The Practicality of Reduction Rules for the Directed Feedback Vertex Set Problem

Tanja, Stefan A.

Award date:
2022

Department of Mathematics and Computer Science
Algorithms, Geometry & Applications

The Practicality of Reduction Rules for the Directed Feedback Vertex Set Problem

Master's thesis

S.A. Tanja

8-9-2022

Abstract

Kernelizations have been used as a form of preprocessing for NP-hard problems, to enable the usage of exact methods, such as *Integer Linear Programming* (ILP) algorithms, by shrinking the input as much as possible. The application of kernelizations is mostly theoretical, so it is relatively unclear how practical kernelizations are, and when they enable the practical use of exact algorithms. In this thesis we investigate how effective kernelizations are in obtaining practical exact algorithms. We focus on the DIRECTED FEEDBACK VERTEX SET problem. We design and implement a kernelization, together with an ILP-based algorithm, and a Branch & Bound algorithm, which both use our kernelization. We then evaluate both algorithms with and without reductions, and we submit our best performing algorithm to the *Parameterized Algorithms and Computational Experiments* challenge to evaluate its performance. We determine that reductions certainly can accelerate an exact algorithm, such as a Branch & Bound algorithm, but already practical ILP-based algorithms can be obtained in conjunction with a strong ILP solver without applying any reductions at all.

Contents

Chapter 1

Introduction

The past few decades, the field of Parameterized Algorithms has made significant progress in finding algorithms for NP-hard problems that run in polynomial time given a fixed parameter k, which captures the hardness of the instances. Unfortunately, the hidden constant in the running time, which is a function of k, is usually exponential. Furthermore, most of the algorithmic results within this field are mainly of theoretical nature. It is therefore unclear whether these algorithmic results are practical, i.e., in terms of running time and ease of implementation.

One technique from this field is to use a preprocessing algorithm, called a *kernelization*, which is composed out of several *reduction rules.* The goal of a kernelization is to *reduce*, i.e., shrink, the size of an input instance such that the resulting reduced instance has the same YES or NO answer (in the context of decision problems). Additionally, the kernelization has hopefully reduced the size of an instance to a sufficiently small size so that any exact algorithm will terminate in reasonable time on this reduced instance. In 1998, Bertossi [2] discussed that a kernelization may sufficiently shrink the size of an instance so that slow exact algorithms can solve the remainder, or the remainder is already so small that it can be solved by hand . However, there are a few problems with a kernelization.

It may not always be the case that a reduced instance is easier for an exact algorithm. Hespe et al. [11] determined that their Branch & Bound algorithm with integrated kernelizations sometimes required more time to find an optimal solution of a VERTEX COVER instance compared to when the kernelization was not performed.

In this thesis, we aim to determine when a kernelization can be useful to obtain more practical exact algorithms, i.e., in terms of running time or memory usage, using different exact algorithmic techniques as a comparison. We will focus on the DIRECTED FEEDBACK VERTEX SET problem, which asks us to find a smallest set of vertices of a directed graph to remove to make the remainder acyclic. The choice of this problem is motivated by this year's (2022) *Parameterized Algorithms and Computational Experiments* (PACE) challenge, which focuses on the Directed Feedback Vertex Set problem. One aim of the international challenge is to bridge the gap between theory and practice, i.e., produce practical implementations of exact algorithms for NP-hard problems.

The primary contributions of this thesis are as follows. We first determine whether a kernelization can accelerate solving an ILP-based algorithm and a Branch & Bound algorithm for the Directed Feedback Vertex Set problem. Additionally, since we indeed want to generate a subset of the constraints for the ILP-formulation of the Directed Feedback Vertex Set problem, we investigate whether a kernelization is indeed helpful to generate smaller ILP models. For the Branch & Bound algorithm we investigate whether kernelizations can be used sparingly throughout the search tree or are needed at each level. Kernelizations may not necessarily be cheap in terms of running time, and may not always manage to significantly reduce the current instances.

In addition to our primary contributions, our secondary contribution is to design an ILP-based algorithm and Branch & Bound algorithm that is as efficient as possible. The PACE challenge is an ideal opportunity to evaluate our best performing algorithm[1].

1.1 Related Work

Some work already has been performed in determining when a kernelization may be useful to apply. As already mentioned, Bertossi [2] determined that a kernelization was highly practical, and Hespe et al. determined that a kernelization may also sometimes increase the running time of a Branch & Bound algorithm.

Ferizovic et al. [7] discuss that a kernelization for the Maximum Cut problem allowed the speed up of several orders of magnitude on state-of-the-art exact solvers, together with managing to solve three instances in two seconds that remained unsolved with a ten-hour time-limit.

Strash [14] determined that the primary strength of a kernelization during a Branch & Reduce comes from the initial kernelization, but not so much the subsequent kernelizations. Additionally, Strash showed that most real-world instances already benefit significantly from a small set of reduction rules. Most of these works have, however, not compared different exact methods with and without kernelizations in great detail.

Regarding the Directed Feedback Vertex Set problem, Fleischer et al. [8] propose six reduction rules. Four of the six reduction rules were either already implicitly present in literature, or were modified from the undirected problem, the Feedback Vertex Set problem. The remaining

[1]As per the competition's rules, we cannot submit multiple algorithms that share a significant portion of the same code base.

two reduction rules are based on pairwise-disjoint cycles starting in the same vertex. Such cycles can be computed using flow algorithms.

Recently, Bergougnoux et al. [1] investigated whether there exists a polynomial kernel for the DIRECTED FEEDBACK VERTEX SET problem. Some of their given reduction rules match the reduction rules given by Fleischer, or make some reduction rules stronger.

Reduction rules can be helpful to apply in both heuristics, as well as exact algorithms. Galinier et al. [10], give a heuristic to find a directed feedback vertex set. In order to improve the performance of the heuristic (in terms of solution-size), they apply some basic reduction rules. They then find a directed feedback vertex set by searching for topological ordering using as many vertices as possible, and returning the vertices excluded in the topological ordering.

Razgon [12] showed that the DIRECTED FEEDBACK VERTEX SET problem could be solved in $\mathcal{O}^*(1.9977^n)$, i.e., an algorithm whose exponential factor is 1.9977^n. A minimum Directed Feedback Vertex Set (DFVS) is found using a Branch & Bound algorithm, which identifies many cases to branch on.

Later, Chen et al. [4] designed a Fixed-Parameter tractable algorithm for the decision version of the DIRECTED FEEDBACK VERTEX SET problem with a running time of $4^k k! \cdot n^{\mathcal{O}(1)}$, where k is the solution size. However, Fleischer et al. [8] showed that an implementation of this algorithm quickly slows down for larger density graphs, and when $k \geq 8$.

During the first iteration of the PACE challenge, in 2016, the FEEDBACK VERTEX SET problem was one of the competition's problems. The ILP-based algorithm implemented by Dell et al. [5] was competitive with the best submitted solvers, of which all relied on different techniques than an ILP. This ILP formulation can be directly applied to the DIRECTED FEEDBACK VERTEX SET problem.

1.2 Overview

Firstly, in Chapter 2 we discuss the notation we will be using in the thesis, how we define reduction rules in optimization problems, and the algorithmic toolkit we will be using. In Chapter 3 we discuss which reduction rules we will be applying on instances of the DIRECTED FEEDBACK VERTEX SET problem. In Chapter 4 we discuss our implementation of an ILP-based algorithm for the problem, which became our best performing algorithm. In Chapter 5, we implement a straightforward Branch & Bound solver, integrating reductions and also ideas from our ILP-based algorithm. In Chapter 6 we discuss and evaluate the performance of our ILP-based algorithm in the context of the PACE challenge, and analyze how effective our kernelization is at decreasing the running time of an ILP solver, and whether smaller linear models are produced when a kernelization is used. In Chapter 7 we analyze the performance of reduction rules in a Branch & Bound algorithm, and investigate whether kernelizations can be used sparingly throughout the search tree or are needed at each level.

Chapter 2

Preliminaries

While this thesis is focused on analyzing the effectiveness of kernelizations, also known as *data reductions*, stemming from the field of Fixed-Parameter Tractability, we require some basic concepts and algorithmic techniques from other domains. In this chapter we make the reader familiar with the relevant concepts (for this thesis) related to graph theory, Fixed-Parameter Tractability, data reductions, exact algorithms, and heuristics. In Section 2.1 we introduce all relevant graph definitions we will be using going forward in the thesis, and we introduce the Directed Feedback Vertex Set problem formally. In Section 2.2, we briefly introduce the field of Fixed-Parameter Tractability and a commonly used technique known as *data reductions* for decision problems, followed by defining the technique for optimization problems. We then discuss two common exact algorithmic techniques, Integer Linear Programming and Branch & Bound, which can solve optimization problems, in Section 2.4. We will need both techniques to evaluate the effectiveness of data reductions for the Directed Feedback Vertex Set problem. Finally, in Section 2.5 we briefly discuss the heuristic technique *Simulated Annealing*, which we can use to improve the performance of both an Integer Linear Program and a Branch & Bound algorithm.

2.1 Graph Definitions

An *undirected* graph is an ordered pair $G = (V, E)$ where V is a set whose elements are called vertices, and E is an *unordered* pair of *edges*. A *directed* graph is also an ordered pair $G = (V, E)$, but where E is an *ordered* pair of edges, i.e., the edges have a *direction*. To make the distinction in which direction the edges are oriented, we let $N^+(v)$ be the set of vertices for which v has an outgoing edge, and let $N^-(v)$ be the set of vertices for which v has an incoming edge, in some directed graph. We denote the *closed forward neighborhood* of a vertex v as $N^+[v] = N^+(v) \cup \{v\}$. Additionally, to disambiguate between the neighborhood of different graphs, we write the graph in the subscript, e.g., $N^+_G(v)$ for some directed graph $G = (V, E)$.

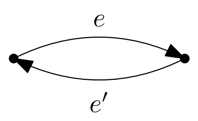

Figure 2.1: A graph with two bidirectional edges.

Since we may consider multiple graphs at once, we need to easily disambiguate between the vertex and edge sets of several graphs. Hence, we also denote the set of vertices and edges of a graph G as $V(G)$ and $E(G)$, respectively.

Furthermore, given a graph $G = (V, E)$, we denote the induced graph of G, given a set of vertices $X \subseteq V$, as $G[X]$, where $V(G[X]) = X$ and $E(G[X]) = \{(u, v) \in E \mid u \in X \land v \in X\}$. Additionally, we abbreviate the union of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ as $G_1 + G_2 = (V_1 \cup V_2, E_1 \cup E_2)$.

Definition 2.1. A directed graph $G = (V, E)$ is *simple* if and only if every edge $(u, v) \in E$ occurs exactly once, i.e., E is not a multiset, and if $(u, u) \notin E$ for every $u \in V$. A directed graph is *non-simple* if at least for one $u \in V$ we have $(u, u) \in E$, or when E is a multiset.

Definition 2.2. Given a directed graph $G = (V, E)$, we say that an edge $e = (u, v) \in E$ is *monodirectional* if $(v, u) \notin E$. Furthermore, we say that an edge $e = (u, v) \in E$ is *bidirectional* if $e' = (v, u) \in E$, and we say that e is bidirectional *with* e'.

Consider the graph G in Figure 2.1, here the edge e is a bidirectional edge, and it is bidirectional with e'. Note that this relation is symmetric, hence, G has two bidirectional edges.

When we only have bidirectional edges in the graph, we have a graph that is very similar to an undirected graph: any path in an undirected graph can be reversed, and the same holds here. We therefore make the following distinction. On the other hand, we can also have a graph without any bidirectional edges.

Definition 2.3. A directed graph $G = (V, E)$ is *bidirected*, if for every edge $e \in E$, e is bidirectional. A directed graph G is *monodirected* if G does not have any bidirectional edges.

As an example, we have a monodirected graph G in Figure 2.2. Observe that for any edge $e = (u, v) \in E$, we have $(v, u) \notin E$.

From a directed graph, we can obtain an *induced undirected graph*, which we define as follows.

Definition 2.4. Let $G = (V, E)$ be a directed graph, then its *induced undirected graph* is given by $\mathcal{U}(G) = (V, E')$ with $E' = \{(u, v) \mid (u, v) \in E \lor (v, u) \in E\}$.

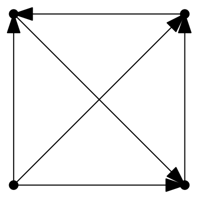

Figure 2.2: A monodirected graph.

Definition 2.5. Given an undirected graph $G = (V, E)$, a set of vertices $Q \subseteq V$ is a *clique* if and only if $(u, v) \in E$, for every $u, v \in Q$, with $u \neq v$.

Definition 2.6. Let $G = (V, E)$ be a directed graph. Let $Q \subseteq V$ be a set of vertices. We say that Q is a *clique* when we have $(u, v) \in E$ and $(v, u) \in E$, for every $u, v \in Q$, with $u \neq v$.

As an easy example, we see that the graph G in Figure 2.1 is a clique.

Since we are interested in *cycles* in directed graphs, we introduce two related definitions.

Definition 2.7. A cycle $C = (v_1, \ldots, v_k)$ in a directed graph $G = (V, E)$ is an ordered sequence of vertices such that $(v_i, v_{i+1}) \in E$ for every $i \in [k]$ and $(v_k, v_1) \in E$, with $k \geq 1$. Furthermore, we write $v_i \in C$ if the cycle C uses v_i somewhere along its path. A cycle is called *simple* if the cycle consists of k distinct vertices, i.e., each $v_i \in C$ occurs exactly once in C.

Definition 2.8. A directed graph $G = (V, E)$ is *acyclic* if there does not exist a cycle C in G.

Finally, we define a *strongly connected component* of a directed graph, and we define a *cut*. We will later see that it is useful to compute cuts usin Finally, we define a *cut*.

Definition 2.9. Given a graph G, a *partition* of G is a pair (X, Y) consisting of two sets of vertices $X \subseteq V(G)$ and $Y \subseteq V(G)$ such that $X \cup Y = V(G)$ and $X \cap Y = \emptyset$.

Definition 2.10. Given a graph G and a partition (X, Y), a *cut* $(X, Y)_G$ is the set of edges

$$(X, Y)_G = \{(u, v) \in E(G) \mid (u \in X \wedge v \in Y) \vee (u \in Y \wedge v \in X)\}.$$

We also write that the cut $(X, Y)_G$ is an $(X, Y)_G$*-cut*. If it is clear which graph is used, then we omit the subscript.

As an example, see Figure 2.3.

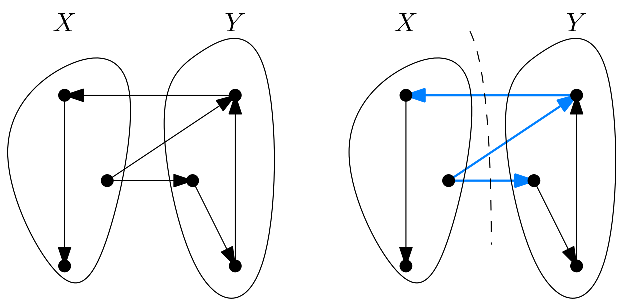

Figure 2.3: A directed graph $G = (V, E)$. The cut between the partition (X, Y) is exactly the set of blue edges, shown in the right-hand side.

The Directed Feedback Vertex Set Problem Let $G = (V, E)$ be a simple directed graph. The Directed Feedback Vertex Set problem asks for a minimum-sized set $X \subseteq V$ such the induced graph $G[V \setminus X]$ is acyclic. We will refer to X as being an optimal "solution" or optimal DFVS (short for Directed Feedback Vertex Set).

2.2 Fixed-Parameter Tractability

The field Fixed-Parameter Tractability has been focusing on the discovery of efficient algorithms when we are also given a parameter that captures the hardness of a decision problem, called a *parameterized* problem. Its definition is as follows.

Definition 2.11. A *parameterized problem* is a language $L \subseteq \Sigma^* \times \mathbb{N}$, where Σ is a fixed, finite alphabet. For an instance $(x, k) \in \Sigma^* \times \mathbb{N}$, k is called the parameter.

As a simple example, for the parameterized variant of the Directed Feedback Vertex Set problem, the Parameterized Directed Feedback Vertex Set problem, we are asked whether there exists a DFVS X of a directed graph $G = (V, E)$ such that $|X| \leq k$, given a (fixed) parameter $k \in \mathbb{N}$. Related to parameterized problems, we have the complexity class of Fixed-Parameter tractable problems.

Definition 2.12. A parameterized problem $L \subseteq \Sigma^* \times \mathbb{N}$ is called *fixed-parameter tractable* (FPT) if there exists an algorithm $\mathcal{A}$ (called a *fixed-parameter tractable algorithm*), a computable function $f : \mathbb{N} \to \mathbb{N}$, and a constant c such that, given $(x, k) \in \Sigma^* \times \mathbb{N}$ the algorithm $\mathcal{A}$ correctly decides whether $(x, k) \in L$ in time bounded by $f(k) \cdot |(x, k)|^c$. The complexity class containing all fixed-parameter tractable problems is called FPT.

Note that for an FPT problem, by fixing the constant k, we obtain a polynomial time algorithm in terms of the input size, since f only depends on

k. However, $f(k)$ will usually be exponential in k, especially considering NP-hard problems. Furthermore, not every parameterized problem is in FPT (unless P = NP). One such problem is the k-Coloring problem, where we aim to decide whether an undirected graph admits a coloring of at most k colors, since otherwise 3-Coloring, an NP-complete problem, admits a polynomial time algorithm.

2.3 Reduction Rules

In the construction of FPT algorithms, *data reductions* play an important role. A *data reduction*, or simply a reduction rule, transforms an instance (I, k) of a parameterized problem Q into an *equivalent* instance (I', k') of Q in polynomial time of $|I|$. Two instances (I, k) and (I', k') are *equivalent* if and only if $(I, k) \in Q \Leftrightarrow (I', k') \in Q$. A reduction rule that transforms one instance (I, k) into an equivalent instance (I', k') is called *safe.*

Using reduction rules, we can design a *preprocessing algorithm*, called a *kernelization*, that reduces the size of the input instance as much as possible. Hopefully, the instance-size has decreased sufficiently enough that an exact algorithm, like a Branch & Bound algorithm, now terminates in reasonable time. We say that the resulting instance obtained from the kernelization is a *kernel* of the original instance, or simply *a kernel.* Alternatively, we may also refer to the kernel as the *reduced instance*, or the *reduced graph*, when talking about graphs.

2.3.1 Towards Optimization Problems

Thus far, we have only discussed what reduction rules are for a decision problem, but since we are working in the context of an optimization problem, we need to slightly modify the definition of a safe reduction rule. Note that for an optimization problem, we want to guarantee that we can recover an optimal solution of the original instance, but it can be one of many optimal solutions. With this in mind, we can formulate our definition in the context of the Directed Feedback Vertex Set problem. We first define the notion of a reduction rule for the Directed Feedback Vertex Set problem that produces a set of *forced vertices.*

Definition 2.13 (Optimization Reduction Rule)**.** Given a directed graph $G = (V, E)$, Let ϕ be a reduction rule that produces a modified graph $G = (V', E')$, with $V' \subseteq V$, and a set $S \subseteq V$ such that $S \cap V' = \emptyset$. We say that S is a set of *forced vertices* if there exists an optimal solution Z of G such that $S \subseteq Z$.

Intuitively speaking, the set of forced vertices is a set of vertices for which we can decide that this set must belong to at least one optimal solution of the input graph, i.e., vertices with a self-loop in non-simple directed graphs must occur in an optimal solution. We can now define a *safe* reduction rule for the DIRECTED FEEDBACK VERTEX SET problem.

