Satisfiability Encodings And User Propagators For The Graph Coloring Problem

Timo Brand

Born 26.11.1999 in Munich, Germany

18.08.2024

Master's Thesis Mathematics

Advisor: Prof. Dr. Held

Second Advisor: Prof. Dr. Mutzel

Forschungsinstitut für Diskrete Mathematik

Mathematisch-Naturwissenschaftliche Fakultät der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Introduction

Graph coloring consists of assigning a minimum number of colors to all vertices of a graph such that no two adjacent vertices are assigned the same color. The graph coloring problem is to find the chromatic number $\chi(G)$, i.e., the smallest number of colors needed for such an assignment. In applications, this can be understood as partitioning items into the smallest number of subsets with compatible elements. Such problems occur, for instance, in timetables, scheduling, or large-scale air traffic flow management [Mar04; Woo69; BB04], register allocation for compiler optimization [Cha82], auction and assignment of broadcast frequencies [Aar+07] or computing Jacobian matrices [GMP05]. As this problem is NP-hard, it is both theoretically challenging and important in practice, so that finding efficient algorithms is of significant interest for real-world applications. Even approximating $\chi(G)$ to within $n^{1-\epsilon}$ for all $\epsilon > 0$ [Zuc06] is NP-hard.

This thesis aims to exactly solve the graph coloring problem for general graphs and discusses several satisfiability-based algorithms to achieve results competitive with methods from the literature.

Structure of the Thesis

The relevant mathematical concepts are introduced in Chapter 1 which additionally surveys related algorithms from the literature. Further, Chapter 2 presents bounding and preprocessing techniques for the graph coloring problem, used to reduce the computational effort of later exact methods. Chapter 3 introduces various satisfiability encodings for graph coloring that use the SAT solver as a black-box, classified as either direct encodings or Zykov-based encodings. An experimental evaluation and comparison with work from the literature is done. Next, Chapter 4 builds upon the previous satisfiability encodings but uses a new interface to the SAT solver to directly interact and influence the solving process. Crucially, this allows pruning the search tree with problem-specific information provided by the user. This and other ideas to potentially improve performance are discussed with their implementation details before analyzing the results in a further experimental evaluation. Finally, Chapter 5 summarizes the results and observations of this thesis and concludes with a discussion of potential future research directions.

Contributions

Besides providing a comprehensive description and survey of all relevant concepts from the literature, this thesis makes the following significant contributions:

- An extensive preprocessing routine for the graph coloring problem is described, including computation of lower and upper bounds as well as reducing the graph with standard rules from the literature. This routine alone solves 50 out of the 137 considered instances, and a detailed performance overview is given.
- A novel preprocessing method to reduce the graph size is introduced, and its effectiveness is compared to standard preprocessing methods.

- Different direct encodings for the graph coloring problem are presented with an efficient and successful implementation that achieves a performance competitive to algorithms of the literature. In particular, the implementation of the partial order encoding solves 93 instances of the DIMACS benchmark set, more than any other method considered. Further, the partial order encoding of this thesis closes the open instance wap07a by computing its chromatic number in less than 3 hours. A new lower bound of 5 is also reported for the instance 1-Insertions_6.
- A description of the Full Encoding based on the Zykov property is given, and a new upper bound on the number of transitivity clauses is proven. The quality of the bound in practice is analyzed. While the evaluation suggests the Full Encoding is not the most suited algorithm for general graphs, it manages to solve the very dense open instance r1000.1c in only 35 minutes.
- The work of Glorian et al. [Glo+19] and their incremental satisfiability algorithm based on the Zykov property is reviewed. As they reported excellent results but cannot repeat the experiments due to lost source code, an implementation aiming to reproduce their results is discussed in detail. Several improvements to their method are proposed, and a faster and more successful checker algorithm is presented. An extensive comparison to the results in the original paper is made, and the different trends in performance are highlighted.
- The hybrid constraint programming/SAT approach of Hébrard and Katsirelos [HK20] is implemented as a pure SAT-based algorithm using the new IPASIR-UP interface from [Faz+23]. Previous and new ideas are discussed for effective pruning during the search, taking advantage of the user propagator features.
- Using further SAT techniques for incremental solving leads to an implementation that supports fully incremental bottom-up solving. As the pruning techniques are stronger when solving in bottom-up, this improves performance and solves 89 instances instead of the 82 solved by the algorithm of [HK20]. This makes the implementation competitive with other state-of-the-art SAT-based algorithms for the graph coloring problem.
- A release of the source code for all algorithms implemented as part of this thesis at `https://github.com/trewes/IncSatGC-release`.

Acknowledgements

First and foremost, I want to thank Prof. Stephan Held for suggesting me such an interesting topic, allowing me so much freedom in my research aims and the helpful supervision. Further thanks are due to Prof. Petra Mutzel for taking on the second correction of this thesis. I also thank the institute for discrete mathematics for providing me access to their computers to complete all my computational experiments. Further thanks are due to Valentin Montmirail, George Katseliros, Fabio Furini, and Katalin Fazekas for the open and helpful discussion of their research.

Finally, I want to express my gratitude to all my friends and family for their continuous support throughout the year. Thank you, Moritz, Thomas, Louis, Jakob, Rania, Daria, Tobias for proofreading my thesis and having made my studies an enjoyable experience.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Preliminaries

This chapter introduces the background and mathematical foundations needed for the rest of this thesis. The graph coloring problem and relevant concepts are defined in Section 1.1. Next, the Conflict-Driven Clause-Learning (CDCL) paradigm used by modern SAT solvers is presented. As this thesis focuses on satisfiability encodings whose decision problem corresponds to the k-coloring problem, satisfiability solvers and their underlying workings play a large role. As such, boolean satisifiability and CDCL are discussed in Section 1.2. The graph coloring problem has been studied extensively, and several effective algorithms exist, some of which are briefly covered in Section 1.3. Since there will be multiple sections with experimental evaluations on different algorithms and encodings, the computational setup including the benchmark instances is presented once in Section 1.4.

1.1 Background

A graph G is a pair (V, E) with the set of vertices V and the set of edges $E \subseteq \{\{u, v\} \mid u, v \in V \text{ and } u \neq v\}$; if the graph is not clear fom the context, $V(G)$ and $E(G)$ are used. As notational shorthand, $n \coloneqq |V|$ and $m \coloneqq |E|$. One should think of n and m as the number of vertices and number of edges, respectively, unless stated otherwise or clear from the context. The graphs of this thesis are undirected and simple, that is, $\{u, v\}$ is the same egde as $\{v, u\}$ and there are no loops $\{u, u\}$ in the graph. Further, all graphs are finite, i.e., $n, m < \infty$. The density $d(G) \coloneqq 2 \cdot m/(n \cdot (n-1)) \in [0, 1]$ is a measure of how many edges are present in the graph, and the complement graph $\overline{G}$ is given on the same vertex set V but with edges $\overline{E} \coloneqq \{\{u, v\} \mid \{u, v\} \notin E\}$, with its number of edges referred to by $\overline{m} \coloneqq \overline{E}$. Given vertices u, v such that $\{u, v\} \in E$, u is said to be adjacent to v, and two edges e, e' are called incident if $e \cap e' \neq \emptyset$. The neighborhood of a vertex is defined as $N_G(v) = \{u \mid \{u, v\} \in E\}$ and shortened to $N(v)$ if the graph is clear from the context. The degree of a vertex is $|N(v)|$ and $\Delta(G)$ refers to the maximum degree of any vertex in G.

The graph coloring problem on a graph G consists of finding the minimal k for which a k-coloring exists, that is, a map $f : V \mapsto \{1, \ldots, k\}$ such that $f(u) \neq f(v)$ for all edges $\{u, v\} \in E$. The numbers $1, \ldots, k$ are called colors and f a color assignment or k-coloring. The smallest such k is denoted as $\chi(G)$ and called the coloring number, the decision problem of whether a graph G admits a k-coloring was shown to be NP-complete by Karp [Kar72]. Any coloring further provides a partition of the vertex set V into k color-classes $\{I_1, \ldots, I_k\}$, where each $I_j \coloneqq \{v \in V \mid f(v) = j\}$ is an independent set. An independent or stable set is a set of vertices that are all pairwise non-adjacent, so it is straightforward to see the color classes must be stable sets from the definition of a coloring. The number $\alpha(G)$ is the size of the largest stable set in a graph G and is called the stability number. Computing an independent set of size $\alpha(G)$ is the maximum independent set (MIS) problem. A complementary concept is that of cliques, a vertex set $C \subseteq V$ such that u and v are adjacent for all $u, v \in C$. The number $\omega(G)$ is likewise the size of the

largest clique in G and is called the clique number. Any clique forms a stable set in the complement graph and vice versa, and both decision problems, i.e., deciding there exists a stable set or a clique of a certain size, are again NP-complete [Kar72].

An important data structure used for algorithms throughout this thesis is that of a bitset. It is used to represent a subset of integers from a certain range $\{0, \dots, n-1\}$ by a sequence of bits with value 0 or 1. One can think of it as a string of length n, where a value of 1 at index i represents that i is part of the subset. This requires $O(n)$ space for possibly a lot fewer elements, but since each element is compactly represented by a single bit, this can be more compact than storing an integer for each element of the subset. This data structure allows for efficient computation of intersection and union for two subsets, as this can be done with simple and fast bitwise "OR" and "AND" operations. The main benefit of bitsets is their bit-parallelism: the logical operations are performed on 32 or 64 bits at a time, depending on the computer architecture, leading to a complexity of $O(n/64) = O(n)$. For the sizes considered in this thesis, the complexity is often near-constant, so it will be practical to think in terms of bitwise operations when discussing algorithms using them. Their compact size and bit-parallelism make them a great tool for efficiently working with subsets of a range of integers; in the application of this work, they will frequently be used to represent subsets of colors or vertices.

1.2 SAT Solving and CDCL

The algorithms for graph coloring in this thesis are based around building an encoding of the k-coloring problem and using a satisfiability solver to find either a satisfying model corresponding to a k-coloring, or determine the unsatisfiability of the instance. Most modern SAT solvers are based on the CDCL algorithm, and as the user propagators presented in Chapter 4 tightly integrate into the solving process, the algorithm is briefly explained here. First, the necessary background and notation for Boolean satisfiability is given.

1.2.1 Boolean Satisfiability

In SAT solving, problems are expressed with Boolean variables that can take a value in $\{0, 1\}$. A literal l is either a variable x or its negation $\neg x$, also written as $\overline{x}$. A clause $C = (l_1 \vee \ldots \vee l_k)$ is a disjunction of literals, and a formula $\mathcal{F} = C_1 \wedge \ldots \wedge C_m$ is built of the conjunction of clauses. Formulas of this form are said to be in Conjunctive Normal Form (CNF), which is the usual format that most SAT solvers work with. They can also be written as a set of clauses $\mathcal{F} = \{C_1, \dots, C_m\}$ where each clause is represented as a set of literals $C = \{l_1 \dots l_k\}$. An assignment α for a CNF formula $\mathcal{F}$ with variables $\text{vars}(\mathcal{F}) = \{x_1, \dots, x_n\}$ is a mapping of a subset X of the variables to $\{0, 1\}$. For a literal $\overline{x}_i = \text{True}$, one can also write $x_i = \text{False}$. The assignment can also be characterized by the literals set to true: $\alpha = \{l_{i_1}, \dots, l_{i_r}\}$. If α maps only a proper subset of the variables, it is called a partial assignment. Given an assignment α, a literal l is satisfied or true if either $l = x_i$ and $\alpha(x) = 1$ resp. $l = \overline{x}_i$ and $\alpha(x) = 0$. A clause is satisfied if any of its literals is true, and a formula is satisfied if all of its clauses are satisfied. An assignment satisfying a formula is also called a model of the formula and denoted λ, and it is the goal of a SAT solver to find such a model (SAT) or determine that the instance is unsatisfiable (UNSAT). To reason about an assignment or model, let $l \in \alpha$ or $l \in \lambda$ mean that the literal l is satisfied or set to true in that assignment, similarly $\{l_1, \dots, l_k\} \subseteq \lambda$. Further, let $\alpha \vDash C$ denote that clause C is satisfied under the partial assignment and $\alpha \nvDash C$ mean that C is violated or unsatisfied under α.

Example 1.2.1. Consider the following formula on the variables $\{a, b, c, d\}$.

$$\mathcal{F} = (a \vee \overline{b} \vee c) \wedge (\overline{a} \vee b) \wedge (\overline{c} \vee d) \wedge (a \vee \overline{d}) \tag{1.1}$$

The assignment $\alpha = \{a \mapsto 1, b \mapsto 1, c \mapsto 1, d \mapsto 0\}$ resp. $\alpha = \{a, b, c, \overline{d}\}$ is not a model of F since $\alpha \not\models (a \vee \overline{d})$, but $\{a, b, c, d\}$ is a model since every clause is satisfied.

1.2.2 Conflict-Driven Clause-Learning

One of the main reasons that satisfiability solvers are so effective in practice is the CDCL paradigm [MS99] that most modern solvers are based on. This is mainly because with CDCL solvers can learn from conflicts encountered during the search and backtrack non-chronologically. The learned information can help avoid branches of the search tree that fail for the same reason as when the conflict was learnt, thus providing significant opportunities to prune the search tree by recognizing the structure of the problem. Backtracking non-chronologically by more than one level further allows skipping large parts of the search tree. The CDCL algorithm has three major parts:

1. Unit propagation
2. Conflict analysis
3. Backtracking

Algorithm 1.1: CDCL

Input: Formula $\mathcal{F}$ in CNF
Output: Either SAT and a model α, or UNSAT

$\alpha \leftarrow \emptyset$ `// Start with empty assignment`
$dl \leftarrow 0$ `// Track current decision level`
while True **do**
  $\alpha \leftarrow$ UnitPropagation$(\mathcal{F}, \alpha)$
  if $\alpha \not\models C$ *for some clause* C *of* $\mathcal{F}$ **then**
    if $dl == 0$ **then**
      return UNSAT
    $dl', C' \leftarrow$ ConflictAnalysis$(\mathcal{F}, \alpha)$
    $\mathcal{F} \leftarrow \mathcal{F} \wedge C'$ `// Add the learned clause to the formula`
    Backtrack$(\mathcal{F}, \alpha, dl')$ `// Reset assignments made after level` dl'
  else
    if α *is a complete assignment* **then**
      return SAT, α
    $dl \leftarrow dl + 1$
    $l \leftarrow$ PickDecisionVariable$(\mathcal{F}, \alpha)$
    $\alpha \leftarrow \alpha \cup l$ `// Decide on an unassigned literal` l

On a high level, CDCL is a backtrack algorithm with the ability to learn from mistakes, pseudocode for the algorithm is given in Algorithm 1.1. Beginning with an empty assignment, a variable is chosen and set to either true or false. This decision creates two subproblems which are in the next level of the search tree. This level is also called the decision level, and for a literal assigned at that level, this is also called the decision level of that literal. The decisions that lead to a node in the search tree affect other literals. For example, if the formula contains the clause $(a \wedge b)$ and a was set to false as a decision, the literal b has to be true to satisfy the clause. This is called unit propagation: in the current

subproblem, the clause $(a \wedge b)$ became the unit clause (b) with only a single literal, as a was set to false and removed from the clause. This can also be understood as the implication $\overline{a} \implies b$. Unit propagation is repeated in the current subproblem until no further unit clause exists. This technique is not unique to CDCL but has already been an important step in previous satisfiability algorithms such as the Davis–Putnam–Logemann–Loveland (DPLL) algorithm [DLL62].

After all propagations have been made, a new unassigned variable is chosen as a decision, creating two more subproblems on the next decision level. This continues until all variables have been assigned and SAT is returned, or an empty clause is encountered, either due to a decision or unit propagation. In this case, conflict analysis is performed to learn from the conflict. This is done by considering the implication graph, which broadly represents which decisions lead to which assignments, and using it to identify the decisions that caused the conflict. The clause produced this way is potentially more useful than just the knowledge that the current assignment is bad. By identifying the reason for the conflict and adding its clause to the formula, this conflict can be avoided in other parts of the search tree and potentially prune more than just the current node. The terms conflict clause, learned clause, and reason clause are used interchangeably.

After finding a conflict and adding its learned clause, the search backtracks to a previous level with no unsatisfied clauses. Without the stronger conflict clause, this would be just the previous level as the partial assignment there had no conflict. With the potentially stronger conflict clause, one can backtrack non-chronologically, i.e., by more than one level. Say that the clause $(a \wedge b \wedge \overline{c})$ has been identified as a reason clause on decision level 5, i.e., it is not satisfied in the current assignment. Further, assume that a and b were assigned to false on decision level 1 and 2, respectively, and c was assigned to true at level 5, the level of the conflict. Instead of simply backtracking to level 4, one can backtrack to level 2. Undoing the assignment $c = \text{True}$, the variables a and b are still set to false, and the reason clause is a unit clause of just $(\overline{c})$ even on the earlier decision level. This unit clause propagates $c = \text{False}$ on level 2 and thus also for all child subproblems of that node, instead of propagating this fact for level 4 if only a chronological backtrack was made. Formally, the level is identified as the second-largest decision level of the literals in the conflict clause. After back-jumping to that level, all but one literal are still falsified; the unassigned literal is set to true by unit propagation. Backtracking non-chronologically allows one to prune parts of the search space, as the propagation from the learnt clause is made at an earlier decision level instead of further down in the tree.

Other major components that are omitted here include smart decision heuristics, i.e., which variable is selected as the decision and what data structures are used to enable fast unit propagation. Further, methods to forget some learned clauses to reduce memory and minimizing the conflict clauses to make it stronger are also important. The review here is by no means exhaustive, and the interested reader is directed towards the concise description in [KLW22] or the thorough chapter of [MLM21]; good visual examples including the implication graph can be found in the online lecture notes of [Jun20]. The pseudocode of the CDCL algorithm given in Algorithm 1.1 was adapted from [MLM21; Jun20]. An example execution of CDCL is given next.

Example 1.2.2. Consider the following formula on variables $\{a, b, c, d\}$:

$$\mathcal{F} = (a \vee b \vee c) \wedge (a \vee b \vee \overline{c}) \wedge (\overline{b} \vee d) \wedge (a \vee \overline{b} \vee \overline{d}) \tag{1.2}$$

There are no unit clauses and thus no unit propagation at the root. After the first decision $a = \text{False}$ is made, and after removing clauses that contain $\overline{a}$ and removing a from clauses

that contain it, the formula again has no unit clause and no unit propagation is done. The formula on decision level 1 now looks as follows:

$$(b \vee c) \wedge (b \vee \overline{c}) \wedge (\overline{b} \vee d) \wedge (\overline{b} \vee \overline{d}) \tag{1.3}$$

Assume the next decision is $b = \text{False}$, again satisfied clauses are removed and b is removed from clauses that contain it. The formula now becomes unsatisfiable: $(c) \wedge (\overline{c})$. Conflict analysis produces the reason clause $(a \vee b)$ since the two decisions lead to the conflict. The algorithm backtracks to level 1 as it is the first decision level where the conflict clause becomes unit. Unit propagation on Equation (1.3) and the learnt clause implies that $b = \text{True}$, leading to another unsatisfiable formula: $(d) \wedge (\overline{d})$. As $a = \text{False}$ is the only decision, the unit clause (a) is learnt, so that after unit propagation, the formula at decision level 0 becomes $(\overline{b} \vee d)$. Deciding on either of $b = \text{False}$ or $d = \text{True}$ satisfies the formula, regardless of the values of the other variables like c.

Note that while the formula is satisfied, the termination criterion of a complete assignment as given in Algorithm 1.1 is not yet met. This is because modern SAT solvers use lazy data structure where clause states are not always maintained accurately, so the termination criterion is whether all variables are assigned [MLM21].

Two more important features of modern satisfiability solvers are incremental solving and assumptions, which go hand in hand. Often, one is interested in not only solving a single instance, but a series of closely related encodings. Instead of rebuilding the encoding and solving it from scratch each time, incremental solving allows reusing information from the previous solving process. This can include variable ordering, polarity cache, clause deletion strategies, learned clauses, and more [ALS13].

Adding new clauses to a problem in-between solving calls is straightforward and follows the same mechanism as learnt clauses. Removing clauses is not as simple and requires new variables to serve as so-called activation literals. For example, instead of the clause $(a \vee b)$, the clause $(a \vee b \vee c)$ is added with c as activation literal. When c is false, the clause $(a \vee b)$ is active, otherwise c is true, and the satisfied clause can be dropped. To enforce certain values for the activation literal, assumptions are used. In particular, given literals $l_1, \ldots, l_n$ that are required to be true, the formula $\mathcal{F} \wedge l_1 \wedge \ldots \wedge l_n$ needs to be solved. Instead of adding the literals as unit clauses, the assumptions are first selected as decision variables and set to the desired value, as this allows keeping learnt clauses when solving the instance with other assumptions. Using assumptions for activation literals will be important in Chapter 4 where externally provided clauses later need to be deactivated again.

1.2.3 Optimization Strategy

As SAT solvers only give an answer to decision problems, a word has to be said about the optimization strategy when using them to solve the graph coloring problem. Given that the encodings seen later represent whether a k-coloring exists for a graph, one can build and solve the encoding for $k \in \{1, \ldots, n-1\}$, where $n = |V(G)|$ does not need to be checked as it is trivially colorable. The value k such that the encoding is satisfiable for k but not for $k-1$ is the chromatic number of the graph. This value can be determined by different methods of traversing the range $[1, |V(G)|]$, most prominently bottom-up or top-down search. The values are iterated in order $k = 1, \ldots, n-1$ resp. $k = n-1, \ldots, 1$ and in bottom-up search, as used by Van Gelder [Van08], the chromatic number is the first k for which the encoding becomes satisfiable. In top-down search, the chromatic number is k if $k-1$ is the first time the encoding becomes unsatisfiable. This can be improved with a good initial lower bound lb and upper bound ub on $\chi(G)$; only values in the range

$[lb, ub)$ need to be checked. Algorithms to compute such bounds are presented in Chapter 2. Another method to reduce the number of decision problems to be solved is to perform binary search on the interval. Usually, there are only two challenging instances to decide: that $\chi(G)$ is satisfiable and $\chi(G) - 1$ is not, and k-coloring problems with k far from that are much easier. Further, solving a monotone sequence of k-coloring problems as with bottom-up or top-down search makes it simpler to take advantage of incremental solving methods, as the following problems are more closely related. For these two reasons, binary search was not investigated as an optimization technique in this thesis.

1.3 Related Work

This section gives an overview of related algorithms from the literature that solve the graph coloring problem exactly. There exist various methods to do so, among the most successful in practice are branch-and-bound, branch-and-price and satisfiability-based algorithms.

A well-known and relatively successful branch-and-bound algorithm is the Dsatur algorithm of Brélaz [Bré79]. At each node of the branching tree, an uncolored vertex is chosen, and for each existing and feasible color, a child node is created where the vertex is assigned that color. Additionally, a child where the vertex uses a new color is created too. Each node thus represents a partial coloring, and a leaf yields a full coloring. A leaf using fewer colors than the current best coloring produces a new upper bound, used during the search to cut off partial colorings already using that many colors. An optimal coloring can thus be found as the minimum among all considered leaves. The focus of the literature for this approach has been to design decision strategies that yield better performance. The original author suggested picking the branching vertex as the most constrained vertex, i.e., with the fewest feasible colors available. Ties are broken by the maximum degree, Sewell [Sew96] instead breaks ties by the maximum number of common available colors in the neighborhood of uncolored vertices. The latter was extended by San Segundo [San12] who proposes to only consider uncolored vertices that are candidates in the tie. Initially, a lower bound in the form of a clique is computed but is not updated during the search. Since vertices are colored, but the graph itself does not change in branching, recomputing a clique does not yield better bounds. To address this shortcoming, Furini, Gabrel, and Ternier [FGT17] propose a reduced graph based on the partial coloring at each node. Vertices that are assigned the same color are merged, and edges are added between vertices colored differently. On this reduced graph, valid lower bounds are computed that allow further pruning of nodes in the search tree that were not possible with the static clique bound computed only at the root.

The branch-and-price methods model each color as a stable set and use a covering formulation for which the number of stable sets is minimized. The independent sets are binary variables of the ILP; as there can be exponentially many independent sets in a graph, a technique called column generation is used. This means independent sets are dynamically generated if they can potentially improve the value of the current problem. If no such independent set exists, the current solution is proven optimal without having to consider all exponential many stable sets as variables. The most studied branch-and-price algorithms use the following ILP formulation, where S is the set of all maximal independent sets.

$$\begin{aligned} \min \quad & \sum_{s \in S} x_s \\ s.t. \quad & \sum_{s \in S : v \in s} x_s \geq 1 \quad \forall v \in V \\ & x_s \in \{0, 1\} \quad \forall s \in S \end{aligned} \tag{CIP}$$

The linear relaxation of Equation (CIP) computes the fractional chromatic number $\chi_f(G)$ which provides a strong lower bound for $\chi(G)$. The strength of the set-covering formulation lies in this strong lower bound, as the root node often already yields a bound close to if not equal $\chi(G)$. This is due to the variables being associated with independent sets, so that no adjacent vertices can be given the same color even in the linear relaxation. This is possible in other classical ILP formulations, which do not yield as strong a bound with their relaxation[JM18]. The first branch-and-price algorithm using the formulation of Equation (CIP) was presented by Mehrotra and Trick [MT96]. The approach was improved by Malaguti, Monaci, and Toth [MMT11] where the algorithm was started with a strong initial set of variables to avoid some column generation iterations. Further, Held, Cook, and Sewell [HCS12] adapt the algorithm to obtain numerically safe results and propose a more effective pricing algorithm. Building on top of previous successes, Hulst [Hul21] presented a branch-and-price-and-cut algorithm with a new branching decision that produced strong results. It solves all graphs with at most 100 vertices of the used benchmark set and computes the fractional chromatic number of the large graph C2000.9.

Another approach that has proven to be quite successful is based on modeling the graph coloring problem as a satisifiability (SAT) problem. Van Gelder [Van08] presented three encodings and symmetry breaking techniques in 2008 and since then, many leaps have been made in the field of satisfiability solvers used for the encodings. In a hybrid approach, Zhou et al. [Zho+14] combine the Dsatur algorithm with SAT solver techniques such as conflict analysis and clause learning, reporting strong results on many instances. Hébrard and Katsirelos [HK20] use SAT techniques in their algorithm based on the Zykov recurrence, combining constraint programming with clause learning and using a propagator to prune nodes during the search. A simpler yet effective approach is that of Heule, Karahalios, and Hoeve [HKH22], using a SAT encoding to find improving cliques as lower bounds and then the assignment encoding with a local search solver to quickly find satisfying solutions.

A novel method that does not fall into the three categories is based on binary decision diagrams. Hoeve [Hoe22] presented an iterative approach that builds and solves a network flow ILP formulated on a binary decision diagram. As the full decision diagram can be of exponential size, the algorithm begins with a trivial decision diagram that is incrementally refined, leading to increasingly stronger lower bounds until eventually the chromatic number is computed. While not performing as well as other exact algorithms, it has the benefit of always having a valid lower bound available which also increases over time. With the branch-and-price method, the linear program at the root needs to be fully solved to obtain a first bound, and for the Dsatur-based branch-and-bound algorithm, only the initially computed lower bound is available.

For a survey of graph coloring problems as well as algorithms, the reader may refer to [MT10]; another survey of exact algorithms is given in [LC18].

1.4 Computational Setup

All algorithms of this thesis were implemented in C++ and compiled using g++11. Further, all experiments were run on a single thread of an AMD EPYC 7702 64-core processor, which takes 4.17 seconds to solve instance r500.5 of the comparative DFMAX benchmark [Tri02]. The implementations of this thesis use the `Bitset` class from version 1.84 of the `Boost` library. Other external sources for SAT solving are the Open-WBO MaxSAT solver [MML14] on top of Glucose 4.2 [AS18], and the most important external source and main solver used for satisfiability is CaDiCal [Bie+24]. In particular, a development

version[1] of the IPASIR-UP interface [Faz+23] was used since it is needed for the algorithms in Chapter 4. To compute an initial clique, CliSAT [San+23] is called with a time limit of one second [2]. The source code for all the algorithms implemented as part of this thesis can be found at https://github.com/trewes/IncSatGC-release.

For the evaluation of the graph coloring algorithms, the same set of 137 DIMACS benchmark instances [JT96] were used throughout this thesis, see [MT96] for a short description of the instances. Unless stated otherwise, experiments were run with a time limit of one hour, and a memory limit of 16 GB. Table 1.1 lists the 137 instances of the benchmark set and provides the number of vertices n, number of edges m and density d as well as the best known lower and upper bounds $\underline{\chi}$ resp. $\overline{\chi}$. The bounds are taken from the literature, mostly as stated in [Hoe22] but updated with new bounds from [Hul21; HKH22; Por20]. The entries in the last two columns are bold if lower and upper bound are equal, i.e., the chromatic number is known, and the instance solved. From a survey of recent graph coloring literature, the only unsolved instances of the benchmark set seem to be 1-Insertions_6, 3-Insertions_5, C2000.5, C2000.9, C4000.5, DSJC1000.1, DSJC1000.5, DSJC1000.9, DSJC250.1, DSJC250.5, DSJC500.1, DSJC500.5, DSJC500.9, flat1000_76_0, latin_square_10, r1000.1c, wap03a, wap04a, wap07a, highlighted with gray in the table.

Table 1.1: Benchmark instances

Instance	n	m	d	$\underline{\chi}$	$\overline{\chi}$
1-FullIns_3	30	100	0.23	**4**	**4**
1-FullIns_4	93	593	0.14	**5**	**5**
1-FullIns_5	282	3247	0.08	**6**	**6**
1-Insertions_4	67	232	0.10	**5**	**5**
1-Insertions_5	202	1227	0.06	**6**	**6**
1-Insertions_6	607	6337	0.03	4	7
2-FullIns_3	52	201	0.15	**5**	**5**
2-FullIns_4	212	1621	0.07	**6**	**6**
2-FullIns_5	852	12201	0.03	**7**	**7**
2-Insertions_3	37	72	0.11	**4**	**4**
2-Insertions_4	149	541	0.05	**5**	**5**
2-Insertions_5	597	3936	0.02	**6**	**6**
3-FullIns_3	80	346	0.11	**6**	**6**
3-FullIns_4	405	3524	0.04	**7**	**7**
3-FullIns_5	2030	33751	0.02	**8**	**8**
3-Insertions_3	56	110	0.07	**4**	**4**
3-Insertions_4	281	1046	0.03	**5**	**5**
3-Insertions_5	1406	9695	0.01	4	6
4-FullIns_3	114	541	0.08	**7**	**7**
4-FullIns_4	690	6650	0.03	**8**	**8**
4-FullIns_5	4146	77305	0.01	**9**	**9**
4-Insertions_3	79	156	0.05	**4**	**4**
4-Insertions_4	475	1795	0.02	**5**	**5**
5-FullIns_3	154	792	0.07	**8**	**8**
5-FullIns_4	1085	11395	0.02	**9**	**9**

[1] https://github.com/arminbiere/cadical/tree/ipasirup-for-rc2, commit b3b2c85964b5eb2482c66e7dcb1efb68a54e0dbb

[2] A binary of their program was made available at https://github.com/psanse/CliSAT

Table 1.1: (continued)

Instance	n	m	d	$\underline{\chi}$	$\overline{\chi}$
abb313GPIA	1557	53356	0.04	**9**	**9**
anna	138	493	0.05	**11**	**11**
ash331GPIA	662	4181	0.02	**4**	**4**
ash608GPIA	1216	7844	0.01	**4**	**4**
ash958GPIA	1916	12506	0.01	**4**	**4**
C2000.5	2000	999836	0.50	99	145
C2000.9	2000	1799532	0.90	390	400
C4000.5	4000	4000268	0.50	107	259
david	87	406	0.11	**11**	**11**
DSJC1000.1	1000	49629	0.10	10	20
DSJC1000.5	1000	249826	0.50	73	82
DSJC1000.9	1000	449449	0.90	216	222
DSJC125.1	125	736	0.09	**5**	**5**
DSJC125.5	125	3891	0.50	**17**	**17**
DSJC125.9	125	6961	0.90	**44**	**44**
DSJC250.1	250	3218	0.10	7	8
DSJC250.5	250	15668	0.50	26	28
DSJC250.9	250	27897	0.90	**72**	**72**
DSJC500.1	500	12458	0.10	9	12
DSJC500.5	500	62624	0.50	43	47
DSJC500.9	500	112437	0.90	123	126
DSJR500.1	500	3555	0.03	**12**	**12**
DSJR500.1c	500	121275	0.97	**85**	**85**
DSJR500.5	500	58862	0.47	**122**	**122**
flat1000_50_0	1000	245000	0.49	**50**	**50**
flat1000_60_0	1000	245830	0.49	**60**	**60**
flat1000_76_0	1000	246708	0.49	72	81
flat300_20_0	300	21375	0.48	**20**	**20**
flat300_26_0	300	21633	0.48	**26**	**26**
flat300_28_0	300	21695	0.48	**28**	**28**
fpsol2.i.1	496	11654	0.09	**65**	**65**
fpsol2.i.2	451	8691	0.09	**30**	**30**
fpsol2.i.3	425	8688	0.10	**30**	**30**
games120	120	638	0.09	**9**	**9**
homer	561	1629	0.01	**13**	**13**
huck	74	301	0.11	**11**	**11**
inithx.i.1	864	18707	0.05	**54**	**54**
inithx.i.2	645	13979	0.07	**31**	**31**
inithx.i.3	621	13969	0.07	**31**	**31**
jean	80	254	0.08	**10**	**10**
latin_square_10	900	307350	0.76	90	97
le450_15a	450	8168	0.08	**15**	**15**
le450_15b	450	8169	0.08	**15**	**15**
le450_15c	450	16680	0.17	**15**	**15**
le450_15d	450	16750	0.17	**15**	**15**
le450_25a	450	8260	0.08	**25**	**25**

Table 1.1: (continued)

Instance	n	m	d	$\underline{\chi}$	$\overline{\chi}$
le450_25b	450	8263	0.08	**25**	**25**
le450_25c	450	17343	0.17	**25**	**25**
le450_25d	450	17425	0.17	**25**	**25**
le450_5a	450	5714	0.06	**5**	**5**
le450_5b	450	5734	0.06	**5**	**5**
le450_5c	450	9803	0.10	**5**	**5**
le450_5d	450	9757	0.10	**5**	**5**
miles1000	128	3216	0.40	**42**	**42**
miles1500	128	5198	0.64	**73**	**73**
miles250	128	387	0.05	**8**	**8**
miles500	128	1170	0.14	**20**	**20**
miles750	128	2113	0.26	**31**	**31**
mug100_1	100	166	0.03	**4**	**4**
mug100_25	100	166	0.03	**4**	**4**
mug88_1	88	146	0.04	**4**	**4**
mug88_25	88	146	0.04	**4**	**4**
mulsol.i.1	197	3925	0.20	**49**	**49**
mulsol.i.2	188	3885	0.22	**31**	**31**
mulsol.i.3	184	3916	0.23	**31**	**31**
mulsol.i.4	185	3946	0.23	**31**	**31**
mulsol.i.5	186	3973	0.23	**31**	**31**
myciel3	11	20	0.36	**4**	**4**
myciel4	23	71	0.28	**5**	**5**
myciel5	47	236	0.22	**6**	**6**
myciel6	95	755	0.17	**7**	**7**
myciel7	191	2360	0.13	**8**	**8**
qg.order100	10000	990000	0.02	**100**	**100**
qg.order30	900	26100	0.06	**30**	**30**
qg.order40	1600	62400	0.05	**40**	**40**
qg.order60	3600	212400	0.03	**60**	**60**
queen10_10	100	1470	0.30	**11**	**11**
queen11_11	121	1980	0.27	**11**	**11**
queen12_12	144	2596	0.25	**12**	**12**
queen13_13	169	3328	0.23	**13**	**13**
queen14_14	196	4186	0.22	**14**	**14**
queen15_15	225	5180	0.21	**15**	**15**
queen16_16	256	6320	0.19	**16**	**16**
queen5_5	25	160	0.53	**5**	**5**
queen6_6	36	290	0.46	**7**	**7**
queen7_7	49	476	0.40	**7**	**7**
queen8_12	96	1368	0.30	**12**	**12**
queen8_8	64	728	0.36	**9**	**9**
queen9_9	81	1056	0.33	**10**	**10**
r1000.1	1000	14378	0.03	**20**	**20**
r1000.1c	1000	485090	0.97	97	98
r1000.5	1000	238267	0.48	**234**	**234**

Table 1.1: (continued)

Instance	n	m	d	$\underline{\chi}$	$\overline{\chi}$
r125.1	125	209	0.03	**5**	**5**
r125.1c	125	7501	0.97	**46**	**46**
r125.5	125	3838	0.50	**36**	**36**
r250.1	250	867	0.03	**8**	**8**
r250.1c	250	30227	0.97	**64**	**64**
r250.5	250	14849	0.48	**65**	**65**
school1	385	19095	0.26	**14**	**14**
school1_nsh	352	14612	0.24	**14**	**14**
wap01a	2368	110871	0.04	**41**	**41**
wap02a	2464	111742	0.04	**40**	**40**
wap03a	4730	286722	0.03	40	43
wap04a	5231	294902	0.02	40	41
wap05a	905	43081	0.11	**50**	**50**
wap06a	947	43571	0.10	**40**	**40**
wap07a	1809	103368	0.06	40	41
wap08a	1870	104176	0.06	**40**	**40**
will199GPIA	701	6772	0.03	**7**	**7**
zeroin.i.1	211	4100	0.19	**49**	**49**
zeroin.i.2	211	3541	0.16	**30**	**30**
zeroin.i.3	206	3540	0.17	**30**	**30**

Chapter 2

Computation of Bounds and Novel Preprocessing

This chapter discusses methods for reducing the computational effort required for solving the sequence of decision problems. There exist two main ways to do this in the context of satisfiability and graph coloring. First, one can reduce the number of decision problems that need to be solved by finding strong initial bounds on $\chi(G)$ with external methods. One then only has to check values in the range of $[lb, ub)$ for k-colorability, instead of $[1, n)$ without bounds. Techniques to compute such lower and upper bounds are discussed in Section 2.1 and Section 2.2.

