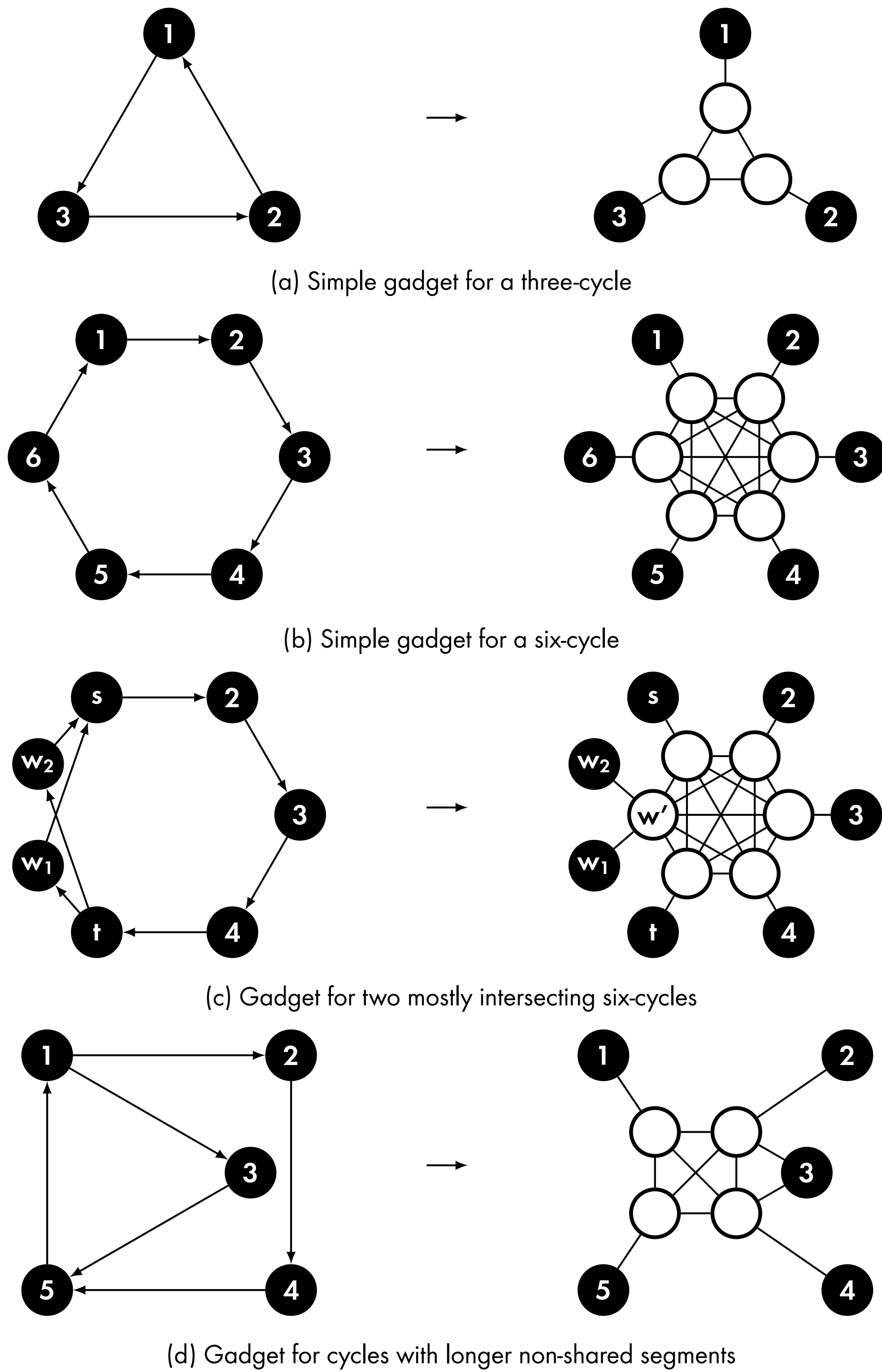


Figure 5.6.: Vertex Cover gadgets

w' of the clique to all $w_1, \ldots, w_i$. Figure 5.6c demonstrates this with w_1, w_2 being connected to w'. If the solution was exclusively inside of the clique, we push any vertex other than w' out to be part of the solution.

This variant can save a large number of vertices. As a result, we would compute it before the general version of the rule and delete the respective edges.

This has been implemented in `reductions.ThreeHittingSet`.

On the graph in Figure 5.7a we would create two cliques for the two three cycles on the left side as depicted in Figure 5.7b. If we used the variation, we could save half of these additional vertices and nearly half the edges as shown in Figure 5.7c. The gadget for the four-cycle remains untouched by this variation as that cycle is not adjacent to other longer cycles.

A vertex outside a minimum solution can never be adjacent to l cliques that have all of their internal vertices inside of the solution as long as it is sharing at most $l-1$ vertices inside of cliques with other vertices.

Proof Suppose a minimum solution did not select such a vertex v. In this case, we could create an improved solution by including v and push all of its adjacent neighbors from cliques that previously had been in the solution to neighbors v shared a vertex in the cliques with.

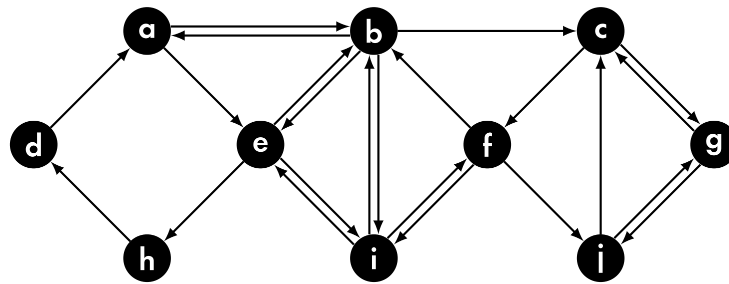
Since our precondition ensured exactly this, there are at most $l-1$ such vertices. As our vertex was adjacent to less than $l-1$ other vertices, we immediately obtain a smaller solution, a contradiction to our initial assumption. □

In principle, we could further extend this approach to allow for several different path constellations. We could introduce several non-shared segments that would each result in a vertex in the clique connected to all such vertices.

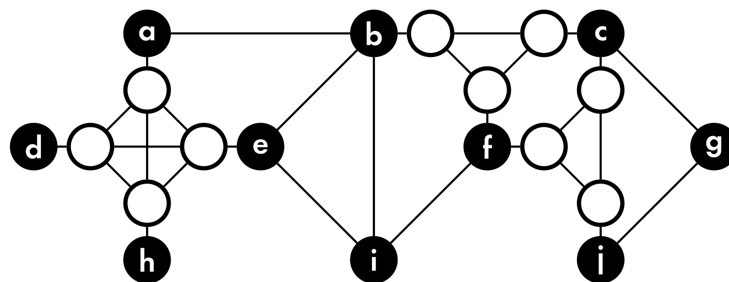
We could furthermore allow for the paths to split away for more than one vertex. In this case, we would need to have one vertex in the clique representing each possible permutation of vertices of the split off paths to be selected for the solution. As such, this quickly becomes infeasible if several paths of length greater than zero are inspected.

An example is shown in Figure 5.6d with a non shared segments with one and two vertices, respectively. Two paths with two internal vertices each would have to be represented with four clique members.

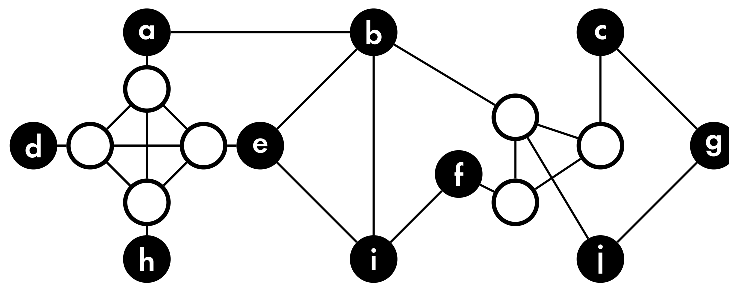
Overall, for all such modifications, we need to ensure that all paths share a single vertex towards which we could push our vertex for a solution.



(a) Mostly undirected instance

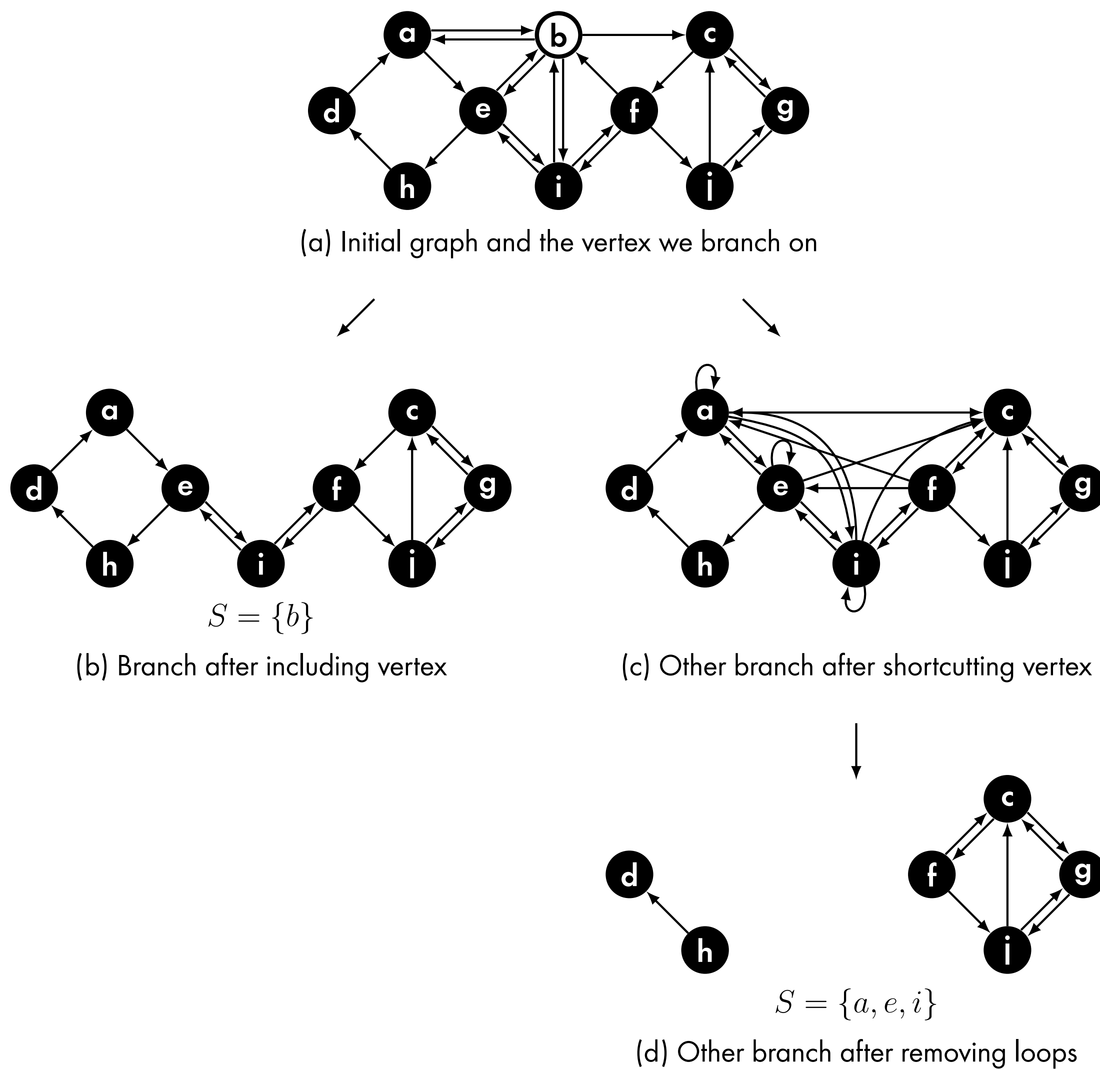


(b) Converted with three regular gadgets



(c) Converted with an optimized and one regular gadget

Figure 5.7.: Introducing gadgets to instances



(a) Initial graph and the vertex we branch on

(b) Branch after including vertex

(c) Other branch after shortcutting vertex

(d) Other branch after removing loops

Figure 5.8.: Branching example

5.6. Branch and bound

The classical approach to solving would be a branching algorithm that in a way tries to solve the problem with brute force, by trying out all possible combinations. In each branching step, such an algorithm takes a binary decision and then evaluates both branches that originate from it, usually applying methods such that it does not need to evaluate all branches towards their end.

In case of DFVS, the obvious choice would be a vertex v included or excluded in the solution. If v is included in the solution, we can effectively remove the vertex from the graph and simply increase the solution size of this branch. If it is excluded, we directly shortcut it, since we know it will not be part of the solution.

We could branch for example on high-degree vertices or vertices on short cycles being part of the solution. It then keeps track of the best achieved solutions and tries to quickly reject branches that are obviously worse than already established bounds.

In the example in Figure 5.8, we select the high degree vertex b, 5.8a and then continue with a branch where it is directly included in the solution (5.8b) and one where we shortcut (5.8c). In this branch the cycle (b, c, f) was shortcut to a single bi-directed edge $\{c, f\}$. Existing bi-directed edges were shortcut to loops on a, e and i which we can also add to the solution (5.8d), effectively applying Reduction rule 1.

An approach that is interesting in this regard is to compute a non-optimum solution using some heuristic first, prioritize these vertices and try to push the lower bound of a possible solution for this branch over the already established possible solution size.

To support such an approach, we need to be able to quickly determine a good upper- and lower bound of such a partial solution. This involves a trade off of a close result and more computation required to achieve such a result. We furthermore need efficient data structures to be able to backtrack quickly.

When the remaining instance is small enough, we can alternatively use other solving approaches detailed above or skip more expensive data reduction rules.

The branch and bound approach did not prove effective compared to reducing problem instances to problems for which highly optimized solvers already exist. We however briefly explain how such bounds could be obtained and how the approach could be extended with data reduction rules.

5.6.1. Upper bounds

Any valid DFVS on a graph is a safe upper bound. The following approaches generally provide non-optimal but still valid solutions for DFVS. Furthermore, as soon as we are able to find a lower bound of the same size, such a valid DFVS immediately becomes the solution to our minimization problem.

A very fast approach is to greedily pick vertices for the solution based on a heuristic, applying data reduction rules after each pick. An obvious and very fast to compute choice are high degree vertices. As we are on directed graphs, vertices however have a separate in-degree ${}^{\rightarrow}\delta\,[v]$ and out-degree $\delta^{\rightarrow}\,[v]$. A naive approach would be to simply add them, ${}^{\rightarrow}\delta\,[v] + \delta^{\rightarrow}\,[v]$. Generally, we want to pick vertices that contribute to as many cycles at the same time as possible. As a result, we achieved best results when weighing vertices that both have a high in- and out-degree higher. Using $\min\left({}^{\rightarrow}\delta\,[v]\,, \delta^{\rightarrow}\,[v]\right)$, we ensured that a vertex with a high degree in only one of the directions would not be picked. Directly multiplying the degrees as ${}^{\rightarrow}\delta\,[v] \cdot \delta^{\rightarrow}\,[v]$

had an even better effect, but does not account as much for imbalanced vertices anymore. We observed the best results with ${}^{\rightarrow}\delta\,[v] \cdot \delta^{\rightarrow}\,[v] \cdot 2 - ({}^{\rightarrow}\delta\,[v] \cdot \delta^{\rightarrow}\,[v])$, which still rewards vertices with a lot of possible cycles while again punishing imbalanced vertices.

> This has been implemented in `heuristic.DegreeHeuristic`.

An improvement at the expense of more computation time was to solve the LP relaxation of the combined formulation from Section 5.4 and then greedily pick the vertex with the highest weight. It would also be possible to directly pick several vertices at the same time, either a fixed quantity on every iteration or using vertices within certain bounds.

> This has been implemented in `heuristic.LpBasedHeuristic`.

An even more expensive and quite different approach that however achieved lower upper bounds was comparable to the iterative Hitting Set approach used in Section 5.1. We recursively compute a Hitting Set on an arbitrary set of cycles, greedily pick these vertices for the solution and, after applying reduction rules, recursively repeat this on the remaining graph.

> This has been implemented in `heuristic.IlpBasedHeuristic`.

Computing it on every level did not make sense, however using it on the first run usually provided a fairly close bound. As the Hitting Set instance on the first step might already be impossible to solve, using a timeout and falling back to other upper bounds is beneficial.

