Universidade Estadual de Campinas

Instituto de Computação

Ieremies Vieira da Fonseca Romero

A Branch-and-Price Algorithm for the Graph Coloring Problem

Algoritmo de Branch-and-Price para o Problema de Coloração de Grafos

CAMPINAS

2025

Ieremies Vieira da Fonseca Romero

A Branch-and-Price Algorithm for the Graph Coloring Problem

Algoritmo de Branch-and-Price para o Problema de Coloração de Grafos

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Dissertation presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Rafael Crivellari Saliba Schouery

Este exemplar corresponde à versão final da Dissertação defendida por Ieremies Vieira da Fonseca Romero e orientada pelo Prof. Dr. Rafael Crivellari Saliba Schouery.

CAMPINAS

2025

Ficha catalográfica
Universidade Estadual de Campinas (UNICAMP)
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

R664b
Romero, Ieremies Vieira da Fonseca, 2000-
A branch-and-price algorithm for the graph coloring problem / Ieremies Vieira da Fonseca Romero. – Campinas, SP : [s.n.], 2025.

Orientador: Rafael Crivellari Saliba Schouery.
Dissertação (mestrado) – Universidade Estadual de Campinas (UNICAMP), Instituto de Computação.

1. Programação linear inteira. 2. Coloração de grafos. 3. Pesquisa operacional. I. Schouery, Rafael Crivellari Saliba, 1986-. II. Universidade Estadual de Campinas (UNICAMP). Instituto de Computação. III. Título.

Informações complementares

Título em outro idioma: Algoritmo de branch-and-price para o problema de coloração de grafos
Palavras-chave em inglês:
Integer linear programming
Graph coloring
Operational research
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Rafael Crivellari Saliba Schouery [Orientador]
Flávio Keidi Miyazawa
Teobaldo Leite Bulhões Júnior
Data de defesa: 24-03-2025
Programa de Pós-Graduação: Ciência da Computação

Objetivos de Desenvolvimento Sustentável (ODS)
ODS: 9. Inovação e infraestrutura

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0002-5801-3774
- Currículo Lattes do autor: https://lattes.cnpq.br/3216045598602403

- Prof. Dr. Rafael Crivellari Saliba Schouery
  Instituto de Computação - UNICAMP
- Prof. Dr. Flávio Keidi Miyazawa
  Instituto de Computação - UNICAMP
- Prof. Dr. Teobaldo Leite Bulhões Júnior
  Centro de Informática - UFPB

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Agradecimentos

Primeiramente, gostaria de agradecer a minha mãe Ana Cláudia, meu irmão Jeremias e meu avô Milton. Por todo apoio incondicional a todos os desafios. Por todos os conselhos de vida e pela educação que me deram, mas também por todas as vezes que voltaram pois eu havia esquecido o uniforme de educação física. Vocês são a melhor família que eu poderia pedir.

À minha namorada, Jasmine, agradeço pelo amor, pelo carinho, por todos os bolos e biscoitos. Por todas as infindáveis reclamações que escutou, mas também por ter me ajudado a dar valor a todas as coisas que fiz. Obrigado por sempre estar ao meu lado, por chorar e rir comigo ao longo desta jornada. Minha vida não teria tantas cores sem sua presença.

Gostaria de agradecer também aos meus amigos mais próximos do dia a dia de pesquisa: Carol, Elisa, João e Renan. Seja pelas inúmeras revisões deste e outros textos, seja pela fofoca nossa de cada dia, esses últimos 2 anos não teriam sido nada sem vocês. Agradeço à Camila, Giovanna, Gustavo e Thales, por todas as conversas intermináveis, discussões filosóficas e fofocas. Obrigado por sempre estarem lá quando precisei.

Agradeço a todos os meus amigos de Fortaleza: Amorim, Arthur, Aloysio, Julio, Matheus, Nishimura, Pedro, Perdigão, Renê, Zeca, Paulinho e Paulão. As piadas mais sem graça, as discussões filosóficas e os servidores de Terraria foram fundamentais para preservar minha sanidade ao longo destes anos.

Ao meu orientador, Prof. Rafael Schouery serei eternamente grato por, naquela aula de MC202 em 2018, ter tomado o tempo de responder um garoto que, mal chegado na graduação, nem sonhava em qual seria os passos para a docência universitária. Obrigado por ser um orientador atencioso e acolhedor, por compreender meus momentos difíceis e me incentivar na retomada. E, acima de tudo, por ser minha maior inspiração como professor e pesquisador.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001. Além disso, este trabalho recebeu apoio do Convênio entre a Universidade Estadual de Campinas e a Banco Santander. As opiniões, hipóteses e conclusões ou recomendações expressas neste material são de responsabilidade do autor e não necessariamente refletem a visão do Santander.

Resumo

O problema de coloração de vértices, um tema central da Teoria dos Grafos, consiste em atribuir cores aos vértices de um grafo de modo que vértices adjacentes não compartilhem a mesma cor, minimizando o número de cores utilizadas. Amplamente aplicado em escalonamento, alocação de recursos e otimização de redes, esse problema desempenha um papel crucial na modelagem e resolução eficiente de cenários reais. Esta dissertação explora métodos para resolver o problema de coloração de vértices por meio de formulações existentes de Programação Linear Inteira (PLI) e do método branch-and-price. Um algoritmo de *branch-and-reduce* é empregado pela primeira vez no problema de precificação correspondente, integrando regras de separação e redução para simplificar a complexidade das instâncias, mantendo a integridade da solução. Testamos a hipótese de que enumerar um maior número de colunas a cada etapa da abordagem de geração de colunas pode resultar em um menor tempo de execução total. Combinamos essa estratégia com um conjunto de heurísticas para encontrar soluções viáveis, permitindo interromper o processo de geração de colunas mais cedo. Além disso, os limites gerados por estas abordagens são utilizados para realizar separações e reduções ainda mais eficientes no problema original. Experimentos computacionais em conjuntos de dados de referência DIMACS e MATILDA validam os métodos propostos, demonstrando melhorias significativas em eficiência.

Abstract

The vertex coloring problem, a cornerstone of Graph Theory, involves assigning colors to vertices of a graph such that no two adjacent vertices share the same color, minimizing the number of colors used. Widely applied in scheduling, resource allocation, and network optimization, this problem plays a crucial role in modeling and solving real-world scenarios efficiently. This dissertation explores methods for solving the vertex coloring problem through exisiting integer linear programming (ILP) formulations and the branch-and-price. A branch-and-reduce framework is used for the first time on the corresponding pricing problem, integrating separation and reduction rules to streamline instance complexity while maintaining solution integrity. We test the hypothesis that enumerating a larger number of columns at each step of the column generation approach might yield faster running time overall. This strategy is combined with a set of heuristics to find feasible solutions, which also allow the column generation process to halt earlier. Additionally, the bounds generated by these approaches are integrated to enhance the effectiveness of separation and reduction rules for the original problem. Computational experiments on benchmark datasets DIMACS and MATILDA validate the proposed methods, demonstrating significant improvements in efficiency.

It starts with one.

Contents

Chapter 1

Introduction

The vertex coloring problem, often referred to as the graph coloring problem, is a classical problem in graph theory that has fascinated researchers for decades. The problem seeks to assign colors to the vertices of a graph such that no two adjacent vertices share the same color, while minimizing the total number of colors used. This deceptively simple problem has a rich history, tracing back to early studies in map coloring and the famous Four Color Theorem [2].

The problem of deciding whether a graph admits a coloring with k colors, for any $k > 2$, is NP-complete, and its optimization version, i.e., finding the minimum value of k for which there is a coloring with k colors, is NP-hard [29]. Despite the continued progress in algorithm design and computational resources, instances with as few as 100 vertices still pose significant challenges for exact algorithms, underscoring the persistent complexity of the problem. This computational challenge emphasizes the need for robust and innovative approaches to tackle both theoretical and practical aspects of the problem.

Understanding and solving the vertex coloring problem is not merely of theoretical interest. It has naturally emerges across a wide range of domains, including scheduling, resource allocation, frequency assignment, and register allocation in compilers [27, 90, 83, 8, 11, 84, 12, 75, 28]. For such applications, exact algorithms for vertex coloring is particularly compelling, as these methods provide optimal solutions and deepen our understanding of the structural properties of graphs. By leveraging insights from combinatorics, optimization, and computer science, exact algorithms offer a powerful lens through which to study this problem, shedding light on its complexities and pushing the boundaries of what is computationally achievable.

Recent advances in optimization methods, particularly in *Branch-and-Price* approaches for the bin-packing and vehicle routing problems, have inspired new directions in tackling the graph coloring problem. Innovations such as flow-based formulations and the strategic addition of cutting planes during the branching process demonstrates significant improvements in solving complex combinatorial problems [74, 64]. This progress motivated us to investigate how these techniques could be adapted and applied to the graph coloring problem.

Many commonly employed techniques, including those involving floating-point arithmetic and complex linear programming solvers, are susceptible to rounding errors and precision issues [3]. These errors can compromise the correctness of decisions made during the algorithm, potentially leading to incorrect solutions or invalid proofs of optimality. This concern highlights the importance of designing numerically safe algorithms that ensure robustness and

accuracy throughout the optimization process, such as done by Baldacci et al. [3] and Held et al. [33].

Furthermore, exact methods for solving the maximum weighted independent set problem — the pricing subproblem in Branch-and-Price formulations for graph coloring — have also seen remarkable developments in recent years [56, 82, 45, 87]. These advancements open new possibilities for improving exact algorithms, both in terms of computational efficiency and the ability to handle challenging instances.

Building on these advances, we approach the vertex coloring problem through a Branch-and-Price framework integrated with a Branch-and-Reduce strategy to solve the pricing problem. The methodology incorporates numerically safe techniques to ensure reliability in optimization processes. Furthermore, we investigate whether generating a larger pool of columns during each column generation iteration improves computational efficiency. Finally, we extensively employ heuristics to iteratively refine the upper bound throughout the optimization process.

The structure of this dissertation is as follows. In Chapter 2, we introduce pre-processing techniques we employ to split and reduce the problem, aiming to simplify instances before solving them. In Chapter 3, we focus on solving the linear relaxations via the column generation approach, with particular attention to handling numerical instability. In Chapter 4, we present our Branch-and-Reduce algorithm, inspired by Xiao et al. [87], which is applied to the pricing problem. In Chapter 5, we demonstrate the effectiveness of our overall approach through computational experiments on commonly used benchmark instances. Finally, in Chapter 6, we conclude the dissertation with a summary of our findings and a discussion of potential directions for future research.

1.1 Applications

The graph coloring problem has many practical applications across various fields. Its ability to model constraints and conflicts is instrumental in contexts such as Operational Research.

The problem characterization provides an efficient framework for tackling discrete scheduling problems, where tasks or events must be assigned to specific time slots or resources without leading to conflicts. For example, in the airline industry, the problem has been applied to optimize crew scheduling, ensuring that no crew member is double-booked or assigned overlapping duties [27, 90].

Timetabling is another classic application of graph coloring, particularly in academic or organizational settings. For instance, universities use graph coloring to create course schedules that prevent clashes between classes requiring the same room or involving the same instructor. Early foundational work in this area includes the work of Werra [83], with later refinements by Burke et al. [8] .

Another application is train platforming problems, where trains must be assigned to platforms in a way that avoids conflicts and optimizes the usage of available infrastructure. For instance, Caprara et al. [11] describes how graph-based models can streamline operations in complex railway systems, reducing delays and improving passenger satisfaction.

In compiler design, graph coloring is instrumental in register allocation, which is the assignment of variables to a limited number of CPU registers in order to minimize the movement of information in and out of those registers. In the interference graph, where nodes represent variables and edges denote simultaneous use, a color represents a register to be used by the corresponding set of variables [84, 12]. Similarly, the problem of frequency assignment in wireless communication systems can be naturally modeled as a graph coloring problem. The aim is to assign frequencies to transmitters while avoiding interference between adjacent transmitters and minimizing the usage of the electromagnetic spectrum. Studies such as those by Smith et al. [75] and Gamst [28] demonstrate how graph coloring ensures efficient spectrum utilization and maintains communication quality. Additionally, in communication networks, graph coloring is used to allocate resources like time slots or channels, optimizing the accessibility of multi-access systems and preventing interference, as shown by Woo et al. [86].

These examples endorse the relevance of investigating the Graph Coloring Problem and its applicability to real-world challenges, where efficient conflict avoidance and resource optimization are critical.

1.2 Mathematical Model

In this section, we introduce notation used throughout this text.

Let $G = (V, E)$ be a simple undirected graph, such that $E \subseteq V \times V$. We denote by $n = |V|$ the number of vertices and by $m = |E|$ the number of edges. Two vertices u and v are said to be *adjacent* if there exists an edge $\{v, u\} \in E$. We say that the *open neighborhood*, denoted $N_G(v)$, of a vertex $v \in V$ in the graph G is the set of vertices adjacent to v, i.e., $N_G(v) = \{u \in V : \{u, v\} \in E\}$. The *closed neighborhood* of v is $N_G[v] = N_G(v) \cup \{v\}$. The *density* of a graph G, denoted by $D(G)$, is the ratio $\frac{2m}{n(n-1)}$. We may omit G from those notations when it is clear from context.

A *proper coloring* is a mapping of labels, which we call *colors*, to vertices, such that no two adjacent vertices receive the same color. We refer to a proper coloring simply as a coloring. On the other hand, a *partial coloring* is the assignment, according to the same rules, of colors to only a subset of vertices. The *chromatic number* of G, denoted $\chi(G)$, is the least number of colors that can constitute a proper coloring of G. Given a partial coloring, a color h is valid for vertex v if it has not been assigned to any vertex in the open neighborhood of v. We say that the set of vertices colored by the same color is a *color class*. A k-coloring of G is a proper coloring of G with k color classes. Throughout the text, we use several well-known results concerning vertex-coloring of graphs. We refer the interested reader to **Bondy1976** for an overview.

For a graph $G = (V, E)$, a set $S \subseteq V$ is *independent* in G if and only if there are no edges in E connecting two elements of S. Notice that a color class is an independent set of G. The *independence number* $\alpha(G)$ is the maximum cardinality of any independent set of G. Let $\pi : V \to \mathbb{R}$ be a vertex weight function, we say that $\alpha(G, \pi)$ is the *maximum weight of an independent set* (MWIS) of vertices on G. We might also omit G and π from the notation when it is clear from context.

An *induced subgraph* $H = (V_H, E_H)$ of $G = (V, E)$ is a graph such that $V_H \subseteq V$ and $E_H =$

$\{\{u, v\} \in E : u, v \in V_H\}$. In other words, H is a subgraph obtained by selecting a subset of the vertices of G and including all edges between these vertices that are present in G. We denote by $G[V']$ the subgraph induced by a set $V' \subseteq V$.

The complement of a graph $G = (V, E)$ is the graph $\overline{G} = (V, \overline{E})$, such that

$$\overline{E} = \{\{u, v\} : \{u, v\} \notin E,\ u, v \in V \text{ and } v \neq u\}.$$

That is to say, two vertices are adjacent in $\overline{G}$ if and only if they are not adjacent in G.

A *clique* of graph G is a subset $Q \subseteq V$ of vertices such that all vertices in Q are pairwise adjacent in G. Note that an independent set in G is a clique in $\overline{G}$.

1.3 Other methods

Before we dive into the techniques we use, we want to give a brief overview of different approaches employed to solve the problem.

Heuristics play a significant role in tackling the Graph Coloring Problem. Early work by Hertz and De Werra [34] introduced the use of tabu search and other local search methods, which were later extended [5, 16, 34, 35, 66]. For a comprehensive overview we refer the reader to the work of Galinier and Hertz [25].

Population-based hybrid algorithms, as proposed by Galinier and Hao [24], generally offer the best performance for the coloring problem [78]. Some other specific heuristic techniques include methods for extracting independent sets [32] or the use of simulated or quantum annealing [80, 73].

When it comes to fast heuristics, those that prioritize quickly finding a "good-enough" solution over extensive optimization, such as DSATUR [7] and RLF [46], stand out. These greedy algorithms are often employed as initialization procedures for more complex methods due to their simplicity and efficiency.

The use of SAT-based methods for the Graph Coloring Problem has also been explored in the literature. Van Gelder [81] introduces formulations based on propositional logic, while Bouhmala and Granmo [6] applies learning automata and random walks to the problem. More recently, Heule et al. [36] presents a hybrid approach combining SAT algorithms for maximum clique solving which exchanges information with an exact graph coloring algorithm. Despite these advancements, the applicability of SAT-based methods remains limited in certain scenarios [15].

Other approaches to the problem, such as Dynamic Programming [9, 17], have gathered theoretical interest but lack practical application. Enumeration algorithms, pioneered by Zykov [91] and further developed by Brélaz [7], have been adapted and improved upon by researchers like San Segundo [68], Sewel [71], and Furini et al. [23].

For a broader perspective on these and related methods, see the surveys by Lima and Carmo [49], Lewis [48], and Malaguti and Toth [54].

1.4 Integer Linear Programming

Linear programming is an optimization technique used to model problems with linear equations and inequalities. In these models, a linear objective function is defined and we seek to minimize (or maximize) its value, subject to a set of linear constraints (including linear equalities and inequalities) [13].

Given a matrix $A \in \mathbb{R}^{m \times n}$ dimensions, and vector b and c of dimensions m and n respectively, we need to find a vector $x \in \mathbb{R}^n$, such that the following criteria are met:

$$\begin{aligned} (P) \quad & \text{minimize} && c^\top x \\ & \text{subject to} && Ax \geq b \\ & && x \in \mathbb{R}^n_+. \end{aligned}$$

This way, $c^\top x$ is our *objective function*, while $\{A_j x \geq b_j : 1 \leq j \leq m\}$ is a set of *constraints* that need to be abide to. A *solution* is an attribution of values to x, and we denote it by $\overline{x}$. A solution $\overline{x}$ satisfies constraint j if its corresponding inequality holds. Also, we say $\overline{x}$ is *feasible* for a linear program if it satisfies all constraints and is *optimal* if the value of $c^\top \overline{x}$ is minimum among all feasible solutions.

Several algorithms can be employed to find optimal solutions. The *simplex algorithm* [14], despite having exponential worst-case time complexity, solves linear programs efficiently in practice and has polynomial time complexity on average. Algorithms such as the *interior-point method* [**Diking67**, 43, 63] and the *ellipsoid method* [30] can also solve linear programs and have polynomial time complexity in the worst-case. However, in practice, the ellipsoid method performs worse on average, while the simplex and interior-point methods are generally competitive.

For a *primal* minimization problem as stated previously in (P), its *dual* (D) is defined as:

$$\begin{aligned} (D) \quad & \text{maximize} && b^\top y \\ & \text{subject to} && A^\top y \leq c \\ & && y \in \mathbb{R}^m_+. \end{aligned}$$

An optimal solution $\overline{x}$ for (P) and a solution $\overline{\pi}$ for (D) are said to be corresponding solutions if $\overline{\pi}$ is feasible for (D) and the solutions satisfy the Complementary Slackness Theorem.

Theorem 1.1 (Complementary Slackness [85])**.** Let (P) be a linear program, and (D) its dual. Suppose $\overline{x}$ and $\overline{\pi}$ are feasible solutions for (P) and (D), respectively. Then, $\overline{x}$ and $\overline{\pi}$ are optimal if, and only if, for every j such that $1 \leq j \leq m$:

- if $\overline{\pi}_j > 0$, then $A_j \overline{x} = b_j$;
- if $A_j \overline{x} < b_j$, then $\overline{\pi}_j = 0$;

The dual problem provides a lower bound for the primal objective value, as stated in the Weak Duality Theorem.

Theorem 1.2 (Weak Duality [85])**.** Let (P) be a linear program and (D) its dual. If $\overline{x}$ and $\overline{\pi}$ are feasible solutions to (P) and (D), respectively, then $c^\top \overline{\pi} \leq b^\top \overline{x}$.

Furthermore, under certain conditions, the primal and dual optimal values coincide, as established by the Strong Duality Theorem.

Theorem 1.3 (Strong Duality [85])**.** Let (P) be a linear program, and (D) its dual. Then:

1. If (P) is infeasible, then (D) is either infeasible or unbounded.
2. If (D) is infeasible, then (P) is either infeasible or unbounded.
3. If both (P) and (D) are feasible, then their optimal value are equal.

For some problems, such as graph coloring, solutions with non-integer values are not suitable because we cannot assign, for example, "half a color" to a vertex. To address this, variables can be restricted to integer values, resulting in an *Integer Linear Program* (ILP). These restrictions are referred to as *integrality constraints*. If only a subset of the variables requires integer values, the program is called a *Mixed Integer Linear Program* (MILP).

What might seem like a small change introduces significant computational challenges [67]. The previously mentioned algorithms are not sufficient to find optimal feasible solutions for integer linear programs. Instead, techniques such as *branch-and-bound* are used.

Branch-and-bound (B&B) is a general optimization framework used to systematically explore and prune the solution space. The algorithm builds a tree where each node represents a subproblem, derived by "branching" on a decision (e.g., choosing a specific independent set to include or exclude) in the context of the graph coloring problem. At each node, bounds are computed for the value of an optimal solution within the subproblem. If a subproblem's bound indicates it cannot outperform the best-known solution, the node is "pruned," avoiding unnecessary computation. This method depends heavily on efficiently computing bounds and selecting branches, as it can otherwise degrade into an exhaustive search. For minimization ILPs, any feasible solution can serve as an upper bound, while solutions to the *linear relaxation* (the corresponding linear program without integrality constraints) provides a lower bound.