Definition 2.14. Given a directed graph $G = (V, E)$, Let ϕ be a reduction rule that produces a modified graph $G = (V', E')$ and a set of forced vertices S. Let Z be any optimal solution of G'. We say that ϕ is *safe* if $Z \cup S$ is an optimal solution of G.

Observe that it is not sufficient to require that $\text{OPT}(G) = \text{OPT}(G') + |S|$, where $\text{OPT}(\,\cdot\,)$ denotes the size of an optimal DFVS of a graph, namely, we do not have the guarantee that an optimal solution Z for G', together with S, yields an optimal solution of G. Using Definition 2.14 we can show that the composition, i.e., the sequential application, of two reduction rules results in a safe reduction rule.

Theorem 2.1. *Let G be a directed graph. Let ϕ_1 and ϕ_2 be two safe reduction rules. Then the composition $\phi_1 \parallel \phi_2$ is safe.*

Proof. Let G_1 and S_1 be the resulting graph and set of forced vertices after applying ϕ_1 on G, respectively. Similarly, let G_2 and S_2 be the resulting graph and set of forced vertices after applying ϕ_2 on G_1, respectively.

Since ϕ_2 is safe, it follows that an optimal DFVS Z of G_2, together with S_2 is an optimal solution of G_1, or simply put, $Z \cup S_2$ is an optimal solution of G_1. Clearly, since $Z \cup S_2$ is an optimal solution of G_1, and since ϕ_1 is safe, it follows that $Z \cup S_2 \cup S_1$ is an optimal solution of G. □

As a consequence, we can safely compose any sequence of safe reduction rules, transforming G into a reduced graph G' and collecting a set of forced vertices S. We can then recover an optimal solution for G', by finding an optimal solution Z of G', and returning $Z \cup S$.

Furthermore, while we may be able to exploit the parameter of a parameterized problem to design reduction rules, we obviously lack this parameter in an optimization problem. One possible solution is to compute an upper bound of a DFVS of G, but we will simply only cover *parameter-free* reduction rules, e.g., vertices without edges can safely be removed in the graph without any knowledge of a parameter.

2.4 Exact Algorithmic Techniques

In order to evaluate the effectiveness of reduction rules in exact algorithmic techniques, we obviously require an exact algorithm where we integrate reduction rules. In this thesis we will be using two such exact algorithmic techniques, namely Integer Linear Programming, for which there exists a large number of commercial and open source solvers, and the paradigm known as Branch & Bound. We briefly discuss the fundamentals of both approaches.

2.4.1 Integer Linear Programming

In an integer linear program, we aim to minimize or maximize a linear objective function, subject to a set of linear constraints, under the additional constraint that every variable is integer valued. For the Directed Feedback Vertex Set problem, we require that all variables are either 0 or 1 valued.

Let $G = (V, E)$ be a directed graph. For each $v_i \in V$, let x_i denote whether v_i belongs to the optimal solution. Let Γ be the set of all simple cycles of G (not necessarily ordered). Firstly, we want to minimize the number of vertices in the solution S. Secondly, for every simple cycle, we want to adopt at least one of its vertices into the solution, breaking the cycle in $G[V \setminus S]$. Thus, for every cycle in Γ, we can easily construct a constraint, namely we take the sum over all variables whose vertex is in the cycle and require that this sum is at least equal to 1. The integer linear program for Directed Feedback Vertex Set can then be formulated as follows.

$$\begin{aligned}
\text{Minimize:} \quad & \sum_{i=1}^{|V|} x_i \\
\text{Subject to:} \quad & \sum_{v_i \in C} x_i \geq 1 \qquad && \forall_C : C \in \Gamma \\
& x_i \in \{0, 1\} && \forall_i : v_i \in V.
\end{aligned}$$

We can then solve the integer linear program, and extract the optimal solution of G by computing $X = \{v_i \mid x_i = 1\}$. One immediate problem with this formulation is that the set Γ may be too large to compute. Consider the directed graph $K_n = (V, E_n)$ where $V = \{v_1, \ldots, v_n\}$ and for every $u, v \in V$ we have the edge $(u, v) \in E_n$. Observe that for every subset $C \subseteq V$, we have $|C|!$ permutations of the same vertex set. Hence, we need to find a smaller set of suitable cycles to use instead of Γ. In Chapter 4 we discuss how we improve on our choice of cycles to obtain an optimal solution more efficiently.

2.4.2 Branch & Bound

The idea of a Branch & Bound algorithm is to create an exhaustive search tree to find the optimal solution for some optimization problem. There are two parts to any Branch & Bound algorithm, firstly, the "Branch" part is considered as exhausting all possible search possibilities per *item*, e.g. vertices, and solving the resulting subproblem after *choosing* said item. From each branch, we obtain a candidate solution (possibly empty), and using our previous choices, we can recover a solution for the subproblem of the respective branch. In the end, the overall optimal solution is collected.

In the context of the Directed Feedback Vertex Set problem, one possible Branch & Bound formulation is branching on vertices, and deciding whether the selected vertex belongs or does not belong to the solution. For the former, we remove the vertex from the graph, and for the latter, we require the vertex to be in the graph. We then find the smallest set of vertices to make the two resulting graphs acyclic, or None, if there does not exist such a set of vertices. Note that for the starting graph G there always exists a solution (removing all vertices from G), but that in subsequent branches, we may accidentally require two or more vertices that have a cycle to never be removed, and thus no solution exists. After collecting both results, recall that we included one vertex in the solution. We therefore include that vertex to the solution of its respective branch, and the smallest of the two solutions is then returned, or we return None if both branches yielded None.

Pruning It is clear that the size of the search space in the above example is already 2^n. To explore fewer parts of the search tree, also known as *pruning* the search space, we compute a lower bound ℓ of the number of vertices in an optimal DFVS of G, and a good upper bound M of G. We can use the upper bound as a budget k: in each branch we are looking for a solution of size at most k, if it exists. Each time we make a choice to include a vertex in the solution in a branch, we decrease k by 1.

Clearly, for any part of the search tree for which the subproblem (a directed graph) contains a cycle and the remaining budget is 0, we can stop including additional vertices in the solution, because we are not going to find a solution that is smaller than M if we collect all the vertex choices up to the root of the search tree.

Given a lower bound ℓ, we may already preemptively decide that a particular branch requires us to include more vertices in the solution than we have budget for, this prevents exploring additional parts of the search tree. It is therefore important to ensure that the lower bound and upper bound are as tight as possible. The earlier we decide that we exceed our budget the more we can prune from the search tree.

Branch & Reduce To further attempt to speed up a Branch & Bound, we can integrate reduction steps into the algorithm, obtaining what is known as a Branch & Reduce algorithm. Reducing has two possible advantages, firstly, we hopefully shrink the size of the graph, and therefore of the search space, and secondly, it allows us to further prune branches. At the very start, we reduce our directed graph, and obtain a set of forced vertices in our solution S. If $|S|$ exceeds our budget, we can cut off the branch, since

$|S|$ is a lower bound. If $|S|$ does not exceed our budget, we reduce our budget by $|S|$, and proceed as normal. In some cases however, reducing is not guaranteed to speed up a Branch & Bound [11].

2.5 Simulated Annealing

As mentioned above, we can use a *heuristic* to improve the performance of both an ILP and a Branch & Bound algorithm. A heuristic is a method of exploring the solution search space of an optimization problem, exploiting specific aspects of the problem. Additionally, heuristics do not have quality guarantee of the output, but we hope that the output is of very good quality. Furthermore, we hope that the computational effort of finding a good solution is low.

There are many approaches to construct a heuristic for an optimization problem, such as a greedy algorithm, a local search, or a simulated annealing algorithm. We focus on a simulated annealing algorithm, given its large flexibility to tune its running time, but possibly at the cost of obtaining worse solutions. However, we first need to discuss some terminology, and what a local search is, since a simulated annealing builds on top of a local search.

We first define a *search space* S as the set of all possible solutions for a given problem. For the Directed Feedback Vertex Set problem, we have that S contains all directed feedback vertex sets of a given directed graph G. Additionally, we are given an objective function $f : S \to \mathbb{R}$ that we aim to minimize or maximize, which may for instance be the size of the solution. For each solution $\mathbf{x} \in S$, we can define a neighborhood $N(\mathbf{x})$ of solutions that are *reachable* from $\mathbf{x}$, i.e., we can transform, $\mathbf{x}$ into a solution $\mathbf{y} \in N(\mathbf{x})$ according to some criteria. Starting from an initial solution $\mathbf{x}_0$, we move to a next possible solution $\mathbf{x}_1$, and continue moving towards subsequent solutions, until some stopping criterion is met.

The key idea of a *local search* is to define a small neighborhood for each solution with respect to size of the search space. For example, one possible neighborhood definition is the set of all directed feedback vertex sets differing only by one vertex, given some initial solution Z.

We can now discuss what a simulated annealing algorithm is. A simulated annealing algorithm aims to minimize a given objective function. As explained above, we start with an initial solution and define an appropriate objective function, and we define how we move from one solution to the next. Like in a local search, we define a small neighborhood, as to how we move from one solution to another solution. If this new solution is a *better* solution, e.g., a smaller solution in the context of the Directed Feedback Vertex Set problem, it is immediately accepted.

The key idea to a simulated annealing algorithm is that worse moves may be accepted, i.e., we replace our current solution with a worse solution, with a small probability, depending on some *temperature* $T \in \mathbb{R}$.[1] Let the *energy variation* δ be defined as $\delta := f(\mathbf{x}') - f(\mathbf{x})$, when we move from one solution $\mathbf{x}$ to a new solution $\mathbf{x}'$. The *acceptance probability* is then given by

$$p = \min(1, e^{-\delta/T}), \tag{2.1}$$

which is also known as the *Metropolis Rule*. Observe that as the temperature T decreases, so does the acceptance probability p.

Cooling scheme During the simulated annealing algorithm, we aim to decrease the temperature to reduce the probability of moving towards a worse solution. This method of changing the temperature is called a *cooling scheme*. There four important decisions to be made when designing the cooling scheme, namely:

1. the starting temperature,
2. when to decrease the temperature,
3. how to decrease the temperature,
4. and when we stop the search.

Firstly, the starting temperature must be high enough such that the search initially resembles a *random search*, i.e., randomly sampling another solution from the search space. Additionally, we want to be able to reach all solutions. However, the starting temperature must not be too high, otherwise we perform a random search for too long.

Secondly, there are two options when we decrease the temperature. For the first option, we can decrease the temperature after every iteration, giving us great flexibility in controlling the running time of the simulated annealing algorithm, at the cost of possibly worse solutions. For the other option, we can decide to decrease the temperature after an *equilibrium* is reached, i.e., we are no longer able to accept a new solution, or every after attempted move.

Thirdly, there are two common ways how we can decrease the temperature. We can either use a linear law $T_{i+1} = T_i - \alpha$, or a geometric law $T_{i+1} = \alpha \cdot T_i$, where i denotes the iteration number and α is a constant. Observe that the constant α controls how quickly the temperature decreases, which brings us to the final point.

Finally, we need to ensure that the temperature is low enough when we stop searching, such that near the end of the search, the search resembles *random iterative improvement*, i.e., only accepting random local moves that

[1]The term *simulated annealing* has its origins in metallurgy; the process of *annealing* is a heat treatment.

improve the current solution. One option is to specify the end temperature by choosing a final acceptance probability p_e, determining the smallest cost difference $(\Delta f)_{\min}$, if it exists, and setting

$$T_e = \frac{(\Delta f)_{\min}}{\ln p_e},$$

and choosing α accordingly for both the linear law and the geometric law. Observe that indeed

$$e^{(\Delta f)_{\min}/T_e} = e^{(\Delta f)_{\min}/((\Delta f)_{\min}/\ln p_e)} = e^{\ln p_e} = p_e.$$

On the other hand, we can stop when no improvement has occurred in a significant number of iterations, or a good number of equilibriums have been reached.

Concluding, there are four important components of a simulated annealing algorithm. Namely,

1. defining an appropriate objective function,
2. choosing the initial solution,
3. defining how we move from one solution to the next,
4. and finally, choosing the cooling scheme.

We will discuss all these points in more detail in Chapter 4.

Chapter 3

Kernelization

In this chapter we discuss the reduction rules we will be using to reduce DIRECTED FEEDBACK VERTEX SET instances. Firstly, we discuss the equivalence of finding a DFVS on bidirected graphs and the solving VERTEX COVER problem on its induced undirected graph. We then discuss some simple reduction rules for the DIRECTED FEEDBACK VERTEX SET problem. Using a modified version of these reduction rules, we can split a directed graph $G = (V, E)$ into a monodirected graph $G_d = (V_d, E_d)$ and a bidirected graph $G_b = (V, E_b)$, with $V_d \subseteq V$, such that an optimal solution for the graph $G_c = G_b + G_d$, together with a forced set of vertices S, is an optimal solution of G. After we have split G, we can apply a vertex cover solver on the induced undirected graph $\mathcal{U}(G_b)$ to hopefully obtain a DFVS of G_d. Finally, we conclude with a brief discussion of possible extra reduction rules that could have been implemented.

3.1 Relation to Vertex Cover

In the VERTEX COVER problem, we are given an undirected graph $G = (V, E)$, and asked to find a minimum-sized set $X \subseteq V$ such that for every $(u, v) \in E$ we have $u \in X$ or $v \in X$.

When $G = (V, E)$ is bidirected, we can prove that an optimal DFVS of G is an optimal vertex cover of $\mathcal{U}(G)$, and vice versa. We will be using this property frequently going forward, and it admits the use of reduction rules for the VERTEX COVER problem in this specific situation. We first make the following observation.

Lemma 3.1. *Let $G = (V, E)$ be a directed graph. Let $G' := \mathcal{U}(G)$ be its induced undirected graph. A vertex cover S of G' is always a DFVS of G, but not necessarily optimal.*

Proof. Since S is a vertex cover of G', this means that the graph $G'[V \setminus S]$ has no edges. Since for every edge $(u, v) \in E$, we have that G' has an undirected edge (u, v), it follows that the graph $G[V \setminus S]$ also has no edges, and thus S is a DFVS of G, but not necessarily optimal. □

Theorem 3.1. *Let $G = (V, E)$ be a bidirected graph and let $G' := \mathcal{U}(G) = (V, E')$ be the induced undirected graph obtained from G. We have that S is an optimal DFVS of G if and only if S is an optimal vertex cover of G'.*

Proof. ($\Rightarrow$) Suppose S is an optimal DFVS of G. Since for every $(u, v) \in E$, we have $(v, u) \in E$, i.e. a cycle consisting of u and v, it must follow that either $u \in S$ or $v \in S$. Hence by construction, S must be a vertex cover of G'.

Suppose, towards a contradiction, that S is not an optimal vertex cover of G. As per the above argument, S is indeed a vertex cover, so it must be that there is a vertex cover S' of G' such that $|S'| < |S|$. We know that S' is also a DFVS by Lemma 3.1, but then S was not an optimal DFVS of G, which is a contradiction.

($\Leftarrow$) Suppose S is an optimal vertex cover of G'. Analogous to previous argument, for every $(u, v) \in E'$, either $u \in S$ or $v \in S$. Since we only have edges between u and v in G if they have an undirected edge in G', it must follow that $G'[V \setminus S]$ has no edges. Hence, S is a DFVS of G. Note that S is also an optimal solution, since any DFVS of G must include at least one of each endpoint for each edge. $\square$

3.2 Reduction Rules

For the reduction rules, we use parameter-free reductions commonly found in the literature [1, 10]. Due to the reduction rules G can become non-simple, i.e., self-loops or parallel edges are introduced. We can easily prevent inserting parallel edges, so we treat this as an extra step in the following reduction rules, if applicable. The reduction rules are to be applied exhaustively, i.e., until no reduction rule can make any progress in removing vertices or edges. Our first reduction rule is based on strongly connected components.

Reduction 3.1. *Let $\mathcal{C} = \{C_1, \ldots, C_k\}$ be the strongly connected components of a directed graph $G = (V, E)$. For every $1 \leq i \leq k$, remove all edges in the cut $(C_i, V \setminus C_i)_G$. If $C_i = \{v\}$ and v does not have a self-loop, remove v from G. The set of forced vertices $S = \emptyset$.*

Before proving the correctness of this reduction rule, we first show that only cycles exist within strongly connected components, that is, no cycle traverses more than one strongly connected component.

Lemma 3.2. *Given a directed graph G, let R be a cycle. Then there exists exactly one strongly connected component C of G such that $R \subseteq C$.*

Proof. Suppose, towards a contradiction, that there exist at least two strongly connected components C_1 and C_2 such that $R \subseteq C_1$ and $R \subseteq C_2$ with $C_1 \neq C_2$. Let $u \in C_1$, and let $v \in C_2$. Since R is a cycle, and since C_1 and

C_2 are strongly connected components, there exists a path from u to v, and vice versa. Observe then that C_1 is not a largest set, which is required for C_1 to be a strongly connected component, which is a contradiction, so there exists at most one strongly connected component C of G, such that $R \subseteq C$.

Clearly, since R is a cycle, there must exist a strongly connected component containing all vertices of R. This concludes the proof. □

We can now prove that Reduction 3.1 is safe.

Lemma 3.3. *Reduction 3.1 is safe.*

Proof. Let $G = (V, E)$ be the original graph, and let $G' = (V', E')$ be the resulting graph after the reduction. It remains to show that an optimal solution of G' is an optimal solution of G. Let X be an optimal DFVS of G, and let Z be an optimal DFVS of G'.

We can almost directly show that Z is an optimal solution of G, by showing that $|X| = |Z|$, and that Z is indeed a DFVS of G, proving our claim. Observe that G' is a subgraph of G, since we only remove edges and vertices of G to obtain G'. Hence, we have that $|X| \geq |Z|$.

It remains to show that Z is a DFVS of G, immediately proving $|Z| \geq |X|$ and the claim. Suppose, towards a contradiction, that Z is not a DFVS of G. Then there exists a cycle M in $G[V \setminus Z]$. By Lemma 3.2, M is contained in a single strongly connected component C of G (with $|C| \geq 2$), or $M = \{v\}$. By construction, if $M = \{v\}$, then v has a self-loop, and so $G[\{v\}]$ is a subgraph of G'. In the second case, we also have that $G[C]$ is a subgraph of G'. So for both cases, $G'[V' \setminus Z]$ must then contain the cycle M, which is a contradiction. Hence, Z is a DFVS of G, and since Z is not necessarily an optimal DFVS of G, it follows that $|X| \leq |Z|$. This concludes the proof. □

The following three reduction rules are found in literature and are frequently used [1, 10]. Observe that Reduction 3.2 produces a set of forced vertices. When applying all the reduction rules exhaustively, we can easily compute the union of all these forced sets of vertices, and return it as a single set.

Reduction 3.2. *If $v \in V$ contains a self-loop, remove v with all its incident edges from G and include v in the set of forced vertices S.*

Reduction 3.3. *If $N^+(v) = \{u\}$ for some $v \in V$ with $u \neq v$, then redirect all vertices in $N^-(v)$ to u and remove v from G. The set of forced vertices $S = \emptyset$.*

Reduction 3.4. *If $N^-(v) = \{u\}$ for some $v \in V$ with $u \neq v$, then redirect u to all vertices in $N^+(v)$ and remove v from G. The set of forced vertices $S = \emptyset$.*

It is straightforward to prove the safety of these three reduction rules. Obviously, vertices with a self-loop must be included in any optimal solution. And for the other two reduction rules, observe that if there is a cycle going through v, then the same cycle must also use u, hence, we can safely remove v from the graph, and redirected its edges.