Additionally, one can reduce the graph on which a graph coloring is to be computed by removing or contracting vertices and adding edges. A reduced graph that is smaller leads to a smaller problem formulation, and some instances can be solved by the reduction rules alone. Preprocessing rules from the literature, and a novel method based on an auxiliary graph, are presented in Section 2.3. Finally, the effectiveness of these techniques is reviewed in Section 2.4.

2.1 Lower Bounds

There exists a variety of lower bounds for $\chi(G)$ in the literature, this chapter briefly mentions some and more thoroughly discusses others. A simple bound is based on the stability number $\alpha(G)$ of G, i.e., the size of a maximal independent set in the graph; and although this is an NP-hard problem [Kar72], efficient algorithms for it exist. Any coloring is a partition into stable sets, and the best case would be that every color uses $\alpha(G)$ many vertices. This yields the following bound:

$$\chi_\alpha(G) \coloneqq \left\lceil \frac{n}{\alpha(G)} \right\rceil \leq \chi(G) \tag{2.1}$$

Another bound is provided by the Hoffman number $\chi_H(G)$ [Hof03]. It is based on the largest and the least eigenvalue λ_1 resp. λ_n of the adjacency matrix of G and as such can be computed in polynomial time.

$$\chi_H(G) \coloneqq 1 - \frac{\lambda_1}{\lambda_n} \leq \chi(G) \tag{2.2}$$

There is also the Lovász number $\vartheta(G)$ [Lov79] that can be stated as the optimal value of a semi-definite program and can therefore be computed in polynomial time [GLS81]. It was originally used to bound the shannon capacity of a graph, but it also provides a lower bound for the chromatic number by the following "sandwich theorem":

$$\alpha(G) \leq \vartheta(G) \leq \chi(\overline{G}) \tag{2.3}$$

The bounds presented so far are not very strong bounds, and the gap to the chromatic number can be arbitrarily large on some graphs. Although this is true for every bound presented in this section, the fractional chromatic number $\chi_f(G)$ and the lower bound based on decisions diagrams [Hoe22] are quite strong in practice but also more challenging to compute. The former can be obtained by solving the linear relaxation of the set covering formulation of Equation (CIP) for the graph coloring problem [HCS12]. The latter builds and solves a network flow ILP on an incrementally growing decision diagram, and the bound improves the more the decision diagram is refined in the process. As these bounds are not relevant for this thesis, they are not presented in detail. Instead, the clique bound, the Mycielsky bound, and a bound based on an auxiliary graph are discussed in the rest of this section.

2.1.1 Clique Bound

The most well-known and used lower bound is the clique bound: given a clique of size k, at least k colors are needed to color the clique since every vertex is connected to every other vertex and needs a distinct color. Computing the clique number $\omega(G)$ exactly or heuristically producing a large clique C in the original graph G thus yields a simple lower bound for the chromatic number. If the bound is tight, i.e., $\omega(G) = \chi(G)$, G is called a perfect graph. While the problem of finding a maximal clique is also NP-hard [Kar72], there exist exact and heuristic algorithms that are very efficient in practice, making their use viable and beneficial for the graph coloring problem. A comprehensive survey and performance comparison of exact and heuristic methods can be found in [WH15].

For this thesis, two kinds of algorithms are required: one that is not necessarily the fastest but can produce a strong initial clique, and one that does not necessarily produce great bounds but runs extremely fast. For the first requirement, the exact combinatorial branch-and-bound algorithm CliSAT [San+23] is called with a time limit of one second. For the latter, a simple greedy technique is employed whose pseudocode is given in Algorithm 2.1. It iterates over the vertices in a given order and attempts to add the vertex to every possible clique in the currently maintained list if possible, otherwise a new singleton clique is created. The vertices are iterated once more to check if they can be added to a clique that was added only after it was processed the first time. There can be at most n cliques, so at most $O(n^2)$ iterations are performed in the nested loops, each requires checking $C \subseteq N(v_i)$ which can be done with a single bitwise operation when using the bitset data structure from Section 1.1. This greedy procedure was also used by [HK20], whose approach to graph coloring is discussed in Chapter 4.

2.1.2 Mycielsky Bound

The previous clique bound was presented as a number, but the bound idea is that of recognizing a certain subgraph and using the chromatic number of that subgraph as bound for the graph it is embedded in. This is the same idea as for the Mycielsky bound, where generalised Mycielsky graphs are recognized and produce a lower bound. Before going into detail how the bound is computed, the Mycielsky transformation is defined.

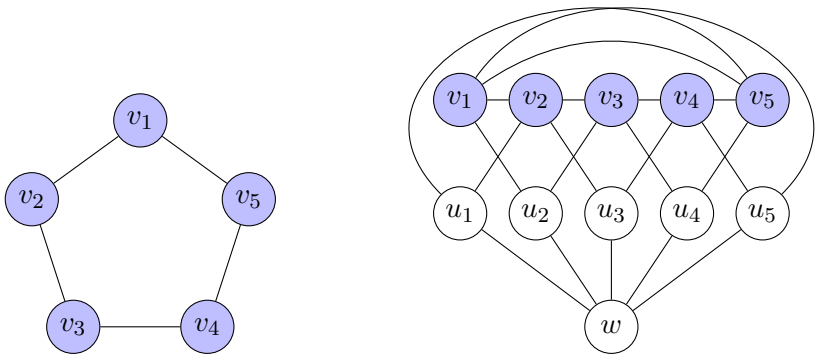
Definition 2.1.1 (Mycielsky transformation [Myc55])**.** Given a graph $G = (V, E)$, the Mycielsky transformation or the Mycelskian $\mu(G)$ is constructed as follows:

1. Begin with a copy of G.
2. For each vertex $v_i \in V$, add a copy u_i connected to all neighbors $v_j \in N_G(v_i)$.
3. Add a vertex w connected to all u_i.

Algorithm 2.1: GreedyCliques

Input: Graph G, ordering $v_1, \ldots, v_n$
Output: List of cliques $\mathcal{C}$

```
1  C ← ∅
2  for i ← 1 to n do
3      foreach C ∈ C do
4          if C ∪ {v_i} is a clique then
5              C ← C ∪ {v_i} // Add v_i to all cliques that admit it
6      if v_i was not added to any clique then
7          C ← C ∪ {v_i} // Otherwise add {v_i} as singleton clique
8  for i ← 1 to n do
9      foreach C ∈ C created after v_i was added do
10         if C ∪ {v_i} is a clique then
11             C ← C ∪ {v_i}
12 return C
```



(a) M_3 (b) M_4

Figure 2.1: Mycielsky graph M_3 and transformation $M_4 = \mu(M_3)$

This results in the graph $\mu(G)$ with vertex set $\mu(V) = V \cup U \cup \{w\}$ and edges $\mu(E) = E \cup \{\{v_i, u_j\}, \{v_j, u_i\} : \{v_i, v_j\} \in E\} \cup \{\{u_i, w\} : u_i \in U\})$.

One can think of this construction in layers: the first layer V is the original graph. The second layer U is almost a copy of the first layer, except that the vertices are only adjacent to vertices in the first layer V and w. Finally, there is the special vertex w that is adjacent to all vertices U of the second layer. This construction has the interesting property that $\mu(G)$ and G have the same clique number while $\chi(\mu(G)) = \chi(G) + 1$ [Myc55; YWH20]. If this transformation is iterated beginning with a triangle-free graph, all graphs in the sequence are triangle-free while their chromatic number grows arbitrarily large. Beginning with the 2-clique $M_2 = K_2$ and defining $M_k = \mu(M_{k-1})$ for $k > 2$, M_k is called the *kth Mycielsky graph* and has chromatic number k. The transformation from M_3 to M_4 is given in Figure 2.1. This shows one family of graphs where the clique bound is clearly insufficient to provide a meaningful lower bound.

Definition 2.1.1 can be extended to pseudo Mycielskians, which are Mycielskians up to added edges and some vertex contraction, For example, vertices v_i and u_i, or v_i and w, could be merged. Both added edges and vertex contractions only constrain the graph coloring problem further and thus do not decrease the chromatic number; as such, they provide the same lower bound if found as subgraph. In [HK20], Hébrard and Katsirelos introduced a greedy procedure to find such embedded pseudo Mycielskians; it is presented here in detail. The algorithm starts with a subgraph and tries to find the next Mycielskian by following the construction in Definition 2.1.1. Beginning with a subgraph $H = (V_H, E_H)$ of G, it is necessary to find vertices U and w that form the second layer and the special vertex. For each $v \in V_H$, a vertex $u \in V$ is sought that is also adjacent to all neighbors of v, and, after computing these u for every vertex v, they form the second layer U. Then, the additional vertex w adjacent to all the u is still required. To compute the second layer, define the following sets of candidates for each $v \in V_H$:

$$S_v \coloneqq \{u : N_H(v) \subseteq N_G(u)\} \tag{2.4}$$

To compute w, choose an element with at least one neighbor in every set S_v:

$$w \in \cap_{v \in V_H} N_G(S_v) \tag{2.5}$$

If (2.5) is not empty, further let $u(v)$ map to any element of S_v adjacent to w and define $U = \{u(v) : v \in V\}$. Note that $u(v) = v$ or $w = v$ for some $v \in V$ is possible, as the algorithm tries to detect pseudo Mycielskians. This collects all the data for constructing the Mycielskian and leads to the following lemma, as stated and proven in [HK20]:

Lemma 2.1.2 ([HK20]). *For the graph*

$$H' = (V \cup U \cup \{w\}, E \cup \bigcup_{v \in V} N_H(v) \times u(v) \cup \bigcup_{u \in U} \{\{u, w\}\})$$

it holds that

$$\chi(H') \geq \chi(H) + 1$$

Proof. The lemma follows by observing that H' is the Mycielskian graph of H embedded in G, possibly with contracted vertices. First, if $u(v) \neq v$ for all $v \in V$, then $H' = \mu(H)$ is the Mycielskian of H by using $u(v_i) = u_i$ as the vertices U of the second layer and w for itself as the special vertex, as in Definition 2.1.1. If $H' \neq \mu(H)$, this is only possible because either:

- For some vertex $v_i \in V_H$, one finds $u(v_i) = v_i$. In this case, consider $\mu(H)$ with vertices u_i and v_i contracted. This graph is isomorphic to H' and still has chromatic number at least $\mu(H)$.
- For some vertex $v_i \in V_H$, one finds $w = v_i$. Consider $\mu(H)$ but now with vertices v_i and w contracted, which is again isomorphic to H'.

The third case of w being among the vertices U cannot happen: for any $v \in V_H$, $u(v) \notin \cap_{v \in V_H} N_G(S_v)$ because $u(v)$ is not a neighbor of itself. □

Algorithm 2.2 gives the pseudocode for the Mycielskian subgraph detection algorithm. It starts with a subgraph H of known chromatic number and builds a larger subgraph H', according to the data needed for Lemma 2.1.2. If successful, H' has chromatic number larger than H. This is repeated until either $|V_H| = |V|$, or the set W of candidates for w becomes empty in an iteration. The algorithm returns the number of successful iterations r, which together with the known chromatic number of H yields a new lower bound of $\chi(H)+r$ for $\chi(G)$.

Algorithm 2.2: MycielskyBound

Input: Graph G, subgraph $H = (V_H, E_H)$
Output: Number of succesful iterations r

```
1  r ← 0
2  while |V_H| < |V| do
3      W ← V
4      foreach v ∈ V_H do S_v ← {v};
5      foreach v ∈ V_H do
6          foreach u ∈ W do
7              if N_H(v) ⊆ N_G(u) then S_v ← S_v ∪ {u} ;
8          W ← W ∩ N_G(S_v)
9      if W ≠ ∅ then
10         r ← r + 1
11         (V'_H, E'_H) ← (V_H, E_H)
12         w ← any element of W
13         V'_H ← V'_H ∪ {w}
14         foreach v ∈ V_H do
15             V'_H ← V'_H ∪ {u any element of N_G(w) ∩ S_v}
16             E'_H ← E'_H ∪ {{u, w}} ∪ {u × N_H(v)}
17         (V_H, E_H) ← (V'_H, E'_H)
18     else break;
19 return r
```

Each iteration of Algorithm 2.2 performs $O(|V_H| \cdot |V|)$ bitset operations: $O(|V_H| \cdot |V|)$ subset and "OR" operations in line 7 and $O(|V_H|)$ "AND" operations in line 8. Additionally, the loop in line 15 requires $O(|V_H|^2)$ time. Now Hébrard and Katsirelos bound the number of total iterations by $\log |V|$, saying that in each iteration the number of vertices in H is at least doubled. This does not always hold when working with pseudo mycielskians, if $u(v_i) = v_i$ for every $v_i \in V_H$ and only w is a new vertex, the number of vertices increases by only one. If G is a clique and one begins the algorithm with a single vertex as subgraph, then $|V|$ iterations are required, extending the clique subgraph by one vertex in each iteration. Thus, only a bound of $|V|$ on the number of iterations can be concluded, which yields a total of $O(|V|^3)$ bitset operations; contrary to the $O(|V|^2)$ claimed in [HK20]. Also briefly noted here is that line 13 is missing in the pseudocode of [HK20].

Since the algorithm is only a greedy procedure, there is no guarantee that it will detect a Mycielskian subgraph if it exists. Further, the choices made in and even before the algorithm matter too: vertex choices in lines 12 and 15 have an impact on which subgraph is found, and different starting subgraphs also lead to different outcomes.

Example 2.1.3. Consider an example run of Algorithm 2.2 on Figure 2.1(b). If the simple 5-cycle $V_H = \{v_1, \dots, v_5\}$ of the first layer is given as the initial subgraph, the algorithm correctly finds $u(v_i) = u_i$ and w as the single vertex in the last layer. If the algorithm is instead called on the subgraph induced by $V_H = \{v_1, u_2, v_3, v_4, v_5\}$, again a simple 5-cycle, it cannot be extended to the next Mycielskian as the set $\cap_{v \in V_H} N_G(S_v)$ is empty. This is because the desired special vertex is w, but it is not adjacent to any vertex in $S_{u_2} = \{v_2\}$.

For their propagator, Hébrard and Katsirelos propose to start their algorithm with a large clique to improve upon the known clique bound. Following this idea but using it

to compute an initial lower bound did not achieve satisfactory results. One issue became clear with the example run on Figure 2.1(b); the algorithm is not able to detect the full Mycielskian even in the Mycielsky graph itself if the wrong starting subgraph was chosen. For that reason, in the initial computation of bounds using Algorithm 2.2, it is called with the empty subgraph. While still many choices are made during the algorithm, this can determine the optimal lower bound at least on the Mycielsky graphs. Note that this is not guaranteed and depends on the vertex ordering leading to the right choices. In this implementation the lexicographically first vertex is chosen if several candidates are available in lines 12 and 15.

An attempt was also made at implementing different choices when choosing w and $u(v)$ during the algorithm. Neither a decision strategy based on degrees, nor on trying to choose $u(v) = v$ if available, to possibly reduce the size of the pseudo Mycielskian, lead to significant changes in the computed bounds.

A discussion of the quality and the types of graphs where this bound is successful is given in Section 2.4. The algorithm is further used for dynamically computing lower bounds in the user propagators presented in Chapter 4.

2.1.3 Bound from Auxiliary Graph

Cornaz and Jost [CJ08] reduced the graph coloring problem to the maximum stable set problem in an auxiliary graph and showed a 1-to-1 correspondence between colorings of the graph G and stable sets of its auxiliary graph G_A.

For the construction of the auxiliary graph G_A, an orientation $\overline{G}_{\rightarrow}$ of the complement graph $\overline{G}$ needs to be specified, i.e., for each edge $\{v, w\} \in \overline{G}$, either (v, w) or (w, v) is chosen as arc according to the orientation. First, the line graph $L(G)$ is defined as having a vertex for each edge in G, and two vertices are adjacent if their corresponding edges are incident.

$$L(G) \coloneqq (E(G), \{e_1, e_2\} : e_1, e_2 \in E(G) \text{ and } |e_1 \cap e_2| = 1) \tag{2.6}$$

Further, a pair of arcs a, b in the directed graph $\overline{G}_{\rightarrow}$ is called simplicial if they are of the form $a = (v_i, v_j), b = (v_i, v_k)$ and either (v_j, v_k) or (v_k, v_j) is an arc of $\overline{G}_{\rightarrow}$, for distinct v_i, v_j, v_k. Then, G_A is the line graph of the complement graph $L(\overline{G})$ minus edges which correspond to simplicial arcs in $\overline{G}_{\rightarrow}$.

The connection between stable sets and colorings is as follows: if a vertex $\{v_i, v_j\} \in V(G_A)$ is added to the stable set S_A, this corresponds to assigning v_i and v_j the same color in G. By doing so, one less color is used for G, such that a total of $k = |V(G)| - |S_A|$ colors are needed. Likewise, for any coloring where v_i and v_j are of the same color class, they correspond to $\{v_i, v_j\}$ being in the stable set of G_A. This also shows why simplicial arcs were removed from $L(\overline{G}_{\rightarrow})$: for (v_i, v_j) and (v_i, v_k) in $\overline{G}_{\rightarrow}$, omitting $\{\{v_i, v_j\}, \{v_i, v_k\}\}$ reflects that if v_j is assigned the same color as v_i, v_k can also still be assigned the same color as v_i and vice versa, i.e., both (v_i, v_j) and (v_i, v_k) can be included in the stable set. The formal theorem for the connection is as follows:

Theorem 2.1.4 ([CJ08]). *For any graph G and any acyclic orientation of its complement graph, there is a one-to-one correspondence between the set of all colorings of G and the set of all stable sets of G_A. Moreover, for any coloring $\{I_1, \ldots, I_k\}$ and its corresponding stable set S_A in G_A, it holds that $|S_A| + k = |V(G)|$. In particular, $\alpha(G_A) + \chi(G) = |V(G)|$.*

From this result, it is also clear how the correspondence can be used to produce a lower bound for $\chi(G)$. Given an upper bound $\overline{\alpha}(G_A)$, Theorem 2.1.4 implies the following inequality.

$$\chi(G) = |V(G)| - \alpha(G_A) \geq |V(G)| - \lceil \overline{\alpha}(G_A) \rceil \tag{2.7}$$

Therefore, any upper bound on the stability number of G_A can be used to compute a lower bound for the chromatic number of G. In [FGT17], an improved DSATUR algorithm was presented that uses this idea to find strong lower bounds and utilizes them to prune the search tree. Furini, Gabrel, and Ternier used the ILP formulation for the maximum stable set problem to efficiently find strong upper bounds and described which solver configurations and cuts they observed to be beneficial for this computation. A computational study on using the correspondence for solving the graph coloring problem was done in [CFM17].

Both papers also highlight the disadvantage of tackling graph coloring with the auxiliary graph: it becomes too large for large, sparse graphs. In [FGT17] the authors only report good results for high-density instances where the bound can be computed fast enough; on some sparser instances the bound could not be computed at all within the time limit of one hour. The study of [CFM17] was limited to 58 out of their benchmark set of 115 instances because the size of the auxiliary graph exceeded 30,000 vertices. As such, the lower bound based on G_A was not used in this thesis. The method and construction of the auxiliary graph is presented here since the correspondence will be used to discuss a new preprocessing method for graph coloring in Section 2.3. Further, its potential use as a dynamic lower bound technique is explored in Section 4.4.2.

2.2 Coloring Heuristics

If a good upper bound on the chromatic number is known, it potentially reduces the number of decision problems that need to be solved to determine $\chi(G)$. Trivially, any valid coloring provides an upper bound and therefore any coloring heuristic can be employed to find an initial bound. Such an algorithm should have polynomial runtime but still produce a good coloring, and a whole range of sophisticated heuristic algorithms exist in the literature. Tabu search [HW87] and local search methods [GH06, Survey] are among the most popular methods. They start with an initial coloring and try to improve it by performing local changes, which can quickly lead to good solutions, but local search is limited by only considering local information of the graph. Population-based hybrid algorithms, such as evolutionary algorithms [GH99] or simulated quantum annealing [TC11], are among the most effective methods and have reported the best solutions on many challenging DIMACS instances. They are more complex in design and require specification of effective operators for meaningful recombination, local optimization and maintaining population diversity. For this thesis, simple and fast methods are preferred, as finding a coloring only serves as initializing the upper bound before solving a range of decision problems. To this end, greedy coloring algorithms are presented in more detail in the rest of this section. The listed algorithms are by no means exhaustive, for more extensive studies on heuristics for the graph coloring problem see [MT10; MKN20].

The simplest heuristic algorithms are typical greedy algorithms, with two main variants. Sequential greedy algorithms determine some fixed vertex ordering, for example, sorting the vertices by degree. The smallest last algorithm by Matula, Marble, and Isaacson [MMI72] orders the vertices $v_1, \ldots, v_n$ such that v_i is a vertex of minimum degree in the subgraph induced by $\{v_1, \ldots, v_i\}$. Several different ordering techniques are discussed in [RA14]. The vertices are then traversed in that order by the greedy algorithm, and each vertex is assigned the first color not assigned to any of its neighbors. There always exists a vertex ordering such that the simple greedy finds an optimal coloring, but finding such an ordering is in itself NP-hard as it solves the graph coloring problem.

The second variant makes dynamic choices of which vertex to color next; most well known is the Dsatur algorithm by Brélaz [Bré79], whose exact variant was briefly described in

Section 1.3. It chooses the next vertex to color according to its **D**egree-**satur**ation, that is, the vertex that maximizes $sat(v) \coloneqq |\{c(u) : \{u, v\} \in E\}|$, the number of different colors it is adjacent to. The original algorithm breaks ties by maximum degree, but some improved strategies have been suggested in the literature [Sew96; San12]. The Dsatur algorithm in turn produces an ordering, also called the Dsatur ordering, and can be used in graph coloring algorithms where the vertex ordering has an impact on solving performance [Hoe22; CFM17]. The Dsatur ordering will also be used for the Zykov-based encodings presented in Section 3.2, and as such, the pseudocode of Dsatur using the degree as tie-break is given in Algorithm 2.3.

Algorithm 2.3: Dsatur

Input: Graph $G = (V, E)$, optionally a clique C
Output: A coloring $c : V \mapsto \mathbb{N}$ of G

```
if C is not empty then
    Assign a different color to each vertex of the clique
while there exists an uncolored vertex do
    v ← arg max_{v∈V} sat(v), ties broken by degree
    c(v) ← first color i not assigned to any of its neighbours
    if i is a new color then
        Attempt to recolor vertex v
return c
```

Two potential improvements to the greedy algorithm are mentioned. One can fix a clique at the beginning of the vertex ordering, whose vertices are each assigned a different color. The idea is that the following vertices are more constrained, and thus detrimental color assignment might be avoided by having already partially constrained the color of some vertices. Second, the greedy algorithms can be augmented by the recolor technique [RA14]: when a new color i is used for a vertex v, it is checked whether there exist two color classes $C_j = \{u \in V : c(u) = j\}$ and C_k with $j < k$ and a vertex u such that $N(v) \cap C_j = \{u\}$ and $N(u) \cap C_k = \emptyset$. If these conditions are met, u can be assigned color k instead, freeing up color j for vertex v, thus avoiding the new color i. Both these improvements are implemented in the Dsatur variant used in this thesis.

2.3 Preprocessing

The previous two sections presented lower resp. upper bounds with the aim of reducing the number of decision problems that need to be solved, or even solving the instance by finding matching lower and upper bounds. Another way to reduce the required effort to solve an instance is to reduce the problem size, in this context by removing vertices, adding edges or contracting two vertices. This section presents standard and less used rules found in the literature, and introduces a novel preprocessing rule based on the auxiliary graph construction defined in Section 2.1.3.

2.3.1 Rules From the Literature

The first two rules from Lemmas 2.3.1 and 2.3.2 are commonly used in graph coloring literature [JM18; HLS09; Hul21]. They focus on removing vertices from the graph that are redundant in the way that an optimal coloring of the original graph can easily be recovered

from the reduced graph. The first paper presenting these rules appears to be "Pre-processing and linear-decomposition algorithm to solve the k-colorability problem" by Lucet, Mendes, and Moukrim [LMM04].

Lemma 2.3.1 (Dominated Vertices)**.** *A vertex u is said to be dominated (by v) if there exists another vertex v with $N(u) \subseteq N(v)$[1]. If u is dominated, it can be removed from the graph and safely assigned the same color as v since it is adjacent to the same or more vertices than u.*

Lemma 2.3.2 (Low Degree Vertices)**.** *Assuming there is some known lower bound k for the chromatic number, vertices u with degree smaller than k are called low-degree vertices and can be removed as well: since u is adjacent to at most $k-1$ vertices and thus at most $k-1$ colors and at least k colors are needed, there is always a color available for u.*

Note that removing vertices with degree $k-1$ might reduce the chromatic number if $\chi(G) = k$, for example, in a k clique every vertex has degree $k-1$ and is removed by Lemma 2.3.2. In that case, one has to account for this when recovering an optimal coloring from the reduced graph, where a removed low-degree vertex might require a new color but k colors will be enough overall. If the reduced graph becomes empty or has fewer vertices than the bound k, the chromatic number is determined to be k by preprocessing alone. This rule can be implemented in linear time with the right graph structure, since for each vertex only a call to compute its degree is required. The rule of Lemma 2.3.1 takes $O(n^2)$ to apply, since for every pair of vertices it is checked whether one dominates the other. These two rules are applied iteratively, first on the original graph and then on the subsequent reduced graphs, until neither reduction rule removes any further vertices.

The same paper [LMM04] presented two further preprocessing rules that instead contract two vertices or add an edge to the graph. These rules do not appear to be used in the general graph coloring literature, except for [HK20, Section 6.3] who seem to have discovered a version themselves in the context of constraint programming and marginal costs. This might be due to the original paper being less known and the rules needing stricter assumptions. For both, it is required that one is searching for a k-coloring and that a subgraph C with chromatic number k has previously been found. The original rules were presented based on a clique of size k but any subgraph whose chromatic number is k can be used just the same.

Lemma 2.3.3 (Vertex Fusion [LMM04])**.** *Let u, v be a pair of non-adjacent vertices such that $u \in C$ and $v \notin C$. If v is adjacent to all vertices of C except u, then u and v must be assigned the same color. The clique needs k different colors and v is adjacent to all but one vertex of C, thus v cannot be assigned a different color than u as this would lead to $k+1$ colors being used. That u and v need to be assigned the same color is realized in the graph by merging the two vertices.*

The second rule is called edge addition and complementary to Lemma 2.3.3. Instead of determining two vertices that need to be assigned the same color, this finds pairs of vertices who need to be assigned different colors and thus an edge is added.

Lemma 2.3.4 (Edge Addition [LMM04])**.** *Let u, v be a pair of non-adjacent vertices in $V \backslash C$. If $\{u, w\} \in E$ or $\{v, w\} \in E$ for all $w \in C$, then u and v must be assigned different colors. This is because u must take a color from the colors assigned to $C \backslash N(u)$, and since $C \backslash N(u) \subseteq N(v)$, v is adjacent to all those colors and must be assigned a color different from that of u. This is realized in the graph by adding the edge $\{u, v\}$.*

[1] u and v have to be non-adjacent but this is implied by $v \notin N(v)$

The check for possible vertex fusions can be implemented in $O(n \cdot |C|)$ for each subgraph used, and edge addition can be checked in $O(n^2 \cdot |C|)$ for each subgraph. These rules can again be applied iteratively and in combination with Lemmas 2.3.1 and 2.3.2.

As Lemmas 2.3.3 and 2.3.4 are only applicable for solving the k-coloring problem and when a subgraph with chromatic number k is known, this makes these preprocessing rules less applicable when trying to compute the chromatic number in general. For this reason, only Lemmas 2.3.1 and 2.3.2 are applied iteratively before calling the satisfiability-based algorithm to determine the chromatic number. The pseudocode of the full preprocessing routine, including computation of initial lower and upper bounds, is given in Algorithm 2.4. While the vertex fusion and edge addition preprocessing rules are not used in this routine, they will see an application inside the user propagators of Chapter 4.

Algorithm 2.4: PreprocessingRoutine

Input: Graph $G = (V, E)$
Output: A (potentially) reduced graph G', initial lower and upper bounds lb and ub

$lb \leftarrow \max\{\text{CliSat}(G), \text{MycielskyBound}(G, \emptyset)\}$
$G' \leftarrow G$
do
  $G' \leftarrow$ remove low degree vertices
  $G' \leftarrow$ remove dominated vertices
while *while* G' *changed and* $|G'| > lb$;
$lb \leftarrow \max\{\text{CliSat}(G'), \text{MycielskyBound}(G', \emptyset)\}$
$ub \leftarrow \text{Dsatur}(G')$
return G', lb, ub

2.3.2 Novel Preprocessing Based on Auxiliary Graph

This section uses the auxiliary graph construction presented in Section 2.1.3. As mentioned previously, the construction has been used for computing the chromatic number [CFM17], or as a lower-bound procedure inside the Dsatur-based branch-and-bound algorithm of [FGT17]. Here, an approach using the auxiliary graph for preprocessing of the graph coloring instance is proposed.

The idea is to apply a set of preprocessing rules for the maximum independent set problem to the auxiliary graph G_A and translate the reductions back to the original graph G. In the context of maximum independents sets, preprocessing rules determine vertices that can be removed from the graph or always inserted into an independent set without changing the optimal value of the instance. For example, if one can determine that there exists an optimal solution not containing the vertex v, it can be removed, and the optimal solution still exists. Likewise, if it is determined that there is an optimal solution that contains v, it can be fixed to be in the independent set, and an optimal solution still exists. The size of the MIS isntance is thus reduced without removing all optimal solutions. Translating this to graph coloring on G, vertices are merged or edges added without removing all optimal solutions.

Applying the preprocessing rules to the auxiliary graph G_A, a list of vertices to be fixed in the stable set, and a list of vertices to be removed is computed. Due to the correspondence described in Section 2.1.3, the following reductions hold for the graph coloring problem on G.

Lemma 2.3.5. *Vertices in G_A that were fixed in the stable set by the preprocessing correspond to two vertices that are assigned the same color; they can be merged in the graph coloring instance.*

Lemma 2.3.6. *Vertices in G_A that were removed by the preprocessing correspond to two vertices that are colored differently; the edge between them can be added in the graph coloring instance.*

These reductions appear similar to Lemmas 2.3.3 and 2.3.4 as again vertices are merged or edges added, but they do not require the assumption of looking for a specific k-coloring, or a subgraph of chromatic number k.

Akiba and Iwata [AI16] describe and use several effective reduction rules to implement an efficient branch-and-reduce algorithm to solve the minimum vertex cover problem, but the reductions can also be applied for the maximum independent set problem. These reductions were built upon and extended to the weighted version of the problem in [Lam+19; Xia+21]. The reductions are usually part of a branch-and-reduce algorithm to solve the problem exactly, but they can also be applied to only the root without branching to serve as preprocessing. The authors of [Xia+21] presented a very efficient implementation and made their source code available[2]. Their algorithm is used in Section 2.4 to compute the maximum stable set preprocessing on G_A. The application of MIS preprocessing to the graph coloring problem is new, and their effectiveness is evaluated in Section 2.4. It is also referred to as MIS-based preprocessing.

As Cornaz, Furini, and Malaguti [CFM17] report significantly improved results when using the max degree vertex ordering for the orientation of the complement graph, this is done for the auxiliary graph-based preprocessing as well. The authors experimentally observed the auxiliary graph obtained with this ordering to contain larger cliques, making the maximum independent set problem easier for MIP solvers. In this work, this is extended to first compute a large clique and put its vertices at the beginning of the vertex ordering.

2.4 Experimental Evaluation

The evaluation of this chapter focuses on the effectiveness of Algorithm 2.4 as a whole on the instances of the benchmark set and highlights where the Mycielsky bound is successful over the clique bound. The individual performance of the presented lower and upper bound algorithms is not discussed, only how effective they are in conjunction with the preprocessing, i.e., the results are the maximum resp. minimum of bounds computed during Algorithm 2.4. Further, the success and strengths of the new auxiliary graph-based preprocessing in simplifying the instances are analyzed.

To begin, the results of the preprocessing routine of Algorithm 2.4 are given in Table 2.1. It lists the instance name and basic data $|V|, |E|$, and the effectiveness of the clique bound *lb-c*, mycielsky bound *lb-m*, and the dsatur heuristic bound *ub*. The last three columns report on the success of the reduction routine, with the number of remaining vertices n' and edges m' after reduction, and the percentage by how much the number of vertices were reduced. Bold entries for the lower and upper bounds mean they match, and the instance was solved in preprocessing. For instances that were reduced completely, the last three columns are highlighted in bold as well. Instances where the initial bounds did not solve the instance and no reduction was achieved are gray.