All of these previous approaches might have greedily picked a non suitable vertex early on, for example a vertex in the center of a very dense area where all its neighbors needed to be included in the solution for other reasons. An approach that aims to mitigate this problem is to take an existing solution and to iteratively push vertices out of it. It was used by Bafna et al. (1999) for their 2-approximation of Feedback Vertex Set. We however use any existing valid though not minimum DFVS S on the graph and remove a vertex v, such that $S' = S \setminus (v)$. If S' is still a DFVS on the graph, we keep it and continue with the next vertex until we have visited all vertices. Such a refined solution is now ensured to be minimal, albeit not minimum. We used the same way of weighting the vertices as for the greedy heuristic above, but started pushing out vertices with a low value first.

> This has been implemented in `tools.DegreeBasedSolutionRefiner`.

This approach can be used as a heuristic to compute the solution in itself. It had roughly the same speed as greedily picking vertices while providing much smaller solutions.

This has been implemented in `heuristic.GreedyRemoveHeuristic`.

5.6.2. Lower bounds

Lower bounds are especially relevant for branch and bound algorithms. As soon as we reach a lower bound higher than an already found solution in our previous branching attempts, we can stop continuing on this branch. However they have less uses outside of branching algorithms when compared with upper bounds. To compute such lower bounds, we can in several ways build upon concepts that we have used for hints in Section 5.3.

Any minimum solution for a subgraph is automatically a lower bound on its vertices. An example would be the Vertex Cover on the undirected subgraph which would imply a lower bound for the complete graph. Similarly, we could compute the sum of minimum solutions of disjoint subgraphs.

As such, we can try to pack disjoint structures of which we know a lower bound, Section 5.3.4. Since they each impose a lower bound on a separate part of the graph, we obtain a lower bound for the whole graph. Cliques generally provide the best of such lower bounds while a good packing of Triforces allows for a lower bound of two on subgraphs where we could only pack one cycle. If we have accounted for all Triforces, we can still attempt to pack disjoint cycles in the remaining graph.

This has been implemented in `lowerbound.TriforceDisjointCyclesLowerBound`.

The LP relaxation of any of the ILPs, especially Section 5.4, again provides a lower bound. We can collect both the aforementioned Triforces and cycles. The lower bounds we provide to the solver however do not need to be disjoint, so we could provide more Triforces and could also use a cycle cover such that for every edge at least one cycle containing it is supplied to the solver. We can simply sum all the weights assigned to vertices, ceil them to the next integer number and obtain our lower bound. This generally has a higher computation cost than the previous approach.

This has been implemented in `lowerbound.LpLowerBound`.

Again, we are allowed to use the minimum solution of any subgraph as the minimum solution for such a subgraph, so we would also be allowed to split the graph into several parts, compute exact solutions on these subgraphs and then take the sum of these exact solutions as the lower

bound. This would only make sense when combined with some strategy that reuses such computations and was not implemented. The results of this computation could also be passed to the LP.

Overall, all of these lower bounds were unsuccessful at providing high lower bounds, thus making branch and bound based solving infeasible.

5.6.3. Branch and reduce

After each decision made, we could also apply data reduction rules to reduce the size of our instance at an ideally fairly low computation cost. This modification to the ordinary branch and bound approach is called branch and reduce (Plachetta and Grinten, 2021). It has been partly successful for Vertex Cover (Hespe et al., 2020).

> This has been implemented in `exact.ManualBranchingSolver`.

Let us look at an example for a branch and reduce approach in Figure 5.9. Instead of looking at a specific graph as an example, we merely overlook the execution of the algorithm as a branching tree on a fictive example that covers the most relevant cases, though not all edge cases. We furthermore take an algorithm that can decide whether to include or exclude a vertex on an individual basis.

Initially, we apply data reduction rules, 5.9a. We then compute bounds using lower bounds from Section 5.6.2 and upper bounds, Section 5.6.1 in 5.9b. From now on, the labels represent the number of vertices we have already picked for the solution, in this case four from 5.9a, the minimum that this branch can at best achieve, currently three and the maximum number of vertices such that this branch does not perform worse than what we already have a solution for. Since our heuristic found a solution of size 14, this limit is at 10. Since we are at the root, this applies for the complete remaining instance.

We then make our first decision. Based on some heuristics, we decided on a vertex. For our first branch, we excluded it, adding three vertices in 5.9c. The bounds, 5.9d, obviously shift by three in this case, as three more vertices are definitely in the solution.

We continue with data reduction, 5.9e which in this case did not add any vertices to the solution but may have successfully removed edges. The lower bound 5.9f computed afterwards was increased by two.

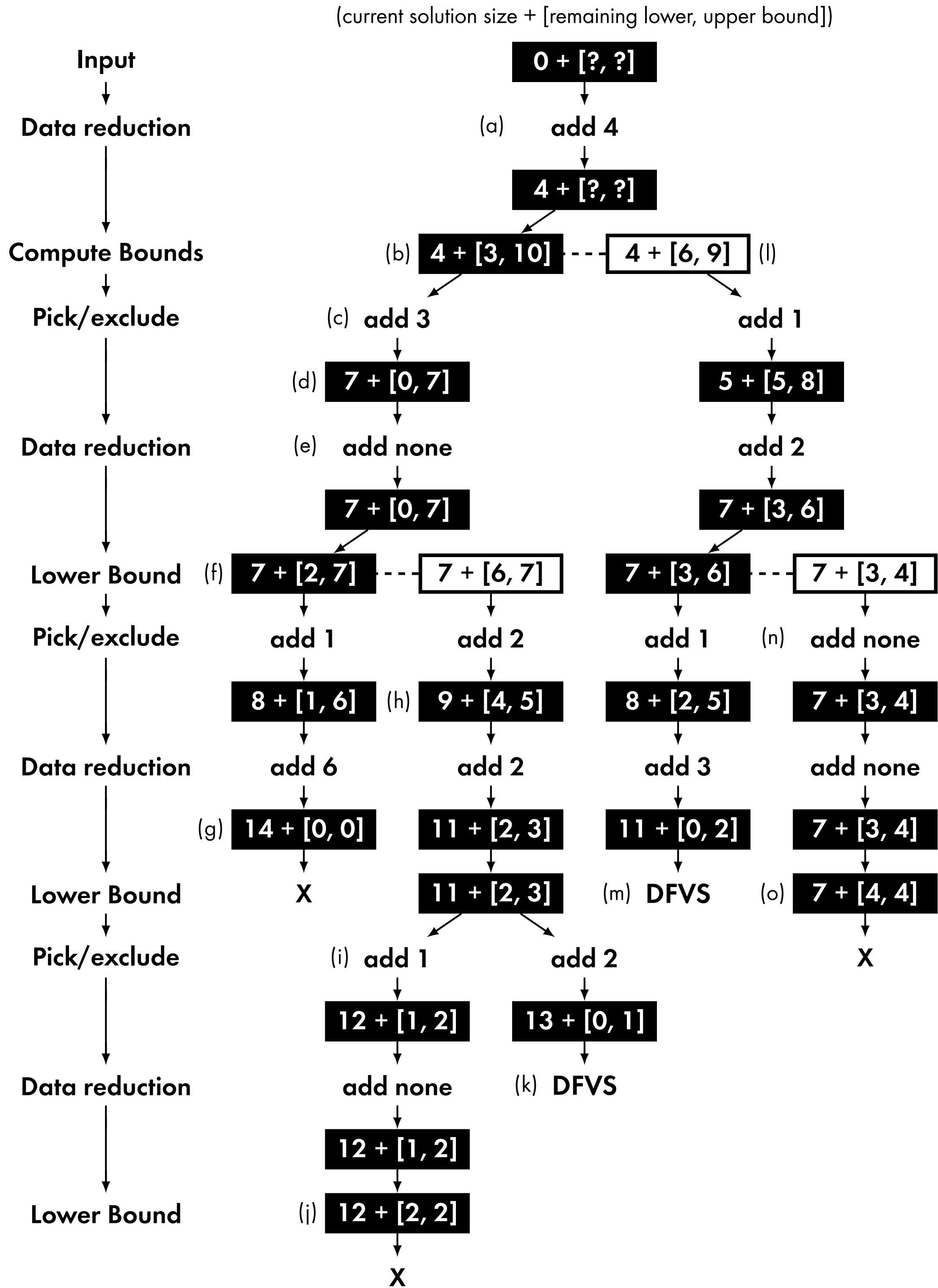


Figure 5.9.: Fictive branch and reduce tree

After repeating this, data reduction itself added eight vertices directly into the solution. We do not even need to compute the lower bound after 5.9g since we already know that our current minimum solution size of 14 in this branch will not be better than the solution already found. We therefore abandon this branch and return to our previous decision.

Instead, we attempt to push the vertex out of the solution first. In this case, this directly adds four vertices to the solution. However, we know a closer lower bound (5.9h) for this branch, as it can be at most better by one vertex as all these vertices could have also been selected in the other branch to obtain a solution.

Our decision to add a vertex (5.9i) draws our branch to the same fate as the previous one after we have computed a new lower bound of two remaining vertices, 5.9j. In this case, our bounds do not need to change. These two vertices are exactly the two vertices immediately added in the second branch which happen to be the remaining vertices for a valid DFVS in 5.9k. In theory, we could only check for remaining cycles, although we probably would have computed data reduction rules in practice as these would quickly remove all vertices from cycle free graphs, making the test for a valid DFVS existing an emptiness check.

With this newly found solution, we go back within our branch to the previous not completely evaluated decision, in this case when we picked a vertex in 5.9c. Our bounds need to be updated again in 5.9l, since we found a better solution. We do not need to search for a worse solution than the current 13. As previously, our lower bound gets raised as well, as a smaller solution on this branch would have been reproducible on the other branch by selecting the same vertices.

We continue by finding an even better DFVS in this new branch, 5.9m, and fail to improve it when undoing the decision. In this example, excluding the vertex from the solution did not lead to the direct inclusion of other vertices, the lower bound however turned out to be too high to be able to improve the current solution any more, 5.9o.

We did not cover data reduction rules adding more vertices than even the maximum. We could alter our reduction rules to also keep track of the bounds as well, employ k-based data reduction rules and immediately stop when exceeding the bounds. In this example, we have seen the importance of propagating bound information upwards, since the bounds of closely related branches are quite often closely related to each other.

5.7. Combining the approaches

Our solving approach is summarized in Figure 5.10. We target 30 minutes as the available time for the PACE Challenge.

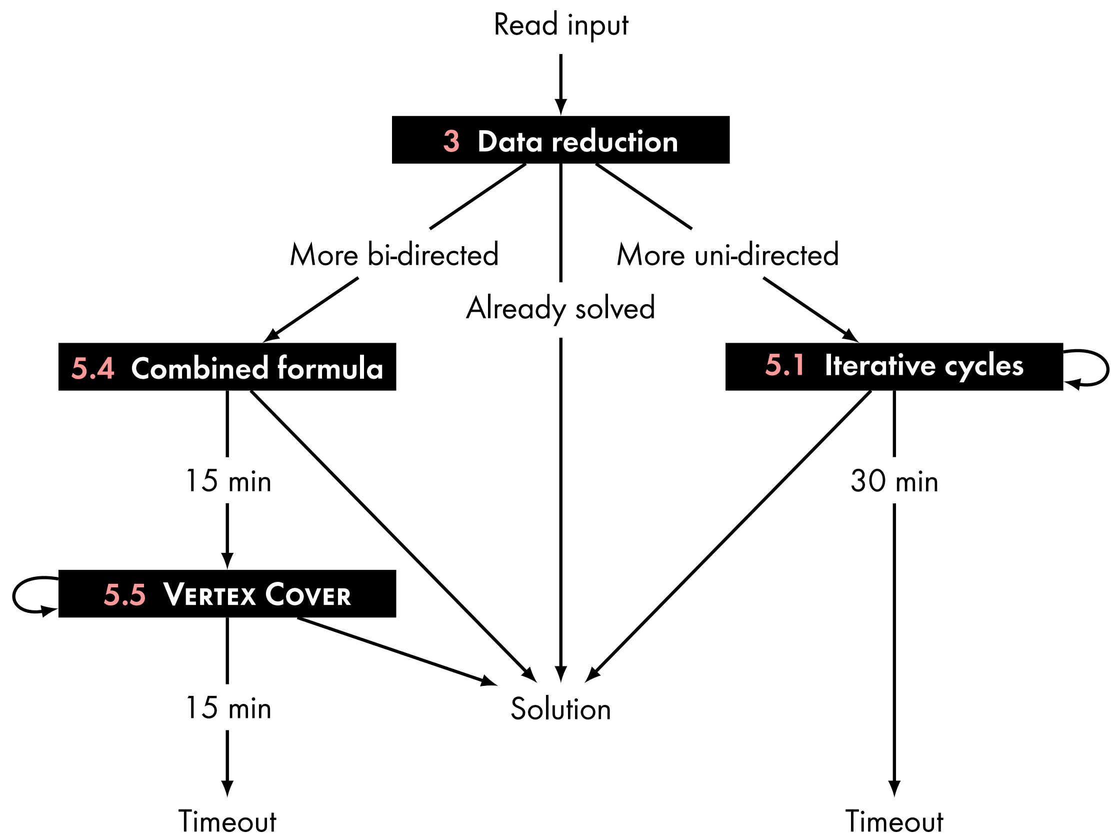


Figure 5.10.: Overview of solving

The first step in solving a given DFVS instance was always to apply the data reduction rules from Chapter 3. We were able to demonstrate in Section 4.3 that these also come with theoretical guarantees with regard to the size of the underlying Feedback Vertex Set. During reduction, we might have split our problem instances into multiple parts which we then solve individually.

We continue with the most effective solving approaches explored in Sections 5.1, 5.4 and 5.5 which are all based on reductions to other problems for which efficient solvers exist. We use timeouts to switch from our combined formulation to Vertex Cover based solving after having spent half of the available time.

We then reconstruct the minimum DFVS from the solution of these problems and apply changes made during reduction in their reverse order.

> This has been implemented in `exact.CombinedDfvsSolver`.

6. Practical Evaluation

We evaluate the practical implementation of the reduction rules and solving techniques introduced before. In some cases, we already mention comparisons with *DAGer*, the winning solver of PACE 2022, which is explained in Section 6.4.1.

6.1. Dataset overview

The evaluation is performed on the dataset that was used in the PACE challenge.

There were four subsets, each contains one hundred instances:

- Public exact instances
- Private exact instances
- Public heuristic instances
- Private heuristic instances

The public instances were available throughout the challenge, the scoring was performed on the instances of the respective track. For the exact track, the results on both public and private instances were combined, for the heuristic track, scoring was performed only on the private instances.

The exact track did not contain instances from real-world datasets (Großmann et al., 2022). Instead, its instances were generated using a variety of methods (Funke et al., 2018).

- **Erdős-Rényi Graphs**
  Given n, m, these are selected from a uniform distribution over all possible graphs of n vertices and m edges.
- **Random Geometric Graphs**
  While there are defined for higher dimensions as well, only $d \in \{2, 3\}$ were used. For $d = 2$, all vertices are placed on a square. This square is then split into overlapping chunks in which vertices are connected with their neighbors. For $d = 3$ the same approach is applied in three dimensions on a cube.

- **Random Hyperbolic Graphs**
  Vertices are placed on a disk in the hyperbolic plane. Edges are created between edges with a similar degree using a radius based approach similar to the geometric graphs.
- **Random Delaunay Graphs**
  These graphs are generated from random points in $d \in \{2, 3\}$-dimensional space which are then connected such that its faces become smallest possible triangles, or tetrahedrons respectively.
- **Barabási-Albert Graph Model**
  Vertices are connected to vertices that are already more closely connected, which is called *preferential attachment*. As this model by default creates acyclic graphs, Großmann et al. added additional random edges to create cycles.

The organizers preselected instances that were not trivial to solve. All instances were provided in the METIS format[1] which is based on adjacency lists.

We mostly consider them from a practical point of view. Our focus was on the exact track, so we will only consider the exact instances unless otherwise noted.

The dataset can be split into five main categories. Most contained additional edges not on cycles that as such were removed by the most basic reduction rules. The instances were therefore categorized after reduction rules were applied.

- **Small instances**
  Around half of the instances are relatively small, the others are larger and more complex. Most of the small instances were already solved by data reduction alone, or in a way that only very few vertices remained. Therefore, it did not make sense to analyze these further.
- **Vertex Cover instances**
  Sometimes called undirected instances, all edges within these are bi-directed.
- **Mostly undirected instances**
  The largest part of edges is structured as in the previous vertex cover instance case. However, a small directed subgraph remains.
- **Directed instances**
  These instances mostly consist of very long cycles with bi-directed edges rarely occurring
- **Huge instances**
  There are three instances that resemble the previous three categories but are much larger. We will exclude them and two additional instances that were not solved by any solver for example when discussing solution sizes of instances.