We can integrate these reductions in a *kernelization algorithm*. Each time we apply a reduction on G, instead of creating a new graph G', we simply make the changes on G directly. By the end of the algorithm, G is the reduced graph G'. We also test whether some reduction rules are *applicable*. This simply amounts to checking whether the conditions for a reduction rule are met, e.g., verifying whether there exists a vertex $v \in V$ such that v has a self-loop.

3.3 Split & Reduce

In this section we discuss how we can split and reduce a directed graph $G = (V, E)$ into a monodirected graph G_d and a bidirected graph G_b such that we can still recover an optimal solution for G. Let $G' = (V', E')$ be the resulting graph after exhaustively applying all the reductions, and let S be the set of forced vertices. Observe that Reductions 3.3 and 3.4 introduce additional edges, hence, G' may contain significantly more bidirectional edges compared to G. Our goal is to exploit this property, using, what we call, a *splitting reduction*.

The goal of the splitting reduction is to further reduce G', and obtain two graphs, a monodirected graph $G_d = (V_d, E_d)$ and a bidirected graph $G_b = (V_b, E_b)$ from a the reduced directed graph G'. The resulting monodirected graph G_b ideally becomes completely empty, but then this means that we managed to split and reduce G' into a bidirected graph G_b, thus a vertex cover of $\mathcal{U}(G_b)$, together with all forced vertices, is an optimal DFVS of G'. We maintain that $V_d \cup V_b \subseteq V'$, but in general V_d and V_b may become completely disjoint. The construction is then as follows.

We give a schematic of the splitting reduction in Figure 3.1. We start the process from the reduced graph G'. We then create a directed graph G_b without any edges using the vertices of a directed graph G', and we then relabel G' to $G_d = (V_d, E_d)$ for convenience. Then, for each pair of bidirectional edges $(u, v) \in E_d$ and $(v, u) \in E_d$, we include these edges in E_b, and we remove these edges from E_d, and the endpoints u and v are included in the set of *restricted* vertices R. For restricted vertices, only Reductions 3.1 and 3.2 are applicable. Essentially, we restrict the above given reductions to a smaller set of vertices. We call these *restricted reductions* for clarity. We then further reduce G_d using these restricted reductions. We will later see that we can also reduce G_b using these restricted reductions. The set of forced vertices is collected, and the process of finding all bidirectional edges

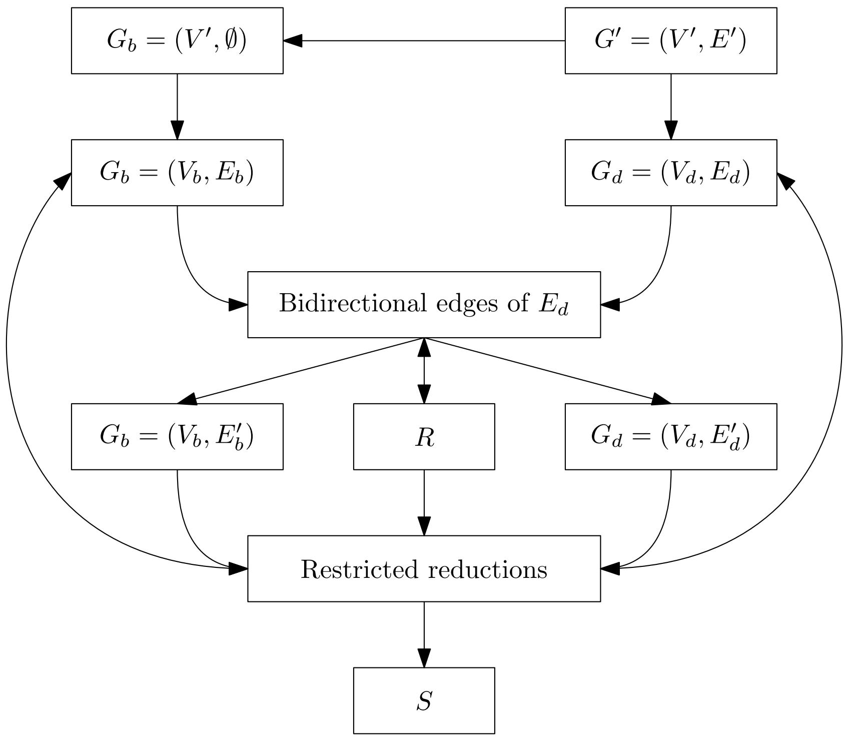

Figure 3.1: Schematic of the splitting reduction.

of G_d can start again. When G_d no longer has any bidirectional edges, we stop. In the end, G_b is a bidirected graph since we included only bidirectional edges.

We can formalize the restricted reductions as follows. The first reduction is not per se a reduction, but it allows us to prove that we can move bidirectional edges from a directed graph G_d to another bidirected graph G_b. We will not explicitly use this reduction, but we will integrate it in the algorithm to compute the bidirected graph G_b.

Reduction 3.5. *If $e = (u, v) \in E(G_d)$ is a bidirectional edge with $e' = (v, u)$, then remove e and e' from G_d, add u and v to G_b, and insert e and e' in G_b. Include u and v to the set of restricted vertices R.*

Reduction 3.6 (Restricted Reduction 3.1)**.** *Let $\mathcal{C} = \{C_1, \ldots, C_k\}$ be the strongly connected components of G_d. For every $1 \leq i \leq k$, remove all edges in the cut $(C_i, V \setminus C_i)_{G_d}$. If $C_i = \{v\}$ and v does not have a self-loop, remove v from G.*

Reduction 3.7 (Restricted Reduction 3.2)**.** *If $v \in V(G_d)$ contains a self-loop, remove v with all its incident edges from G_d and G_b and include v in the solution S.*

Algorithm 1 Reduce & Split

```
Require: A directed graph $G = (V, E)$.
 1: procedure SPLITTING($G$)
 2:     $(G', S) \leftarrow$ REDUCE($G$)
 3:     $G_b = (V, E_b)$ with $E_b \leftarrow \emptyset$
 4:     $R \leftarrow \emptyset$
 5:     while $G'$ contains bidirectional edges do
 6:         for each pair of bidirectional edges $(u, v)$ and $(u, v)$ of $G'$ do
 7:             $E_b \leftarrow E_b \cup \{(u, v), (v, u)\}$
 8:             Remove the edges $(u, v)$ and $(v, u)$ from $G'$
 9:             $R \leftarrow R \cup \{u, v\}$
10:         $(G', G_b, S') \leftarrow$ RESTRICTEDREDUCE($G', G_b, R$)
11:         $S \leftarrow S \cup S'$
12:     return $(G', G_b, S)$
```

Reduction 3.8 (Restricted Reduction 3.3)**.** *If $N^+_{G_d}(v) = \{u\}$ for some $v \in V(G_d)$ with $u \neq v$ and $v \notin R$, then redirect all vertices in $N^-_{G_d}(v)$ to u and remove v from G_d.*

Reduction 3.9 (Restricted Reduction 3.4)**.** *If $N^-_{G_d}(v) = \{u\}$ for some $v \in V$ with $u \neq v$ and $v \notin R$, then redirect u to all vertices in $N^+_{G_d}(v)$ and remove v from G_d.*

Since we restricted the reductions to split the graph, it therefore makes sense to only apply the splitting reduction after we have exhaustively applied the nonrestricted reductions first, followed by splitting the graph. Putting it all together, to the original input graph G into a monodirected graph and a bidirected graph, we use Algorithm 1.

Restricted Reductions While this may seem to heavily restrict the proceeding reduction process, this is actually not the case. We can prove that restricted vertices can still be removed from the graph if they either belong to a strongly connected component consisting of only itself, or these vertices can be adopted into the solution because they have a self-loop. The latter is initially impossible, since the graph is exhaustively reduced, but the former may kickstart the reduction process.

The reason why we restrict the number of applicable reduction rules is because moving the bidirectional edges over to G_b is not a safe reduction if we only consider the remainder of G'. Consider the directed graph G in Figure 3.2. Here, the bidirectional edges of v are removed, and v and v' marked as restricted and depicted in red. According to Reduction 3.4 or Reduction 3.3, we can redirect the edge from (w, v) to (w, u) and remove v. We can do apply the same reduction to v', obtaining the graph shown

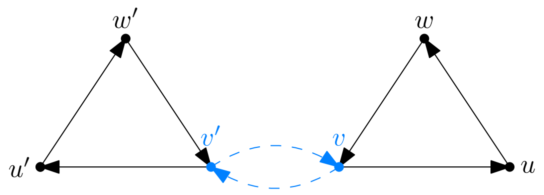

Figure 3.2: A directed graph G where we remove and remember the directional edges (v, v') and (v', v).

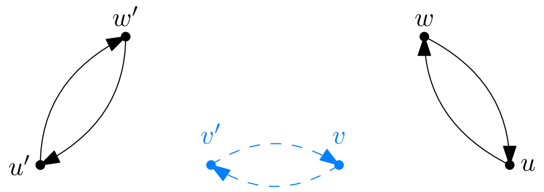

Figure 3.3: The resulting graph after applying Reduction 3.4 on v and v'.

in Figure 3.3, but recall that we still need to include either v' or v in our solution to break that cycle. Note that we now require at least 3 vertices in our solution, whereas the optimal solution contains exactly 2 vertices, since there are two independent cycles that can be broken with v' and v in the solution.

Correctness Since we split G' into two graphs G_b and G_b, we need a slightly different definition to prove that the performed reduction rules are safe.

Definition 3.1. Let ϕ be a reduction rule that takes a directed graph $G_d = (V_d, E_d)$ and a bidirected graph $G_b = (V_b, E_b)$ and produces modified graphs $G'_d = (V'_d, E'_d)$ and $G'_b = (V'_b, E'_b)$, and a set of forced vertices $S \subseteq V_d$ with $S \cap V'_d = \emptyset$ and $S \cap V'_b = \emptyset$. Let Z be an optimal solution of $G'_d + G'_b$. We say that ϕ is *safe* if $Z \cup S$ is an optimal solution of $G_d + G_b$.

Initially, we set $G_b = (V, \emptyset)$, and set $G_d = G'$, where G' is the graph after exhaustively applying Reductions 3.1, 3.2, 3.3 and 3.4. We can now prove the correctness of the restricted variants.

Lemma 3.4. *Reduction 3.5 is safe.*

Proof. Let $e = (u, v) \in E$ be a bidirectional edge with $e' = (v, u) \in E$ in the graph G_d. Let G'_d be the graph with e and e' removed, and let G'_b be the graph obtained from inserting the bidirectional edges (u, v) and (v, u) in G_b. Observe that $G_d + G_b = G'_d + G'_b$. Hence, an optimal solution Z of $G'_d + G'_b$ is an optimal solution of $G_d + G_b$, that is, $S = \emptyset$. □

Before we prove the correctness of Reduction 3.6 we make a few observations. A strongly connected component C of G_d may belong to a larger

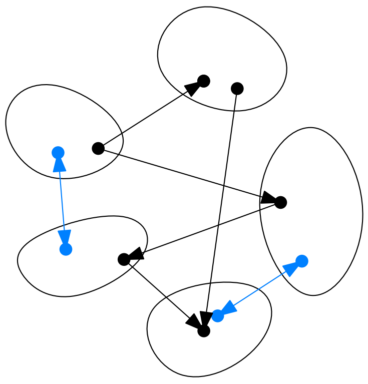
Figure 3.4: Five strongly connected components of the graph G_d that belong to the same strongly connected component in the graph $G_d + G_b$. Not all vertices are drawn.

strongly connected component from $G_d + G_b$ because we now include the bidirectional edges, see Figure 3.4. Furthermore, after we cut the black edges from the graph shown in Figure 3.4, we obtain three strongly connected components in the reduced graph $G'_d + G'_b$. Observe that still one strongly connected component of G_d is either a strongly connected component of $G'_d + G'_b$, or it is part of a strongly connected component of $G'_d + G'_b$.

Lemma 3.5. *Reduction 3.6 is safe.*

Proof. Let X be an optimal solution of $G_c := G_d + G_b$, and let Z be an optimal solution of $G'_c := G'_d + G'_b$. Observe that G'_c is a subgraph of G_c, since we only remove edges and vertices. Hence, X must a DFVS of G'_c, and so $|X| \geq |Z|$, since X is not necessarily an optimal solution of G'_c.

It remains to show that Z is a DFVS of G_c. We first observe that $G_b = G'_b$, since we only cut edges in the graph G_d to obtain G'_d. Suppose, towards a contradiction, that Z is not a DFVS of $G_c = (V_c, E_c)$, then there must exist a cycle L in the graph $G_c[V_c \setminus Z]$. Then there is a strongly connected component C such that $L \subseteq C$.

Observe that Z definitely breaks all the cycles in the strongly connected components of G_d, since each such strongly connected component may belong to a larger strongly connected component in G'_c. Hence, L must traverse over multiple strongly connected components of $G_d[L]$. We observe that L traverses over at least one pair of vertices (u, v) with $u, v \in R$ such that both $e = (u, v)$ and $e' = (v, u)$ are edges of G_b, and therefore also G'_b. It cannot be the case that both e and e' are not edges of G'_b and G_b, then e and e' must be edges of G_d, and so Z must break this cycle. Since Z is at least a vertex cover of G'_b and G_b, and so Z also breaks the cycle between u and v in G_b. Hence, R cannot possibly use u or v, and so R is not a cycle in $G_c[C]$. Which is a contradiction. Therefore, Z is a DFVS of G_c, and so

$|Z| \geq |X|$, since Z is not necessarily an optimal solution of G'_c. It follows that $|Z| = |X|$, and since Z is a DFVS of G_c, Z must be an optimal DFVS of G_c. $\square$

Lemma 3.6. *Reduction 3.7 is safe.*

Proof. Let $v \in V(G_d)$ be a vertex with a self-loop. Let X be an optimal solution of $G_d + G_b$, and let Z be an optimal solution of $G'_d + G'_b$, with $S = \{v\}$. We first show that $|Z \cup S| = |Z| + 1 \leq |X|$. Observe that the graph $G'_d + G'_b$ is a subgraph of $G_b + G_d$, since we only removed v. Secondly, we have that $v \in X$, otherwise X is not a DFVS of $G_d + G_b$. Hence, $X \setminus \{v\}$ is a solution of $G'_d + G'_b$. We therefore conclude that

$$|Z \cup S| = |Z| + 1 \leq |X \setminus \{v\}| + 1 = |X| - 1 + 1 = |X|.$$

It remains to show that $Z \cup S$ is a DFVS of $G_c := G_d + G_b$, showing that $|Z \cup S| \geq |X|$. Let $V_c = V(G_c)$. Suppose that $Z \cup S$ is not a DFVS of G_c, and let $C = (v_1, \dots, v_k)$ be a cycle in the graph $G_c[V_c \setminus (Z \cup S)]$. Clearly $C \cap S = \emptyset$, since we removed v from both G_d and G_b, which has a self-loop. Observe that every edge used in the cycle C is present in the graph $G'_d + G'_b$ since we removed all incident edges of v in both G_d and G_b, resulting in G'_d and G'_b, respectively. Hence, Z is therefore not a DFVS of $G'_d + G'_b$, which is a contradiction. Hence, $Z \cup S$ is a DFVS of G_c. This concludes the proof. $\square$

Lemma 3.7. *Reduction 3.8 is safe.*

Proof. Let X be an optimal solution of $G_d + G_b$, and let Z be an optimal solution of $G'_d + G'_b$. Let v be a vertex such that $N^+_{G_d}(v) = \{u\}$ with $v \notin R$ in the graph G_d. Observe that $v \notin V(G_b)$, otherwise $v \in R$, and by construction, $G_b = G'_b$. Let C be a cycle in G_d such that $v \in C$. Since $v \notin R$, it must be that $N^+_{G_d}(v) = N^+_{G_d+G_b}(v) = N^+_{G_d+G'_b}(v) = \{u\}$. Since $N^+_{G_d+G'_b}(v) = \{u\}$, it follows that $u \in C$ as well. Secondly, observe that C also exists in the graph $G_d + G_b = G_d + G'_b$. By redirecting all the incoming edges from v to u, we have that $C \setminus \{v\}$ is a cycle of $G'_d + G_b = G'_d + G'_b$. We can now show that Z is an optimal solution of $G_d + G_b$.

We first show that Z is a DFVS of $G_d + G_b$. Let C be as defined above. As already mentioned, $C \setminus \{v\}$ is a cycle of $G'_d + G'_b$, so $Z \cap C \neq \emptyset$. Furthermore, any cycle C' in $G_d + G_b$ with $C' \cap \{v\} = \emptyset$ also exists in $G'_d + G'_b$. Hence, Z is indeed a DFVS of $G_d + G_b$. Hence, $|X| \leq |Z|$, since Z may not necessarily be an optimal solution for $G_d + G_b$.

To show that Z is indeed an optimal solution, we show that X is also a DFVS of $G'_d + G'_b$. Let C be a cycle in the graph $G'_d + G'_b$. If C does not exist in G_d, then we know that we need v to complete the cycle C in G_d.

Here we need a case distinction. If $v \notin X$, we already must have $C \cap X \neq \emptyset$. On the other hand, suppose that $v \in X$.

We know that $u \in C$, since $N^+_{G_d+G'_b}(v) = \{u\}$. Note that we can exchange v for u in X and still have that X is a DFVS of $G_d + G_b$ using the same number of vertices. And thus we return to the first case where $v \notin X$. Finally, if C is a cycle in $G_d + G_b$, i.e., C is a cycle that does not use v in $G_d + G_b$, then we also must have $C \cap X \neq \emptyset$. Hence, X is a DFVS of $G'_d + G'_b$, and so $|Z| \leq |X|$. This concludes the proof. □

Lemma 3.8. *Reduction 3.9 is safe.*

Proof. The proof is analogous to the proof of Lemma 3.7. □

We have now shown correctness of all the restricted variants of the reduction rules. This is sufficient to show that the splitting algorithm is correct.

Theorem 3.2. *Let S, G_d and G_b, be the set of forced vertices, the monodirected graph, and the bidirected graph after performing the splitting reduction on G, respectively. Let Z be an optimal DFVS of the graph $G_d + G_b$, then $Z \cup S$ is an optimal DFVS of G.*

Proof. Let $G' = (V', E')$ be the graph after initially reducing G using the nonrestricted reduction rules, and let S_0 be the first set of forced vertices. We have that Reductions 3.5, 3.7, 3.8 and 3.4 are safe, and so we can merge all forced vertices in a single set S' — different than the set S which we return. Since initially $G_d = G'$ and $G_b = (V', \emptyset)$. We can now trivially apply the construction of Theorem 2.1, and obtain that $Z \cup S'$ is an optimal DFVS of G'. Since all the initial reductions are safe, we obtain that $Z \cup S' \cup S_0$ is an optimal DFVS of G. Finally, we have that $S_0 \cup S' = S$, which concludes the proof. □

As mentioned above, we aim to find a vertex cover of $\mathcal{U}(G_b)$, which hopefully results in recovering an optimal DFVS of G. The following theorem proves that we can indeed do this, given that the vertex cover is a DFVS of G_d, i.e., we do not need to verify whether it is a DFVS for the graph G_d+G_b

Theorem 3.3. *Let S be set of forced vertices collected after applying Algorithm 1. Suppose Z is an optimal vertex cover of $\mathcal{U}(G_b)$, with $G_b = (V_b, E_b)$. If Z is a DFVS of $G_d = (V_d, E_d)$, then $Z \cup S$ is an optimal DFVS of G.*

Proof. The main goal of the proof is to show that Z is an optimal DFVS of the graph $G_d + G_b$, from there we can directly apply Theorem 3.2 and obtain the desired result. Let X be an optimal DFVS of the graph G_d+G_b. We show that Z is an optimal DFVS of $G_d + G_b$ by showing that $|Z| = |X|$ and that Z is a DFVS of $G_d + G_b$.