[2] https://github.com/ployts/MWIS/

Table 2.1: Preprocessing performance

Instance	n	m	d	*lb-c*	*lb-m*	*ub*	n'	m'	Red
1-FullIns_3	30	100	0.23	3	**4**	**4**	**0**	**0**	**100**
1-FullIns_4	93	593	0.14	3	**5**	**5**	25	84	73.1
1-FullIns_5	282	3247	0.08	3	**6**	**6**	57	325	79.8
1-Insertions_4	67	232	0.10	2	2	5	67	232	0.0
1-Insertions_5	202	1227	0.06	2	2	6	202	1227	0.0
1-Insertions_6	607	6337	0.03	2	2	7	607	6337	0.0
2-FullIns_3	52	201	0.15	4	**5**	**5**	9	22	82.7
2-FullIns_4	212	1621	0.07	4	5	6	31	124	85.4
2-FullIns_5	852	12201	0.03	4	6	7	76	489	91.1
2-Insertions_3	37	72	0.11	2	2	4	37	72	0.0
2-Insertions_4	149	541	0.05	2	2	5	149	541	0.0
2-Insertions_5	597	3936	0.02	2	2	6	597	3936	0.0
3-FullIns_3	80	346	0.11	5	**6**	**6**	11	35	86.2
3-FullIns_4	405	3524	0.04	5	5	7	43	199	89.4
3-FullIns_5	2030	33751	0.02	5	6	8	91	672	95.5
3-Insertions_3	56	110	0.07	2	2	4	56	110	0.0
3-Insertions_4	281	1046	0.03	2	2	5	281	1046	0.0
3-Insertions_5	1406	9695	0.01	2	2	6	1406	9695	0.0
4-FullIns_3	114	541	0.08	6	**7**	**7**	13	51	88.6
4-FullIns_4	690	6650	0.03	6	**8**	**8**	37	216	94.6
4-FullIns_5	4146	77305	0.01	6	6	9	113	926	97.3
4-Insertions_3	79	156	0.05	2	2	4	79	156	0.0
4-Insertions_4	475	1795	0.02	2	2	5	475	1795	0.0
5-FullIns_3	154	792	0.07	7	**8**	**8**	15	70	90.3
5-FullIns_4	1085	11395	0.02	7	**9**	**9**	43	294	96.0
C2000.5	2000	999836	0.50	14	12	206	2000	999836	0.0
C2000.9	2000	1799532	0.90	74	60	530	2000	1799532	0.0
C4000.5	4000	4000268	0.50	15	13	376	4000	4000268	0.0
DSJC1000.1	1000	49629	0.10	6	5	28	1000	49629	0.0
DSJC1000.5	1000	249826	0.50	14	10	116	1000	249826	0.0
DSJC1000.9	1000	449449	0.90	66	54	294	1000	449449	0.0
DSJC125.1	125	736	0.09	4	3	6	125	736	0.0
DSJC125.5	125	3891	0.50	10	9	21	125	3891	0.0
DSJC125.9	125	6961	0.90	34	31	50	125	6961	0.0
DSJC250.1	250	3218	0.10	4	4	10	250	3218	0.0
DSJC250.5	250	15668	0.50	12	8	39	250	15668	0.0
DSJC250.9	250	27897	0.90	43	33	89	250	27897	0.0
DSJC500.1	500	12458	0.10	5	4	16	500	12458	0.0
DSJC500.5	500	62624	0.50	13	10	66	500	62624	0.0
DSJC500.9	500	112437	0.90	56	41	163	500	112437	0.0
DSJR500.1	500	3555	0.03	**12**	9	**12**	**0**	**0**	**100**
DSJR500.1c	500	121275	0.97	83	77	87	289	40442	42.2
DSJR500.5	500	58862	0.47	122	117	125	486	57251	2.8
abb313GPIA	1557	53356	0.04	8	7	10	853	15858	45.2
anna	138	493	0.05	**11**	7	**11**	**0**	**0**	**100**
ash331GPIA	662	4181	0.02	3	3	5	661	4180	0.2

Table 2.1: (continued)

Instance	n	m	d	lb-c	lb-m	ub	n'	m'	Red
ash608GPIA	1216	7844	0.01	3	3	5	1215	7843	0.1
ash958GPIA	1916	12506	0.01	3	3	5	1915	12505	0.1
david	87	406	0.11	**11**	7	**11**	**0**	**0**	**100**
flat1000_50_0	1000	245000	0.49	14	11	117	1000	245000	0.0
flat1000_60_0	1000	245830	0.49	14	11	114	1000	245830	0.0
flat1000_76_0	1000	246708	0.49	14	11	114	1000	246708	0.0
flat300_20_0	300	21375	0.48	11	9	43	300	21375	0.0
flat300_26_0	300	21633	0.48	11	8	42	300	21633	0.0
flat300_28_0	300	21695	0.48	12	7	44	300	21695	0.0
fpsol2.i.1	496	11654	0.09	**65**	27	**65**	**0**	**0**	**100**
fpsol2.i.2	451	8691	0.09	**30**	27	**30**	86	1926	80.9
fpsol2.i.3	425	8688	0.10	**30**	27	**30**	86	1926	79.8
games120	120	638	0.09	**9**	5	**9**	**0**	**0**	**100**
homer	561	1628	0.01	**13**	11	**13**	**0**	**0**	**100**
huck	74	301	0.11	**11**	7	**11**	**0**	**0**	**100**
inithx.i.1	864	18707	0.05	**54**	41	**54**	95	3330	89.0
inithx.i.2	645	13979	0.07	**31**	**31**	**31**	124	2760	80.8
inithx.i.3	621	13969	0.07	**31**	**31**	**31**	124	2760	80.0
jean	80	254	0.08	**10**	4	**10**	**0**	**0**	**100**
latin_square_10	900	307350	0.76	90	90	137	900	307350	0.0
le450_15a	450	8168	0.08	15	5	17	407	7802	9.6
le450_15b	450	8169	0.08	15	5	16	410	7824	8.9
le450_15c	450	16680	0.17	15	15	26	450	16680	0.0
le450_15d	450	16750	0.17	15	6	26	450	16750	0.0
le450_25a	450	8260	0.08	**25**	**25**	**25**	264	5831	41.3
le450_25b	450	8263	0.08	**25**	**25**	**25**	294	6240	34.7
le450_25c	450	17343	0.17	25	25	30	435	17096	3.3
le450_25d	450	17425	0.17	25	25	29	433	17106	3.8
le450_5a	450	5714	0.06	5	4	10	450	5714	0.0
le450_5b	450	5734	0.06	5	4	9	450	5734	0.0
le450_5c	450	9803	0.10	5	5	6	450	9803	0.0
le450_5d	450	9757	0.10	5	4	6	450	9757	0.0
miles1000	128	3216	0.40	**42**	29	**42**	**0**	**0**	**100**
miles1500	128	5198	0.64	**73**	**73**	**73**	**0**	**0**	**100**
miles250	128	387	0.05	**8**	4	**8**	**0**	**0**	**100**
miles500	128	1170	0.14	**20**	9	**20**	**0**	**0**	**100**
miles750	128	2113	0.26	**31**	11	**31**	**0**	**0**	**100**
mug100_1	100	166	0.03	3	3	4	100	166	0.0
mug100_25	100	166	0.03	3	3	4	100	166	0.0
mug88_1	88	146	0.04	3	2	4	88	146	0.0
mug88_25	88	146	0.04	3	3	4	88	146	0.0
mulsol.i.1	197	3925	0.20	**49**	33	**49**	**0**	**0**	**100**
mulsol.i.2	188	3885	0.22	**31**	**31**	**31**	71	1264	62.2
mulsol.i.3	184	3916	0.23	**31**	**31**	**31**	71	1264	61.4
mulsol.i.4	185	3946	0.23	**31**	**31**	**31**	73	1325	60.5
mulsol.i.5	186	3973	0.23	**31**	**31**	**31**	72	1293	61.3

Table 2.1: (continued)

Instance	n	m	d	lb-c	lb-m	ub	n'	m'	Red
myciel3	11	20	0.36	2	**4**	**4**	**0**	**0**	**100**
myciel4	23	71	0.28	2	**5**	**5**	17	46	26.1
myciel5	47	236	0.22	2	**6**	**6**	42	211	10.6
myciel6	95	755	0.17	2	**7**	**7**	90	725	5.3
myciel7	191	2360	0.13	2	**8**	**8**	186	2325	2.6
qg.order100	10000	990000	0.02	100	100	102	10000	990000	0.0
qg.order30	900	26100	0.06	30	30	32	900	26100	0.0
qg.order40	1600	62400	0.05	40	40	41	1600	62400	0.0
qg.order60	3600	212400	0.03	60	60	62	3600	212400	0.0
queen10_10	100	1470	0.30	10	10	13	100	1470	0.0
queen11_11	121	1980	0.27	11	11	15	121	1980	0.0
queen12_12	144	2596	0.25	12	12	15	144	2596	0.0
queen13_13	169	3328	0.23	13	13	17	169	3328	0.0
queen14_14	196	4186	0.22	14	14	18	196	4186	0.0
queen15_15	225	5180	0.21	15	15	19	225	5180	0.0
queen16_16	256	6320	0.19	16	16	21	256	6320	0.0
queen5_5	25	160	0.53	**5**	**5**	**5**	25	160	0.0
queen6_6	36	290	0.46	6	6	8	36	290	0.0
queen7_7	49	476	0.40	7	7	9	49	476	0.0
queen8_12	96	1368	0.30	12	12	13	96	1368	0.0
queen8_8	64	728	0.36	8	8	11	64	728	0.0
queen9_9	81	1056	0.33	9	9	12	81	1056	0.0
r1000.1	1000	14378	0.03	**20**	**20**	**20**	46	651	95.4
r1000.1c	1000	485090	0.97	92	87	104	686	227525	31.4
r1000.5	1000	238267	0.48	234	216	239	966	230416	3.4
r125.1	125	209	0.03	**5**	**5**	**5**	**0**	**0**	**100**
r125.1c	125	7501	0.97	**46**	**46**	**46**	**0**	**0**	**100**
r125.5	125	3838	0.50	36	36	37	109	3323	12.8
r250.1	250	867	0.03	**8**	5	**8**	**0**	**0**	**100**
r250.1c	250	30227	0.97	**64**	**64**	**64**	68	2270	72.8
r250.5	250	14849	0.48	65	57	67	235	13968	6.0
school1	385	19095	0.26	**14**	10	**14**	355	18934	7.8
school1_nsh	352	14612	0.24	14	10	15	326	14493	7.4
wap01a	2368	110871	0.04	41	40	48	1760	91229	25.7
wap02a	2464	111742	0.04	40	40	45	1698	86519	31.1
wap03a	4730	286722	0.03	40	40	52	4090	263417	13.5
wap04a	5231	294902	0.02	40	40	49	4032	251649	22.9
wap05a	905	43081	0.11	**50**	39	**50**	665	34684	26.5
wap06a	947	43571	0.10	40	40	43	699	35378	26.2
wap07a	1809	103368	0.06	40	37	44	1605	96780	11.3
wap08a	1870	104176	0.06	40	31	44	1598	95118	14.5
will199GPIA	701	6772	0.03	6	4	7	660	5836	5.8
zeroin.i.1	211	4100	0.19	**49**	46	**49**	**0**	**0**	**100**
zeroin.i.2	211	3541	0.16	**30**	28	**30**	**0**	**0**	**100**
zeroin.i.3	206	3540	0.17	**30**	28	**30**	**0**	**0**	**100**

The preprocessing alone can solve 50 out of 137 instances, all but three of them in less than 0.25 seconds, the other three in less than 1.2 seconds, and preprocessing over all benchmark instances takes up an average of 2.8 seconds . It should be highlighted that the success is an interplay between all three parts presented in this chapter. For example, for the dense instance r1000.1c with 1000 vertices, CliSat finds an initial clique of size 91 within the time limit of 1 second. This is not increased by the Mycielsky bound, but after applying the preprocessing rules, the graph is reduced to 686 vertices, and a second call to CliSAT on this reduced graph produces a clique of size 92. While r1000.1c is not solved in preprocessing, it shows that the performance of each part is connected to the others. Another interesting example is instances 5-FullIns_4 with 1083 vertices. Initially, the Mycielsky bound is 5 which does not improve on the optimal clique bound of 7, but the graph is reduced to only 43 vertices. On this graph, the Mycielsky bound is computed to be 9, proven to be optimal by the Dsatur heuristic that finds a 9-coloring. This highlights that neither the lower bound, the preprocessing, nor the coloring heuristic alone are enough to solve some instances.

Reviewing the Mycielsky bound, it improves on the clique bound on 17 out of the 137 instances, 14 of which are solved in preprocessing due to the bound improvement. The clique bound is better than the Mycielsky bound for 59 graphs, so it should not be used as the only bound and more so to potentially improve on other bounds. It is interesting to list on which instances the Mycielsky bound has an advantage: all Mycielskians M_4 to M_8 are solved. Note that instances mycielk of the Dimacs benchmark corresponds to M_{k+1}. This is good, as the algorithm should be able to at least recognize Mycielskians themselves. Further, the algorithm is very successful on the FullIns instances: all but 3-FullIns_4, 4-FullIns_5 and 2-FullIns_4, 2-FullIns_5, 3-FullIns_5 are solved in preprocessing due to the Mycielsky bound, and the bound is still increased from the clique bound for the last three instances. The FullIns and Insertion graphs are created by taking a mycielsky graph and adding vertices and edges to increase size but not density. As such, they are, or contain, pseudo Mycielskians, which explains why the Mycielsky bound is successful on the FullIns instances but not why it is unsuccessful for the Insertion instances. Since detailed information on how these instances are built is not readily available from the literature, this question was not investigated further. An answer might help improve the choices in Algorithm 2.2 and yield an algorithm finding improved lower bounds on the Insertion graphs too. Unfortunately, the Mycielsky bound in the current implementation is not able to improve on the clique bound in cases where the graph is not connected to pseudo Mycielskians, at least on the DIMACS benchmark set. In the user propagators presented in Chapter 4, Algorithm 2.2 will be used and evaluated as a lower bound procedure during the search process of the propagator; the details are given in Section 4.4.

Regarding the preprocessing rules removing vertices, 77 out of the 137 instances had some success, ranging from as few as a single vertex to removing every vertex, which happens for 17 instances. On average, 34% of vertices are removed, which increases to 60.5% when considering only instances that had at least some reduction. Reducing the instance size by about a third on average also means reducing the encoding size by about a third, depending on the encoding. That this reduced size can be beneficial was already seen with CliSAT finding a larger clique on r1000.1c after removing vertices.

Next, the effective reduction percentage of the MIS-based preprocessing is analyzed. This is presented in Table 2.2, similarly as the reduction percentages in Table 2.1, though the reduction there was measured in terms of removed vertices, and here only instances where the reduction had an effect are shown. Since the auxiliary graph only reports decisions for two non-adjacent vertices u, v, the reduction is either to merge two vertices, which also means removing one, or to add an edge between u and v. This does not reduce the

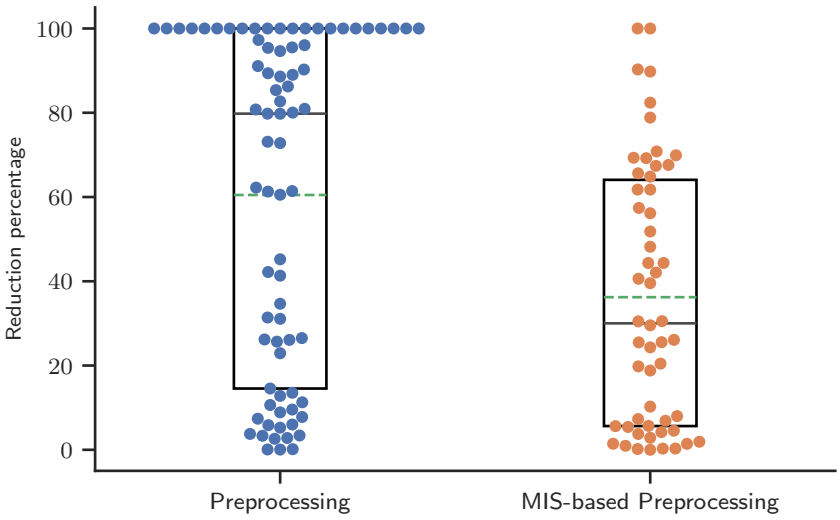


Figure 2.2: Reduction percentages of the two preprocessing methods

size of the graph but introduces a new constraint. For this reason, the reduction here is measured in terms of how many decisions were fixed in preprocessing, out of the $|\overline{E}|$ pairs of non-adjacent vertices.

The construction of the auxiliary graph and computing of the MIS preprocessing were given a time limit of 900 seconds, and a memory limit of 20 GB. This preprocessing reduces the problem size on 53 instances for an average of 14.3%, increasing to 36.9% when considering only instances where the reduction had some effect. It is only able to completely solve the two instances r125.1c and r250.1c. This exemplifies how the chromatic number can be determined from solving the MIS problem exactly: for r125.1c, 79 vertices were inserted into the maximum independent set, yielding an optimal coloring using $125 - 79 = 46$ colors.

A visual comparison of the reduction performances for the two methods is given in Figure 2.2. Instances where the reduction was unsuccessful are left out of both the swarmplot and the boxplot, the middle black line of the boxplot represents the median, and the green line is the average.

One thing to remark is that neither of the reductions is strictly stronger than the other. While the normal preprocessing generally reports higher reductions and on more instances, the MIS-based preprocessing completely reduces instance r250.1c, though the normal preprocessing routine also solves this instance with matching lower and upper bounds. On r250.5, a slightly higher reduction is achieved as well. But empirically, the preprocessing of Algorithm 2.4 clearly outperforms the MIS-based preprocessing, both in number of instances completely reduced and average reduction. Additionally, the MIS-based preprocessing incurs quite a large overhead: the auxiliary graph has $\overline{m}$ vertices and $O(n^3)$ many vertices. To compute the preprocessing result and even build the graph, the algorithm ran into the time limit two times, and the memory limit 18 times. As such, the MIS-based preprocessing seems to not be of use in practice, and the normal preprocessing routine is preferred. One advantage, however, is that no lower bound needs to be computed for this reduction.

Since neither preprocessing strictly implies the other, one can also combine the two preprocessing methods. When running the MIS-based preprocessing on the reduced graph of the normal preprocessing routine, this only shows a negligible effect. Only six instances yield a further reduction, but of at most 1 contraction of vertices and at most 18 edges added, leading to no significant gain in combining the methods this way.

Table 2.2: MIS-based preprocessing performance

Instance	n	$\overline{m}$	inserted	removed	Red
1-FullIns_3	30	335	1	98	29.6
1-FullIns_4	93	3685	6	1490	40.6
1-FullIns_5	282	36374	36	16092	44.3
2-FullIns_3	52	1125	2	497	44.4
2-FullIns_4	212	20745	19	10730	51.8
2-FullIns_5	852	350325	58	201036	57.4
3-FullIns_3	80	2814	3	1577	56.1
3-FullIns_4	405	78286	23	48326	61.8
4-FullIns_3	114	5900	4	3822	64.8
4-FullIns_4	690	231055	28	163592	70.8
5-FullIns_3	154	10989	5	7677	69.9
DSJR500.1	500	121195	0	8	0.0
DSJR500.1c	500	3475	37	651	19.8
DSJR500.5	500	65888	0	1262	1.9
anna	138	8960	55	7989	89.8
ash331GPIA	662	214610	0	658	0.3
david	87	3335	16	2044	61.8
fpsol2.i.1	496	111106	227	87389	78.9
fpsol2.i.2	451	92784	95	44624	48.2
fpsol2.i.3	425	81412	69	32126	39.5
homer	561	155452	219	127870	82.4
huck	74	2400	11	999	42.1
inithx.i.1	864	354109	345	245137	69.3
inithx.i.2	645	193711	87	59058	30.5
inithx.i.3	621	178541	62	43345	24.3
jean	80	2906	10	1955	67.6
le450_15a	450	92857	0	883	1.0
le450_15b	450	92856	1	1332	1.4
le450_25a	450	92765	1	3487	3.8
le450_25b	450	92762	0	1317	1.4
miles1000	128	4912	0	279	5.7
miles1500	128	2930	0	123	4.2
miles250	128	7741	11	1573	20.5
miles500	128	6958	0	13	0.2
miles750	128	6015	0	273	4.5
mulsol.i.1	197	15381	59	10035	65.6
mulsol.i.2	188	13693	15	4159	30.5
mulsol.i.3	184	12920	10	3284	25.5
mulsol.i.4	185	13074	10	3329	25.5
mulsol.i.5	186	13232	10	2480	18.8
myciel3	11	35	0	1	2.9
r1000.1c	1000	14410	10	1467	10.2
r1000.5	1000	261233	0	750	0.3
r125.1	125	7541	3	1965	26.1
r125.1c	125	249	**79**	**170**	**100**
r125.5	125	3912	0	212	5.4

Table 2.2: (continued)

Instance	n	$\overline{m}$	inserted	removed	Red
r250.1	250	30258	0	1701	5.6
r250.1c	250	898	**186**	**712**	**100**
r250.5	250	16276	5	1296	8.0
school1	385	54825	8	3769	6.9
school1_nsh	352	47164	8	3431	7.3
zeroin.i.1	211	18055	101	16199	90.3
zeroin.i.2	211	18614	58	12826	69.2
zeroin.i.3	206	17575	53	11791	67.4

Chapter 3

Satisfiability Encodings of the Graph Coloring Problem

In Chapter 1, the graph coloring problem and satisfiability methods were introduced. This chapter presents satisfiability encodings for the graph coloring problems, which can then be used to compute the chromatic number with a SAT solver. Since SAT solvers only produce a yes or no answer in the form of SAT or UNSAT, these encodings and their solution represent whether a k-coloring exists for a specific natural number k. For this thesis, the encodings can be separated into two categories:

- **Direct Encodings:** the number of colors allowed is directly encoded in the number of variables, and the variables themselves represent a coloring.
- **Zykov-based Encodings:** the variables represent whether two vertices are assigned the same color or not; there is no explicit coloring. Though, one can be extracted from this information. Additionally, the number of allowed colors needs to be separately encoded as clauses with methods for at-most-k constraint.

Further, direct encodings generally have one or two variables for each vertex in the graph and each allowed color, leading to $O(k \cdot n)$ variables for the k-coloring problem of a graph. Zykov-based encodings have a variable for each pair of non-adjacent vertices in the graph, and auxiliary variables to encode the number of colors allowed plus the respective at-most-k constraints. This thesis's focus will be on different approaches using the Zykov-based encodings, but also presents the simpler direct encodings for comparison in Section 3.1.

The strength of the direct encodings partially lie in their compact formulation, especially for sparse graph. Their downside is that they work with explicit colorings, which introduces many symmetries to the problem coming from color permutations. This can be addressed by adding symmetry-breaking constraints that cut off many of these permuted colorings. However, while these can cut off all but one optimal solution, they can over-constrain the problem and again make it harder to solve. Zykov-based encodings, on the other hand, do not work with explicit colorings but instead with "color-relations". As will be discussed in Section 3.2.1, this avoids all symmetries coming from color permutations at the cost of the generally larger encoding size, especially for sparse graphs.

Glorian et al. presented an incremental solving approach using the Zykov-based encoding in [Glo+19], described in detail in Section 3.2.2. In the experimental evaluation of their algorithm, the authors report exceptional results, solving several open instances of the DIMACS benchmark set and beating other state-of-the-art solvers. Unfortunately, the results are not repeatable by the authors or by others since the source code is not available anymore[1]. For that reason, one major motivation for this chapter is to address the issue of replicating these exceptional results. Since this cannot be done by repeating the experiments with the original source code, the algorithm is reproduced in a new implementation to verify the results.

[1]The incorrect source code is published at https://github.com/Mystelven/picasso

3.1 Direct Encodings

Two types of encodings which are classified as direct encodings are presented:

- **Assignment Encoding:** assignment variables $x_{v,i}$ represent whether vertex $v \in V$ is assigned color $i \in \{1, \dots, k\}$.
- **Partial Order Encoding:** ordering variables $y_{v,i}$ represent whether vertex $v \in V$ is assigned a color larger than i, assuming the natural order on $\{1, \dots, k\}$.

3.1.1 Assignment Encoding

An intuitive way to formulate the k-coloring problem as a satisfiability instance is to use variables to represent the assignment of a color to a vertex. Given a graph G and a number $k \in \mathbb{N}$, variables $x_{v,i}$ for each $v \in V$ and $i \in \{1, \dots, k\}$ are created. If $x_{v,i} = \text{True}$, vertex v is assigned color i. Definition 3.1.1 gives the full encoding with clauses to ensure that the assignment is a valid coloring.

Definition 3.1.1 (Assignment Encoding)**.** Given a graph $G = (V, E)$ and a $k \in \mathbb{N}$, the assignment encoding of k-colorability is given by the following clauses.

$$\bigvee_{i=1}^{k} x_{v,i} \qquad v \in V \tag{3.1}$$

$$\overline{x}_{v,i} \vee \overline{x}_{w,i} \qquad \{v, w\} \in E, i \in \{1, \dots, k\} \tag{3.2}$$

$$\overline{x}_{v,i} \vee \overline{x}_{v,j} \qquad v \in V, 1 \leq i < j \leq k \tag{3.3}$$

Clauses (3.1) guarantee that at least one $x_{v,i}$ for a fixed v is true so that each vertex is assigned a color. With (3.2), $x_{v,i}$ and $x_{w,i}$ cannot both be true, which asserts that adjacent vertices get assigned different colors. Finally, (3.3) constrain each vertex to be assigned at most one color by a simple pairwise encoding of the at-most-one constraint.

Remark 3.1.2. Clauses (3.3) are sometimes omitted in practice, since the requirement of each vertex only being assigned one color is not strictly necessary; a valid coloring can be obtained by simply choosing one of the assigned colors. Leaving them out works well for local search methods for satisfiability, as the problem is less constrained [HKH22]. They can help in eliminating some symmetric solutions, though, and are beneficial for proving non-colorability of some instances [YWH20].

The obtained encoding is quite compact, especially for sparser graphs, with $k \cdot n$ variables, $n + k \cdot m$ clauses, and $k \cdot n + 2k \cdot m$ literals in total when omitting the clauses for the at-most-one color constraint.

One disadvantage of this compact formulation is that it works with explicit colorings. Even though it only has a polynomial number of variables and clauses, a factorial set of symmetric solutions is generated by the many coloring permutations: each coloring with k colors generates $k!$ permutations that are again valid solutions. This is not only a problem for the SAT case when there exist solutions but also in the UNSAT case. To fully prove there exists no valid coloring, it is necessary to disprove every permutation: if the assignment of $x_{v,i}$ is proven not to lead to a satisfying assignment, this still needs to be done for other permutations $\pi \in S_k$ and $x_{v,\pi(i)}$. Different additional constraints have been proposed to break some or all the symmetries [Van08; HKH22; MZ08], at the cost of making the problem more constrained and potentially harder to solve.

In this thesis, only simple symmetry-breaking constraints are discussed. One of them is based on the observation that vertex v can be assigned one of the colors in $\{1, \dots, v\}$ and any colorings that would assign v a larger color can be excluded.

$$\overline{x}_{v,i} \qquad \forall v < i, i \in \{1, \dots, k\} \tag{3.4}$$

Another efficient and simple way to break some symmetries, requiring no changes to the formulation, is to fix the color of some vertices and thus remove any permutation including those colors. Given a large clique $C = \{v_1, \dots, v_s\}$, one can assign a different color to each clique vertex by setting $x_{v_i,i} = \text{True}$ for $i = 1, \dots, s$. Since all vertices in a clique need to be colored differently, this does not cut off all optimal colorings, but many permutations are removed. This has a significant impact if $|C|$ is close to k since by this simple method, only $(k - |C|)!$ color permutations remain. On has to take care to not add Clause (3.4) for vertices v that were part of the clique and fixed.

A second disadvantage of the assignment model is the asymmetric impact of decision values in the search tree: while $x_{v,i} = \text{True}$ means a new color assignment to v and forbidding color i for all neighbors of v, setting $x_{v,i} = \text{False}$ only changes the problem very slightly. In a branch-and-bound approach, this can lead to an unbalanced search tree, but it can also be problematic since learnt conflict clauses of the SAT solver might be weaker. Conflicts mostly arise from color assignments, and thus their conflict clause contains literals forbidding a color which might weaken the possible strength of the learned clause.

A more systematic way to avoid many of the symmetries previously mentioned and avoid the asymmetric decisions of the assignment encoding is given by the partial order encoding in the next section.

3.1.2 Partial Order Encoding

Instead of variables directly assigning a color to a vertex, the partial order encoding uses variables $y_{v,i}$ indicating a vertex v using any color strictly greater than i. For this purpose, the canonical ordering on the integers is assumed for the colors. The color of a vertex can be concluded as the color i such that $y_{v,i-1} = \text{True}$ and $y_{v,i} = \text{False}$, i.e., v has color strictly greater than $i-1$ but not greater than i. Thus a color assignment can be extracted from these variables; if it satisfies the clauses from Definition 3.1.3 it is a valid k-coloring. The partial order encoding for the graph coloring problem was first introduced in [JM18] for integer linear programming formulations but later adapted to different satisfiability encodings in [Kau20] and more thoroughly in [FJM24]. The presentation here follows the one in the latter source.

Definition 3.1.3 (Partial Order Encoding)**.** Given a graph $G = (V, E)$ and a $k \in \mathbb{N}$, the partial order encoding of k-colorability is given by the following clauses.

$$\overline{y}_{v,k} \qquad v \in V \tag{3.5}$$

$$y_{v,i} \vee \overline{y}_{v,i+1} \qquad v \in V, i \in \{1, \dots, k-1\} \tag{3.6}$$

$$y_{v,1} \vee y_{w,1} \qquad \{v, w\} \in E \tag{3.7}$$

$$\overline{y}_{v,i} \vee y_{v,i+1} \vee \overline{y}_{w,i} \vee y_{w,i+1} \qquad \{v, w\} \in E, i \in \{1, \dots, k-1\} \tag{3.8}$$

Clauses (3.5) say no vertex can have color larger than k, while Clauses (3.6) assert that if v has color larger than $i+1$, it must also have color larger than i, i.e., $y_{v,i+1} \Rightarrow y_{v,i}$. Clauses (3.7) and (3.8) ensure that adjacent vertices are assigned different colors so that a valid coloring is obtained.

Like the assignment encoding from the previous section, the partial order encoding has $k \cdot n$ variables, although Clauses (3.5) directly assigns n variables to false which could be removed from the formulation. It also has $k \cdot n + k \cdot m$ clauses, a bit more than in the assignment encoding, but the clauses only have at most four literals compared to the clauses with up to k literals.

A problem in the assignment encoding was that assigning $x_{v,i} = \text{False}$ gave almost no new information. The vertex could not be assigned this color anymore, but this is not restrictive at all considering all permutations of colorings are still valid. The decisions in the partial order encoding are more informative: setting $y_{v,i} = \text{True}$ implies $y_{v,j} = \text{True}$ for $j < i$ and $y_{v,i} = \text{False}$ implies $y_{v,j} = \text{False}$ for $j > i$ because of Clause (3.6).

The simple symmetry-breaking constraints from Equation (3.4) easily translate to the partial order encoding:

$$\overline{y}_{i,i} \qquad\qquad i \in \{1, \ldots, k\} \tag{3.9}$$

Again, a clique can be used to fix some variables in the beginning and further decrease the size of the search space. Given a large clique $C = v_1, \ldots, v_s$, setting $y_{v_i,i-1} = \text{True}$ and $y_{v_i,i} = \text{False}$ for $i = 1, \ldots, s$ will assign color i to vertex v_i. Likewise, one cannot add Clause (3.9) for vertices v that were fixed as part of the clique.

One can combine the two previous methods of encoding the graph coloring problem to potentially benefit from both, obtaining a hybrid formulation that includes both assignment variables $x_{v,i}$ and partial order variables $y_{v,i}$ [Kau20; FJM24].

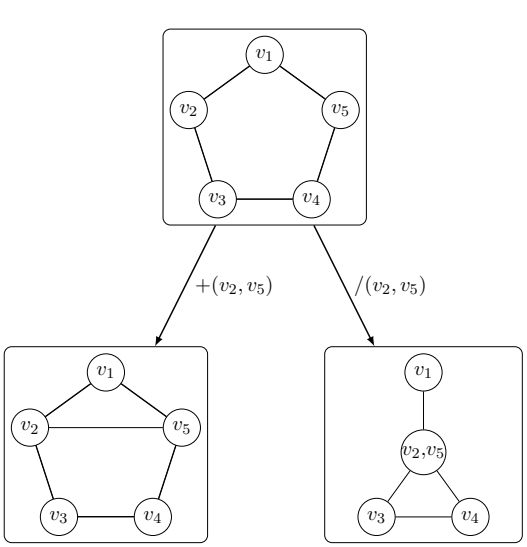
3.2 Zykov-Based Encodings

The encodings of this section are based on the Zykov recurrence relation of Equation (3.10), described in [Zyk49]. Given two non-adjacent vertices u, v of the graph, $G/(uv)$ is obtained by merging the two vertices, and $G + (uv)$ is the graph obtained by adding the edge $\{u, v\}$.

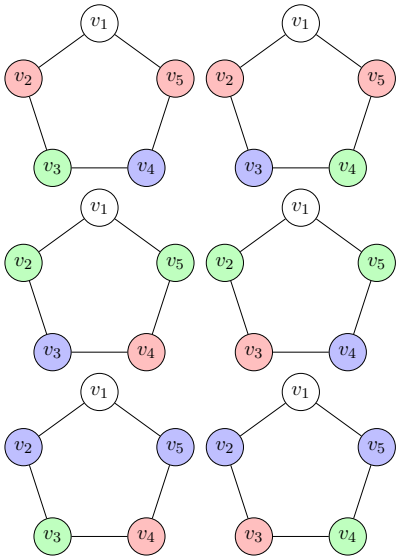
$$\chi(G) = \min\{\chi(G/(u, v), \chi(G + (uv)))\} \tag{3.10}$$

The recurrence formalizes the observation that in any coloring, two non-adjacent vertices are either assigned the same color, or different ones. One can solve the two graph coloring problems created by forcing them to have the same color, i.e., by merging them, or by separating two vertices,i.e., adding an edge. Any coloring of the subproblems corresponds to a coloring of the original graph, and a minimal coloring can be found as the smallest coloring among the two subproblems. Further, this recurrence can be used to build a binary tree of graph coloring problems, called a Zykov tree. Figure 3.1(a) gives the root and first level of the tree for the 5-cycle C_5. The interesting property of this tree is that its leaf nodes are complete graphs, and that the chromatic number can be found as the size of the smallest complete graph among the leaves. The former is true because the recurrence stops only when no more non-adjacent vertices exist. By the decisions of separating or merging non-adjacent vertices made to reach the leaf node, one can recover a coloring of G from a coloring of the leaf. To see that an optimal coloring can be found among the leaves, one can take any optimal coloring and produce such a leaf by merging all vertices of the same color-class and adding edges between vertices of different colors; this produces a complete graph of size $\chi(G)$.

One immediate benefit of algorithms based on Equation (3.10) is that they do not work with explicit colorings. Instead of assignments of a color to a vertex, only relationships between two vertices are considered; either two vertices have the same color or must have



(a) Root and child nodes in the Zykov tree of C_5.



(b) Permutations of partial coloring that all corresponding to the same node of (a).

Figure 3.1: (a) Partial Zykov tree and (b) partial colorings all leading to the bottom right node of (a) by contracting vertices of the same color.

different ones. As such, the search space traversed by Zykov-based branch-and-bound algorithms, or the satisfiability approach presented next, is smaller than in an approach that branches on colors, such as the direct encoding from Section 3.1. One way to see this is to observe that any (partial) coloring via assignments corresponds to a node in the Zykov tree: merge all vertices that were assigned the same color and add edges between all vertices that were assigned different colors. Any permutation of that coloring will correspond to that same node in the tree. Quantitative results in terms of nodes traversed and tree size will be discussed in the experimental evaluation of Chapter 4, as the external propagators will provide access to these statistics. The effect can already be observed in the example of Figure 3.1(b).

A branch-and-bound algorithm using this recurrence is described in the survey [LC18, Section 3.2], and a worst case analysis of the tree size using cliques as lower bounds is given in the paper by McDiarmid [McD79]. This recurrence is also commonly used as the branching scheme in column generation approaches of the graph coloring problem [MT96; HCS12]. Since the subproblems are again coloring problems, this leads to a more balanced branch-and-bound tree than branching on the independent set variables of their formulation. An interesting approach is the one of Schaafsma, Heule, and Van Maaren [SHV09]: they work with the assignment model from Section 3.1.1 but add edge variables to learn stronger clauses during the CDCL loop. When encountering a conflict containing assignment variables, they encode all symmetric clauses of the coloring conflict only using edge variables. This way, stronger conflict clauses can be learnt that cover all coloring permutations while still being able to branch on the assignment variables.

3.2.1 Description of the Full Zykov Encoding

This image of the Zykov tree and the colorings found among the nodes will be helpful in understanding the satisfiability encoding presented next. In the recurrence, the branching is done on any two non-adjacent vertices in G and exactly those non-edges will be the

decision variables in the encoding. That is, for each non-edge $\{i, j\} \notin E$, define boolean variables s_{ij} where $s_{ij} = \text{True}$ means i and j are contracted and receive the same color. Likewise, when set to False, this means i and j are assigned different colors corresponding to adding an edge. Since only undirected graphs are considered, only s_{ij} for $i < j$ are important and any s_{ij} with $i > j$ is treated as a renaming of s_{ij}. Variables s_{ij} for an edge $\{i, j\} \in E$ might also occur to simplify the description, they are assumed to be false since this corresponds to having an edge. This also helps later in describing a tighter upper bound to the required number of clauses. Given a (partial) assignment X of the variables s_{ij}, there is an associated graph G_X that is obtained from G by contracting vertices i, j if $s_{ij} \in X$ and adding edges $\{i, j\}$ if $\overline{s}_{ij} \in X$.

Since contracting vertices can add edges between the new vertex and old neighbors, these edges also have to be set in the variables of the encoding after the decision to assign s_{ij} to true was made. This is called path consistency and entails the following: if u, v and v, w are assigned the same color (they are merged, s_{uv} and s_{vw} are true), then also u and w need to have the same color. As a clause: $s_{ij} \wedge s_{jk} \implies s_{ik}$, or $\overline{s}_{ij} \vee \overline{s}_{jk} \vee s_{ik}$ in CNF. Next to the $\overline{m} \in O(n^2)$ variables, these $O(n^3)$ transitivity clauses are also required to obtain a valid coloring.

$$\text{transitivity}(i, j, k) \coloneqq \overline{s}_{ij} \vee \overline{s}_{jk} \vee s_{ik} \qquad i, j, k \in V \tag{3.11}$$

Half of these can be left out due to the symmetry $\text{transitivity}(i, j, k) = \text{transitivity}(k, j, i)$, the total $3\binom{n}{3} = 1/2n(n-1)(n-2)$ required transitivity clauses are then as follows and collectively called $\text{transitivity}(G)$.

$$\begin{aligned} \text{transitivity}(i, j, k) &= \overline{s}_{ij} \vee \overline{s}_{jk} \vee s_{ik} \\ \text{transitivity}(j, i, k) &= \overline{s}_{ji} \vee \overline{s}_{ik} \vee s_{jk} \\ \text{transitivity}(i, k, j) &= \overline{s}_{ik} \vee \overline{s}_{kj} \vee s_{ij} \end{aligned} \qquad 1 \leq i < j < k \leq n \tag{3.12}$$

Their size can easily be bounded by $O(nm)$ as done in a similar context in [Ngu+17], but even be tightened further to $O(n \cdot \Delta(\overline{G})^2)$, a new result of this thesis proven in Proposition 3.2.1. While this can still be close to $O(n^3)$ for very sparse graphs, it is a much better bound for dense graphs where $\Delta(\overline{G})$ is small.

Proposition 3.2.1. *Assuming s_{ij} to be false if the edge $\{i, j\}$ is present in the graph, the number of transitivity clauses reduces from the $3\binom{n}{3}$ in Equation (3.12) to only $O(n \cdot \Delta(\overline{G})^2)$, where $\Delta(\overline{G})$ is the maximum degree of the complement graph $\overline{G}$.*

Proof. Considering a vertex i and pair j, k, the clause $\text{transitivity}(i, j, k) = \overline{s}_{ij} \vee \overline{s}_{jk} \vee s_{ik}$ is only necessary if both $\{i, j\} \notin E$ and $\{i, k\} \notin E$, otherwise the clause is trivially satisfied and can be left out. Thus, the clauses that remain are of the following form: $\text{transitivity}(i, j, k)$ where i is any vertex and j and k are not adjacent to i, and their number can be bound by $O(n \cdot \Delta(\overline{G})^2)$. □

With these transitivity clauses as part of the encoding, any full assignment X of the s_{ij} variables will lead to the graph G_X being a leaf node of the Zykov tree. This can be any leaf, and most likely does not yield an optimal coloring. To encode the number of colors used by the graph obtained from the assignment, variables c_j for $j = 1, \ldots, n$ are added and defined as in Equation (3.13), which can be translated into CNF using $O(n^2)$ clauses.

$$c_j \iff \bigwedge_{i<j} \overline{s}_{ij} \tag{3.13}$$

By this definition, c_j indicates whether vertex j needs a new color or not. If all s_{ij} are false for $i < j$, this means j does not have the same color as any previous vertex and thus needs a new color. Likewise, if s_{ij} is true for some $i < j$, vertex j has the same color as a previous vertex and does not need a new one. Since the first vertex cannot have the same color as any previous vertex, it always gets a new color and c_1 is assumed to be True. The literal c_1 can also be left out entirely but is included here to simplify case descriptions. The number of colors used by an assignment can then be counted as $\sum_{j=1}^{n} c_j$, i.e., the number of c_j set to true which corresponds to the number of vertices which need a new color.

An optimal coloring can then be found as the satisfying assignment with the fewest c_j set to true, while still remaining satisfiable. This optimization is not a natural satisfiability option, but there exist two possibilities to compute an optimal coloring this way. An instance consisting of the clauses transitivity(G) and definitions in Equation (3.13) can be encoded as a partial MaxSAT problem where the previous clauses are considered hard clauses and have to be satisfied. Then unit literals $\overline{c}_j$ for $j = 1, \ldots, n$ are added as soft clauses, i.e., the objective is to maximize the number of c_j set to false while satisfying the hard clauses.