The *reduced cost* of a variable provides valuable insight into the optimization problem. For a minimization problem, the reduced cost of a variable x_i is defined as:

$$c_i - \sum_{j=1}^{m} y_j A_{ij},$$

where c_i is the coefficient of x_i in the objective function, y_j are the dual variables associated with the constraints, and A_{ij} represents the coefficients of x_j in the constraints.

The reduced cost indicates how much the objective value would decrease if the value of x_j were increased by one unit, assuming all other variables remain constant. If the reduced cost of a variable with value zero is negative, incrementing that variable's value can improve the objective value.

1.5 Coloring Formulation

As proposed by Mehrotra and Trick [57], a way of seeing the graph coloring problem is to imagine it as a *Set Cover problem* (SC), where the available sets are all the independent sets for the graph.

As such, let $\mathcal{S}$ be the family of all independent sets of the graph $G = (V, E)$ that we want to color. We use binary variables x_S to indicate if set $S \in \mathcal{S}$ is used or not. This way, we formulate the problem as:

$$\begin{aligned} \text{(SC)} \quad \text{minimize} \quad & \sum_{S \in \mathcal{S}} x_S \\ \text{subject to} \quad & \sum_{S : v \in S} x_S \geq 1 \qquad \forall v \in V \\ & x_S \in \{0, 1\} \quad \forall S \in \mathcal{S}. \end{aligned}$$

The first set of constraints, called cover constraints, guarantees all vertices are contained in at least one chosen set. The objective function aims to minimize the number of chosen sets in the same way we want to use the least number of colors.

It is important to note the cover constraints are in itself a relaxation for the coloring problem since, this way, we could "use two colors" in a vertex. This is easily solvable by the fact that any subset of an independent set is also independent.

On the other side, the number of independent sets in a graph can be exponential compared to the number of vertices, making it infeasible to enumerate them all. To avoid doing so, we can start from a subset of variables $\mathcal{S}' \subseteq \mathcal{S}$, solve the linear relaxation, and then find new variables to add to the model such they improve the current solution. The intuition comes from the same process the simplex algorithm does in order to find columns to pivot, computing the *reduced cost* and adding a column with minimum value of it to the basis.

A more intuitive way of looking at this circumstance is through the corresponding dual problem[1]:

$$\begin{aligned} \text{(dual-SC)} \quad \text{maximize} \quad & \sum_{v \in V} \pi_v \\ \text{subject to} \quad & \sum_{v \in S} \pi_v \leq 1 \quad \forall S \in \mathcal{S}' \\ & \pi_v \geq 0 \quad \forall v \in V \\ & \pi \in \mathbb{R}^n. \end{aligned}$$

Since we did not enumerate all primal variables, the corresponding dual is still missing some constraints. Because of that, those missing constraints could be violated making the corresponding dual solution infeasible.

In other words, to find which constraints are needed, we need to find $S \in \mathcal{S}$ such that $\sum_{v \in S} \overline{\pi}_v > 1$. That is, an independent set with weight greater than 1, where the vertices'

[1]The corresponding dual problem referenced here comes from the the linear relaxation of the Set Cover (SC) formulation, since an integer program does not have a corresponding dual directly.

weights are the corresponding dual solution values. To prove that such a set does not exist, we need to show that the weighted independence number $\alpha(G, \overline{\pi})$ is at most 1. Formally, we need to find

$$\alpha(G, \overline{\pi}) = \max\left\{\sum_{v \in S} \overline{\pi}_v : S \in \mathcal{S}\right\} \tag{1.1}$$

which is an instance of the *Maximum Weighted Independent Set* problem.

Because we need to work on a reduced set of variables, using a branch-and-bound approach would not suffice. *Branch-and-price* (B&P), on the other hand, extends the B&B framework by incorporating *column generation*, a strategy to handle problems with an exponential number of variables [19]. In this approach, the problem begins with a restricted problem using a manageable subset of variables. The algorithm alternates between solving this relaxation and identifying new variables to add by solving a pricing problem (e.g., finding an independent set with reduced cost less than zero or, equivalently, dual weight greater than one).

The current state-of-the-art branch-and-price approach for graph coloring by Held et al. [33] employs various methods to generate new columns. Before solving the problem exactly, the authors first attempt to identify violated constraints using three greedy heuristics. If these heuristics cannot find an independent set with a weight greater than 1, the method employs either a clique enumeration approach or a branch-and-bound algorithm. While both are exact algorithms for the MWIS problem, the former is applied to dense graphs (with density $D > 0.8$), and the latter is used for sparser cases.

1.6 Other formulations

Besides the model that we show in the previous section, plenty of formulations for the problem are present in the literature. We go over the most prominent ones.

Allocation Model Given an upper bound (ub) to the chromatic number (such as the maximum degree of a vertex or a heuristic solution), we define binary variables x_{vh} if vertex v is allocated to color h and y_h if color h is used at least once. This way, we can produce the following formulation:

$$\begin{aligned}
\text{minimize} \quad & \sum_{h=1}^{n} y_h \\
\text{subject to} \quad & \sum_{h=1}^{n} x_{vh} = 1 && \forall v \in V \\
& x_{vh} + x_{uh} \leq y_h && \{u, v\} \in E, h = 1, \ldots, ub \\
& x_{vh} \in \{0, 1\} && \forall v \in V \\
& y_h \in \{0, 1\} && h = 1, \ldots, ub.
\end{aligned}$$

Although a simple formulation, this model sees little interest nowadays. This comes from two major drawbacks:

1. It has many symmetries, since colors are indistinguishable from one another. A solution

that uses k colors has $k!$ permutations of what is effectively the same solution.

2. The linear relaxation is extremely weak, which means that solving it will land us far from the optimal integer solution value.

Some have tried to improve its performance, such as the works from Méndez-Díaz and Zabala [59] and Méndez-Díaz and Zabala [58], which adds valid inequalities through a *branch-and-cut* algorithm. In a branch-and-cut process, we follow the paradigm of branch-and-bound but, at each linear relaxation solved, we try to strengthen it by adding valid inequalities. To strengthen a linear relaxation is to cut out fractional solutions, while a valid inequality is one that does not remove any integer solution from the set of feasible solutions of the relaxation.

Representatives Model Campêlo et al. [10] proposed a formulation where each color is represented by a vertex. To achieve this, they introduced binary variables x_{uv} for each pair of non-adjacent vertices $u, v \in V$, where x_{uv} indicates whether vertex v represents the color of vertex u. Additionally, x_{vv} is used to specify if v is its own representative. Let $\bar{N}(v)$ denote the set of non-adjacent vertices to v. The formulation is written as:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{v \in V} x_{vv} \\
\text{subject to} \quad & \sum_{v \in \bar{N}(u) \cup \{u\}} x_{uv} = 1 && \forall u \in V, \\
& x_{uv} + x_{wv} \leq x_{vv} && v \in V, \forall \{u, w\} \in G[\bar{N}(v)] \\
& x_{uv} \in \{0, 1\} && \forall u \in V, v \in \bar{N}(u).
\end{aligned}
$$

The first set of constraints ensures that every vertex has exactly one representative among all vertices that are not neighbors to it or itself. The second set guarantees that two adjacent vertices do not share the same representative.

This formulation, like the previous, suffers from multiple symmetries, as the choice of which vertex acts as the representative does not affect the composition of the color classes. To address this, the authors proposed additional valid constraints to strengthen the model, such as requiring the representative of a color class to be the vertex with the lowest index. Computational experiments conducted by Jabrayilov and Mutzel [39] demonstrated that this formulation is also competitive compared to the others, specially in dense instances.

Hybrid Partial Ordering Proposed by Jabrayilov and Mutzel [39] and later extended by the same authors [40], this formulation uses a mix of allocation model and partial ordering. To do so, they define a partial ordering of the union of vertex set V and ordered set of colors $(1, \ldots, ub)$, where ub is an upper bound for the chromatic number. We say vertex v is colored by color h if $v \succ h$ and, in the case of $h > 1$, $v \not\succ h - 1$.

The mathematical formulation is quite convoluted and, since it is not be the focus of this work, we limit ourselves to the example in Figure 1.1. On the bottom, we see an ordering of vertices and colors. The corresponding graph and coloring can be found on top of it. For example, vertex g is colored by color 2 since $g \succ 2$ and $1 \succ g$. Note that the ordering of the color is kept, so $1 \succ 2 \succ 3 \ldots ub - 1 \succ ub$.

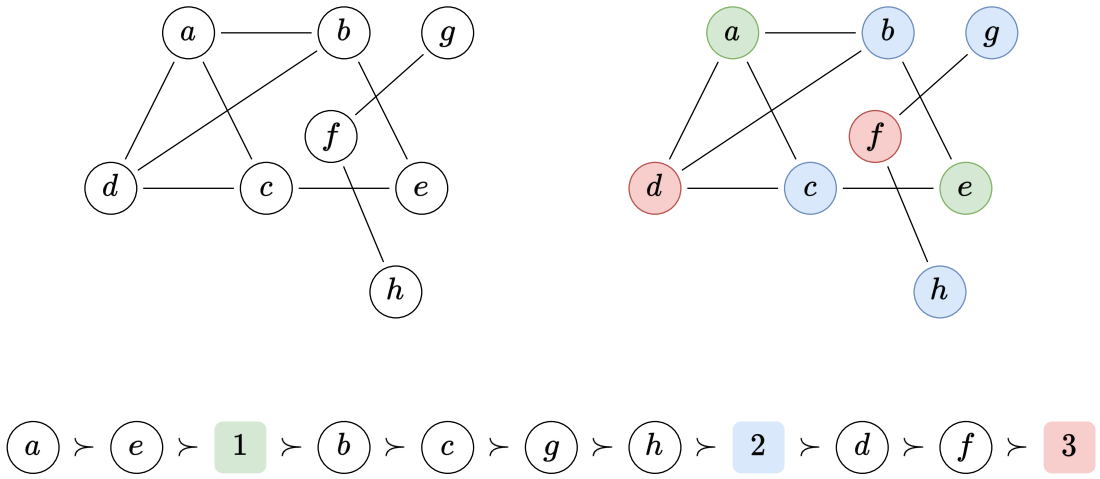

Figure 1.1: Example of coloring using partial ordering. On the top left, the graph we want to color, and on the right, an example of such coloring. The corresponding solution to the Hybrid Partial Ordering formulation is represented bellow the graphs.

According to the authors and their computational experiments, this formulation is competitive with the assignment and representatives formulations, specially in sparse graphs. They were also able to close the instance abb313GPIA.

Decision Diagram. Recently, Hoeve [37] proposed the use of decision diagrams with network flow to solve the coloring problem. The authors construct a decision diagram of independent sets, where each decision arc corresponds to inserting or not a vertex to the set and each node represents the vertices that can still be chosen for the independent set. In this approach, each color class is a path along the decision diagram of independent sets and the chromatic number is the least amount of paths that determine a partition of the graph.

Formally, for a problem P defined by an ordered set of variables $X = \{x_1, x_2, \ldots, x_n\}$, a decision diagram is constructed as a simple acyclic directed graph with $n + 1$ levels. The first level contains a single vertex, r, called the root, while the last level contains a single vertex, t. Level i consists of a set of nodes associated with the variable x_i, each having arcs to nodes in level $i + 1$. These arcs are labeled either 0 or 1, corresponding to the value of the associated variable.

In Figure 1.2, we show an example. The numbers inside the nodes represent the set of available vertices (i.e., the state), while the dashed arcs correspond to 0-arcs (indicating that the vertex is not included), and the solid arcs correspond to 1-arcs (indicating that the vertex is included in the independent set). On the first level, we can still add any vertex we wish, since none have been picked yet. The corresponding variable, x_a, indicates whether we chose to insert vertex a in our independent set. If we do, represented by the 1-arc, we will no longer be able to chose vertex c, giving us the resulting state $\{b, d\}$. On the other hand, if we chose to not insert vertex a, we are left with all the others to chose from on the next levels.

Hoeve [37] applied this technique to the independent set formulation, where each variable indicates whether or not a vertex is included in the set. If a decision diagram would be constructed to represent the coloring problem exactly, it could be solved using a flow formulation.

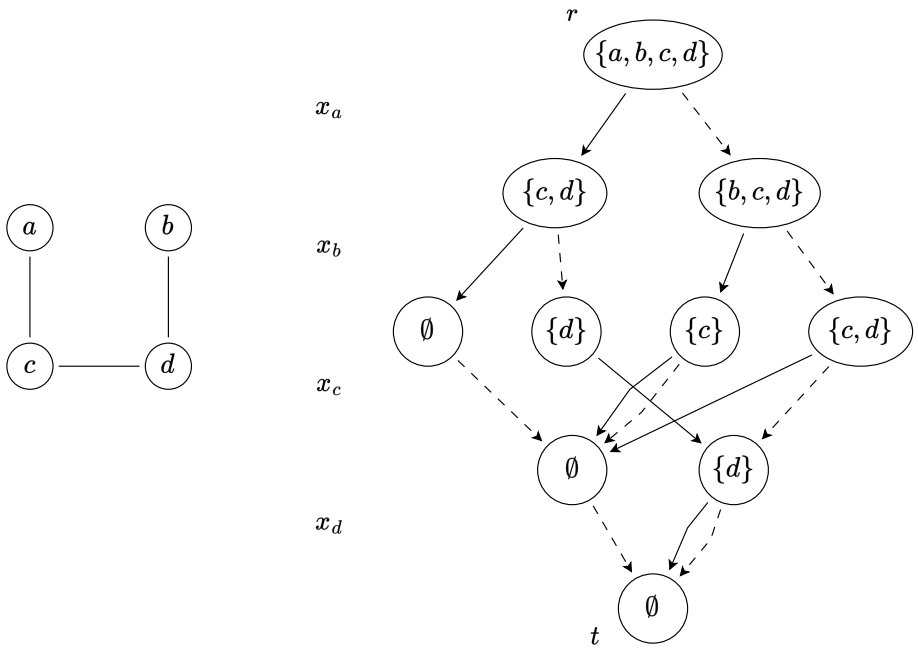

Figure 1.2: Example from Hoeve [37]. On the left, is the graph to be colored; on the right, is the corresponding decision diagram.

In this formulation, each r, t-path would correspond to a color class, and, as demonstrated by the authors, the objective function would yield the chromatic number. Unfortunately, such diagrams may contain an exponential number of nodes, requiring more sophisticated techniques specific to decision diagrams. Moreover, its minimum size for solving the graph coloring problem depends heavily on the chosen ordering, making it a critical parameter for the algorithm.

The authors report competitive results compared to other state-of-the-art methods by employing specific decision diagram strategies to address the challenges mentioned above. In summary, they were able to provide more lower bounds than other works, but the branch-and-price approach of Held et al. [33] still managed to close more instances.

Chapter 2

Separate and Reduce

A key strategy for significantly reducing computational effort is to shrink the search space by intelligently dividing the problem into smaller subproblems and refining it using previously obtained bounds. These methods can help us make hard instances more manageable, either by decomposing them into smaller subproblems or by simplifying their structures without losing essential properties. Separating and reducing instances is crucial for mitigating the exponential growth in computational time inherent to NP-hard problems like graph coloring.

One idea that will become underlying in the next section is that, by (re)introducing some vertices into the graph, the chromatic number cannot be reduced. This means that, a lower bound computed on an induced subgraph is still a valid lower bound on the original graph. So, by this logic, by removing vertices, we can compute valid lower bounds for the chromatic number in a smaller graph.

2.1 Separate

A simple observation is that the chromatic number of a graph G is equal to the maximum chromatic number of each of its connected components, i.e.

$$\chi(G) = \max\{\chi(H) : H \text{ is a connected component of } G\}.$$

This means we can separate the graph into its connected components, solve each one and merge the solutions. As an example, if $\{h_1, h_2, \ldots, h_k\}$ and $\{h'_1, h'_2, \ldots, h'_l\}$ are a proper coloring of the two connected components H and H' of G, then

$$\left\{h_1 \cup h'_1, h_2 \cup h'_2, \ldots, h_{\max\{k,l\}} \cup h'_{\max\{k,l\}}\right\}$$

is a proper coloring of G.

A coloring constructed by splitting a disconnected graph into its components is valid because no edges connect vertices in different components, so coloring each separately does not violate the original graph's constraints. Since each component must be colored with at least its chromatic number, the overall chromatic number of the graph is at least the maximum chromatic number among its components. Moreover, as the chromatic number of any connected

component serves as a valid lower bound for $\chi(G)$, if a component H has a chromatic number of k and another component H' already admits a k-coloring, then solving H' to optimality becomes unnecessary even if its chromatic number is lower, thereby simplifying the problem.

This idea can be further explored by altering the order in which each connected component is solved, with the goal of enhancing the effectiveness of the reductions described below (see Section 2.2). By prioritizing components with the most vertices, we aim to obtain lower bounds that can simplify the resolution of smaller connected components. On the other hand, if we start from the smallest connected component, we could obtain some lower bound, albeit less than we would otherwise, but speed up the solution of bigger connected components, which (usually) are the hardest. We put those ideas to the test later in our experiments

Another useful lemma comes from the connected components of the graph's complement.

Lemma 2.1. Let G be a graph, and $\overline{V}_1, ..., \overline{V}_k$ be the connected components of $\overline{G}$. Then, $\chi(G) = \sum_{i=1}^{k} \chi(G[\overline{V}_i])$.

This comes from the fact that, for any two connected components $\overline{V}_i$ and $\overline{V}_j$ from $\overline{G}$, all vertices $v \in \overline{V}_i$ are adjacent to all vertices $u \in \overline{V}_j$ in the original graph G. Therefore, no color used in one connected component of $\overline{G}$ can be used in another connected component.

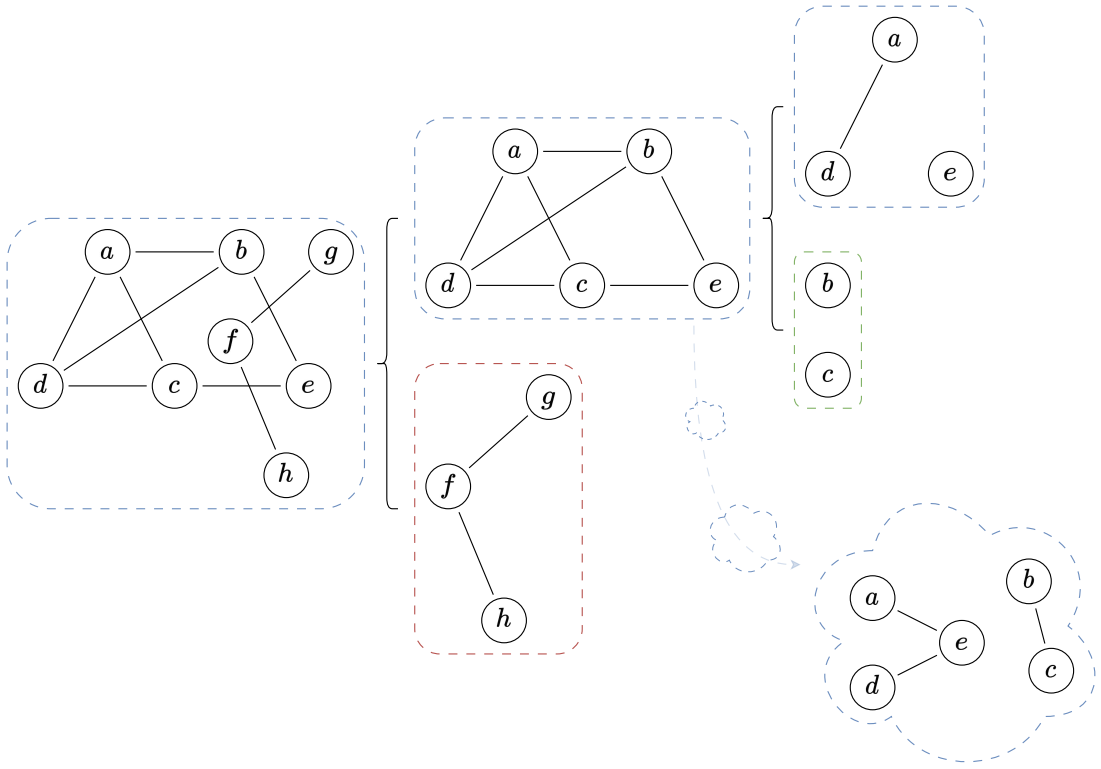

Figure 2.1: Illustration of how we can separate a graph in order to facilitate finding an optimal coloring. The original graph, on the left, is split two times. First, since it is not connected, and a second time, since its complement (represented in the cloud shape figure) is disconnected.

Lastly, we note that those reductions can be applied multiple times: by separating the graph because it is disconnected, we can still check for the condition in Lemma 2.1, and repeat

this process until no more separations can be done. An example of such a recursive process can be found in Figure 2.1. When converging back to the original graph, we need to apply the correct rule to unite solutions from different parts based on the reason for separating.

2.2 Reduce

Another well-known fact about the chromatic number of a graph stated in Lemma 2.2 helps us further simplify the problem by finding subgraphs.