[1] METIS GRAPH Files, John Burkardt, Department of Scientific Computing, Florida State University
`https://people.sc.fsu.edu/~jburkardt/data/metis_graph/metis_graph.html`

The definition of a Vertex Cover instance is not a difficult choice. To differentiate between directed and mostly undirected instances, we draw a line at the absolute count of edges containing a majority of bi-directed or uni-directed edges, counting bi-directed edges as two vertices. We give a detailed overview on the number of vertices, edges and solution sizes and the resulting classification in Table A.1 on page 140.

6.2. Evaluation of reduction rules

We examine the efficiency of the data reduction rules at the point where they are usually applied. A very general reduction rule with a large running time requirement will therefore be applied after its faster special case rules have been applied exhaustively. We have implemented the reduction rules in Figure 6.1.

For each reduction rule, we will have applied all previous data reduction rules exhaustively. We will then apply this reduction rule and previous rules until both the current and previous rules have been applied exhaustively. A reduction rule that does not remove a lot of vertices itself, but allows other rules to be applied further a lot will be attributed all improvements of the previous rules.

In Figure 6.2a, we display how many vertices of the solution are directly added by our reduction rules. This mostly corresponds with the removed vertices in Figure 6.2b, as every vertex added to the solution is also removed from the graph. As a very first step, we performed a single application of Reduction rule 4, which did not add vertices for the solution. About half of the instances were directly solved by our reduction rules. Only on instances where these did not already solve large parts of the instance, our more complex rules were able to make small contributions. We can immediately see the effect of our edge based rules in Figure 6.2c.

The time finished in Figure 6.3 indicates the time after which all rules including the current one had been applied exhaustively for the first time. The initial removal of vertices with Reduction rule 4 was expensive on instances with very large numbers of vertices, although it otherwise completed very quickly. We were able to find only a very small number of small crowns for our partial implementation of Reduction rule 10 after having applied all other reduction rules. However, we decided to keep them since they did not take long to compute either and might be helpful on instances suitably structured.

The data reduction rules completed on all instances, both exact and heuristic within reasonable time. We give detailed results for running times on individual instances as well as graph and solution sizes before and after reduction in Table A.1.

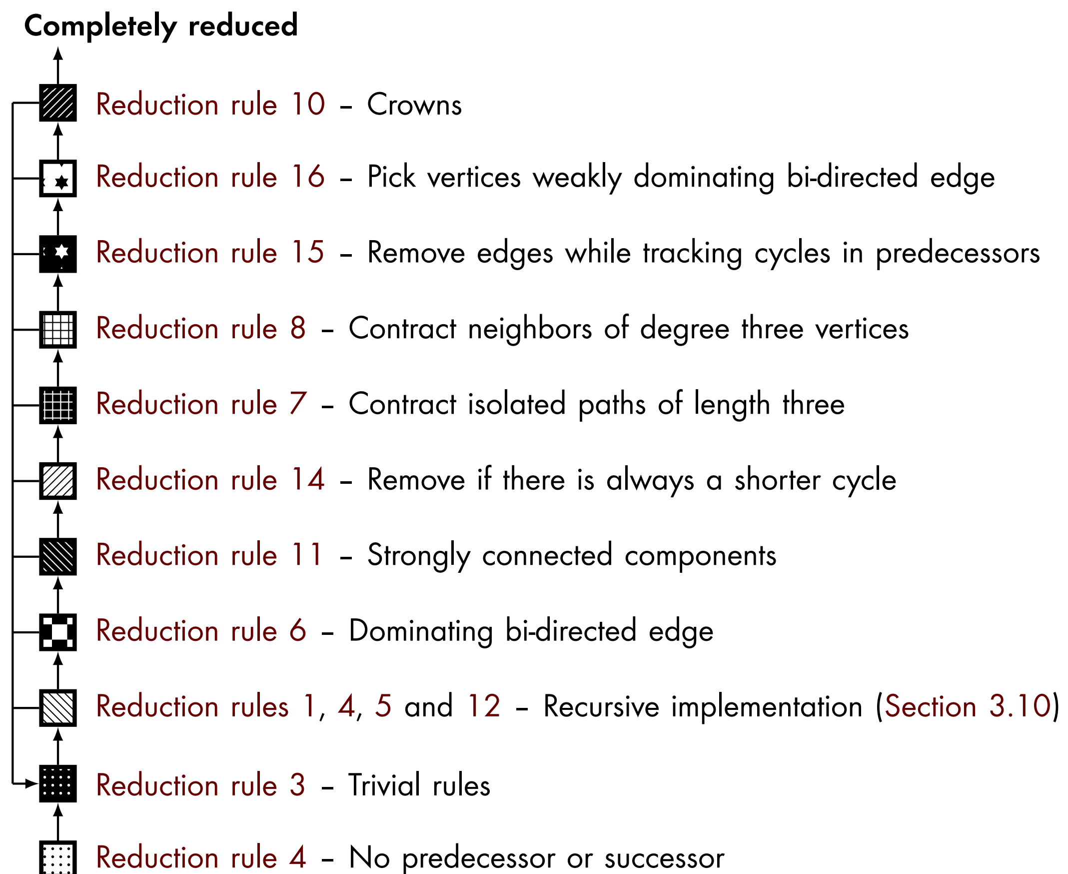
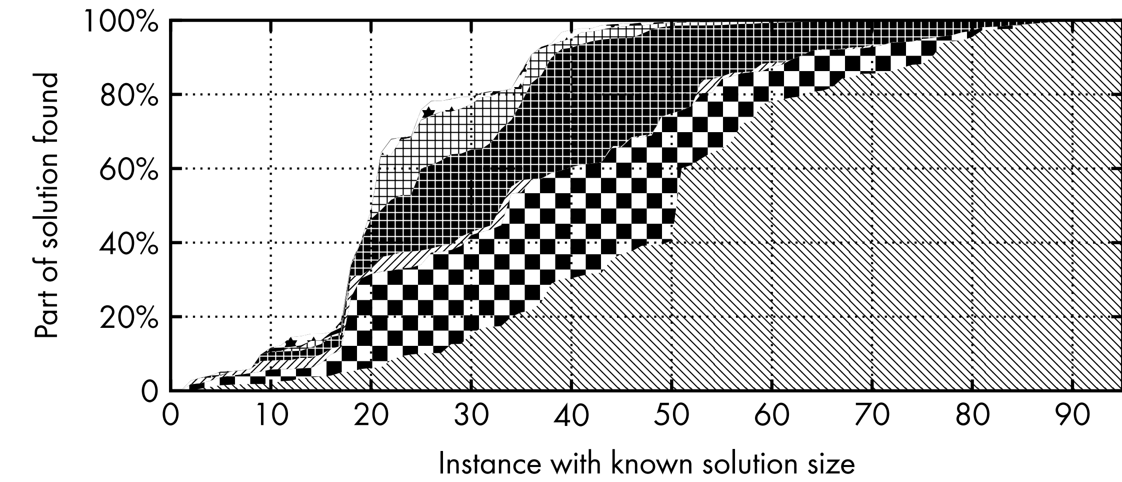
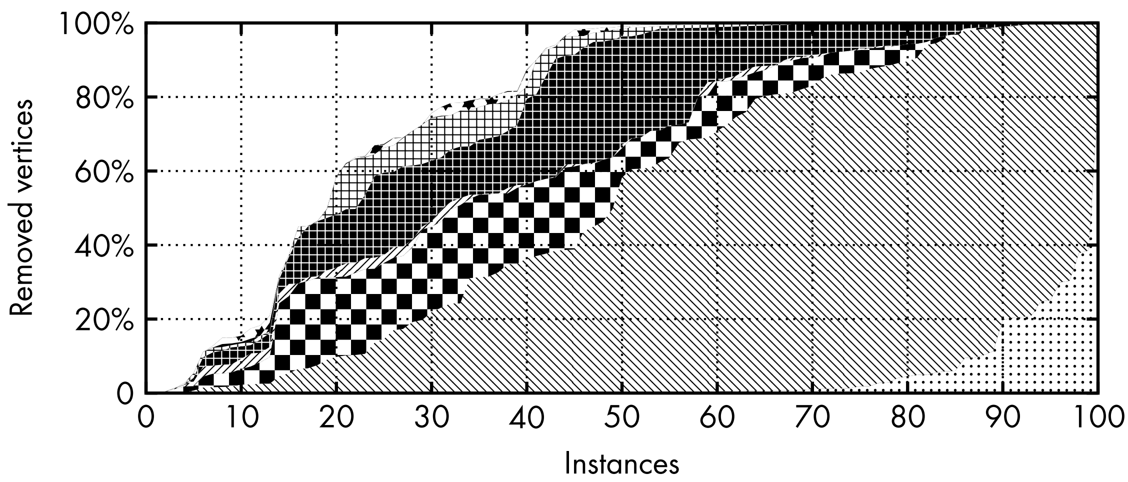


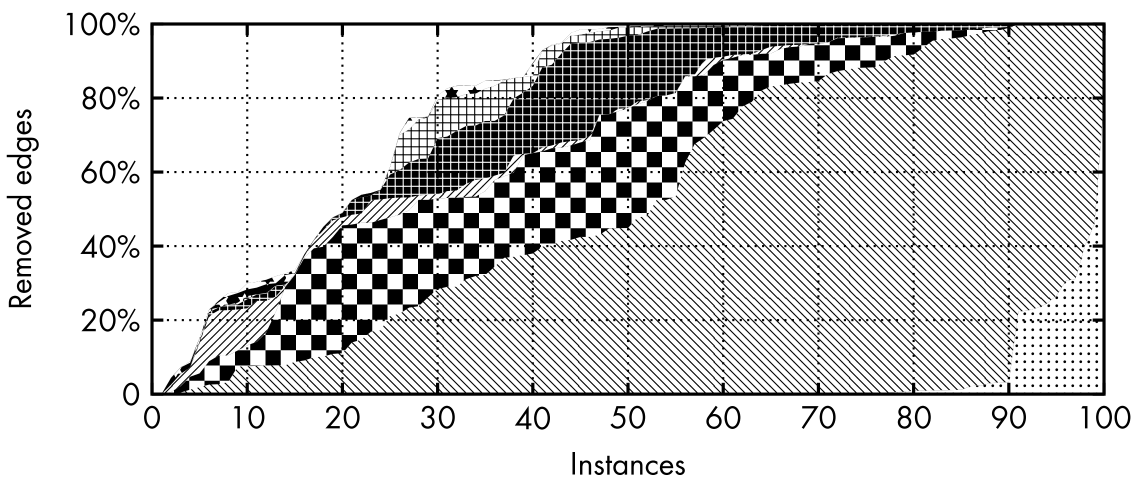
Figure 6.1.: Implemented reduction rules



(a) Part of vertices directly added to the solution



(b) Vertices removed



(c) Edges removed

Figure 6.2.: Effects of applying reduction rules

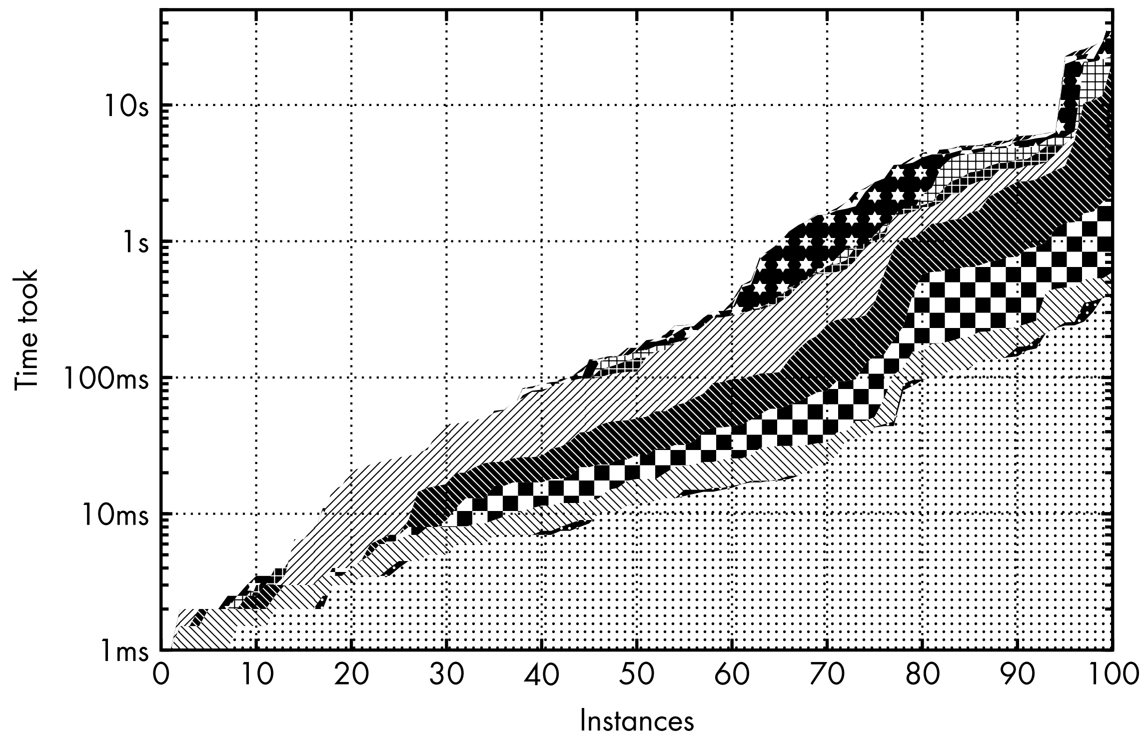


Figure 6.3.: Reduction time

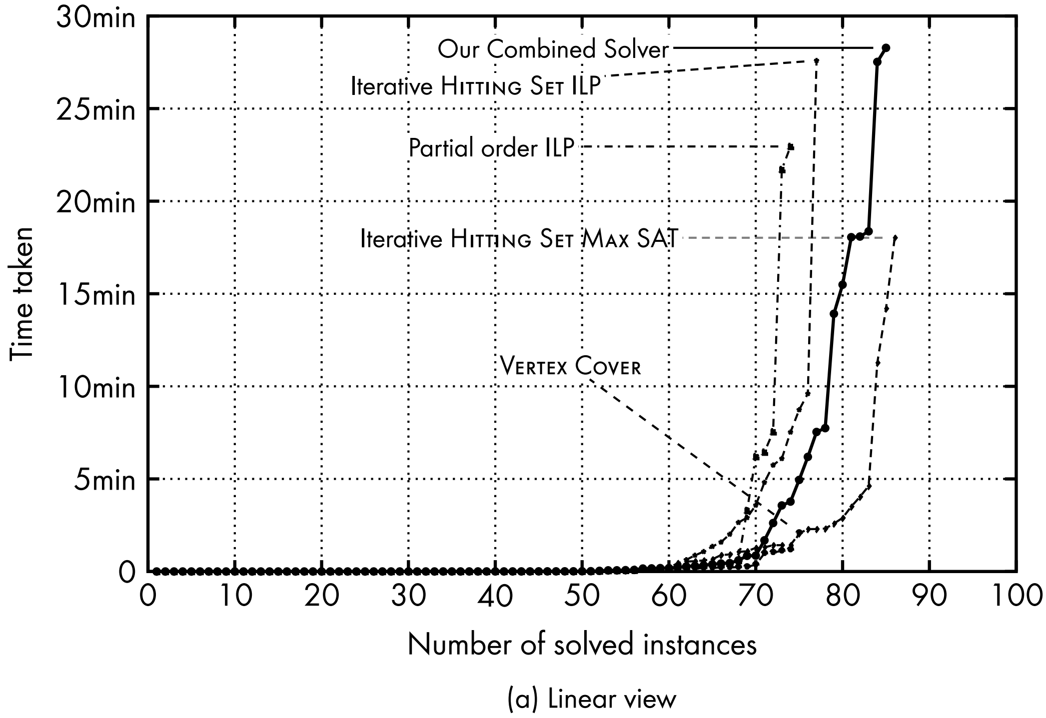
6.3. Evaluation of solving techniques

We compare the different solving techniques we implemented with each other. An overview is shown in Figure 6.4. The more instances solved, the better.