It is also possible to encode k-colorability as a decision problem using the presented clauses. While not a natural satisfiability constraint, many options exist to encode cardinality constraints into a satisfiability problem. To limit the number of used colors, so-called at-most-k constraints are of particular interest. The full encoding then looks as follows:

Definition 3.2.2 (Full Encoding)**.** Given a graph $G = (V, E)$ and a $k \in \mathbb{N}$, the Full (Zykov) Encoding of k-colorability is given by the following clauses.

$$\begin{rcases} \text{transitivity}(i, j, k) \\ \text{transitivity}(j, i, k) \\ \text{transitivity}(i, k, j) \end{rcases} \quad 1 \leq i < j < k \leq n \tag{3.14}$$

$$c_j \iff \bigwedge_{i<j} \overline{s}_{ij} \qquad j \in \{1, \ldots, n\} \tag{3.15}$$

$$\sum_{j=1}^{n} c_j \leq k \tag{3.16}$$

Clauses (3.14) assert that a satisfying assignment corresponds to a valid leaf node in the Zykov tree and thus a graph coloring. Clauses (3.15) and (3.16) count and constrain the numbers of colors used by a satisfying assignment to k.

The kind of encoding used for the cardinality constraints (3.16) is not specified as part of the encoding and different methods can be used to enforce the constraint. In Section 3.1.1, at-most-one constraints were already used in (3.3) to assign each vertex at most one color. This was done by taking every pair of colors and enforcing that not both $x_{v,i}$ and $x_{v,j}$ can be set to true via the clause $\overline{x}_{v,i} \vee \overline{x}_{v,j}$. This is the pairwise encoding and can be extended to the binomial encoding of at-most-k constraints: take any subset of size $k+1$ of the n variables considered and add a clauses enforcing at least one of them has to be false. While this encoding requires no auxiliary variables, it suffers from the need for $\binom{n}{k+1}$ clauses of size $k+1$. The sequential encoding introduced in [Sin05] encodes a circuit that sequentially counts the number of variables set to true and needs $O(k \cdot n)$ new variables but only $O(k \cdot n)$ clauses. More variants exist, such as the cardinality network encoding [Así+11] or the totalizer encoding [Mar+14]. A partial survey of encodings can be found in [FG10]. For now it is not important which specific encoding is chosen, just that the at-most-k constraints can be translated into clauses in CNF.

Two downsides to the encoding as presented so far are the $O(n \cdot \Delta(\overline{G})^2) = O(n^3)$ many transitivity clauses needed to ensure path consistency, and the added complexity of the cardinality encoding needed to constrain the number of colors used. For the former, n new variables need to be introduced, and $O(n^2)$ clauses are added for the definition, as well as having to use extra methods to encode the cardinality constraints, likely with auxiliary variables and additional clauses.

To somewhat help with the latter issue, one improvement uses a large clique C. Unlike with the direct encodings, this does not allow fixing any of the s_{ij} variables, but it can help with the cardinality constraints. If the vertices are reordered, and the vertices of the clique are put first, all s_{ij} are false for $i < j \leq |C|$ since these are all edges in the clique. This forces the first $|C|$ of the c_j variables to be true, each clique vertex needs a new color. These variables can then be left out of the cardinality constraint, the at-most-k constraint can be reduced to an at-most-$(k - |C|)$ constraint. Since the number of variables and clauses for a cardinality encoding often depends on k this can reduce the size of their encoding by a large factor, especially if the clique is of size close to k. Further, putting a clique or a dense subgraph first in the vertex ordering might increase the reasoning strength of the cardinality encoding for the SAT solver. If a large degree vertex comes earlier in the ordering, fewer s_{ij} variables need to be proven to be false to identify the need of a new color for that vertex.

To address the first issue of the many transitivity clauses, Glorian et al. presented an incremental satisfiability framework [Glo+19] that is discussed in the next section.

3.2.2 CEGAR Approach and Incremental SAT Solving

The approach of Glorian et al. [Glo+19] is inspired by Counter-Example-Guided Abstraction Refinement (CEGAR), which was originally introduced as a tool for model checking and automated software verification [Cla+00]. The idea of CEGAR is to start with an under-abstraction of the problem, that is an encoding without some clauses of the full encoding, and then solve this under-approximation. Either the under-approximation is not satisfiable, then neither is the full encoding, or a model is found. This is tested to see whether it is a model for the full encoding, and if so, the algorithm terminates with a satisfying model. Otherwise, the model is a counter-example, and the encoding is refined with additional clauses to exclude this model, hence the name of CEGAR.

In [Glo+19] the approach is applied to graph coloring using the full Zykov encoding in the previous section. Instead of solving one SAT instance with all transitivity clauses for a specific k, several more and more constrained problems are solved incrementally. The encoding is initially relaxed to contain none of the clauses transitivity(G) and solved. If the instance is already unsatisfiable, the instance with all of transitivity(G) is unsatisfiable as well. Otherwise, a satisfying assignment, i.e., a model, is found. This model is tested for being a model of the Full Encoding; here that means checking whether it corresponds to a valid graph coloring. If this is the case, a valid model to the Full Encoding is returned. Otherwise, this model is treated as a counter-example, and some of the transitivity clauses that it violates are added as new clauses to the instance. The instance is then solved again, and the previous steps are repeated until UNSAT is returned, or a model corresponding to a graph coloring is found. Figure 3.2 shows a diagram of the solving loop. As mentioned before, the original authors reported excellent results with this approach, although their source code disappeared, and they are unable to repeat the experiments. One goal of this chapter is to reimplement their algorithm and ideally reproduce their excellent results as claimed.

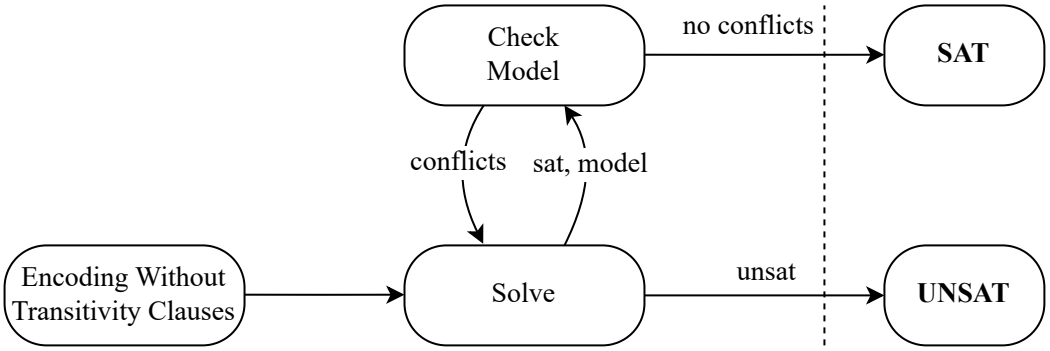


Figure 3.2: Diagram of the CEGAR solving loop for graph coloring. The process starts in the bottom-left node.

The main part of this approach is then to design a check algorithm, an efficient way to return violated transitivity clauses to be added to the instance or quickly determine that the model is valid. Important factors of this algorithm are the runtime but also which conflicts are returned. One immediate check algorithm is to simply go through all transitivity constraints, test whether they are fulfilled and if not, add them to the list of conflicts. This naive approach has a cubic runtime complexity and thus is not the most efficient, but it does produce a list of all conflicts in each call. The pseudocode is given in Algorithm 3.1.

Algorithm 3.1: NaiveCheck

Input: Graph G, model λ
Output: List of conflicts T

```
1  T ← ∅
2  for i ← 1 to n do
3      for j ← i + 1 to n do
4          for k ← j + 1 to n do
5              if λ ⊭ transitivity(i, j, k) then
6                  T ← T ∪ {(i, j, k)}
7              if λ ⊭ transitivity(j, i, k) then
8                  T ← T ∪ {(j, i, k)}
9              if λ ⊭ transitivity(i, k, j) then
10                 T ← T ∪ {(i, k, j)}
11 return T
```

The authors of [Glo+19] presented an algorithm based on building a coloring from the obtained model and directly checking for conflicts from the built coloring. Their algorithm is referred to as ColorCheck and described next, with pseudocode given in Algorithm 3.2. Given the graph G, an integer k, and a model λ satisfying the current encoding, each vertex is first assigned a set of all possible colors $\{1, 2, \ldots, k\}$. Then, for each vertex u that still has available colors, $c[u]$ is fixed to the first possible color. Next, it is checked what this color assignment means for vertices v with $u < v$: if $s_{uv} = \text{False}$ in the model λ, this means u and v get different colors and $c[u]$ is removed from $c[v]$ (line 8). Otherwise $s_{uv} = \text{True}$ and v is assigned the same color chosen for u (line 12). If $c[v]$ becomes empty in the process (line 9 or line 13), it is checked if a reason for this conflict can be deduced. The reasons

are violated transitivity constraints, for example, in line 9 vertices w, v have the same color ($s_{wv} \in \lambda$) and w, u have the same color ($c[w] = c[u]$) but u, v are assigned different colors ($\overline{s}_{uv} \in \lambda$). Thus, the transitivity constraint transitivity$(u, w, v) = \overline{s}_{wu} \vee \overline{s}_{wv} \vee s_{uv}$ is violated and added to T. lines 15 and 16 similarly produce the other two types of transitivity constraints from Equation (3.12). The correctness of this algorithm is given formally by Proposition 3.2.3.

Algorithm 3.2: ColorCheck

Input: Graph G, model λ, integer k
Output: List of conflicts T

```
1  T ← ∅, c ← a map
2  for u ← 1 to |V| do  c[u] ← {1, 2, ..., k} ;
3  for u ← 1 to |V| do
4      if c[u] ≠ ∅ then
5          c[u] ← {i} where i was the first element of c[u]
6          for v ← u + 1 to |V| do
7              if s̄_uv ∈ λ then
8                  c[v] ← c[v] \ c[u]
9                  if c[v] = ∅ and ∃w < u s.t. s_wv ∈ λ and c[w] = c[u] then
10                     T ← T ∪ {(u, w, v)}
11             else
12                 c[v] ← c[v] ∩ c[u]
13                 if c[v] = ∅ then
14                     Let w < u with c[w] ≠ ∅ s.t. ① s_wv ∈ λ and c[w] ≠ c[u]
                                                or ② s̄_wv ∈ λ and c[w] = c[u]
15                     case ① do  T ← T ∪ {(w, v, u)}
16                     case ② do  T ← T ∪ {(w, u, v)}
17 return T
```

Proposition 3.2.3 ([Glo+19]). *Let $G = (V, E)$ a graph, k an integer, and λ a model that satisfies an under-abstraction of* transitivity(G)*, the clauses Equation (3.13) defining the c_j and clauses encoding the constraint $\sum_{j=1}^{n} c_j \leq k$. If* ColoringCheck$(G, \lambda, k)$ *returns $T = \emptyset$ then it is possible, following the model λ, to color G with k colors. Otherwise, $\exists (i, j, k) \in T$ then λ violates transitivity(i, j, k).*

Before the adapted proof of Proposition 3.2.3 is reported here, a slight correction in the algorithm is highlighted. Originally, the check for $c[w] \neq \emptyset$ in line 14 was not present. This allows $c[w] = \emptyset$ and $c[v] = \emptyset$ to happen, in which case $c[w] \neq c[u]$ is true, and the triple is wrongfully reported as a conflict. It is intended that w is colored and u is colored differently from which the information for the conflict is deduced, this fails if w is not colored at all. This causes a triple involving w with $c[w] = \emptyset$ to be added to T which is not responsible for the conflict. Further, the actual conflict involving a different w with $c[w] \neq \emptyset$ is not considered anymore. This does not render the algorithm incorrect but violates Proposition 3.2.3: it is possible to have $(u, v, w) \in T$ for which λ does not violate transitivity(u, v, w), contrary to the last part of the proposition. Luckily, if there exist violated transitivity clauses, the algorithm in [Glo+19] still finds at least one of them. The change to the algorithm presented here avoids this and only finds conflicts, i.e.,

$(u, v, w) \in T$ implying $\lambda \not\models \text{transitivity}(u, v, w)$ holds for Algorithm 3.2. More differences between the original source and the algorithm presented here, especially regarding the implementation, are discussed in Section 3.3. Another insufficiency in the proof of [Glo+19] is also addressed in the version given here.

Proof of Proposition 3.2.3. The proof begins with showing that if T is non-empty, $\lambda \not\models \text{transitivity}(i, j, k)$ for $(i, j, k) \in T$. Consider the two cases where triples are added to T:

- If (u, w, v) is added in line 10, that means $w < u$ was found with $c[w] = c[u] = \{i\}$ which implies that $s_{wu} \in \lambda$, since otherwise i would have been removed from $c[u]$ when considering w in the outer loop. Further $s_{wv} \in \lambda$ and $\overline{s}_{wv} \in \lambda$ by assumption so that $\{s_{wu}, s_{wv}, \overline{s}_{uv}\} \subseteq \lambda$ which means that $\lambda \not\models \text{transitivity}(u, w, v)$.
- If (w, v, u) is added in line 15 or (w, u, v) is added in line 16, $c[u] = \{i\}$ and $s_{uv} \in \lambda$ but $i \notin c[v]$. The latter implies that i has been previously removed in line 8 or line 12 when some vertex $w < u$ was considered. Either ① $s_{wv} \in \lambda$ and $c[w] \neq \{i\}$, thus $\overline{s}_{wu} \in \lambda$ because $c \in c[u]$ since otherwise i would have previously been removed from $c[u]$. Additionally, $s_{uv} \in \lambda$ and $s_{wv} \in \lambda$ by assumption so that $\{s_{wv}, s_{uv}, \overline{s}_{wu}\} \subseteq \lambda$ which means that $\lambda \not\models \text{transitivity}(w, v, u)$. This analysis does not hold if $c[w] = \emptyset$ despite the check $c[w] \neq c[u]$ being satisfied, which was overlooked in the original proof of [Glo+19]. If ② $\overline{s}_{wv} \in \lambda$ and $c[w] = \{i\}$, then $s_{wu} \in \lambda$ since $c \in c[u]$. Again, by assumption $s_{uv} \in \lambda$ so that $\{s_{wu}, s_{uv}, \overline{s}_{wv}\} \subseteq \lambda$ which means that $\lambda \not\models \text{transitivity}(w, u, v)$.

Now if T is empty at the end of the algorithm, one needs to show that a k-coloring can be constructed from the model λ; this is done by proving that the map c yields a valid k-coloring. Suppose that a vertex was not colored, i.e., $\exists v \in V$ s.t. $c[v] = \emptyset$ and $T = \emptyset$ at the end of the algorithm. Consider the first time $c[v]$ becomes empty for some vertex v. In line 12, a conflict is always added, and a w as required always exists as one of the two cases has to hold for $c[v]$ to become empty if $s_{uv} \in \lambda$. The only place where $c[v]$ can become empty but no conflict is added is line 8. In that case, for all $w < u$ it holds: ($\overline{s}_{wv} \in \lambda$ or $c[w] \neq c[u]$). Here is another gap in [Glo+19]; the authors assume that $\overline{s}_{wv} \in \lambda$ for all $w < u$ without showing why ($\overline{s}_{wv} \notin \lambda$ and $c[w] \neq c[u]$) cannot happen.

Claim. If $c[v]$ becomes empty in line 9 but the conditions for a conflict are not satisfied, then $\overline{s}_{wv} \in \lambda$ for all $w < u$.

Proof of claim.. Assume $s_{wv} \in \lambda$ and the condition of adding a conflict is not satisfied, i.e., $c[w] \neq c[u]$ which implies $\overline{s}_{wu} \in \lambda$. Since $s_{wv} \in \lambda$ but $c[v] = \emptyset$, there exists a vertex u' with $w < u' < v$ such that $s_{wu'} \in \lambda$ and $\overline{s}_{u'v} \in \lambda$. The vertex u' is then responsible for making $c[v]$ empty, and as such $u' = u$ because the first time $c[v]$ becomes empty is considered, which happened with u used in the outer loop. The contradiction then comes from $\overline{s}_{wu} \in \lambda$ and $s_{wu'} \in \lambda$, meaning that the assumption $s_{wv} \in \lambda$ is false, proving the claim.

Now $\overline{s}_{wv} \in \lambda$ for all $w < u$ but further $\overline{s}_{wv} \in \lambda$ for all $w < v$, since otherwise v has the same color as a previous vertex w and $c[w]$ would have become empty before $c[v]$. Since $c[v]$ is empty and $\overline{s}_{wv} \in \lambda$ for all $w < v$, there exist k vertices w_j s.t. $c[w_j] = \{j\}$ for each $j \in 1, 2, \ldots, k$; let w_j further be the vertex that was first assigned color j. For each w_j, $c_{w_j} \in \lambda$ because otherwise there exists a $w < w_j$ with $s_{ww_j} \in \lambda$ and $c[w] \neq \{j\}$, contradicting $c[w_j] = \{j\}$. In total, there are k true literals c_{w_j} for $j = 1, \ldots, k$ and $w_j < v$, plus c_v since v required a new color as well, so that $\lambda \not\models \sum_{j=1}^{n} c_j \leq k$, a contradiction to λ being a model of the under-abstraction containing the cardinality constraint.

This shows that the map c assigns each vertex a color if T is empty at the end of the algorithm. That this uses k colors is trivial since no more than k colors were available,

and to see that this is a valid coloring, one only needs to verify that $c[u] \neq c[v]$ for edges $\{u, v\}$. Since $\{u, v\}$ is an edge, s_{uv} is assumed to be false and the color of u is removed from $c[v]$ when considering u in the outer loop. The algorithm thus concludes with a valid k-coloring, finishing the proof.

□

The authors of [Glo+19] observed the checking phase to be their bottleneck, and a lot of runtime was spent in Algorithm 3.2. In an effort to remedy this, and to consider what impact different checking algorithms have on the performance, a new checking algorithm is presented in this thesis. To this end, a definition is needed.

Definition 3.2.4 (Pairing graph)**.** Let $G = (V, E)$ be a graph, and λ a model satisfying the current under-abstraction of transitivity(G). The pairing graph $P(G, \lambda) = (V, E')$ is defined on the same vertex set V and with edges $E' \coloneqq \{\{u, v\} : s_{uv} \in \lambda\}$.

An edge $\{u, v\} \in P(G, \lambda)$ means u, v are paired and get assigned the same color. A conflict in the pairing graph is an incomplete triangle with only two out of three sides; this corresponds to a violated transitivity constraint. To find such incomplete triangles, and with that, violated clauses, one can adapt triangle listing algorithms such as ones discussed in [SW05]. In particular, the "node-iterator" algorithm extends to the problem at hand quite nicely: iterate over all nodes $j \in P(G, \lambda)$ and test for each pair i, k of neighbors whether the edge $\{i, k\}$ is missing in $P(G, \lambda)$. If so, a conflict is found. The pseudocode for this algorithm to find violated transitivity clauses is given in Algorithm 3.3, and the correctness is stated in Proposition 3.2.5.

Algorithm 3.3: TriangleCheck

Input: Graph G, model λ
Output: List of conflicts T

```
1 T ← ∅
2 P ← P(G, λ)
3 foreach j ∈ V(P) do
4     foreach pair i ≠ k ∈ N_P(j) do
5         if {i, k} ∉ E(P) then
6             T ← T ∪ {(i, j, k)}
7 return T
```

Proposition 3.2.5. *Let $G = (V, E)$ a graph and λ a model that satisfies an under-abstraction of* transitivity(G)*, the clauses Equation (3.13) defining the c_j, and clauses encoding the constraint $\sum_{j=1}^{n} c_j \leq k$. If* TriangleCheck$(G, \lambda)$ *returns $T = \emptyset$ then it is possible, following the model λ, to color G with k colors. Otherwise, $\exists (i, j, k) \in T$ then λ violates transitivity(i, j, k). The algorithm runs in $O(n \cdot \Delta(\overline{G})^2)$.*

Proof. The second part of the statement is easy to see: if there is a violated transitivity constraint, then there is an incomplete triangle in $P(G, \lambda)$ and it is found by the algorithm. Vice versa, if such a triangle exists, that means the model violated the corresponding transitivity constraint. Now if $T = \emptyset$, by the previous observation λ satisfies all transitivity constraints, which means a valid graph coloring can be extracted. Since λ additionally satisfies $\sum_{j=1}^{n} c_j \leq k$, the coloring uses at most k colors.

The pairing graph can be computed in $O(n^2)$ by iterating over each variable s_{uv}, and the "node-iterator" algorithm requires $O(n \cdot \Delta(G)^2)$ on a graph G. Since the adapted

algorithm is run on $P(G, \lambda)$ which can only possibly have an edge if the two vertices were non-adjacent in G, $\Delta(P(G, \lambda))$ can be bounded by $\Delta(\overline{G})$. □

A big difference of Algorithm 3.3 compared to Algorithm 3.2 is that the former, just like the naive Algorithm 3.1, finds all transitivity conflicts violated in the current model λ. The coloring-based Algorithm 3.2 finds a lot fewer conflicts in each call, while still making sure it finds at least one if it exists. The latter seems more aligned with the CEGAR approach, whose idea is to avoid the cumbersome amount of transitivity clauses by working with smaller under-abstractions. Therefore, a variant of Algorithm 3.3 called SparseTriangleCheck was also tested that adds only some of the violated clauses. It differs by immediately moving on to the next vertex j if a conflict was found and added in line 6.

3.2.3 Implementation Details

While some details were given when describing the Zykov-based encodings, more specifics of the implementation are discussed here, and the differences to the implementation of [Glo+19] are highlighted.

Beginning with the direct encodings, in Definition 3.1.1 and Section 3.1.2 the pre-assigning or fixing of variables for vertices of a large initial clique was already described, and simple symmetry-breaking constraints were added. They further allow for some incrementality when solving k-coloring problems for decreasing k. In the case of the assignment encoding, one can simply add unit clauses $(\overline{x}_{v,i})$ for all vertices v when going from the encoding of the k-coloring problem to $k-1$, as this simply disallows vertices to have color k. Similarly, one can add the unit clauses $(\overline{y}_{v,k-1})$, disallowing any vertex to have a color larger than $k-1$. For top-down solving, the direct encodings only have to be built once for the largest k and the unit clauses are added to produce the encoding for smaller k. This incrementality has the possible benefit of reusing learnt conflict clauses, though this is unlikely as any learnt clause containing $\overline{x}_{v,k}$ or $\overline{y}_{v,k-1}$ is satisfied after the addition of the unit clauses and thus dropped. Nonetheless, the solver can reuse other information of the search process, such as the variable score of the decisions heuristic or the previous variable phase. Without using assumptions, it is not possible to use incremental solving for the bottom-up search, as clauses from a more constrained problem are not safe to reuse in a less constrained problem where more colors are allowed. Using assumptions does not seem practical here, as part of the encoding would have to be rebuilt anyhow.

The assignment encoding with and without the at-most-one color constraints are called "Assignment Encoding(AMO)" and "Assignment Encoding", respectively, see Definition 3.1.1. The partial order encoding of Definition 3.1.3 is called "Partial Order Encoding".

The next presentation of implementation details for the full Zykov-based encoding and the CEGAR approaches is done in a way to highlight the differences to [Glo+19]. Their implementation is referred to as "Picasso". The main details for the implementation of this thesis are the following.

- No variables are added for edges in the graph, and all clauses being trivially satisfied because there is an edge present are left out as well, leading to the bound on the transitivity clauses of Proposition 3.2.1. These redundant variables and clauses are added in Picasso, though this should not make a significant difference as unit propagation is extremely fast in SAT solvers.

- Initial bounds are computed, and the graph is reduced according to the preprocessing presented in Algorithm 2.4. Search for the chromatic number is then conducted in the range $[lb, ub)$, either top-down or bottom-up as described in Section 1.2.3. This

preprocessing is not done in Picasso, and their best performing configuration does bottom-up search starting at 1.

- The vertices are ordered with a large clique C at the beginning, and the remaining vertices are ordered as computed in the Dsatur heuristic. This allows reducing the at-most-k constraint to one of at-most-$(k - |C|)$, and further strengthens the reasoning ability as discussed in Section 3.2.1. This is not done in Picasso, in a personal communication one of the authors said that, while it is a clear theoretical improvement, it is not better from an engineering standpoint.
- The state-of-the-art SAT solver CaDiCal is used with the version described in Section 1.4 as it was quite successful in recent years of the SAT competition, while Picasso uses the slightly older Glucose 4.0.
- The totalizer encoding [Mar+14] is used for the at-most-k constraints. In particular, the interface of the Open-WBO MaxSAT solver [MML14] is adapted to add the constraints to CaDiCal that was not originally supported. That implementation has the crucial functionality of incremental constraints; with additional variables and setting them via assumptions, the value of k can be changed without needing to rebuild the encoding, even in bottom-up search. The encoding used is different from both the cardinality network encoding mentioned to be used in [Glo+19], and the adder encoding used in their incorrect source code. The former does not allow for incremental constraints in Open-WBO and thus cannot be used here, the latter is an encoding for pseudo-boolean constraints but works with coefficients set to 1. It does not officially support incrementality with the Open-WBO interface, but it can be used nonetheless.
- In [Glo+19], a time limit of 900 seconds was used with a memory limit of 32 GB. As described in Section 1.4, limits of one hour and 16 GB are used in this thesis.

The Zykov-based encodings are called "Full Encoding" for Definition 3.2.2 and "CEGAR Naive", "CEGAR Paper", "CEGAR Triangle" and "CEGAR Sparse" for the CEGAR approaches using Algorithms 3.1 to 3.3 and the slight variant of Algorithm 3.3; sometimes they are referred to without the CEGAR suffix when clear from the context. As initial testing hinted at the vertex ordering, and the use of a clique in particular, having a significant impact, a variant "Full Encoding (lex)" using the lexicographic ordering is also tested.

The highlighted differences of this thesis and the work of [Glo+19] all appear to be beneficial, but this does not mean this is true in the empirical results. For this reason, a variant as close as possible to the description in [Glo+19] is also implemented. In short, this means the redundant variables for edges are added with respective unit clauses, no initial bounds and preprocessing are computed, search is bottom-up starting at 1, the lexicographic vertex ordering is used, and Glucose 4.0 is used as SAT solver with a time limit of one hour. This version is called "Paper Configuration".

3.3 Experimental Evaluation

The evaluation of the satisfiability encodings follows three goals: first, analyze the number of transitivity clauses in the Full Encoding and how many are necessary with the CEGAR approaches. Further, a comparison is made with the new theoretical bound of Proposition 3.2.1. Second, a similar evaluation as done in [Glo+19] is repeated for the re-implemented algorithm here, and the data of the paper and results obtained here are

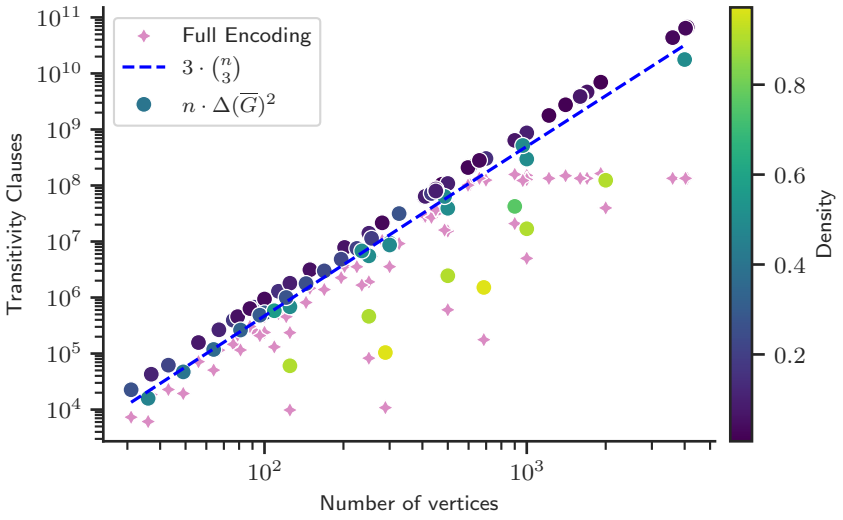


Figure 3.3: Different theoretical bounds compared to the number of transitivity clauses in the Full Encoding. The dots represent the bound $n \cdot \Delta(\overline{G})^2$ for different densities

contrasted. Finally, the Full Encoding and CEGAR approaches are compared to the simpler direct encodings, and a comparison with methods from the literature is performed. These goals are addressed individually in the next subsections.

Better results were achieved when performing bottom-up search for the chromatic number, both in this work and in [Glo+19]. Therefore, all results for algorithms implemented in this thesis and presented in this evaluation use bottom-up search.

3.3.1 Number of Transitivity Clauses

A new bound of $n \cdot \Delta(\overline{G})^2$ on the number of transitivity clauses was proved in Section 3.2.1. This bound depends on the maximum degree of the complement graph, which can be quite large even if the initial graph is dense. In particular, if the graph contains an isolated vertex, $\Delta(\overline{G})$ will be $O(n)$ and the bound is no better than $O(n^3)$. It is thus interesting to see whether it has an advantage over the bound of $3 \cdot \binom{n}{3} = 1/2 \cdot n(n-1)(n-2)$ and how both compare to the number of transitivity clauses used by the Full Encoding in practice. In Figure 3.3, the bounds and the Full Encoding are plotted for the instances of the benchmark set. Since the bound $3 \cdot \binom{n}{3}$ only depends on n, it is plotted as the diagonal, while the novel bound depends on n and the degrees of the graph involved and as such is only plotted for each instance. As the density, or more specifically $\Delta(\overline{G})$, is an important factor in the bound, it is further visualized by a gradient color scale.

One can observe that the simple binomial bound is most often slightly tighter than the novel bound. This is not fully surprising as the benchmark set consists of many very sparse instances where $\Delta(\overline{G})$ is most likely close to n. For these instances, the number of clauses from the Full Encoding is quite close to the simple binomial bound. However, for the few denser instances, the number of transitivity clauses can be a lot lower and $n \cdot \Delta(\overline{G})^2$ orders of magnitude better than the simpler bound. This concludes that it can be a contributing factor when bounding the number of transitivity clauses, but more likely only for dense instances.

The underlying idea of Glorian et al. [Glo+19] is that not all transitivity clauses are strictly necessary to solve an instance. While the performance is evaluated in Section 3.3.2, it

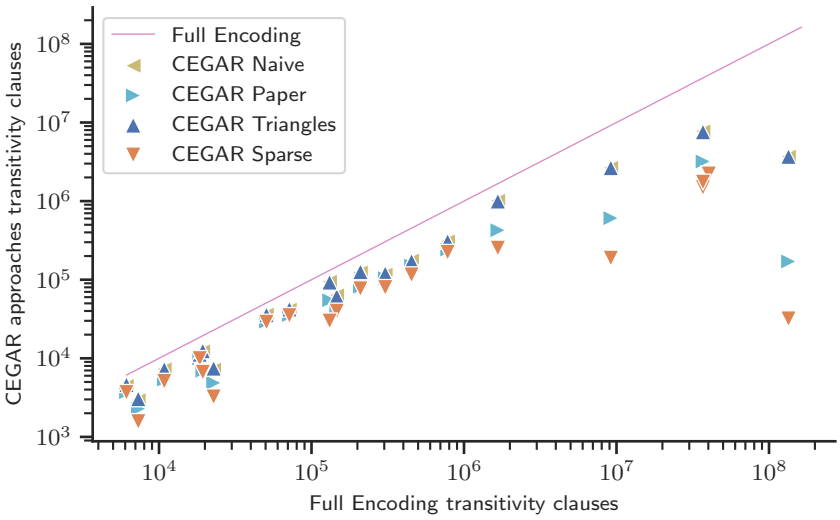


Figure 3.4: Comparative plot of required transitivity clauses in the Full Encoding and the CEGAR appraoches

is interesting to see here how many clauses are used in practice by the CEGAR approach. As graph coloring is a global optimization problem on the entire graph where local constraints are not enough to solve it, potentially all transitivity constraints need to be considered. An empirical evaluation is presented in Figure 3.4, which plots the number of transitivity clauses required by the CEGAR algorithm compared to number in the Full Encoding. The instances are limited to those that were solved by the CEGAR method to be representative with regard to how many clauses are required to solve an instance. How far points are below the diagonal represents how many fewer clauses were necessary in the CEGAR case.

Trivially, all points are below the diagonal. One trend to notice from this plot is that the more transitivity clauses are present in the Full Encoding, the larger the difference to the CEGAR approaches is. While between 10^4 and 10^6 the points are quite close to the diagonal line, the difference becomes significant for more clauses. This implies that for larger instances, more of the transitivity clauses are redundant, while not that much is to be gained for instances that are small to begin with. Considering the CEGAR variants that use different checker algorithms, the Naive and the Triangle checker produce almost identical results. This is not surprising since both find every conflict in every iteration. Likewise, the other two variants using the Paper and the Sparse checker can solve some instances with a lot fewer clauses as they only add some of the existing conflicts. This means that even in a single iteration where conflicts are found, many of them are redundant. Looking further, Paper and Sparse produce a different number of conflicts in each iteration, with the latter consistently solving instances with fewer clauses than the other algorithms. This confirms the hypothesis of [Glo+19] that many transitivity clauses are not required to solve an instance. Additionally, check algorithms that do not add all violated clauses to the encoding further lead to fewer clauses being added while solving. The variant using the Sparse checker newly introduced in Section 3.2.2 outperforms other variants in terms of clauses required. Whether this leads to a similar reduction in total solving time is considered in the next subsection.

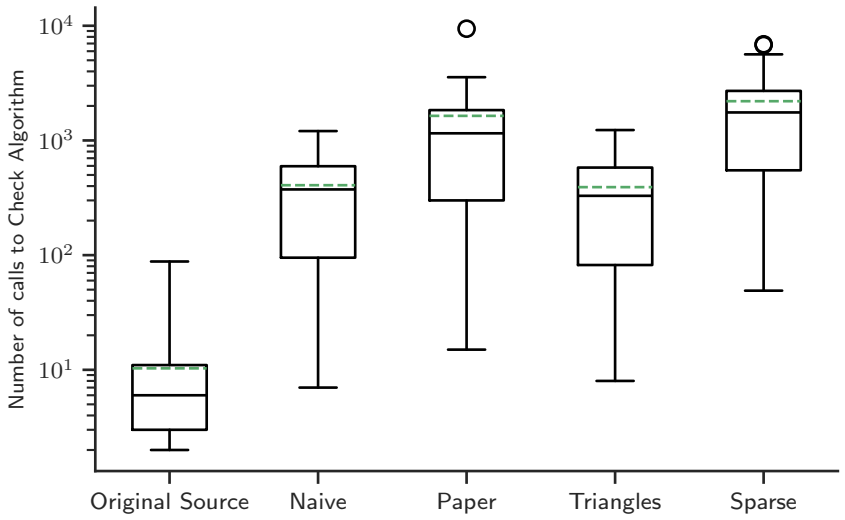


Figure 3.5: Distribution of CEGAR iterations for different checking algorithms

3.3.2 Performance Trends Compared to Original Paper

In the original paper, Glorian et al. provide some further evaluation results besides comparing the number of instances solved on their benchmark set. This includes the number of CEGAR iterations in the form of a boxplot and aggregate results of the time spent in different parts of the algorithm. The data obtained by the re-implementation of this thesis is compared to the data as presented in the paper.

For the number of iterations, i.e., the number of times the check algorithm is called while solving an instance, the authors give enough information to reproduce their data in a boxplot. This boxplot, called "Original Source" in the figure, and others for the number of iterations for the CEGAR algorithms as re-implemented are given in Figure 3.5. The black line is the median number, the green dashed line represents the average, and the empty circles represent rare outliers significantly larger than other data-points. Again, the data for the algorithm using Naive or Triangle as the checker is nearly identical. As they find every conflict in every iteration, they generally require fewer calls to solve an instance than the two other methods. Despite that, the data from [Glo+19] suggests that their checker, which does not add every conflict in each iteration, requires significantly fewer calls overall. Comparing the data of the paper, i.e., Original Source in the figure, with the Naive checker, 75% of instances require more iterations than the maximum number for the original source, and the average is about 50 times that of the paper. It is highlighted that the plot uses a logarithmic y-scale. Further considering the CEGAR variants of this thesis, an observation, unsurprising after that of the previous subsection, is both Paper and Sparse require more iterations than Naive or Triangles. As they find fewer conflicts in each iteration, this helps in adding fewer transitivity clauses but requires more iterations. For this reason, Sparse performs slightly more iterations than Paper.

When analyzing their runtime data, the authors of [Glo+19] observed the check algorithm to be the bottleneck, out of the encoding phase, the check or CEGAR phase, and the solving phase. Although giving their results in aggregated form, they report that on average, about 60–70% of the runtime is spent looking for violated constraints with their check algorithm. This is quite surprising given that they perform so few CEGAR loops, and the check algorithm has a runtime complexity of $O(n^3)$ while the satisfiability solver is unlikely to run in polynomial time on the NP-hard graph coloring problem. Further,

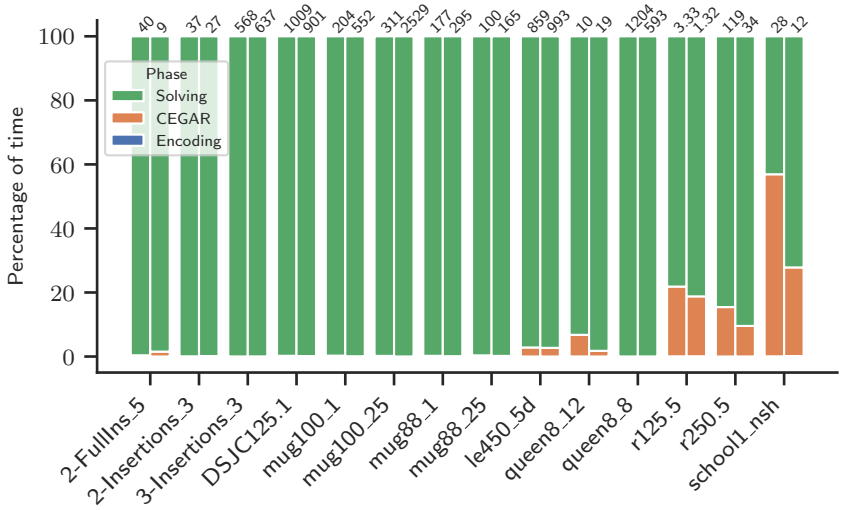


Figure 3.6: Distribution of time spent in different phases of the algorithm. The left and right bar of an instance is the data for the Paper and the Sparse checker, respectively

about 20% of the runtime is spent on building the encoding, and the solving phase makes up the rest. A similar evaluation for the CEGAR Paper and Sparse implementation of this thesis is done in Figure 3.6. The algorithm time is split up into the three phases adding up to a bar representing 100% of the time; the left resp. right bar of an instance is the data for the Paper resp. the Sparse checker. The number above each bar represents the total algorithm time to give a perspective on how much time was spent in each phase. The instances considered are restricted to graphs that were solved and exclude those where the sum of all phases is less than a second as these are not as representative. Similarly, the time spent on preprocessing is excluded. An immediate observation is that, overwhelmingly, most runtime is spent in the solving phase of the algorithm, i.e., inside the SAT solver. The only remarkable instances are r250.5 and school_nsh, where the CEGAR phase takes up a decently visible amount of runtime, and the instances are not solved too quickly. For the latter and the Paper checker, the CEGAR phase makes up about 50% of the runtime, which goes down to about 30% for the Sparse checker. The time spent on building the encoding is not visible in this comparison and makes up a negligible amount of runtime in the implementation of this work. The clear bottleneck is the SAT solver used to solve the instance.