By correctness of the splitting reduction, observe that we need to break at least all the edges (which are actually cycles of two bidirectional edges) in G_b, so a vertex cover of $\mathcal{U}(G_b)$ must be a lower bound, i.e., we have $|Z| \leq |X|$.

Consider the graph $G_c = (G_d + G_b) = (V_c, E_c)$. We show that $G_c[V_c \setminus Z]$ is acyclic. Suppose, towards a contradiction, that $G_c[V_c \setminus Z]$ contains a cycle $C = (v_1, \ldots, v_k)$. Since Z is a vertex cover of G_b, it follows that $G_b[V_b \setminus Z]$ has no edges. Hence, the cycle C must use edges remaining in $G_d[V_d \setminus Z]$, but since Z is also a DFVS of G_d, there does not exist a cycle, obtaining our contradiction. This gives that $|Z|$ is in fact a DFVS for $G_d + G_b$, and so $|Z| \geq |X|$. This gives that $|X| = |Z|$. Clearly, Z is an optimal DFVS of $G_d + G_b$, and by Theorem 3.2, we obtain our desired result. $\square$

We can now construct our first algorithm to obtain a DFVS of the graph G, using the Vertex Cover solver of Hespe et al. [11], which is based on multiple phases of Branch & Reduce and Branch & Bound. If we fail to find a DFVS of G_d, we will use an integer linear program formulation, which we will discuss in the next chapter.

3.4 Extra Reduction Rules

After the conclusion of the *Parameterized Algorithms and Computational Experiments*, we were able to see which reduction rules other teams used. We give a very brief summary of possible additional reduction rules that could have been used together with our reduction rules. Several teams adapted reduction rules for the Vertex Cover problem to be suitable for the Directed Feedback Vertex Set problem — but it not quite clear how they realized their adaptations. Fellows et al. [6] give an extensive list of reduction rules for the Vertex Cover problem. For example, their reduction rule based *crown decomposition* was successfully applied to the Directed Feedback Vertex Set problem, with some modifications. A *crown decomposition* is a partition of a nonempty independent set C, called a crown, a *head* H, and a remainder R that does not share any edges between C. The crown C has a matching *into* H, i.e., the edges between C and H have a matching of size $|H|$. The crown can then be adopted into the solution, and both the head and the crown can then be removed from the graph.

Furthermore, Reduction 7 of Fellows et al. [6] was also successfully applied. The reduction rule states that, given an undirected graph G and two adjacent vertices u and v, if $N(u) \subseteq N[v]$, then u can be adopted into the solution and can be removed from G with all its incident edges.

Besides adapting reduction rules for the Vertex Cover problem, some teams also managed to adapt reduction rules for the Maximum Independent Set problem. The Maximum Independent Set problem can be seen as the complement of the Vertex Cover problem: if $S \subseteq V$ is a maximum *independent set*, i.e., all vertices of S do not share any edges, of an undirected graph $G = (V, E)$, then $V \setminus S$ is a minimum vertex cover.

At least one team managed to reduce cliques using a reduction rule from Butenko et al. [14].

Chapter 4

Integer Linear Programming

In the previous chapter we explained how we reduce a directed graph G into a monodirected graph G_d and a bidirected graph G_b. If an optimal vertex cover of G_b is not a DFVS of G_d, we need a different approach to solving the problem exactly. As the title of the chapter suggests, we will be using a standard Integer Linear Programming formulation, as explained in Chapter 3. In this chapter, we discuss how we generate the cycles which we will be using in the ILP, as well as how we try to improve the performance of the ILP by generating additional constraints.

4.1 Edge Cycle Covers

Let us first recall the Integer Linear Programming (ILP) formulation from Chapter 3. Given the set of all simple cycles Γ of a directed graph $G = (V, E)$, we can find an optimal DFVS of G by solving the following ILP formulation:

$$\begin{aligned}
\text{Minimize:} \quad & \sum_{i=1}^{|V|} x_i \\
\text{Subject to:} \quad & \sum_{v_i \in C} x_i \geq 1 && \forall_C : C \in \Gamma \\
& x_i \in \{0, 1\} && \forall_i : i \in \{1, \ldots, |V|\},
\end{aligned}$$

and where we collect the solution $X = \{v_i \mid x_i = 1\}$.

Unfortunately, the size of Γ may be too large to efficiently compute. Instead, we can use a *constraint generation scheme*, that is, we generate a cycle of G, solve the ILP, extracting the optimal solution X, and verify whether it results in a DFVS of G, and if not, we include another cycle disjoint from X into the same ILP, and repeat this process until X is a DFVS of G.

An immediate issue with this approach is that we need to restart the ILP a significant number of times. To remedy this, we are looking to compute a *set* of (multiple) cycles using significantly fewer cycles compared to Γ, but which is still large enough to significantly reduce the number of times we restart the ILP.

In the remainder of the section, we design an algorithm to compute a small set of cycles. Later, we discuss how we are going to use this algorithm to generate our constraints, as this algorithm may report a set of cycles that is actually too small, i.e., we do not obtain a DFVS of G after solving the ILP and extracting its solution. At the basis of our algorithm, we are going to use a set of cycles derived from an *edge cycle cover*. The resulting set of cycles will not necessarily be an edge cycle cover, however.

Definition 4.1. An *edge cycle cover* $\mathcal{C}$ is a family of cycles of a graph G such that

$$\bigcup_{C \in \mathcal{C}} E(G[C]) = E(G).$$

Observe that not every directed graph has an edge cycle cover. If a directed graph has strongly connected components with a single vertex (without a self-loop), then we cannot find a cycle that contains any of those incident edges. For our purposes, the absence of an edge cycle cover is not necessarily problem, we just need to include every edge that is part of a cycle into at least one constraint.

Fortunately, however, since we exhaustively applied all reduction rules, G_d contains only strongly connected components, and has no edges between any other strongly connected component. Hence, we can conclude the following fact.

Observation 4.1. *The monodirected graph G_d has an edge cycle cover.*

To construct an edge cycle cover, it suffices to use a trivial algorithm, which turns out to be quite efficient. For every edge (u, v), we construct a cycle of the graph by finding a (shortest) path from v to u. Observe, however, that two cycles using the same set of vertices but different edges generate exactly the same constraint. It therefore does not make much sense in using this complete edge cycle cover. From this set of cycles, we will therefore only use only cycles with a distinct set of vertices, which is therefore not necessarily an edge cycle cover.

Furthermore, we want to limit the size of the generated cycles in the edge cycle cover, so that we reduce the number of variables used in the constraints. If we find a cycle C, we attempt to find a smaller cycle C' that uses a subset of the vertices of C. In Figure 4.1, this would be the cycle from v to w, using the edge $a = (w, v)$. We can find these *shortcuts* easily by checking whether there exists an edge, from a vertex other than v and to a vertex other than w. Hence, starting from an edge cycle cover, we can

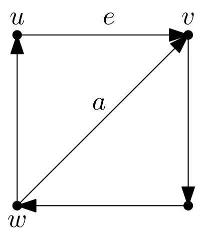

Figure 4.1: Finding a smaller cycle using the shortest path from v to u in a directed graph.

reduce the cover by trying to find these smaller cycles, if they exist, and using those cycles instead. Again, we ensure that the resulting set of cycles contains only cycles with a distinct set of vertices. We call this set of cycles a *shortcut cycle set*, and it is also the resulting set of cycles we will be using in the coming sections.

For bidirected graphs, we can use a more straightforward algorithm to compute the shortcut cycle set. Observe that for G_b, we compute the edge cycle cover consisting of every vertex pair that share two bidirectional edges, since for every edge (u, v) in G_b, we already have the edge (v, u), and since G_b does not have any self-loops, this is the smallest cycle. In other words, for every pair of bidirectional edges (u, v) and (v, u), we either create the cycle (u, v) or (v, u) — depending on the order of iteration. Therefore, we can simply use $E(G_b)$ as the shortcut cycle set — for every edge $(u, v) \in E(G_b)$ we have that (u, v) is also a cycle.

Secondly, observe that we construct exactly the vertex cover constraints when we use the shortcut cycle set of $E(G_b)$ in the ILP. So we are guaranteed to obtain a DFVS of G_b from the ILP. Suppose that π is a path between two vertices u and v, then we write $V(\pi)$ as the set of all vertices traversed by π, including u and v. Tying everything together, we obtain the following algorithm to construct a shortcut cycle set for any directed graph G, see Algorithm 2. In order to test whether a cycle C uses a distinct set of vertices, we sort C, and test using a hash function whether we already included the same order of vertices in the shortcut cycle set, if not, we include C — a different order of vertices causes the hash to be different.

4.2 Computing an Upper Bound

As mentioned above, computing a single shortcut cycle set $\mathcal{C}$ for a directed graph $G = (V, E)$ may not result in finding a DFVS of G after solving the corresponding ILP. We can run an efficient heuristic over the ILP to obtain a reasonably small upper bound Z, which allows us to verify whether Z is a DFVS of G. If Z is not a DFVS of G, we compute a shortcut cycle set of $G[V \setminus Z]$, until we manage to find a DFVS of G. The hope is then that

Algorithm 2 Shortcut Cycle Set Construction

```
Require: A directed graph $G$.
1: procedure SCS($G$)
2:     if $G$ is bidirected then
3:         return $E(G)$
4:     $\mathcal{C} \leftarrow \emptyset$
5:     for $(u, v) \in E(G)$ do
6:         Find the shortest path $\pi$ from $v$ to $u$ in $G$, if it exists
7:         Let $C$ be the smallest cycle in the graph $G[V(\pi)]$
8:         if $C \neq \emptyset$ then
9:             Sort $C$
10:            if $C \notin \mathcal{C}$ then                ▷ Decide using a hash function.
11:                $\mathcal{C} \leftarrow \mathcal{C} \cup \{C\}$
12:    return $\mathcal{C}$
```

we increase the likelihood that an optimal solution of the generated ILP is then also a DFVS.

Hitting Set We first observe that the standard ILP formulation is equivalent to the Hitting Set problem. In the Hitting Set problem we are given a universe U and a family of sets $\mathcal{F}$ such that $F \subseteq U$ for every $F \in \mathcal{F}$. The task is to find a smallest set $X \subseteq U$ such that for every $F \in \mathcal{F}$ we have $F \cap X \neq \emptyset$. For simplicity, we say that $(U, \mathcal{F})$ is a Hitting Set instance. Observe that we can immediately transform the shortcut cycle set that we have into a Hitting Set instance, which also must be an optimal solution for the ILP that we generate using the same shortcut cycle set. We will therefore design a heuristic for the Hitting Set problem.

4.2.1 Simulated Annealing for Hitting Set

To find an upper bound of a Hitting Set instance, we will be designing and using a simulated annealing algorithm. Recall from Chapter 2 the four key components of a simulated annealing algorithm:

1. defining an appropriate objective function,
2. choosing the initial solution,
3. defining how we move from one solution to the next,
4. and finally, designing the cooling scheme.

The first point is easily solved, namely, we define the objective function as the size of the current hitting set, which we obviously aim to minimize. It remains to discuss how we choose the initial solution, how we move from one solution to the next, and how we choose the cooling scheme.

Initial Solution Two standard initial solutions are either a random solution, or solution obtained by a greedy algorithm. We will use a simple greedy algorithm, to compute the initial solution, which may not necessarily be a minimal solution, but it will be sufficient to start the simulated annealing from. We will later modify this algorithm to make it suitable for finding moves.

We iteratively select elements from U that cover as many sets of $\mathcal{F}$ that are currently not *hit*, until we hit every set $F \in \mathcal{F}$. The pseudocode is given in Algorithm 3.

Algorithm 3 Greedy Hitting Set

```
Require: A Hitting Set instance universe $(U, \mathcal{F})$
1: procedure GreedyHS$(U, \mathcal{F})$
2:     $S \leftarrow \emptyset$
3:     $\mathcal{F}' \leftarrow \mathcal{F}$
4:     while $\mathcal{F}' \neq \emptyset$ do
5:         $x \leftarrow \arg\max_{u \in U \setminus S} |\{F \in \mathcal{F}' \mid u \in F\}|$
6:         $\mathcal{F}' \leftarrow \mathcal{F}' \setminus \{F \in \mathcal{F}' \mid x \in F\}$
7:         $S \leftarrow S \cup \{x\}$
8:     return $S$
```

Moves Let S be the current solution. The idea is to choose an element $u \in S$ uniformly at random, i.e., with probability $p = 1/|S|$, to take out from S. Since Algorithm 3 generally gives decent solutions, we are going to modify it slightly to obtain a new candidate solution S' such that $u \notin S'$, resulting in Algorithm 4. Note that we are only interested in hitting just the sets in $M = \{F \in \mathcal{F} \mid F \cap S \setminus \{u\} = \emptyset\}$, and that we need to exclude u from $U \setminus S'$. In general, it may be the case that there does not exist such a hitting set S', since there may exists an $F \in \mathcal{F}$ with $F = \{u\}$. One can apply a reduction rule at the start to include all such elements in the solution, but since we are including cycles of at least two vertices at all times, we will not apply such a reduction. Hence, there always exists a hitting set S' with $u \notin S'$. The pseudocode is given in Algorithm 4.

Additionally, we will maintain a data structure to efficiently determine all the sets in M. We create an adjacency list for each element in U containing all the sets it hits. When we take out an element $u \in S$, clearly only those sets that u hits might become unhit. Furthermore, we maintain a data structure that counts how many variables of the current solution hit each set of $\mathcal{F}$. Clearly, when the count is equal to 1 for one of the sets that u hits, this set becomes unhit when u is removed from S. After accepting a move, we update the counts, which is a fairly efficient operation.

Given S' and S, we can compute $\delta = |S'| - |S|$, and decide with probability $p = \min(1, e^{-\delta/T})$ to accept this move. Assuming we accept this move,

Algorithm 4 Greedy Fixing HITTING SET

```
Require: A HITTING SET instance $(U, \mathcal{F})$ such that $|F| \geq 2$ for every
         $F \in \mathcal{F}$, a current hitting set $S$, and an element $u \in S$.
Ensure: A hitting set $S'$ with $u \notin S'$.
1: procedure FIXGREEDYHS($U, \mathcal{F}, S, u$)
2:     $S' \leftarrow S \setminus \{u\}$
3:     $\mathcal{F}' \leftarrow \{F \in \mathcal{F} \mid F \cap S' = \emptyset\}$
4:     while $\mathcal{F}' \neq \emptyset$ do
5:         $x \leftarrow \arg\max_{u \in (U \setminus S') \setminus \{u\}} |\{F \in \mathcal{F}' \mid u \in F\}|$
6:         $\mathcal{F}' \leftarrow \mathcal{F}' \setminus \{F \in \mathcal{F}' \mid x \in F\}$
7:         $S' \leftarrow S' \cup \{x\}$
8:     return $S'$
```

we replace S by S'. During the search, we keep track of our best hitting set S^*, and if $|S'| < |S^*|$, we replace S^* by S'.

Cooling Scheme Recall that there are four key elements of the cooling scheme, namely,

1. the starting temperature,
2. when to decrease the temperature,
3. how to decrease the temperature,
4. and when we stop the search.

We will first go over the last three points, before determining the starting temperature. After each time we select an element out of S, we are going to decrease the temperature. Next, we will be using the geometric law to $T_{i+1} = \alpha \cdot T_i$ quickly stop performing the random search given our starting temperature (which we choose later).

Furthermore, we want to stop searching after a fixed number of iterations k. Since we want that the final temperature to be low after these k iterations, we are going to choose our target temperature T_e, and derive the constant α. We recall that we need to choose a small final acceptance probability p_e and determine the smallest cost difference $(\Delta f)_{\min}$ in order to determine the target temperature T_e. Observe that for each local move, we decide to remove one element u from our hitting set S, and thus $(\Delta f)_{\min} = -1$. For our final acceptance probability, we choose $p_e = 10^{-9}$. We then obtain that our target temperature T_e is given by

$$T_e = -\frac{1}{\ln 10^{-9}}.$$

We can now derive the constant α. Observe that $T_i = \alpha^i \cdot T_0$, where T_0 is the starting temperature, and recall that we have a fixed number of iterations k, so we want that $T_k = T_e$. Observe that for $\alpha = (T_e/T_0)^{1/k}$, we indeed have that

$$T_k = \alpha^k \cdot T_0 = ((T_e/T_0)^{1/k})^k \cdot T_0 = (T_e/T_0) \cdot T_0 = T_e.$$

Finally, it remains to choose the starting temperature. We choose the starting temperature $T_0 = 5$, which is high enough such that, initially, the search resembles a random search, and it is low enough that we are only performing a random iterative improvement later in the search.

Tying it all together Now that all essential parts of the simulated annealing algorithm have been discussed, we can give the resulting algorithm. We choose $k = 10^6$, and is chosen such that the implementation of the algorithm completes within a second in many instances. The final algorithm is then given in Algorithm 5.

Algorithm 5 Simulated Annealing Hitting Set

```
1: $k \leftarrow 10^6$
2: $T \leftarrow 5$
3: $T_e \leftarrow -1/\ln 10^{-9}$
4: $\alpha \leftarrow (T_e/T)^{1/k}$
5: $S \leftarrow \textsc{GreedyHS}(U, \mathcal{F})$
6: $S^* \leftarrow S$
7: for $i = 1$ to $k$ do
8:     Let $u \in S$ be a uniformly randomly selected element
9:     $S' \leftarrow \textsc{FixGreedyHS}(U, \mathcal{F}, S, u)$
10:    $\delta \leftarrow |S'| - |S|$
11:    if $\delta \leq 0$ or $e^{-\delta/T} \geq \texttt{rand}(0,1)$ then
12:        $S \leftarrow S'$
13:        if $|S| < |S^*|$ then
14:            $S^* \leftarrow S$
15:    $T \leftarrow \alpha \cdot T$
16: return $S^*$
```

4.3 Improving ILP Relaxation

Many ILP solvers include solving the ILP relaxation as a means to compute a lower bound, which is then subsequently used to prune the search space the ILP solver explores. For our problem that means relaxing $x_i \in \{0,1\}$ to $x_i \in [0,1]$ for every $i \in \{1,\dots,|V|\}$. The objective value $\sum_{i=1}^{|V|} x_i$ is then a lower bound for the number of required vertices in the optimal solution, after which it can function as a way to prune the search space of the ILP. Hence, if we can minimize the difference between this lower bound and the optimal (integer) solution, we may obtain a significantly faster algorithm. For this purpose, we are including extra constraints to the ILP, which we can compute efficiently as well.

Suppose we have a directed graph $G = (V, E)$ consisting of a clique of n vertices, then the optimal solution for G is $n-1$ vertices. When we relax the ILP, the optimal objective value is $1/2 \cdot n$, since for every pair $(v_i, v_j) \in E$, we need $x_i + x_j \geq 1$, and this is satisfied by choosing $x_i = 1/2$ for every $i \in \{1, \ldots, n\}$. Therefore, the larger the cliques in a directed graph, the worse the lower bound of the relaxation becomes.