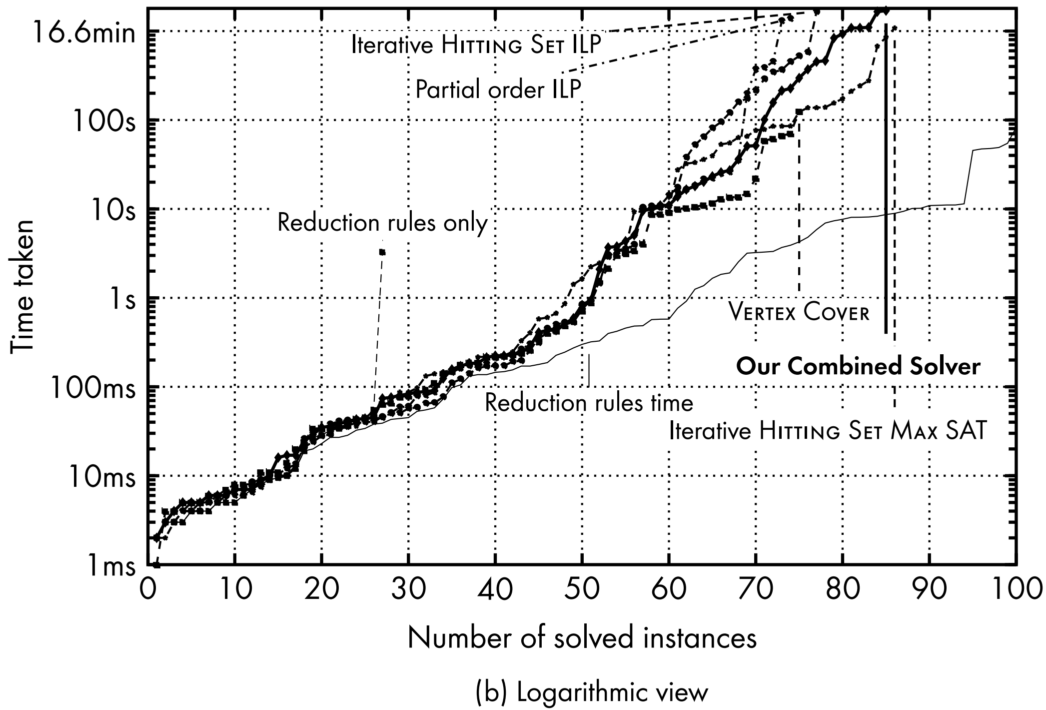
We can observe the following:

1. The iterative Hitting Set solved with ILP is fairly slow, but solved more instances when compared with induced partial order ILPs in the long run.
2. Vertex Cover based solving either completed quickly and generally faster than other solvers within two minutes or not at all.
3. Our combined solver that we submitted to PACE is outperformed by solving iterative Hitting Set with Max SAT, using *EvalMaxSAT* internally.
4. A few instances are already solved by reduction rules and we include the completion of reduction rules on the instances as a hint.

This however hides that some techniques not completing on individual instances that others can solve. Table 6.1 illustrates this. It differentiates for the combinations of solvers that were able to solve an instance in general and then counts how often the respective solver achieved the



(a) Linear view



(b) Logarithmic view

Figure 6.4.: Comparison of our own solving approaches

Count	Iterative Hitting Set			Partial order
	ILP	Max SAT	Vertex Cover	ILP
41	20	1	12	8
7		7		
4	4			
3		3		0
3		0	3	
3			3	
2	0	2		0
2	2	0		
1		0	1	0
1†	1			
1††				
27^R				
Solved	77	86	75	74

† This instance was not solved by *DAGer*
†† This instance was only solved by *DAGer*
R Completely solved by reduction rules

Table 6.1.: Fastest completion of solving techniques on instances that were solved

best result. For example on the two instances that were solvable by iterative Hitting Set with ILP, iterative Hitting Set with Max SAT and induced partial order ILP reductions, Max SAT was always faster than the other two.

Detailed results on the fifty larger instances can be found in Table A.2 on page 141. If we group the instances according to our classification from Section 6.1, as in Table A.4 on page 143, we are able to observe some patterns.

1. The iterative Hitting Set ILP is our only solver effectively solving directed instances, in two cases taking about the same amount of time as *DAGer* even being able to solve an instance that *DAGer* does not return a result on.
2. The Vertex Cover solver unsurprisingly performs well on Vertex Cover instances. However, the Max SAT solver is able to complete on instances that the Vertex Cover solver does not solve.
3. The same appears for mostly directed instances, however in the reverse direction. Here, the Vertex Cover usually completes faster as well, if it returns a result at all.
4. The induced partial order ILP formulation that we used for our PACE submission is outperformed by the iterative Max SAT formulation on almost all instances In contrast to Max SAT based solving, it is not able to solve any additional instances.

Table 6.2.: Overview of PACE 2022 exact solver submissions solving 150 instances

#	Name, Paper	Approach	Solved
1	*DAGer* Kiesel and Schidler, 2023	Using a dynamic MAXSAT solver	185
2	Our submission† Bergenthal et al., 2022	Depending on instance: Iterative reduction to ILP, direct reduction to ILP or partially iterative reduction to VERTEX COVER	165
3	*Mount Doom*† Angrick et al., 2023	Iterative reduction to VERTEX COVER	152
4	*G^2OAT* Červený et al., 2022	Branch and reduce	151
D	*DVFS* Meiburg, 2022	If possible direct reduction to HITTING SET, using a dynamic ILP solver	175
D	*DiVerSeS*	Depending on instance: Branch and reduce or iterative reduction to HITTING SET	160

D Disqualified because of errors but resubmitted † Student submission

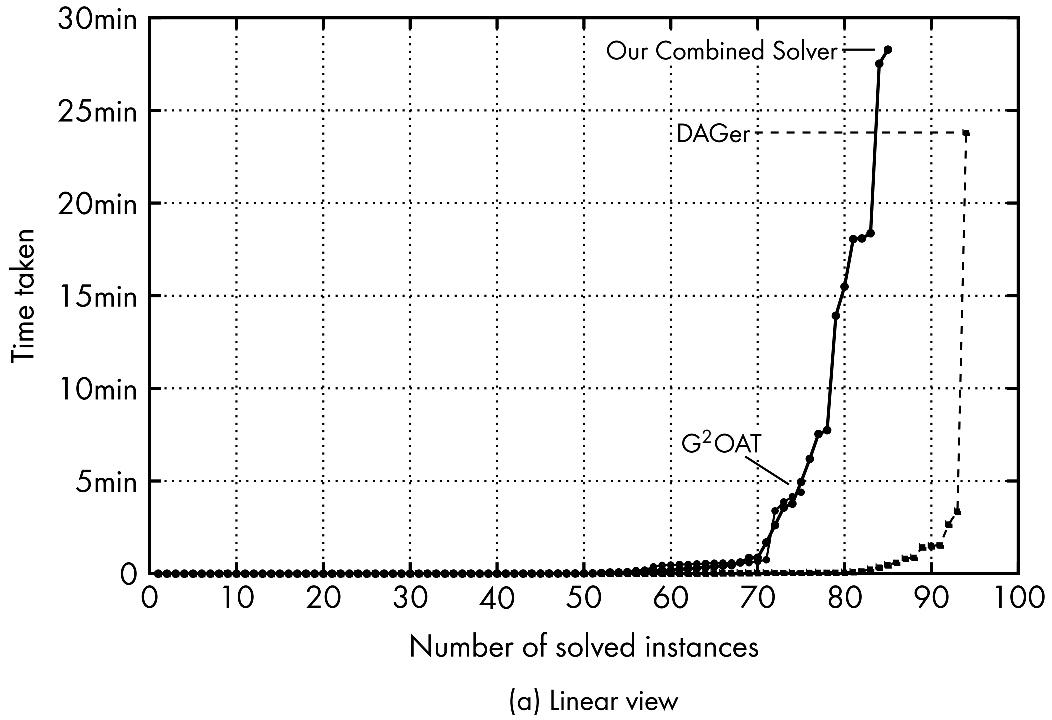
6.4. Comparison with other PACE submissions

We briefly discuss the approaches employed by other solvers for the PACE 2022 exact track. For an overview of the solving approaches, see Table 6.2. This overview is insofar interesting, as the teams used a large variety and a lot of different approaches in their submissions. In most cases, we attempted to follow similar approaches and already gave an overview on them in Chapters 3 and 5. We only considered solvers that solved 150 instances in total on public and private instances as determined by Großmann et al. (2022).

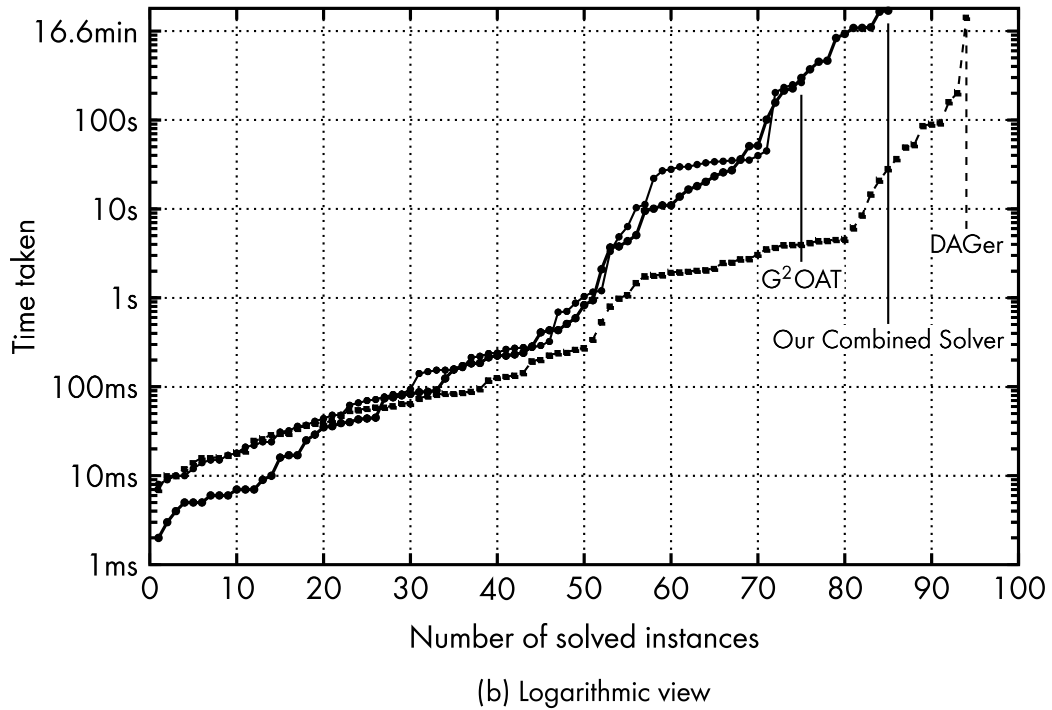
We compare their running time on instances in Figure 6.5. While only the count of solved instances mattered for the PACE challenge apart from ties, taking less time is preferable. .

In these comparisons, our solvers had a slight advantage since we took all measures within our application, eliminating our typical overhead of starting the Java Virtual Machine while imposing a minor delay by calling the other solvers from within our application. The results would, however closely resemble typical scenarios where DFVS needs to be solved within some Java based environment.

> This has been implemented in `performance.PerformanceMeasurement`.



(a) Linear view



(b) Logarithmic view

Figure 6.5.: Comparison of solvers

6.4.1. DAGer

The winning solver of PACE 2022 was *DAGer*, created by Kiesel and Schidler (2022, 2023)[2].

It used the *EvalMaxSAT* solver for solving Max SAT, Problem 7. They added cycle constraints dynamically during solving and thereby followed an iterative approach similar to Section 5.1, however without having to call the internal solver again on each run. It could thus retain its learned constraints for CDCL.

They only applied the basic and well known reduction rules and immediately proceeded with interactive solving.

6.4.2. Mount Doom

The idea behind the solver *Mount Doom*[3] by Angrick et al. (2022) was that DFVS is very similar to Vertex Cover, Problem 2 on instances that mostly contain bi-directed edges (Angrick et al., 2023). They used an iterative reduction in a similar way as we did in Section 5.1, however instead of solving the Hitting Set instances via ILPs as in our case, they applied the same reduction to Vertex Cover that we described in Section 5.5.

Internally, they used *WeGotYouCovered* to solve Vertex Cover on the iteratively expanded undirected subgraphs while applying data reduction rules after each addition of gadgets.

6.4.3. G^2OAT

G^2OAT[4] was the only of the more successful solvers directly implementing a branching algorithm as explained in Section 5.6 (Červený et al., 2022). They heavily rely on data reduction rules. For preprocessing, they would also use more expensive rules and turn them off during branching.

As a lower bound, they first greedily remove maximal bi-directed cliques, then compute a lower bound for Vertex Cover on the undirected subgraph using Linear Programming. They then remove all vertices incident to bi-directed edges, apply reduction rules on the remaining graph which is a subgraph of the directed subgraph. When only directed edges remain, they pack disjoint cycles.

[2] Dagger, André Schidler, GitHub `https://github.com/ASchidler/dfvs`
[3] Mount Doom, Ben Bals, GitHub `https://github.com/BenBals/mount-doom/tree/exact`
[4] G^2OAT, Prague Technical University GitLab
`https://gitlab.fit.cvut.cz/pace-challenge/2022/goat/exact/`

6.4.4. DVFS

DVFS[5] relied entirely on a Hitting Set formulation as we described in Section 5.1 (Meiburg, 2022). They only applied the most basic Reduction rules 4, 5 and 11 and quickly proceeded to finding an initial set of induced cycles. Whenever possible, they would try to enumerate all induced cycles. The set of cycles would then be used as a Hitting Set and, if not all cycles were enumerated, interactively increased using *SCIP* as an interactive ILP solver.

6.4.5. DiVerSeS

DiVerSeS[6] decided on using branch-and-reduce based solving or iterative Hitting Set based solving using *FindMinHS*[7] (Swat, 2022b) depending on the instance. Its heuristic variant (Swat, 2022a) was the winner on the heuristic track.

6.5. Creating an improved solver

Before the results of the PACE challenge were published, we were not aware of *EvalMaxSAT*, although we evaluated using SAT based solving using an SMT solver. Using the simple reduction explained in Section 2.3.5 on page 31, we were able to directly use it within our iterative solver replacing the ILP solver. We already incorporated this alternative solving approach in our observations in Section 6.3.

Consequently, we are able to build an improved solver. We keep the iterative ILP for directed instances, where it had vastly better performance than Max SAT. Otherwise we first perform Vertex Cover based solving before we resort to iterative Max SAT, as it proved more stable and there would be hope for it completing on instances that are structurally unsolvable for the Vertex Cover solver. We use a slightly larger time limit on Vertex Cover instances. Furthermore, we improved our Vertex Cover based solving approach to also reduce arbitrarily large cycles, even though this was never necessary on our dataset. The updated approach is shown in Figure 6.6.

We can observe the result in Figure 6.7. When comparing this with our submitted solver, we solve nine additional instances. Our new solver is still outperformed by *DAGer* on almost all instances. We however were able to close the gap on a couple of instances and were even able to finish computing on an instance that *DAGer* did not solve on our setup within the time limit.