Finally, what is of most practical interest is the performance of the algorithms with regard to the number of solved instances and average runtime. Glorian et al. [Glo+19] report great results of the CEGAR algorithm on their benchmark set; in particular, the CEGAR variant can solve more than twice as many instances as the Full Encoding. The Full Encoding and the re-implementation of the CEGAR variants were run on the DIMACS benchmark set and with one hour and 16 GB limits as specified in Section 1.4. As mentioned before, due to the better performance in [Glo+19] and in the implementation here, search for the chromatic number is done bottom-up. A survival plot of the performance is given in Figure 3.7, where a point at (x, y) means that the algorithm manages to solve y instances within a time limit of x seconds. The plot begins at 50 instances, as those were solved in preprocessing independent of the configuration. On top of that, about 12 instances are trivial for all configurations and solved very quickly. After that, the performance spreads for different algorithms. Again, one can observe that CEGAR Naive and Triangles

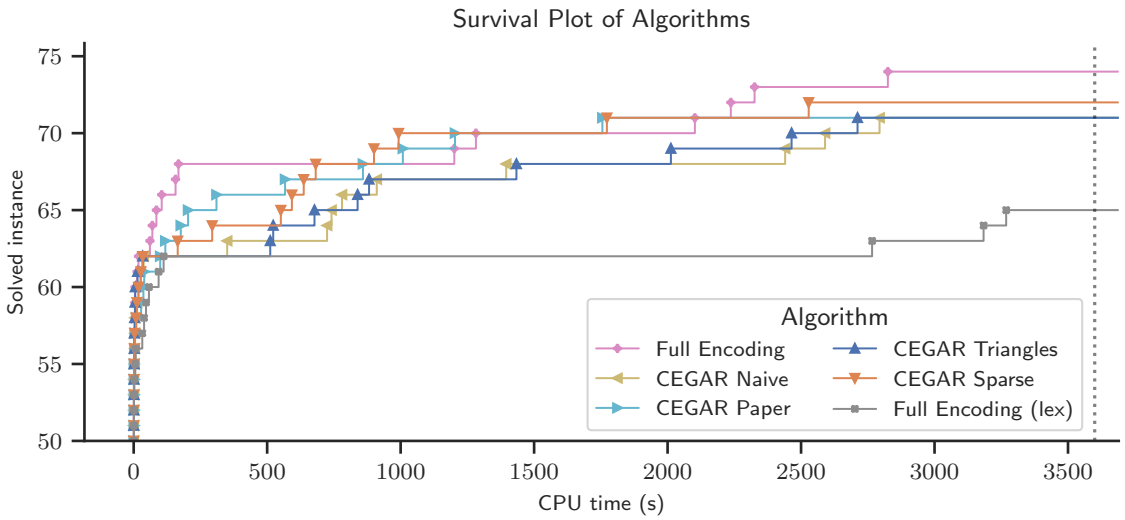


Figure 3.7: Survival plot for Full Encoding and CEGAR variants

behave almost the same. With the final time-limit of 3600 seconds, CEGAR variants Naive, Paper, and Triangles all solve 71 instances, with Sparse being able to solve one more. While for a few time limits Paper and Sparse perform slightly better than the Full Encoding, it ultimately solves two more instances than the best performing CEGAR algorithm and is noticeably better for smaller time limits. To consider the impact of the vertex ordering on solving performance, the variant Full Encoding (lex) was proposed that takes the lexicographic vertex ordering as present in the instance file. As conjectured after the theoretical observation in reasoning strength and initial testing, the vertex ordering makes a significant difference. While the 12 easy instances are also solved quickly, the Full Encoding without improved vertex ordering struggles to solve other instances afterward.

To conclude, while the new check algorithm Sparse brings some improvement over the check algorithm of [Glo+19], none of them outperform the Full Encoding. This is despite the reduction in the number of required transitivity clauses observed before. These results strongly contrast the ones reported in the original paper, where their CEGAR algorithm is not just better but solves more than twice as many instances. While results here are skewed by the many instances solved in preprocessing, the CEGAR algorithms still do not perform as well as the Full Encoding. Further, the vertex ordering with a clique at the beginning has a notable impact on performance.

As the presentation of the encodings in this thesis differed from the original source, it might be the case that the proposed improvements do not have the desired effect and in fact hinder performance. For that reason, a version called Paper Configuration was implemented as well. This aims to be as close to the description of [Glo+19] as possible, as was detailed at the end of Section 3.2.3. The results are quite disappointing; without preprocessing and using the beneficial vertex ordering only 13 instances are solved. This is far from the results obtained in Figure 3.7, and even further from the results claimed in [Glo+19]. While this approves of the proposed improvements to the algorithm presented in this work, it leaves open how the results of Glorian et al. were achieved. Given the performance of the Paper Configuration algorithm, it is unclear to the author what unconsidered implementation details could lead to such a huge performance discrepancy.

Specific numbers on the performance of the algorithms discussed so far are provided in the next section, along with a comparison to the direct encodings and methods of the literature.

3.3.3 Computational Study

The previous evaluation focused on the CEGAR algorithms and their performance characteristics. Next, their competitiveness is compared more globally to the direct encodings of Section 3.1 and SAT methods from other works. In particular, the assignment encoding with and without at-most-one constraints, the partial order encoding, the full Zykov encoding, and CEGAR Sparse are considered.

From the literature, recent papers that use satisfiability methods, achieve strong results, and publicly provide their source code are considered. Namely, gc-cdcl by Hébrard and Katsirelos [HK20][2], CliColCom by Heule, Karahalios, and Hoeve [HKH22][3], and most recently, several encodings by Faber, Jabrayilov, and Mutzel [FJM24][4]. Their partial order encoding POP-S performed best and is used for this evaluation. For gc-cdcl, their bottom-up variant gc-cdcl↑ is used. All external source code is compiled and run on the same machine as specified in Section 1.4.

The gc-cdcl method is also based on the Zykov property and essentially uses the same variables as the Full Encoding and the CEGAR variants. Hébrard and Katsirelos propose to enforce the transitivity constraints through a dedicated propagator, additionally used to avoid encoding the cardinality constraints. Further, they use it to interact with the search procedure and cut off parts of the search tree deemed useless, which is not possible with the static Zykov-based encodings. In CliColCom, an interaction between cliques and colorings is used. With better cliques, they can find better colorings and vice versa; to find and prove the chromatic number, the assignment encoding without at-most-one color constraints but more symmetry-breaking constraints is used. Most recent are the encodings of [FJM24], which translates the improved partial order ILP formulation [JM18] to a satisfiability encoding. Their best performing encoding is the partial order encoding POP-S with strict symmetry breaking constraints, which was presented here as the partial order encoding in Section 3.1.2 with more relaxed symmetry breaking.

In Figure 3.8 a survival plot for the different satisfiability encodings is given. Compared to other encodings, the Zykov-based Full Encoding and CEGAR Sparse perform worse than the rest, with 74 and 72 instances solved, respectively. The implementation of gc-cdcl↑ is able to solve 82 instances within the time limit, showing an advantage over the Full Encoding and CEGAR Sparse but not performing as well as the simpler direct encodings. Among those, the performance is relatively close, with all of them solving between 89 and 93 instances of the benchmark set. All code and experiments were compiled and run on the same machines, so even if similar encodings are used, the implementation differences that are present lead to different results. The assignment encoding, CliColCom, and the assignment encoding with at-most-one color constraints solve 89, 90, and 91 instances, respectively. The method POP-S based on the partial order encoding with additional symmetry breaking constraints also solves 91 instances. The implementation of the partial order encoding in this thesis manages to solve 93 instances and is consistently the fastest, meaning it solves the most instances even for smaller time limits.

As the experiments were run on a different machine than the one in their respective papers, and some include a certain randomness factor, the results here differ slightly. For instance, gc-cdcl was reported to solve 83 instances on the same benchmark set in [HKH22], whereas in these experiments only 82 instances were solved within the time limit. Likewise, 88 instances are reported to be solved by CliColCom in [HKH22] and 90 instances are solved in this evaluation. Their algorithm relies on a random local search solver for the

[2] https://bitbucket.org/gkatsi/gc-cdcl/src/master/
[3] https://github.com/marijnheule/clicolcom
[4] https://github.com/s6dafabe/popsatgcpbcp

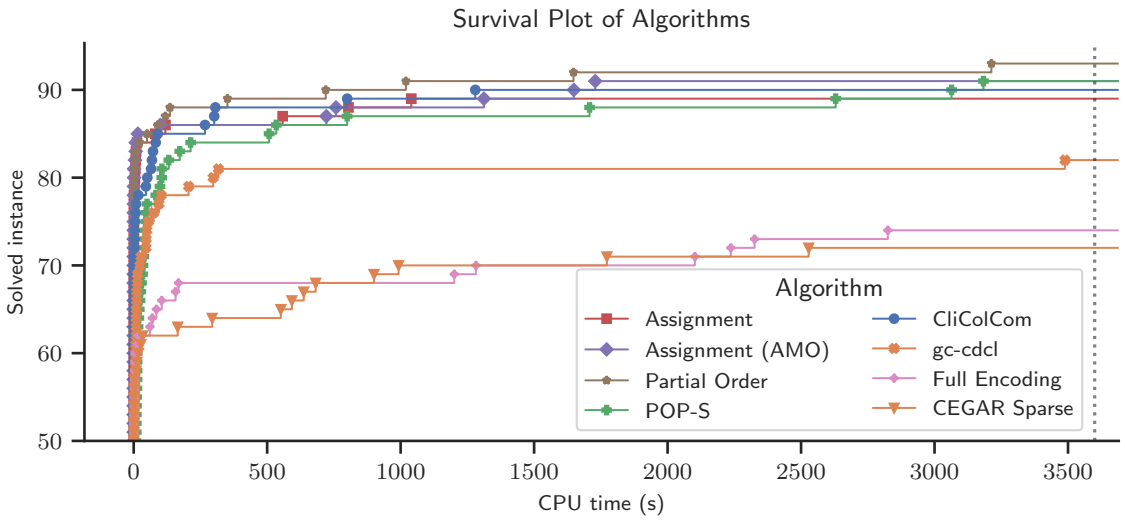


Figure 3.8: Survival plot for the direct encodings, Full Encoding, CEGAR Sparse and, from the literature, gc-cdcl↑, CliColCom and POP-S

SAT problem, so this can explain the difference when using different seeds and a different machine. With POP-S, the same number of instances as reported in [FJM24] is solved, using the same SAT solver Kissat 3.1.1 [BF22]. Switching to CaDiCal as the satisfiability solver, the performance degrades and only 87 instances are solved. Especially on the wap instances, using Kissat seems to make an important difference, solving five instances where CaDiCal solves only one. Similarly, the partial order encoding of this thesis using CaDiCal does not perform as well on the wap instances, only solving three of them. It makes up for this by solving other instances that POP-S cannot, among them the larger and denser instances DSJR500.5, DSJR500.1c and r1000.5 where the strict symmetry breaking constraints potentially degrade the performance of POP-S. It is briefly noted that the runtimes of POP-S reported here and visualized in the survival plot differ slightly from the runtimes presented in the source paper as their evaluation did not include the time spent in preprocessing. This also allows instances to potentially run for longer than the specified time limit of one hour, though none of the solved instances require more than one hour even when taking the preprocessing time into account.

A more detailed but still aggregated report of the results is given in Table 3.1. The instances are grouped by their respective class, and for each class the number of solved instances as well as the average lower and upper bound are given for the satisfiability encodings considered in this section. As CliColCom does not readily provide bounds, only the numbers of solved instances are listed. Gray cells mean that the algorithm achieved the best result, if only a single algorithm achieved the result, it is further highlighted in bold. This helps point out the differences between the algorithms, for example, CliColCom is the only instance that solves 10 of the le450 instances. This is due to the local search solver being successful and and finding a satisfiable solution, while the algorithms using a SAT solver struggle with these instances. Likewise, it shows the strength of POP-S on the wap instances. As the search for the chromatic number was done bottom-up, improved upper bounds between different algorithms mean either more solved instances or that the initial coloring heuristic computed better bounds. The appendix provides a full table that reports the performance for each individual instance, see Table A.1.

Interestingly, the Full Encoding achieves the best average lower and upper bound on the class of the r instances, despite solving the same number as the direct encodings.

This is because the direct encodings solve r1000.5 which the Full encoding does not, but surprisingly, it solves the previously open instance r1000.1c within the time limit of one hour, determining the chromatic number to be 98. Only recently, a new lower bound of 97 was manged to be proven in [HKH22] using the assignment encoding, which times out after 24 hours when trying to prove that no 97-coloring exists. Given the noncompetitive performance of the Full Encoding compared to the direct encodings, it is unusual that the so far unsolved instance is solved in only 35 minutes. This shows a particular instance where the downside of the Full Encoding becomes an advantage: having a variable present for each non-edge of the graph. While this is quite hindering for sparse graphs, for r1000.1c with a density of 97%, the number of variables is relatively low for such a large graph with large chromatic number. As the direct encodings have $O(k \cdot m)$ variables, many more variables are required. Regarding the CEGAR algorithm, they are unable to solve the instance in bottom-up search, but in top-down search the Full Encoding, CEGAR Naive and CEGAR Triangles solve it as well.

Observing great performance with the partial order encoding of this thesis but lacking results on the wap instances because CaDiCal is used, the wap instances were further investigated. The partial order encoding is used to generate the satisfiability encoding for a 40-coloring of wap07a, and this formular is then solved with Kissat. Providing Kissat the hint that the instance is likely satisfiable, it manages to find a satisfying assignment and thus a 40-coloring for wap07a within 3 hours. This closes another open instance of the DIMACS benchmark set as wap07a contains a clique of size 40. Further experiments on other instances yield an improved lower bound of 5 for the graph 1-Insertions_6. To the authors' knowledge, this is the first time this bound is reported, although it was solved by all direct encodings in only a few seconds. Further, CliColCom and POP-S are both able to prove this lower bound. The former does not list or highlight this, and it did not appear in the latter because of an implementation detail in the reporting of the lower bound that hides this. Lastly, the lower bound of 6 on the instance DSJC250.1 is confirmed by the assignment encoding with at-most-one color constraint in a little less than one hour. This bound was previously reported in [Por20], but required about 36 hours using up to 20 threads on a multicore CPU.

To conclude the experimental evaluation of this chapter, the results of the CEGAR algorithm claimed in [Glo+19] were not reproduced, despite many proposed improvements to the formulation and method. A variant as close to the description of the paper performed remarkably poorly, and overall very different trends for the number of CEGAR iterations and runtime were observed. Turning to an evaluation of all different satisfiability encodings presented here and selected encodings from the literature, the Zykov-based Full Encoding and CEGAR variants are reported to not be competitive to the direct encodings. Only on the very dense instance r1000.1c does the Full Encoding achieve exceptional results, determining a chromatic number of 98 for the previously unsolved instance. For the assignment encoding, adding the at-most-one color constraints is determined to have a benefit for the considered instances. Although the direct encodings achieve relatively similar results, the partial order encoding of this thesis performs best, solving 93 instances of the benchmark set compared to the 91 of Assignment (AMO) and POP-S. Further experiments solved the open instance wap07a, and determined an improved lower bound of 5 for the graph 1-Insertions_6.

Table 3.1: Aggregated performance results of different satisfiability encodings

		Assignment			Assignment (AMO)			Partial Order			Full Encoding			CEGAR Sparse			gc-cdcl↑			POP-S			CCC
Class	#	Opt.	*lb*	*ub*	Opt.	*lb*	*ub*	Opt.	*lb*	*ub*	Opt.	*lb*	*ub*	Opt.	*lb*	*ub*	Opt.	*lb*	*ub*	Opt.	*lb*	*ub*	Opt.
Books	5	5	11.20	11.20	5	11.20	11.20	5	11.20	11.20	5	11.20	11.20	5	11.20	11.20	5	11.20	11.20	5	11.20	11.20	5
CX000	3	0	38.33	370.67	0	38.33	370.67	0	37.33	370.67	0	35.67	370.67	0	37.00	370.67	0	34.33	384.00	0	30.67	383.00	0
DSJC	12	1	25.67	74.75	1	**25.75**	74.75	1	25.58	74.75	1	24.75	74.75	1	24.83	74.75	1	22.67	75.50	1	22.67	75.75	1
DSJR	3	2	73.00	74.00	3	73.00	73.00	3	73.00	73.00	2	73.00	74.00	2	73.00	74.00	1	71.00	76.00	1	71.33	77.33	3
FullIns	14	14	6.79	6.79	14	6.79	6.79	14	6.79	6.79	12	6.64	6.79	12	6.64	6.79	13	6.71	6.79	14	6.79	6.79	14
GPIA	5	5	5.60	5.60	5	5.60	5.60	5	5.60	5.60	0	4.60	6.40	1	5.60	6.40	4	5.60	6.00	5	5.60	5.60	4
Insertions	11	4	4.27	5.18	4	4.27	5.18	4	4.27	5.18	4	3.55	5.18	2	3.45	5.18	2	3.73	5.18	4	3.64	5.18	4
flat	6	0	16.00	79.00	0	16.00	79.00	0	15.83	79.00	0	14.00	79.00	0	14.67	79.00	0	14.00	78.83	0	15.00	**78.00**	0
fpsol2	3	3	41.67	41.67	3	41.67	41.67	3	41.67	41.67	3	41.67	41.67	3	41.67	41.67	3	41.67	41.67	3	41.67	41.67	3
games120	1	1	9.00	9.00	1	9.00	9.00	1	9.00	9.00	1	9.00	9.00	1	9.00	9.00	1	9.00	9.00	1	9.00	9.00	1
inithx	3	3	38.67	38.67	3	38.67	38.67	3	38.67	38.67	3	38.67	38.67	3	38.67	38.67	3	38.67	38.67	3	38.67	38.67	3
latin	1	0	90.00	137.00	0	90.00	137.00	0	90.00	137.00	0	90.00	137.00	0	90.00	137.00	0	90.00	**125.00**	0	90.00	132.00	0
le450	12	8	15.00	17.58	8	15.00	17.58	8	15.00	17.58	5	15.00	18.25	5	15.00	18.25	8	15.00	**17.00**	8	15.00	17.08	**10**
miles	5	5	34.80	34.80	5	34.80	34.80	5	34.80	34.80	5	34.80	34.80	5	34.80	34.80	5	34.80	34.80	5	34.80	34.80	5
mug	4	4	4.00	4.00	4	4.00	4.00	4	4.00	4.00	4	4.00	4.00	4	4.00	4.00	4	4.00	4.00	4	4.00	4.00	4
mulsol	5	5	34.60	34.60	5	34.60	34.60	5	34.60	34.60	5	34.60	34.60	5	34.60	34.60	5	34.60	34.60	5	34.60	34.60	5
myciel	5	5	6.00	6.00	5	6.00	6.00	5	6.00	6.00	5	6.00	6.00	5	6.00	6.00	5	6.00	6.00	4	5.60	6.00	4
qg	4	3	57.50	58.00	3	57.50	58.00	3	57.50	58.00	0	57.50	59.25	0	57.50	59.25	3	57.50	58.00	3	57.50	61.50	3
queen	13	6	10.85	12.92	6	10.85	12.92	8	10.92	**12.46**	5	10.77	13.08	5	10.77	13.08	6	10.85	13.00	8	10.92	12.92	5
r	9	8	63.78	64.67	8	63.78	64.67	8	63.78	64.67	8	**64.00**	**64.56**	7	63.89	65.22	7	62.33	67.00	7	62.22	66.11	7
school	2	2	14.00	14.00	2	14.00	14.00	2	14.00	14.00	2	14.00	14.00	2	14.00	14.00	2	14.00	14.00	2	14.00	14.00	2
wap	8	2	41.38	46.50	3	41.38	46.00	3	41.38	46.00	1	41.38	46.88	1	41.38	46.88	1	41.38	47.00	**5**	41.38	**45.00**	4
zeroin	3	3	36.33	36.33	3	36.33	36.33	3	36.33	36.33	3	36.33	36.33	3	36.33	36.33	3	36.33	36.33	3	36.33	36.33	3
Total	137	89	24.01	39.58	91	**24.02**	39.53	**93**	23.99	**39.49**	74	23.69	39.74	72	23.77	39.78	82	23.39	40.04	91	23.34	40.00	90

Chapter 4

User Propagators for the Graph Coloring Problem

In Chapter 3, satisfiability encodings were introduced that are static in the sense that the satisfiability solver is treated as a blackbox. The incremental CEGAR approach was only slightly less static. Information from a satisfying model was used to refine and modify the current encoding, but again it was simply passed to the satisfiability solver for an answer. As such, the SAT solver does not take advantage of problem-specific information, even if the encoding is refined by the graph coloring counter-examples. Hébrard and Katsirelos [HK20] proposed a fully dynamic method, their hybrid constraint programming/satisfiability approach uses a propagator to directly access and interact with the solving process. This method is the focus of this chapter. More precisely, the implementation of a user propagator based on the Zykov encoding to solve the coloring problem is described. This is also referred to as Zykov propagator.

First, propagators in the context of satisfiability are presented in Section 4.1. Their use for the Zykov-based encoding to enforce the transitivity constraints is explained in Section 4.2. Different decision strategies that take advantage of the problem structure are discussed in Section 4.3. Another crucial factor for the performance of this method is the ability to interact with the solver and prune parts of the search space; different ways to do so are explored in Section 4.4. Many of the involved ideas can be transferred to a user propagator for the assignment encoding. This is explored in Section 4.5. An experimental evaluation of the propagator for the Zykov-based encoding is performed in Section 4.6.

4.1 Propagators for Satisfiability

The method of constraint propagation and a user-defined propagator is more common in the field of constraint programming. While the definition of constraint programming is not necessary here, it is useful to introduce what a propagator does and how it works. Broadly, propagation can be understood as local reasoning on the constraints of a problem. It is an algorithm that enforces constraints by deducing variable assignments based on the given constraints and current assignment. In SAT solving, the constraints are usually present in the form of a clause, and one form of propagation is already a core part of SAT solvers: boolean constraint propagation or unit propagation. Given a clause with all but one unassigned literal being falsified, the last literal must be true to satisfy the clause; the literal is set to true by unit propagation. In the context of satisfiability, propagators aim to produce exactly such unit propagations, without the clause explicitly being part of the encoding. Instead, the propagator observes the variable assignment and produces the unit propagation when it is deduced from an implicit constraint. For example, given variables s_{uv}, s_{uw} and s_{vw} for the Zykov-based encoding, a propagator for the transitivity constraints would propagate $s_{vw} = \text{True}$ as soon as it observes s_{uv} and s_{uw} to become true. Only later would it produce the clause $\overline{s}_{uv} \wedge \overline{s}_{uw} \wedge s_{vw}$ if it is required in a conflict analysis.

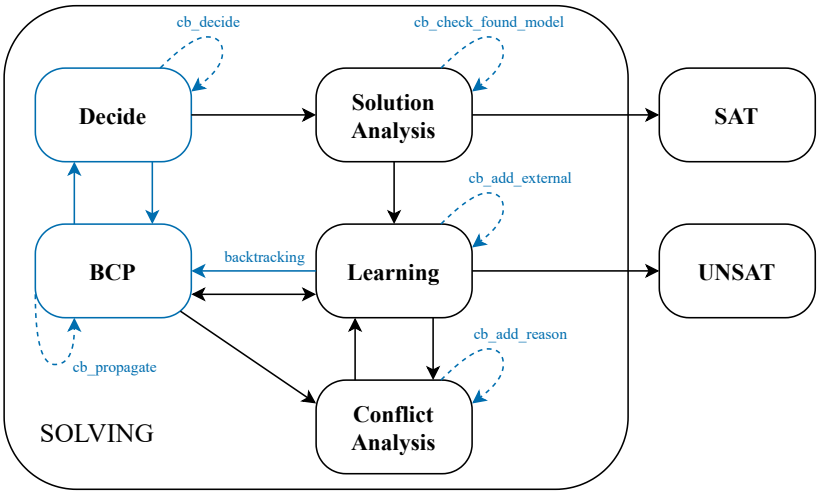


Figure 4.1: Overview of the states in the solving process and provided user callbacks. Adapted from [Faz+23]

In a more general context such as constraint programming, propagators can be used to enforce arbitrary constraints [Gen+14]. They can also be understood as a type of lazy clause generation, where the constraint is not present in the instance but instead generated during solving should it become violated [OSC07].

Previously, designing a custom propagator for a satisfiability problem required modifying the source code of an existing solver and implementing function calls at the right points in the solving process. But recently, Fazekas et al. [Faz+23] presented an interface of a user propagator for CDCL on top of the SAT solver CaDiCal [Bie+24]. This provides callbacks which notify the user about changes in the search process such as variable assignments, decisions, and backtracks. Further, it allows the user to influence the solving process via callbacks to produce propagations, add external clauses, design custom decision strategies, or verify that the satisfying assignment is correct. These callbacks and their interaction with the solving process are visualized in Figure 4.1. The different states should be familiar from the description of CDCL in Section 1.2, however, a complete understanding is not necessary for the discussion of this chapter. The main thing needed for the algorithms presented here is that the user propagator interface allows for three important things:

cb_propagate Propagations can be given without necessarily adding a clause to the encoding. The explanation clause is only needed if the propagation is part of a conflict.

cb_decide Custom decisions strategies that use additional problem information can be implemented.

cb_add_external Clauses can be added during the solving process and at any decision level; these can be used to prune the search tree and force backtracking.

Working with the user propagator and influencing the solving process always requires an explanation that is understandable to the SAT solver. A propagation of a literal can be made, but if it is later involved in a conflict, the clause which caused unit propagation of the literal has to be supplied. Likewise, when a node is to be pruned in the search tree, a clause is necessary that is unsatisfiable in the current node and thus causes backtracking. These explanation clauses are required for the satisfiability solver to prove unsatisfiability. The choice of which clause is produced also has an impact.

SAT solvers generally try to minimize the length of the learned clause during conflict analysis, this reduces memory usage and solving time as the learnt clause is more effective [SB09]. In the context of MinCostSAT [MM02] and MaxSAT [Oli+23], reducing the size of blocking clauses or learned clauses has been identified as a significant factor for performance. Shortening and thus strengthening the reason clause leads to measurable performance gains in their applications. As propagators in constraint programming often work without explanations, this appears to be a less important concern in their study, though an equivalent of clause learning in constraint programming exists. This is called (generalized) NoGoods [KB05] and similarly tries to learn assignments that are not contained in any solution.

4.2 Transitivity Propagation

One major motivation for using a propagator is to avoid explicitly adding the $O(n \cdot \Delta(\overline{G})^2)$ many transitivity clauses as this becomes overwhelming for large, sparse graphs. To still enforce transitivity on the variables s_{uv}, the propagator is responsible for delivering the propagation $s_{vw} = \text{True}$ whenever s_{uv} and s_{uw} become true. To do this efficiently, the propagator maintains a graph H equal to G_X, according to the current partial assignment X. That is, if a variable is assigned $s_{uv} = \text{True}$, vertices u and v are contracted. If it is assigned $s_{uv} = \text{False}$, an edge is added between the two vertices. To keep track of the merged vertices, for each $v \in G$ the propagator maintains a set $\text{bag}(v)$ of merged vertices that it belongs to. Further, it stores the unique representative $\text{rep}(v)$ of the bag that v was merged into. Additionally, it is tracked which vertices are still in the graph, i.e., were not merged into another vertex, which can also be deduced from $rep(v) = v$. Initially, $\text{rep}(v) = v$ and $\text{bag}(v) = \{v\}$ for all v as no vertices have been contracted.

Now, if a literal is assigned, the propagator is notified and updates the graph accordingly. Say the assignment was $s_{uv} = \text{True}$, this corresponds to contracting u and v in H and assigning them the same color. This implies all the vertices of $\text{bag}(u)$ and of $\text{bag}(v)$ are assigned the same color, and the propagator sets $s_{u'v'}$ to true for all $u' \in \text{bag}(u)$ and $v' \in \text{bag}(v)$. The representative is updated to one of $rep(u)$ or $rep(v)$ for all vertices in $\text{bag}(u) \cup \text{bag}(v)$. Further, any vertex $u' \in \text{bag}(u)$ cannot be assigned any of the colors of $v' \in N(\text{bag}(v)) \backslash N(\text{bag}(u))$, so $s_{u'v'}$ is set to false by the propagator, and symmetrically for $v' \in \text{bag}(v)$. If the propagator is notified of $s_{uv} = \text{False}$, an edge is added between u and v in H and the literals $s_{u'v'}$ are set to false for $u' \in \text{bag}(u)$ and $v' \in \text{bag}(v)$.

If the propagator is notified of $s_{uv} = \text{True}$ or $s_{uv} = \text{False}$ but u and v were already contracted or separated, the propagator does nothing. This is important so that the propagator only considers each variable once, which then leads to the complexity of the transitivity propagation over a branch of the search tree being $O(\overline{m}) = O(n^2)$. This means that for a path from the root node to a full assignment in the search tree, the transitivity propagation only has quadratic complexity, while for the Full Encoding $O(n^3)$ transitivity clauses are considered. This already shows a theoretical advantage of using an external propagator.

Furthermore, the subgraph H, on which the contractions and edge additions are performed, needs to be able to be efficiently restored to the state of a previous level. If a conflict is encountered in the search, the solver backtracks to an earlier decision level and the partial assignment of that level. Likewise, the graph needs to be restored to the state of that level. The simplest way to do this is to keep a copy of the graph for each decision level, but this requires a lot of space. One could also store the graph of the root and, after each jump to a previous decision level, perform the assignments from the root to the new

level on the graph to reconstruct the graph at the decision level. This avoids having a copy for each decision level but still does a lot of redundant work, especially since the solver often backtracks only by a single level.

The most space- and time-efficient implementation is to make the graph operations invertible, so that for backtracking, each vertex contraction and edge addition performed can be reversed. For edge addition, this is straightforward, as the edge simply has to be removed from the graph again. Undoing contractions is more complex, as one has to recover the previous vertex bags $\text{bag}(u)$ and $\text{bag}(v)$, and the respective representatives after they were merged when contracting u and v. To achieve this, additional data has to be stored for each level. This includes what the vertex representatives $\text{rep}(v)$ were at that time, information to recover the vertex bag $\text{bag}(v)$, and any added edges to be removed again. Since the contraction of two vertices u and v requires the computation of $N(v) \backslash N(u)$, the graph uses an adjacency matrix presentation with bitsets. These allow for efficient bit-parallel computation of such set operations.

So far, this section described how the propagations of the transitivity constraints are handled. Should a propagation be involved in a conflict, the user propagator further needs to provide the reason clause that caused the propagation. This is simply done by storing the clause when the propagation is given, so that it can easily be retrieved by the propagator when it is needed to analyze a conflict. In particular, if $s_{vw} = \text{True}$ is propagated because $s_{uv} = \text{True}$ and $s_{uw} = \text{True}$, the clause $(\overline{s}_{uv} \wedge \overline{s}_{uw} \wedge s_{vw})$ is stored.

4.3 Decision Strategies

Modern state-of-the-art satisfiability solvers come with their own sophisticated decision strategies. Many of them follow the main principle of choosing variables as decisions that recently were part of a conflict by keeping track of "variable activity" values [BF15]. In the first heuristic of this kind, called Variable State Independent Decaying Sum (VSIDS) [Mos+01], each variable starts with a counter initialized to 0 and is increased when the variable is part of a conflict clause. The unassigned variable with the largest value is chosen, and periodically all counters are divided by a constant, explaining the "Decaying Sum" in the name. Different variants of this idea exist, for example, Exponential VSIDS (EVSIDS) [ES03] or Average Conflict-Index Decision Score (ACIDS) [BF15]. Another heuristic is called Learning-Rate Branching (LRB) [Lia+16], which picks a variable that maximizes a metric $P(x, I)/|I|$ called "learning rate". For a certain interval of time I, the function $P(x, I)$ measures the number of conflicts in I that the variable x was involved in.

This is just to say that the decision heuristic is a crucial part of the satisfiability solver. A tremendous amount of effort has been put into analyzing it, improving its performance, and coming up with effective strategies. State-of-the-art SAT solvers are designed to perform well for a wide range of instances and applications. Therefore, their decision heuristics are general-purpose strategies that achieve satisfactory results for most cases. Given a specific class of problems, one might design a custom decisions strategy that can take advantage of the known problem structure. Ideas to do so with the Zykov propagator for graph coloring are discussed next.

One well-known and competitive strategy in the graph coloring context is the Dsatur heuristic of Brélaz [Bré79]. As presented in Section 2.2, the next vertex is chosen as the one that maximizes the saturation value $sat(v) \coloneqq |\{c(u) : \{u, v\} \in E\}|$. This uses the partial coloring assigned so far to deduce the most constrained vertex. As the Zykov-based encoding does not work with explicit colorings but only color relations, this strategy is not straightforward to use in the Zykov propagator.

In [HK20], Hébrard and Katsirelos proposed a way to emulate the Dsatur heuristic without the need for coloring variables and a partial assignment. As discussed in Section 3.2, any partial coloring maps to a node in the Zykov tree, and likewise one can map a node in the tree to a (non-unique) partial coloring. To do so, a maximal clique C in the current graph G_X is chosen and each vertex is assigned a different color. A vertex adjacent to many of the clique vertices then corresponds to a vertex with high saturation degree. Let v be the vertex that is not in the clique and maximizes $|N(v) \cap C|$, breaking ties by the highest degree. Further, choose any vertex $u \in C \backslash N(v)$. If the clique is isolated in G, the intersection $N(v) \cap C$ is empty for all v. To avoid this, cliques are iterated until a non-isolated maximal clique is found, or in the rare case that no such clique exists, any non-adjacent vertices u and v are chosen. The decision passed to the solver via the propagator callback is $s_{uv} = \text{True}$, which corresponds to assigning v the same color as u in the partial coloring constructed from the clique C. If this assignment is refuted for all $u \in C \backslash N(v)$, i.e., a conflict clause is learned that implies $s_{uv} = \text{False}$ at the current level, v becomes adjacent to all vertices of the clique, thus yielding a larger clique $C \cup \{v\}$. This corresponds to a new color being used in the partial coloring coming from the improved clique.

The authors of [HK20] suggest that this branching strategy can be more flexible than decisions based on committing to a coloring as in the Dsatur algorithm. There, a clique is only computed once at the beginning of the root node as a lower bound. Since the algorithm only assigns colors, the graph structure is not changed, making computations of a clique during the algorithm useless. This was also addressed in [FGT17], where lower bounds taking advantage of the previous coloring decisions are used during the search. With the branching strategy of [HK20], a maximal clique is computed on G_X for each decision and is not restricted by the choice of the initially computed clique at the start of the algorithm.

Several different branching heuristics were presented in [Hul21], although in the context of a column generation algorithm for graph coloring. Hulst compared different branching heuristics in their branch-and-price implementation and observed their new strategy called ISUN to perform best for them. It branches on the non-adjacent vertices u and v that maximizes the sum of their degrees, as computed in Equation (4.1). The reasoning is that it will produce the largest change on the graph, making the search tree shallower or producing conflicts sooner.

$$\arg \max_{u,v \in V : \{u,v\} \notin E} |N(u)| + |N(v)| \tag{4.1}$$

The performance of these custom decision strategies for the Zykov propagator is discussed in the experimental evaluation of this chapter. A comparison to the general-purpose decision heuristic of the SAT solver is also made.

4.4 Search Tree Pruning

While the two previous sections presented ways to enforce the necessary transitivity constraints and influence the search with decision strategies, this section discusses methods that directly integrate into the solving process by pruning parts of the search tree. This follows the ideas presented in [HK20] which use subgraph-based lower bounds computed during the search to cut off nodes that cannot lead to a solution. When the current graph G_X contains a subgraph with chromatic number larger or equal to the current upper bound, it is clear that no smaller coloring can be found if the search was continued on this graph.

This fact is used to cut off the current partial assignment and backtracking by supplying the right reason clause to the SAT solver.

This technique is introduced in detail, and the potential of other lower bounds not based on witness subgraphs is discussed. Since finding these bounds can be expensive, different strategies of when to compute them are presented. Further, the novel idea of computing improved upper bounds during search is introduced. Additionally, methods to apply preprocessing rules to the subproblems in a way compatible with the SAT solver are described.

4.4.1 Subgraph-Based Lower Bounds

Each node in the Zykov tree again corresponds to a graph coloring problem, this time on the graph G_X built from the partial assignment X. The advantage of this is that the same methods used in Chapter 2 can be used on G_X to simplify the problem. In particular, any lower or upper bound computed on G_X is a lower resp. upper bound for the subproblem. Likewise, possible preprocessing rules from Section 2.3 can be applied and further simplify the subproblem.