Obviously, finding all cliques of the graph is too time consuming, so we are going to restrict ourselves to computing all cliques of three vertices of the graph G_b — observe that G_d does not have any cliques since it is a monodirected graph. Henceforth, we call a clique of three vertices a *3-clique.* For every 3-clique $\{v_i, v_j, v_k\}$, we can include the constraint $x_i + x_j + x_k \geq 2$ to the ILP.

Since G_b is bidirected, there exists a fairly simple algorithm to efficiently compute the set of all 3-cliques of G_b. Let $(u, v) \in E(G_b)$ be an arbitrary edge. Then for every $w \in N^+_{G_b}(v)$, we test whether $w \in N^+G_b(u)$, and if it is, then $\{u, v, w\}$ is a 3-clique, and we can construct the corresponding constraint.

Technically, we can also include constraints for all the 4-cliques, but experiments showed that this rarely resulted in better lower bounds over including the constraints generated by the 3-cliques.

4.4 The Algorithm

In Chapter 3 we have seen how we can shrink the input graph G into two smaller graphs $G_d = (V, E_d)$, $G_b = (V, E_b)$, and obtain a set of forced vertices S, which left us to find an optimal DFVS of $G_c = (V, E_d \cup E_b)$, since we could not recover a DFVS using the vertex cover of $\mathcal{U}(G_b)$. Now that we have seen how we can create a small set $\mathcal{C}$ of cycles by computing shortcut cycle sets, and verifying whether the ILP using $\mathcal{C}$ has a small solution that is a DFVS, we can formulate an exact algorithm for the Directed Feedback Vertex Set problem, see Algorithm 6.

It remains to show that Algorithm 6 terminates, since this is not clear from lines 3-6 and 13-19, and that it is indeed correct. We first show that the algorithm terminates.

Theorem 4.1. *Algorithm 6 terminates.*

Proof. To prove that Algorithm 6 terminates, it is sufficient to show that the loops in line 3-6 and lines 13-19 eventually terminate — for all other

Algorithm 6 ILP

```
Require: A directed graph $G_d = (V, E_d)$, a bidirected graph $G_b = (V, E_b)$,
         and a set of forced vertices $S$
Ensure: An optimal DFVS for the original graph $G$
 1: $\mathcal{C} \leftarrow \text{SCS}(G_d) \cup \text{SCS}(G_b)$      ▷ Shortcut Cycle Set computations.
 2: $Z \leftarrow$ HittingSet-SA$(V, \mathcal{C})$      ▷ Simulated Annealing Algorithm.
 3: while $Z$ is not a DFVS of $G_d$ do
 4:     $\mathcal{C}' \leftarrow \text{SCS}(G_d[V \setminus Z])$
 5:     $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}'$
 6:     $Z \leftarrow$ HittingSet-SA$(V, \mathcal{C})$
 7: Create the ILP with the cycles $\mathcal{C}$
 8: Add constraints generated by 3-cliques of $G_b$ to the ILP
 9: Set $Z$ as the initial solution of the ILP
10: Let $X$ be the optimal solution of the ILP
11: if $|X| = |Z|$ then
12:     return $Z \cup S$
13: while $X$ is not a DFVS of $G_d$ do
14:     $\mathcal{C}' \leftarrow \text{SCS}(G_d[V \setminus X])$
15:     $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}'$
16:     Include $\mathcal{C}'$ to the ILP
17:     $L \leftarrow$ HittingSet-SA$(V, \mathcal{C})$
18:     Set $L$ as the new initial solution of the ILP
19:     Let $X$ be the optimal solution of the ILP
20: return $X \cup S$
```

routines and instructions it is easy to see that they terminate. Observe that each time that Z (or X) is not a DFVS of G_d, we compute another shortcut cycle set, which consists of at least one simple cycle. Clearly, G_d only has a finite number of simple cycles, and so eventually, we must find a DFVS of G_d, showing that the algorithm terminates.

□

Now that we have shown that the algorithm terminates, it remains to show that the algorithm is correct, that is, when we return $Z \cup S$ or $X \cup S$, we return an optimal solution for the original input graph G.

Lemma 4.1. *If X is an optimal solution of the ILP generated in Algorithm 6 and X is a DFVS of G_d, then X is an optimal DFVS of the graph $G_c = (V, E_d \cup E_b)$.*

Proof. Observe that the ILP contains a strict subset of the cycles present in G_c, namely for every cycle $C \in \mathcal{C}$, we have that the cycle C uses only edges of G_b or only edges of G_d, but not both. Hence, if X is a DFVS of G_c, then it must be an optimal DFVS of G_c since it is an optimal solution of the ILP. It therefore remains to show that X is a DFVS of G_c.

Suppose, towards a contradiction, that X is not a DFVS of G_c. As mentioned above, after solving the ILP and collecting the optimal solution X, we are guaranteed to find a vertex cover of G_b. Therefore, the graph $G_b[V \setminus X]$ cannot have any edges. Hence, if there is a cycle C in the graph $G_c[V \setminus X]$, it must only use edges present in G_d. But since X is a DFVS of G_d, it must be the case that $X \cap C \neq \emptyset$, which is a contradiction that C is a cycle of $G_c[V \setminus X]$. □

While X may not necessarily be a DFVS of G_d, we know that the found upper bound Z satisfies the ILP and is a DFVS of G_d. Thus, if $|X| = |Z|$, then we have that Z is an optimal solution of the ILP such that Z is a DFVS of G_d. Hence, we obtain the following corollary from Lemma 4.1.

Corollary 4.1. *If X is an optimal solution of the ILP generated in Algorithm 6 and Z is a DFVS of G_d, and satisfies the ILP with $|X| = |Z|$, then Z is an optimal DFVS of the graph $G_c(V, E_d \cup E_b)$.*

Theorem 4.2. *Algorithm 6 is correct.*

Proof. By Lemma 4.1, Corollary 4.1, and by the correctness of the reduction step (see Theorem 3.2), Algorithm 6 indeed reports an optimal DFVS for the original input graph G. □

Chapter 5

Branch & Bound

In this chapter, we describe our other exact algorithm, based on the Branch & Bound principle. Recall that a Branch & Bound algorithm is an exhaustive search where we select a set of vertices H (possibly a singleton) to include in our solution of the current subproblem, and subsequently compute the subproblem that does not contain any vertices of H, and for which we aim to find an optimal solution. The set of vertices that we remove depends on our branching strategy. A good branching strategy may significantly improve the running time. Furthermore, we need to compute good lower and upper bounds to prune the search space as much as possible. It is important that the lower and the upper bounds are as close as possible to each other to seriously prune the search space.

We first discuss how we can integrate the reduction rules of Chapter 3 into a Branch & Bound algorithm, the branching strategy we use, and how we can compute new subproblems and find optimal solutions for each subproblem. We then discuss how we can find a lower bound for a directed graph. Finally, we discuss how such a lower bound, in conjunction with an upper bound can be used to significantly prune the search space.

5.1 Branch & Reduce

As the very first step, even before we start the Branch & Reduce routine that we will be designing in this section, we apply only the nonrestricted reduction rules defined in Chapter 3 on the input graph G, we defer the majority of the reduction work to the initial calls of the Branch & Reduce algorithm. After we have reduced G to G' and a set of forced vertices S, we compute all strongly connected components $\mathcal{C}$ of G'. Observe that we only need to find an optimal solution for each of the strongly connected components of G', together with S, to find an optimal solution of G. We also compute an upper bound for each strongly connected component in $\mathcal{C}$, see also Section 5.3. We supply these strongly connected components to the Branch & Reduce algorithm.

At each recursive call, the first operation is to apply the complete kernelization, i.e., starting with the nonrestricted reduction rules, followed by splitting the resulting graph into a monodirected graph G_d and a bidirected graph G_b, together with all forced vertices S — see also the Reduce & Split algorithm in Chapter 3. This kernelization in each recursive call is what makes a Branch & Bound algorithm a Branch & Reduce algorithm. We will later see how we construct another directed graph from G_d and G_b to use as the input for the Branch & Reduce algorithm.

Branching Strategy We now discuss our branching strategy. Suppose that $G_d + G_b$ is not acyclic. Suppose that there exists a bidirectional edge $(u, v) \in E(G_b)$. Then, if we do not want to choose u to belong in the solution of $G_d + G_b$, then all of $N^+_{G_b}(u)$ must be at least in the solution, otherwise we cannot make G_b acyclic. Hence, either u or all of $N^+_{G_b}(u)$ are part of an optimal solution of $G_d + G_b$. We can quickly find such vertices, if they exists.

Clearly, it may be the case that there does not exist such a u in G_b. Instead, we will branch over a smallest cycle of G_d. Clearly, at least one vertex of any cycle must be removed, and by choosing the smallest cycle, we hopefully do not create too many subsequent branches, therefore limiting the search space. Observe that we can find a smallest cycle of G_b by computing the *shortcut cycle set* of G_b, as described in Chapter 4. Recall that this set is constructed by selecting only the smallest cycles with a distinct set of vertices from an edge cycle cover of $G_d + G_b$, which exists. We then sort this set of cycles in ascending order in terms of the size of the cycles. We can additionally sort the vertices of the smallest cycle in ascending order in terms of their in-degree and out-degree: namely we use the key $|N^-_{G_d}(u)| \cdot |N^+_{G_d}(u)|$ to sort these vertices. The intuition behind this key is that we want to branch over vertices with both a high in-degree and out-degree. If the sum were to be used instead, we may branch over a vertex with a very high out-degree, but a very small in-degree, possibly breaking relatively few cycles.

Computing new subproblems Suppose that we have chosen a set of vertices H to include in the solution of our current subproblem. Clearly, it remains to find an optimal solution Z for the graph $G_c := G_d + G_b$ that does not contain any vertices of H, which is the next subproblem we want to solve. When we combine Z with H, we have obtained a possible solution for G_c. We will therefore construct these subproblems as follows.

We firstly create the new graph G_c, and then remove H from this graph. Each time we branch, we create a new copy of G_c, to prevent G_c changing throughout the recursive call as we also apply reductions on the obtained subproblems, which ultimately also change G_c. This is by far the easiest solution, and is even used by winning solvers of the PACE challenge [11][1]. The subsequent search tree is then tasked to find an optimal DFVS Z for this resulting subproblem. Then, a candidate solution for our current subproblem is $Z \cup H$.

After having computed the solutions of several subproblems, which may also be None for reasons that we will see later, we need to construct an optimal solution for our current subproblem. To do this, we will keep track of a current best solution, initially set at None. Each time we find a solution Z (that is not None) after a branch, we determine whether $Z \cup H$ is a smaller solution, and if so, we use $Z \cup H$ as our best solution. Observe that we recover an optimal solution for the original problem in this way. For each cycle we need to at least remove one vertex, and by obtaining smallest solutions for each branch, we can eventually construct a smallest solution for our original problem.

5.2 Lower Bounds

In Chapter 4 we have discussed that the ILP relaxation gives a lower bound for the number of required vertices in the optimal solution. Recall that we repeatedly compute an upper bound Z for the Hitting Set instance, using the simulated annealing algorithm we defined, consisting of all the shortcut cycle sets that we want to use in our ILP. When Z is a DFVS of G_d, we believe that we have a suitable set of cycles to use in our ILP. We can use this set of cycles to use as a lower-bound computation.

However, in the context of a Branch & Bound where we frequently will need to compute a lower bound, it does not make much sense to spend a significant amount of computing upper bound Z, which is a DFVS of G_b. Observe that we may both require the computation of many shortcut cycle sets until Z is a DFVS of G_d, and additionally, running the simulated annealing algorithm may also be slow when many iterations are used, i.e., 10^6. For both reasons, we fix the number of iterations to find the upper bound Z at a small number, e.g., five iterations, and reduce the number of iterations of the simulated annealing by a factor 100, which reduces the running time by about a factor 100 as well. Experiments showed that 10^6 iterations required about 700ms, and 7ms per branch is acceptable. We adjust the cooling scheme accordingly.

Having computed an appropriate set of cycles, we can also include the 3-cliques of G_b to improve the lower bound. After this, we construct the relaxed ILP-formulation, and solve it, obtaining a lower bound $\ell \in \mathbb{R}$. Observe that $\lceil \ell \rceil$ is also a lower bound, so we return this value as our lower bound. The pseudocode to compute a lower bound for $G_d + G_b$ is then given in Algorithm 7 (we do not explicitly need to compute $G_d + G_b$). Observe

[1]We inspected their source code to conclude this fact.

that in the event that G_b is not empty, we first need to run a simulated annealing algorithm before verifying whether Z is a DFVS of G_d, otherwise we will probably have a very small set of cycles to compute our lower bound for. We will account for one iteration of the 5 when G_b is not empty.

Algorithm 7 ILP-relaxation

```
Require: A monodirected graph $G_d = (V, E)$, a bidirected graph $G_b$
 1: procedure LOWER($G_d, G_b$)
 2:     $\mathcal{C} \leftarrow \mathrm{SCS}(G_d) \cup \mathrm{SCS}(G_b)$        ▷ Shortcut Cycle Set computations.
 3:     $k \leftarrow 5$
 4:     $Z \leftarrow \emptyset$
 5:     if $E(G_b) \neq \emptyset$ then
 6:         $Z \leftarrow$ HITTINGSET-SA($V, \mathcal{C}, 10^4$)
 7:         $k \leftarrow 4$
 8:     while $Z$ is not a DFVS of $G_d$ and $k > 0$ do
 9:         $Z \leftarrow$ HITTINGSET-SA($V, \mathcal{C}, 10^4$)
10:         $\mathcal{C}' \leftarrow \mathrm{SCS}(G_d[V \setminus Z])$
11:         $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}'$
12:         $k \leftarrow k - 1$
13:     Create the ILP with the cycles $\mathcal{C}$, with variables $x_i \in [0, 1]$
14:     Add constraints generated by 3-cliques of $G_b$ to the ILP
15:     Solve the ILP, and let $\ell$ be its optimal objective value
16:     return $\lceil \ell \rceil$
```

5.3 Upper Bounds

Recall from Chapter 2 that we can use upper bounds as a budget k to prune the search space. We may also supply budgets that are smaller than an actual upper bound in order to prune the search space even more. However, we may then determine that a subproblem does not admit a solution of size k, e.g., if we have that G_d or G_b contains a cycle but a budget of 0, we can stop the search for a solution. In such a situation we return NONE, thus each recursive call may output an optimal DFVS of its subproblem of size at most k, or NONE if its subproblem does not admit a DFVS of size at most k. In the remainder of this section we discuss how we aim to exploit upper bound computations to prune the search space as much as possible.

Initially, we can compute an upper bound of $G_d + G_b$ after applying our initial reductions before starting the Branch & Reduce algorithm. We have two choices of algorithms here, the simulated annealing algorithm of Galinier et al. [10], and adapting Algorithm 6 to return the found upper bound Z, see Algorithm 8. In practice, the simulated annealing algorithm of Galinier et al., based on maintaining topological orderings, sometimes manages to give better solutions, but showed to be significantly slower. While it is very important to prune the search space as much as possible, we do want to find a good upper bound quickly. Algorithm 8 manages to terminate significantly faster, while still giving many solutions close to optimal, or optimal. We use this algorithm to compute an upper bound for each strongly connected component of G', i.e., the graph obtained by only applying the nonrestricted reduction rules. Since Z is an actual upper bound (and not just the size of an upper bound), we can instruct our Branch & Reduce algorithm to search for a solution of size at most $|Z| - 1$, if it exists. If such a solution does not exist, it must be that Z is an optimal solution, which we can immediately return.

Algorithm 8 Finding a Hitting Set based upper bound of $G_d + G_b$

```
Require: A monodirected graph $G_d = (V, E)$, a bidirected graph $G_b$
Ensure: A DFVS $Z$ of $G_d + G_b$
 1: procedure Upper($G_d, G_b$)
 2:     $\mathcal{C} \leftarrow$ SCS($G_d$) $\cup$ SCS($G_b$)        ▷ Shortcut Cycle Set computations.
 3:     $Z \leftarrow \emptyset$
 4:     if $E(G_b) \neq \emptyset$ then
 5:         $Z \leftarrow$ HittingSet-SA($V, \mathcal{C}, 10^6$)
 6:     while $Z$ is not a DFVS of $G_d$ do
 7:         $Z \leftarrow$ HittingSet-SA($V, \mathcal{C}, 10^6$)
 8:         $\mathcal{C}' \leftarrow$ SCS($G_d[V \setminus Z]$)
 9:         $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}'$
10:     return $Z$
```

Furthermore, observe that Algorithm 7 gives an upper bound Z for the Hitting Set instance, which may also be a DFVS of G_d, but not necessarily. Rather than computing a new upper bound, which is relatively costly, we can use Z instead. We may possibly improve current budget for this branch, and subsequent branches, but if not, we are not going to spend additional time to look for a good upper bound of our current subproblem. Spending more time to find an upper bound for each node of the search tree to tighten our budget is too costly. We hope that our initial upper bound is of sufficient quality such that we already have very tight budgets at each recursive call. When we have managed to find a DFVS Z with $|Z| < k$, where k is our current budget, we can apply the same trick as above, where we tighten the budget to $|Z| - 1$. If Z is not a DFVS of G_d, we simply continue searching for a DFVS of size at most k.

Additionally to (possibly) using the DFVS of Algorithm 7, we can use the solution X of a recursive call, if it exists, and replace our budget with $|X| - 1$, and proceed to our next branches. Observe that we keep tightening our budgets as much as possible each time we have a candidate solution. Note that, this is also a smaller solution than our best solution found thus far.

Finally, recall that we can prune the search space using the set of forced vertices, and our computed lower bounds. If $|S| > k$, or $\ell > k$, we can stop the search and return None. Since we want to stop searching as soon as possible, we perform both of these checks immediately after running the kernelization. When $|S| \leq k$ or $\ell \leq k$, we start branching, and recover the smallest solution, if it exists.

Chapter 6

ILP Evaluation

In this chapter we aim to evaluate the effectiveness of our kernelization for our ILP-based algorithm. We analyze whether the use of a kernelization can reduce the running the time of our ILP-based algorithm presented in Chapter 4, and whether the models that we require to solve the problem are smaller in size. First however, we briefly discuss the results of the Parameterized Algorithms and Computation Experiments challenge, as our ILP-based algorithm was the best performing exact algorithm. We then discuss how we evaluate our ILP-based algorithm, and which instances we will be using. After, we gather and briefly discuss all the results. Finally, we conclude how effective our reductions proved to be.

GitHub Repository `https://github.com/satanja/Thesis`.

6.1 PACE

This year's edition of the Parameterized Algorithms and Computation Experiments challenge (PACE 2022) hosts the Directed Feedback Vertex Set problem. The challenge consists of two categories. The first category is the exact track, where the goal is to design an efficient as possible exact solver — this is also the track we participated in. And secondly, a heuristic track, where the goal is to give a DFVS that is as close to optimal as possible, but not necessarily optimal. Since our focus is on the exact track, we will henceforth only focus on the exact track.

For evaluation, 200 instances were used in the challenge, of which 100 *public* instances were released in February 2022 for participants to test their algorithms against — we later discuss how these instances were selected. Participants were required to submit their solver (as an executable) to the platform Optil.io[1]. For each instance, the solver had a maximum time limit of 30 minutes to find a solution. Participants were scored by the total number of solutions found within this time limit per instance.

[1] `https://optil.io`

Additionally, the platform would give feedback about the correctness and optimality of the computed solutions for the public instances, i.e., users could see per instance whether the computed solution used as many vertices as the smallest *known* solution, whether the solution used more vertices than the smallest known solution, or whether the solution was not a DFVS of the input graph. While highly unlikely, it may therefore be that the reported solution is marked as optimal, while it was not optimal; the solution may have been the smallest known solution for that particular instance. Solvers were required to give optimal solutions at all times, and were subject to disqualification if they failed to produce optimal solutions.