Lemma 2.2. Let G be a graph, and G' a subgraph of G. Then $\chi(G) \geq \chi(G')$.

Initially proposed by Seidman [70], the concept of k-core can be applied to reduce the number of vertices considered when solving the coloring problem.

Definition 2.3. Let G' be a subgraph resulting of iteratively removing vertices with a degree less than k from graph G. Each connected component of G' is called a k-core of G.

The concept of k-core relates to the *degeneracy* $\delta^*(G)$ of a graph, which is the greatest value of k such that G contains a non-empty k-core . We refer the reader to Malliaros et al. [55] for a survey of algorithms that apply this idea.

It is important to note that, by this process, we might end up with multiple components (k-cores), but since we would apply the separation rules explained earlier, we will consider it as a connected subgraph.

Given a lower bound k to the chromatic number of G, we can find the k-core of G in polynomial time. A useful property of such k-core is given by Theorem 2.4.

Theorem 2.4. Let G be a graph and k a lower bound on $\chi(G)$. A coloring to its k-core can be extended to a valid coloring for the original graph with the same number of colors in polynomial time.

It is possible to see this fact since, by iterating in the reverse order in which we removed the vertices, we can guarantee that there will be an available color to assign to each vertex. Because k is a lower bound on $\chi(G)$, there are at least k colors, and if vertex v was removed, it had less than k neighbors, so there must already exist a color valid for v.

The extent to which this reduction simplifies the graph is directly proportional to the closeness of the lower bound for the chromatic number. Because of that, now is a good time to revisit the known lower bounds for the chromatic number of a graph G. The first one comes from the clique number ($\omega(G)$), the maximum size of a clique in the graph, since it must be colored by $\omega(G)$ colors. By finding the independence number ($\alpha(G)$), the maximum size of an independent set in the graph, we can bound the chromatic number of a graph to be, at least, $n/\alpha(G)$. Additionally, there is the Lovász number of the complementary graph $\vartheta(\overline{G})$ [50], which is also a lower bound to $\chi(G)$. It can be approximated in polynomial time, using semi-definite programming and the ellipsoid method, but this would be outside of our scope. Our last lower bound is the fractional chromatic number $\chi_f(G)$, which comes from linear relaxation of the SC formulation to solve the coloring problem. Therefore, $\lceil \chi_f(G) \rceil$ is a valid lower bound and we will expand on that in Chapter 3.

Regarding the strength of these bounds, it is known that $\frac{n}{\alpha(G)} \leq \chi_f(G)$ [69], and $\omega(G) \leq \chi_f(G) \leq \chi(G)$. The fractional chromatic number approximates the chromatic number within a logarithmic factor [51]

$$\frac{\chi(G)}{1 + \ln \alpha(G)} \leq \chi_f(G) \leq \chi(G).$$

Moreover, by The Lovász "sandwich theorem" [44], we also have that $\omega(G) \leq \vartheta(\overline{G}) \leq \chi(G)$.

Unfortunately, α, ω, and χ_f are NP-hard to compute. Therefore, we will use heuristics to approximate the first two, while the third will be computed exactly. On the bright side, by eliminating vertices, the graph, or its complement, might become disconnected, so we may further separate using our separations rules, as explained in Section 2.1.

Lastly, there are two well-known reductions to our problem. First, when a vertex is *universal*, i.e., adjacent to all other vertices, it can be removed and later assigned to a new color. Second, for a pair of non-adjacent vertices u, v, if $N(u) \subseteq N(v)$, we say vertex v dominates over u. When that happens, we can remove vertex u, and later color it with the same color v has been assigned to.

Chapter 3

Coloring

As we glossed over at Section 1.5, we base our work on the formulation proposed by Mehrotra and Trick [57]. By considering each color class as an independent set of vertices and recognizing a valid coloring as a partition of the vertex set, we arrive at the *set partitioning formulation*. Let $\mathcal{S}$ be the family of all independent sets of the graph $G = (V, E)$, we can model the coloring problem as a set partition problem, where we aim to partition the set of vertices V into color classes $S \in \mathcal{S}$. An integer linear programming formulation for such a problem can be written as such:

$$\text{(SP)} \quad \text{minimize} \quad \sum_{S \in \mathcal{S}} x_S \tag{3.1}$$

$$\text{subject to} \quad \sum_{S \in \mathcal{S} : v \in S} x_S = 1 \qquad \forall v \in V \tag{3.2}$$

$$x_S \in \{0, 1\} \quad \forall S \in \mathcal{S} \tag{3.3}$$

where x_S is a binary decision variable that indicates whether the independent set S is used. Our objective function, Equation (3.1), aims to minimize the number of color classes used, given that all vertices are in exactly one set, as required by Equation (3.2).

Such formulation can be relaxed into a set covering problem by relaxing Equation (3.2) into Equation (3.4), while maintaining the rest of the formulation.

$$\sum_{S \in \mathcal{S} : v \in S} x_S \geq 1 \quad \forall v \in V. \tag{3.4}$$

A solution of such can be transformed into a valid coloring by simply assigning one color to each S and choosing one of the colors used to cover vertex v. This process can be accomplished in linear time. Overall, this relaxation simplifies the model, making it more tractable to solve, at the cost of a computationally efficient post-processing step.

At this point, we still need to address two issues: first, we may have an exponential number of variables, which means it is prohibitive to enumerate them all; secondly, solving an ILP model is NP-hard [29]. We address both challenges in this section, beginning with the second one.

To address the latter, we implement a *branch-and-bound* algorithm, leveraging the *linear*

relaxation of the model as a lower bound, alongside heuristics to generate feasible solutions that serve as upper bounds. By relaxing Equation (3.3), we can arrive at the following model:

$$\begin{aligned} \text{(LM)} \quad \text{minimize} \quad & \sum_{S \in \mathcal{S}} x_S \\ \text{subject to} \quad & \sum_{S \in \mathcal{S}: v \in S} x_S \geq 1 \qquad \forall v \in V \\ & x_S \in \mathbb{R}_+ \quad \forall S \in \mathcal{S}. \end{aligned}$$

Note that we do not need to restrict our real variables to be less than or equal to 1, since we are minimizing the number of sets used, and there is no reason to use the same set more than once. With a fractional optimal solution x with value lb for LM and an upper bound ub, we can define the gap as $\text{gap} = ub - \lceil lb \rceil$. If the gap between the current lower and upper bounds is not zero, we need to split the search space using what we call branching. We will further explore this topic in Section 3.4.

By solving LM, we obtain the *fractional chromatic number*, the strongest known lower bound for the chromatic number of a graph [69]. Together with upper bounds from primal heuristics such as DSATUR, Relax-and-Fix, and Rounding, further explained in Section 3.3, we can narrow our branch-and-bound search.

We are left with the problem of the exponential number of variables. For that, we employ the idea of using a restricted set of variables $\mathcal{S}'$ in the Model LM, arriving at a model we call *restricted* (RLM). After solving it using general LP solvers, we need to check if there is any set yet to be enumerated that could improve the solution.

To achieve this, we compute the *reduced cost* of a variable. If the reduced cost is negative, it implies that including the variable in the restricted set of variables could lead to an improved solution when re-optimized.

Another, more intuitive way of looking at this, is to observe the corresponding dual problem of Model RLM:

$$\begin{aligned} \text{(dual-RLM)} \quad \text{maximize} \quad & \sum_{v \in V} \pi_v \\ \text{subject to} \quad & \sum_{v \in S} \pi_v \leq 1 \qquad \forall S \in \mathcal{S}' \\ & \pi_v \in \mathbb{R}_+ \quad \forall v \in V. \end{aligned}$$

Here, we need to maximize the summing of all vertex weights, subjected to the sum of weights in each independent set is less than or equal to 1. Similarly to what we have done for the primal problem, we will be working with a restricted subset of sets, which, in this case, will correspond to constraints.

Given a pair of optimal solutions $(\overline{x}, \overline{\pi})$ for RLM and dual-RLM, by the Strong Duality Theorem 1.3, it is also a pair of optimal solutions for the constricted models if, and only if, $\overline{\pi}$ is feasible for the dual of LM, as $\overline{x}$ is feasible for LM. This corresponds to confirming that no

constraint is violated, that is,

$$1 - \sum_{v \in S} \overline{\pi}_v \geq 0 \quad \forall S \in \mathcal{S}. \tag{3.5}$$

If that is not the case, we need to add some of those violated constraints to our restricted model and re-optimize. Note that the left-hand side of Equation (3.5) is exactly the reduced cost of a variable associated with $S \in \mathcal{S}$.

The process of determining whether a dual constraint is violated is known as the *pricing problem*. Solving the restricted linear model by starting with a restricted set of columns, $\mathcal{S}'$, and dynamically generating additional columns through the pricing problem is referred to as the *column generation* technique. Consequently, *branch-and-price* algorithm is the combination of a a branch-and-bound search tree with a linear relaxation solved using column generation.

The traditional branching scheme, referred to as *variable branching* by Malaguti and Toth [54], introduces additional constraints, such as prohibiting the use of specific variables. These constraints modify the pricing problem, preventing certain variables from being included in the model, even if their reduced costs would otherwise justify their selection. In our case, this entails solving the maximum weighted independent set problem, which is itself NP-hard [4], with the added restriction that the solution must exclude sets from such list of "prohibited" subsets. This modified pricing problem is referred to as the *constrained pricing* problem by Morrison et al. [62], and we will avoid it by using the scheme proposed by Foster and Ryan [21] (see Section 3.4).

There is another challenge lurking around our computations: the imprecision of floating point arithmetic. Operating with variable precision may introduce cumulative error, potentially causing premature termination of the process. Because of that, commercial solvers, such as Gurobi [31], use a threshold of how much a constraint can be violated in order to speed up computations. If no dual constraints are violated by more than the specified threshold, the solver considers the solution feasible, which may lead to an inaccurate lower bound.

Therefore, there are two moments we need to take such imprecision into account: when retrieving the dual solution to solve Equation (3.5), and, after solving the Model RLM, when computing a numerically safe lower bound. To address that, we will have to compute a fixed-point dual solution $\overline{\pi}_{\text{fixed}}$ and, based on such solution, a lower bound we know it is valid, lb_{safe}.

In this chapter, we will dive into how we employ such *branch-and-price*, but we will leave how we prove that our solution satisfies Equation (3.5) for the Chapter 4. A high-level overview of our branch-and-price algorithm can be found in Algorithm 1.

Algorithm 1: Branch-and-Price algorithm.

Input : A graph $G = (V, E)$, and a global lower bound *global_lb* (based on any pre-processing, see Chapter 2).
Output : An optimal valid coloring.

Add $(G, global\_lb)$ as a node to be explored in the branch tree
Let $\mathcal{S}' \leftarrow \emptyset$ be the set of all enumerated color classes. // All color classes generated by the various methods during our algorithm will be added to this set.
$ub \leftarrow$ an initial upper bound by the DSATUR heuristic
while *there are unexplored nodes in the search tree with lower bound* $<$ *ub* **do**
    Chose an unexplored node (G, lb) and mark it as explored
    repeat
        Solve the Model RLM using G and $\mathcal{S}'$ via Gurobi [31]
        Try to improve the upper bound by using the Rounding heuristic // Section 3.3
        Compute the fixed-point weights $\overline{\pi}_{\text{fixed}}$ // Section 3.1
        Generate new columns with the pricing algorithm $MWIS(G, \overline{\pi}_{\text{fixed}})$ // Chapter 4
    until *no new sets are generated* ***or*** *we can stop earlier* // Section 3.2
    Reapply reduction by k-core, if possible // Section 2.2
    Compute a upper bound using heuristics and update *ub* if possible // Section 3.3
    $lb_{\text{safe}} \leftarrow$ numerically safe lower bound // Section 3.1
    if $\lceil lb_{\text{safe}} \rceil < ub$ **then**
        Branch with $\lceil lb_{\text{safe}} \rceil$ as lower bound // Section 3.4
        Add the created nodes to the list of unexplored nodes
return *ub*

3.1 Fixed Point Arithmetic

We start by addressing the problem of numerical errors. As we previously explained, commercial linear program solvers use floating-point arithmetic, which can build up error, and lead us to a invalid lower bound.

For illustration, let's exaggerate the numbers for simplicity: suppose we have an upper bound of 5, and we are solving a node with a current value of 4.001. If the pricing algorithm fails to find an independent set of weight greater than 1, we will wrongly conclude that we have a feasible dual solution and, therefore, a lower bound, and, in this case, we close the gap by rounding up. On the other hand, because of numerical errors, we could have missed a set with weight 1.002, which in turn, could make us terminate the relaxation at 3.999 and miss out on a potential 4-coloring.

Although we cannot find an optimal dual solution because of those problems using floating-point arithmetic, any feasible dual solution is a lower bound, by the Weak Duality Theorem 1.2.

Some works have shown how to find such feasible dual solution, such as the work from Farley [18], which has been used by Held et al. [33] to find a numerically safe lower bound for the coloring problem, and extended by Baldacci et al. [3] to dual variables corresponding to valid inequalities added for the primal model. The idea is to convert the floating-point dual solution $\overline{\pi}_{\text{float}}$ into a fixed-point dual solution $\overline{\pi}_{\text{fixed}}$ by rounding and find an upper bound $\overline{\alpha}(G, \overline{\pi}_{\text{fixed}})$ to the reduced cost of all variables. We can make the dual solution $\overline{\pi}_{\text{float}}$ feasible based on that bound, therefore obtaining a numerically safe lower bound lb_{safe} to our problem.

For the following generic primal problem, we can obtain the corresponding dual:

$$\begin{aligned} &\text{(primal)} && \text{minimize} && c^\top x && \qquad && \text{(dual)} && \text{maximize} && b^\top \pi \\ &&& \text{subject to} && Ax \geq b && && && \text{subject to} && A^\top \pi \leq c \\ &&&&& x \in \mathbb{R}^n_+, && && &&&& \pi \in \mathbb{R}^m_+. \end{aligned}$$

Given a dual solution $\overline{\pi}$, not necessarily feasible, the process proposed by Farley [18] is to find by how much the constraints of the dual are violated and scale the corresponding solution to make it feasible. In other words, as $\overline{\pi}, c \geq 0$, find

$$\lambda = \max_{i \in \{1,2,\ldots,n\}} \frac{A_i^\top \overline{\pi}}{c_i}.$$

It is possible to see that, by dividing each value in $\overline{\pi}$ by λ, the solution we arrive is feasible, since no constraint could be violated. Thus, $b^\top \overline{\pi}/\lambda$ is a valid lower bound for the value of an optimal primal solution $\overline{x}^*$.

Since we do not have all the dual constraints laid out for us, if we manage to find an upper bound for it, we can apply the same scaling. By working with fixed precision, we can guarantee that no constraint is violated by a value greater than the precision used, making it the upper bound we need to apply the aforementioned method. It remains to define how much precision we will need to use.

It is possible to simulate fixed-point arithmetic by scaling our float-point values to integer values. As done by Held et al. [33], we can obtain integer values $\overline{\pi}_{\text{fixed}}$ for the weight of vertices from the floating-point weight $\overline{\pi}_{\text{float}}$ as $\overline{\pi}_{\text{fixed}}(v) := \lfloor K\overline{\pi}_{\text{float}}(v) \rfloor$ where K is a large, carefully chosen, integer. As such, we have

$$\overline{\pi}_{\text{float}}(v) - \frac{1}{K} \leq \overline{\pi}_{\text{fixed}}(v) \leq \overline{\pi}_{\text{float}}(v),$$

which gives us an absolute $\frac{n}{K}$-approximation of the sum of all vertices' weights. We can then solve the maximum weighted independent set in Equation (1.1) for those new weights. Held et al. [33] chooses $K := \lfloor I_{\max}/n \rfloor$, where $I_{\max}$ is the maximum value we can store in an integer. Note we have to divide the value by n to avoid overflow at any step of the algorithm.

If our pricing algorithm, described in Chapter 4, fails to find an independent set with weight greater than 1, we can be certain that $1 + n/K$ is an upper bound to λ, and the aforementioned process of finding a feasible primal solution can be applied. Choosing K this way, allows us to solve the MWIS problem with precision n/K.

On the other hand, we need to take into consideration that a commercial solver, such as

Gurobi [31], has a maximum precision it works with (in this case, 10^{-9}, if set by the user). As such, if we find a constraint that is violated, but by an amount lower than that, the solver will fail to take it into consideration, which might result in endless loops. To avoid that, we limit K to, at maximum, 10^9.

Another point Held et al. [33, section 4] makes is that we can reduce the value of the weights without altering the strength of the lower bound. We can compute the maximum value r we can reduce from the total weight of all vertices as such:

$$r = \max\left\{r \in \mathbb{N} : \left\lceil \frac{\overline{\pi}_{\text{fixed}} - r}{K} \right\rceil = \left\lceil \frac{\overline{\pi}_{\text{fixed}}}{K} \right\rceil \right\} = \max\left\{0, \left(\sum_{v \in V} \overline{\pi}_{\text{fixed}}(v)\right) \bmod K - 1 \right\}.$$

Two approaches are proposed by Held et al. [33] on how to distribute this value among the vertices: one which uniformly distributes between all nodes and another which is based on reducing only the neighborhood of one specifically chosen vertex in the graph. They also demonstrate that no strategy dominates the other, so we use the first one.

For instance, if our numerically stable lower bound at a certain point is 4.25, applying the described vertex weight reduction technique might yield a new lower bound of 4.01. Despite this reduction, the practical lower bound remains 5, as the solution must ultimately be an integer. By employing this approach, we can terminate the column generation process earlier, as it reduces the number of independent sets with weights exceeding 1 while preserving the same effective lower bound.

3.2 Early stop

It is possible to arrive at a valid lower bound at any moment of the column generation process if we have an upper bound for how much the dual constraints are violated [18], in a process similar to what Malaguti et al. [53, equation (15)] does. In our case, the most violated constraint will correspond to the maximum weighted independent set (MWIS) of G with weights $\overline{\pi}_{\text{fixed}}$. Let $\bar{\alpha}(G, \overline{\pi}_{\text{fixed}})$ be an upper bound on the MWIS of G, we can obtain the lower bound $\overline{\pi}_{\text{fixed}} / \bar{\alpha}(G, \overline{\pi}_{\text{fixed}})$ for the chromatic number.

Although the fractional chromatic number can be greater than such lower bound, the latter can be obtained much earlier while solving the linear relaxation. This has two main advantages: we can devise a criteria that could stop the process of generating new columns, and apply the k-core reduction, as explained in Section 2.2, further reducing our graph.

When we have an upper bound ub and the current dual solution $\overline{\pi}$, if

$$\frac{\overline{\pi}_{\text{fixed}}}{\bar{\alpha}(G, \overline{\pi}_{\text{fixed}})} > ub - 1, \tag{3.6}$$

we conclude that the current B&P node cannot improve the current upper bound, thus it can be pruned.

Given a dual solution with fixed-point precision, we can compute the minimum value $\bar{\alpha}(G, \overline{\pi}_{\text{fixed}})$ must be for the current B&P node to not be pruned. We will use this fact to also guide our search in the pricing algorithm (Chapter 4).

3.3 Heuristics

One of the most critical factors affecting the performance of a branch-and-bound algorithm is the effectiveness of its primal heuristics, which directly influence the number of explored nodes. Additionally, any color class generated during the application of the following heuristics is incorporated into the restricted set of variables $\mathcal{S}$.

DSATUR-based greedy heuristic. The more prominent heuristic for the graph coloring problem is based on the saturation degree ordering, called DSATUR [7]. During such heuristic, we assign to each vertex an available color with the least index, one that has not been used in any of its neighbors. If none is available, we create a new color for it. We make those decisions following a non-increasing ordering of saturation degree, the number of different colors already used in the neighborhood of a vertex. When all vertices are colored, we have arrived at a feasible coloring for the given graph.

Not only this heuristic is fast, but, in practice, it also has decent performance on the number of colors used. We use it as a starting upper bound when initializing the algorithm, and for completing the coloring when using the rounding heuristic.

Rounding heuristic. When we have a fractional coloring, we might apply the idea of rounding up variables that are close to 1, using the intuition that those are the ones the model has the "most certainty". For that, we start from a fractional solution for the Model RLM, find sets that have the corresponding variable value greater than 0.55, add those to the current solution, and remove the vertices on those from the graph. With the remaining graph, we use the DSATUR heuristic to complete the solution. Since all chosen and generated sets are independent, and all vertices have to be colored by either stage, we arrive at a proper coloring.

As this heuristic can be applied to any valid fractional coloring (i.e., any solution for RLM), we not only employ it after finding the optimal value for RLM, but also during each step of solving the linear relaxation, before generating new columns. This has been a great addition to our algorithm when used with the early stop criteria described previously in Section 3.2.

Relax-and-Fix heuristic. Following a similar idea, we can arrive at our final heuristic. Starting from a fractional optimal solution $\overline{x}$ for our Model RLM with value lb and an upper bound ub, we can define the gap as $\text{gap} = ub - \lceil lb \rceil$. We know that we can only increase the value of our variables by a total amount of $\text{gap} - 1$, since otherwise, we would not improve the current upper bound. Based on that, we round up, in non-increasing order, variables that have its value in x greater than 0.55, while $\sum_{S \in R}(1 - x_S) \leq \text{gap}$, where R is the set of independent sets which the corresponding variables were rounded up. After that, we call for the column generation procedure to re-optimize the model, disregarding those vertices that are already covered, i.e., vertices $v \in \bigcup_{S \in R} S$. We repeat this process, alternating between rounding up and re-optimizing until all vertices are in at least, one chosen set, and since all sets are independent, we will terminate with a feasible coloring.