The focus of the propagator presented in [HK20] was on using improved lower bounds to prune the search tree and force a backtrack. This is done by finding a lower bound on G_X and comparing it to the current upper bound. Assume the lower bound of G_X is $k+1$ and the decision problem is currently looking for a k-coloring. By the Zykov property of Equation (3.10), no k-coloring can be found in the current node or any of its children. Thus, the node can be disregarded, and the search backtracks. To do this with the propagator, the callback to add external clauses to the encoding is used.

As with the propagations, the SAT solver requires an explanation in the form of a clause that causes the backtrack. Before, the propagation was given, and the reason only supplied when needed for a conflict. Since here the conflict is present at the current node, also the explanation clause has to be given then. The two lower bounds used for the propagator are the clique bound and the Mycielsky bound as presented in Algorithms 2.1 and 2.2. This time, the Mycielsky algorithm is initialized with a maximum clique computed by the greedy clique algorithm, contrary to starting with en empty subgraph as done in the preprocessing of Algorithm 2.4. Both bounds are based on a witness subgraph H that have a specific chromatic number, which allows for a convenient explanation to the SAT solver. The subgraph leads to a bound conflict and thus cannot exist as it currently does, in particular, one of the edges that was added should have been a contraction instead. The full clause looks as follows:

$$\bigvee_{\{u,v\} \in E(H)} s_{uv} \tag{4.2}$$

If the witness graph H is present, every s_{uv} is set to false. Adding the clause of Equation (4.2) means that at least one of the added edges must be a contraction of vertices instead. This then avoids the subgraph and the bound conflict it causes. The vertices u and v of the variables s_{uv} were chosen here as the representatives but could be chosen as any vertex in $u' \in \text{bag}(u)$ and $v' \in \text{bag}(v)$ as their variable $s_{u'v'}$ is also false. These clauses can even cause non-chronological backtracking as not all previous decisions must have been involved in the subgraph.

For cliques, this leads to an explanation using $O(|H|^2)$ many positive literals. Hébrard and Katsirelos experimented with finding shorter explanation clauses involving negative literals, as an existing edge can sometimes also be explained by the contraction that caused it. They reported that sometimes explanations of length $O(|H|)$ can be found, but this did not pay off. While the clauses tended to be much shorter and increased

the number of conflicts per second, the overall number of conflicts and runtime increased significantly [HK20]. Therefore, only the explanation clause of Equation (4.2) is used.

In the example of Figure 3.1(a), a partial Zykov tree for the 5-cycle as root graph was given. For this graph, both the contraction and the edge addition lead to graphs containing a 3-clique. The clique bound recognizes this, and both subproblems can be pruned when looking for a 2-coloring. This determines that the instance is unsatisfiable without having to look at all subproblems and shows the effect of pruning the search tree with dynamic lower bounds. Further, the Mycielsky bound can already recognize a lower bound of 3 at the root node.

The clique bound inside the Zykov propagator can also be used to replace the encoding of the cardinality constraints as they can be handled with bound conflicts as well. If one is looking for a k-coloring, all leaf nodes of size $k+1$ are pruned as the clique bound produces a conflict. If a leaf of size k and thus a satisfying assignment is found, the propagator is updated to prune all nodes containing a clique of size $k-1$ instead so that the search continues and produces a $(k-1)$-coloring next. In general, there is no clear answer whether encoding the cardinality constraints or enforcing them by a propagator is better. This question was discussed by Abío et al. [Abí+13]. In their experiments, they observed propagation to yield better results when many cardinality constraints are present. However, encoding the constraint performed better than propagation for instances with few and large cardinality constraints. Since only a single, large cardinality constraint is used for the Zykov-based encoding, it is also used in the Zykov propagator even if not strictly necessary to guarantee that at most k colors are used. Hébrard and Katsirelos do not use an encoding for the cardinality constraints but enforce them with the propagator.

An improvement is proposed here that allows for fully incremental bottom-up solving, which was not considered in [HK20]. They implemented bottom-up search but had to reset the propagator and restart the search each time k was increased. This is due to the explanation clauses of bound conflicts becoming invalid; if a node was pruned by a clique of size $k+1$, this node might be relevant when next looking for a $(k+1)$-coloring. When the propagator is reset to start solving for the next k, the conflict clauses are removed from the encoding. This allows the node to be considered again, but all other learnt clauses or information from incremental solving are lost too. Using activation literals as described at the end of Section 1.2.2, the added explanation clauses of Equation (4.2) for bound conflicts can be deactivated again. This allows one to go from a more constrained to a less constrained problem without having to rebuild the encoding.

Performing bottom-up search is likely to be the stronger approach, as this way the upper bound for conflicts is as tight as possible, which can make the pruning-based propagator a lot more effective.

4.4.2 Global Lower Bounds

As already seen in Section 2.1, other lower bounds for the graph coloring problem exist, and they can similarly be computed on the current graph G_X. The auxiliary graph-based lower bound is promising in particular. For their improved Dsatur algorithm, Furini, Gabrel, and Ternier [FGT17] rebuild the reduced graph, the auxiliary graph, and the maximum independent set formulation for each node where the bound is computed. In the context of the Zykov propagator, however, one can design a fully incremental approach that avoids this repeated work.

The assignments made inside the Zykov propagator mean either two vertices are merged, or the edge between them is added. In the maximum independent set ILP formulation on the auxiliary graph, this corresponds to setting the variable of that non-edge to 1 or 0,

respectively. This was previously described in Section 2.1.3 when the bound was introduced. Consequently, the assignments made in the Zykov propagator directly correspond to the fixing of certain variables in the ILP. This allows building the auxiliary graph and the maximum independent set ILP once at the root node and fixing variables for the assignment X to obtain an ILP that provides a lower bound for $\chi(G_X)$. For example, given a node with partial assignment $s_{u,v} = \text{True}$ and $s_{x,y} = \text{False}$, the current graph G_X has merged u and v and added the edge $\{x, y\}$. To obtain the MIS ILP formulation for this graph, one only has to fix the corresponding variables in the ILP to 1 and 0, respectively. Modern ILP solvers support incremental solving and take advantage of information from previous solving calls. This strong lower bound could provide great pruning power while potentially not being too costly to compute when making use of a fully incremental implementation.

One difficulty with this bound is that it does not immediately provide a witness of the bound in the form of a subgraph. Instead, a lower bound for the full graph is returned, also called a global lower bound here. One could create a reason clause like Equation (4.2) but for all of G_X, i.e., forbid one of the currently existing edges. This causes a backtrack but does not produce a strong conflict clause for the SAT solver to use. While the current node is cut off, only the exact current assignment and graph are forbidden instead of learning a shorter and likely reusable clause as with the previous subgraph-based bounds. There, any occurring graph G_X where the particular clique or pseudo Mycielskian exists as subgraph is excluded from the search.

One could also try to use the full list of decisions that lead to the current node as the conflict clause. However, this has the same issue as before that only the current node is cut off, and the provided clause is not relevant in other parts of the search tree. Both these approaches only lead to chronological backtracking so that little is gained, and the benefit is likely outweighed by the computational effort it took to compute such lower bounds.

In a cursory experiment, the list of decisions was used as the explanation clause for the clique bound, which led to poor results. As explained, these clauses can only prune the current node and nothing is learnt from the subgraph that causes the conflict. This might also be one factor why using the clique bound during the Dsatur algorithm of Furini, Gabrel, and Ternier [FGT17] did not lead to a gain in performance. While the current node is cut off, this is outweighed by the cost of computing the clique bound at each node without learning and applying the bound conflicts to other parts of the search tree. Combining the Dsatur algorithm with a mechanism to learn such information would potentially avoid this and increase performance.

A promising idea is to compute these bounds only on a chosen subgraph. If the subgraph is small enough but still provides a good lower bound for the chromatic number of the full graph, it can use the same explanations as for the clique or Mycielsky bound. Identifying the smallest subgraph with a large chromatic number is not straightforward; one heuristic is to iteratively remove the vertex of minimal degree until only a fixed number of vertices remain. This allowed Held, Cook, and Sewell [HCS12] to compute new lower bounds on some challenging instances. Choosing the subgraph this way has the further advantage that, for example, the auxiliary graph-based lower bound from Section 2.1.3 is easier to compute, as a denser graph leads to a smaller MIS instance. Determining a strategy of when and on which dense subgraph to compute this lower bound is a promising idea that could prove beneficial for the Zykov propagator, but was not implemented and tested in this thesis.

4.4.3 Adaptive Propagation

As neither the clique bound nor the Mycielsky bound are cheap to compute, both are only called after the unit propagation of the SAT solver and the transitivity propagation are finished. Further, the Mycielsky bound is more expensive than the greedy clique algorithm, requiring $O(n^3)$ bitset operations compared to the $O(n^2)$ of the latter. The clique bound is computed at every node, but this was observed to be too costly for the Mycielsky bound in [HK20]. They follow a strategy proposed by Stergiou [Ste08] in the context of propagators for constraint programming to apply the more expensive bound only at selected nodes. In particular, the Mycielsky bound is only computed after backtracking. If the stronger bound causes further backtracking, it is again computed until it does not lead to another bound violation. Additionally, a threshold t is set for the Mycielsky bound. Since it takes a maximal clique C as the starting subgraph, it is only executed if $k + 1 - |C| < t$, i.e., if the gap between the clique bound C and the target bound $k + 1$ is smaller than the threshold.

In constraint programming, the strategy of when to call a propagator is an important field of study itself, with a lot more depth than presented in the scope of this thesis. A survey and evaluation on adaptive constraint propagation in constraint programming is given in [Ste21]. Other than heuristics at which node a propagator should be executed, they also review methods to only run propagators on certain variables or for certain values. In the Zykov propagator, the transitivity propagator is always run to enforce the transitivity constraints. Moreover, the bound conflicts are a property of the graph G_X at the current node, so these approaches seem too granular to be of use for the clique and Mycielsky bound.

Another heuristic concerned with selecting for which nodes to execute more expensive propagators is PrePreak introduced by Woodward, Choueiry, and Bessiere [WCB18]. They propose a method to recognize thrashing, defined as repeatedly exploring similar (fruitless) subtrees during search, which leads to a lot of redundant work. In their approach, thrashing is detected by counting the number of backtracks at each level and geometrically adjusting the frequency of when a stronger propagator is called. The idea is that intelligently calling a stronger propagator can prune nodes of the search tree where thrashing takes place and not explore too many similar subtrees, without wasting runtime when the expensive propagator will not be effective enough. This was not implemented for the Zykov propagator in this thesis but might provide another boost to performance.

4.4.4 Improving Upper Bounds During Search

The previous focus was on pruning the search by finding lower bound conflicts and learning strong conflict clauses from the witness subgraph. This had the goal of speeding up solving of the instance by reducing the size of the search space. When looking for a k-coloring, one can of course stop traversing the search space when a k-coloring is found. Inside the SAT solver, this is done by finding a satisfying assignment that corresponds to a leaf in the Zykov tree of size k. With the user propagator, external algorithms could be used to also look for a k-coloring during the search. Indeed, the subproblems are again graph coloring problems and a k-coloring of one of them corresponds to a k-coloring of the original graph. Thus, instead of finding a k-coloring from a satisfying assignment, an external graph coloring algorithm can be called on the current graph G_X. If successful, there is no more need to traverse the search space with the satisfiability solver.

The same algorithms as presented in Section 2.2 can be used. As they are already called before the search to find an upper bound, it is of course redundant to call them again on the original graph, but the graph G_X based on the current partial assignment. Besides

running the Dsatur algorithm on this graph, another coloring heuristic more focused on speed is also tested. Namely, the greedy independent set sequential coloring procedure previously proposed and used in the context of maximum cliques [San+23; San+13], also called ISEQ. It takes advantage of the graph being represented as a bitset and uses their bit-parallelism to optimize the performed operations. It additionally uses the recolor technique mentioned in Section 2.2 to switch colors when this can lead to a better solution. As the name suggests, it is a sequential coloring heuristic and thus needs no time to dynamically select the next vertex to color contrary to the Dsatur algorithm. To further save runtime, no vertex ordering is computed before calling ISEQ. Instead, the fixed initial vertex order is used, restricted to the vertices in G_X. It is tested as a counterpart to Dsatur during the search since it is heavily focused on runtime and less on finding a great coloring. Like the Mycielsky bound, the graph coloring heuristic is only computed after backtracking.

4.4.5 Preprocessing of Subproblems

Besides computing improved lower and upper bounds on the subproblems encountered during search, one can again apply the preprocessing rules of Section 2.3. As the degree usually increases with contractions and edge additions, the removal of low-degree vertices is not considered. For dominated vertices, a dominated vertex v is not removed but merged into the other vertex u, which corresponds to setting the variable s_{uv} to true. While the effect of this preprocessing rule integrates nicely with the propagator, it is not clear whether a proper reason clause for this propagation exists. Only literals that must be set to true are propagated since otherwise the formula is unsatisfied. This is not necessarily the case with dominated vertices: if a k-coloring exists for G, then it also exists for G with the two dominated vertices merged, but it can also exist if the vertices are instead assigned different colors. Thus, setting $s_{uv} = \text{True}$ cannot be done as propagation since it does not necessarily have to hold for a satisfactory assignment to exist and can lead to incorrect clauses being learnt because of that. Instead of applying this preprocessing rule as propagation, it is simply chosen as a decision. The idea is that while it does not need to hold to find a satisfying assignment, it can be beneficial to make that decision in the search as it cannot increase the chromatic number. This potentially guides the search in the right direction.

The two further preprocessing rules of vertex fusion and edge addition require stricter assumptions but also yield stronger implications which can be useful in this context. They required one to look for a k-coloring and know of a subgraph with chromatic number k and determined that certain vertices need to be assigned the same color or need to be assigned different colors. This means that s_{uv} must be true or false, respectively. Due to this, these preprocessing rules can be used as propagations during search, and a reason clause can be provided. This was already done in [HK20], where they refer to these propagations as positive and negative pruning. The explanation of the propagations follows the same reasoning as that of the clique or Mycielsky bound, adding a literal for each edge in the involved subgraph plus the positive or negative literal to be propagated. Where the vertex fusion and edge addition were too cumbersome to use beforehand, they now integrate well with the Zykov propagator. When looking for a k-coloring, cliques are computed during the search for lower bound-based pruning. If a clique of size k is found, it can be used for the two preprocessing rules. As positive pruning is the less expensive rule, it is called at every node, while negative pruning again is only used after backtracking.

4.5 Transfer of Methods to Assignment Propagator

One can also design a propagator for the assignment encoding using similar ideas as for the Zykov propagator. It can be used to enforce that adjacent vertices get assigned different colors, and optionally the at-most-one color per vertex constraints. It is not immediate whether handling these binary clauses with a propagator would be beneficial. At least for the many at-most-one constraints, better results were observed in [Abí+13] when encoding the cardinality constraints. For this thesis, an assignment propagator was implemented that handles all constraints for adjacent vertices and also for at-most-one color per vertex.

Further, a custom decision strategy for the assignment encoding is tested. Instead of the default strategy of the SAT solver, the Dsatur strategy was implemented with the improvements proposed in [San12]. With this strategy, the search essentially follows the same steps as in the branch-and-bound Dsatur algorithm, as the same decisions are made. But the SAT approach learns more information during the search by analyzing conflicts and producing conflict clauses, which is not done in the Dsatur algorithm. The Dsatur decision generally leads to good performance in the context of graph coloring, as such it was also imitated by [HK20] in their propagator approach, as described in Section 4.3. Still, the default strategies of satisfiability solvers are well crafted for SAT instances and potentially make better decisions for learning stronger conflicts. Both strategies were implemented and will be tested in the experimental evaluation.

The search tree pruning techniques of Section 4.4 can potentially also be translated to work for the assignment propagator. While the graph does not change with different color assignments, a reduced graph can be built as previously detailed in [FGT17]. For a partial coloring, vertices that have been assigned the same color can be merged, and edges added to all vertices that have been assigned a different color. If a vertex is not assigned a color, it can be the case that a color was forbidden for it by that literal being false, so edges can also be added to uncolored vertices. This is something that does not occur with the Dsatur algorithm that only works with positive assignments.

Like the graph G_X in the Zykov propagator, the reduced graph can be used to compute lower bounds in the assignment propagator. If this lower bound determines no improved coloring can be found, this part of the search tree can be pruned. Options for computing such lower bounds are the clique and Mycielsky bound, or the auxiliary graph-based lower bound that was used in the Dsatur algorithm of [FGT17]. For the subgraph-based lower bounds, the subgraph was forbidden in the Zykov propagator by adding all edge literals to the reason clause. As the assignment propagator works only with assignments, the subgraph causing the bound conflict in the reduced graph has to be explained and forbidden by a clause using assignment variables. An attempt was made to find the right reason clause to explain the pruning of the search tree but was unsuccessful in the timeframe of this thesis. Such pruning using the clique bound did not achieve satisfactory results in the work of Furini, Gabrel, and Ternier, and using the auxiliary graph-based bound only improved the performance for relatively dense graphs. Nonetheless, with the learning capability of a satisfiability-based approach, this could lead to an improvement over just the assignment encoding.

4.6 Experimental Evaluation

This evaluation focuses on the performance of the Zykov propagator approach and again has three goals. First, the effect of the pruning techniques on the search tree is analyzed. Second, the different propagator configurations are evaluated against the well-performing

satisfiability encodings of Chapter 3. Finally, the different parts of the algorithm and their contribution to the performance are investigated in a factor analysis. The next subsections cover these goals, but first, the implementation details and the considered configurations are described. The Zykov propagator reported better results with bottom-up search and is used as the search strategy for all configurations presented.

All Zykov propagator algorithms use similar propagator-independent settings to those used for the previous Zykov-based encodings. In particular, the preprocessing of the instance is enabled, and the vertices are ordered again with a clique at the beginning. First, the default Zykov propagator algorithm configuration is presented, called ZP Default. It uses the default decision strategy of the underlying SAT solver and the clique and Mycielsky bound-based search tree pruning. The Mycielsky bound is only called after backtracking and with a threshold of 1. Search for the chromatic number is done bottom-up, using activation literals to enable fully incremental solving as described at the end of Section 4.4.1. Thus, the algorithm will mostly be concerned with proving unsatisfiability, and the ideas of Section 4.4.4 to find improved colorings during search are not used. The preprocessing of subproblems as presented in Section 4.4.5 is not used by default.

For the main evaluation, two other configurations are considered, described here by their differences to the default configuration. They are chosen to show the difference that pruning makes compared to only the transitivity propagation, and further, compare with the best performing configuration. Additionally, other configurations will play a role in the factor analysis of Section 4.6.2, but they are presented as needed and are not a focus in Sections 4.6.1 and 4.6.3.

ZP None Neither cliques nor the Mycielsky bound are used for pruning.

ZP Default Default configuration, as described above.

ZP Improved Uses positive pruning and deciding first on dominated vertices as decisions.

For comparison, gc-cdcl of Hébrard and Katsirelos [HK20] uses the clique and Mycielsky bound for pruning, the latter without any threshold. Further, they use the positive pruning technique and the Dsatur decision strategy as default and do not encode the cardinality constraints in the formula.

For the assignment propagator, the default configuration AP Default uses no additional features and only propagates the adjacency constraints and at-most-one color constraints. Since pruning techniques were not implemented for the assignment propagator, the only other configuration is AP Dsatur. It uses the Dsatur decision strategy instead of the default of the SAT solver and is analyzed in the factor analysis alongside the other Zykov propagator configurations.

4.6.1 Effect of Pruning on the Search Tree

With the CEGAR approach presented in Section 3.2.2, it was already shown that not all transitivity clauses are necessary to solve an instance. There, new transitivity clauses were iteratively added to the encoding, and the instance solved again. With the user propagator for satisfiability, propagations can be given without immediately adding the transitivity clause to the formula. Only when the propagation is part of a conflict is it added to explain the conflict. This has the potential to require even fewer transitivity clauses to solve an instance. Figure 4.2 gives a visual comparison for the number of transitivity clauses used by the Full Encoding, CEGAR Sparse, and the three Zykov propagator configurations.

Algorithm ZP base does no pruning and only handles the transitivity clauses. Even so, it still solves many instances with fewer transitivity clauses than CEGAR Sparse. Combining

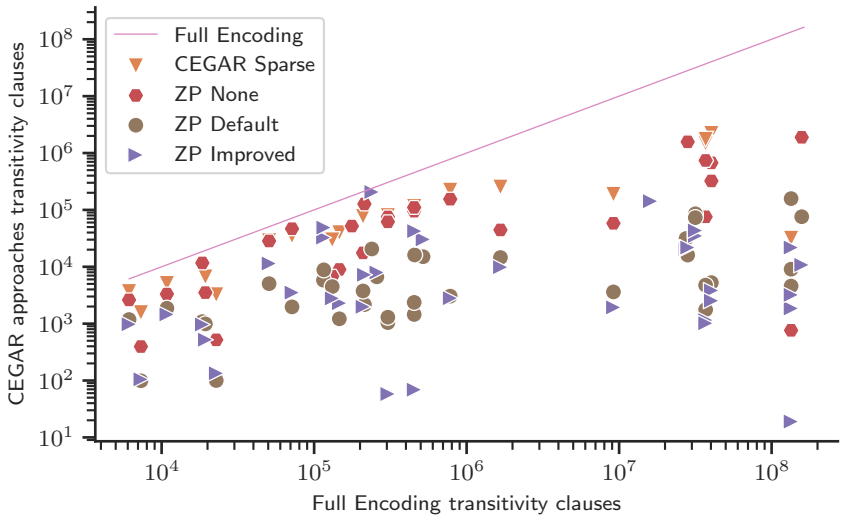


Figure 4.2: Comparative plot of required transitivity clauses in the Full Encoding, CEGAR Sparse and different Zykov propagator configurations

this with additionally pruning parts of the search tree, ZP Default and ZP Improved require even fewer transitivity clauses, often by a significant factor. While both variants with pruning behave similarly, ZP Improved tends to use fewer transitivity clauses in general, with some exceptions. One can see that even with ZP None, a large number of transitivity clauses are still required, although it improves upon CEGAR Sparse. The only way to avoid a larger number of clauses is by cutting away parts of the search space so that the clauses of that part do not need to be added to the instance. This is what happens for ZP Default and ZP Improved, which often use fewer clauses by one or more orders of magnitude.

In Figure 4.3, some characteristics of the search tree are visualized for the instances solved by all three algorithms but not in preprocessing. For the different propagators and graph coloring instances, the number of nodes and the maximum decision level of the tree are given as points in the scatter-plot. One can observe that the number of nodes required for solving an instance goes down significantly when using search tree pruning techniques compared to the Zykov propagator without pruning. This effect is still stronger for ZP Improved. For the maximum decision level of the tree, no similar trend is visible; for most instances, the level does not go down remarkably with pruning. This is likely due to the satisfiable instances, where the search reaches all the way down to a leaf node to produce a satisfiable assignment.

Figure 4.4 gives a more fine-grained picture of the search tree structure. For selected instances, the plot compares the tree size for algorithm configurations ZP None and ZP Improved. The nodes visited during the algorithms on the instances are collected, and the figure shows the level of the nodes on the x-axis, compared to how often nodes of that level were visited. This count is measured on the y-axis. The surface under each curve corresponds to the number of visited nodes in total. One can observe a strong difference in the number of nodes visited between the two algorithms. ZP None without pruning visits many more nodes than ZP Improved, which is able to prune large parts of the search tree. Further, the peak density, i.e., the level where the most nodes have been visited, is much lower and at a lower level for ZP Improved than for ZP None. Although these are just exemplary instances, selected because their level distributions are similar and fit in one plot, this shows the potential of the pruning techniques to significantly reduce the search effort.

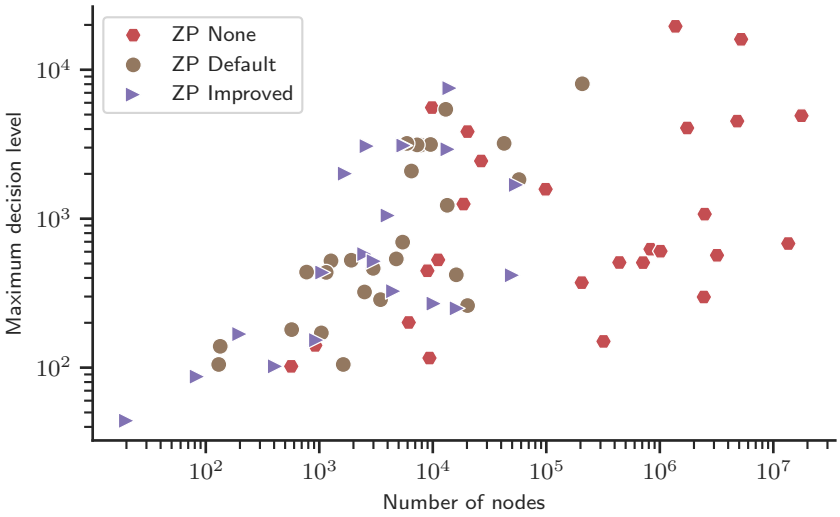


Figure 4.3: Comparative plot of the number of nodes and the maximum decision level in the search tree for different Zykov propagator configurations

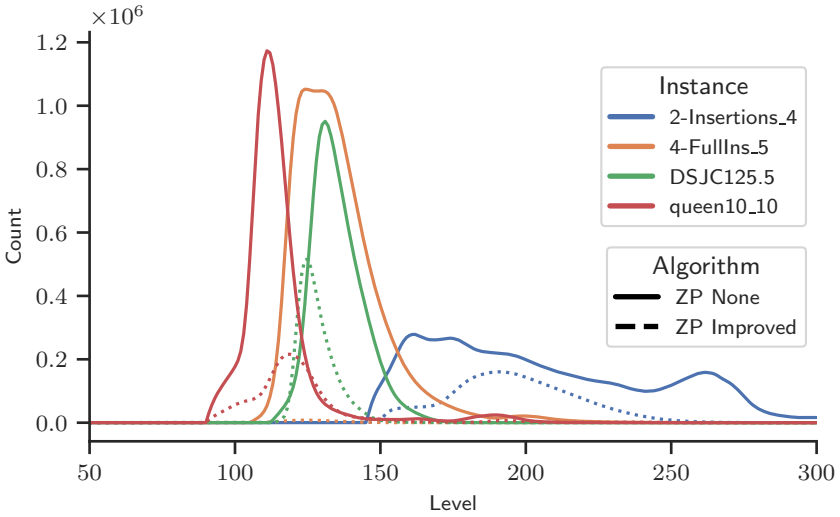


Figure 4.4: The plot visualizes how often nodes at each decision level are visited during the algorithm, for two different configurations and four example instances

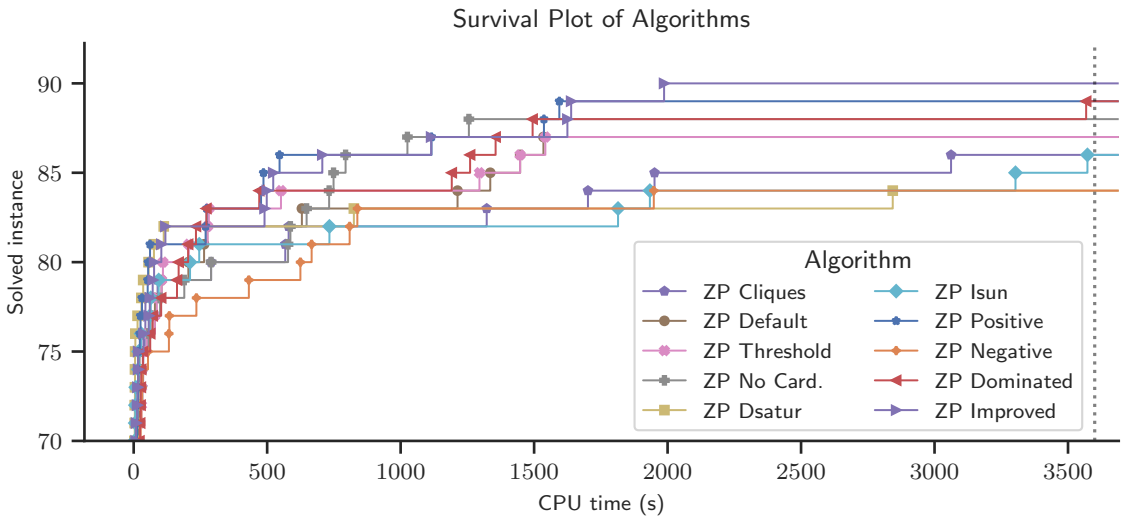


Figure 4.5: Surival plot for the different Zykov propagator configurations, providing an overview of what factors impact performance

4.6.2 Factor Analysis

To analyze the performance impact and identify beneficial strategies, this section presents further configurations. They either leave out or add features to the default settings and are described next.

ZP Cliques	Only cliques are used for pruning.
ZP Threshold	Using a threshold of 10 instead of 1 for when to call the Mycielsky bound.
ZP No Card	Does not add the encoding of the cardinality constraints to the formula.
ZP Dsatur	Uses the decision strategy that emulates Dsatur decisions.
ZP ISUN	Uses the ISUN decision strategy.
ZP Positive	Uses positive pruning.
ZP Negative	Uses negative pruning.
ZP Dominated	Dominated vertices are chosen first as decisions, before default strategy.

A performance overview of all Zykov-propagator configurations is given in the survival plot of Figure 4.5. Although quite densely packed for smaller time limits, differences in performance are noticeable when considering the total number of instances solved.

First, the problem-specific decision strategies used in ZP Dsatur and ZP ISUN seem to degrade performance. Although taking advantage of the structure of the graph coloring problem, the default decision strategy of the SAT solver is more successful when tackling the problem as a satisfiability instance. The configuration of ZP Cliques that does not use the Mycielsky bound for pruning also performs slightly worse. This suggests that the additional pruning power of the Mycielsky bound provides an improvement to the algorithm. As observed in [HK20], negative pruning is too costly to provide a benefit. This is despite adapting it to only be called after backtracking, like the more expensive Mycielsky bound.

Three configurations using additional features show a positive effect on performance. These are ZP No Card., ZP Positive, and ZP Dominated. The first solves one more

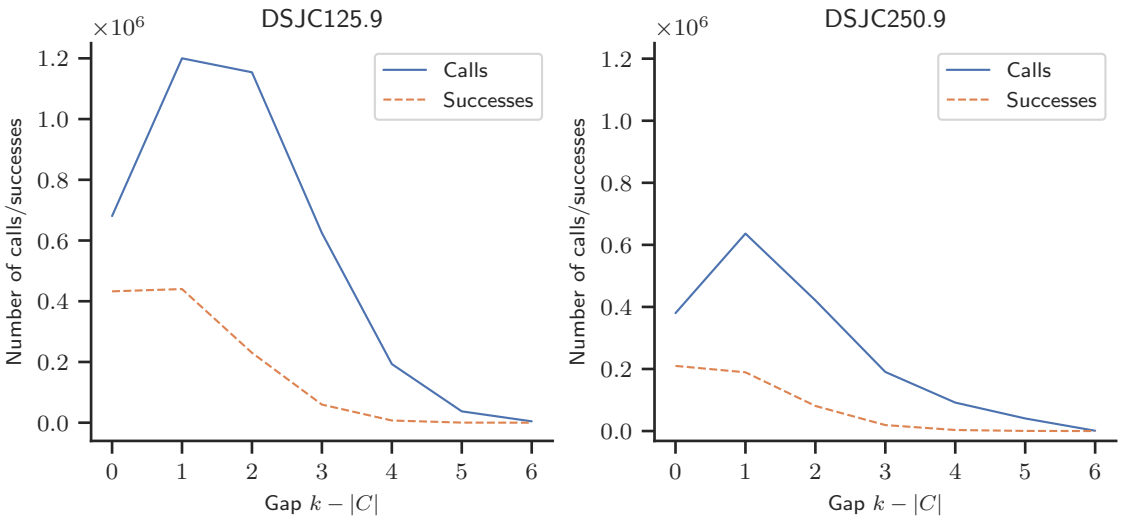


Figure 4.6: The number of calls and number of successfully computed bound conflicts are plotted for different values of $k - |C|$ for the two instances DSJC125.9 and DSJC250.9. Here, the algorithm is looking for a k-coloring and $|C|$ is a maximal clique computed before calling the Mycielsky bound

instance and is slightly faster than the default configuration. This shows that the additional reasoning strength of encoding the cardinality constraints is not necessary. The propagator can successfully handle them by itself, even slightly improving the results. The latter two configurations each solve two more instances than ZP Default, confirming the benefit of positive pruning as observed in [HK20]. Further, the idea proposed in this thesis of using dominated vertices as "probably good" decisions in the propagator seems to pay off as well. Combining these two methods within ZP Improved leads to an additional improvement and is the best performing configuration, solving 90 instances.

Not mentioned so far is ZP Threshold, which uses a threshold of 10 for when to compute Mycielsky bounds. By default, this value is 1 so that when looking for a k-coloring, the pruning only takes place if the gap between k and $|C|$ is quite small. Here C is a maximal clique computed before deciding to call the Mycielsky algorithm. This has no significant effect on performance, likely due to search being done bottom-up and the gap between k and $|C|$ being usually small anyway. In fact, only on a few instances is the Mycielsky bound called for gaps larger than two, and only for instances DSJC125.9 and DSJC250.9 is the bound called for a gap of 6. Figure 4.6 shows the success of the bound for each size of the gap, comparing the number of times the Mycielsky bound is called, and the number of times it successfully produced a bound conflict. Despite the additional pruning, ZP Threshold is only slightly faster but does not solve more instances than ZP Default.

When considering top-down search, the threshold of when to compute the Mycielsky bound plays a more important rule. As search is done top-down, k is often much larger than $|C|$, so that a larger gap does allow for many more calls to the Mycielsky bound. With this search strategy, ZP Threshold actually solves two more instances than the default in top-down. However, both do not perform as well as the default configuration does in bottom-up search.

In Section 4.4.4, the idea was proposed to also look for better upper bounds during search. Instead of pruning parts of the search tree, if a k-coloring is found by a heuristic on a subproblem, the search can be stopped as the goal was achieved. Since the subproblems at each node are again coloring problems, any previous coloring heuristic can be reused.

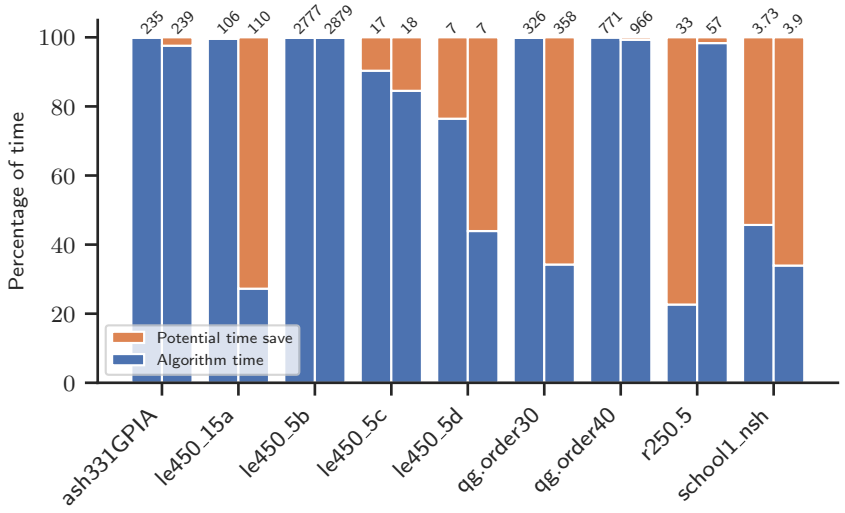


Figure 4.7: Potential time saved by employing coloring heuristics during search. The left and right bars of an instance correspond to the ISEQ and the Dsatur heuristic, respectively. The number above each bar is the total algorithm time in seconds

In this thesis, two algorithms were tested. The slower Dsatur heuristic, which computes the vertex ordering dynamically but produces better results, and ISEQ, which takes the vertex ordering as is and very quickly computes a coloring. The experiments for this idea are not as detailed as for the previous configurations. Their aim is simply to obtain an estimate of how much such an approach could help in finding an optimal coloring earlier. The tests were done for top-down search where finding better coloring during search is more likely to have a benefit. To measure the potential of this approach, the time-point where the coloring heuristic finds a k-coloring, and the time-point where the SAT solver finds a k-coloring are compared. The difference between these is then the potential time saved if the search had stopped after the heuristic computed the coloring.

The results of this evaluation are visualized in Figure 4.7. Only instances are listed that were solved but not in preprocessing, and where the heuristic provided a time improvement of at least one second. The left bars correspond to the ISEQ heuristic, and the right bars are for the Dsatur heuristic. The orange part represents how much earlier the algorithm could have terminated, that is, how much time could have been saved with the respective coloring heuristic. For example, on the instance qg.order30 there is no time saved by the ISEQ heuristic, i.e., it did not find an improving coloring during search. The Dsatur heuristic finds better colorings and does so much earlier, as 65% of the runtime could have been saved compared to running the algorithm without coloring heuristic. Though the heuristic only had a noticeable time saving potential on nine instances, in cases where it is effective, it can be significant. Most notably are instances r250.5 for ISEQ and le450_15a and qg.order30 for Dsatur. On these, the algorithm could have terminated earlier and saved about 60%–70% of runtime. Though the Dsatur heuristic is more expensive, as seen by the total runtime of the algorithm above each bar, it is also able to find improved colorings much earlier than ISEQ.

The results of Figure 4.7 look promising but are limited to instances that were solved and where the coloring heuristic provided some potential speedup. Further, the additional coloring heuristic during search incur their own computational cost. For comparison, using ISEQ leads to an increase of 3.7%, measured over the solved instances, and for Dsatur the runtime increases by 12.5%. While on a few instances the coloring heuristics showed

good potential, this might not be worthwhile over the additional runtime they cost over all instances, especially the ones where they are unsuccessful.