Furthermore, submitted solvers could only use *open source* libraries or dependencies. This constitutes source code that is publicly and freely accessible under an open source license. Closed source Integer Linear Programming solvers, such as Gurobi[2], were therefore prohibited to use. Additionally, per team, at most three solvers could be submitted that did not share a significant part of each other's code base. Finally, solvers were prohibited from using multithreading techniques, with two small exceptions: other threads and processes may be started as long as there is exactly one non-blocking process or thread, or as long as their purpose is to pipe input and output between other threads and processes.

With these rules in mind, we decided to use COIN-OR's Mixed Integer Linear Programming solver Cbc[3]. This was the best performing open source ILP library for which a Rust[4] interface existed, in the form of a so called *wrapper*. Unfortunately, very few performant open source ILP library wrappers exist for Rust, excluding options like Google's OR-Tools[5] or SCIP[6]. We later see that using a good ILP solver may contribute significantly in obtaining good results in the competition.

Instances Schulz et al. [13], the program committee of this year's challenge, generated a large pool of (synthetic) instances, using *KaGen* [9]. Initially, a large pool of instances were generated. From this pool, instances that had fewer than 1000 strongly connected components, had more edges than vertices, and were of at most 50MB were selected. Then, the exact solver of the organization was used to exclude instances that were solvable within a second. Instances were then uniformly sampled at random. In particular, the instance pool eventually contained both easy and hard instances [13].

[2]`https://www.gurobi.com`

[3]`https://github.com/coin-or/Cbc`

[4]`https://www.rust-lang.org/`

[5]`https://developers.google.com/optimization/`

[6]`https://www.scipopt.org/`

6.1.1 Results

During development of the solver, it became clear that an earlier version of Algorithm 6 already managed to find the most solutions compared to the Branch & Reduce algorithm. For this reason, we decided to focus on an ILP-based algorithm, and improve it as much as possible, obtaining Algorithm 6.

As mentioned in Chapter 3, we use the vertex solver of Hespe et al. for finding a vertex cover of the induced undirected graph of G_b, which hopefully is a DFVS of G_d. In some situations, however, their solver already exceeds the time limit of 30 minutes, while it may not even be the case that a vertex cover of G_b gives us a DFVS of G_d. To possibly improve the solver's performance for the challenge, we apply a fixed time limit of 5 minutes for the vertex cover solver, and simply continue with the ILP as if a vertex cover of G_b is not a DFVS of G_d. After 5 minutes, it was generally highly likely that the vertex cover solver would exceed the 30 minute time limit. We break off the search in the hope that Cbc can find the solution, even if it is very unlikely to.

In total, 13 teams participated in the exact track, all results can be seen in Table 6.1. Three teams were disqualified due to computing a suboptimal on exactly one of the 100 private instances. Unfortunately, this includes our solver as well. This was due to a small bug in the restricted version of a reduction rule that ultimately did not contribute much to solving instances more quickly using the ILP-based algorithm. For this reason, we omitted this reduction rule from the thesis and the solver.

If we disregard our unfortunate disqualification, we see that the solver managed to produce an acceptable number of 144 solved instances out of 200 instances. Without the disqualification, this would have resulted in an acceptable 5th position — or a 7th position if the other the disqualification of the two disqualified solvers are disregarded as well. It is clear that our ILP-based algorithm is indeed a relatively efficient solver on synthetic instances, but it is unclear how the solver performs in real-world applications. It would have, in fact, been the best solver written in Rust, as the other teams have used C++, C or Java as their programming language of choice.

6.2 Evaluation

In addition to the PACE challenge, we want to evaluate our ILP-based algorithm more in-depth. We have two primary goals, based on analyzing the performance of our ILP-based algorithm when the original instance has been reduced, and when it has not been reduced. Since our ILP-based algorithm requires a monodirected and a bidirected graph and a set of forced vertices, we transform a directed a directed graph G into a monodirected graph and a bidirected graph. Instead of applying a kernelization on the

Rank	Team	Solver	Points
1	A. Schidler, R. Kiesel	raki123	185
2	E. Gerhard et al.	grapa-java	165
3	S. Angrich et al.	mt-doom	152
4	R. Červený et al.	goat_exact	151
5	H. Froese et al.	THS_exact	140
6	T. Behr	mndmky	130
7	H. Dickel et al.	DUM	125
8	Y. Mizutani	yos	120
9	R. Götz	rubengoetz	88
10	A. Jain et al.	DRIP	32
-	A. Meiburg	Timeroot	DQ (175)
-	S. Swat	swats	DQ (160)
-	S.A. Tanja	satanja	DQ (144)

Table 6.1: Results of the PACE 2022 Challenge, exact track. Disqualified solvers were later resubmitted, and their number of solved instances is given between parentheses. (Only the first initial of the other participants are known.)

input graph, we simply move all bidirectional edges to a new graph, which gives us a monodirected and a bidirected graph. This is hardly a reduction, and requires very little time. Furthermore, the set of forced vertices that we supply is empty when no reductions are performed. Our research goals are then as follows.

1. Firstly, we want to determine whether enabling reduction rules can be used to speed up an exact solver, and in this case, whether Algorithm 6 manages to find an optimal solution for a kernel of a particular instance more quickly than for the instance itself. Henceforth, for clarity, we write AlgILP and AlgILP + K when we run our ILP-based algorithm on the original instance or its kernel, respectively. We measure the entire running time of our ILP-based algorithm since even if the last ILP model is easier to solve, we may have had a higher cost setting up that model, which is not desirable.

   In line with the PACE challenge, we also want to decide whether turning the reductions on leads to the solver finding more solutions within the specified time limit.

2. Secondly, we want to investigate how the resulting models, i.e., the models that yielded a DFVS, of the original instance and its kernel compare. We will measure the number of constraints in those models, and the number of *alive variables*. We call variables that appear at least once in a constraint alive variables. This number may be smaller

than the number of vertices, e.g., when a vertex is not part of any cycle its variable will also not be contained in any constraint.

3. Thirdly, we also investigate how many ILP solver restarts are required to recover a DFVS of G_d by solving the ILP. Recall, in Algorithm 6, after we start solving the ILP, we may need to generate additional shortcut cycles sets until the ILP finally manages to find a DFVS of G_d, restarting the ILP solver each time. Clearly, restarting the ILP solver is costly, and each time we require more shortcut cycle sets, we say that we required an additional *ILP restart*

   During development of the solver, we naively implemented the reductions. On the public instances of the PACE challenge, the reductions typically completed within one minute, which was of acceptable performance. However, for significantly larger graphs, we expect that the reductions will require a significant amount of time to complete. The used reduction rules can however be implemented more efficiently. We therefore simply ignore the running time of the reductions themselves, and only consider the ILPs generated on the reduced graphs, and the time for the ILP solver to recover an optimal solution.

Algorithm Modifications As per the rules of the challenge, we were restricted in choosing only open source ILP solvers. For our own purposes, however, we will be using Gurobi version 9.5.2 instead of Cbc, which is known as one of the best ILP solvers in the market.

Additionally, since Gurobi managed to find solutions for all bidirected graphs of the public instances of the PACE challenge, we disabled finding a vertex cover for G_b during our reductions, and immediately continue with solving the ILP. Gurobi, unfortunately, does not necessarily report optimal solutions by default. During its execution a lower bound ℓ and an upper bound m are computed. When $(m - \ell)/m$ is within a certain threshold `MIPGap`, the solution is assumed to be optimal. By default, `MIPGap` is set to 10^{-4}, and so for large solutions, like some solutions for instances of the PACE challenge, we may obtain a suboptimal solution. Hence, we set `MIPGap` to 0, which also affects the running time of Gurobi.

Environment All our experiments will be performed on an AMD 5900x processor, a 12 core and 24 thread processor, at factory settings — the processor supports *overclocking*[7]. Additionally, this system has a total of 16 GB of memory at its disposal. The system runs Windows 11. Due to the large volume of instances of the PACE challenge and the other instance pools we will be generating, we will run 20 solvers (each as a single thread) receiving their own instance at a time — each time an instance finishes, we

[7]Increasing the base CPU clock frequency.

start a new solver thread on the next instance, if it exists. However, the more solver threads we run, the slower their individual running time, due to limited cache capacity per core, context switching of the operating system, and the CPU lowering its clock speed automatically. When performing our experiments on a smaller instance pool of at most 15 instances, we will run 10 solver threads instead, giving a more accurate representation of the running time.

6.3 Instances

After the submission deadline of the PACE challenge, all 200 instances were made publicly available. We will thus be using all of these instances. In addition to these instances we will be generating other instances how AlgILP + K compares to AlgILP. We will consider two classes of graphs, namely *tournaments* and *cyclic grids.*

6.3.1 Tournaments

A *tournament* is a directed graph $G = (V, E)$ such that for every $u, v \in V$, either $(u, v) \in E$ or $(v, u) \in E$, but not both. Intuitively, we can see the directed edges as the relation "*u beat v*". Given a number of vertices n, we can generate a tournament very easily. For every $u, v \in V$, decide with equal probability to include (v, u) or (u, v). Since this results in very dense graphs, we can additionally supply a *density* parameter d, where

$$d = \frac{|E|}{1/2 \cdot n(n-1)} = \frac{2|E|}{n^2 - n}.$$

Note that the maximum number of edges in a tournament is $1/2n(n - 1)$. If we sample every possible edge with probability d, we obtain that the expectation of the number of edges is our desired number of edges. This is slightly more straightforward to implement than selecting $d \cdot 1/2 \cdot n(n - 1)$ monodirectional edges uniformly at random, at the cost of only obtaining an approximate density — which we have regardless since we introduce bidirectional edges with some probability p.

Without inserting bidirectional edges, it is unlikely that our reductions manage to reduce a relatively dense tournament. There are going to be few strongly connected components, and the probability that each vertex has at least two ingoing or outgoing edges is high. However, since we know that we can split off the bidirectional edges of the tournament, the hope is that we can at least make some progress reducing such a graph, especially when the probability to introduce a bidirectional edge is high.

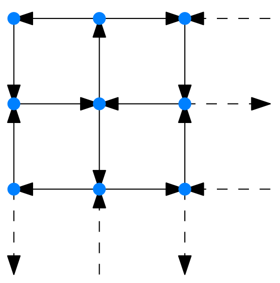

Figure 6.1: A cyclic grid. Observe that for any grid square, we either have a clockwise or a counter-clockwise cycle.

6.3.2 Cyclic Grids

We place k^2 vertices in a 2-dimensional grid with size $k \times k$. Then, for each square of 4 vertices, we can construct a cycle between those vertices such that we only need monodirectional edges, see Figure 6.1. Observe that our kernelization work really well on these graphs, see Figure 6.2. Recall that our kernelization first applies all nonrestricted reduction rules on the directed graph. Using only these nonrestricted reduction rules, we can already completely reduce the input graph. Starting from the corners, we can apply Reduction 3.3 and Reduction 3.4, until we obtain a self-loop on a vertex, which we can reduce using Reduction 3.2.

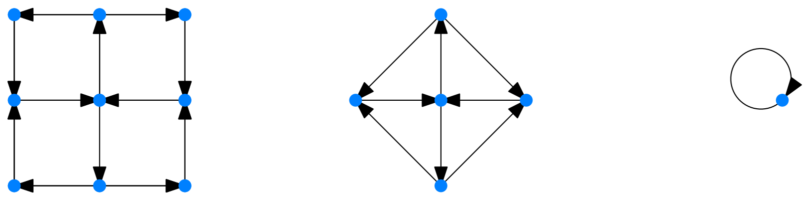

Figure 6.2: Completely reducing a 3×3 cyclic grid using only nonrestricted reduction rules.

Note that the way we generate our constraints, we include a cycle for each grid cell. When additionally bidirectional edges are introduced, those smaller constraints will instead be used initially, reducing the amount of available information to recover a DFVS of the graph. We expect that additional constraints will be generated using the Hitting Set simulated annealing algorithm. Hopefully, when the graph is reduced, significantly fewer constraints will be needed.

6.3.3 Amending the Tournaments and Cyclic Grids

For cyclic grids, reduction rules are very effective, i.e., it can be shown that the reduced graph is empty. On the other hand, tournaments are generally hard to solve using AlgILP and AlgILP + K.

We can introduce bidirectional edges to a tournament in order to hopefully obtain an easier instance, since we can move these bidirectional edges to another graph. We can also introduce bidirectional edges to cyclic grids in order to prevent the cyclic grid from being completely reduced. For every edge in the resulting graph, we can decide with probability $0 \leq p < 1$ to introduce the reverse direction to the graph. We choose $p < 1$, otherwise the graph becomes bidirected, for which we already established that Gurobi already performs really well (using the constraints we generate).

6.3.4 Instance Selection

We generated tournaments of 500 and 1000 vertices, with a density d starting from 0.05, up to 1 with increments of 0.05, and p starting from 0.2, up to 0.9, with increments of 0.05. We then selected the 15 slowest instances for which we could find their solution within 5 minutes using AlgILP. The specifications of these instances can be found in Table 6.2. The number of vertices and edges are given for both the original graph, and its kernel, the latter being denoted by $|V_k|$ and $|E_k|$, respectively. These instances were sorted by the difference (AlgILP + K) – AlgILP in ascending order, and were then labeled.

Instance	$\|V\|$	$\|E\|$	$\|V_k\|$	$\|E_k\|$	d	p
t_01	500	3684	492	3321	0.25	0.55
t_02	1000	4842	832	3304	0.15	0.7
t_03	1000	5821	936	4724	0.2	0.5
t_04	500	3087	467	2721	0.25	0.3
t_05	500	2017	353	1385	0.2	0.2
t_06	500	2243	412	1810	0.2	0.25
t_07	500	2914	482	2316	0.2	0.7
t_08	500	2802	465	2090	0.2	0.65
t_09	500	2428	443	1942	0.15	0.85
t_10	500	3150	487	2947	0.25	0.35
t_11	500	2779	465	2572	0.25	0.2
t_12	500	2444	447	2052	0.15	0.9
t_13	500	2925	466	2692	0.25	0.25
t_14	500	3653	491	3293	0.25	0.5
t_15	1000	6136	956	5036	0.2	0.55

Table 6.2: Specification of the selected tournament instances.

The cyclic grids turned out to be relatively easy to solve, especially for graphs with few vertices, so we generated fairly large graphs starting from 10,000 vertices, up to 50,000 vertices with increments of 500. For a particular choice of number of vertices n, we would create a $\lceil\sqrt{n}\rceil \times \lceil\sqrt{n}\rceil$ grid, slightly using more vertices when n was not a perfect square. The probability to introduce bidirectional edges was limited to starting from 0.1 up to 0.5, again with increments of 0.05. Again, we selected the 15 slowest instances for which we could find their solution within 5 minutes using AlgILP. The specifications of these instances can be found in Table 6.3. These instances were sorted by the difference (AlgILP + K) − AlgILP in ascending order, and were then labeled.

Instance	$\|V\|$	$\|E\|$	$\|V_k\|$	$\|E_k\|$	p
gr_01	15129	33043	13148	28834	0.1
gr_02	47524	108786	41852	94477	0.15
gr_03	46656	106684	41312	93102	0.15
gr_04	18225	39768	15896	34753	0.1
gr_05	44521	101859	39584	89179	0.15
gr_06	49284	112955	43361	97633	0.15
gr_07	48400	110857	43146	97290	0.15
gr_08	11025	24047	9243	20269	0.1
gr_09	49729	113934	43853	98993	0.15
gr_10	45796	104931	40352	90942	0.15
gr_11	48841	111771	43472	97978	0.15
gr_12	4096	8465	2987	6300	0.05
gr_13	17689	38621	15616	34195	0.1
gr_14	20164	44125	17554	38474	0.1
gr_15	13689	29911	11988	26281	0.1

Table 6.3: Specification of the selected cyclic grid instances.

6.3.5 Measurement Deviations

In the subsequent sections we run the experiments once. We measured the running the time of AlgILP + K, starting with 10 threads like in most experiments. We picked one of our generated instances, namely t_07, and measured a standard deviation of 142ms over 7 separate runs, see Table 6.4[8]. Henceforth, we assume that the single measurements are therefore fairly accurate, but running times that are close to each other, i.e., within 300ms, about twice the standard deviation, should be considered to be about the same.

[8]We have a higher running time in Table 6.8 than all these 7 measurements, which is most likely due to higher ambient temperatures, and the CPU reducing its maximum clock speed a little because of it. Most of the experiments were conducted during a heatwave, while we measured the standard deviation after the heatwave.

Run	Running time
1	20.15
2	19.78
3	20.01
4	19.72
5	19.94
6	19.95
7	19.92

Table 6.4: Running times (in seconds) of instance t_07 when started in a pool of 15 instances and 10 available threads.

6.4 Results

6.4.1 PACE Instances

We first analyze the results for the instances of the PACE challenge, for both the original instances and their kernels. In line with the PACE challenge, we enforce a time limit of 30 minutes on the input instances.

For all original graphs, AlgILP managed to find a total of 187 solutions out of 200 instances within 30 minutes per instance, significantly outperforming Cbc. Using AlgILP + K instead, a total of 189 solutions were found. For each original graph for which we could find a solution using AlgILP, we also managed to find the solution using AlgILP + K, but also found two additional solutions, showing that it can certainly be beneficial to use reductions to accelerate an exact solver.

In order to compare the results further, we exclude those instances for which we were unable to reduce the input instance or for which we failed to compute a solution within 30 minutes. This yielded a pool of 184 instances, i.e., only three instances were unable to be reduced. We then compared the running time of AlgILP and AlgILP + K. In Table 6.5, we show all instances where AlgILP + K was faster or slower by at least 30 seconds. These instances are again sorted by the difference (AlgILP + K)−AlgILP in ascending order.

Unfortunately, we see that applying the kernelization may also increase the running time significantly for a good number of instances. We also see that a good number of instances benefitted from being reduced. For the remaining instances not included in the table, the reductions do not seem to have had a very significant effect in reducing the overall running time. For 119 instances, the absolute difference was within 1 second.

We choose the first 5 instances and the last 5 instances from Table 6.5 to investigate further. We will measure how many alive variables and constraints there are in the generated models for both AlgILP and AlgILP + K. We write m_0 and n_0 for the number of constraints and alive variables

Instance	AlgILP	AlgILP + K
e_189	537.35	167.09
e_187	432.97	125.32
e_174	384.37	137.67
e_182	363.85	170.62
e_167	396.38	212.94
e_110	240.76	140.10
e_149	278.91	178.60
e_121	220.81	131.56
e_183	208.69	124.80
e_105	124.33	76.66
e_165	105.93	59.87
e_173	1371.05	1328.60
e_177	408.82	372.73
e_175	73.24	41.73
e_170	74.98	43.91
e_153	258.70	332.77
e_180	264.13	340.51
e_148	74.25	293.49
e_181	474.85	911.05
e_172	822.16	1280.06
e_186	132.81	767.18

Table 6.5: Running time (in seconds) for some instances of the PACE challenge, for both AlgILP and AlgILP + K.

of the ILP models generated by AlgILP, respectively. Similarly, we define m_k and n_k, but for AlgILP + K instead. We also give the relative difference r_m and r_n for both the number of constraints and alive variables, respectively. The results are given in Table 6.6.