This heuristic is more computationally costly since we have to find the optimal [1] solution for each residual graph while generating new columns. It is important to note that, since we generate new sets through the branch-and-price process, improving the current upper bound

[1] Note that here we use optimal to refer to best numerically safe lower bound we are able to find by using the methods of fixed precision we described. This might differ from the actual optimal value of the formulation.

may warrant re-running the heuristics in the hope of further enhancing it.

3.4 Branching

With our lower and upper bounds in hand, we still might not be able to prove a coloring to be optimal if there is a gap between those bounds. In that case, we *branch* our search, splitting the search space into two problems. Traditionally, for a generic integer linear program with binary variables, this is done by selecting a variable whose value is fractional in the current optimal solution and creating two new problems: one where such variable has value 0, and another where it has value 1. In other words, we check the possibility of it being in an optimal solution or not, therefore covering all the original space of feasible solutions.

Malaguti et al. [53] calls this type of branching as *variable branching*, since we use the variables to dictate how we split the search space. Although intuitive and simple to implement, this strategy has two major drawbacks. First, even though the branch where we set an independent set to 1 generates a much simpler problem, since we can just disregard all the nodes in that set, the other branch barely changes it, only one variable among what is, in many cases, an exceedingly large number. Not only that, but on those branches, we would have to avoid generating columns that are prohibited (set to 0), as those might come up in our pricing algorithm. In that case, we have to solve what we call a *constrained pricing problem*, which greatly increases the challenge.

Another possibility is to use the *Ryan-Foster scheme* [21], which has been used for binary matrix models, such as bin-packing [74] and coloring [33]. On that scheme, when applied to coloring, we split the search space by choosing two non-adjacent vertices in the graph and creating two problems: one where both vertices are in the same color class, and another where they are in different. We will call it *contract* the first operation , while the other, *conflict*.

The strength of such strategy comes from how to implement such decisions. For the contract operation, we merge the chosen vertices into one adjacent to the union of the neighborhoods of the vertices, meaning they will be in the same color class, while for the conflict, we simply add an edge between them. This allows for the two new problems to be themselves coloring problems, without any additional constraints. Because of that, Malaguti et al. [53] calls this strategy an *edge branching*.

There are some important notes to take here. We store the instances on a stack to process the branch tree in a depth-first approach and to better utilize the data structure defined in **??**. Other orderings are not considered in our work, but the literature has given some thoughts about that, without any clear advantages over the DFS one [62].

Also, we do not remove independent sets that become invalid by the branching. When doing a contract operation, we substitute all occurrences of the chosen vertices in the sets of $\mathcal{S}$ for the new vertex, and, while doing a conflict operation, for any independent set which contains both, we create two new ones, removing one of the chosen vertices in each one.

What is left is to decide which pair of vertices will be chosen at each node of our branch tree. There are a plethora of rules for that in the literature, so we might as well test most of them.

1. Used by Held et al. [33], they define $p(v, w)$ as such:

$$p(v, w) := \frac{2 \sum_{S \in \mathcal{S}: v, w \in S} x_S}{\sum_{S \in \mathcal{S}: v \in S} x_S + \sum_{S \in \mathcal{S}: w \in S} x_S}.$$

   They chose the pair whose value is closest to 0.55.

2. Proposed by Mehrotra and Trick [57], the authors choose a vertex v on the most fractional column S_1, i.e., the column with corresponding variable closest to 0.5, select a column S_2 which covers v, and then select a vertex $w \in (S_1 \setminus S_2) \cap (S_2 \setminus S_1)$.

3. Similarly, as proposed by Foster and Ryan [21], it is possible to choose the vertices "at most in doubt if they are in the same color class or not". Silva and Schouery [74] represent it by the concept of affinity, defined by $\delta_{v,u} = \sum_{S \in \mathcal{S}: \{u,v\} \subseteq S} x_S$. They choose to branch on the pair whose affinity is closest to 0.55.

With all those techniques combined, we can divide the problem as much as we want, in search of better bounds, until there is no gap between them.

Chapter 4

Pricing

In Chapter 3, we discuss how we restrict the problem to a certain subset of variables since it is not viable to enumerate them all. By not giving the full range of possible color classes, we cannot guarantee that an optimal solution of RLM is also optimal for the linear relaxation of the unrestricted one. We must instead prove that there are no new variables, across all possible ones, that could be added and improve the current solution. If that is the case, then we have proven the current solution optimal for the unrestricted problem.

To determine whether a variable can reduce the value of the objective function, we have to compute its *reduced cost* for the current solution. If the value is negative, it means that the corresponding dual constraint is violated, thus we could improve the primal solution by making the primal variable available to the restricted model. Finding such variables is what we call the *pricing problem* or *column generation* procedure.

In our case, the reduced cost of a variable can be computed as 1 minus the sum of the weights of vertices in the corresponding set. The weight of vertex v is given by the value of the dual variable $\overline{\pi}(v)$ on a corresponding dual solution $\overline{\pi}$. In other words, we must check whether $1 - \sum_{v \in S} \overline{\pi}(v) \geq 0$ for all $S \in \mathcal{S}$. One way to check is to find a *maximum weighted independent set* (MWIS). If it has a weight at most 1, then no set could have a negative reduced cost, and the current solution is proven optimal.

For simplicity, in this chapter, since we will be talking mostly about the MWIS problem, we will refer to the weight function as π. Bear in mind that, in the context of our branch-and-price, this will come from the value of our dual solutions $\overline{\pi}$.

Let $G = (V, E)$ be a graph, and $\pi : V \to [0, 1]$ a weight function for each vertex. We can extend this notation where S is a vertex set to $\pi(S) = \sum_{v \in S} \pi(v)$. We define the maximum weighted independent set value $\alpha(G, \pi) = \max_{S \in \mathcal{S}} \pi(S)$, where $\mathcal{S}$ is the family of all independent sets in G. When G and π are clear from the context, we may omit it. For a general purpose LP solver, such as Gurobi [31], we can obtain the weight of each vertex by asking for the dual value of each constraint in our primal solution.

Plenty of times, more than one variable might have a negative reduced cost. Since most of our computational cost is in the pricing algorithm, it would be unwise to ignore those that might not be optimal, but correspond to violated dual constraints. One of our research hypotheses, as described in Chapter 1, is that enumerating a larger number of variables in the column generation process could yield faster solving time. Finding plenty of sets with negative

reduced cost could greatly speed up our technique, since the reduced cost is proportional to how much we could improve the solution by adding the variable.

Proving the optimality of a solution requires demonstrating that the upper bound $\bar{\alpha}(G, \pi)$, where π is the corresponding dual solution, is less than or equal to 1, but this process is time-consuming. Sometimes, it is possible to skip this process by employing a fast heuristic to find sets with negative reduced cost and return such columns to the model, for re-optimization. This is employed by many works in branch-and-price for the coloring problem [61, 53, 33]. The focus of this work on the benefits of enumerating a larger number of sets with high weight, we did not implement a heuristic for those, but it can be done in future works.

Lastly, not always we need to search for sets with weight greater than 1, as discussed in Section 3.2. Because of the gap between the current upper bound and the current lower bound, we can compute the needed weight and use this value instead (if greater than 1). For simplicity, we will refer to the weight of 1 as the threshold for the remainder of this dissertation.

4.1 Branch-and-reduce

Held et al. [33] solved the MWIS problem using *branch-and-bound*. They check possible decisions, such as taking (or not) a vertex in the set to be constructed, and prune branches that, by some upper bound, would not surpass the current best solution. The authors use a weighted clique cover heuristic to obtain such bounds, the same one we use and explain in Section 4.3. Although the authors add any set that has a negative reduced cost found while searching, their technique focuses on finding the optimal value. To compensate for that, Held et al. [33] use a fast heuristic method for the first few seconds instead of an exact method.

While studying the literature about the MWIS problem, most exact methods have little to no care about finding more sets with high weight on the way to the maximum. Recently, there have been improvements in how to solve the MWIS problem ([56, 82, 45, 87]), but one of them caught our attention: the work by Xiao et al. [87]. They apply a technique similar to Held et al. [33], called *branch-and-reduce*, which, in addition to taking similar steps of branching and pruning based on bounds, also applies reduction rules to further narrow the search space. This is especially useful to us since, by reducing the graph to those "vertices that matter the most", we could find not only the optimal value but also to better invest our computational resource to find plenty of sets with weight greater than 1.

Algorithm 2: Branch and Reduce

Input : The remaining vertex-weighted graph G with weights π.
Output : A maximum weighted independent set and a family of sets with weight greater than 1.

```
1  best ← ∅
2  new ← ∅
3  stack ← [{G, ∅}]
4  while stack is not empty do
5      (G, c) ← pop stack
6      (G, c) ← apply reduction rules to G
7      if G is "small enough" then
8          all ← enumerate all possible independent sets of G
9          Add those sets in all that have weight greater than 1
10         best ← max solution in all and best in terms of weight
11     else
12         s ← c + Greedy(G)
13         if π(s) > 1 then
14             Add s to new
15         if π(s) > π(best) then
16             best ← s
17         ub ← π(c) + CliqueCover(G)
18         if ub > 1 and ub > π(best) then
19             Find a confined vertex v of maximum degree and its
                confining set S_v
20             Add (G − N[S_v], c + S_v) and (G − v, c) to stack
21 return (best, new)
```

Algorithm 2 provides a pseudocode for the branch-and-reduce algorithm used in our work. We begin with the best solution and the current solution being the empty set (with weight zero). At each step, we might update the problem by either altering the graph and/or adding vertices to the current solution. First, we apply our reduction rules, explained in detail in Section 4.2, and then we try to expand the current solution to an insertion-wise maximal solution. If the current graph is small enough, we can enumerate all possible independent sets, save those that are heavier than 1, and return the best found among those (and the current best). Then, we compute an upper bound for the MWIS (see Section 4.3); if this upper bound implies that we cannot obtain a solution better than the current best solution, we return the latter value. Otherwise, we branch into two smaller subproblems, as it will be explained in Section 4.3, and return the best solution found between those and the best so far.

Note that, while applying the algorithm, we save all independent sets with a weight greater than 1 we come across, even if they are not optimal.

4.2 Reduction rules

As we commented in the introduction of this chapter, we want to reduce the number of vertices without altering the MWIS value from the graph. Some reduction rules will only remove vertices from the graph, while others may alter it, so we might need to do some "translations" to recover the actual MWIS from the algorithm. We will note when we need to make them, and also how they will work while discussing each rule.

Most of the rules work by proving that, for some subset of vertices in the graph, there exists a maximum weighted independent set that either contains the whole subset or none of it. By proving that, we can alter the graph in order to work with fewer vertices, easing up our problem.

Lastly, unless explicitly mentioned, Xiao et al. [87] proposed all rules, lemmas, and definitions in this section.

Rule 1. (Lamm et al. [45]) Let v be a vertex such that $\pi(v) \geq \pi(N[v])$. We can add v to the solution and remove $N[v]$ from the graph.

The first rule allows us to make a decision which would be made in any maximum weight independent set, since adding v is always a better or equivalent option than any subset of its neighbors. More generally, we have the following rule.

Rule 2. Let v be a vertex such that $\pi(v) \geq \alpha(G[N(v)], \pi)$, we can add v to the solution and remove $N(v)$ from the graph.

Since Rule 2 requires solving the MWIS for the subgraph, and this is a time-consuming task, we only test this rule for vertices with a "small enough" neighborhood. In those cases, we simply enumerate all possible independent sets to find its MWIS. From our tests, a neighborhood of 8 vertices is a good limit to apply Rule 2.

To further generalize Rules 1 and 2, it is possible to arrive at the definition of *heavy sets*. Such a set is a better (or equal) decision than any other independent set that contains a neighbor of such a heavy set.

Definition 4.1. A *heavy set* H is an independent set of vertices such that, for any independent set I in the induced subgraph $G[N(H)]$, $\pi(I) \leq \pi(N(I) \cap H)$ holds.

Following this definition, it is possible to determine that at least one MWIS contains such a set, which allows us to arrive at Lemma 4.2.

Lemma 4.2. For each heavy set there is at least one MWIS containing it. So we can add S to our current solution.

Identifying those sets would be useful, but it is computationally expensive, so we will not find them all. We can restrict it to those which are somewhat easier to check, as Rule 3 does.

Rule 3. Let u and v be two non-adjacent vertices with at least one common neighbor. If the number of their neighbors $|N(\{v, u\})|$ is at most 8, then we check whether $\{v, u\}$ is a heavy set.

The literature often uses the idea of one vertex dominating another [20, 1, 45, 72]. If $N(u) \subseteq N(v)$, we say that vertex v dominates u, and, for weighted graphs, if it is also true that $\pi(u) \geq \pi(v)$, we can eliminate the dominated vertex. To generalize such an idea, Xiao and Nagamochi [88] introduced the concept of *unconfined vertex*, extended to the weighted variant by Xiao et al. [87]. The intuition is to assume that the vertex belongs to every MWIS and try to find something that proves otherwise.

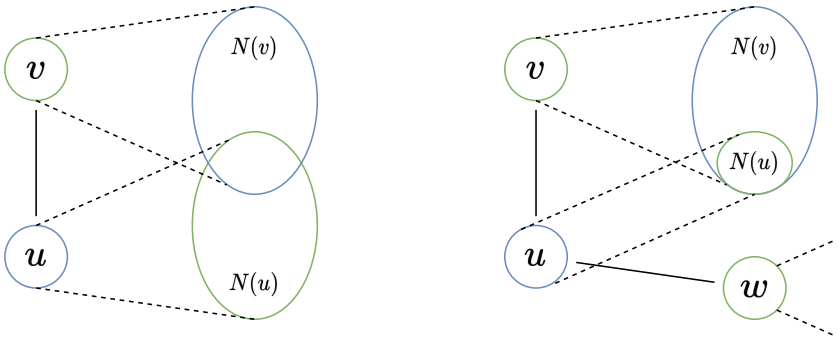

Figure 4.1: Two scenarios while solving the MWIS problem. In both cases, the color of the vertices correspond to the neighborhood they are on, and $\pi(u) \geq \pi(v)$.

For an example, consider the configuration illustrated on the left in Figure 4.1, where $\pi(u) \geq \pi(v)$, vertex v has its neighborhood $N(v)$ highlighted in blue, and one of its neighbors, vertex u, has its neighborhood $N(u)$ highlighted in green. Including v in an independent set necessitates the exclusion of all vertices in $N(v)$. Suppose the weight of u satisfies the inequality:

$$\pi(u) \geq \pi(v) + \pi\left(N(u) \setminus N(v)\right). \tag{4.1}$$

This inequality implies that the weight of u alone is at least the combined weight of v and the neighbors of u which are not neighbors of v. Consequently, selecting u instead of v allows the inclusion of additional vertices from $N(v) \setminus N(u)$, potentially yielding an independent set of equal or greater total weight. This creates a contradiction: if v were present in every MWIS, replacing v with u and adjusting the remaining vertices would produce an alternative MWIS excluding v. Therefore, the initial assumption that v is in every MWIS cannot hold under the given condition.

Now, suppose that is not the case, i.e., for every $u \in N(v)$, Equation (4.1) is false. Following our previous reasoning, we cannot conclude anything about v. But could we expand our analysis for a bigger set?

We now shift our focus to the right side of Figure 4.1, where w is the only neighbor of u that is not adjacent to v. By our supposition, v is in every MWIS. Assume that there is an MWIS I which does not contain w. Thus, we could swap v with u, and arrive at a set with greater or equal weight, since $\pi(u) \geq \pi(v)$, which contradicts the fact that v is in every MWIS. Therefore, if our initial supposition is true, w must also be in every MWIS. We can now repeat a similar reasoning, now considering v and w to be in every MWIS, which is the process we formalize below.

Lemma 4.3. If a vertex subset S is contained in all MWIS, then it must hold that, for each vertex $u \in N(S)$, any MWIS I satisfies that $\pi(u) < \pi(I \cap N(u))$.

Proof. For some S, suppose it exists a MWIS I such that $\pi(u) \geq \pi(I \cap N(u))$ for some $u \in N(S)$. Then, we could obtain a new MWIS by adding u and removing $S \cap N(u)$, which is a contradiction to the supposed fact that S is contained in all MWIS. □

It is worth defining some concepts to help us better write the following algorithm.

Definition 4.4. Let S be a set of vertices. A vertex $u \in N(S)$ is called a *child* of S if $\pi(u) \geq \pi(S \cap N(u))$. A child u is called an *extending child* if $|N(u) \setminus N[S]| = 1$ and $\pi(u) < \pi(N(u) \setminus N(S))$. The single vertex in $N(u) \setminus N[S]$ is called a *satellite* of S.

With that in mind, we can move into Algorithm 3. The idea is to suppose that vertex v is in every MWIS ($S := \{v\}$), and use Definition 4.4 to try to find an extending child to expand the set. If, at any point, we find a contradiction, we can conclude that v is not in every MWIS, and, therefore, can be removed from the graph. Note that a child is a vertex which makes the condition of Lemma 4.3 false.

Algorithm 3: Determine whether a vertex is confined or not.

Input : A vertex v
Output : Whether v is confined or unconfined

```
1 S ← {v}
2 while there exists an extending child to S do
3 |   Extend S by including the corresponding satellite to S
4 if there exists a child u such that π(u) ≥ π(N(u) \ N(S)) then
5 |   Halt and conclude v is unconfined
6 else
7 |   Conclude that the set S confines v, making it confined
```

The idea behind Algorithm 3 is to suppose the set $S := \{v\}$ is contained in all MWIS and, using Lemma 4.3, expand it. If, at any point, we find a contradiction, is means that S was never contained in all MWIS in the first place, so we can remove it from the graph without altering the MWIS' value. Note that, by halting and calling the vertex confined, we do not guarantee that it is contained in all MWIS, but we will use the constructed confining set to develop the branch strategy in Section 4.3.

The Algorithm 3 is illustrated in Figure 4.2, which started on the vertex with weight 0.2. The first graph represents a state where the set S contains two elements, which is expanded to S' in the next iteration. If the algorithm halts with S', we can conclude the initial vertex is confined by set S'. Suppose that the vertex with weight 0.4 was not present. In this case, we would halt in the first graph, with S, and conclude that the original vertex is unconfined.

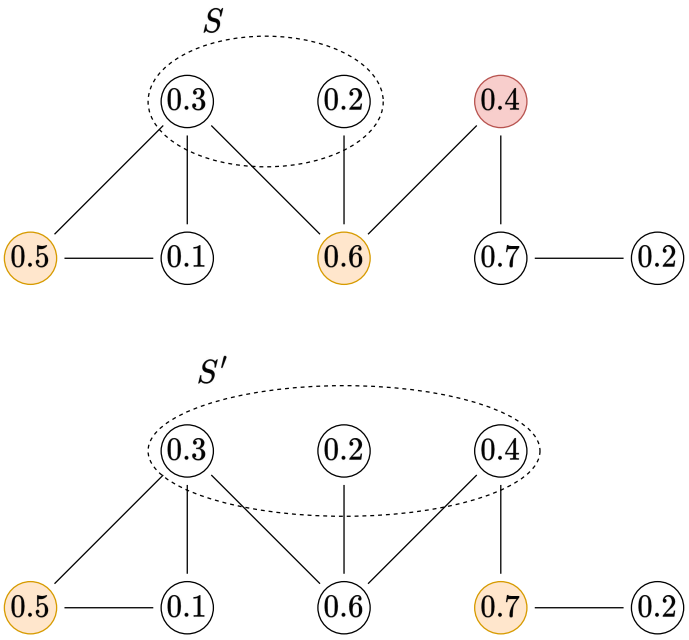

Figure 4.2: Example of an application of Algorithm 3. The number inside each vertex is their corresponding weight, while the orange vertices are *children*, and the red one is a sattelite. On the top, a state where the set S contains two vertices, and, on the botton, the next iteration of the algorithm.

Rule 4. Let v be an unconfined vertex, as determined by Algorithm 3. Then, we can remove v it from the graph without altering the MWIS value.

There are only two possibilities regarding its confining set, as described in Lemma 4.5, which will later be useful the define our branching rule (see Section 3.4).

Lemma 4.5. Let v be a confined vertex as defined by Algorithm 3. Either there is a MWIS that does not contain v or all MWIS contain the set S_v that confines v.

Proof. By how we constructed S_v, our initial supposition is that v is in every MWIS. If this is false, and it can be removed from G without reducing $\alpha(G)$. Otherwise, by Lemma 4.3, each satellite vertex must also be included, and, therefore, S_v must be contained in every MWIS. □

The following two rules come from the definition of a *simultaneous set*. Such a set is one that any MWIS either contain all of its vertices or none of them. If we find one simultaneous set, we can *merge* then into a single vertex, to simplify it.

Definition 4.6. To *merge* a set of vertices S into a single vertex is to create a vertex v^* such that, $N(v^*) = N(S)$, $\pi(v^*) = \sum_{u \in S} \pi(u)$ and S is removed from G.