[5] DVFS_PACE2022, Timeroot, GitHub `https://github.com/Timeroot/DVFS_PACE2022/`
[6] pace-2022, Sylwester Swat , GitHub `https://github.com/swacisko/pace-2022`
[7] David Stangl, findminhs, GitHub `https://github.com/Felerius/findminhs/`

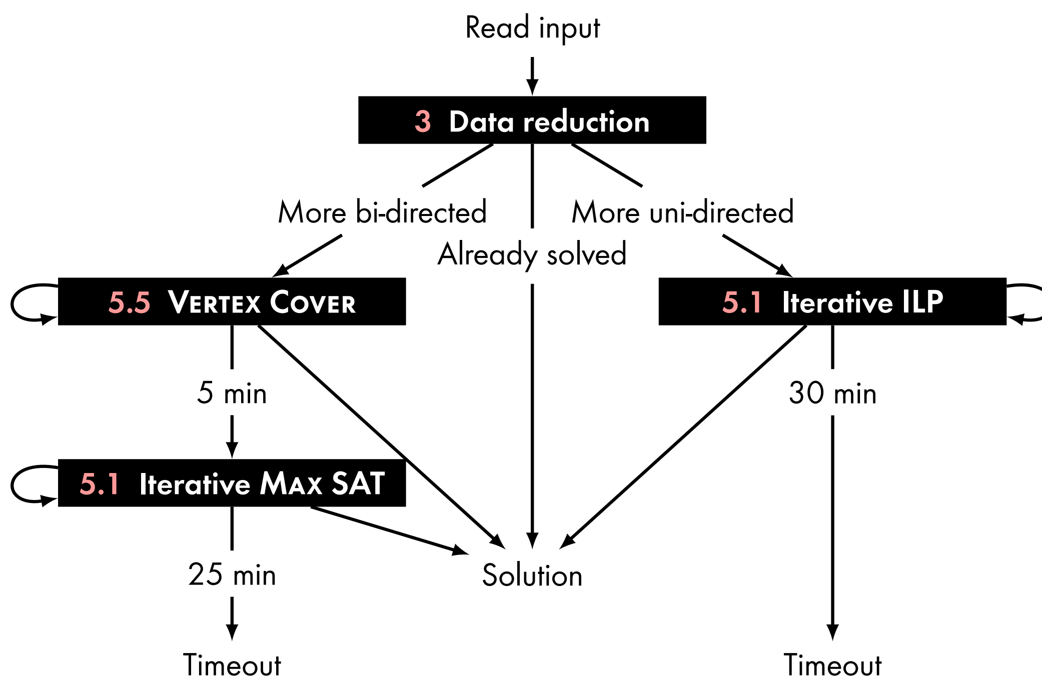
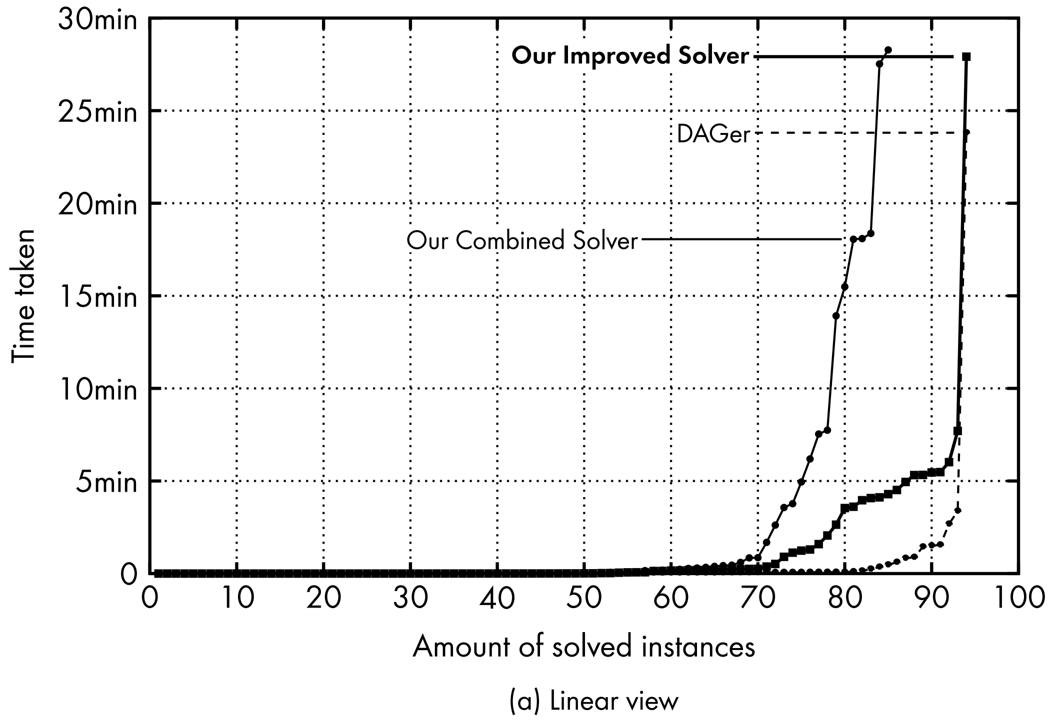
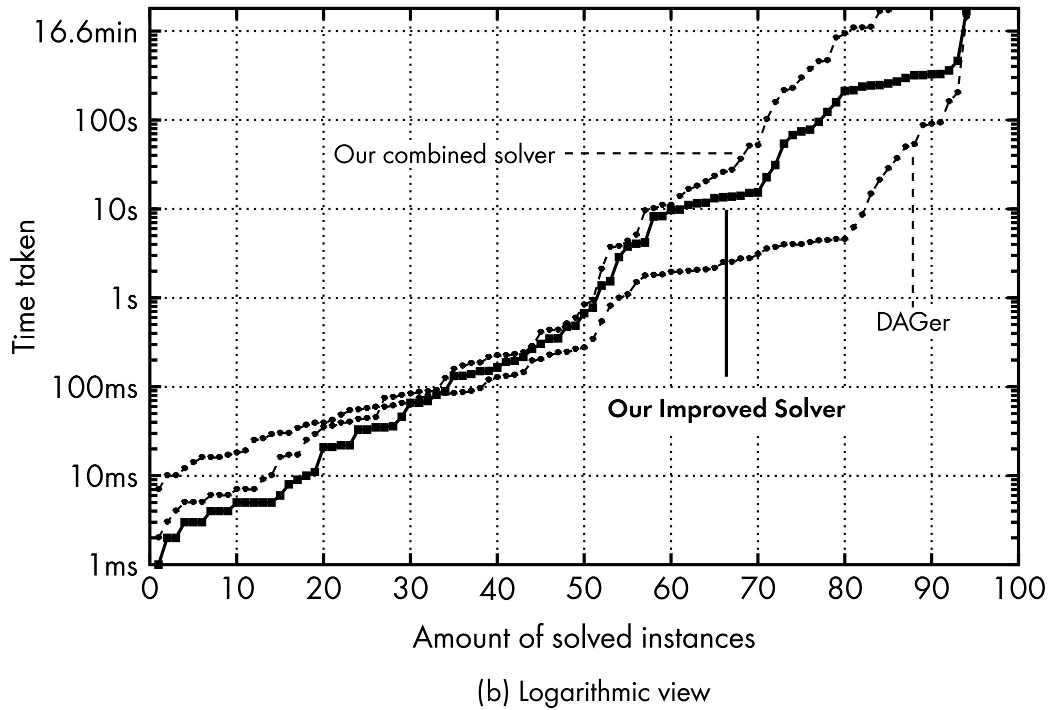


Figure 6.6.: Overview of our improved solving approach

This has been implemented in `exact.ImprovedDfvsSolver`.



(a) Linear view



(b) Logarithmic view

Figure 6.7.: Comparison of our improved solving approaches and DAGer

7. Conclusion

We have presented several approaches to solve DFVS in theory and practice. We will first give an overview of our most interesting contributions and important observations.

After that, we discuss how research on DFVS and very similar related problems could be developed further. Parts of our observations can also be translated to the practical solving of other **NP**-hard problems. We furthermore review the research process itself and possible future applications of DFVS.

7.1. Summary

We were able to reach the main objectives of this thesis. We furthermore made several interesting observations that we also presented.

We have presented an overview of the existing data reduction rules. For each rule, we presented basic information on its running time and its effect on the graph, the origin of the rule and how it can be applied.

We repeated the important result of Lin and Jou (2000), that only induced cycles are relevant for solving. Based on this result, we created Reduction rules 14 and 15 that each solve the **NP**-complete inverse Edge on Induced Cycle problem heuristically to be able to remove edges. We have obtained both a rule that can be computed quickly and another that is more effective at the expense of slightly more, though still polynomial, running time.

We furthermore presented Reduction rules 17 and 18 that generalize the domination based rules that have been known for a long time and are directed adaptations of reduction rules for Vertex Cover from bi-directed edges to longer cycles. When incorporating our results from the searches we performed for the Edge on Induced Cycle problem, we are able to further generalize the cycle domination based rules. We present this generalization for bi-directed edges in Reduction rule 16 on page 64.

Regarding steps towards a polynomial kernel for DFVS, we demonstrated that Reduction rule 14 on page 60 allows us to subsume the more complicated rules of the kernel of Bergougnoux et al. (2021) without the need to supply a feedback vertex set of our instance. At the same time, we were able to show that this is not sufficient as a kernel for general DFVS.

We have implemented the most promising data reduction rules and made this implementation open source. We were able to verify their effect and gave an overview on their performance on different types of instances. Future implementations could use this as the basis for selecting fitting reduction rules for the instances that need to be solved.

Regarding the practical approaches to solving reduced instances, iteratively solving Hitting Set proved most effective. Our linear reduction to Integer Linear Program however had a good trade off with regard to its overall stability and performance and presented an approach that to our knowledge was unique among the challenge participants.

We were able to improve the reduction from Hitting Set to Vertex Cover with improved gadgets that we have discovered. These allow us to use a single gadget where we would have needed multiple. This therefore substantially reduces the number of created vertices and edges. Since it completed very quickly on a large number of instances that mostly resemble a Vertex Cover problem, we used a Vertex Cover as the primary solver for the respective instances.

On graphs with mostly longer cycles, our existing strategy of an iterative reduction to Hitting Set and further reducing this to Integer Linear Program remained successful. Notably, this solving strategy completed on an instance that *DAGer* did not solve within the time limit.

We were able to modify our solver to be able to compete with the winning submission of the PACE challenge, *DAGer*, by simply adapting their internal Max SAT solver. Apart from the specific reduction itself, we were able to completely reuse our existing iterative implementation that was initially created for ILP.

Overall, our new combined solver now completes on as many instances as *DAGer* within the time limit, although it is slower on most of them. Within our own solver, there was a major difference between the performance of the ILP solver when compared to the Max SAT solver. On directed instances, the reduction to ILP was better than Max SAT. At the same time, we did not even deeply integrate into the external solver to introduce new constraints interactively during the ongoing computation. We could adapt our technique of choosing the reduction based on the instance and reducing to ILP as well, also relying on constraints interactively created during the runtime of the external solver. It is very plausible that the performance of *DAGer* could be further improved in this way.

7.2. Further research on Directed Feedback Vertex Set

The open question of a polynomial kernel for DFVS remains. We provided a small step towards the possible discovery of such a kernel with our new reduction rule.

We highlighted the importance of induced cycles for solving the problem. Future approaches might extend upon our search for the absence of such cycles, although we already showed that eliminating all edges not on an induced cycle would not be sufficient, as we demonstrated in our example for our kernel in Section 4.3 on page 84.

A practical though not resource efficient way to compute results would be parallelization. The solvers for Max SAT and ILP that we used to solve the instances we reduced to already support parallel execution. This would be a natural starting point and, since most of the computation time was spent on the solving of the problem instead of the data reduction rules, as evident from Figure 6.4 on page 117, it would apply to the largest part of the computation. Furthermore, applying a few branching-and-reduce steps could easily be parallelized and might further be improved by communication between branches solved in parallel. While data reduction did take long in practice, several of our rules would admit efficient parallelization. All the local rules can be applied in parallel when ensuring that they are applied disjoint (see Kreowski, Kuske, et al., 2008). The same applies to our search for edges not on induced cycles, as the set of induced cycles does not change during deletion.

Vertex-weighted versions of DFVS may be more suitable for practical applications. Fortunately, large parts of the data reduction rules can be adapted to respect weights, though re-implementing might require more work for a couple of rules. Edges not on induced cycles as eliminated by our new Reduction rule 14 on page 60 keep being irrelevant, the rules can thus be used as previously. Domination rules need to be modified to only include a higher-weight vertex. All the solving approaches that we explained in Chapter 5 on page 85 can be directly adapted for weighted DFVS. For ILP and SAT reductions, we associate the constraint with the weight as a penalty of selecting it. Branching remains completely unchanged, although we might want to consider new branching strategies accounting for weights. In the worst case, if weights can be represented with ideally small and possibly scaled integers, we could split vertices and create the respective number of siblings with copies of their predecessors and successors.

It would furthermore be interesting to look into the practical solving of Subset Directed Feedback Vertex Set, which can be seen as a generalization of DFVS (Chitnis et al., 2012). We try to find a subset $S \subseteq W$ of vertices of a subset $W \subseteq V(G)$, such that all cycles passing through W are covered. If $W = V(G)$, this becomes DFVS. Several rules and solving techniques can be directly adapted. For example, Reduction rules 4 and 5 do not close any cycles and could be applied on the whole graph. Several rules for vertices on the edge of W, that are connected to vertices in $V(G) \setminus W$ could be introduced to directly include them if they are the only vertex

in W that lies on a cycle. We would need to modify our searches for example for Reduction rule 15 not to stop when having found cycles completely within $V(G) \setminus W$, but could reuse our arguments if any vertex of such cycles is contained in W.

One of the most effective additions to current solvers would be a support of iteratively solving DFVS. We could allow the solver to keep its current internal state and allow the later addition of edges and/or vertices with new edges. This would be comparable to the lazy addition of constraints to the MAX SAT solver as used by *DAGer*. Alternatively, we could merely provide the solver with an instance and a minimum DFVS on a large, possibly induced, subgraph of it. This could be supported by existing FPT algorithms using iterative compression such as the algorithm of Chen et al. (2008). When adding vertices, we could directly add them into the solution and attempt performing iterative compression on this solution of size $k + 1$. Adding edges would be possible while including either of their vertices if their addition would introduce a cycle that is not covered by the solution. Both of these additions would be great opportunities for future PACE challenges, and could for example constitute an additional track.

7.3. Solving other NP-hard problems

Large parts of our methods and methodology may be transferred to solving other problems. We will first discuss solving approaches and then comment on the research process.

Applying reduction rules first has already been common practice. However, we see a tendency, given the reduced instance, to perform a reduction to other problems for which highly optimized solvers exist. This was the approach used by most other successful solvers within the PACE challenge. Furthermore, many of the used solvers in turn relied on this approach as well, creating a chain down towards the most basic and at the same time universal problems such as SAT.

Lazy constraint evaluation combined with a good selection of an internal solver gave *DAGer* the edge in the PACE challenge. Otherwise, our practice of selecting the specific solving approach based on instance parameters was of great practical use, especially when instances have structures that closely map to another problem as in our case VERTEX COVER.

From the methodological point of view, we have been fairly successful in using an easy-to use and rather high-level programming language. We were programming in a very flexible environment, were able to rapidly prototype our ideas and were able to implement them easily. Our development style was further supported by the fact that solving the problem was arguably easiest to perform via reduction. We were able to externalize the "heavy lifting" to a highly optimized native implementation and reduced the impact of our implementation not being fully optimized.

We furthermore hugely benefited from automated testing. Several teams slipped up in some minor detail and were disqualified. We essentially performed a fuzzing by computing the minimum solution with different combinations of enabled data reduction rules and several different solving approaches that were implemented largely independently. Using this approach, we were able to quickly identify subtle errors in our implementation that would only occur in very rare edge cases. Finally, we performed mutation based testing[1] to assess which parts of our code were actually tested and paid greater attention on sections that were not covered by tests.

7.4. Future practical applications

Several practical applications on graphs may benefit from computing a minimum DFVS and then performing further computations, for example using dynamic programming on the remaining directed acyclic graph. Vertices in the DFVS would be handled separately. This would both follow the idea of Kuosmanen et al. (2018) and could use their approach of transferring dynamic programming from sequences to directed acyclic graphs using a Path Cover.

As such, DFVS might become much more relevant when even larger parts of our economy, for example transportation become digitized and thus centrally manageable. For much frequented systems, the gains of exact algorithms versus approximation or heuristics based approaches may outweigh the additional computational costs and resource footprint of exact computation.

[1] Using PIT, `https://pitest.org/`, which provides great support for mutation based testing in the Java ecosystem, since it does not have to rely on compiled code which would be far harder to perform mutations on.

8. Bibliography

Abu-Khzam, Faisal N. (2007). "Kernelization Algorithms for d-Hitting Set Problems". In: *Algorithms and Data Structures*. Ed. by Frank Dehne, Jörg-Rüdiger Sack, and Norbert Zeh. Berlin, Heidelberg: Springer, pp. 434–445. ISBN: 978-3-540-73951-7. DOI: 10.1007/978-3-540-73951-7_38 (cit. on pp. 53, 72).

Abu-Khzam, Faisal N., Michael R. Fellows, Michael A. Langston, and W. Henry Suters (2007). "Crown Structures for Vertex Cover Kernelization". In: *Theory of Computing Systems* 41.3, pp. 411–430. ISSN: 1432-4350, 1433-0490. DOI: 10.1007/s00224-007-1328-0 (cit. on pp. 51, 53).

Achterberg, Tobias, Timo Berthold, Thorsten Koch, and Kati Wolter (2008). "Constraint Integer Programming: A New Approach to Integrate CP and MIP". In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Ed. by Laurent Perron and Michael A. Trick. Berlin, Heidelberg: Springer, pp. 6–20. ISBN: 978-3-540-68155-7. DOI: 10.1007/978-3-540-68155-7_4 (cit. on p. 35).