It is immediately clear that the reductions proved to be very effective in reducing the first 5 instances. We see that the number of constraints are decreased by several hundreds, and the same holds for the number of variables, with large relative differences. Frequently, more than half of the number of alive variables was reduced. For the last 5 instances, this is completely different. The number of constraints barely decreases, and neither do the number of variables. The relative differences are also significantly smaller. It is obvious that reductions are most effective when they manage to significantly reduce the input graph, and therefore the required number of constraints and alive variables in the resulting linear models, to reduce the running time. On the other hand, when not so much progress is made in reducing the input graphs, the kernels may be harder to solve.

Finally, we determined how many ILP solver restarts are required for

Instance	m_0	n_0	m_k	n_k	r_m (%)	r_n (%)
e_189	2062	661	1451	243	29.63	63.24
e_187	3247	712	2442	427	24.79	40.03
e_174	2191	652	1574	236	28.16	63.80
e_182	1699	673	1374	240	19.13	64.34
e_167	4497	821	4173	563	7.20	31.43
e_180	467435	32765	467220	32717	0.05	0.15
e_148	27506	2033	27351	2010	0.56	1.13
e_181	468490	32767	468314	32731	0.04	0.11
e_172	127581	4095	127565	4091	0.01	0.10
e_186	21917	2023	21769	1971	0.68	2.57

Table 6.6: The total number of required constraints and alive variables produced by AlgILP and AlgILP + K for the PACE instances.

during the execution of AlgILP and AlgILP + K, see Table 6.7. As can clearly be seen, the instances that benefitted the most from the reductions required no ILP solver restarts. Additionally, we see that we may need more ILP solver restarts for AlgILP + K compared to AlgILP, see instance e_186.

Instance	AlgILP	AlgILP + K
e_189	3	0
e_187	1	0
e_174	0	0
e_182	5	0
e_167	2	0
e_180	0	0
e_148	0	0
e_181	0	0
e_172	0	0
e_186	0	1

Table 6.7: Number of constraint generation rounds needed to find a DFVS for the PACE instances.

6.4.2 Tournaments

For the selected tournaments, the reductions seemed to be relatively effective at accelerating the required solve time, see Table 6.8. For the first instance, already 50 seconds was saved by computing its kernel. For the last instance, however, the solve time exceeded the 5 minute limit by 9 seconds, being a little over 20 seconds slower. It is again clear that some of these instances

Instance	AlgILP	AlgILP + K
t_01	299.55	249.76
t_02	297.06	267.33
t_03	108.06	86.18
t_04	29.52	13.78
t_05	22.49	9.88
t_06	10.83	4.62
t_07	23.34	20.50
t_08	7.59	6.03
t_09	5.91	5.28
t_10	6.35	10.21
t_11	190.54	194.63
t_12	20.95	26.72
t_13	11.60	21.52
t_14	82.31	100.52
t_15	288.91	309.82

Table 6.8: Running time (in seconds) of both AlgILP and AlgILP + K for the tournament instances.

benefitted more from the reductions than others.

Secondly, we determine the number of constraints and alive variables for both AlgILP and AlgILP + K, see Table 6.9. We can clearly see that the vast majority of the models that AlgILP + K produced required fewer constraints and alive variables. However, for t_13, we see that the number of constraints has actually gone up by 7.08%, so in some circumstances, the reductions do not necessarily reduce the number of constraints in our ILP-formulation. This can be due to the fact that we may need to compute more shortcut cycle sets for the initial set of cycles that we use for the constraint generation — no ILP restarts were needed to solve this instance when using AlgILP + K.

Additionally, we see that the relative difference in the number of constraints and variables does not always need to be very significant in order to save much time. The kernel of instance t_01 already saved about 50 seconds, but only 1.31% and 1.40% of the constraints and variables were removed, respectively. On the other hand, even if relatively more constraints and variables were removed, such as for instance t_15, it may still be the case that AlgILP + K needs more time.

Finally, it remains to determine the number of ILP solver restarts we need when using AlgILP and AlgILP + K. Unfortunately, we observe again that we may need additional ILP solver restarts for AlgILP + K compared to AlgILP for some instances, see Table 6.10. We also observe that when the number of ILP solver restarts decreased, the running time

Instance	m_0	n_0	m_k	n_k	r_m (%)	r_n (%)
t_01	1602	499	1581	492	1.31	1.40
t_02	2010	987	1652	832	17.81	15.70
t_03	2355	993	2171	936	7.81	5.74
t_04	1754	496	1507	467	14.08	5.85
t_05	1285	475	816	353	36.50	25.68
t_06	1015	485	875	412	13.79	15.05
t_07	1197	494	1158	482	3.26	2.43
t_08	1105	496	1029	465	6.88	6.25
t_09	1110	495	971	443	12.52	10.51
t_10	1509	498	1502	487	0.46	2.21
t_11	3000	497	2804	465	6.53	6.44
t_12	1151	494	1026	447	10.86	9.51
t_13	1667	498	1785	466	-7.08	6.43
t_14	1614	499	1561	491	3.28	1.60
t_15	2445	996	2344	956	4.13	4.02

Table 6.9: The total number of required constraints and alive variables produced by AlgILP and AlgILP + K for the tournaments.

generally improves.

6.4.3 Cyclic Grids

While the reductions seemed relatively effective to accelerate the ILP solver for the tournaments, this is not the same story for the cyclic grids. Against our expectations, the reductions were not of significant aid here. The running times of AlgILP and AlgILP + K are given in Table 6.11. For the last instance, finding the optimal solution using AlgILP + K required a total of 16 minutes and 23 seconds, significantly worse than the original instance.

Looking at the number of constraints and alive variables in the resulting models, see Table 6.12, we see that introducing the bidirectional edges made the instances significantly harder to reduce. Even though at least 10% of the number of alive vertices were removed for the kernels, AlgILP + K required more time than AlgILP for a lot of the instances. Even for gr_12, where 27.04% of the number of alive variables were removed for the model produced by AlgILP + K, this was still not enough to result in a faster solve time. Additionally, we expected a larger decrease in the number of constraints.

If we look at the number of ILP solver restarts, see Table 6.13, we can see that for the majority of the instances additional ILP solver restarts were needed. Even for instances that terminated faster, such as gr_02, additional ILP solver restarts were needed.

Instance	AlgILP	AlgILP + K
t_01	0	0
t_02	0	0
t_03	0	0
t_04	2	0
t_05	0	0
t_06	0	0
t_07	0	0
t_08	0	0
t_09	0	0
t_10	0	0
t_11	4	7
t_12	0	0
t_13	0	0
t_14	0	0
t_15	0	0

Table 6.10: The number of ILP solver restarts for AlgILP and AlgILP + K for the tournaments.

6.5 Conclusion

It is clear that the ILP-based algorithm preforms fairly well when Cbc was used, and performs exceptionally well when Gurobi was used instead of Cbc, managing to even more solutions than the best solver of the PACE challenge, but we are most likely using a better system.

Furthermore, Gurobi even managed to outperform the vertex cover solver from Hespe et al., managing to find a DFVS of a bidirected graph quicker than the vertex cover solver could find a vertex cover for its induced undirected graph.

Regarding the reduction rules, we also saw that reduction rules can indeed be used to find solutions for what would have been otherwise too hard instances for the 30 minute time limit, but this number was limited to 2 instances. Especially for the instances where many constraints and alive variables were removed from the resulting ILP models, the running time drastically decreased.

However, when few constraints or alive variables are pruned, the reductions can either cause the solve time to decrease or increase. Clearly, we cannot know how many constraints and alive variables are needed to recover a DFVS before applying AlgILP or AlgILP + K. Additionally, sometimes even more ILP solver restarts were needed for the kernels, further increasing the running time required.

We can conclude that when the reductions manage to cause a significant

Instance	AlgILP	AlgILP + K
gr_1	119.12	79.67
gr_2	55.42	27.75
gr_3	67.98	48.57
gr_4	36.46	30.15
gr_5	34.23	42.72
gr_6	39.70	58.72
gr_7	39.28	59.58
gr_8	34.18	60.68
gr_9	38.00	65.73
gr_10	33.91	72.97
gr_11	53.24	106.99
gr_12	117.44	176.80
gr_13	42.44	111.81
gr_14	67.89	229.11
gr_15	52.51	983.00

Table 6.11: Running time (in seconds) of both AlgILP and AlgILP + K for the cyclic grids.

reduction in the size of the graph the running time may very well decrease. On the other hand, when the instances are already "hard", and the reductions do not manage to significantly reduce the graph, the resulting running time may very well increase compared to the original instance.

Additionally, the question is raised whether a few reduction rules should even be implemented when an already strong ILP solver is used, such as Gurobi. Considering that without any reductions Gurobi already managed to recover 187 solutions of the PACE challenge, one may be better off generating a good ILP model before spending time implementing reductions. Such an ILP solver is already highly optimized, and it may prove itself useful without much work already. It seems that already the "generic" ILP reductions, which have no information about the fact that we are attempting to solve the Directed Feedback Vertex Set problem, are sufficiently strong to already make AlgILP practical.

We do observe that reductions manage to generally shrink the required models to solve the problem, which is indeed desirable, especially when models may be very large. A potential sacrifice of running time may be worth the reduction in required memory in some situations.

Future work should determine whether a set of very strong reductions rules, that managed to significantly reduce the input graph on a large range of instances, can significantly reduce the running time of already hard instances.

Instance	m_0	n_0	m_k	n_k	r_m (%)	r_n (%)
gr_01	12762	15100	11623	13146	8.92	12.94
gr_02	38676	47340	36648	41850	5.24	11.60
gr_03	38081	46471	36132	41309	5.12	11.11
gr_04	15380	18202	14103	15896	8.30	12.67
gr_05	36352	44354	34758	39580	4.38	10.76
gr_06	40203	49077	37972	43360	5.55	11.65
gr_07	39595	48222	37841	43140	4.43	10.54
gr_08	9286	11007	8198	9239	11.72	16.06
gr_09	40637	49556	38497	43849	5.27	11.52
gr_10	37300	45609	35266	40347	5.45	11.54
gr_11	39869	48670	37995	43470	4.70	10.68
gr_12	3629	4094	2765	2987	23.81	27.04
gr_13	14912	17664	13819	15613	7.33	11.61
gr_14	17047	20117	15601	17551	8.48	12.76
gr_15	11561	13667	10652	11988	7.86	12.29

Table 6.12: The total number of required constraints and alive variables produced by AlgILP and AlgILP + K for the cyclic grids.

Instance	AlgILP	AlgILP + K
gr_01	0	0
gr_02	0	1
gr_03	0	0
gr_04	0	0
gr_05	0	0
gr_06	0	1
gr_07	0	0
gr_08	0	1
gr_09	0	1
gr_10	0	0
gr_11	0	2
gr_12	0	0
gr_13	0	0
gr_14	0	0
gr_15	0	2

Table 6.13: The number of ILP solver restarts for AlgILP and AlgILP + K for the tournaments

Chapter 7

Branch & Reduce Evaluation

Recall that we integrated the kernelization from Chapter 3 in each recursive call of a Branch & Bound algorithm, obtaining a Branch & Reduce algorithm in Chapter 5. In this chapter we discuss how effective and practical our reduction rules are for a Branch & Bound algorithm. We first discuss how we will evaluate our original Branch & Bound algorithm, and the Branch & Reduce algorithm. We then discuss which instances we will be using and how they are generated. We then gather and discuss the results, followed by a conclusion of the most important results.

7.1 Evaluation

Let us first briefly recall how the Branch & Reduce algorithm from Chapter 5 works. Initially, we first perform a simple kernelization using only the nonrestricted reduction rules. Then, this resulting graph is partitioned into its strongly connected components, and then for each of those components we compute a good upper bound and start the Branch & Reduce algorithm. During each recursive call of the Branch & Reduce algorithm we first apply the complete kernelization, see Split & Reduce 1, obtaining G_d and G_b. We then test whether we already exceeded our budget, or our lower bound exceeds the budget. If we did not exceed our budget for both cases, we choose a set of vertices H to be part of the solution of $G_d + G_c$, which we remove from a copy of G_d+G_b, obtaining the next subproblem. We maintain the smallest solution of G_d+G_b, if it exists, and return it. In the end, all the results of the Branch & Reduce applications or the upper bounds, if proven optimal, for each strongly connected component are then combined with the set of forced vertices found during the simple kernelization, recovering the optimal solution.

For our experiments, we disable this initial simple kernelization. Additionally, we can easily revert our Branch & Reduce algorithm back to a Branch & Bound algorithm by taking out the kernelization at each node of the search tree. Instead of applying the kernelization, we simply split the current directed graph G' into a monodirected and a bidirected graph G_d and G_b by moving all bidirectional edges to a new graph G_b, similar to our approach in Chapter 6. Since we have no set of forced vertices, we no longer verify whether we already exceed the budget.

Furthermore, we decide to leave out the VERTEX COVER solver of Hespe et al. we included in our kernelization (see also Chapter 3), since their algorithm runs in several Branch & Reduce and Branch & Bound phases, making it unclear for us which of both of their algorithms managed to produce the solution. We have the following two objectives in our analysis.

1. Firstly, we aim to investigate whether a kernelization may accelerate a Branch & Bound algorithm, in the form a Branch & Reduce algorithm. We will measure the running time of an instance for both the Branch & Bound algorithm and the Branch & Reduce algorithm. We will later refer to the Branch & Reduce algorithm and the Branch & Bound algorithm, i.e., without any enabled reduction rules, as "BnR" and "BnB" respectively.

2. Secondly, we investigate whether a kernelization can be used more sparingly in a Branch & Bound to further decrease the running time.

To elaborate on the second objective, we note that our kernelization algorithm can be made significantly more efficient, like we briefly discussed in Chapter 6. This may mean that during the Branch & Reduce algorithm we spend a significant amount of time verifying whether we can apply each reduction rule, without eventually reducing the graph. Clearly, this is costly when performed at each node of the search tree.

Additionally, another problem with our Branch & Reduce algorithm is that the vertices which we include for our current subproblem and remove from its copy to obtain our next subproblem are not necessarily good vertices to remove for the kernelization: these vertices may not enable any reduction rules, or may only enable a limited amount of applications, compared to another vertex occurring in at least one cycle. It is however hard to decide beforehand what would be a good choice of a vertex for the kernelization.

Hence, given both problems, it may be better to alternate our Branch & Reduce algorithm with our Branch & Bound algorithm, e.g., every 2 levels, see Figure 7.1. Here, we consider a search tree where we have sufficiently many vertices in the bidirected graph G_b, causing the consistent fan-out of two branches at a time, but the number of branches per node may be larger when we need to branch over a smallest cycle of G_d.

Not only do we prevent verifying whether any reduction rule is even applicable, we hopefully have managed to create a subproblem where we can perform a significant amount of work on later on. On the other hand however, we may inadvertently increase the running time because we are not applying the kernelizations frequently enough. We will always start

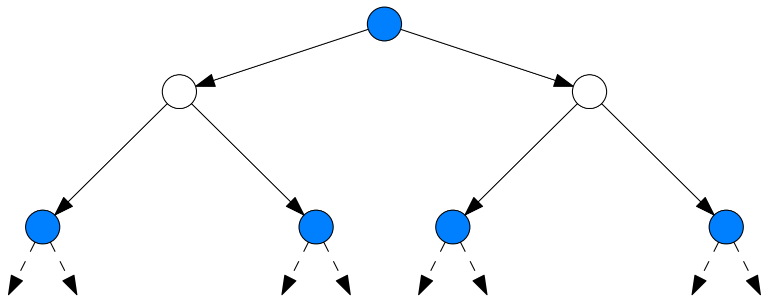

Figure 7.1: Alternating a Branch & Reduce with a Branch & Bound algorithm every 2 levels of the resulting search tree. All the blue nodes in the search tree are the nodes where we use the Branch & Reduce algorithm on the current graph.

with a Branch & Reduce algorithm, and then use a Branch & Bound every $k \in \{5, 10, 15, 20, 25\}$ levels. For clarity, we will henceforth denote this hybrid variant as BnR/k.

7.2 Instances

In Chapter 6 we generated two sets of instances, tournaments and cyclic grids. As a relatively fair comparison, we aim to reuse these instances — we refer to Section 6.3 for the structure of these graphs. However, since BnB is rather slow compared to our ILP-based algorithm, we generate two new pools of instances such that we obtain a good set of instances for BnB. We shortly describe the parameters used to generate the new pools of instances and our expectations for how well BnR works on these instances. Instances were selected in a similar fashion compared to Chapter 6. In order to gather the results as quickly as possible, we used 20 solver threads on the initial instance pools. These resulting running times were used to collect the slowest 15 instances using BnB. Subsequent experiments were performed with at most 10 solver threads at a time.

Additionally, we will also use the PACE instances, and we further investigate the 15 slowest instances that terminate within 30 minutes for BnR. This set includes six instances for which BnB exceeded the 30 minute time-limit.

We do note that we use 20 threads to collect all results of the PACE instances, which skews the running times a little more, but all solutions are found within the first 105 instances, so this should not matter significantly. For the 15 selected PACE instances, we use 10 threads again.

7.2.1 Tournaments

Recall that we require the number of vertices, an approximate density $0 \leq d \leq 1$ and a probability $0 \leq p < 1$ to introduce a bidirectional edge in the tournament. We generated tournaments starting with 200 vertices, up to 700 vertices, with increments of 50 vertices. For the density d, we start from 0.05, up to 0.3, with increments of 0.05. Finally, the probability to introduce a bidirectional edge p starts from 0.2, up to 0.5, with increments of 0.05. The selected instances[1] are then as follows, see Table 7.1. Instances were labeled according the ascending order in the running time difference between BnR and BnB + K.

Since these tournaments are relatively hard instances for our kernelization, we expect that the resulting subproblems after selecting and removing one or more vertices are also relatively hard instances for our kernelization. Therefore, we expect that the BnR and the BnB algorithm perform about equally well. Furthermore, we therefore expect that using the kernelizations more sparingly will probably decrease the running time, since we no longer spend as much time on the kernelizations.

Instance	$\|V\|$	$\|E\|$	d	p
bt_01	650	3089	0.20	0.30
bt_02	550	2318	0.20	0.20
bt_03	650	3586	0.20	0.50
bt_04	550	3403	0.25	0.30
bt_05	700	3113	0.20	0.20
bt_06	350	2218	0.25	0.50
bt_07	250	1319	0.25	0.30
bt_08	350	2254	0.25	0.45
bt_09	450	2826	0.25	0.35
bt_10	550	3531	0.25	0.35
bt_11	300	2249	0.30	0.35
bt_12	300	2427	0.30	0.45
bt_13	200	1537	0.30	0.50
bt_14	200	1235	0.30	0.20
bt_15	400	2277	0.25	0.25

Table 7.1: Specification of the selected tournaments.

[1]While we intended to disable all reduction rules, we accidentally still enabled the initial simple kernelization. We will, however, keep the same set of instances. We will refer to the algorithm with these settings as BnB + K.

7.2.2 Cyclic Grids

For the Cyclic grids, recall that we choose the number of vertices n as $\lceil\sqrt{x}\rceil^2$ for some number x. We let x start at 400, up to 10,000, with increments of 200. Additionally, the probability to introduce a bidirectional edge starts from 0.05, up to 0.8, with increments of 0.05. The selected instances are then as follows, see Table 7.2. Similarly to the selected tournaments, these instances were labeled according the ascending order in the running time difference between BnR and BnB.