When translating a solution, if v^* is present, we need to substitute it to the corresponding set it was merged from. Since v^* had the same weight then the set it represents, there is no need to change the weight of the set.

Rule 5. Let S be an independent set and all vertices on it have the same neighborhood, then S is a simultaneous set.

In this case, one can see that, if a vertex in a set described by Rule 5 is contained in a MWIS, no conflict would prevent the rest of the set from also being in the MWIS.

Rule 6. Let u and v be a pair of confined vertices by S_u and S_v, respectively. If $u \in S_v$ and $v \in S_u$, then $\{u, v\}$ is a simultaneous set.

By our definition of confining set S_v of vertex v, either v is not in all MWIS or its confining set is. If $u \in S_v$ and $v \in S_u$, means that they are both in all MWIS or none of them is.

The concept of an *alternative set* is similar to simultaneous set. We say that a set S is an alternative set if there is a MWIS which contains either S or $N(S)$.

Rule 7. If S is an alternative set, we can introduce a new vertex v^* with $N(v^*) = N(N(S))$, remove all vertices in $N[S]$, and set its weight as $\pi(v^*) = \pi(N(S)) - \pi(S)$.

Proof. To prove that the rule is valid, we need to prove $\alpha(G', \pi) = \alpha(G, \pi) - \pi(S)$, where G' is the graph after the proposed change, and $\pi(v) \geq 0$ for every vertex v of G. Let v^* be the new vertex introduced in G'. Let I be a MWIS of G such that I contains either S or $N(S)$. If $S \subseteq I$, then define $I' = I \setminus S$. Note that I' is an independent set of G', and its weight is

$$\pi(I') = \pi(I) - \pi(S).$$

Otherwise, if $N(S) \subseteq I$, define $I' = (I \setminus N(S)) \cup \{v^*\}$. Set I' is an independent set of G', and its weight is

$$\pi(I') = \pi(I) - \pi(N(S)) + \pi(v^*).$$

Since $\pi(v^*) = \pi(N(S)) - \pi(S)$, it follows that $\pi(I') = \pi(I) - \pi(S)$. In both cases, we have $\alpha(G', \pi) \geq \alpha(G, \pi) - \pi(S)$.

Now, we need to prove $\alpha(G) \geq \alpha(G') + \pi(S)$. Let I' be a MWIS of G'. If $v^* \in I'$, define $I = (I' \setminus \{v^*\}) \cup N(S)$. Note that I is an independent set of G, and its weight is

$$\pi(I) = \pi(I') - \pi(v^*) + \pi(N(S)) = \pi(I') + \pi(S).$$

Otherwise, if $v^* \notin I'$, define $I = I' \cup S$. This set I is an independent set of G, and its weight is

$$\pi(I) = \pi(I') + \pi(S).$$

In both cases, we have $\alpha(G) \geq \alpha(G') + \pi(S)$. □

For Rule 7, we have the following translation rule: if v^* is in the solution, add $N(S)$, otherwise, add S. In both cases, the weight of the corresponding nodes is already in the solution. To find such alternative sets, Xiao et al. [87] states the following lemmas.

Lemma 4.7. Let v be a vertex such that $N(v)$ is an independent set, and $u \in N(v)$ be a neighbor of v with minimum weight. If $\pi(N(v)) - \pi(u) \leq \pi(v) < \pi(N(v))$, then $\{v\}$ is an alternative set.

Proof. If there is a MWIS that contains either v or $N(v)$, the lemma is fulfilled. So we may assume that there is MWIS I that t does not contain v or at least one neighbor of v. Because of our hypothesis, even if the missing neighbor is the one with the least weight, we could still remove $N(v)$ from I and add v to arrive at a set heavier than I. So the set $I' = (I \setminus N(v)) \cup \{v\}$ is heavier than I, which is a contradiction. □

Xiao et al. [87] also propose two other ways of finding alternative sets, as explained in the Lemmas 4.8 and 4.9.

Lemma 4.8. Let (v_1, v_2, v_3, v_4) be a path such that $d(v_2) = d(v_3) = 2$. If $\pi(v_1) \geq \pi(v_2) \geq \pi(v_3) \geq \pi(v4)$, then $\{v_2\}$ is an alternative set.

Proof. It is possible to see that, any MWIS contains at least one vertex in $N[v_2]$, so we assume there is a MWIS I containing exactly one of $N(v_2)$. If $v_1 \in I$ and $v_3 \notin I$, then $v_4 \in I$ since one of $N[v_3]$ must be contained in I. Therefore, we can replace v_4 with v_3 and arrive at another MWIS which contains $N(v_2)$. If $v_1 \notin I$ and $v_3 \in I$, then we can replace v_3 with v_2 and arrive at another MWIS, this time, with v_2. Thus, $\{v_2\}$ is an alternative set. □

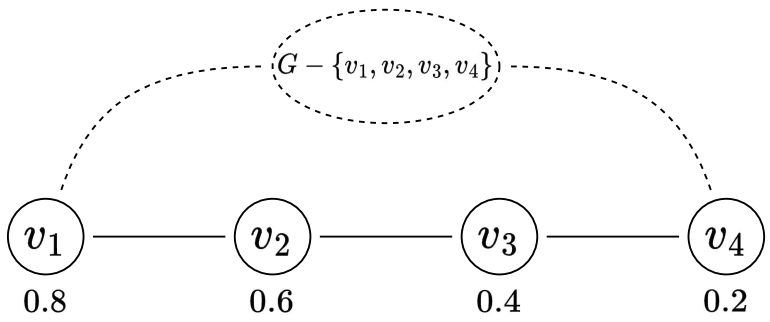

Figure 4.3: An example of Lemma 4.8.

An example of a graph where Lemma 4.8 is applicable can be found in Figure 4.3. It is possible to extend the Lemma 4.8 to arrive at the case where v_1 and v_4 are neighbors, making it a cycle.

Lemma 4.9. Let $(v_1, v_2, v_3, v_4, v_1)$ be a cycle such that $d(v_2) = d(v_3) = 2$. If $\pi(v_1) \geq \pi(v_2) \geq \pi(v_3)$, then $\{v_2\}$ is an alternative set.

An example of a graph where Lemma 4.9 is applicable can be found in Figure 4.4. With all that in mind, we can check for each of the Lemmas 4.7 to 4.9 to apply Rule 7.

Lastly, we need to investigate the case of isolated vertices, introduced by Lamm et al. [45]. A vertex is called isolated if the graph induced by its neighbors is a clique. To reduce such cases, the authors proposed the following rule.

Rule 8. (Lamm et al. [45]) Let v be a vertex such that $G[N(v)]$ is a clique, then

1. remove all $u \in N(v)$ such that $\pi(u) \leq \pi(v)$;

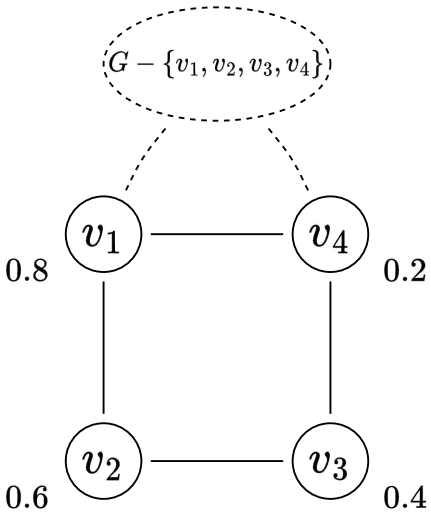

Figure 4.4: An example of Lemma 4.9.

2. for those remaining, if there are any, update $\pi(u) := \pi(u) - \pi(v)$, and remove v from the graph.

For Rule 8, it is important to track the vertices falling under the second case. If none of the vertices removed by this rule are included in the solution, we must add v to the solution since its weight has already been accounted for.

To apply such rules, we follow the ordering bellow [87]:

1. Rules 1 and 8 on vertices with degree 1.
2. Rules 7 and 8 on vertices with degree 2.
3. Rule 5.
4. Rule 2, Rule 7 with Lemma 4.7 and Rule 8.
5. Rules 4 and 6.

Steps 1, 2, and 4 keep a queue of vertices to be checked. On the initialization of the branch-and-reduce algorithm, we add all vertices to all queues, but as they are consumed, they are only re-added when their neighborhood changes. As of steps 3 and 5, they are computed when the total size of the graph reduces by 10%. From our tests, Rule 3 is not worth our time for the reductions it provides.

4.3 Branching and upper bounds

After reducing the graph, we are left with either a *small enough* graph or the need to branch into two smaller subproblems. In the first case, we could list all possible independent sets, but this approach would be computationally expensive. Therefore, it is crucial to carefully define what constitutes a "small" graph. Since a maximum weight independent set (MWIS) can be

found in polynomial time for any graph with a maximum degree of 2, we can enumerate all combinations of vertices with higher degrees. As proposed by Xiao et al. [87], we do such enumeration if the graph contains fewer than 9 vertices with a degree greater than 2.

For each possible combination of such high-degree vertices, the remaining graph consists of paths or cycles, as only vertices with degrees less than 3 remain.

To solve the MWIS problem in a path graph, we can use a *dynamic programming* approach in linear time. Let G be a path graph composed of vertices $(v_0, v_1, v_2, \ldots, v_n)$, with $n-1$ edges connecting consecutive vertices in this order and non-negative weights. We define $\mathrm{dp}[i]$ as the maximum weight of an independent set considering the first i vertices of the path graph.

The DP recurrence can be defined as follows:

$$\mathrm{dp}[i] = \begin{cases} w(v_0) & \text{if } i = 0, \\ \max\{w(v_0),\ w(v_1)\} & \text{if } i = 1, \\ \max\{\mathrm{dp}[i-1],\ \mathrm{dp}[i-2] + w(v_i)\} & \text{if } i \geq 2. \end{cases} \tag{4.2}$$

As a base, we have the cases when $i = 0$ or $i = 1$. In the first, we have only one choice, while in the latter, we can choose between the vertices v_0 and v_1. As for our recursion, at each value of i, we need to decide what is greater: the weight of the MWIS up to vertex v_{i-1} or the weight of the MWIS up to vertex v_{i-2} plus the weight of the i^{th} vertex. We can see that the second case is a valid solution since the graph is a path and the i^{th} vertex is adjacent only to the vertices v_{i-1} and v_{i+1}, which was not considered yet.

As for cycles, we can break them into two problems by choosing one vertex on it. In one, we add the chosen vertex to the solution and remove its two neighbors, while on the other, we simply remove it. In both cases, we are left with a path graph, which can have a MWIS found by Equation (4.2).

On the other hand, if the graph was not *small enough*, we need to divide it into small problems, splitting the search space into two. As we had previously mentioned, Xiao et al. [87] suggests a branching scheme based on Algorithm 3.

Lemma 4.5 gives us a way to branch our decisions: on one side, we can remove v from the graph, while on the other, we can add S_v to the current solution and remove $N(S_v)$ from the graph, creating two branch nodes.

An important remark here is that since using Rule 4 is very time-consuming, we avoid it while checking for ways to reduce the graph. But as we do need to run Algorithm 3, we might come across some unconfined vertices while trying to branch, and, therefore, we can apply the rule.

A confined vertex with a larger neighborhood would be the most useful since more vertices would be deleted by the branch that adds its confining set to the solution. Because of that, we test each vertex, in non-ascending order of degree, to see if it is a suitable one for branching.

We can prune branches that are unfruitful by an upper bound to the MWIS value. As our reduction rules can make the decision to add some vertices to the current solution, we need to determine an upper bound in the remaining graph. If it is lower than the weight of the heaviest set found so far or lower than our target value, we do not need to further search into this problem, and we can prune it away.

As an upper bound, we can use the *weighted clique cover* heuristically generated by the algorithm proposed by Lamm et al. [45]. Such a cover consists of a collection of cliques $C_1, C_2, \ldots, C_k \subseteq V$ with associated weights $w_1, w_2, \ldots, w_k$, satisfying $C_1 \cup C_2 \cup \cdots \cup C_k = V$ and $\sum_{i:v \in C_i} w_i \geq \pi(v)$ for all vertices v. The weight of the clique cover is defined as the sum of the weights of all cliques. This value serves as an upper bound to the MWIS value because the intersection of any clique and an independent set can contain at most one vertex. Since the weight of every vertex v is less than the sum of the weights of the cliques containing v, the total weight of any independent set cannot exceed the clique cover's total weight.

To compute this bound, we employ the same method used by Lamm et al. [45]. The algorithm is initialized with an empty set of cliques C and iterates over vertices in descending order of weight. For each vertex, the algorithm identifies the clique in C with the maximum weight to which the vertex can be added while preserving the clique property. If no such clique is found, a new clique containing only the vertex is created, with its weight set equal to the vertex's weight. The final weighted clique cover value is the sum of the weights of all cliques in C.

Chapter 5

Experiments

This chapter presents a comprehensive analysis of the experimental results obtained from evaluating our algorithm against the benchmark set by Held et al. [33], which is the only code we had access to. We begin by examining a variety of instance sets commonly used in the literature, called DIMACS [**DIMACS**], including random, geometric, and application-specific graphs, to assess the algorithm's effectiveness in diverse scenarios. Additionally, we explore the broader instance space defined by Smith-Miles and Bowly [76], called MATILDA, which promises to provide a more systematic coverage of graph properties influencing computational difficulty.

By analyzing performance across these sets, we aim to offer insights into the strengths and limitations of our algorithm, identifying the classes of instances where it excels and those where improvements are needed. Where possible, we also draw comparisons with results reported in the literature to provide a more holistic evaluation of our algorithm's performance. This chapter serves as a critical step in validating our contributions and situating them within the current state-of-the-art.

All experiments were conducted on an Intel Xeon CPU E5-2630 v4 @ 2.20GHz processor with 64 GB of RAM, running Ubuntu 22.04.3. The code was written in C++17, compiled and linked using GCC 11.4.0. As the general-purpose LP solver for both programs, we utilized Gurobi 11.0.3 [31]. It is important to note that the original paper by Held et al. [33] employed a significantly older version of the solver (version 3). Fortunately, no modifications were required to adapt the code to the newer version. To ensure a fair comparison, all experiments were executed using a single thread.

Traditionally, the literature on exact algorithms for the coloring problem has relied on the DIMACS instance set [**DIMACS**], which contains 137 instances. More recently, Smith-Miles and Bowly [76] proposed a significantly larger instance set comprising 8278 instances, designed to better cover the instance space of all possibilities, following the methodology outlined in Smith-Miles and Lopes [77]. By employing metrics such as density, algebraic connectivity, and energy, this larger instance set provides a more comprehensive representation of where each heuristics outperforms others, offering a clearer picture of their relative performance.

During this analysis, we will refer to instances as closed or solved by an algorithm when it is able to obtain an optimal coloring of an instance. The gap of an algorithm for an instance

is the difference between the upper and lower bound divided by the lower bound. As such, a solved instance has gap of zero, while an instance with lower bound 10 and upper bound 15 has a gap of 50%. We might also refer to the gap as an interval, such as $[\text{lb}, \text{ub}]$, where the values represent the lower and upper bounds, respectively.

5.1 DIMACS' instances

The DIMACS dataset comes from a series of initiatives aimed at promoting research on computational methods for graph coloring problems, evaluating alternative approaches through a shared testbed, and fostering discussions on current and future directions in computational combinatorial optimization. This instance set is the most widely used for the coloring problem and serves as a benchmark for evaluating algorithms.

A comparison of the performance of our algorithm and Held et al. [33] implementation can be found in Figure 5.1. It compares the accumulated amount of instances each algorithm was able to solve by the time it took and, on the right side, the accumulated amount of instances by the gap it left at the end. As an example, in 10 seconds, our code was able to solve 35% of instances. On the right side, we can see that Held et al. [33] code is able to obtain tighter gaps, with more than 50% of instances with gap lower than 1. From this figure we see that, in this dataset, both algorithms are able to solve around the same number of instance up to 3000 seconds. On those final moments, Held et al. [33] is able to close some instances, which guarantees it a higher place on the remainder of the curve.

On Figure 5.2 we see the accumulated distribution of time ratios on instances solved by both. Values lower than 1 indicates that our algorithm was faster. It is possible to see that our algorithm was able to solve 20% of instances in a third of the time it took Held et al. [33] to solve the same instances. It is also possible to observe that we are able to solve 45% of instances faster than Held et al. [33] can, with almost 10% being solved more than 10 times faster.

Among the 55 instances that both algorithms were able to solve, most of which were closed faster by the algorithm from Held et al. [33]. Curiously, when we filter to the 21 instances that took one algorithm at least a second to solve, we see that our algorithm was able to be faster in 11 of those. Across instances that the time difference was greater than 10 times, 7 out of 15 Held et al. [33] is able to finish in less than 0.025 with lower bound heuristics (e.g., clique number), which we do not posses.

Additionally, we observed that our algorithm required significantly more time during Gurobi's environment initialization, despite both implementations using the same version of the solver. While the reason for this discrepancy is yet to be determined, we believe it only has a meaningful impact on instances with a runtime of less than 0.5 seconds.

Across this section, we present numerous tables with the result from different classes of instances, but they all have the same set of columns. The columns n and D correspond to number of vertices and density of the instance, respectively. After that, $\lceil \chi_f \rceil$ represents the lower bound given by the fractional chromatic number, i.e., the final value when solving the linear relaxation of the root node, before any branching. The next two columns, marked by "Known", are the best known lower bounds across multiple publications we were able to find.

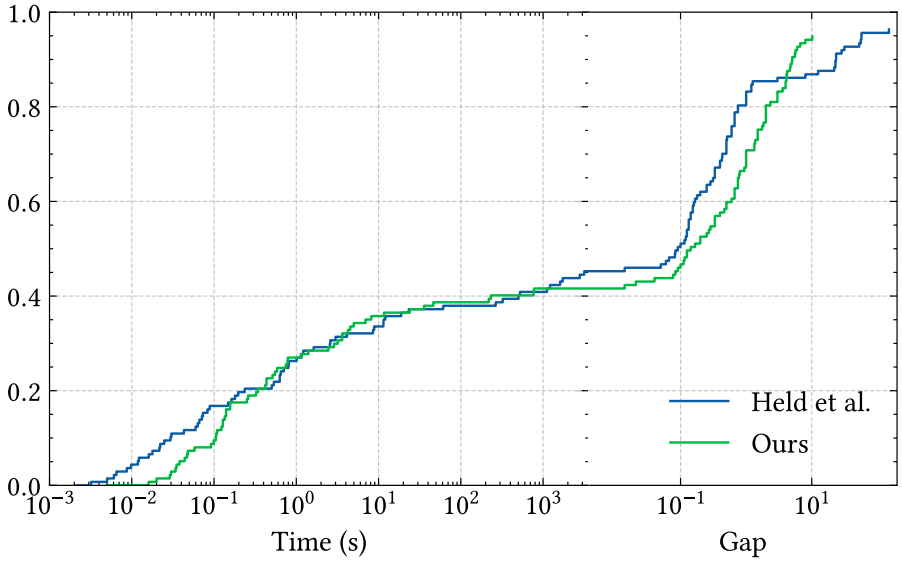

Figure 5.1: Cumulative distribution of instances solved by time and their gap on the DIMACS' instance set. In blue, the results of Held et al. [33] code, in green, our results.

Figure 5.2: Cumulative distribution of proportion of time of our algorithm in relation to what Held et al. [33] algorithm took in each instance both solved.

Do note that some results on those columns might come from different sources [79, 22, 89, 37, 41, 60, 65, 26, 80, 35, 38, 61, 52, 53, 59, 5]. The fact that we know the optimal coloring for an instance does not mean an exact algorithm was able to find both the lower and upper bound, some upper bounds were obtained by heuristics methods.

We then have a comparison between Held et al. [33] and ours results on the root node, each with number of sets generated, iterations and time to solve the root. There are some cases where our algorithm does not need to solve the linear relaxation in order to close the instance. When an algorithm did not solve the root, we use a dash to represent so, in all of the three corresponding columns. On those cases, only the separation and reduction rules with the DSATUR heuristic are enough to close the gap, so we only report the time it took, without the number of sets or iterations. The last 6 columns correspond to the final result from both Held et al. [33] and ours algorithm, with the lower and upper bounds found, and the total time spent. If the time spent was the time limit of 1 hour, we represent it by using a dash.

Some instances stand out in terms of the time required to solve them, as highlighted in Table 5.1. There are 11 instances which only one algorithm was able to solve, 9 of those Held et al. [33] implementation was the one to do so. Most of this performance comes from the fact we were unable to close the root node. We currently believe our lack of pricing heuristics might be what is causing such differences.