Angrick, Sebastian, Ben Bals, Katrin Casel, Sarel Cohen, Tobias Friedrich, Niko Hastrich, Theresa Hradilak, Davis Issac, Otto Kißig, Jonas Schmidt, and Leo Wendt (2022). "PACE Solver Description: Mount Doom - An Exact Solver for Directed Feedback Vertex Set". In: *17th International Symposium on Parameterized and Exact Computation (IPEC 2022)*. Ed. by Holger Dell and Jesper Nederlof. Vol. 249. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 28:1–28:4. ISBN: 978-3-959-77260-0. DOI: 10.4230/LIPIcs.IPEC.2022.28 (cit. on p. 121).

– (2023). "Solving Directed Feedback Vertex Set by Iterative Reduction to Vertex Cover". In: *21st International Symposium on Experimental Algorithms (SEA 2023)*. Ed. by Loukas Georgiadis. Vol. 265. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 10:1–10:14. ISBN: 978-3-959-77279-2. DOI: 10.4230/LIPIcs.SEA.2023.10 (cit. on pp. 119, 121).

Ausiello, Giorgio, Alessandro D. D'Atri, and Marco Protasi (1980). "Structure preserving reductions among convex optimization problems". In: *Journal of Computer and System Sciences* 21.1, pp. 136–153. ISSN: 0022-0000. DOI: 10.1016/0022-0000(80)90046-X (cit. on p. 29).

Avellaneda, Florent (2020). *A short description of the solver EvalMaxSAT*. Ed. by Fahiem Bacchus, Jeremias Berg, Matti Järvisalo, and Ruben Martins. Helsinki: University of Helsinki, Department of Computer Science. URL: https://hdl.handle.net/10138/318451 (cit. on p. 33).

Bafna, Vineet, Piotr Berman, and Toshihiro Fujito (1999). "A 2-Approximation Algorithm for the Undirected Feedback Vertex Set Problem". In: *SIAM Journal on Discrete Mathematics* 12.3, pp. 289–297. ISSN: 0895-4801, 1095-7146. DOI: 10.1137/S0895480196305124 (cit. on pp. 73, 105).

Bang-Jensen, Jørgen and Gregory Z Gutin (2009). *Digraphs: theory, algorithms and applications*. Springer Monographs in Mathematics. London: Springer London. ISBN: 978-0-857-29041-0. DOI: 10.1007/978-1-84800-998-1 (cit. on p. 15).

Bang-Jensen, Jørgen, Alessandro Maddaloni, and Saket Saurabh (2016). "Algorithms and Kernels for Feedback Set Problems in Generalizations of Tournaments". In: *Algorithmica* 76.2, pp. 320–343. DOI: 10.1007/S00453-015-0038-2 (cit. on pp. 12, 72).

Barabási, Albert-László and Réka Albert (1999). "Emergence of Scaling in Random Networks". In: *Science* 286.5439, pp. 509–512. DOI: 10.1126/science.286.5439.509 (cit. on p. 12).

Barrett, Clark, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli (2021). "Satisfiability Modulo Theories". In: *Handbook of Satisfiability*. Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. 2nd ed. Amsterdam: IOS Press. Chap. 33, pp. 1267–1329. ISBN: 978-1-643-68160-3. DOI: 10.3233/FAIA201017 (cit. on p. 32).

Bentert, Matthias, Fedor V. Fomin, Petr A. Golovach, Tuukka Korhonen, William Lochet, Fahad Panolan, M. S. Ramanujan, Saket Saurabh, and Kirill Simonov (2024). *Packing Short Cycles*. arXiv: 2410.18878 (cit. on p. 87).

Berg, Jeremias, Matti Järvisalo, Ruben Martins, Andreas Niskanen, and Tobias Paxian (2024). "MaxSAT Evaluation 2024: Solver and Benchmark Descriptions". In: Department of Computer Science Series of Publications B. URL: https://hdl.handle.net/10138/584878 (cit. on p. 33).

Bergenthal, Moritz, Jona Dirks, Thorben Freese, Jakob Gahde, Enna Gerhard, Mario Grobler, and Sebastian Siebertz (2022). "PACE Solver Description: GraPA-JAVA". In: *17th International Symposium on Parameterized and Exact Computation 2022*. Ed. by Holger Dell and Jesper Nederlof. Dagstuhl: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 30:1–30:4. DOI: 10.4230/LIPICS.IPEC.2022.30 (cit. on pp. 12, 119).

Bergougnoux, Benjamin, Eduard Eiben, Robert Ganian, Sebastian Ordyniak, and M. S. Ramanujan (2021). "Towards a Polynomial Kernel for Directed Feedback Vertex Set". In: *Algorithmica* 83.5, pp. 1201–1221. ISSN: 1432-0541. DOI: 10.1007/s00453-020-00777-5 (cit. on pp. 3, 4, 13–15, 36, 38, 42–44, 67, 72–75, 78, 80, 82, 84, 126).

Biere, Armin, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger (2020). "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020". In: *Proceedings of SAT Competition 2020 – Solver and Benchmark Descriptions*. Ed. by Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Vol. B-2020-1. Department of Computer Science Report Series B. University of Helsinki, pp. 51–53. URL: http://hdl.handle.net/10138/318754 (cit. on pp. 31, 96).

Biere, Armin, Matti Järvisalo, and Benjamin Kiesl (2021). "Preprocessing". In: *Handbook of Satisfiability*. Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. 2nd ed. Amsterdam: IOS Press. Chap. 9, pp. 391–435. ISBN: 978-1-643-68160-3. DOI: 10.3233/FAIA200992 (cit. on p. 36).

Biere, Armin, Daniel Le Berre, Emmanuel Lonca, and Norbert Manthey (2014). "Detecting Cardinality Constraints in CNF". In: *Theory and Applications of Satisfiability Testing – SAT 2014*. Ed. by Carsten Sinz and Uwe Egly. Cham: Springer International Publishing, pp. 285–301. ISBN: 978-3-319-09284-3. DOI: 10.1007/978-3-319-09284-3_22 (cit. on p. 96).

Bittner, Paul Maximilian, Thomas Thüm, and Ina Schaefer (2019). "SAT Encodings of the At-Most-k Constraint". In: *Software Engineering and Formal Methods*. Ed. by Peter Csaba Ölveczky and Gwen Salaün. Cham: Springer International Publishing, pp. 127–144. ISBN: 978-3-030-30446-1. DOI: 10.1007/978-3-030-30446-1_7 (cit. on p. 95).

Bodlaender, Hans L. and Thomas C. van Dijk (2010). "A Cubic Kernel for Feedback Vertex Set and Loop Cutset". In: *Theory of Computation Systems* 46.3, pp. 566–597. DOI: 10.1007/S00224-009-9234-2 (cit. on p. 73).

Bodlaender, Hans L., Rodney G. Downey, Michael R. Fellows, and Danny Hermelin (2009). "On problems without polynomial kernels". In: *Journal of Computer and System Sciences* 75.8, pp. 423–434. ISSN: 0022-0000. DOI: 10.1016/j.jcss.2009.04.001 (cit. on p. 23).

Červený, Radovan, Michal Dvořák, Xuan Thang Nguyen, Jan Pokorný, Lucie Procházková, Jaroslav Urban, Václav Blažej, Dušan Knop, Šimon Schierreich, and Ondřej Suchý (2022). *Description of the G^2OAT Solver for PACE 2022 Exact track*. Prague: Faculty of Information Technology, Czech Technical University. URL: https://gitlab.fit.cvut.cz/pace-challenge/2022/goat/exact/-/blob/master/paper.pdf (cit. on pp. 58, 59, 74, 119, 121).

Chakradhar, Srimat T., Arun Balakrishnan, and Vishwani D. Agrawal (1994). "An exact algorithm for selecting partial scan flip-flops". In: *Proceedings of the 31st Annual Design Automation Conference*. DAC 1994. San Diego, CA: Association for Computing Machinery, pp. 81–86. ISBN: 0-89791-653-0. DOI: 10.1145/196244.196285 (cit. on p. 11).

Chen, Jianer, Yang Liu, Songjian Lu, Barry O'sullivan, and Igor Razgon (2008). "A Fixed-Parameter Algorithm for the Directed Feedback Vertex Set Problem". In: *Journal of the ACM* 55.5. ISSN: 0004-5411. DOI: 10.1145/1411509.1411511 (cit. on pp. 22, 26, 85, 128).

Chitnis, Rajesh, Marek Cygan, Mohammadtaghi Hajiaghayi, and Dániel Marx (2012). "Directed Subset Feedback Vertex Set Is Fixed-Parameter Tractable". In: *Automata, Languages, and Programming*. Ed. by Artur Czumaj, Kurt Mehlhorn, Andrew Pitts, and Roger Wattenhofer. Berlin, Heidelberg: Springer, pp. 230–241. ISBN: 978-3-642-31594-7. DOI: 10.1007/978-3-642-31594-7_20. arXiv: 1205.1271 (cit. on p. 127).

Chong, Edwin K. P. and Stanislaw H. Żak (2008). "Introduction to Linear Programming". In: *An Introduction to Optimization*. John Wiley & Sons, Ltd. Chap. 15, pp. 297–331. ISBN: 978-1-118-03334-0. DOI: 10.1002/9781118033340.ch15 (cit. on pp. 33, 34).

Cozzens, Margaret (Midge) (2015). "Chapter 2 - Food Webs and Graphs". In: *Algebraic and Discrete Mathematical Methods for Modern Biology*. Ed. by Raina S. Robeva. Boston, MA: Academic Press, pp. 29–49. ISBN: 978-0-12-801213-0. DOI: 10.1016/B978-0-12-801213-0.00002-2 (cit. on p. 10).

Darwiche, Adnan and Knot Pipatsrisawat (2021). "Complete Algorithms". In: *Handbook of Satisfiability*. Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. 2nd ed. Amsterdam: IOS Press. Chap. 3, pp. 101–132. ISBN: 978-1-643-68160-3. DOI: 10.3233/FAIA200986 (cit. on p. 93).

Deutscher, David, Isaac Meilijson, Martin Kupiec, and Eytan Ruppin (2006). "Multiple knockout analysis of genetic robustness in the yeast metabolic network". en. In: *Nature Genetics* 38.9, pp. 993–998. ISSN: 1061-4036, 1546-1718. DOI: 10.1038/ng1856 (cit. on p. 8).

Dey, Sujit, Miodrag Potkonjak, and Rabindra Roy (1994). "Synthesizing designs with low-cardinality minimum feedback vertex set for partial scan application". In: *Proceedings of IEEE VLSI Test Symposium*, pp. 2–7. DOI: 10.1109/VTEST.1994.292342 (cit. on p. 11).

Dinitz, Yefim (1970). "Algorithm for solution of a problem of maximum flow in a network with power estimation". In: *Doklady Akademii Nauk SSSR* 194.4. Title translated from Russian. The author appears as E.A. Dinic in the original publication, pp. 754–757. URL: https://www.mathnet.ru/eng/dan/v194/i4/p754 (cit. on p. 20).

– (2006). "Dinitz' Algorithm: The Original Version and Even's Version". In: *Theoretical Computer Science: Essays in Memory of Shimon Even*. Ed. by Oded Goldreich, Arnold L. Rosenberg, and Alan L. Selman. Berlin, Heidelberg: Springer, pp. 218–240. ISBN: 978-3-540-32881-0. DOI: 10.1007/11685654_10 (cit. on p. 20).

Dirks, Jona and Enna Gerhard (2024). "A Novel Heuristic for Finding Vertices and Edges not on Induced Cycles". In: *Studierendenkonferenz Informatik SKILL 2024*. To appear. Bonn: Gesellschaft für Informatik (cit. on pp. 13, 63).

Dirks, Jona, Enna Gerhard, Mario Grobler, Amer E. Mouawad, and Sebastian Siebertz (2024). "Data Reduction for Directed Feedback Vertex Set on Graphs Without Long Induced Cycles". In: *SOFSEM 2024: Theory and Practice of Computer Science - 49th International Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2024, Cochem, February 19-23, 2024, Proceedings*. Ed. by Henning Fernau, Serge Gaspers, and Ralf Klasing. Vol. 14519. Lecture Notes in Computer Science. Springer International Publishing, pp. 183–197. DOI: 10.1007/978-3-031-52113-3_13. arXiv: 2308.15900 (cit. on pp. 13, 53, 72).

Dom, Michael, Jiong Guo, Falk Hüffner, Rolf Niedermeier, and Anke Truss (2010). "Fixed-parameter tractability results for feedback set problems in tournaments". In: *Journal of Discrete Algorithms* 8.1, pp. 76–86. ISSN: 1570-8667. DOI: 10.1016/j.jda.2009.08.001 (cit. on p. 72).

Dzulfikar, M. Ayaz, Johannes K. Fichte, and Markus Hecher (2019). "The PACE 2019 Parameterized Algorithms and Computational Experiments Challenge: The Fourth Iteration". In: *14th International Symposium on Parameterized and Exact Computation (IPEC 2019)*. Ed. by Bart M. P. Jansen and Jan Arne Telle. Vol. 148. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 25:1–25:23. ISBN: 978-3-959-77129-0. DOI: 10.4230/LIPIcs.IPEC.2019.25 (cit. on p. 27).

Eppstein, David, Maarten Löffler, and Darren Strash (2013). "Listing All Maximal Cliques in Large Sparse Real-World Graphs". In: *ACM Journal of Experimental Algorithmics* 18. ISSN: 1084-6654. DOI: 10.1145/2543629 (cit. on p. 96).

Eppstein, David and Darren Strash (2011). "Listing All Maximal Cliques in Large Sparse Real-World Graphs". In: *Experimental Algorithms*. Ed. by Panos M. Pardalos and Steffen Rebennack. Berlin, Heidelberg: Springer, pp. 364–375. ISBN: 978-3-642-20662-7. DOI: 10.1007/978-3-642-20662-7_31 (cit. on p. 96).

Erdős, Paul and Richard Rado (1960). “Intersection Theorems for Systems of Sets”. In: *Journal of the London Mathematical Society* s1-35.1, pp. 85–90. DOI: 10.1112/jlms/s1-35.1.85 (cit. on p. 67).

Even, Guy, Joseph Naor, Baruch Schieber, and Madhu Sudan (1998). “Approximating Minimum Feedback Sets and Multicuts in Directed Graphs”. In: *Algorithmica* 20.2, pp. 151–174. ISSN: 0178-4617. DOI: 10.1007/PL00009191 (cit. on p. 22).

Even, Shimon and Robert Tarjan (1975). “Network Flow and Testing Graph Connectivity”. In: *SIAM Journal on Computing* 4.4, pp. 507–518. DOI: 10.1137/0204043 (cit. on pp. 20, 84).

Fellows, Michael R., Lars Jaffke, Aliz Izabella Király, Frances A. Rosamond, and Mathias Weller (2018). “What Is Known About Vertex Cover Kernelization?” In: *Essays Dedicated to Juraj Hromkovič on the Occasion of His 60th Birthday*. The journal version of this paper contained an error that made the contained reduction rules unusable. This has been fixed in the updated version on arXiv.org. Cham: Springer International Publishing, pp. 330–356. DOI: 10.1007/978-3-319-98355-4_19. arXiv: 1811.09429v4 (cit. on pp. 36, 46, 47, 50).

Fellows, Michael R., Jan Kratochvil, Matthias Middendorf, and Frank Pfeiffer (1995). “The complexity of induced minors and related problems”. In: *Algorithmica* 13.3, pp. 266–282. ISSN: 1432-0541. DOI: 10.1007/BF01190507 (cit. on p. 53).

Fleischer, Lisa K., Bruce Hendrickson, and Ali Pınar (2000). “On Identifying Strongly Connected Components in Parallel”. In: *Parallel and Distributed Processing*. Ed. by José Rolim. Berlin, Heidelberg: Springer, pp. 505–511. ISBN: 978-3-540-45591-2. DOI: 10.1007/3-540-45591-4_68 (cit. on p. 54).

Fleischer, Rudolf, Xi Wu, and Liwei Yuan (2009). “Experimental Study of FPT Algorithms for the Directed Feedback Vertex Set Problem”. In: *Algorithms - ESA 2009*. Ed. by Amos Fiat and Peter Sanders. Berlin, Heidelberg: Springer, pp. 611–622. ISBN: 978-3-642-04128-0. DOI: 10.1007/978-3-642-04128-0_55 (cit. on pp. 41, 43, 44, 53, 67).

Fomin, Fedor V., Fabrizio Grandoni, and Dieter Kratsch (2009). “A measure & conquer approach for the analysis of exact algorithms”. In: *Journal of the ACM* 56.5. ISSN: 0004-5411. DOI: 10.1145/1552285.1552286 (cit. on p. 47).

Fomin, Fedor V., Daniel Lokshtanov, Saket Saurabh, and Meirav Zehavi (2019a). “Open Problems”. In: *Kernelization: Theory of Parameterized Preprocessing*. Cambridge University Press, pp. 467–473. DOI: 10.1017/9781107415157.026 (cit. on p. 12).