Finally, the further configuration AP Dsatur of the assignment propagator was also tested. In it, the Dsatur strategy is used to pick the next vertex and assign it a color. Choosing such a problem-specific decision strategy for the assignment propagator degrades performance, solving only 84 instances compared to the 88 instances with the strategy of the SAT solver. Again, the default of the SAT solver is more appropriate to tackle the satisfiability problem. This is despite not directly taking advantage of the graph coloring structure, which matches the observations for the Zykov propagator. Likewise, this implies that the Dsatur strategy, as used in the exact Dsatur-based branch-and-bound algorithm or coloring heuristics, is not the best strategy. Therefore, improvements to existing algorithms not based on satisfiability could be made by finding a way to translate the SAT solver decisions to, for example, the branch-and-bound algorithm.

4.6.3 Comparison With Satisfiability Encodings

Lastly, the competitiveness of the Zykov propagator approach is evaluated. The Full Encoding and the propagators are all based on the same Zykov property, though the former builds a static encoding, and the latter solves the problem by enforcing constraints with a user propagator. Further, the Zykov propagators can interact with the solving process instead of using the SAT solver as a black box and help reduce the search effort by pruning parts of the search tree. The Full Encoding is chosen for comparison over one of the CEGAR algorithms as it performed better. The partial order encoding is compared to as well, since it performed best of the satisfiability encodings presented in Chapter 3. Additionally, the three Zykov propagator configurations are tested. First, ZP None to see what impact just the propagation of the transitivity clauses can have over the Full Encoding. Then, ZP Default to see whether lower bound-based pruning leads to a noticeable reduction in search effort by itself and thus solving more instances. Finally, ZP Improved uses positive pruning and the strategy of first deciding on dominated vertices. The propagator algorithm gc-cdcl of [HK20] is used for comparison as the implementation of the Zykov propagators here is based on their new ideas. Next to that, CliColCom [HKH22] and POP-S [FJM24] are also used as comparison to SAT-based algorithms from the literature.

The survival plot in Figure 4.8 visualizes the performance of the listed algorithms on the DIMACS benchmark set described in Section 1.4. Comparing the Full Encoding and ZP None, the latter generally solves instances faster and solves 77 instances, while the Full Encoding solves only 74. This shows the advantage of using a propagator to handle the transitivity constraints. While the CEGAR approach also led to fewer transitivity clauses being required, the frequent resolving caused more overhead so that the Full Encoding with all clauses performed better. Here, the propagator manages to solve instances needing fewer transitivity clauses and improves on the performance of the Full Encoding.

Despite this, only focusing on minimizing the number of transitivity clauses does not yield competitive results by itself. However, using search tree pruning techniques as described in Section 4.4 for the two configurations ZP Default and ZP Improved leads to a significant increase in performance. The algorithms can solve 87 and 90 instances, respectively. Further, this improves on the implementation gc-cdcl of [HK20], that solves 82 instances within one hour. This is likely because the implementation of this thesis is a pure satisfiability-based algorithm, while that of Hébrard and Katsirelos is a constraint programming/SAT hybrid approach. Further, they design a problem-specific decision strategy emulating the Dsatur algorithm choices, while here the default strategy of the SAT solver is used.

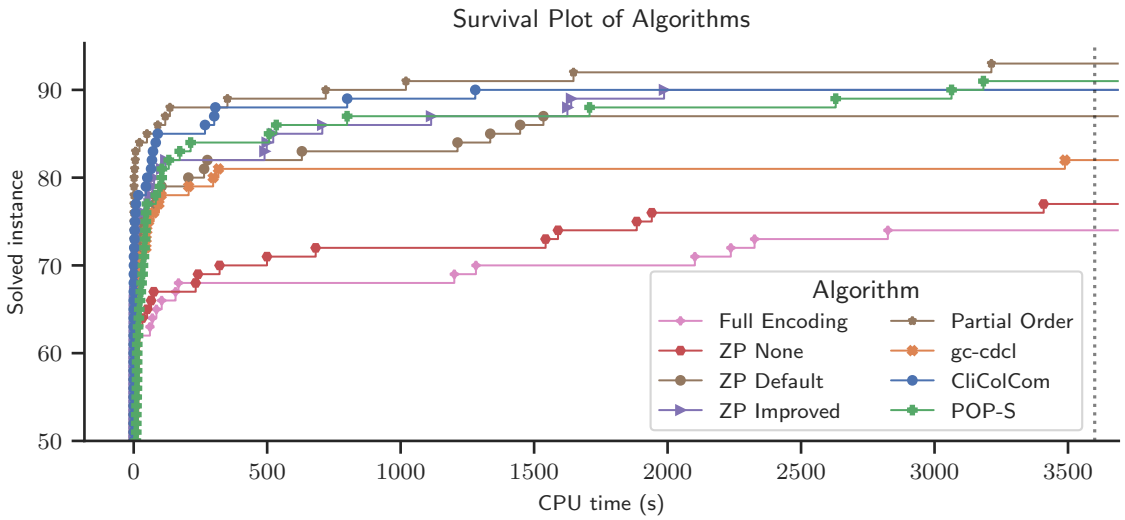


Figure 4.8: Surival plot for the partial order encoding, Full Encoding, the three Zykov proapgator configurations and algorithms gc-cdcl, CliColCom and POP-S

The partial order encoding was the best performing satisfiability encoding of Chapter 3 and even outperformed the state-of-the-art satisfiability-based methods of the literature. As such, it is not surprising that it is again the best performing method in this evaluation. It solves 93 instances of the benchmark set, 3 more than the best performing Zykov propagator configuration ZP Improved. It should be highlighted that ZP Improved is still a very competitive algorithm. Until the work of Heule, Karahalios, and Hoeve [HKH22], gc-cdcl was a state-of-the-art approach to graph coloring. It solves 82 instances of the benchmark set, and the newer CliColCom manages to improve on this by solving 90 instances, the same number of instances that ZP Improved is able to solve. Only recently in the work of Faber, Jabrayilov, and Mutzel [FJM24] was the partial order encoding POP-S presented that manages to solve 91 instances. It takes a bit longer, but after 5900 seconds, ZP Improved solves wap06a and matches the 91 solved instances. Considering these results, ZP Improved can clearly compete with state-of-the-art satisfiability-based methods from the literature.

The results obtained for the Assignment propagator were rather underwhelming. Contrasting the increased performance when using a propagator for the Zykov-based encoding, the propagator for the assignment encoding did not yield better results. In fact, its performance got worse, solving only 88 instances compared to the 91 that the assignment encoding with at-most-one constraints manages. There could be multiple reasons for this.

The most likely one is that the number of clauses in the assignment encoding is not large enough to lead to an advantage for the propagator. The $O(m \cdot k)$ clauses for a k-coloring problem can simply all be added without causing the same overhead as the $O(n^3)$ transitivity clauses did. Further, almost all the clauses are binary clauses, which can be resolved very efficiently by modern SAT solvers. Additionally, the focus of this thesis is on the Zykov-based encoding and the Zykov propagator, the latter of which produces great results. As such, the implementation of the assignment propagator is not as well optimized as the Zykov propagator, providing further explanation for the lacking performance.

A more detailed but still aggregated report of the results is given here in Table 4.1. The format is the same as in the evaluation of Chapter 3, with instances being grouped by their class and performance being reported in terms of solved instances and average lower and upper bounds. Table A.2 in the appendix reports the performance for each instance.

Table 4.1: Aggregated performance results of different Zykov propagator configurations and SAT encodings

		ZP None			ZP Default			ZP Improved			AP Default			Partial Order			gc-cdcl↑			POP-S		CCC
Class	#	Opt.	*lb*	*ub*	Opt.	*lb*	*ub*	Opt.	*lb*	*ub*	Opt.	*lb*	*ub*	Opt.	*lb*	*ub*	Opt.	*lb*	*ub*	Opt.	*lb*	*ub*	Opt.
Books	5	5	11.20	11.20	5	11.20	11.20	5	11.20	11.20	5	11.20	11.20	5	11.20	11.20	5	11.20	11.20	5	11.20	11.20	5
CX000	3	0	37.00	370.67	0	37.00	370.67	0	37.33	371.67	0	**38.33**	370.67	0	37.33	370.67	0	34.33	384.00	0	30.67	383.00	0
DSJC	12	1	24.92	74.75	1	25.50	74.75	1	**25.67**	74.75	1	25.58	74.75	1	25.58	74.75	1	22.67	75.50	1	22.67	75.75	1
DSJR	3	2	73.00	74.00	2	73.00	74.00	3	73.00	73.00	2	73.00	74.00	3	73.00	73.00	1	71.00	76.00	1	71.33	77.33	3
FullIns	14	12	6.64	6.79	14	6.79	6.79	14	6.79	6.79	14	6.79	6.79	14	6.79	6.79	13	6.71	6.79	14	6.79	6.79	14
GPIA	5	1	5.60	6.40	2	5.60	6.20	4	5.60	5.80	5	5.60	5.60	5	5.60	5.60	4	5.60	6.00	5	5.60	5.60	4
Insertions	11	3	3.64	5.18	4	4.27	5.18	4	4.09	5.18	4	4.27	5.18	4	4.27	5.18	2	3.73	5.18	4	3.64	5.18	4
flat	6	0	14.67	79.00	0	15.67	79.00	0	15.67	79.00	0	15.83	79.00	0	15.83	79.00	0	14.00	78.83	0	15.00	**78.00**	0
fpsol2	3	3	41.67	41.67	3	41.67	41.67	3	41.67	41.67	3	41.67	41.67	3	41.67	41.67	3	41.67	41.67	3	41.67	41.67	3
games120	1	1	9.00	9.00	1	9.00	9.00	1	9.00	9.00	1	9.00	9.00	1	9.00	9.00	1	9.00	9.00	1	9.00	9.00	1
inithx	3	3	38.67	38.67	3	38.67	38.67	3	38.67	38.67	3	38.67	38.67	3	38.67	38.67	3	38.67	38.67	3	38.67	38.67	3
latin	1	0	90.00	137.00	0	90.00	137.00	0	90.00	137.00	0	90.00	137.00	0	90.00	137.00	0	90.00	**125.00**	0	90.00	132.00	0
le450	12	7	15.00	17.75	10	15.00	15.75	10	15.00	15.75	8	15.00	17.58	8	15.00	17.58	8	15.00	17.00	8	15.00	17.08	10
miles	5	5	34.80	34.80	5	34.80	34.80	5	34.80	34.80	5	34.80	34.80	5	34.80	34.80	5	34.80	34.80	5	34.80	34.80	5
mug	4	4	4.00	4.00	4	4.00	4.00	4	4.00	4.00	4	4.00	4.00	4	4.00	4.00	4	4.00	4.00	4	4.00	4.00	4
mulsol	5	5	34.60	34.60	5	34.60	34.60	5	34.60	34.60	5	34.60	34.60	5	34.60	34.60	5	34.60	34.60	5	34.60	34.60	5
myciel	5	5	6.00	6.00	5	6.00	6.00	5	6.00	6.00	5	6.00	6.00	5	6.00	6.00	5	6.00	6.00	4	5.60	6.00	4
qg	4	1	57.50	58.75	2	57.50	58.25	2	57.50	58.50	3	57.50	58.00	3	57.50	58.00	3	57.50	58.00	3	57.50	61.50	3
queen	13	5	10.77	13.08	8	10.92	12.46	8	10.92	12.46	6	10.85	12.92	8	10.92	12.46	6	10.85	13.00	8	10.92	12.92	5
r	9	8	**64.00**	**64.56**	7	63.78	65.22	7	63.67	65.22	7	63.78	65.22	8	63.78	64.67	7	62.33	67.00	7	62.22	66.11	7
school	2	2	14.00	14.00	2	14.00	14.00	2	14.00	14.00	2	14.00	14.00	2	14.00	14.00	2	14.00	14.00	2	14.00	14.00	2
wap	8	1	41.38	46.88	1	41.38	46.88	1	41.38	46.88	2	41.38	46.50	3	41.38	46.00	1	41.38	47.00	**5**	41.38	**45.00**	4
zeroin	3	3	36.33	36.33	3	36.33	36.33	3	36.33	36.33	3	36.33	36.33	3	36.33	36.33	3	36.33	36.33	3	36.33	36.33	3
Total	137	77	23.80	39.68	87	23.96	39.47	90	23.96	**39.46**	88	**24.00**	39.62	**93**	23.99	39.49	82	23.39	40.04	91	23.34	40.00	90

Chapter 5

Conclusion

In this thesis, different satisfiability-based approaches to exactly solving the graph coloring problem have been discussed. Additionally, a strong preprocessing routine was described, and a novel preprocessing method to reduce the problem size was proposed and tested. The graph coloring algorithms focused on two methods: satisfiability encodings that use the SAT solver as a blackbox, and user propagators to directly interact with the solving procedure and prune parts of the search tree with external information. Both approaches were shown to be very competitive with state-of-the-art methods from the literature.

Various satisfiability encodings were presented in Chapter 3, among them the direct encodings that include the assignment encoding and the partial order encoding. Despite their simplicity and inherent symmetry, they produced great results when combined with proper preprocessing and fixing the color of vertices in a large clique. In particular, the partial order encoding showed to be the best performing algorithm among all methods of this thesis and solved more instances than any other approach, including recent SAT-based algorithms from the literature. Further, it solved the previously open instance wap07a.

Besides the direct encodings, different methods to solve the graph coloring problem with an encoding based on the Zykov property were investigated. In particular, an implementation of the algorithm introduced in [Glo+19] was described in detail and several improvements were proposed. Originally aiming to reproduce their excellent results as their source code is not available anymore, very different trends in solving effort and time were observed, and the method solved a lot fewer instances than was reported in their paper. Further, an implementation as close as possible to the description in [Glo+19] produced even worse results. These observations suggest that either many essential implementation details were omitted in the paper, or something was incorrect in their computation of their experimental results. Despite a newly proposed check algorithm that is faster and performs better than the original, the Full Encoding outperforms the CEGAR strategy. Although not competitive with the direct encodings or algorithms from the literature, the Full Encoding managed to solve the previously open instance r1000.1c of the DIMACS benchmark set.

The Full Encoding and CEGAR variants were concluded to be too static to compete with the direct encodings. Although the number of transitivity clauses could be reduced, the search space remained too large to achieve competitive performance by itself. Therefore, Chapter 4 focused on implementing a user propagator to interact with the solving procedure and use external information to prune the search tree. This built upon and improved the ideas of Hébrard and Katsirelos [HK20] that implemented such a propagator in their constraint programming/SAT hybrid algorithm. Here, a pure SAT-based implementation is detailed, using the recent interface from [Faz+23]. Taking advantage of the incremental features such a SAT approach offers, assumptions were used to implement a fully incremental bottom-up search for the graph coloring problem. Bottom-up solving allows for stronger pruning during search, which leads to significant performance gains compared to the implementation of [HK20]. Although not beating the great performance of the partial

order encoding, this still proves the Zykov propagator approach to be competitive with the direct encodings and state-of-the-art methods of the literature. A factor analysis discussed the different components of the algorithm and their performance impact.

5.1 Discussion and Outlook

The direct encodings performed extremely well for their simplicity, showing the strength of modern satisfiability solvers to handle instances with many inherent symmetries. This is mainly due to the CDCL paradigm that enables the solver to learn from previous conflicts and use that information in other parts of the search. Comparing the assignment encoding and the partial order encoding showed how important the chosen translation of the graph coloring problem into a SAT formula is. For the assignment encoding, conflict clauses mostly contained negative literals that correspond to disallowing a color for a vertex. This can be quite a weak property if many colors are available. For the partial order encoding, a negative or positive literal means much more: a vertex must have color larger or smaller than a certain number, which is much more restricting when further exploring the search space. It is an interesting direction to explore other direct encodings that provide their own new strengths in tackling the graph coloring problem.

As mentioned previously, only reducing the number of transitivity clauses is not enough to make Zykov-based encodings perform as well as the direct encodings. This means the CEGAR approaches, even if implemented in such a way to outperform the Full Encoding, are still limited by the size of the search space. Therefore, it is unclear what and if anything can boost the performance of a CEGAR-based approach to that of the direct encodings if only working with a satisfiability encoding.

The Zykov propagator addresses this shortcoming by not treating the SAT solver as a blackbox but using the external propagator interface to prune the search tree with externally computed information. This information is unavailable to the satisfiability solver if a static encoding is used. In the original work of [HK20] and in this thesis, the clique and Mycielsky bound were used to produce bound conflicts, as they are based on determining a witness subgraph. These had the benefit of also providing strong conflict clauses to be reused during the search instead of causing a single chronological backtrack. This made up for their computational cost and proved effective in reducing the search effort.

The idea of using other lower bounds for such pruning is also promising, for example, the auxiliary graph-based lower bound and how it could be used were discussed in Section 4.4.2. The bound itself is quite strong but also costly, as observed in [FGT17] that used it to prune nodes in a Dsatur-based branch-and-bound algorithm. It was explained how the bound could be used inside the Zykov propagator, and a design was presented that takes advantage of reusing the ILP formulation instead of rebuilding it at every node. Although not implemented, this additional lower bound could provide further pruning potential and lead to even better performance by providing a larger reduction of the search space.

To use global lower bounds such as the auxiliary graph-based bound inside the Zykov propagator, two issues will need to be addressed. First, as they are not subgraph-based like the clique or Mycielsky bound, they do not provide a strong explanation of the part of the problem that caused the bound conflict. This problem was already discussed in Section 4.4.2, and most promising seems to be choosing a small dense subgraph on which to compute the bounds. Should this produce a bound conflict, the smaller subgraph potentially still provides a strong conflict clause. Further, the auxiliary graph-based bound or fractional chromatic number are much more efficient to compute on dense graphs. Therefore, another opportunity to improve the algorithm could be to design a good strategy that determines

the right subgraph on which to compute such stronger global bounds.

Second, to balance pruning power with computational cost, an important factor is the strategy of when to compute which bound. This question was already considered in the Zykov propagator of this thesis, as the clique bound was computed at every node, and the Mycielsky bound only after backtracking. The experimental evaluation confirmed this to be a good choice for the Zykov propagator and the bounds that were considered. A brief review of other ideas from the constraint programming literature was done in Section 4.4.3. Finding a good strategy for propagation becomes additionally important if more expensive bounds are to be used during search, such as the auxiliary graph-based bound or the fractional chromatic number. The goal is to recognize when a node is likely able to be pruned and only calling the expensive bounds on such nodes. Or further, recognizing where the pruning is important to avoid thrashing, i.e., the exploration of similar fruitless subtrees. Even a more expensive bound is likely to be beneficial if such subtrees can be cut off. The PrePeak heuristic [WCB18] could be an interesting idea to apply to the Zykov propagator with such bounds. It aims at recognizing thrashing and dynamically adjusts the levels at which to call stronger propagators, so that strong lower bounds are only used at nodes where it pays off.

It was mentioned in Section 4.5 that the pruning techniques that proved effective for the Zykov propagator might also be used inside the Assignment propagator. The assignment propagator that only handles the propagation of constraints did not show improved performance by itself, likely due to the SAT solver being able to handle the fewer constraints more effectively. Furini, Gabrel, and Ternier [FGT17] attempted to use clique-based pruning for their Dsatur-based branch-and-bound implementation and did not report improved results.

However, combining the produced bound conflicts with the learning capabilities of the satisfiability solver, this could still lead to an improvement inside an assignment propagator. The bound would not only be used to prune the current node, but further a conflict clause added to the formula which could prune other parts of the search tree without computing the bound another time. This would require two further things. First, the assignment encoding visits many more nodes than a Zykov-based encoding does, due to color symmetries caused by working with an explicit coloring, as was observed in Section 3.2. As such, new strategies have to be designed for when to compute the lower bounds as calling the clique bound at every node is likely too expensive. On top of this, a very efficient implementation is required to keep the reduced graph of the partial coloring in sync with the assignments made during search. Potentially, this could be done in a lazy way to avoid some of the computational overhead, but many new implementation details need to be described for this. Second, it is not immediately clear what reason clause can be used to explain the pruning for the assignment propagator. The partial coloring leads to a reduced graph on which the bound conflict is produced; for the subgraph-based clique and Mycielsky bounds, this witness subgraph needs to be translated into a conflict of the assignment variables. Further, the same reduced graph and thus its bound conflict can be produced by different coloring permutations. To fully take advantage of the pruning potential, a conflict clause needs to be produced that also eliminates symmetric conflicts from the search.

These open questions provide some obstacles to a successful implementation of an assignment propagator that takes full advantage of the pruning possibilities. Addressing these will require considerable work, but with the recent success of SAT-based methods for graph coloring and the impressive results of the Zykov propagator, this is a promising direction to further improve performance. Everything mentioned for the assignment propagator applies similarly to a potential propagator for the partial order encoding. With

the partial order variables, likely stronger conflict clauses can be learnt which strengthens the pruning potential of the lower bounds. As such, designing a propagator for the partial order encoding that addresses the previously mentioned hurdles is another promising avenue for future research.

Both for the Zykov propagator and the assignment propagator, designing a custom decision strategy that takes advantage of the problem structure did not lead to better results. Despite good empirical evidence supporting the Dsatur strategy in graph coloring algorithm, the default decision strategy of the SAT solver performed better. This might be due to the default strategy simply being well tuned for SAT problems and therefore beating decision strategies that do not treat the problem as such. This is reasonable, as the Dsatur strategy only makes positive assignments, i.e., choosing a vertex and assigning it a certain color. For a satisfiability-based approach, however, it might be advantageous to also make negative decisions, i.e., decide to forbid a color for a vertex instead of assigning it.

Possibly, this also hints at the Dsatur strategy not being the best choice for the graph coloring problem after all. This would imply that taking parts of the SAT solver and translating them into other algorithms could make them better. One example is the SAT solver decision strategy that performs better than the decisions based on the Dsatur strategy. If one can find a way to further translate the learning capabilities and non-chronological backtracking into one of the existing algorithms, it is not unreasonable this should also improve their performance. This was done for the Dsatur-based branch-and-bound algorithm in [Zho+14]. They showed improved results over the branch-and-bound algorithm without CDCL features and observed the search tree to be significantly smaller. In their implementation called cdclGCP, the Dsatur decision strategy is used. Based on this previous success, it seems promising to transfer SAT solving techniques to other graph coloring algorithms as well.

Appendix A

Appendix

Table A.1: Detailed performance results of different satisfiability encodings

	Assignment			Assignment (AMO)			Partial Order			Full Encoding			CEGAR Sparse			gc-cdcl			POP-S			CCC
Instance	*lb*	*ub*	time	*lb*	*ub*	time	*lb*	*ub*	time	*lb*	*ub*	time	*lb*	*ub*	time	*lb*	*ub*	time	*lb*	*ub*	time	time
1-FullIns_3	4	4	0.0	4	4	0.0	4	4	0.0	4	4	0.0	4	4	0.0	4	4	0.0	4	4	0.4	0.0
1-FullIns_4	5	5	0.1	5	5	0.1	5	5	0.1	5	5	0.1	5	5	0.1	5	5	0.1	5	5	0.4	0.0
1-FullIns_5	6	6	0.2	6	6	0.1	6	6	0.1	6	6	0.1	6	6	0.2	6	6	0.8	6	6	1.9	0.3
1-Insertions_4	5	5	0.5	5	5	0.5	5	5	0.6	5	5	1201	4	5	tl	4	5	tl	5	5	1.8	2.2
1-Insertions_5	5	6	tl	5	6	tl	5	6	tl	4	6	tl	4	6	tl	4	6	tl	4	6	tl	tl
1-Insertions_6	5	7	tl	5	7	tl	5	7	tl	3	7	tl	3	7	tl	4	7	tl	4	7	tl	tl
2-FullIns_3	5	5	0.1	5	5	0.1	5	5	0.1	5	5	0.1	5	5	0.1	5	5	0.0	5	5	0.3	0.0
2-FullIns_4	6	6	0.1	6	6	0.1	6	6	0.1	6	6	0.1	6	6	0.2	6	6	0.4	6	6	0.8	0.1
2-FullIns_5	7	7	0.2	7	7	0.2	7	7	0.2	7	7	5.2	7	7	9.1	7	7	297.6	7	7	7.5	3.6
2-Insertions_3	4	4	0.0	4	4	0.0	4	4	0.0	4	4	4.5	4	4	26.9	4	4	0.7	4	4	0.2	0.0
2-Insertions_4	4	5	tl	4	5	tl	4	5	tl	4	5	tl	4	5	tl	4	5	tl	3	5	tl	tl
2-Insertions_5	4	6	tl	4	6	tl	4	6	tl	3	6	tl	3	6	tl	4	6	tl	3	6	tl	tl
3-FullIns_3	6	6	0.1	6	6	0.1	6	6	0.1	6	6	0.1	6	6	0.1	6	6	0.1	6	6	0.3	0.0
3-FullIns_4	7	7	0.1	7	7	0.2	7	7	0.2	7	7	0.2	7	7	0.2	7	7	1.0	7	7	1.7	0.2
3-FullIns_5	8	8	7.7	8	8	7.7	8	8	7.0	7	8	tl	7	8	tl	8	8	26.9	8	8	40.6	45.8
3-Insertions_3	4	4	0.1	4	4	0.1	4	4	0.1	4	4	85.1	4	4	636.9	4	4	14.0	4	4	0.4	0.1
3-Insertions_4	4	5	tl	4	5	tl	4	5	tl	3	5	tl	3	5	tl	4	5	tl	3	5	tl	tl
3-Insertions_5	4	6	tl	4	6	tl	4	6	tl	2	6	tl	3	6	tl	3	6	tl	3	6	tl	tl
4-FullIns_3	7	7	0.1	7	7	0.1	7	7	0.1	7	7	0.1	7	7	0.1	7	7	0.1	7	7	0.3	0.0
4-FullIns_4	8	8	0.2	8	8	0.2	8	8	0.2	8	8	0.2	8	8	0.2	8	8	3.5	8	8	4.2	1.1
4-FullIns_5	9	9	119.0	9	9	99.5	9	9	90.4	8	9	tl	8	9	tl	8	9	tl	9	9	213.4	799.9
4-Insertions_3	4	4	0.1	4	4	0.2	4	4	0.1	4	4	1282	3	4	tl	3	4	tl	4	4	0.4	0.2
4-Insertions_4	4	5	tl	4	5	tl	4	5	tl	3	5	tl	3	5	tl	3	5	tl	3	5	tl	tl
5-FullIns_3	8	8	0.1	8	8	0.1	8	8	0.1	8	8	0.1	8	8	0.1	8	8	0.0	8	8	0.4	0.1
5-FullIns_4	9	9	0.2	9	9	0.2	9	9	0.2	9	9	0.3	9	9	0.2	9	9	6.8	9	9	10.7	8.6
C2000.5	18	206	tl	18	206	tl	18	206	tl	14	206	tl	16	206	tl	16	211	tl	17	210	tl	tl
C2000.9	78	530	tl	78	530	tl	76	530	tl	78	530	tl	78	530	tl	70	560	tl	61	562	tl	tl
C4000.5	19	376	tl	19	376	tl	18	376	tl	15	376	tl	17	376	tl	17	381	tl	14	377	tl	tl
DSJC1000.1	7	28	tl	7	28	tl	7	28	tl	6	28	tl	7	28	tl	7	27	tl	6	27	tl	tl
DSJC1000.5	18	116	tl	18	116	tl	18	116	tl	14	116	tl	16	116	tl	15	116	tl	16	115	tl	tl
DSJC1000.9	71	294	tl	71	294	tl	71	294	tl	70	294	tl	70	294	tl	61	302	tl	58	299	tl	tl
DSJC125.1	5	5	0.1	5	5	0.1	5	5	0.1	5	5	156.8	5	5	900.2	5	5	1.5	5	5	0.4	7.0
DSJC125.5	14	21	tl	14	21	tl	14	21	tl	13	21	tl	13	21	tl	13	21	tl	13	22	tl	tl
DSJC125.9	39	50	tl	39	50	tl	39	50	tl	40	50	tl	39	50	tl	37	51	tl	37	51	tl	tl
DSJC250.1	6	10	tl	7	10	tl	6	10	tl	6	10	tl	6	10	tl	6	10	tl	5	10	tl	tl

Table A.1: (continued)

	Assignment			Assignment (AMO)			Partial Order			Full Encoding			CEGAR Sparse			gc-cdcl			POP-S			CCC
Instance	*lb*	*ub*	time	*lb*	*ub*	time	*lb*	*ub*	time	*lb*	*ub*	time	*lb*	*ub*	time	*lb*	*ub*	time	*lb*	*ub*	time	time
DSJC250.5	15	39	tl	15	39	tl	15	39	tl	14	39	tl	14	39	tl	13	36	tl	14	37	tl	tl
DSJC250.9	48	89	tl	48	89	tl	48	89	tl	48	89	tl	47	89	tl	43	91	tl	44	92	tl	tl
DSJC500.1	7	16	tl	7	16	tl	7	16	tl	6	16	tl	6	16	tl	6	16	tl	6	16	tl	tl
DSJC500.5	17	66	tl	17	66	tl	17	66	tl	16	66	tl	16	66	tl	14	66	tl	15	65	tl	tl
DSJC500.9	61	163	tl	61	163	tl	60	163	tl	59	163	tl	59	163	tl	52	165	tl	53	170	tl	tl
DSJR500.1	12	12	0.1	12	12	0.1	12	12	0.1	12	12	0.1	12	12	0.1	12	12	0.2	12	12	2.7	0.1
DSJR500.1c	85	85	4.7	85	85	4.9	85	85	6.0	85	85	3.6	85	85	3.7	79	92	tl	80	89	tl	50.9
DSJR500.5	122	125	tl	122	122	1729	122	122	135.8	122	125	tl	122	125	tl	122	124	tl	122	131	tl	68.4
abb313GPIA	9	9	1.0	9	9	1.0	9	9	1.6	8	10	tl	9	10	tl	9	11	tl	9	9	46.4	tl
anna	11	11	0.0	11	11	0.0	11	11	0.0	11	11	0.0	11	11	0.0	11	11	0.0	11	11	0.3	0.0
ash331GPIA	4	4	0.2	4	4	0.2	4	4	0.2	3	5	tl	4	5	tl	4	4	4.7	4	4	3.3	0.1
ash608GPIA	4	4	0.3	4	4	0.3	4	4	0.3	3	5	tl	4	5	tl	4	4	46.1	4	4	9.8	0.4
ash958GPIA	4	4	0.4	4	4	0.4	4	4	0.4	3	5	tl	4	5	tl	4	4	205.8	4	4	23.7	1.0
david	11	11	0.0	11	11	0.0	11	11	0.0	11	11	0.0	11	11	0.0	11	11	0.0	11	11	0.3	0.0
flat1000_50_0	17	117	tl	17	117	tl	17	117	tl	14	117	tl	16	117	tl	15	117	tl	16	114	tl	tl
flat1000_60_0	18	114	tl	18	114	tl	17	114	tl	15	114	tl	16	114	tl	15	115	tl	16	114	tl	tl
flat1000_76_0	18	114	tl	18	114	tl	18	114	tl	15	114	tl	16	114	tl	15	115	tl	16	115	tl	tl
flat300_20_0	14	43	tl	14	43	tl	14	43	tl	13	43	tl	13	43	tl	13	43	tl	14	42	tl	tl
flat300_26_0	14	42	tl	14	42	tl	14	42	tl	13	42	tl	13	42	tl	13	41	tl	14	41	tl	tl
flat300_28_0	15	44	tl	15	44	tl	15	44	tl	14	44	tl	14	44	tl	13	42	tl	14	42	tl	tl
fpsol2.i.1	65	65	0.1	65	65	0.1	65	65	0.1	65	65	0.1	65	65	0.1	65	65	6.5	65	65	34.7	0.1
fpsol2.i.2	30	30	0.1	30	30	0.1	30	30	0.1	30	30	0.1	30	30	0.2	30	30	2.5	30	30	26.7	0.1
fpsol2.i.3	30	30	0.1	30	30	0.1	30	30	0.1	30	30	0.2	30	30	0.2	30	30	2.5	30	30	29.5	0.1
games120	9	9	0.0	9	9	0.0	9	9	0.0	9	9	0.0	9	9	0.0	9	9	0.0	9	9	0.4	0.0
homer	13	13	0.1	13	13	0.1	13	13	0.1	13	13	0.1	13	13	0.1	13	13	0.0	13	13	1.4	0.0
huck	11	11	0.0	11	11	0.0	11	11	0.0	11	11	0.0	11	11	0.0	11	11	0.0	11	11	0.3	0.0
inithx.i.1	54	54	0.2	54	54	0.2	54	54	0.2	54	54	0.2	54	54	0.2	54	54	9.2	54	54	45.4	0.2
inithx.i.2	31	31	0.2	31	31	0.2	31	31	0.2	31	31	0.2	31	31	0.2	31	31	5.3	31	31	42.5	0.1
inithx.i.3	31	31	0.2	31	31	0.2	31	31	0.2	31	31	0.2	31	31	0.2	31	31	5.0	31	31	43.9	0.1
jean	10	10	0.0	10	10	0.0	10	10	0.0	10	10	0.0	10	10	0.0	10	10	0.0	10	10	0.3	0.0
latin_square_10	90	137	tl	90	137	tl	90	137	tl	90	137	tl	90	137	tl	90	125	tl	90	132	tl	tl
le450_15a	15	15	6.1	15	15	0.4	15	15	0.6	15	17	tl	15	17	tl	15	15	23.3	15	15	7.6	0.2
le450_15b	15	15	0.7	15	15	0.3	15	15	0.4	15	16	tl	15	16	tl	15	15	46.8	15	15	7.4	0.2
le450_15c	15	26	tl	15	26	tl	15	26	tl	15	26	tl	15	26	tl	15	24	tl	15	23	tl	82.6
le450_15d	15	26	tl	15	26	tl	15	26	tl	15	26	tl	15	26	tl	15	23	tl	15	24	tl	65.2

Table A.1: (continued)

	Assignment			Assignment (AMO)			Partial Order			Full Encoding			CEGAR Sparse			gc-cdcl			POP-S			CCC
Instance	*lb*	*ub*	time	*lb*	*ub*	time	*lb*	*ub*	time	*lb*	*ub*	time	*lb*	*ub*	time	*lb*	*ub*	time	*lb*	*ub*	time	time
le450_25a	25	25	0.2	25	25	0.2	25	25	0.2	25	25	0.2	25	25	0.2	25	25	1.3	25	25	8.1	0.1
le450_25b	25	25	0.2	25	25	0.2	25	25	0.2	25	25	0.2	25	25	0.2	25	25	0.6	25	25	8.1	0.1
le450_25c	25	30	tl	25	30	tl	25	30	tl	25	30	tl	25	30	tl	25	29	tl	25	29	tl	tl
le450_25d	25	29	tl	25	29	tl	25	29	tl	25	29	tl	25	29	tl	25	28	tl	25	29	tl	tl
le450_5a	5	5	0.1	5	5	0.1	5	5	0.1	5	10	tl	5	10	tl	5	5	49.3	5	5	3.3	0.1
le450_5b	5	5	0.1	5	5	0.1	5	5	0.1	5	5	2825	5	5	1773	5	5	55.8	5	5	3.3	0.1
le450_5c	5	5	0.1	5	5	0.1	5	5	0.1	5	5	2237	5	5	681.8	5	5	9.8	5	5	5.4	0.1
le450_5d	5	5	0.1	5	5	0.1	5	5	0.1	5	5	2326	5	5	992.4	5	5	8.9	5	5	5.2	0.1
miles1000	42	42	0.1	42	42	0.1	42	42	0.1	42	42	0.1	42	42	0.0	42	42	1.3	42	42	12.9	0.0
miles1500	73	73	0.1	73	73	0.1	73	73	0.1	73	73	0.1	73	73	0.1	73	73	5.2	73	73	49.8	0.1
miles250	8	8	0.0	8	8	0.0	8	8	0.0	8	8	0.0	8	8	0.0	8	8	0.0	8	8	0.3	0.0
miles500	20	20	0.0	20	20	0.0	20	20	0.0	20	20	0.0	20	20	0.0	20	20	0.1	20	20	0.4	0.0
miles750	31	31	0.0	31	31	0.0	31	31	0.0	31	31	0.0	31	31	0.0	31	31	0.5	31	31	4.7	0.0
mug100_1	4	4	0.0	4	4	0.1	4	4	0.0	4	4	69.8	4	4	551.2	4	4	0.2	4	4	0.3	0.0
mug100_25	4	4	0.0	4	4	0.0	4	4	0.0	4	4	105.0	4	4	2529	4	4	0.2	4	4	0.3	0.0
mug88_1	4	4	0.0	4	4	0.0	4	4	0.0	4	4	60.9	4	4	294.1	4	4	0.2	4	4	0.3	0.0
mug88_25	4	4	0.0	4	4	0.0	4	4	0.0	4	4	18.0	4	4	164.9	4	4	0.1	4	4	0.3	0.0
mulsol.i.1	49	49	0.1	49	49	0.1	49	49	0.1	49	49	0.1	49	49	0.1	49	49	2.0	49	49	11.2	0.1
mulsol.i.2	31	31	0.1	31	31	0.1	31	31	0.1	31	31	0.1	31	31	0.1	31	31	1.0	31	31	12.6	0.1
mulsol.i.3	31	31	0.1	31	31	0.1	31	31	0.1	31	31	0.1	31	31	0.1	31	31	1.0	31	31	12.8	0.1
mulsol.i.4	31	31	0.1	31	31	0.1	31	31	0.1	31	31	0.1	31	31	0.1	31	31	1.0	31	31	13.3	0.1
mulsol.i.5	31	31	0.1	31	31	0.1	31	31	0.1	31	31	0.1	31	31	0.1	31	31	1.0	31	31	13.1	0.0
myciel3	4	4	0.0	4	4	0.0	4	4	0.0	4	4	0.0	4	4	0.0	4	4	0.0	4	4	0.3	0.0
myciel4	5	5	0.1	5	5	0.1	5	5	0.1	5	5	0.1	5	5	0.1	5	5	0.0	5	5	0.4	0.0
myciel5	6	6	0.1	6	6	0.1	6	6	0.1	6	6	0.1	6	6	0.1	6	6	0.0	6	6	0.8	0.6
myciel6	7	7	0.1	7	7	0.1	7	7	0.1	7	7	0.1	7	7	0.1	7	7	0.2	7	7	2630	1280
myciel7	8	8	0.2	8	8	0.2	8	8	0.2	8	8	0.2	8	8	0.2	8	8	1.1	6	8	tl	tl
qg.order100	100	102	tl	100	102	tl	100	102	tl	100	102	tl	100	102	tl	100	102	tl	100	116	tl	tl
qg.order30	30	30	3.4	30	30	0.4	30	30	1.4	30	32	tl	30	32	tl	30	30	0.4	30	30	38.3	4.7
qg.order40	40	40	12.5	40	40	1.2	40	40	49.8	40	41	tl	40	41	tl	40	40	94.7	40	40	131.6	90.2
qg.order60	60	60	80.9	60	60	15.7	60	60	351.2	60	62	tl	60	62	tl	60	60	3488	60	60	3183	267.2
queen10_10	10	13	tl	10	13	tl	11	11	719.5	10	13	tl	10	13	tl	10	14	tl	11	11	534.1	tl
queen11_11	11	15	tl	11	15	tl	11	11	1648	11	15	tl	11	15	tl	11	14	tl	11	11	507.2	tl
queen12_12	12	15	tl	12	15	tl	12	15	tl	12	15	tl	12	15	tl	12	16	tl	12	16	tl	tl
queen13_13	13	17	tl	13	17	tl	13	17	tl	13	17	tl	13	17	tl	13	17	tl	13	17	tl	tl