When looking at instances were the difference in time to solve was more than 10 times than the other algorithm, as shown in Table 5.2, we see 14 instances. 9 of them Held et al. [33] was able to close faster, but 8 of those were in less than 0.02 seconds. In contrast, on the remaining 5 instances that we are able to solve faster, we see running times well above the 6 seconds mark. It is possible to observe that there are still some work to do on those easy

				Known		Held et al. Root			Ours Root			Held et al.			Ours		
Instance	n	D	$\lceil\chi_f\rceil$	LB	UB	Sets	It.	Time	Sets	It.	Time	LB	UB	Time	LB	UB	Time
2-Insertions_3	37	.108	3	4	4	107	69	**.04**	316	85	.25	**4**	4	502.03	3	4	-
DSJC125.1	125	.095	5	5	5	1041	438	**3.00**	80259	79	644.73	5	**5**	2790.85	5	6	-
flat300_20_0	300	.477	20	20	20	1211	487	20.61	-	-	-	**20**	20	1114.11	18	20	-
flat300_26_0	300	.482	26	26	26	2797	935	39.51	-	-	-	**26**	26	1732.85	24	26	-
queen10_10	100	.594	10	11	11	486	179	**.60**	10268	26	78.44	**11**	11	265.39	10	11	-
school1	385	.258	14	14	14	8259	2152	125.06	-	-	-	**14**	**14**	1609.64	13	15	-
school1_nsh	352	.237	14	14	14	4781	1204	60.27	-	-	-	**14**	14	3168.81	13	14	-
will199GPIA	701	.028	7	7	7	354	203	2.92	-	-	-	**7**	7	3.00	4	7	-
DSJR500.5	500	.472	122	122	122	763	245	65.49	3488	46	**34.83**	122	132	-	122	**122**	35.66
r1000.5	1000	.477	234	234	234	2184	555	831.72	27581	73	**788.19**	234	248	-	234	**234**	791.52

Table 5.1: Results for instances that only one algorithm is able to close, but not the other.

				Known		Held et al. Root			Ours Root			Held et al.			Ours		
Instance	n	D	$\lceil\chi_f\rceil$	LB	UB	Sets	It.	Time	Sets	It.	Time	LB	UB	Time	LB	UB	Time
1-FullIns_3	30	.230	4	4	4	22	19	**.00**	114	43	.08	4	4	**.00**	4	4	.47
2-FullIns_3	52	.152	5	5	5	26	22	**.01**	181	65	.11	5	5	**.01**	5	5	.55
3-FullIns_3	80	.109	6	6	6	31	25	**.01**	157	89	.09	6	6	**.01**	6	6	.15
4-FullIns_3	114	.084	7	7	7	32	26	**.02**	372	138	.31	7	7	**.02**	7	7	.33
5-FullIns_3	154	.067	8	8	8	33	26	**.03**	341	166	.29	8	8	**.02**	8	8	1.03
DSJC125.9	125	.898	43	44	44	402	354	**.36**	391	11	1.74	44	44	**11.43**	44	44	203.75
DSJR500.1	500	.028	12	12	12	176	150	**.92**	6888	214	5.07	12	12	524.29	12	12	**6.03**
DSJR500.1c	500	.972	85	85	85	6204	6118	**21.18**	2320	54	24.80	85	85	1213.20	85	85	**26.34**
r125.1	125	.027	5	5	5	13	10	**.01**	-	-	.48	5	5	**.00**	5	5	.48
r125.5	125	.495	36	36	36	132	71	.56	300	29	**.39**	36	36	23.31	36	36	**.49**
r250.1c	250	.971	64	64	64	360	297	**.31**	-	-	.50	64	64	61.13	64	64	**.50**
r250.5	250	.477	65	65	65	382	141	5.99	1175	40	**3.53**	65	65	322.83	65	65	**4.04**
jean	80	.080	10	10	10	20	12	**.01**	-	-	.46	10	10	**.01**	10	10	.46
miles250	128	.048	8	8	8	22	16	**.02**	-	-	.15	8	8	**.01**	8	8	.15

Table 5.2: Results for instances that the running time of the algorithms differ by a factor greater than 10.

				Known		Held et al. Root			Ours Root			Held et al.			Ours		
Instance	n	D	$\lceil\chi_f\rceil$	LB	UB	Sets	It.	Time	Sets	It.	Time	LB	UB	Time	LB	UB	Time
DSJC125.1	125	.095	5	5	5	1041	438	**3.00**	80259	79	644.73	5	**5**	2790.85	5	6	-
DSJC125.5	125	.502	16	17	17	552	207	**.94**	3197	20	38.90	16	18	-	16	18	-
DSJC125.9	125	.898	43	44	44	402	354	**.36**	391	11	1.74	44	44	**11.43**	44	44	203.75
DSJC250.1	250	.103	7	7	8	3096	1161	724.27	-	-	-	**7**	10	-	4	10	-
DSJC250.5	250	.503	26	26	28	1421	489	**13.39**	30828	18	2420.36	26	30	-	26	**29**	-
DSJC250.9	250	.896	71	72	72	1915	1827	**6.08**	1302	16	30.24	71	72	-	71	72	-
DSJC500.1	500	.100	-	9	12	-	-	-	-	-	-	-	16	-	4	16	-
DSJC500.5	500	.502	43	43	47	3859	1305	296.03	-	-	-	**43**	65	-	32	**64**	-
DSJC500.9	500	.901	123	123	126	5457	5296	**63.98**	4081	17	585.62	-	-	-	123	127	-
DSJR500.1	500	.028	12	12	12	176	150	**.92**	6888	214	5.07	12	12	524.29	12	12	**6.03**
DSJR500.1c	500	.972	85	85	85	6204	6118	**21.18**	2320	54	24.80	85	85	1213.20	85	85	**26.34**
DSJR500.5	500	.472	122	122	122	763	245	65.49	3488	46	**34.83**	122	132	-	122	**122**	35.66
DSJC1000.1	1000	.099	-	10	20	-	-	-	-	-	-	-	**25**	-	3	27	-
DSJC1000.5	1000	.500	-	73	82	-	-	-	-	-	-	-	**114**	-	19	115	-
DSJC1000.9	1000	.900	-	216	222	-	-	-	-	-	-	-	301	-	215	**275**	-

Table 5.3: Results for the *DSJ* instance set.

				Known		Held et al. Root			Ours Root			Held et al.			Ours		
Instance	n	D	$\lceil\chi_f\rceil$	LB	UB	Sets	It.	Time	Sets	It.	Time	LB	UB	Time	LB	UB	Time
r125.1	125	.027	5	5	5	13	10	**.01**	-	-	.48	5	5	**.00**	5	5	.48
r125.1c	125	.968	46	46	46	67	23	**.02**	-	-	.12	46	46	**.02**	46	46	.12
r125.5	125	.495	36	36	36	132	71	.56	300	29	**.39**	36	36	23.31	36	36	**.49**
r250.1	250	.028	8	8	8	42	36	**.06**	179	88	.24	8	8	**.04**	8	8	.24
r250.1c	250	.971	64	64	64	360	297	**.31**	-	-	.50	64	64	61.13	64	64	**.50**
r250.5	250	.477	65	65	65	382	141	5.99	1175	40	**3.53**	65	65	322.83	65	65	**4.04**
r1000.1	1000	.029	20	20	20	104	78	1.71	605	75	**.87**	20	20	1.62	20	20	**1.41**
r1000.1c	1000	.971	96	96	98	151716	151611	3323.50	11820	93	**1920.82**	96	107	-	96	**98**	-
r1000.5	1000	.477	234	234	234	2184	555	831.72	27581	73	**788.19**	234	248	-	234	**234**	791.52

Table 5.4: Results for the *r* instance set.

instances, but it is also clear that our approach is able to provide good results when looking at those harder ones.

Lastly, we analyze each instance class from the DIMACS instance set.

DSJ. The first set of instances originates from the work of Johnson et al. [42]. DSJC instances are standard (n, p) random graphs with n vertices, where each pair of vertices has a probability p of being adjacent. DSJR instances are geometric graphs, while DSJR.c instances are complements of geometric graphs. The results for this instance set are shown in Table 5.3.

Both algorithms are able to close 4 out of 15 instances. Held et al. [33] algorithm is able to close $\mathrm{DSJC125.1}$ using three fourths of the time limit, but our cannot lower the upper bound. On the other hand, on $\mathrm{DSJR500.5}$, our algorithm is able to close the instance in a tenth of the time limit, while theirs cannot. Across the 3 instances solved by both, we are faster in 2 of them. Note that in plenty of instances, theirs is able to solve the root node faster than we do while enumerating less sets, and running more iterations, but this does not translate into faster total running time.

				Known		Held et al. Root			Ours Root			Held et al.			Ours		
Instance	n	D	$\lceil \chi_f \rceil$	LB	UB	Sets	It.	Time	Sets	It.	Time	LB	UB	Time	LB	UB	Time
fpsol2.i.1	496	.095	65	65	65	90	25	**.81**	-	-	.85	65	65	**.81**	65	65	.85
fpsol2.i.2	451	.086	30	30	30	83	45	.87	-	-	**.47**	30	30	.80	30	30	**.47**
fpsol2.i.3	425	.096	30	30	30	71	37	.64	-	-	**.48**	30	30	.63	30	30	**.48**
inithx.i.1	864	.050	54	54	54	93	38	**2.51**	-	-	3.01	54	54	**2.56**	54	54	3.01
inithx.i.2	645	.067	31	31	31	42	13	**.58**	-	-	.79	31	31	**.53**	31	31	.79
inithx.i.3	621	.073	31	31	31	45	16	**.66**	-	-	.80	31	31	**.71**	31	31	.80
mulsol.i.1	197	.203	49	49	49	96	49	**.22**	-	-	.50	49	49	**.20**	49	49	.50
mulsol.i.2	188	.221	31	31	31	41	11	**.05**	-	-	.19	31	31	**.06**	31	31	.19
mulsol.i.3	184	.233	31	31	31	47	16	**.08**	-	-	.16	31	31	**.08**	31	31	.16
mulsol.i.4	185	.232	31	31	31	47	18	**.08**	-	-	.18	31	31	**.09**	31	31	.18
mulsol.i.5	186	.231	31	31	31	44	15	**.07**	-	-	.14	31	31	**.07**	31	31	.14
zeroin.i.1	211	.185	49	49	49	73	26	.15	-	-	**.14**	49	49	.16	49	49	**.14**
zeroin.i.2	211	.160	30	30	30	39	11	**.06**	-	-	.19	30	30	**.06**	30	30	.19
zeroin.i.3	206	.168	30	30	30	40	12	**.07**	-	-	.17	30	30	**.07**	30	30	.17

Table 5.5: Results for the *Register Allocation* instance set.

Random. This instance set is also random, with the first number representing the vertex count and the second indicating the probability of two vertices being connected. The results for this instance set are shown in Table 5.4.

There are notable differences in performance across the instances. For r1000.1, r125.1, r125.1c, and r250.1, the performances of both algorithms are similar. However, for r125.5, r250.5, and r250.1c, our algorithm outperformed Held et al. [33] by a large margin. For r250.1c, we achieved the same result as Held et al. [33] but in significantly less time, 0.50 seconds versus 61.13 seconds. In the case of r1000.5, only our algorithm was able to close the instance, and for r1000.1c, while neither algorithm solved it, ours reached a better upper bound.

Register Allocation. This instance set is based on one of the applications mentioned in Chapter 1: register allocation, and it is derived from real code compilation. The results for this instance set are shown in Table 5.5.

Here we want to remind the reader about the instances where no column generation was necessary to solve the root node. In this set, no instance required us to even create the linear model, our reduction and separation rules sufficed in order to close the root.

We observe that both algorithms solved all instances, most of the time in less than a second. However, our performance on these very fast instances was somewhat affected, as we only solved 3 instances faster than Held et al. [33]. As previously mentioned, we hypothesize that this under-performance is due to Gurobi's environment initialization, despite both programs using the same version of the solver.

Optical Network Design. This instance set is derived from real-life optical network design problems, where each vertex represents a light-path in the network, and edges correspond to intersecting paths. The results for this instance set are shown in Table 5.6.

This instance set is highly sparse, which poses a challenge for formulations based on inde-

				Known		Held et al. Root			Ours Root			Held et al.			Ours		
Instance	n	D	$\lceil\chi_f\rceil$	LB	UB	Sets	It.	Time	Sets	It.	Time	LB	UB	Time	LB	UB	Time
wap01a	2368	.040	-	41	41	-	-	-	-	-	-	-	47	-	7	47	-
wap02a	2464	.037	-	40	40	-	-	-	-	-	-	-	46	-	9	46	-
wap03a	4730	.026	-	40	43	-	-	-	-	-	-	-	57	-	5	**56**	-
wap04a	5231	.022	-	40	41	-	-	-	-	-	-	-	**46**	-	7	49	-
wap05a	905	.105	50	50	50	122	62	8.42	171	56	**.73**	50	50	8.56	50	50	**1.19**
wap06a	947	.097	40	40	40	1801	466	264.07	-	-	-	**40**	44	-	19	**43**	-
wap07a	1809	.063	-	40	41	-	-	-	-	-	-	-	47	-	9	**46**	-
wap08a	1870	.060	-	40	40	-	-	-	-	-	-	-	**44**	-	9	45	-

Table 5.6: Results for the *Optical Network* instance set.

				Known		Held et al. Root			Ours Root			Held et al.			Ours		
Instance	n	D	$\lceil\chi_f\rceil$	LB	UB	Sets	It.	Time	Sets	It.	Time	LB	UB	Time	LB	UB	Time
le450_5a	450	.057	5	5	5	32688	8799	1282.09	-	-	-	**5**	10	-	3	10	-
le450_5b	450	.057	5	5	5	28373	7713	1298.43	-	-	-	**5**	**7**	-	3	9	-
le450_5c	450	.097	5	5	5	55381	14813	1114.67	-	-	-	**5**	11	-	4	**8**	-
le450_5d	450	.097	5	5	5	23412	6257	513.76	-	-	-	**5**	11	-	4	11	-
le450_15a	450	.081	15	15	15	929	274	24.72	-	-	-	**15**	17	-	6	17	-
le450_15b	450	.081	15	15	15	856	249	22.92	-	-	-	**15**	17	-	8	**16**	-
le450_15c	450	.165	15	15	15	16334	4145	725.33	-	-	-	**15**	24	-	6	**23**	-
le450_15d	450	.166	15	15	15	18020	4769	853.12	-	-	-	**15**	24	-	6	24	-
le450_25a	450	.082	25	25	25	147	69	2.62	1149	63	**2.49**	25	25	**2.59**	25	25	3.72
le450_25b	450	.082	25	25	25	186	82	4.21	960	42	**1.45**	25	25	4.13	25	25	**.63**
le450_25c	450	.172	25	25	25	1311	351	37.52	-	-	-	**25**	28	-	9	28	-
le450_25d	450	.172	25	25	25	1218	322	31.68	-	-	-	**25**	29	-	8	**28**	-

Table 5.7: Results for the *Leighton* instance set.

pendent sets, as the number of possible sets increases exponentially. With only one instance solved by both algorithms, we found that our approach was faster in closing that instance. Additionally, we observed that in 3 instances, we were able to find a better upper bound among the tested algorithms, while Held et al. [33] achieved a better upper bound on 2 instances.

Leighton. This instance set consists of Leighton graphs, each with a known optimal solution. Such graphs are related to Leighton's Theorem [47], which states that if a pair of finite graphs have a common covering space, then they have a common finite covering. The results for this instance set are shown in Table 5.7.

Like the previous instance sets, this is a set of sparse instances. However, in this case, the fractional chromatic number provides a tight lower bound for the chromatic number of the graph. We observe that Held et al. [33] algorithm is able to solve the root node for all instances, whereas our algorithm can only do so for the instances theirs was able to close. Despite this, our algorithm consistently produces better or equal upper bounds compared to Held et al. [33].

Almost 3. This instance set consists of graphs that are almost 3-colorable but contain a "hard-to-find" four-clique embedded. The results for this instance set are shown in Table 5.8.

				Known		Held et al. Root			Ours Root			Held et al.			Ours		
Instance	n	D	$\lceil \chi_f \rceil$	LB	UB	Sets	It.	Time	Sets	It.	Time	LB	UB	Time	LB	UB	Time
mug88_1	88	.038	4	4	4	351	320	**.73**	6918	183	4.26	4	4	**.63**	4	4	4.97
mug88_25	88	.038	4	4	4	381	340	**.63**	6606	195	4.60	4	4	**.65**	4	4	5.02
mug100_1	100	.034	4	4	4	487	443	**1.14**	11103	218	8.40	4	4	**1.16**	4	4	11.14
mug100_25	100	.034	4	4	4	434	385	**1.10**	15772	227	11.28	4	4	**1.01**	4	4	10.03

Table 5.8: Results for the *Almost 3* instance set.

				Known		Held et al. Root			Ours Root			Held et al.			Ours		
Instance	n	D	$\lceil \chi_f \rceil$	LB	UB	Sets	It.	Time	Sets	It.	Time	LB	UB	Time	LB	UB	Time
myciel3	11	.364	3	4	4	11	9	**.00**	25	16	.08	4	4	**.02**	4	4	.04
myciel4	23	.281	4	5	5	33	24	**.01**	79	31	.10	5	5	**11.51**	5	5	61.82
myciel5	47	.218	4	6	6	85	51	**.02**	323	55	.48	4	6	-	4	6	-
myciel6	95	.169	4	7	7	149	76	**.12**	1979	105	11.65	4	7	-	4	7	-
myciel7	191	.130	5	8	8	334	200	**2.42**	12634	221	288.31	5	8	-	5	8	-

Table 5.9: Results for the *Mycielski* instance set.

Although the set is composed of "easy" instances, our algorithm does not perform particularly well here, being, on average, an order of magnitude slower than the other algorithm.

Mycielski. These instances are based on the Mycielski transformation. The graphs are challenging to solve because they are triangle-free, yet their chromatic number increases as the problem size grows. The results for this instance set are shown in Table 5.9.

In this set, we observe a very similar performance in terms of bounds, with a notable difference in the time to solve the myciel4 instance.

Insertion. k-Insertion graphs and Full Insertion graphs are generalizations of Mycielski graphs, with additional inserted nodes that increase the graph size without significantly increasing its density. The results for this instance set are shown in Table 5.10.

This is a more challenging set of instances, with most of them not being solved within the time limit. Held et al. [33] was able to solve one more instance than our algorithm, and for the instances that both algorithms closed, Held et al. [33]'s approach was considerably faster. This difference is primarily due to Held et al. [33] closing the instances at the root relaxation much faster than our algorithm, likely due to more effective pricing heuristics.

Latin Squares. This instance set is derived from the Latin square problem. A Latin square is an $n \times n$ array filled with symbols, where each symbol appears exactly once in each row and exactly once in each column. The results for this instance set are shown in Table 5.11.

This set includes some challenging instances, where neither algorithm was able to solve the problem. We also observe that, in these cases, our algorithm resulted in worse upper bounds compared to the other approach.

School. This instance set is based on class scheduling graphs, both with and without study halls (_nsh), another practical case highlighted in Chapter 1. The results for this instance set are shown in Table 5.12. We observe that these are challenging instances; however, Held et al.

				Known		Held et al. Root			Ours Root			Held et al.			Ours		
Instance	n	D	$\lceil\chi_f\rceil$	LB	UB	Sets	It.	Time	Sets	It.	Time	LB	UB	Time	LB	UB	Time
1-FullIns_3	30	.230	4	4	4	22	19	**.00**	114	43	.08	4	4	**.00**	4	4	.47
1-FullIns_4	93	.139	4	5	5	48	29	**.02**	1961	120	2.36	4	5	-	4	5	-
1-FullIns_5	282	.082	4	6	6	150	73	**.49**	70395	251	526.56	4	6	-	4	6	-
2-FullIns_3	52	.152	5	5	5	26	22	**.01**	181	65	.11	5	5	**.01**	5	5	.55
2-FullIns_4	212	.072	5	6	7	82	47	**.13**	9147	185	22.29	5	6	-	5	6	-
2-FullIns_5	852	.034	5	7	7	187	106	1.59	-	-	-	**5**	7	-	4	7	-
3-FullIns_3	80	.109	6	6	6	31	25	**.01**	157	89	.09	6	6	**.01**	6	6	.15
3-FullIns_4	405	.043	6	7	7	63	48	**.14**	8271	424	44.00	6	7	-	6	7	-
3-FullIns_5	2030	.016	6	8	8	200	107	5.38	-	-	-	**6**	8	-	4	8	-
4-FullIns_3	114	.084	7	7	7	32	26	**.02**	372	138	.31	7	7	**.02**	7	7	.33
4-FullIns_4	690	.028	7	7	8	75	55	**.33**	465550	606	2030.02	7	8	-	7	8	-
4-FullIns_5	4146	.009	7	9	9	200	114	32.33	-	-	-	**7**	9	-	5	9	-
5-FullIns_3	154	.067	8	8	8	33	26	**.03**	341	166	.29	8	8	**.02**	8	8	1.03
5-FullIns_4	1085	.019	8	9	9	90	56	**.73**	23785	1129	622.86	**8**	9	-	6	9	-
1-Insertions_4	67	.105	3	5	5	224	101	**.12**	1429	98	1.87	3	5	-	3	5	-
1-Insertions_5	202	.060	3	6	6	1216	378	**5.02**	101561	240	2209.24	3	6	-	3	6	-
1-Insertions_6	607	.034	4	4	7	21765	18622	2088.64	-	-	-	**4**	7	-	3	7	-
2-Insertions_3	37	.108	3	4	4	107	69	**.04**	316	85	.25	**4**	4	502.03	3	4	-
2-Insertions_4	149	.049	3	5	5	924	547	**2.98**	33132	194	162.03	3	5	-	3	5	-
2-Insertions_5	597	.022	3	6	6	3281	1728	210.93	-	-	-	3	6	-	3	6	-
3-Insertions_3	56	.071	3	4	4	210	106	**.09**	852	151	1.01	3	4	-	3	4	-
3-Insertions_4	281	.027	3	5	5	1701	978	30.23	-	-	-	3	5	-	3	5	-
3-Insertions_5	1406	.010	-	4	6	-	-	-	-	-	-	**3**	6	-	2	6	-
4-Insertions_3	79	.051	3	4	4	388	180	**.33**	2304	205	3.67	3	4	-	3	4	-
4-Insertions_4	475	.016	3	5	5	3740	2077	296.63	-	-	-	**3**	5	-	2	5	-

Table 5.10: Results for the *Insertion* instance set.