– (2019b). “Problem Definitions”. In: *Kernelization: Theory of Parameterized Preprocessing*. Cambridge University Press, pp. 477–482. DOI: 10.1017/9781107415157.028 (cit. on pp. 25, 26, 28, 29, 31, 32).

– (2019c). “What Is a Kernel?” In: *Kernelization: Theory of Parameterized Preprocessing*. Cambridge University Press, pp. 1–12. DOI: 10.1017/9781107415157.003 (cit. on pp. 21, 22).

Ford, Lester R. and Delbert R. Fulkerson (1956). “Maximal Flow Through a Network”. In: *Canadian Journal of Mathematics* 8, pp. 399–404. DOI: 10.4153/CJM-1956-045-5 (cit. on p. 20).

Funke, Daniel, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz (2018). "Communication-Free Massively Distributed Graph Generation". In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 336–347. DOI: 10.1109/IPDPS.2018.00043. arXiv: 1710.07565 (cit. on p. 111).

Gerhard, Enna (2021). "Bi-Kelly-Width". Supervised by Sebastian Siebertz. Bachelor Thesis. University of Bremen. URL: https://www.szi.uni-bremen.de/wp-content/uploads/2021/08/Bachelorarbeit-Enna-Gerhard-Digital-Edition.pdf (cit. on pp. 15, 22).

Großmann, Ernestine, Tobias Heuer, Christian Schulz, and Darren Strash (2022). "The PACE 2022 Parameterized Algorithms and Computational Experiments Challenge: Directed Feedback Vertex Set". In: *17th International Symposium on Parameterized and Exact Computation (IPEC 2022)*. Ed. by Holger Dell and Jesper Nederlof. Vol. 249. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 26:1–26:18. ISBN: 978-3-959-77260-0. DOI: 10.4230/LIPIcs.IPEC.2022.26 (cit. on pp. 9, 12, 23, 111, 112, 119).

Hespe, Demian, Sebastian Lamm, Christian Schulz, and Darren Strash (2020). "WeGotYouCovered: The Winning Solver from the PACE 2019 Challenge, Vertex Cover Track". In: *2020 Proceedings of the SIAM Workshop on Combinatorial Scientific Computing (CSC)*, pp. 1–11. DOI: 10.1137/1.9781611976229.1. arXiv: 1908.06795 (cit. on pp. 27, 36, 107).

Hunter, Paul and Stephan Kreutzer (2008). "Digraph Measures: Kelly Decompositions, Games, and Orderings". In: *Theoretical Computer Science* 399. DOI: 10.1016/j.tcs.2008.02.038 (cit. on p. 89).

Jansen, Bart M. P. and Hans L. Bodlaender (2013). "Vertex Cover Kernelization Revisited - Upper and Lower Bounds for a Refined Parameter". In: *Theory of Computing Systems* 53.2, pp. 263–299. DOI: 10.1007/S00224-012-9393-4 (cit. on p. 72).

Kahn, A. B. (1962). "Topological sorting of large networks". In: *Communications of the ACM* 5.11, pp. 558–562. ISSN: 0001-0782. DOI: 10.1145/368996.369025 (cit. on pp. 25, 43, 71, 86).

Karp, Richard M. (1972). "Reducibility among Combinatorial Problems". In: *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations*. Ed. by Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger. Boston, MA: Springer US, pp. 85–103. ISBN: 978-1-468-42001-2. DOI: 10.1007/978-1-4684-2001-2_9 (cit. on pp. 11, 21, 22, 25–28).

Kiesel, Rafael and André Schidler (2022). "PACE Solver Description: DAGer - Cutting out Cycles with MaxSAT". In: *17th International Symposium on Parameterized and Exact Computation (IPEC 2022)*. Ed. by Holger Dell and Jesper Nederlof. Vol. 249. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 32:1–32:4. ISBN: 978-3-959-77260-0. DOI: 10.4230/LIPIcs.IPEC.2022.32 (cit. on p. 121).

– (2023). "A Dynamic MaxSAT-based Approach to Directed Feedback Vertex Sets". In: *2023 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pp. 39–52. DOI: 10.1137/1.9781611977561.ch4. arXiv: 2211.06109 (cit. on pp. 119, 121).

Knuth, Donald E. (1976). "Big Omicron and big Omega and big Theta". In: *ACM SIGACT News* 8.2, pp. 18–24. ISSN: 0163-5700. DOI: 10.1145/1008328.1008329 (cit. on p. 19).

Kreowski, Hans-Jörg, Renate Klempien-Hinrichs, and Sabine Kuske (2006). "Some essentials of graph transformation". In: *Recent Advances in Formal Languages and Applications*. Studies in Computational Intelligence. Berlin, Heidelberg: Springer, pp. 229–254. DOI: 10.1007/978-3-540-33461-3_9 (cit. on p. 40).

Kreowski, Hans-Jörg, Sabine Kuske, and Grzegorz Rozenberg (2008). "Graph Transformation Units – An Overview". In: *Concurrency, Graphs and Models: Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*. Berlin, Heidelberg: Springer, pp. 57–75. ISBN: 978-3-540-68676-7. DOI: 10.1007/978-3-540-68679-8_5 (cit. on pp. 40, 127).

Kun, Ádám, Balázs Papp, and Eörs Szathmáry (2008). "Computational identification of obligatorily autocatalytic replicators embedded in metabolic networks". In: *Genome Biology* 9.3, R51. ISSN: 1474-760X. DOI: 10.1186/gb-2008-9-3-r51 (cit. on p. 8).

Kuosmanen, Anna, Topi Paavilainen, Travis Gagie, Rayan Chikhi, Alexandru Tomescu, and Veli Mäkinen (2018). "Using Minimum Path Cover to Boost Dynamic Programming on DAGs: Co-linear Chaining Extended". In: *Research in Computational Molecular Biology*. Ed. by Benjamin J. Raphael. Cham: Springer International Publishing, pp. 105–121. ISBN: 978-3-319-89929-9. DOI: 10.1007/978-3-319-89929-9_7 (cit. on p. 129).

Leiserson, Charles E. and James B. Saxe (1991). "Retiming synchronous circuitry". In: *Algorithmica* 6.1-6, pp. 5–35. ISSN: 0178-4617, 1432-0541. DOI: 10.1007/BF01759032 (cit. on p. 11).

Levy, Hanoch and David W Low (1988). "A contraction algorithm for finding small cycle cutsets". In: *Journal of Algorithms* 9.4, pp. 470–493. ISSN: 0196-6774. DOI: 10.1016/0196-6774(88)90013-2 (cit. on pp. 41, 43, 44).

Li, Chu Min, Hua Jiang, and Felip Manyà (2017). "On minimization of the number of branches in branch-and-bound algorithms for the maximum clique problem". In: *Computers & Operations Research* 84, pp. 1–15. ISSN: 0305-0548. DOI: 10.1016/j.cor.2017.02.017 (cit. on p. 27).

Li, Chu Min and Felip Manyà (2021). "MaxSAT, Hard and Soft Constraints". In: *Handbook of Satisfiability*. Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. 2nd ed. Amsterdam: IOS Press. Chap. 23, pp. 903–927. ISBN: 978-1-643-68160-3. DOI: 10.3233/FAIA201007 (cit. on p. 32).

Li, Ruiming, Chun-Yu Lin, Wei-Feng Guo, and Tatsuya Akutsu (2021). "Weighted minimum feedback vertex sets and implementation in human cancer genes detection". In: *BMC Bioinformatics* 22.1, p. 143. ISSN: 1471-2105. DOI: 10.1186/s12859-021-04062-2 (cit. on p. 11).

Lichtenstein, Orna and Amir Pnueli (1985). "Checking that finite state concurrent programs satisfy their linear specification". In: *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL 1985. New Orleans, LA: Association for Computing Machinery, pp. 97–107. ISBN: 0-89791-147-4. DOI: 10.1145/318593.318622 (cit. on p. 11).

Lin, Hen-Ming and Jing-Yang Jou (2000). "On computing the minimum feedback vertex set of a directed graph by contraction operations". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 19.3, pp. 295–307. DOI: 10.1109/43.833199 (cit. on pp. 11, 41, 43, 44, 53, 54, 58, 59, 125).

Lokshtanov, Daniel, M. S. Ramanujan, and Saket Saurabh (2018). "When Recursion is Better than Iteration: A Linear-Time Algorithm for Acyclicity with Few Error Vertices". In: *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, January 7-10, 2018*. Ed. by Artur Czumaj. SIAM, pp. 1916–1933. DOI: 10.1137/1.9781611975031.125 (cit. on p. 26).

Lokshtanov, Daniel, M. S. Ramanujan, Saket Saurabh, Roohani Sharma, and Meirav Zehavi (2019). "Wannabe Bounded Treewidth Graphs Admit a Polynomial Kernel for DFVS". In: *Algorithms and Data Structures*. Ed. by Zachary Friggstad, Jörg-Rüdiger Sack, and Mohammad R Salavatipour. Cham: Springer International Publishing, pp. 523–537. ISBN: 978-3-030-24766-9. DOI: 10.1007/978-3-030-24766-9_38 (cit. on pp. 11, 72).

Lubiw, Anna (1988). "A note on odd/even cycles". In: *Discrete Applied Mathematics* 22.1, pp. 87–92. ISSN: 0166-218X. DOI: 10.1016/0166-218X(88)90125-4 (cit. on p. 53).

Meiburg, Alex (2022). *Reduction Rules and ILP Are All You Need: Minimal Directed Feedback Vertex Set*. arXiv: 2208.01119 (cit. on pp. 119, 122).

Mnich, Matthias and Erik Jan van Leeuwen (2017). "Polynomial kernels for deletion to classes of acyclic digraphs". In: *Discrete Optimization* 25, pp. 48–76. ISSN: 1572-5286. DOI: 10.1016/j.disopt.2017.02.002 (cit. on p. 72).

Morgado, Antonio, Carmine Dodaro, and Joao Marques-Silva (2014). "Core-Guided MaxSAT with Soft Cardinality Constraints". In: *Principles and Practice of Constraint Programming*. Ed. by Barry O'Sullivan. Cham: Springer International Publishing, pp. 564–573. ISBN: 978-3-319-10428-7. DOI: 10.1007/978-3-319-10428-7_41 (cit. on p. 33).

Nutov, Zeev and Raphael Yuster (2004). "Packing Directed Cycles Efficiently". In: *Mathematical Foundations of Computer Science 2004*. Ed. by Jiří Fiala, Václav Koubek, and Jan Kratochvíl. Berlin, Heidelberg: Springer, pp. 310–321. ISBN: 978-3-540-28629-5. DOI: 10.1007/978-3-540-28629-5_22 (cit. on p. 68).

Plachetta, Rick and Alexander van der Grinten (2021). "SAT-and-Reduce for Vertex Cover: Accelerating Branch-and-Reduce by SAT Solving". In: *Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pp. 169–180. DOI: 10.1137/1.9781611976472.13 (cit. on p. 107).

Prestwich, Steven (2021). "CNF Encodings". In: *Handbook of Satisfiability*. Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. 2nd ed. Amsterdam: IOS Press. Chap. 2, pp. 75–100. ISBN: 978-1-643-68160-3. DOI: 10.3233/FAIA200985 (cit. on p. 31).

Savelsbergh, Martin W. P. (1994). "Preprocessing and Probing Techniques for Mixed Integer Programming Problems". In: *ORSA Journal on Computing* 6.4, pp. 445–454. ISSN: 0899-1499. DOI: 10.1287/ijoc.6.4.445 (cit. on p. 36).

Sun, Hao (2024). "A Constant Factor Approximation for Directed Feedback Vertex Set in Graphs of Bounded Genus". In: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2024)*. Ed. by Amit Kumar and Noga Ron-Zewi. Vol. 317. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 18:1–18:20. ISBN: 978-3-95977-348-5. DOI: 10.4230/LIPIcs.APPROX/RANDOM.2024.18 (cit. on p. 22).

Swat, Sylwester (2022a). "PACE Solver Description: DiVerSeS - A Heuristic Solver for the Directed Feedback Vertex Set Problem". In: *17th International Symposium on Parameterized and Exact Computation, IPEC 2022, September 7-9, 2022, Potsdam*. Ed. by Holger Dell and Jesper Nederlof. Vol. 249. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 27:1–27:3. DOI: 10.4230/LIPICS.IPEC.2022.27 (cit. on p. 122).

– (2022b). *PACE Solver Description: Finding optimal feedback vertex sets of directed graphs using DiVerSeS*. Version v1.0.1. The exact solver description is only contained in Version v.1.0.1. Filepath: /DiVerSeS_description/DiVerSeS_description_exact.pdf. DOI: 10.5281/zenodo.6657522 (cit. on pp. 41, 43, 44, 54, 58, 122).

Tamura, Takeyuki, Kazuhiro Takemoto, and Tatsuya Akutsu (2010). "Finding Minimum Reaction Cuts of Metabolic Networks Under a Boolean Model Using Integer Programming and Feedback Vertex Sets". In: *International Journal of Knowledge Discovery in Bioinformatics (IJKDB)* 1.1, pp. 14–31. ISSN: 1947-9115. DOI: 10.4018/jkdb.2010100202 (cit. on p. 8).

Tarjan, Robert (1972). "Depth-First Search and Linear Graph Algorithms". In: *SIAM Journal on Computing* 1.2, pp. 146–160. DOI: 10.1137/0201010 (cit. on p. 54).

Van Bevern, René (2013). "Towards Optimal and Expressive Kernelization for d-Hitting Set". en. In: *Algorithmica*. ISSN: 0178-4617, 1432-0541. DOI: 10.1007/s00453-013-9774-3 (cit. on p. 67).

Vanderbei, Robert J. (2020). *Linear Programming: Foundations and Extensions*. Vol. 285. International Series in Operations Research & Management Science. Cham: Springer International Publishing. ISBN: 978-3-030-39415-8. DOI: 10.1007/978-3-030-39415-8 (cit. on pp. 33, 34).

Weihe, Karsten (1998). "Covering Trains by Stations or the Power of Data Reduction". In: *Proceedings of Algorithms and Experiments (ALEX98)*. Ed. by Roberto Battati, Alan Bertossi, and Silvano Martello. Trento (cit. on p. 53).

Xiong, Ziliang and Mingyu Xiao (2024). *A Simplified Parameterized Algorithm for Directed Feedback Vertex Set*. arXiv: 2410.15411 (cit. on p. 26).

All references have been verified on November 22, 2024. Digital Object Identifiers (DOIs) can be resolved through https://www.doi.org/. Articles published via arXiv can be accessed via https://www.arxiv.org/.

References in the text use surnames except for authors sharing a surname, in this case full initials are added. Three or more authors are shortened to et al., only the first author appears unless this author appears in several different constellations. If there are multiple publications for the same group of authors in a single year, a letter is added to the year to distinguish them. In the bibliography, entries are grouped if they have been published by the same authors.

A. Appendix

A.1. Thanks

I would like to thank everyone that supported me during the creation of this thesis. From the members of the original project to the theory group in which we continued our research to everyone that made me a cup of tea while I was writing: thank you! I was able to write in the most hospitable of places – from staying with friends to spontaneous co-working sessions to all the lovely cafes and tea rooms, both locally and abroad.

It is needless to say that I will always remember this special time of working on the DIRECTED FEEDBACK VERTEX SET problem.

A.2. Implementation

The implementation can be found in the GitLab repository of the department at `https://gitlab.informatik.uni-bremen.de/grapa/java`. The packages of the solver are located at `de.uni.bremen.grapa.solver`, the packages of the library at `de.uni.bremen.grapa.library`.

An archive containing the source code of the improved solver and the extended performance measurements conduced for this thesis accompanies this submission.

A.3. Tables

The following pages contain detailed information regarding the effects of data reduction rules and different solving approaches.