Recall that the cyclic grids were designed such that they were easy to reduce with the simple kernelization, i.e., before splitting the directed graph. By introducing the bidirectional edges, we made these graphs harder to reduce, but most likely not impossible to be reduced. Especially with repeated applications of the complete kernelization, we expect that these instances can be significantly reduced further down the search tree. Therefore, we expect that BnB will be significantly slower than the BnR algorithm on many of these instances. Furthermore, by delaying the kernelizations in the search tree, we expect that the running time will significantly increase the longer we delay the kernelizations.

Instance	$\|V\|$	$\|E\|$	p
bgr_01	3844	13217	0.75
bgr_02	3249	10855	0.70
bgr_03	2601	8668	0.70
bgr_04	6241	19706	0.60
bgr_05	3600	12722	0.80
bgr_06	3249	10533	0.65
bgr_07	2601	9429	0.85
bgr_08	1225	2492	0.05
bgr_09	2401	8462	0.80
bgr_10	3600	10984	0.55
bgr_11	3481	11321	0.65
bgr_12	2809	9696	0.75
bgr_13	5625	17294	0.55
bgr_14	5476	14559	0.35
bgr_15	6241	20389	0.65

Table 7.2: Specification of the selected cyclic grids.

7.2.3 Measurement Deviations

In the subsequent sections we run the experiments once, as we did in Chapter 4. We measured the running time of BnR on all tournament instances, starting with 10 threads like in most experiments, and picked one instance,

Run	Running time
1	23.39
2	22.29
3	23.67
4	23.78
5	23.65
6	23.86

Table 7.3: Running times (in seconds) of instance bt_02 when started in a pool of 15 instances and 10 available threads.

namely bt_02, and measured a standard deviation of 580ms over 6 separate runs, see Table 7.3. Henceforth, we assume that the single measurements are therefore relatively accurate, but running times that are close to each other, i.e., within 1 second, about twice the standard deviation, should be considered about the same.

7.3 Results

7.3.1 PACE Instances

Using BnB, we managed to find 65 solutions, and BnR managed to find 76 solutions. We observe that BnR also managed to find the solutions BnB found, clearly showing the strength of applying reductions in a (general) Branch & Bound algorithm. We compare the running time of all 76 solutions, assigning a default of 1800 seconds when BnB did not recover the solution, but BnR did.

For those 11 solutions that could only be found with BnR, the amount of time saved is significant, saving −1700 to −1600 seconds. Additionally, only the two instances required more time when using BnR: an additional 0.45s, and 0.68s, respectively. This is a clear indication that including reductions in a Branch & Bound algorithm is a very good choice to reduce the total running time, even if more computational work is performed per node of the search tree. For the remaining instances, a smaller decrease in the running time was observed, saving between 50 seconds and 1.34 seconds.

When looking at the results of the PACE challenge, this solver would have obtained a second-to-last place with 76 solved instances, see Table 6.1. The major difference between our current solver and the best performing Branch & Reduce algorithms is that those teams used significantly more reduction rules, some of which we discussed in Chapter 3. The winner of last year's competition [3], which focused on the CLUSTER EDITING problem, managed to already completely reduce their input graph on many instances before even starting a Branch & Reduce algorithm. Furthermore, their lower bounds and upper bounds were also very strong, i.e., equal in size in many cases, which contributed to solving the instances more easily. Clearly, achieving such results requires a significant cost in development.

Finally, we take the 15 slowest instances for which BnR managed to find a solution within 30 minutes per instance. For 6 instances, namely e_065, e_077, e_086, e_096, e_097, and e_098, BnB failed to obtain a solution within 30 minutes. The other 9 instances contain instances for which BnB algorithm was either not significantly slower or was significantly slower.

Since the solutions were all recovered within 5 minutes, we enforce a time limit of 10 minutes when delaying the kernelization. For most instances, alternating BnR with BnB hardly matters in the running time. However, for e_105, we see that we fail to obtain a solution within 10 minutes for BnR/5, but for the others we manage to obtain a solution significantly faster compared to BnR.

For e_065, an instance for which BnB failed to report a solution within 30 minutes, alternating BnR with BnB does not really seem to affect the solvability of the instance. For BnR/25, only 33.70 seconds are needed to solve the instance. On the other hand, for e_077, we see that the running time significantly increases the more we wait with a kernelization. This suggests that it is vital that the kernelization is performed as early as possible in the search tree.

Ins.	BnR	BnR/5	BnR/10	BnR/15	BnR/20	BnR/25	Class.
e_008	14.52	13.39	13.56	13.45	13.42	13.74	o
e_065	12.22	12.29	17.02	29.58	16.87	33.70	-
e_068	10.07	11.34	13.98	13.69	20.21	20.80	-
e_073	12.07	11.52	12.12	12.39	12.19	13.02	o
e_077	77.79	89.13	140.19	224.96	418.72	453.38	-
e_080	39.89	23.43	17.67	17.75	18.21	17.97	+
e_081	19.07	15.82	19.72	21.38	30.77	22.52	-
e_082	9.60	9.01	8.90	9.24	9.44	9.57	o
e_086	187.54	211.44	272.85	277.88	467.03	-	-
e_093	76.35	72.87	75.72	75.93	75.73	76.45	o
e_096	20.00	19.64	20.28	21.58	20.91	21.13	o
e_097	48.72	46.37	48.63	48.14	50.02	49.44	o
e_098	108.09	104.56	107.80	107.56	112.53	111.10	o
e_099	10.06	9.90	10.10	10.12	10.41	9.85	o
e_105	295.51	-	399.03	182.07	186.44	184.84	-/+

Table 7.4: Running time (in seconds) of BnR and BnR/k, where $k \in \{5, 10, 15, 20, 25\}$, for the PACE instances.

7.3.2 Tournaments

As briefly mentioned during the instance selection of the tournaments, we also compute the running time of the BnB algorithm with the initial simple kernelization, i.e., BnB + K. The instances are sorted in ascending order by the difference between the running time of the BnB + K and the Branch & Reduce algorithms.

We first observe that BnR is slower on only one instance compared to BnB, namely for bt_07. This clearly shows that sometimes a kernelization may not be effective. In general, it is clear that using the kernelizations still helped reducing the running time of these instances compared to BnB, which we did not necessarily expect. This may be due to the fact that eventually, in the search tree, we have removed sufficiently many vertices to start reducing more. For the last 4 instances, BnR is marginally faster with a difference of at most 10 seconds.

If we compare the BnB + K algorithm to our Branch & Reduce algorithm, we see that only for instance bt_15 did the Branch & Reduce require more time, and only just under 2 seconds. Clearly, this shows that it is beneficial to continue using kernelizations instead of only using them before running BnB. The BnB + K algorithm is however slower on three instances compared to BnB, but for BnR these same instances are faster than BnB, stressing the fact performing the kernelization frequently is important.

We now focus on whether we can use a kernelization more sparingly, and

Instance	BnB	BnB + K	BnR
bt_01	226.46	160.47	59.04
bt_02	30.56	81.64	19.78
bt_03	238.01	189.75	143.53
bt_04	203.47	184.85	143.00
bt_05	948.07	187.74	151.34
bt_06	129.58	122.80	88.92
bt_07	1.90	45.29	21.83
bt_08	92.82	88.24	70.23
bt_09	55.18	68.30	51.10
bt_10	88.99	67.27	57.50
bt_11	75.75	69.27	60.03
bt_12	80.51	78.82	70.15
bt_13	78.17	72.70	65.53
bt_14	67.92	61.52	55.59
bt_15	51.51	45.95	47.65

Table 7.5: The running time (in seconds) of BnB, the Branch & Bound algorithm with the initial simple kernelization enabled (BnB + K), and BnR for the tournaments.

alternate the Branch & Reduce algorithm with a Branch & Bound algorithm, see Table 7.6. While for some number of instances, such as bt_14 and bt_15, delaying the kernelization barely matters. For the majority of the instances, alternating BnR with BnB definitely has a negative impact in the running time, such as for instances bt_03, bt_04 and bt_05, that each time require more solve time. This again suggests that further down the search tree, we can start reducing more.

The large exception is instance bt_01, that needed almost an entire minute when the kernelization was applied every level of the search tree, to needing under 4 seconds when the kernelization was only applied every 25 levels. This instance clearly benefitted from the fact that we waited with applying a kernelization — although the running time did increase for BnR/5. This may be due to the fact that we were able to find a good set of vertices to remove which enabled the use of many applications of the reduction rules.

Ins.	BnR	BnR/5	BnR/10	BnR/15	BnR/20	BnR/25	Class.
bt_01	59.04	77.92	5.10	4.41	4.01	3.77	+
bt_02	19.78	14.52	24.14	31.79	54.43	34.30	-
bt_03	143.53	149.81	149.34	158.85	182.23	187.37	-
bt_04	143.00	149.68	151.71	155.28	165.59	170.66	-
bt_05	151.34	167.14	161.95	171.91	161.90	177.66	-
bt_06	88.92	102.16	107.78	98.84	108.92	116.10	-
bt_07	21.83	26.21	28.31	30.91	31.39	29.50	-
bt_08	70.23	77.63	79.52	79.57	79.72	83.65	-
bt_09	51.10	50.43	52.18	79.70	50.50	52.61	o
bt_10	57.50	51.57	54.35	50.45	52.06	57.58	+
bt_11	60.03	62.35	64.29	62.29	67.32	70.58	-
bt_12	70.15	72.81	77.92	76.45	72.97	74.94	-
bt_13	65.53	67.54	70.99	71.64	70.94	72.76	-
bt_14	55.59	55.13	57.70	55.79	56.71	57.24	o
bt_15	47.65	47.52	47.69	50.07	45.31	49.19	o

Table 7.6: Running time (in seconds) of BnR and BnR/k, where $k \in \{5, 10, 15, 20, 25\}$, for the tournaments.

7.3.3 Cyclic Grids

For the cyclic grids, the results are more in line with our expectations. In Table 7.7 we determined the running time for BnR and BnB. As mentioned above, we labeled and sorted the instances according to their difference — the smaller the better the BnR performed. We observe that for every instance a significant amount of time was saved by applying the kernelization at each node of the search tree, as expected. The smallest difference, i.e.,

Instance	BnB	BnR
bgr_01	284.73	67.65
bgr_02	254.07	54.02
bgr_03	229.35	33.32
bgr_04	298.45	119.89
bgr_05	234.51	65.30
bgr_06	203.47	35.84
bgr_07	202.21	44.62
bgr_08	160.33	4.77
bgr_09	155.06	38.25
bgr_10	147.07	38.97
bgr_11	146.73	41.75
bgr_12	142.09	37.16
bgr_13	158.73	97.35
bgr_14	142.93	89.24
bgr_15	141.83	112.70

Table 7.7: Running time (in seconds) of BnB and the BnR for the cyclic grids.

for bgr_15, was already a decrease in just under 30 seconds when using BnR. For bgr_01 almost 2 minutes were saved.

When we look at whether the kernelization can be delayed, we see that for the majority of instances we indeed increase the running time the longer we wait with applying a kernelization again, see Table 7.8. Only for two instances, bgr_14 and bgr_15, the running time slightly decreased.

7.4 Conclusion

We have seen that a kernelization is definitely a very useful tool in accelerating a Branch & Bound algorithm. Even for instances that are hard to reduce, such as our tournaments, the repeated application of a kernelization in our BnR algorithm proved to be helpful. In some situations however, a Branch & Reduce may actually be slower than the Branch & Bound algorithm, but this was a rare occurrence for the investigated instances. However, to truly obtain a practical Branch & Reduce algorithm, one may need to invest a large amount of time in the development of good reduction rules, upper bounds and lower bounds.

What was particularly clear for the PACE instances is that the kernelizations need to be applied as early as possible to see a good decrease in the running times. However, as we saw for both the tournaments and cyclic grids, the running time can be further improved for a select number of instances by delaying the kernelization. This should, perhaps be decided with

Ins.	BnR	BnR/5	BnR/10	BnR/15	BnR/20	BnR/25	Class.
bgr_01	67.65	66.67	67.31	71.14	70.99	78.56	-
bgr_02	54.02	58.08	60.05	63.65	65.57	75.31	-
bgr_03	33.32	35.38	40.95	41.60	42.60	46.84	-
bgr_04	119.89	118.41	116.25	116.52	116.10	117.52	o
bgr_05	65.30	68.94	70.83	73.83	77.28	84.74	-
bgr_06	35.84	34.92	38.29	36.49	37.94	42.84	-
bgr_07	44.62	50.16	54.63	57.36	63.24	71.38	-
bgr_08	4.77	7.14	9.97	10.70	13.92	10.84	-
bgr_09	38.25	39.13	43.91	47.31	49.46	53.54	-
bgr_10	38.97	37.10	38.56	37.60	39.27	40.10	o
bgr_11	41.75	41.66	42.55	42.28	43.81	44.65	-
bgr_12	37.16	38.75	42.04	43.45	45.61	49.69	-
bgr_13	97.35	97.50	94.36	96.87	94.06	97.57	o
bgr_14	89.24	87.51	84.05	85.68	84.90	86.11	+
bgr_15	112.70	112.22	108.72	110.91	108.62	109.47	+

Table 7.8: Running time (in seconds) of BnR and BnR/k, where $k \in \{5, 10, 15, 20, 25\}$, for the cyclic grids.

a different metric than the current depth of the search tree, as still the majority of instances did not see any benefit when delaying the kernelization.

Chapter 8

Conclusion

The goal of this thesis was to determine whether the kernelization of a Directed Feedback Vertex Set instance could accelerate common exact algorithmic methods, like an ILP algorithm or a Branch & Bound algorithm. Additionally, for the ILP algorithm, we wondered whether the resulting ILP models were actually smaller when a kernelization was used, given that we were computing a subset of all the possible constraints to solve the ILP. For the Branch & Bound algorithm, we wondered about the necessity of applying the reduction rules as frequently as possible. Before coming to the overall conclusions, we briefly summarize all our previous work.

In Chapter 3, we have seen that a directed graph G can be reduced and split into a monodirected graph G_d, a bidirected graph G_b, and a set of forced vertices S, such that an optimal solution of $G_d + G_b$, together with S is an optimal solution of G.

We then designed an ILP-based algorithm that generated a relatively good set of cycles based on finding an edge cycle cover of G_d+G_b. Additional sets of cycles, and thus constraints, were generated when a relatively smaller upper bound of the current constraints did not turn out to be a DFVS of G_d, or when the exact solution of the ILP was not a DFVS of G_d.

For our Branch & Bound algorithm, we integrated the kernelization in each recursive call, obtaining a Branch & Reduce algorithm. To prune the search space, we drew inspiration form our ILP-based algorithm to compute relatively good upper and lower bounds.

In Chapter 6, we concluded that while kernelizations certainly can accelerate the overall running time of the ILP-based algorithm (including the ILP solver), this was usually only limited to instances that were significantly reduced. When the kernelization does not manage to significantly reduce the input graph, a strong ILP solver, like Gurobi, may not benefit from the kernelization. We determined that Gurobi has sufficiently strong generic preprocessing, that reductions may not even be necessary to obtain an efficient algorithm.

Additionally, the resulting models after the kernelization generally used fewer variables and constraints for all but one of the investigated instances. As a bonus, we also determined that our ILP-based algorithm manages to outperform the best solver of the PACE 2022 challenge.

However, for our Branch & Reduce algorithm we determined that a kernelization is certainly useful to accelerate a Branch & Bound algorithm. We have also seen that for some instances we can definitely refrain from applying a kernelization in each level of the Branch & Reduce search tree, but generally speaking, the more the kernelizations are performed early in the search tree, the smaller the running time. Unfortunately, to truly obtain a practical Branch & Reduce algorithm, it is key to have very strong reduction rules. In that case, setting up an ILP-based algorithm may be easier, as reductions may not even be necessary to implement: setting up a good model is far more important.

8.1 Future Work and Improvements

Given that our kernelization has been implemented relatively naively, we disregarded the running time of the kernelization during our evaluation of the ILP-based algorithm. However, this is certainly a contributing cost. Since we also need more reduction rules to see a significant decrease in the running time of purely the ILP-based algorithm, the question is therefore whether we will see an overall decrease in the running time in this situation, i.e., more reductions but a more efficient kernelization.

For the Branch & Reduce algorithm, more efficient kernelizations also help in determining whether we can use the kernelizations more sparingly throughout the search tree. Additionally, we can possibly further accelerate the Branch & Reduce algorithm by choosing vertices occurring in a cycle that enable the application of many reduction rules. Our branching strategy is in this sense decoupled from the kernelizations we designed, which may not be ideal.

Finally, to the best of our knowledge, we have not seen a “revert kernelization” data structure to efficiently revert the kernelization process, or general vertex deletion operations. Such a data structure may be useful in a Branch & Reduce algorithm, i.e., primarily for reducing the number of copies of our graphs that we make throughout the search. These copies are especially costly as they require the operating system to allocate new memory, in the worst case, an exponential number of times.

Bibliography

[1] Benjamin Bergougnoux, Eduard Eiben, Robert Ganian, Sebastian Ordyniak, and M. S. Ramanujan. Towards a polynomial kernel for directed feedback vertex set. *Algorithmica*, 83(5):1201–1221, may 2021.

[2] Alan A. Bertossi. Covering trains by stations or the power of data reduction. 1998.

[3] Thomas Bläsius, Philipp Fischbeck, Lars Gottesbüren, Michael Hamann, Tobias Heuer, Jonas Spinner, Christopher Weyand, and Marcus Wilhelm. A branch-and-bound algorithm for cluster editing. In *SEA*, 2022.

[4] Jianer Chen, Yang Liu, Songjian Lu, Barry O'sullivan, and Igor Razgon. A fixed-parameter algorithm for the directed feedback vertex set problem. *J. ACM*, 55(5), nov 2008.

[5] Holger Dell, Thore Husfeldt, Bart M. P. Jansen, Petteri Kaski, Christian Komusiewicz, and Frances A. Rosamond. The First Parameterized Algorithms and Computational Experiments Challenge. In *11th International Symposium on Parameterized and Exact Computation (IPEC 2016)*, volume 63 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:9, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[6] Michael R. Fellows, Lars Jaffke, Aliz Izabella Király, Frances A. Rosamond, and Mathias Weller. *What Is Known About Vertex Cover Kernelization?*, pages 330–356. Springer International Publishing, Cham, 2018.

[7] Damir Ferizovic, Demian Hespe, Sebastian Lamm, Matthias Mnich, Christian Schulz, and Darren Strash. *Engineering Kernelization for Maximum Cut*, pages 27–41.

[8] Rudolf Fleischer, Xi Wu, and Liwei Yuan. Experimental study of FPT algorithms for the directed feedback vertex set problem. In *Algorithms - ESA 2009*, pages 611–622, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[9] Daniel Funke, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. Communication-free massively distributed graph generation. In *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21 – May 25, 2018*, 2018.

[10] Philippe Galinier, Eunice Lemamou, and Mohamed Wassim Bouzidi. Applying local search to the feedback vertex set problem. *Journal of Heuristics*, 19(5):797–818, Oct 2013.

[11] Demian Hespe, Sebastian Lamm, Christian Schulz, and Darren Strash. *WeGotYouCovered: The Winning Solver from the PACE 2019 Challenge, Vertex Cover Track.* 2020.

[12] Igor Razgon. *Computing A Directed Feedback Vertex Set in* $O^*(1.9977^n)$, pages 70–81. in Proceedings of the 10th Italian Conference on Theoretical Computer Science. World Scientific, 2007.

[13] Christian Schulz, Ernestine Großmann, Tobias Heuer, and Darren Strash. Private Communication, June 2022.

[14] Darren Strash. On the power of simple reductions for the maximum independent set problem. In *Computing and Combinatorics*, pages 345–356, Cham, 2016. Springer International Publishing.