				Known		Held et al. Root			Ours Root			Held et al.			Ours		
Instance	n	D	$\lceil\chi_f\rceil$	LB	UB	Sets	It.	Time	Sets	It.	Time	LB	UB	Time	LB	UB	Time
latin_square_10	900	.760	90	90	97	894	220	57.37	-	-	-	90	129	-	90	**126**	-
qg.order30	900	.065	30	30	32	877	307	166.04	-	-	-	**30**	**32**	-	6	36	-
qg.order40	1600	.049	40	40	40	1500	740	972.68	-	-	-	**40**	**42**	-	9	45	-
qg.order60	3600	.033	-	60	60	-	-	-	-	-	-	-	**63**	-	12	68	-
qg.order100	10000	.020	-	100	100	-	-	-	-	-	-	-	**106**	-	15	115	-

Table 5.11: Results for the *Latin* instance set.

				Known		Held et al. Root			Ours Root			Held et al.			Ours		
Instance	n	D	$\lceil\chi_f\rceil$	LB	UB	Sets	It.	Time	Sets	It.	Time	LB	UB	Time	LB	UB	Time
school1	385	.258	14	14	14	8259	2152	125.06	-	-	-	**14**	**14**	1609.64	13	15	-
school1_nsh	352	.237	14	14	14	4781	1204	60.27	-	-	-	**14**	14	3168.81	13	14	-

Table 5.12: Results for the *School* instance set.

				Known		Held et al. Root			Ours Root			Held et al.			Ours		
Instance	n	D	$\lceil \chi_f \rceil$	LB	UB	Sets	It.	Time	Sets	It.	Time	LB	UB	Time	LB	UB	Time
anna	138	.052	11	11	11	17	8	**.01**	-	-	.04	11	11	**.02**	11	11	.08
david	87	.109	11	11	11	17	8	**.00**	19	14	.03	11	11	**.01**	11	11	.08
huck	74	.111	11	11	11	23	14	**.01**	-	-	.05	11	11	**.01**	11	11	.05
jean	80	.080	10	10	10	20	12	**.01**	-	-	.46	10	10	**.01**	10	10	.46

Table 5.13: Results for the *Games and Books* instance set.

				Known		Held et al. Root			Ours Root			Held et al.			Ours		
Instance	n	D	$\lceil \chi_f \rceil$	LB	UB	Sets	It.	Time	Sets	It.	Time	LB	UB	Time	LB	UB	Time
miles250	128	.048	8	8	8	22	16	**.02**	-	-	.15	8	8	**.01**	8	8	.15
miles500	128	.144	20	20	20	37	19	**.03**	70	21	.04	20	20	**.03**	20	20	.10
miles750	128	.260	31	31	31	51	22	.07	63	20	**.04**	31	31	.07	31	31	.07
miles1000	128	.396	42	42	42	76	36	.19	85	21	**.07**	42	42	**.18**	42	42	.28
miles1500	128	.640	73	73	73	96	24	.21	84	18	**.13**	73	73	.24	73	73	**.11**

Table 5.14: Results for the *Miles* instance set.

[33]'s algorithm was able to solve them.

Games and Books. The graphs in this instance set are derived from Donald Knuth's Stanford GraphBase. For a given work of literature, a graph is constructed where each node represents a character, and two nodes are connected by an edge if the corresponding characters interact in the book. The results for this instance set are shown in Table 5.13.

Knuth generated graphs for five classic works: Tolstoy's *Anna Karenina* (anna), Dickens' *David Copperfield* (david), Homer's *Iliad* (homer), Twain's *Huckleberry Finn* (huck), and Hugo's *Les Misérables* (jean).

The games120 graph represents the games played in a college football season. In this graph, nodes correspond to college teams, and an edge connects two nodes if the respective teams played against each other during the season. Knuth provides this graph for the 1990 college football season.

Two noteworthy observations can be made regarding this set: for the instances homer and jean, our algorithm took significantly more time to close these instances compared to the other algorithm.

Miles. The graphs in this instance set are derived from Donald Knuth's Stanford GraphBase. Similar to geometric graphs, nodes in these graphs are placed in space, with two nodes connected if they are sufficiently close. The nodes represent a set of United States cities, and the distances between them are given by road mileage from 1947. The results for this instance set are shown in Table 5.14.

There is little difference in performance between the compared algorithms for this instance set, as both were able to close all instances in less than half a second.

Queen. The graphs in this instance set are derived from Donald Knuth's Stanford GraphBase. For an $q \times q$ chessboard, a queen graph is constructed where each node represents a

				Known		Held et al. Root			Ours Root			Held et al.			Ours		
Instance	n	D	$\lceil \chi_f \rceil$	LB	UB	Sets	It.	Time	Sets	It.	Time	LB	UB	Time	LB	UB	Time
queen5_5	25	.533	5	5	5	5	2	**.00**	5	1	.02	5	5	**.01**	5	5	.02
queen6_6	36	.460	7	7	7	59	35	**.02**	204	20	.09	7	7	.50	7	7	**.12**
queen7_7	49	.405	7	7	7	99	45	**.04**	477	20	.40	7	7	1.21	7	7	**.48**
queen8_8	64	.361	9	9	9	186	80	**.10**	1133	23	2.39	9	9	8.84	9	9	**2.41**
queen8_12	96	.300	12	12	12	111	53	**.12**	2979	24	3.17	12	12	12.06	12	12	**8.34**
queen9_9	81	.652	9	10	10	405	151	**.30**	3023	21	11.34	10	10	**18.85**	10	10	190.88
queen10_10	100	.594	10	11	11	486	179	**.60**	10268	26	78.44	**11**	11	265.39	10	11	-
queen11_11	121	.545	11	11	11	569	203	**1.05**	39703	27	141.79	11	12	-	11	12	-
queen12_12	144	.504	12	12	12	698	231	**1.60**	107982	31	504.37	12	13	-	12	13	-
queen13_13	169	.469	13	13	13	724	254	**2.61**	639886	36	2425.43	13	15	-	13	**14**	-
queen14_14	196	.438	14	14	14	938	293	4.51	-	-	-	**14**	**16**	-	11	19	-
queen15_15	225	.411	15	15	15	971	297	7.86	-	-	-	**15**	**17**	-	9	20	-
queen16_16	256	.387	16	16	17	1136	360	11.87	-	-	-	**16**	21	-	8	21	-

Table 5.15: Results for the *Queen* instance set.

				Known		Held et al. Root			Ours Root			Held et al.			Ours		
Instance	n	D	$\lceil \chi_f \rceil$	LB	UB	Sets	It.	Time	Sets	It.	Time	LB	UB	Time	LB	UB	Time
flat300_20_0	300	.477	20	20	20	1211	487	20.61	-	-	-	**20**	20	1114.11	18	20	-
flat300_26_0	300	.482	26	26	26	2797	935	39.51	-	-	-	**26**	26	1732.85	24	26	-
flat300_28_0	300	.484	28	28	28	1886	642	24.70	-	-	-	**28**	**33**	-	27	41	-
flat1000_50_0	1000	.490	-	50	50	-	-	-	-	-	-	**50**	113	-	17	**110**	-
flat1000_60_0	1000	.492	-	60	60	-	-	-	-	-	-	-	**112**	-	16	113	-
flat1000_76_0	1000	.494	-	72	81	-	-	-	-	-	-	-	115	-	18	115	-

Table 5.16: Results for the *Flat* instance set.

square on the board. Two nodes are connected by an edge if their corresponding squares lie in the same row, column, or diagonal.

Unlike some other graphs, the coloring problem on this graph has a natural interpretation: given such a chessboard, is it possible to place q sets of q queens on the board so that no two queens from the same set share the same row, column, or diagonal? The answer is yes if, and only if, the graph's chromatic number is q. The results for this instance set are shown in Table 5.15.

For this instance set, the performance of both algorithms is quite similar for smaller instances. Although we solve instances queen8_8 and queen8_12 faster, the algorithm by Held et al. [33] solves queen9_9 much faster and is the only one capable of solving queen10_10.

For instances not solved by either algorithm, we observe mixed results: our algorithm produced a better upper bound in one case but failed to match Held et al. [33]'s upper bound in three others. Overall, for unsolved instances, their algorithm demonstrated superior performance.

Flat. A given graph G is called flat if each edge of G belongs to at most two triangles of G. The results for this instance set are shown in Table 5.16. Here we are able to see once more our algorithm has some trouble raising the lower bound in instance flat_300_20_0 and

				Known		Held et al. Root			Ours Root			Held et al.			Ours		
Instance	n	D	$\lceil \chi_f \rceil$	LB	UB	Sets	It.	Time	Sets	It.	Time	LB	UB	Time	LB	UB	Time
abb313GPIA	1557	.044	8	9	9	15743	7192	2828.20	-	-	-	8	10	-	-	-	-
ash331GPIA	662	.019	4	4	4	286	187	5.30	-	-	-	**4**	6	-	3	6	-
ash608GPIA	1216	.011	4	4	4	2013	1163	1843.92	-	-	-	**4**	6	-	2	**5**	-
ash958GPIA	1916	.007	-	4	4	-	-	-	-	-	-	-	6	-	2	6	-
will199GPIA	701	.028	7	7	7	354	203	2.92	-	-	-	**7**	7	3.00	4	7	-

Table 5.17: Results for the *Matrix Partitioning* instance set.

flat_300_26_0, even if we are able to provide a better upper bound to instance flat1000_50_0 than Held et al. [33] could.

Matrix Paritioning. Graphs obtained from a matrix partitioning problem in the segmented columns approach to determine sparse Jacobian matrices. The results for this instance set are shown in Table 5.17. It is possible to observe a dificulty for Held et al. [33] algorithm to find better upper bounds, while ours was able to get similar or better upper bounds. Interestingly, on instance will199GPIA, even though Held et al. [33] took 3 seconds, ours was not able to raise the lower bound.

5.2 MATILDA's Instances

While it could be argued that the DIMACS instances are sufficient to demonstrate the effectiveness of an algorithm for solving the graph coloring problem—given that they cover some of the most important classes of instances—it became evident early in this research that these instances alone do not provide a comprehensive evaluation, particularly in terms of algorithmic speed. With only 10% of instances solved requiring more than a second to be solved, the rate at which algorithms achieve that is insufficient to definitively determine which performs "faster".

To enhance our analysis, we utilized a set of instances proposed by Smith-Miles and Bowly [76], which is called the MATILDA dataset. The authors employed a methodology to identify the instance characteristics that most contribute to its "difficulty". Their findings revealed that density, algebraic connectivity, and energy are the three primary properties influencing an instance's hardness for the studied heuristics algorithms. The algebraic connectivity of graph G is the second-smallest eigenvalue of the Laplacian matrix of G and reflects how well connected G is. The energy of graph G is the sum of absolute values of the eigenvalues of the adjacency matrix of G.

Using genetic algorithms, they generated a dataset comprising over 8000 instances, effectively spanning the space of possible instances based on these properties. To visualize this space, the authors projected the three-dimensional feature space onto a plane using the fol-

lowing formula:

$$\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0.559 & 0.614 & 0.557 \\ -0.702 & -0.007 & 0.712 \end{bmatrix} \begin{bmatrix} \text{density} \\ \text{algebraic connectivity} \\ \text{energy} \end{bmatrix}$$

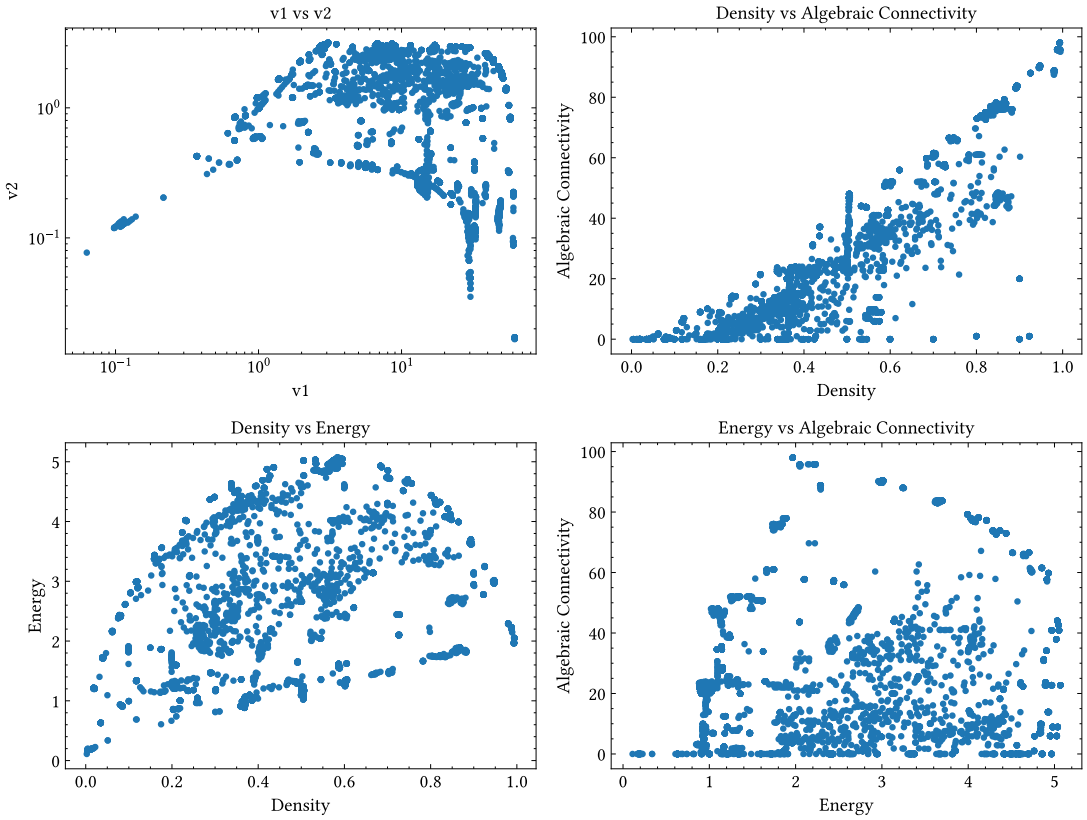

Figure 5.3: Distribution of MATILDA's instances across the instance space. Each graph represents a projection of such space in a plane.

The coverage of the instance space is illustrated in Figure 5.3. However, there are some drawbacks to these instances. Most notably, all instances contain exactly 100 vertices. Additionally, approximately 20% of them have chromatic numbers below 3, making them solvable in polynomial time. To avoid cluttering our results with such trivial instances, we excluded them from our tests, and we are left with 6630 instances.

Figure 5.4 and Figure 5.5 illustrate the performance of our implementation compared to the algorithm proposed by Held et al. [33].

As shown in Figure 5.4, around the 10-second mark, both algorithms close the same number of instances. However, Held et al. [33] solves more instances in the remaining time. Upon examining the logs of instances solved by their algorithm but not by ours, we observe that in many cases, our implementation could not sufficiently increase the lower bound to close the instance. Conversely, there are only a few cases where we failed to achieve the same upper bound.

In Figure 5.5, we observe that our algorithm solves 35% of instances faster than Held et al.

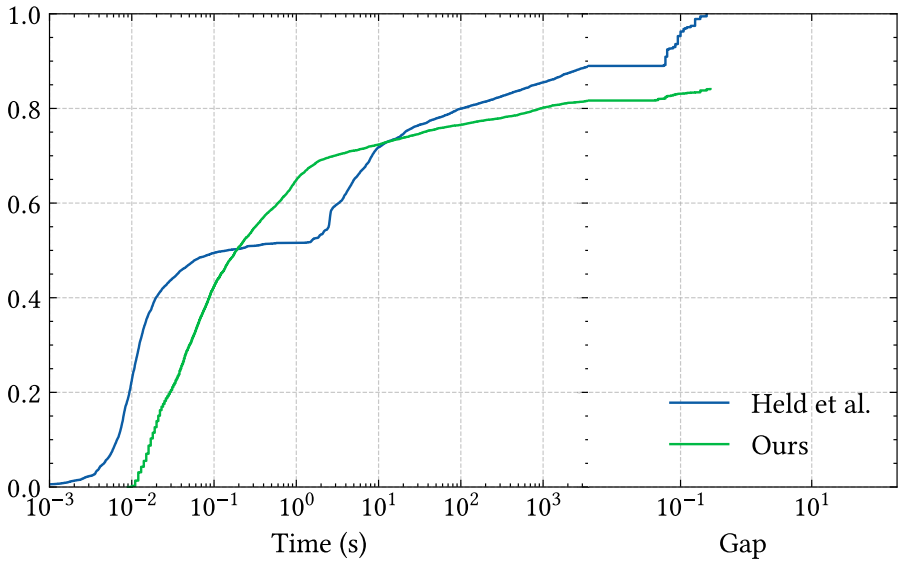

Figure 5.4: Cumulative distribution of instances solved by time and their gap on the MATIDA instance set. In blue, the results of Held et al. [33] code, in green, our results.

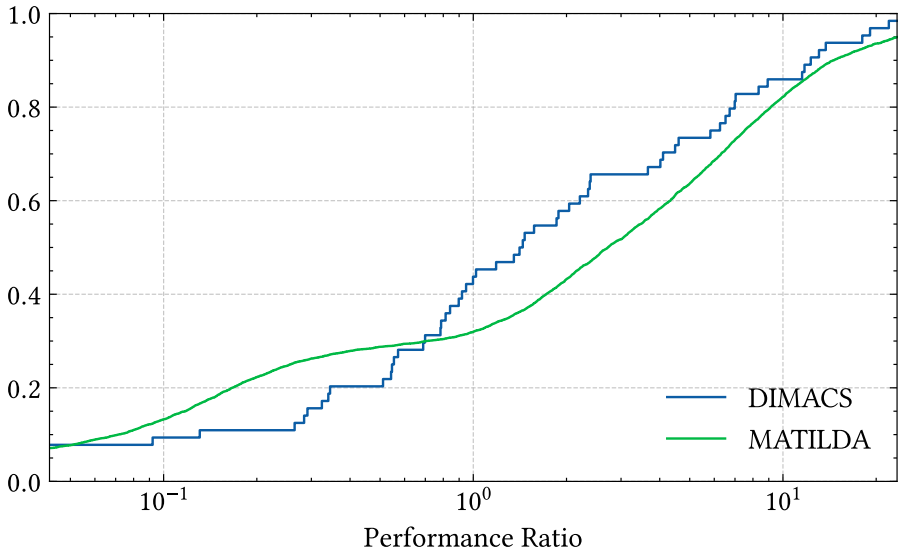

Figure 5.5: Cumulative distribution of proportion of time of our algorithm in relation to what Held et al. [33] algorithm took in each instance both solved. In blue, the DIMACS instance set, and in green, the MATILDA instance set.

[33], with over 10% being solved ten times faster. This performance differs from our results on the DIMACS instance set, where we solved 45% of instances faster, although with fewer cases showing such a significant disparity.

Chapter 6

Conclusions

Graph coloring, a foundational problem in combinatorial optimization, poses significant challenges owing to its computational complexity and broad applicability. This dissertation addresses these challenges through an advanced Integer Linear Programming (ILP) approach, integrating a branch-and-price framework with a branch-and-reduce methodology for its pricing problem.

We investigate the impact of employing a novel method for solving the maximum weighted stable set problem proposed by Xiao et al. [87] on the performance of graph coloring algorithms. Our analysis reveals that certain rules and reductions from the original framework are unnecessary or counterproductive in this context, as the pricing problems encountered here are less complex than the general MWIS instances addressed by Xiao et al. [87].

Our work is benchmarked against existing literature, particularly the code from Held et al. [33]. The main challenges the current state of our work endures are:

1. **Root Node Closure Difficulty:** Our algorithm struggles to close the root node in many cases, particularly where Held's succeeds (e.g., $9/11$ instances in Table 5.1). This can is attributed to a lack of pricing heuristics, which Held's implementation leverages to tighten lower bounds efficiently. This leads to slower improvements in the lower bound, which in turn affects the overall performance on several instances. Implementing fast heuristics with an efficient local search, much like Held et al. [33] has done, can help us conquer this weakness.

2. **Initialization Overhead:** Our approach incurs higher initialization costs with Gurobi, impacting performance on very small instances (e.g., Register Allocation), even though both use the same solver version. This is an issue which will require some extensive testing to pin down the reasons, but it also an important topic for future work for us.

3. **Lower Bound Limitations:** Most of the cases where our method fails to solve an instance is due to its inability to raise lower bounds. This shortfall leads to either longer solution times or the inability to match Held's final upper bounds in certain cases. We believe works such as the one by Morrison et al. [62] can help us overcome this challenge.

4. **Performance on Easy Instances:** Held's algorithm dominates on instances solvable via fast heuristics (e.g., clique-based lower bounds), solving $8/15$ instances in Table 5.2 in less than 0.03 seconds, where our approach lags. We believe the addition of such heuristics with our robust separation and reduction techniques can be of great value for us.