	Before reduction				After reduction				Reduct.	
Ins.	n	m	undir.	k	n	m	undir.	k	Time	Type
051	32k	246k	116k	21k	69	271	134	45	817 ms	UNDIR
052	16k	77k	38k	10k	13	62	31	9	241 ms	VC
053	2047	10k	1	58	742	4k	10	55	6 s	DIR
054	15k	103k	37k	8k	1962	10k	4k	1215	1233 ms	UNDIR
055	16k	149k	74k	12k	1638	10k	5k	1171	886 ms	VC
056	2048	31k	15k	1741	1123	12k	6k	893	145 ms	VC
057	1024	25k	12k	915	643	11k	5k	545	95 ms	VC
058	32k	304k	144k	24k	7k	50k	25k	4k	6 s	UNDIR
059	32k	304k	144k	24k	6k	44k	22k	4k	4 s	UNDIR
060	32k	272k	116k	21k	7k	47k	23k	4k	6 s	UNDIR
061	1023	5k	2	73	514	3k	17	70	2486 ms	DIR
062	32k	255k	102k	20k	6k	42k	20k	4k	6 s	UNDIR
063	32k	288k	130k	22k	7k	55k	27k	5k	5 s	UNDIR
064	2035	25k	10k	1469	1731	17k	8k	1256	354 ms	UNDIR
065	32k	255k	102k	20k	5k	38k	18k	3k	4 s	UNDIR
066	32k	238k	90k	19k	6k	39k	18k	4k	5 s	UNDIR
067	2015	22k	7k	1314	1647	15k	6k	1112	2570 ms	UNDIR
068	32k	238k	90k	19k	5k	36k	17k	3k	5 s	UNDIR
069	2047	6k	1	68	634	3k	15	64	3 s	DIR
070	65k	288k	137k	38k	0	0	0	0	1486 ms	VC
071	65k	303k	151k	40k	14	60	30	10	1753 ms	VC
072	65k	303k	151k	40k	14	62	31	10	1712 ms	VC
073	2040	27k	11k	1526	1735	19k	9k	1301	296 ms	UNDIR
074	1024	25k	12k	918	700	12k	6k	602	105 ms	VC
075	1023	3k	4	55	388	1893	7	52	1060 ms	DIR
076	65k	653k	310k	48k	21k	163k	81k	15k	27 s	UNDIR
077	2016	22k	7k	1345	1664	15k	6k	1138	2138 ms	UNDIR
078	65k	618k	278k	46k	23k	182k	90k	16k	25 s	UNDIR
079	65k	652k	310k	48k	21k	165k	82k	15k	27 s	UNDIR
080	64k	548k	220k	42k	20k	138k	66k	13k	24 s	UNDIR
081	2048	28k	12k	1570	1731	20k	10k	1322	183 ms	UNDIR
082	130k	573k	272k	76k	6	24	12	4	5 s	VC
083	131k	603k	301k	80k	84	430	215	56	4 s	VC
084	1023	5k	0	73	563	3k	10	71	2918 ms	DIR
085	131k	604k	302k	80k	47	214	107	32	5 s	VC
086	131k	604k	302k	80k	35	164	82	25	4 s	VC
087	1024	23k	10k	842	969	19k	9k	797	217 ms	UNDIR
088	2005	22k	8k	1322	1585	14k	6k	1074	1716 ms	UNDIR
089	32k	320k	160k	25k	7k	49k	24k	5k	4 s	VC
090	2043	27k	11k	1518	1783	20k	9k	1332	270 ms	UNDIR
091	32k	321k	160k	25k	6k	46k	23k	4k	3 s	VC
092	1023	5k	0	65	548	3k	9	65	3 s	DIR
093	4k	62k	28k	3k	3k	48k	24k	2822	604 ms	UNDIR
094	1023	3k	6	57	424	2010	8	55	1329 ms	DIR
095	883	1799	7	51	243	855	12	46	103 ms	DIR

Table A.1.: Effects of reduction with all reduction rules enabled

	Iterative Hitting Set			Partial order	
Ins.	ILP	Max SAT	Vertex Cover	ILP	*DAGer*
051	**1461 ms**	2233 ms	1617 ms	1478 ms	**808 ms**
052	501 ms	579 ms	535 ms	**446 ms**	**341 ms**
053	**14 s**	18 min			**2062 ms**
054	3 s	9 s	**3 s**	3 s	**537 ms**
055	3 min	39 s	**1985 ms**	16 s	**992 ms**
056	9 min	**33 s**		6 min	**1083 ms**
057	8 min	**27 s**		3 min	**1815 ms**
058		3 min	**32 s**	7 min	**2755 ms**
059	5 min	123 s	23 s	**21 s**	**2494 ms**
060	4 min	137 s	**18 s**	25 s	**3 s**
061	**95 s**				**89 s**
062	64 s	85 s	**16 s**	18 s	**2741 ms**
063	7 min	155 s	**17 s**	35 s	**4 s**
064		**63 s**			**1991 ms**
065	120 s	85 s	**12 s**	15 s	**3 s**
066	81 s	76 s	**13 s**	25 s	**4 s**
067		**85 s**			**2148 ms**
068	52 s	53 s	**13 s**	22 s	**4 s**
069	**37 s**	14 min			**6 s**
070	3 s	**2968 ms**	4 s	3 s	**1792 ms**
071	3 s	3 s	3 s	**3 s**	**1765 ms**
072	**2903 ms**	3 s	3 s	3 s	**1482 ms**
073		**65 s**		22 min	**1953 ms**
074		**36 s**		6 min	**8 s**
075	**174 s**				**49 s**
076			**62 s**		**52 s**
077		**173 s**			**2043 ms**
078			**65 s**		86 s
079			**72 s**		**28 s**
080		11 min	**55 s**		92 s
081		**55 s**		21 min	**4 s**
082	17 s	10 s	11 s	**10 s**	**3 s**
083	10 s	9 s	**8 s**	10 s	**3 s**
084	**6 min**				**3 min**
085	10 s	10 s	**8 s**	10 s	**4 s**
086	10 s	9 s	**7 s**	10 s	**3 s**
087		**32 s**			**3 s**
088		**139 s**			**1940 ms**
089		4 min	**21 s**		36 s
090		**78 s**			**2515 ms**
091		4 min	**175 s**		**21 s**
092	**158 s**				160 s
093		**137 s**			**14 s**
094	**27 min**				
095					**23 min**

Table A.2.: Performance of individual solving techniques on individual instances

	Before reduction				After reduction				Reduct.	
Ins.	n	m	**undir.**	k	n	m	**undir.**	k	**Time**	**Type**
053	2047	10k	1	58	742	4k	10	55	6 s	DIR
061	1023	5k	2	73	514	3k	17	70	2486 ms	DIR
069	2047	6k	1	68	634	3k	15	64	3 s	DIR
075	1023	3k	4	55	388	1893	7	52	1060 ms	DIR
084	1023	5k	0	73	563	3k	10	71	2918 ms	DIR
092	1023	5k	0	65	548	3k	9	65	3 s	DIR
094	1023	3k	6	57	424	2010	8	55	1329 ms	DIR
095	883	1799	7	51	243	855	12	46	103 ms	DIR
051	32k	246k	116k	21k	69	271	134	45	817 ms	UNDIR
054	15k	103k	37k	8k	1962	10k	4k	1215	1233 ms	UNDIR
058	32k	304k	144k	24k	7k	50k	25k	4k	6 s	UNDIR
059	32k	304k	144k	24k	6k	44k	22k	4k	4 s	UNDIR
060	32k	272k	116k	21k	7k	47k	23k	4k	6 s	UNDIR
062	32k	255k	102k	20k	6k	42k	20k	4k	6 s	UNDIR
063	32k	288k	130k	22k	7k	55k	27k	5k	5 s	UNDIR
064	2035	25k	10k	1469	1731	17k	8k	1256	354 ms	UNDIR
065	32k	255k	102k	20k	5k	38k	18k	3k	4 s	UNDIR
066	32k	238k	90k	19k	6k	39k	18k	4k	5 s	UNDIR
067	2015	22k	7k	1314	1647	15k	6k	1112	2570 ms	UNDIR
068	32k	238k	90k	19k	5k	36k	17k	3k	5 s	UNDIR
073	2040	27k	11k	1526	1735	19k	9k	1301	296 ms	UNDIR
076	65k	653k	310k	48k	21k	163k	81k	15k	27 s	UNDIR
077	2016	22k	7k	1345	1664	15k	6k	1138	2138 ms	UNDIR
078	65k	618k	278k	46k	23k	182k	90k	16k	25 s	UNDIR
079	65k	652k	310k	48k	21k	165k	82k	15k	27 s	UNDIR
080	64k	548k	220k	42k	20k	138k	66k	13k	24 s	UNDIR
081	2048	28k	12k	1570	1731	20k	10k	1322	183 ms	UNDIR
087	1024	23k	10k	842	969	19k	9k	797	217 ms	UNDIR
088	2005	22k	8k	1322	1585	14k	6k	1074	1716 ms	UNDIR
090	2043	27k	11k	1518	1783	20k	9k	1332	270 ms	UNDIR
093	4k	62k	28k	3k	3k	48k	24k	2822	604 ms	UNDIR
052	16k	77k	38k	10k	13	62	31	9	241 ms	VC
055	16k	149k	74k	12k	1638	10k	5k	1171	886 ms	VC
056	2048	31k	15k	1741	1123	12k	6k	893	145 ms	VC
057	1024	25k	12k	915	643	11k	5k	545	95 ms	VC
071	65k	303k	151k	40k	14	60	30	10	1753 ms	VC
072	65k	303k	151k	40k	14	62	31	10	1712 ms	VC
074	1024	25k	12k	918	700	12k	6k	602	105 ms	VC
082	130k	573k	272k	76k	6	24	12	4	5 s	VC
083	131k	603k	301k	80k	84	430	215	56	4 s	VC
085	131k	604k	302k	80k	47	214	107	32	5 s	VC
086	131k	604k	302k	80k	35	164	82	25	4 s	VC
089	32k	320k	160k	25k	7k	49k	24k	5k	4 s	VC
091	32k	321k	160k	25k	6k	46k	23k	4k	3 s	VC

Table A.3.: Overview of properties of instances grouped by instance type

	Iterative Hitting Set			Partial order	
Ins.	**ILP**	**Max SAT**	**Vertex Cover**	**ILP**	*DAGer*
053	**14 s**	18 min			**2062 ms**
061	**95 s**				**89 s**
069	**37 s**	14 min			**6 s**
075	**174 s**				**49 s**
084	**6 min**				**3 min**
092	**158 s**				160 s
094	**27 min**				
095					**23 min**
051	**1461 ms**	2233 ms	1617 ms	1478 ms	**808 ms**
054	3 s	9 s	**3 s**	3 s	**537 ms**
058		3 min	**32 s**	7 min	**2755 ms**
059	5 min	123 s	23 s	**21 s**	**2494 ms**
060	4 min	137 s	**18 s**	25 s	**3 s**
062	64 s	85 s	**16 s**	18 s	**2741 ms**
063	7 min	155 s	**17 s**	35 s	**4 s**
064		**63 s**			**1991 ms**
065	120 s	85 s	**12 s**	15 s	**3 s**
066	81 s	76 s	**13 s**	25 s	**4 s**
067		**85 s**			**2148 ms**
068	52 s	53 s	**13 s**	22 s	**4 s**
073		**65 s**		22 min	**1953 ms**
076			**62 s**		**52 s**
077		**173 s**			**2043 ms**
078			**65 s**		86 s
079			**72 s**		**28 s**
080		11 min	**55 s**		92 s
081		**55 s**		21 min	**4 s**
087		**32 s**			**3 s**
088		**139 s**			**1940 ms**
090		**78 s**			**2515 ms**
093		**137 s**			**14 s**
052	501 ms	579 ms	535 ms	**446 ms**	**341 ms**
055	3 min	39 s	**1985 ms**	16 s	**992 ms**
056	9 min	**33 s**		6 min	**1083 ms**
057	8 min	**27 s**		3 min	**1815 ms**
071	3 s	3 s	3 s	**3 s**	**1765 ms**
072	**2903 ms**	3 s	3 s	3 s	**1482 ms**
074		**36 s**		6 min	**8 s**
082	17 s	10 s	11 s	**10 s**	**3 s**
083	10 s	9 s	**8 s**	10 s	**3 s**
085	10 s	10 s	**8 s**	10 s	**4 s**
086	10 s	9 s	**7 s**	10 s	**3 s**
089		4 min	**21 s**		36 s
091		4 min	**175 s**		**21 s**

Table A.4.: Performance of solving techniques grouped by instance type

List of Figures

1.1. Simplified fictive example of a metabolic network with an optimum solution . . 8
1.2. Processes waiting on each other in a deadlock 9
1.3. Example graph with minimum DFVS 10
1.4. Example of using DFVS for graph layouts 11

2.1. Basic graph examples 15
2.2. Path examples 18
2.3. A directed graph, split into its subgraphs and undirected structures 18
2.4. Conflicting properties of algorithms 22
2.5. Overview of reductions from DFVS to other problems 24
2.6. DFVS examples 26
2.7. A reduction from Vertex Cover to DFVS 27
2.8. Examples of a minimum Feedback Vertex Set 28
2.9. A reduction from Vertex Cover to Hitting Set 29
2.10. Reduction from DFVS to Hitting Set to Vertex Cover 30

3.1. Applying reduction rules to a DFVS instance and solving it 36
3.2. A log of changes to be applied after solving and reconstructing the solution . . 39
3.3. Possible neighborhood layouts of vertices with degree three 49
3.4. Strong connected components 56
3.5. Edge not on a directed cycle 59
3.6. Edge with edges allowed for search and forbidden vertices marked 60
3.7. Sets of disallowed successors on the example from Figure 3.6 61
3.8. Directed minor to exclude for our Edge on Induced Cycle heuristic 63
3.9. Introduce edge if more disjoint paths than vertices in solution exist 69
3.10. Tunnel examples 71

4.1. A vertex contributing to different potential edges 73
4.2. Undirected forest outside of the underlying Feedback Vertex Set 78
4.3. Maximal inner and outer paths with internal vertices of degree two 79
4.4. The structure of the forest after contracting degree two vertices exhaustively . . 79
4.5. Example of an application of Reduction rule 22 81
4.6. Example of an application of Reduction rule 23 82
4.7. Counterexample with an arbitrarily large graph of solution size two 84

5.1. Iterative solving adding cycle packings having to add almost all cycles 88

5.2. Example of a linear and an induced partial order 90
5.3. Shared edge on three-cycles 94
5.4. Examples of Triforces 96
5.5. Structure of constraints of the combined formulation 98
5.6. Vertex Cover gadgets 100
5.7. Introducing gadgets to instances 102
5.8. Branching example 103
5.9. Fictive branch and reduce tree 108
5.10. Overview of solving 110

6.1. Implemented reduction rules 114
6.2. Effects of applying reduction rules 115
6.3. Reduction time 116
6.4. Comparison of our own solving approaches 117
6.5. Comparison of solvers 120
6.6. Overview of our improved solving approach 123
6.7. Comparison of our improved solving approaches and DAGer 124

List of Tables

3.1. Overview of reduction rules 37

6.1. Fastest completion of solving techniques on instances that were solved 118
6.2. Overview of PACE 2022 exact solver submissions solving 150 instances . . . 119

A.1. Effects of reduction with all reduction rules enabled 140
A.2. Performance of individual solving techniques on individual instances 141
A.3. Overview of properties of instances grouped by instance type 142
A.4. Performance of solving techniques grouped by instance type 143

List of Algorithms

2.1. Breadth First Search 20

2.2. Verifying that a solution is a DFVS 25

5.1. Iterative Hitting Set generation 86

5.2. Hitting Set ILP Formulation 86

5.3. Partial order Extended SAT formulation 91

5.4. Partial order ILP formulation 92

5.5. Combined ILP formulation 97

Statutory declaration

The following statements are mandatory and therefore given in German.

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Teile meiner Arbeit, die wortwörtlich oder dem Sinn nach anderen Werken entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht. Gleiches gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet, dazu zählen auch KI-basierte Anwendungen oder Werkzeuge. Die Arbeit wurde in gleicher oder ähnlicher Form noch nicht als Prüfungsleistung eingereicht. Die elektronische Fassung der Arbeit stimmt mit der gedruckten Version überein. Mir ist bewusst, dass wahrheitswidrige Angaben als Täuschung behandelt werden.

Die Abschlussarbeit wird zwei Jahre nach Studienabschluss dem Archiv der Universität Bremen zur dauerhaften Archivierung angeboten.

Ich bin damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

Eingereichte Arbeiten können nach § 18 des Allgemeinen Teil der Bachelor- bzw. der Masterprüfungsordnungen der Universität Bremen mit qualifizierter Software auf Plagiatsvorwürfe untersucht werden. Zum Zweck der Überprüfung auf Plagiate erfolgt das Hochladen auf den Server der von der Universität Bremen aktuell genutzten Plagiatssoftware.

Ich bin nicht damit einverstanden, dass die von mir vorgelegte und verfasste Arbeit zum o.g. Zweck dauerhaft auf dem externen Server der aktuell von der Universität Bremen genutzten Plagiatssoftware, in einer institutionseigenen Bibliothek (Zugriff nur durch die Universität Bremen), gespeichert wird.