That being said, it is our understaing that there are some important advantages to our method:

1. **Harder Instances:** On computationally intensive instances (e.g., $\mathrm{DSJR500.5}$, $\mathrm{r1000.5}$), our algorithm closes gaps faster or solves instances Held's cannot. For example, it solves $\mathrm{r1000.5}$ exclusively and reduces runtime by 10 times on $\mathrm{DSJR500.5}$.

2. **Effective Reduction Rules:** For Register Allocation instances, our separation and reduction rules alone suffice to close gaps without invoking the LP solver, demonstrating robustness in structured, application-driven graphs.

3. **Upper Bound Quality:** Achieves better upper bounds in several cases (e.g., Optical Network Design instances), even when failing to close the instance. This demonstrate the importance of our heuristics.

4. **Balanced Performance:** While slower on trivial instances, our algorithm shows competitive or better results on mid-to-large instances requiring more than 1 second runtime (e.g., $11/21$ such instances in DIMACS dataset were faster).

The identified challenges provide tangible directions for future enhancements. Once these areas are addressed, the algorithm's performance could improve even further and bridge the performance gap.

In summary, this work advances the state-of-the-art in graph coloring by refining and extending Integer Linear Programming techniques. By integrating branch-and-price with branch-and-reduce methodologies and critically adapting the MWIS framework of Xiao et al. [87], we demonstrate a computationally efficient and robust approach capable of solving instances which other algorithms might struggle. While our results validate the competitiveness of the proposed algorithm, the outlined future directions—such as heuristic enhancements and tighter bounds for small-chromatic graphs—hold promise for an even better result. This dissertation not only deepens the understanding of graph coloring and show how enumerating more than one column at each iteration of process can be beneficial for such problem.

Bibliography

[1] T. Akiba and Y. Iwata, "Branch-and-reduce exponential/fpt algorithms in practice: A case study of vertex cover", *Theoretical Computer Science*, vol. 609, 2016. DOI: 10.1016/j.tcs.2015.09.023.

[2] K. I. Appel, *Every planar map is four colorable* (AMS ebook collection), Online-Ausg., W. Haken, Ed. American Mathematical Society, 2012, 1741 pp.

[3] R. Baldacci, S. Coniglio, J.-F. Cordeau, and F. Furini, "A numerically exact algorithm for the bin-packing problem", *INFORMS Journal on Computing*, vol. 36, no. 1, 2024. DOI: 10.1287/ijoc.2022.0257.

[4] P. Berman and T. Fujito, "On approximation properties of the independent set problem for degree 3 graphs", in *Algorithms and Data Structures*. Springer Berlin Heidelberg, 1995. DOI: 10.1007/3-540-60220-8_84.

[5] I. Blöchliger and N. Zufferey, "A graph coloring heuristic using partial solutions and a reactive tabu scheme", *Computers and Operations Research*, vol. 35, no. 3, 2008. DOI: 10.1016/j.cor.2006.05.014.

[6] N. Bouhmala and O.-C. Granmo, "Solving graph coloring problems using learning automata", in *Evolutionary Computation in Combinatorial Optimization*. Springer Berlin Heidelberg, 2008. DOI: 10.1007/978-3-540-78604-7_24.

[7] D. Brélaz, "New methods to color the vertices of a graph", *Communications of the ACM*, vol. 22, no. 4, 1979. DOI: 10.1145/359094.359101.

[8] E. K. Burke, B. McCollum, A. Meisels, S. Petrovic, and R. Qu, "A graph-based hyper-heuristic for educational timetabling problems", *European Journal of Operational Research*, vol. 176, no. 1, 2007. DOI: 10.1016/j.ejor.2005.08.012.

[9] J. Byskov, "Enumerating maximal independent sets with applications to graph colouring", *Operations Research Letters*, vol. 32, no. 6, 2004. DOI: 10.1016/j.orl.2004.03.002.

[10] M. Campêlo, V. Campos, and R. Corrêa, "On the asymmetric representatives formulation for the vertex coloring problem", *Discrete Applied Mathematics*, vol. 156, no. 7, 2008. DOI: 10.1016/j.dam.2007.05.058.

[11] A. Caprara, L. Kroon, M. Monaci, M. Peeters, and P. Toth, "Chapter 3 passenger railway optimization", in *Transportation*. Elsevier, 2007. DOI: 10.1016/s0927-0507(06)14003-7.

[12] F. C. Chow and J. L. Hennessy, "The priority-based coloring approach to register allocation", *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 4, 1990. DOI: 10.1145/88616.88621.

[13] V. Chvátal, *Linear programming*, 17e print. New York, NY: Freeman, 2003.

[14] G. B. Dantzig, "Reminiscences about the origins of linear programming", *Operations Research Letters*, vol. 1, no. 2, 1982. DOI: 10.1016/0167-6377(82)90043-8.

[15] C. Desrosiers, P. Galinier, and A. Hertz, "Efficient algorithms for finding critical subgraphs", *Discrete Applied Mathematics*, vol. 156, no. 2, 2008. DOI: 10.1016/j.dam.2006.07.019.

[16] R. Dorne and J.-K. Hao, "Tabu search for graph coloring, t-colorings and set t-colorings", in *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*. Springer US, 1999. DOI: 10.1007/978-1-4615-5775-3_6.

[17] D. Eppstein, "Small maximal independent sets and faster exact graph coloring", in *Graph Algorithms and Applications 4*, World Scientific, 2006. DOI: 10.1142/9789812773296_0006.

[18] A. A. Farley, "A note on bounding a class of linear programming problems, including cutting stock problems", *Operations Research*, vol. 38, no. 5, 1990. DOI: 10.1287/opre.38.5.922.

[19] D. Feillet, "A tutorial on column generation and branch-and-price for vehicle routing problems", *4OR*, vol. 8, no. 4, 2010. DOI: 10.1007/s10288-010-0130-z.

[20] H. Fernau, "Parameterized algorithms for d-hitting set: The weighted case", *Theoretical Computer Science*, vol. 411, no. 16–18, 2010. DOI: 10.1016/j.tcs.2010.01.001.

[21] B. A. Foster and D. M. Ryan, "An integer programming approach to the vehicle scheduling problem", *Journal of the Operational Research Society*, vol. 27, no. 2, 1976. DOI: 10.1057/jors.1976.63.

[22] F. Furini, V. Gabrel, and I.-C. Ternier, "Lower bounding techniques for dsatur-based branch and bound", *Electronic Notes in Discrete Mathematics*, vol. 52, 2016. DOI: 10.1016/J.ENDM.2016.03.020.

[23] F. Furini, V. Gabrel, and I.-c. Ternier, "An improved dsatur-based branch-and-bound algorithm for the vertex coloring problem", *Networks*, vol. 69, no. 1, 2016. DOI: 10.1002/net.21716.

[24] P. Galinier and J.-K. Hao, *Journal of Combinatorial Optimization*, vol. 3, no. 4, 1999. DOI: 10.1023/a:1009823419804.

[25] P. Galinier and A. Hertz, "A survey of local search methods for graph coloring", *Computers Operations Research*, vol. 33, no. 9, 2006. DOI: 10.1016/j.cor.2005.07.028.

[26] P. Galinier, A. Hertz, and N. Zufferey, "An adaptive memory algorithm for the k-coloring problem", *Discrete Applied Mathematics*, vol. 156, no. 2, 2008. DOI: 10.1016/j.dam.2006.07.017.

[27] M. Gamache, A. Hertz, and J. O. Ouellet, "A graph coloring model for a feasibility problem in monthly crew scheduling with preferential bidding", *Computers Operations Research*, vol. 34, no. 8, 2007. DOI: 10.1016/j.cor.2005.09.010.

[28] A. Gamst, "Some lower bounds for a class of frequency assignment problems", *IEEE Transactions on Vehicular Technology*, vol. 35, no. 1, 1986. DOI: 10.1109/t-vt.1986.24063.

[29] M. Garey and D. Johnson, "The complexity of near-optimal graph coloring", *Journal of the ACM*, vol. 23, no. 1, 1976. DOI: 10.1145/321921.321926.

[30] M. Grötschel, L. Lovász, and A. Schrijver, "The ellipsoid method and its consequences in combinatorial optimization", *Combinatorica*, vol. 1, no. 2, 1981. DOI: 10.1007/bf02579273.

[31] L. Gurobi Optimization, *Gurobi optimizer reference manual*, 2024.

[32] J.-K. Hao and Q. Wu, "Improving the extraction and expansion method for large graph coloring", *Discrete Applied Mathematics*, vol. 160, no. 16–17, 2012. DOI: 10.1016/j.dam.2012.06.007.

[33] S. Held, W. Cook, and E. Sewell, "Maximum-weight stable sets and safe lower bounds for graph coloring", *Mathematical Programming Computation*, vol. 4, no. 4, 2012. DOI: 10.1007/s12532-012-0042-3.

[34] A. Hertz and D. De Werra, "Using tabu search techniques for graph coloring", *Computing*, vol. 39, no. 4, 1987. DOI: 10.1007/bf02239976.

[35] A. Hertz, M. Plumettaz, and N. Zufferey, "Variable space search for graph coloring", *Discrete Applied Mathematics*, vol. 156, no. 13, 2008. DOI: 10.1016/j.dam.2008.03.022.

[36] M. J. H. Heule, A. Karahalios, and W.-J. van Hoeve, "From cliques to colorings and back again", en, in *28th International Conference on Principles and Practice of Constraint Programming*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 235, 2022. DOI: 10.4230/LIPICS.CP.2022.26.

[37] W.-J. van Hoeve, "Graph coloring with decision diagrams", *Mathematical Programming*, vol. 192, no. 1-2, 2021. DOI: 10.1007/s10107-021-01662-x.

[38] R. v. d. Hulst, "A branch-price-and-cut algorithm for graph coloring", M.S. thesis, University of Twente, 2021.

[39] A. Jabrayilov and P. Mutzel, "New integer linear programming models for the vertex coloring problem", in *LATIN 2018: Theoretical Informatics*. Springer International Publishing, 2018. DOI: 10.1007/978-3-319-77404-6_47.

[40] A. Jabrayilov and P. Mutzel, "Strengthened partial-ordering based ilp models for the vertex coloring problem", 2022.

[41] Y. Jin, J.-P. Hamiez, and J.-K. Hao, "Algorithms for the minimum sum coloring problem: A review", *Artificial Intelligence Review*, vol. 47, no. 3, 2016. DOI: 10.1007/s10462-016-9485-7.

[42] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, "Optimization by simulated annealing: An experimental evaluation; part ii, graph coloring and number partitioning", *Operations Research*, vol. 39, no. 3, 1991. DOI: 10.1287/opre.39.3.378.

[43] N. Karmarkar, "A new polynomial-time algorithm for linear programming", *Combinatorica*, vol. 4, no. 4, 1984. DOI: 10.1007/bf02579150.

[44] D. E. Knuth, "The sandwich theorem", *The Electronic Journal of Combinatorics*, vol. 1, no. 1, 1994. DOI: 10.37236/1193.

[45] S. Lamm, C. Schulz, D. Strash, R. Williger, and H. Zhang, "Exactly solving the maximum weight independent set problem on large real-world graphs", in *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, 2019. DOI: 10.1137/1.9781611975499.12.

[46] F. Leighton, "A graph coloring algorithm for large scheduling problems.", *Journal of Research of the National Bureau of Standards*, vol. 84, no. 6, 1979. DOI: 10.6028/JRES.084.024.

[47] F. T. Leighton, "Finite common coverings of graphs", *Journal of Combinatorial Theory, Series B*, vol. 33, no. 3, 1982. DOI: 10.1016/0095-8956(82)90042-9.

[48] R. M. R. Lewis, *Guide to Graph Colouring Algorithms and Applications, Algorithms and Applications*. Springer, 2015.

[49] A. M. D. Lima and R. Carmo, "Exact algorithms for the graph coloring problem", *Revista de Informática Teórica e Aplicada*, vol. 25, no. 4, 2018. DOI: 10.22456/2175-2745.80721.

[50] L. Lovasz, "On the shannon capacity of a graph", *IEEE Transactions on Information Theory*, vol. 25, no. 1, 1979. DOI: 10.1109/tit.1979.1055985.

[51] L. Lovász, "On the ratio of optimal integral and fractional covers", *Discrete Mathematics*, vol. 13, no. 4, 1975. DOI: 10.1016/0012-365x(75)90058-8.

[52] Z. Lü and J.-K. Hao, "A memetic algorithm for graph coloring", *European Journal of Operational Research*, vol. 203, no. 1, 2010. DOI: 10.1016/j.ejor.2009.07.016.

[53] E. Malaguti, M. Monaci, and P. Toth, "An exact approach for the vertex coloring problem", *Discrete Optimization*, vol. 8, no. 2, 2011. DOI: 10.1016/j.disopt.2010.07.005.

[54] E. Malaguti and P. Toth, "A survey on vertex coloring problems", *International Transactions in Operational Research*, vol. 17, no. 1, 2009. DOI: 10.1111/j.1475-3995.2009.00696.x.

[55] F. D. Malliaros, C. Giatsidis, A. N. Papadopoulos, and M. Vazirgiannis, "The core decomposition of networks: Theory, algorithms and applications", *The VLDB Journal*, vol. 29, no. 1, 2019. DOI: 10.1007/s00778-019-00587-4.

[56] C. Mccreesh, P. Prosser, K. Simpson, and J. Trimble, "On maximum weight clique algorithms, and how they are evaluated", in *Lecture Notes in Computer Science*. Springer International Publishing, 2017. DOI: 10.1007/978-3-319-66158-2_14.

[57] A. Mehrotra and M. A. Trick, "A column generation approach for graph coloring", *INFORMS Journal on Computing*, vol. 8, no. 4, 1996. DOI: 10.1287/ijoc.8.4.344.

[58] I. Méndez-Díaz and P. Zabala, "A branch-and-cut algorithm for graph coloring", *Discrete Applied Mathematics*, vol. 154, no. 5, 2006. DOI: 10.1016/j.dam.2005.05.022.

[59] I. Méndez-Díaz and P. Zabala, "A cutting plane algorithm for graph coloring", *Discrete Applied Mathematics*, vol. 156, no. 2, 2008. DOI: 10.1016/j.dam.2006.07.010.

[60] L. Moalic and A. Gondran, "Variations on memetic algorithms for graph coloring problems", *Journal of Heuristics*, vol. 24, no. 1, 2017. DOI: 10.1007/s10732-017-9354-9.

[61] D. Morrison, E. Sewell, and S. Jacobson, "Solving the pricing problem in a branch-and-price algorithm for graph coloring using zero-suppressed binary decision diagrams", *INFORMS Journal on Computing*, vol. 28, no. 1, 2016. DOI: 10.1287/ijoc.2015.0667.

[62] D. R. Morrison, J. J. Sauppe, E. C. Sewell, and S. H. Jacobson, "A wide branching strategy for the graph coloring problem", *INFORMS Journal on Computing*, vol. 26, no. 4, 2014. DOI: 10.1287/IJOC.2014.0593.

[63] Y. Nesterov and A. Nemirovskii, *Interior-Point Polynomial Algorithms in Convex Programming*. Society for Industrial and Applied Mathematics, 1994. DOI: 10.1137/1.9781611970791.

[64] A. Pessoa, R. Sadykov, E. Uchoa, and F. Vanderbeck, "A generic exact solver for vehicle routing and related problems", *Mathematical Programming*, vol. 183, no. 1–2, 2020. DOI: 10.1007/s10107-020-01523-z.

[65] D. Porumbel, J.-K. Hao, and P. Kuntz, "A search space "cartography" for guiding graph coloring heuristics", *Computers and Operations Research*, vol. 37, no. 4, 2010. DOI: 10.1016/j.cor.2009.06.024.

[66] D. C. Porumbel, J.-K. Hao, and P. Kuntz, "Position-guided tabu search algorithm for the graph coloring problem", in *Learning and Intelligent Optimization*. Springer Berlin Heidelberg, 2009. DOI: 10.1007/978-3-642-11169-3_11.

[67] L. Rabiner, "Combinatorial optimization:algorithms and complexity", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 32, no. 6, 1984. DOI: 10.1109/tassp.1984.1164450.

[68] P. San Segundo, "A new dsatur-based algorithm for exact vertex coloring", *Computers and Operations Research*, vol. 39, no. 7, 2012. DOI: 10.1016/j.cor.2011.10.008.

[69] E. R. Scheinerman, *Fractional Graph Theory, A Rational Approach to the Theory of Graphs* (Dover Books on Mathematics), D. H. Ullman, Ed. Newburyport: Dover Publications, 2013.

[70] S. B. Seidman, "Network structure and minimum degree", *Social Networks*, vol. 5, no. 3, 1983. DOI: 10.1016/0378-8733(83)90028-x.

[71] E. C. Sewel, "An improved algorithm for exact graph coloring", *DIMACS Series in Discrete Mathmatics and Theoretical Computer Science*, 1996. DOI: 10.1090/dimacs/026/17.

[72] H. Shachnai and M. Zehavi, "A multivariate framework for weighted fpt algorithms", *Journal of Computer and System Sciences*, vol. 89, 2017. DOI: 10.1016/j.jcss.2017.05.003.

[73] C. Silva, A. Aguiar, P. M. V. Lima, and I. Dutra, "Mapping graph coloring to quantum annealing", *Quantum Machine Intelligence*, vol. 2, no. 2, 2020. DOI: 10.1007/s42484-020-00028-4.

[74] R. F. F. da Silva and R. C. S. Schouery, "A branch-and-cut-and-price algorithm for cutting stock and related problems", 2023. DOI: 10.48550/ARXIV.2308.03595.

[75] D. Smith, S. Hurley, and S. Thiel, "Improving heuristics for the frequency assignment problem", *European Journal of Operational Research*, vol. 107, no. 1, 1998. DOI: 10.1016/s0377-2217(98)80006-4.

[76] K. Smith-Miles and S. Bowly, "Generating new test instances by evolving in instance space", *Computers and Operations Research*, vol. 63, 2015. DOI: 10.1016/j.cor.2015.04.022.

[77] K. Smith-Miles and L. Lopes, "Measuring instance difficulty for combinatorial optimization problems", *Computers and Operations Research*, vol. 39, no. 5, 2012. DOI: 10.1016/j.cor.2011.07.006.

[78] W. Sun, "Heuristic algorithms for graph coloring problems", Theses, Université d'Angers, 2018.

[79] I.-C. Ternier, "Exact algorithms for the vertex coloring problem and its generalisations", Ph.D. dissertation, Université Paris sciences et lettres, 2017.

[80] O. Titiloye and A. Crispin, "Quantum annealing of the graph coloring problem", *Discrete Optimization*, vol. 8, no. 2, 2011. DOI: 10.1016/j.disopt.2010.12.001.

[81] A. Van Gelder, "Another look at graph coloring via propositional satisfiability", *Discrete Applied Mathematics*, vol. 156, no. 2, 2008. DOI: 10.1016/j.dam.2006.07.016.

[82] P.-J. Wan, X. Jia, G. Dai, H. Du, and O. Frieder, "Fast and simple approximation algorithms for maximum weighted independent set of links", in *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*. IEEE, 2014. DOI: 10.1109/infocom.2014.6848102.

[83] D. de Werra, "An introduction to timetabling", *European Journal of Operational Research*, vol. 19, no. 2, 1985. DOI: 10.1016/0377-2217(85)90167-5.

[84] D. de Werra, C. Eisenbeis, S. Lelait, and B. Marmol, "On a graph-theoretical model for cyclic register allocation", *Discrete Applied Mathematics*, vol. 93, no. 2–3, 1999. DOI: 10.1016/s0166-218x(99)00105-5.

[85] L. A. Wolsey, *Integer programming*, Second edition. Hoboken, NJ: Wiley, 2021, 1316 pp., Literaturverz. S. 245 - 260.

[86] T.-K. Woo, S. Su, and R. Newman, "Resource allocation in a dynamically partitionable bus network using a graph coloring algorithm", *IEEE Transactions on Communications*, 1991. DOI: 10.1109/26.120165.

[87] M. Xiao, S. Huang, Y. Zhou, and B. Ding, "Efficient reductions and a fast algorithm of maximum weighted independent set", in *Proceedings of the Web Conference 2021*, ser. WWW '21, ACM, 2021. DOI: 10.1145/3442381.3450130.

[88] M. Xiao and H. Nagamochi, "Confining sets and avoiding bottleneck cases: A simple maximum independent set algorithm in degree-3 graphs", *Theoretical Computer Science*, vol. 469, 2013. DOI: 10.1016/j.tcs.2012.09.022.

[89] Y. Zhou, B. Duval, and J.-K. Hao, "Improving probability learning based local search for graph coloring", *Applied Soft Computing*, vol. 65, 2018. DOI: 10.1016/j.asoc.2018.01.027.

[90] N. Zufferey, P. Amstutz, and P. Giaccari, “Graph colouring approaches for a satellite range scheduling problem”, *Journal of Scheduling*, vol. 11, no. 4, 2008. DOI: 10.1007/s10951-008-0066-8.

[91] A. A. Zykov, “Über einige eigenschaften der streckenkomplexe”, Russian, *Mat. Sb., Nov. Ser.*, vol. 24, 1949.