Table A.1: (continued)

Instance	Assignment			Assignment (AMO)			Partial Order			Full Encoding			CEGAR Sparse			gc-cdcl			POP-S			CCC
	lb	*ub*	time	*lb*	*ub*	time	*lb*	*ub*	time	*lb*	*ub*	time	*lb*	*ub*	time	*lb*	*ub*	time	*lb*	*ub*	time	time
queen14_14	14	18	tl	14	18	tl	14	18	tl	14	18	tl	14	18	tl	14	18	tl	14	19	tl	tl
queen15_15	15	19	tl	15	19	tl	15	19	tl	15	19	tl	15	19	tl	15	19	tl	15	21	tl	tl
queen16_16	16	21	tl	16	21	tl	16	21	tl	16	21	tl	16	21	tl	16	21	tl	16	23	tl	tl
queen5_5	5	5	0.0	5	5	0.0	5	5	0.0	5	5	0.0	5	5	0.0	5	5	0.0	5	5	0.6	0.0
queen6_6	7	7	0.0	7	7	0.0	7	7	0.0	7	7	0.4	7	7	0.7	7	7	0.5	7	7	0.6	0.0
queen7_7	7	7	0.0	7	7	0.0	7	7	0.1	7	7	0.1	7	7	0.4	7	7	0.1	7	7	0.8	0.0
queen8_12	12	12	0.2	12	12	0.0	12	12	0.1	12	12	1.0	12	12	18.2	12	12	0.3	12	12	1.8	0.0
queen8_8	9	9	8.9	9	9	7.9	9	9	3.6	9	9	167.8	9	9	592.6	9	9	10.8	9	9	4.7	17.3
queen9_9	10	10	558.1	10	10	721.6	10	10	21.7	9	12	tl	9	12	tl	10	10	317.4	10	10	16.2	tl
r1000.1	20	20	0.8	20	20	0.8	20	20	0.8	20	20	0.8	20	20	0.8	20	20	1.0	20	20	16.4	0.2
r1000.1c	96	104	tl	96	104	tl	96	104	tl	98	98	2102	97	104	tl	84	110	tl	82	105	tl	tl
r1000.5	234	234	1040	234	234	1312	234	234	1020	234	239	tl	234	239	tl	233	249	tl	234	246	tl	tl
r125.1	5	5	0.0	5	5	0.0	5	5	0.0	5	5	0.0	5	5	0.1	5	5	0.0	5	5	0.3	0.0
r125.1c	46	46	0.1	46	46	0.1	46	46	0.1	46	46	0.1	46	46	0.1	46	46	9.0	46	46	82.7	0.1
r125.5	36	36	0.2	36	36	0.2	36	36	0.2	36	36	1.6	36	36	1.4	36	36	6.6	36	36	19.3	0.1
r250.1	8	8	0.0	8	8	0.1	8	8	0.0	8	8	0.0	8	8	0.1	8	8	0.0	8	8	0.7	0.0
r250.1c	64	64	0.2	64	64	0.2	64	64	0.2	64	64	0.3	64	64	0.3	64	64	103.0	64	64	104.9	0.3
r250.5	65	65	0.5	65	65	0.4	65	65	0.5	65	65	12.6	65	65	33.5	65	65	74.9	65	65	105.7	3.5
school1	14	14	1.1	14	14	1.1	14	14	1.1	14	14	1.1	14	14	1.1	14	14	7.5	14	14	33.4	0.6
school1_nsh	14	14	0.3	14	14	0.3	14	14	0.3	14	14	3.5	14	14	12.1	14	14	11.1	14	14	21.0	0.1
wap01a	41	48	tl	41	48	tl	41	48	tl	41	48	tl	41	48	tl	41	47	tl	41	41	3064	306.0
wap02a	40	45	tl	40	45	tl	40	45	tl	40	45	tl	40	45	tl	40	45	tl	40	40	1708	301.7
wap03a	40	52	tl	40	52	tl	40	52	tl	40	52	tl	40	52	tl	40	54	tl	40	55	tl	tl
wap04a	40	49	tl	40	49	tl	40	49	tl	40	49	tl	40	49	tl	40	47	tl	40	49	tl	tl
wap05a	50	50	1.0	50	50	1.0	50	50	1.0	50	50	1.0	50	50	1.0	50	50	13.0	50	50	98.5	0.7
wap06a	40	40	804.5	40	40	757.8	40	40	118.3	40	43	tl	40	43	tl	40	44	tl	40	40	172.6	72.3
wap07a	40	44	tl	40	44	tl	40	44	tl	40	44	tl	40	44	tl	40	45	tl	40	45	tl	tl
wap08a	40	44	tl	40	40	1649	40	40	3213	40	44	tl	40	44	tl	40	44	tl	40	40	799.5	tl
will199GPIA	7	7	0.2	7	7	0.2	7	7	0.2	6	7	tl	7	7	2.6	7	7	2.1	7	7	5.7	0.3
zeroin.i.1	49	49	0.1	49	49	0.1	49	49	0.1	49	49	0.1	49	49	0.1	49	49	2.2	49	49	12.1	0.0
zeroin.i.2	30	30	0.1	30	30	0.1	30	30	0.1	30	30	0.1	30	30	0.1	30	30	0.9	30	30	9.7	0.0
zeroin.i.3	30	30	0.1	30	30	0.0	30	30	0.1	30	30	0.1	30	30	0.0	30	30	0.9	30	30	9.9	0.0
#Solved		89			91			93			74			72			82			91		90

Table A.2: Detailed performance results of different Zykov propagator configurations

	ZP None			ZP Default			ZP Improved			AP Default		
Instance	*lb*	*ub*	time	*lb*	*ub*	time	*lb*	*ub*	time	*lb*	*ub*	time
1-FullIns_3	4	4	0.0	4	4	0.0	4	4	0.0	4	4	0.0
1-FullIns_4	5	5	0.1	5	5	0.1	5	5	0.1	5	5	0.1
1-FullIns_5	6	6	0.2	6	6	0.1	6	6	0.1	6	6	0.1
1-Insertions_4	4	5	tl	5	5	16.0	5	5	118.8	5	5	2.1
1-Insertions_5	4	6	tl	5	6	tl	5	6	tl	5	6	tl
1-Insertions_6	3	7	tl	5	7	tl	4	7	tl	5	7	tl
2-FullIns_3	5	5	0.1	5	5	0.1	5	5	0.1	5	5	0.1
2-FullIns_4	6	6	0.1	6	6	0.1	6	6	0.1	6	6	0.1
2-FullIns_5	7	7	7.9	7	7	0.8	7	7	1.0	7	7	0.2
2-Insertions_3	4	4	13.0	4	4	0.1	4	4	0.1	4	4	0.0
2-Insertions_4	4	5	tl	4	5	tl	4	5	tl	4	5	tl
2-Insertions_5	3	6	tl	4	6	tl	4	6	tl	4	6	tl
3-FullIns_3	6	6	0.1	6	6	0.1	6	6	0.1	6	6	0.1
3-FullIns_4	7	7	0.2	7	7	0.2	7	7	0.2	7	7	0.2
3-FullIns_5	7	8	tl	8	8	79.5	8	8	5.9	8	8	10.8
3-Insertions_3	4	4	241.0	4	4	0.5	4	4	1.0	4	4	0.1
3-Insertions_4	4	5	tl	4	5	tl	4	5	tl	4	5	tl
3-Insertions_5	3	6	tl	4	6	tl	3	6	tl	4	6	tl
4-FullIns_3	7	7	0.1	7	7	0.1	7	7	0.1	7	7	0.1
4-FullIns_4	8	8	0.2	8	8	0.2	8	8	0.2	8	8	0.2
4-FullIns_5	8	9	tl	9	9	1213	9	9	76.4	9	9	201.0
4-Insertions_3	4	4	1941	4	4	3.0	4	4	8.6	4	4	0.2
4-Insertions_4	3	5	tl	4	5	tl	4	5	tl	4	5	tl
5-FullIns_3	8	8	0.1	8	8	0.1	8	8	0.1	8	8	0.1
5-FullIns_4	9	9	0.3	9	9	0.3	9	9	0.2	9	9	0.3
C2000.5	16	206	tl	17	206	tl	17	209	tl	18	206	tl
C2000.9	78	530	tl	77	530	tl	77	530	tl	78	530	tl
C4000.5	17	376	tl	17	376	tl	18	376	tl	19	376	tl
DSJC1000.1	7	28	tl	7	28	tl	7	28	tl	7	28	tl
DSJC1000.5	16	116	tl	17	116	tl	17	116	tl	18	116	tl
DSJC1000.9	70	294	tl	70	294	tl	70	294	tl	71	294	tl
DSJC125.1	5	5	321.8	5	5	1.8	5	5	1.1	5	5	0.1
DSJC125.5	13	21	tl	14	21	tl	14	21	tl	14	21	tl
DSJC125.9	39	50	tl	40	50	tl	40	50	tl	39	50	tl
DSJC250.1	6	10	tl	6	10	tl	6	10	tl	6	10	tl
DSJC250.5	14	39	tl	15	39	tl	15	39	tl	15	39	tl
DSJC250.9	48	89	tl	49	89	tl	50	89	tl	48	89	tl
DSJC500.1	6	16	tl	7	16	tl	7	16	tl	7	16	tl
DSJC500.5	16	66	tl	16	66	tl	16	66	tl	17	66	tl
DSJC500.9	59	163	tl	60	163	tl	61	163	tl	60	163	tl
DSJR500.1	12	12	0.1	12	12	0.1	12	12	0.1	12	12	0.1
DSJR500.1c	85	85	3.8	85	85	8.5	85	85	6.6	85	85	3.6
DSJR500.5	122	125	tl	122	125	tl	122	122	1639	122	125	tl
abb313GPIA	9	10	tl	9	10	tl	9	9	706.6	9	9	0.8
anna	11	11	0.0	11	11	0.0	11	11	0.0	11	11	0.0
ash331GPIA	4	5	tl	4	4	263.7	4	4	103.6	4	4	0.2
ash608GPIA	4	5	tl	4	5	tl	4	4	1625	4	4	0.3

Table A.2: (continued)

	ZP None			ZP Default			ZP Improved			AP Default		
Instance	*lb*	*ub*	time	*lb*	*ub*	time	*lb*	*ub*	time	*lb*	*ub*	time
ash958GPIA	4	5	tl	4	5	tl	4	5	tl	4	4	0.5
david	11	11	0.0	11	11	0.0	11	11	0.0	11	11	0.0
flat1000_50_0	16	117	tl	17	117	tl	17	117	tl	17	117	tl
flat1000_60_0	16	114	tl	17	114	tl	17	114	tl	17	114	tl
flat1000_76_0	16	114	tl	17	114	tl	17	114	tl	18	114	tl
flat300_20_0	13	43	tl	14	43	tl	14	43	tl	14	43	tl
flat300_26_0	13	42	tl	14	42	tl	14	42	tl	14	42	tl
flat300_28_0	14	44	tl	15	44	tl	15	44	tl	15	44	tl
fpsol2.i.1	65	65	0.1	65	65	0.1	65	65	0.1	65	65	0.1
fpsol2.i.2	30	30	0.1	30	30	0.2	30	30	0.1	30	30	0.2
fpsol2.i.3	30	30	0.2	30	30	0.2	30	30	0.2	30	30	0.1
games120	9	9	0.0	9	9	0.0	9	9	0.0	9	9	0.0
homer	13	13	0.1	13	13	0.1	13	13	0.1	13	13	0.1
huck	11	11	0.0	11	11	0.0	11	11	0.0	11	11	0.0
inithx.i.1	54	54	0.2	54	54	0.2	54	54	0.2	54	54	0.2
inithx.i.2	31	31	0.2	31	31	0.2	31	31	0.2	31	31	0.2
inithx.i.3	31	31	0.2	31	31	0.2	31	31	0.2	31	31	0.2
jean	10	10	0.0	10	10	0.0	10	10	0.0	10	10	0.0
latin_square_10	90	137	tl	90	137	tl	90	137	tl	90	137	tl
le450_15a	15	17	tl	15	15	79.6	15	15	56.2	15	15	1.3
le450_15b	15	15	3409	15	15	49.7	15	15	36.7	15	15	0.3
le450_15c	15	26	tl	15	15	1535	15	15	498.4	15	26	tl
le450_15d	15	26	tl	15	15	1448	15	15	522.8	15	26	tl
le450_25a	25	25	0.2	25	25	0.2	25	25	0.2	25	25	0.2
le450_25b	25	25	0.2	25	25	0.2	25	25	0.2	25	25	0.2
le450_25c	25	30	tl	25	30	tl	25	30	tl	25	30	tl
le450_25d	25	29	tl	25	29	tl	25	29	tl	25	29	tl
le450_5a	5	5	681.8	5	5	30.1	5	5	16.7	5	5	0.1
le450_5b	5	5	1590	5	5	28.9	5	5	16.5	5	5	0.1
le450_5c	5	5	1.5	5	5	34.4	5	5	14.0	5	5	0.1
le450_5d	5	5	1543	5	5	23.7	5	5	12.7	5	5	0.1
miles1000	42	42	0.1	42	42	0.1	42	42	0.1	42	42	0.1
miles1500	73	73	0.1	73	73	0.1	73	73	0.1	73	73	0.1
miles250	8	8	0.0	8	8	0.0	8	8	0.0	8	8	0.0
miles500	20	20	0.0	20	20	0.0	20	20	0.0	20	20	0.0
miles750	31	31	0.0	31	31	0.0	31	31	0.0	31	31	0.0
mug100_1	4	4	65.2	4	4	0.4	4	4	0.1	4	4	0.1
mug100_25	4	4	75.2	4	4	0.6	4	4	0.0	4	4	0.1
mug88_1	4	4	50.3	4	4	0.2	4	4	0.1	4	4	0.1
mug88_25	4	4	32.4	4	4	0.3	4	4	0.1	4	4	0.0
mulsol.i.1	49	49	0.1	49	49	0.1	49	49	0.1	49	49	0.1
mulsol.i.2	31	31	0.1	31	31	0.1	31	31	0.1	31	31	0.1
mulsol.i.3	31	31	0.1	31	31	0.1	31	31	0.1	31	31	0.1
mulsol.i.4	31	31	0.1	31	31	0.1	31	31	0.1	31	31	0.1
mulsol.i.5	31	31	0.1	31	31	0.1	31	31	0.1	31	31	0.1
myciel3	4	4	0.0	4	4	0.0	4	4	0.0	4	4	0.0
myciel4	5	5	0.1	5	5	0.1	5	5	0.1	5	5	0.1

Table A.2: (continued)

	ZP None			ZP Default			ZP Improved			AP Default		
Instance	*lb*	*ub*	time	*lb*	*ub*	time	*lb*	*ub*	time	*lb*	*ub*	time
myciel5	6	6	0.1	6	6	0.1	6	6	0.1	6	6	0.1
myciel6	7	7	0.1	7	7	0.2	7	7	0.1	7	7	0.2
myciel7	8	8	0.2	8	8	0.2	8	8	0.2	8	8	0.2
qg.order100	100	102	tl	100	102	tl	100	102	tl	100	102	tl
qg.order30	30	30	499.4	30	30	204.8	30	30	60.4	30	30	0.2
qg.order40	40	41	tl	40	41	tl	40	40	490.1	40	40	0.6
qg.order60	60	62	tl	60	60	1336	60	62	tl	60	60	21.1
queen10_10	10	13	tl	11	11	630.1	11	11	1987	10	13	tl
queen11_11	11	15	tl	11	11	276.0	11	11	1114	11	15	tl
queen12_12	12	15	tl	12	15	tl	12	15	tl	12	15	tl
queen13_13	13	17	tl	13	17	tl	13	17	tl	13	17	tl
queen14_14	14	18	tl	14	18	tl	14	18	tl	14	18	tl
queen15_15	15	19	tl	15	19	tl	15	19	tl	15	19	tl
queen16_16	16	21	tl	16	21	tl	16	21	tl	16	21	tl
queen5_5	5	5	0.0	5	5	0.0	5	5	0.0	5	5	0.0
queen6_6	7	7	0.2	7	7	0.1	7	7	0.1	7	7	0.0
queen7_7	7	7	0.2	7	7	0.1	7	7	0.1	7	7	0.0
queen8_12	12	12	0.4	12	12	0.4	12	12	0.2	12	12	0.0
queen8_8	9	9	232.3	9	9	2.3	9	9	2.1	9	9	8.7
queen9_9	9	12	tl	10	10	15.6	10	10	71.2	10	10	822.8
r1000.1	20	20	0.8	20	20	0.8	20	20	0.8	20	20	0.8
r1000.1c	98	98	1885	96	104	tl	95	104	tl	96	104	tl
r1000.5	234	239	tl	234	239	tl	234	239	tl	234	239	tl
r125.1	5	5	0.0	5	5	0.0	5	5	0.0	5	5	0.1
r125.1c	46	46	0.1	46	46	0.1	46	46	0.1	46	46	0.1
r125.5	36	36	0.4	36	36	0.7	36	36	0.4	36	36	0.2
r250.1	8	8	0.0	8	8	0.1	8	8	0.0	8	8	0.1
r250.1c	64	64	0.3	64	64	0.2	64	64	0.3	64	64	0.3
r250.5	65	65	5.6	65	65	8.1	65	65	7.2	65	65	0.9
school1	14	14	1.1	14	14	1.2	14	14	1.2	14	14	1.1
school1_nsh	14	14	1.0	14	14	6.6	14	14	0.4	14	14	0.3
wap01a	41	48	tl	41	48	tl	41	48	tl	41	48	tl
wap02a	40	45	tl	40	45	tl	40	45	tl	40	45	tl
wap03a	40	52	tl	40	52	tl	40	52	tl	40	52	tl
wap04a	40	49	tl	40	49	tl	40	49	tl	40	49	tl
wap05a	50	50	1.0	50	50	1.0	50	50	1.0	50	50	1.0
wap06a	40	43	tl	40	43	tl	40	43	tl	40	40	1173
wap07a	40	44	tl	40	44	tl	40	44	tl	40	44	tl
wap08a	40	44	tl	40	44	tl	40	44	tl	40	44	tl
will199GPIA	7	7	0.6	7	7	104.4	7	7	0.5	7	7	0.2
zeroin.i.1	49	49	0.1	49	49	0.1	49	49	0.1	49	49	0.1
zeroin.i.2	30	30	0.1	30	30	0.1	30	30	0.1	30	30	0.1
zeroin.i.3	30	30	0.1	30	30	0.1	30	30	0.1	30	30	0.1
#Solved		77			87			90			88	

Bibliography

[Aar+07] Karen I Aardal, Stan PM Van Hoesel, Arie MCA Koster, Carlo Mannino, and Antonio Sassano. "Models and solution techniques for frequency assignment problems". In: *Annals of Operations Research* 153 (2007), pp. 79–129 (cit. on p. iii).

[Abí+13] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Peter J Stuckey. "To encode or to propagate? The best choice for each constraint in SAT". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2013, pp. 97–106 (cit. on pp. 61, 65).

[AI16] Takuya Akiba and Yoichi Iwata. "Branch-and-reduce exponential/FPT algorithms in practice: A case study of vertex cover". In: *Theoretical Computer Science* 609 (2016), pp. 211–225 (cit. on p. 23).

[ALS13] Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. "Improving glucose for incremental SAT solving with assumptions: Application to MUS extraction". In: *International conference on theory and applications of satisfiability testing*. Springer. 2013, pp. 309–317 (cit. on p. 5).

[AS18] Gilles Audemard and Laurent Simon. "On the glucose SAT solver". In: *International Journal on Artificial Intelligence Tools* 27.01 (2018), p. 1840001 (cit. on p. 7).

[Así+11] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. "Cardinality networks: a theoretical and empirical study". In: *Constraints* 16 (2011), pp. 195–221 (cit. on p. 37).

[BB04] Nicolas Barnier and Pascal Brisset. "Graph coloring for air traffic flow management". In: *Annals of operations research* 130 (2004), pp. 163–178 (cit. on p. iii).

[BF15] Armin Biere and Andreas Fröhlich. "Evaluating CDCL variable scoring schemes". In: *Theory and Applications of Satisfiability Testing–SAT 2015: 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings 18*. Springer. 2015, pp. 405–422 (cit. on p. 58).

[BF22] Armin Biere and Mathias Fleury. "Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022". In: *Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions*. Ed. by Tomas Balyo, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Vol. B-2022-1. Department of Computer Science Series of Publications B. University of Helsinki, 2022, pp. 10–11 (cit. on p. 51).

[Bie+24] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. "CaDiCaL 2.0". In: *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC Canada, July 24-27, 2024, Proceedings, Part I*. Ed. by Arie Gurfinkel and Vijay Ganesh. Vol. 14681. LNCS. Springer, 2024, pp. 133–152. DOI: 10.1007/978-3-031-65627-9_7 (cit. on pp. 7, 56).

[Bré79] Daniel Brélaz. "New methods to color the vertices of a graph". In: *Communications of the ACM* 22.4 (1979), pp. 251–256 (cit. on pp. 6, 19, 58).

[CFM17] Denis Cornaz, Fabio Furini, and Enrico Malaguti. "Solving vertex coloring problems as maximum weight stable set problems". In: *Discrete Applied Mathematics* 217 (2017), pp. 151–162 (cit. on pp. 19, 20, 22, 23).

[Cha82] Gregory J Chaitin. "Register allocation & spilling via graph coloring". In: *ACM Sigplan Notices* 17.6 (1982), pp. 98–101 (cit. on p. iii).

[CJ08] Denis Cornaz and Vincent Jost. "A one-to-one correspondence between colorings and stable sets". In: *Operations Research Letters* 36.6 (2008), pp. 673–676 (cit. on p. 18).

[Cla+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-guided abstraction refinement". In: *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings 12*. Springer. 2000, pp. 154–169 (cit. on p. 38).

[DLL62] Martin Davis, George Logemann, and Donald Loveland. "A machine program for theorem-proving". In: *Communications of the ACM* 5.7 (1962), pp. 394–397 (cit. on p. 4).

[ES03] Niklas Eén and Niklas Sörensson. "An extensible SAT-solver". In: *International conference on theory and applications of satisfiability testing*. Springer. 2003, pp. 502–518 (cit. on p. 58).

[Faz+23] Katalin Fazekas, Aina Niemetz, Mathias Preiner, Markus Kirchweger, Stenfa Szeider, and Armin Biere. "IPASIR-UP: User Propagators for CDCL". In: *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, Alghero, Italy*. Ed. by Meena Mahajan and Friedrich Slivovsky. Vol. 271. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 8:1–8:13. DOI: 10.4230/LIPICS.SAT.2023.8 (cit. on pp. iv, 8, 56, 75).

[FG10] Alan M Frisch and Paul A Giannaros. "SAT encodings of the at-most-k constraint: Some old, some new, some fast, some slow". In: (2010) (cit. on p. 37).

[FGT17] Fabio Furini, Virginie Gabrel, and Ian-Christopher Ternier. "An Improved DSATUR-Based Branch-and-Bound Algorithm for the Vertex Coloring Problem". In: *Networks* 69.1 (2017), pp. 124–141 (cit. on pp. 6, 19, 22, 59, 61, 62, 65, 76, 77).

[FJM24] Daniel Faber, Adalat Jabrayilov, and Petra Mutzel. "SAT Encoding of Partial Ordering Models for Graph Coloring Problems". In: *arXiv preprint arXiv:2403.15961* (2024) (cit. on pp. 33, 34, 50, 51, 72, 73).

[Gen+14] Ian P Gent, Christopher Jefferson, Steve Linton, Ian Miguel, and Peter Nightingale. "Generating custom propagators for arbitrary constraints". In: *Artificial Intelligence* 211 (2014), pp. 1–33 (cit. on p. 56).

[GH06] Philippe Galinier and Alain Hertz. "A survey of local search methods for graph coloring". In: *Computers & Operations Research* 33.9 (2006), pp. 2547–2562 (cit. on p. 19).

[GH99] Philippe Galinier and Jin-Kao Hao. "Hybrid evolutionary algorithms for graph coloring". In: *Journal of combinatorial optimization* 3 (1999), pp. 379–397 (cit. on p. 19).

[Glo+19] Gael Glorian, Jean-Marie Lagniez, Valentin Montmirail, and Nicolas Szczepanski. "An incremental sat-based approach to the graph colouring problem". In: *Principles and Practice of Constraint Programming: 25th International Conference, CP 2019, Stamford, CT, USA, September 30–October 4, 2019, Proceedings 25*. Springer. 2019, pp. 213–231 (cit. on pp. iv, 31, 38–49, 52, 75).

[GLS81] Martin Grötschel, László Lovász, and Alexander Schrijver. "The ellipsoid method and its consequences in combinatorial optimization". In: *Combinatorica* 1 (1981), pp. 169–197 (cit. on p. 13).

[GMP05] Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothen. "What color is your Jacobian? Graph coloring for computing derivatives". In: *SIAM review* 47.4 (2005), pp. 629–705 (cit. on p. iii).

[HCS12] Stephan Held, William Cook, and Edward C Sewell. "Maximum-weight stable sets and safe lower bounds for graph coloring". In: *Mathematical Programming Computation* 4.4 (2012), pp. 363–381 (cit. on pp. 7, 14, 35, 62).

[HK20] Emmanuel Hébrard and George Katsirelos. "Constraint and satisfiability reasoning for graph coloring". In: *Journal of Artificial Intelligence Research* 69 (2020), pp. 33–65 (cit. on pp. iv, 7, 14, 16, 17, 21, 50, 55, 59–61, 63–66, 69, 70, 72, 75, 76).

[HKH22] Marijn JH Heule, Anthony Karahalios, and Willem-Jan van Hoeve. "From cliques to colorings and back again". In: *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2022 (cit. on pp. 7, 8, 32, 50, 52, 72, 73).

[HLS09] Pierre Hansen, Martine Labbé, and David Schindl. "Set covering and packing formulations of graph coloring: Algorithms and first polyhedral results". In: *Discrete Optimization* 6.2 (2009), pp. 135–147 (cit. on p. 20).

[Hoe22] Willem-Jan van Hoeve. "Graph coloring with decision diagrams". In: *Mathematical Programming* 192.1 (2022), pp. 631–674 (cit. on pp. 7, 8, 14, 20).

[Hof03] Alan J Hoffman. "On eigenvalues and colorings of graphs". In: *Selected Papers Of Alan J Hoffman: With Commentary*. World Scientific, 2003, pp. 407–419 (cit. on p. 13).

[Hul21] R.P. van der Hulst. "A branch-price-and-cut algorithm for graph coloring". MA thesis. University of Twente, August 2021. URL: http://essay.utwente.nl/88263/ (cit. on pp. 7, 8, 20, 59).

[HW87] Alain Hertz and D de Werra. "Using tabu search techniques for graph coloring". In: *Computing* 39.4 (1987), pp. 345–351 (cit. on p. 19).

[JM18] Adalat Jabrayilov and Petra Mutzel. "New integer linear programming models for the vertex coloring problem". In: *LATIN 2018: Theoretical Informatics: 13th Latin American Symposium, Buenos Aires, Argentina, April 16-19, 2018, Proceedings 13*. Springer. 2018, pp. 640–652 (cit. on pp. 7, 20, 33, 50).

[JT96] David S Johnson and Michael A Trick. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*. Vol. 26. American Mathematical Soc., 1996 (cit. on p. 8).

[Jun20] Tommi Junttila. *Propositional satisfiability and SAT solvers*. https://users.aalto.fi/~tjunttil/2020-DP-AUT/notes-sat/index.html. [Online; accessed 17-July-2024]. 2020 (cit. on p. 4).

[Kar72] Richard M. Karp. "Reducibility among Combinatorial Problems". In: *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*. Ed. by Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger. Boston, MA: Springer US, 1972, pp. 85–103. ISBN: 978-1-4684-2001-2. DOI: 10.1007/978-1-4684-2001-2_9. URL: https://doi.org/10.1007/978-1-4684-2001-2_9 (cit. on pp. 1, 2, 13, 14).

[Kau20] Matthias Kaul. "Algorithms for Exact Graph Coloring". MA thesis. University of Bonn, 2020 (cit. on pp. 33, 34).

[KB05] George Katsirelos and Fahiem Bacchus. "Generalized nogoods in CSPs". In: *AAAI*. Vol. 5. 2005, pp. 390–396 (cit. on p. 57).

[KLW22] Tom Krüger, Jan-Hendrik Lorenz, and Florian Wörz. "Too much information: Why CDCL solvers need to forget learned clauses". In: *Plos one* 17.8 (2022), e0272967 (cit. on p. 4).

[Lam+19] Sebastian Lamm, Christian Schulz, Darren Strash, Robert Williger, and Huashuo 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)*. SIAM. 2019, pp. 144–158 (cit. on p. 23).

[LC18] Alane Marie de Lima and Renato Carmo. "Exact algorithms for the graph coloring problem". In: *Revista de Informática Teórica e Aplicada* 25.4 (2018), pp. 57–73 (cit. on pp. 7, 35).

[Lia+16] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. "Learning rate based branching heuristic for SAT solvers". In: *Theory and Applications of Satisfiability Testing–SAT 2016: 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings 19*. Springer. 2016, pp. 123–140 (cit. on p. 58).

[LMM04] Corinne Lucet, Florence Mendes, and Aziz Moukrim. "Pre-processing and linear-decomposition algorithm to solve the k-colorability problem". In: *International Workshop on Experimental and Efficient Algorithms*. Springer. 2004, pp. 315–325 (cit. on p. 21).

[Lov79] László Lovász. "On the Shannon capacity of a graph". In: *IEEE Transactions on Information theory* 25.1 (1979), pp. 1–7 (cit. on p. 13).

[Mar+14] Ruben Martins, Saurabh Joshi, Vasco Manquinho, and Inês Lynce. "Incremental cardinality constraints for MaxSAT". In: *Principles and Practice of Constraint Programming: 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings 20*. Springer. 2014, pp. 531–548 (cit. on pp. 37, 44).

[Mar04] Dániel Marx. "Graph colouring problems and their applications in scheduling". In: *Periodica Polytechnica Electrical Engineering (Archives)* 48.1-2 (2004), pp. 11–16 (cit. on p. iii).

[McD79] Colin McDiarmid. "Determining the chromatic number of a graph". In: *SIAM Journal on Computing* 8.1 (1979), pp. 1–14 (cit. on p. 35).

[MKN20] Taha Mostafaie, Farzin Modarres Khiyabani, and Nima Jafari Navimipour. "A systematic study on meta-heuristic approaches for solving the graph coloring problem". In: *Computers & Operations Research* 120 (2020), p. 104850 (cit. on p. 19).

[MLM21] Joao Marques-Silva, Inês Lynce, and Sharad Malik. "Conflict-driven clause learning SAT solvers". In: *Handbook of satisfiability*. ios Press, 2021, pp. 133–182 (cit. on pp. 4, 5).

[MM02] Vasco M Manquinho and João P Marques-Silva. "Search pruning techniques in SAT-based branch-and-bound algorithms for the binate covering problem". In: *IEEE Transactions on computer-Aided Design of integrated Circuits and Systems* 21.5 (2002), pp. 505–516 (cit. on p. 57).

[MMI72] David W Matula, George Marble, and Joel D Isaacson. "Graph coloring algorithms". In: *Graph theory and computing*. Elsevier, 1972, pp. 109–122 (cit. on p. 19).

[MML14] Ruben Martins, Vasco Manquinho, and Inês Lynce. "Open-WBO: A modular MaxSAT solver". In: *Theory and Applications of Satisfiability Testing–SAT 2014: 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings 17*. Springer. 2014, pp. 438–445 (cit. on pp. 7, 44).

[MMT11] Enrico Malaguti, Michele Monaci, and Paolo Toth. "An exact approach for the vertex coloring problem". In: *Discrete Optimization* 8.2 (2011), pp. 174–190 (cit. on p. 7).

[Mos+01] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. "Chaff: Engineering an efficient SAT solver". In: *Proceedings of the 38th annual Design Automation Conference*. 2001, pp. 530–535 (cit. on p. 58).

[MS99] Joao P Marques-Silva and Karem A Sakallah. "GRASP: A search algorithm for propositional satisfiability". In: *IEEE Transactions on Computers* 48.5 (1999), pp. 506–521 (cit. on p. 3).

[MT10] Enrico Malaguti and Paolo Toth. "A survey on vertex coloring problems". In: *International transactions in operational research* 17.1 (2010), pp. 1–34 (cit. on pp. 7, 19).

[MT96] Anuj Mehrotra and Michael A Trick. "A column generation approach for graph coloring". In: *informs Journal on Computing* 8.4 (1996), pp. 344–354 (cit. on pp. 7, 8, 35).

[Myc55] Jan Mycielski. "Sur le coloriage des graphs". In: *Colloquium Mathematicae*. Vol. 3. 2. 1955, pp. 161–162 (cit. on pp. 14, 15).

[MZ08] Isabel Méndez-Díaz and Paula Zabala. "A cutting plane algorithm for graph coloring". In: *Discrete Applied Mathematics* 156.2 (2008), pp. 159–179 (cit. on p. 32).

[Ngu+17] Dang Phuong Nguyen, Michel Minoux, Viet Hung Nguyen, Thanh Hai Nguyen, and Renaud Sirdey. "Improved compact formulations for a wide class of graph partitioning problems in sparse graphs". In: *Discrete Optimization* 25 (2017), pp. 175–188 (cit. on p. 36).

[Oli+23] Albert Oliveras, Chunxiao Li, Darryl Wu, Jonathan Chung, and Vijay Ganesh. "Learning shorter redundant clauses in sdcl using maxsat". In: *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2023 (cit. on p. 57).

[OSC07] Olga Ohrimenko, Peter J Stuckey, and Michael Codish. "Propagation= lazy clause generation". In: *Principles and Practice of Constraint Programming–CP 2007: 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007. Proceedings 13*. Springer. 2007, pp. 544–558 (cit. on p. 56).

[Por20] Daniel Porumbel. "Projective cutting-planes". In: *SIAM Journal on Optimization* 30.1 (2020), pp. 1007–1032 (cit. on pp. 8, 52).

[RA14] Ryan A Rossi and Nesreen K Ahmed. "Coloring large complex networks". In: *Social Network Analysis and Mining* 4 (2014), pp. 1–37 (cit. on pp. 19, 20).

[San+13] Pablo San Segundo, Fernando Matia, Diego Rodriguez-Losada, and Miguel Hernando. "An improved bit parallel exact maximum clique algorithm". In: *Optimization Letters* 7.3 (2013), pp. 467–479 (cit. on p. 64).

[San+23] Pablo San Segundo, Fabio Furini, David Álvarez, and Panos M Pardalos. "CliSAT: A new exact algorithm for hard maximum clique problems". In: *European Journal of Operational Research* 307.3 (2023), pp. 1008–1025 (cit. on pp. 8, 14, 64).

[San12] Pablo San Segundo. "A new DSATUR-based algorithm for exact vertex coloring". In: *Computers & Operations Research* 39.7 (2012), pp. 1724–1733 (cit. on pp. 6, 20, 65).

[SB09] Niklas Sörensson and Armin Biere. "Minimizing learned clauses". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2009, pp. 237–243 (cit. on p. 57).

[Sew96] Edward C Sewell. "An improved algorithm for exact graph coloring". In: *DIMACS Series in Computer Mathematics and Theoretical Computer Science* 26 (1996), pp. 359–373 (cit. on pp. 6, 20).

[SHV09] Bas Schaafsma, Marijn JH Heule, and Hans Van Maaren. "Dynamic symmetry breaking by simulating Zykov contraction". In: *Theory and Applications of Satisfiability Testing-SAT 2009: 12th International Conference, SAT 2009, Swansea, UK, June 30-July 3, 2009. Proceedings 12*. Springer. 2009, pp. 223–236 (cit. on p. 35).

[Sin05] Carsten Sinz. "Towards an optimal CNF encoding of boolean cardinality constraints". In: *International conference on principles and practice of constraint programming*. Springer. 2005, pp. 827–831 (cit. on p. 37).

[Ste08] Kostas Stergiou. "Heuristics for dynamically adapting propagation". In: *ECAI 2008*. IOS Press, 2008, pp. 485–489 (cit. on p. 63).

[Ste21] Kostas Stergiou. "Adaptive constraint propagation in constraint satisfaction: review and evaluation". In: *Artificial Intelligence Review* 54.7 (2021), pp. 5055–5093 (cit. on p. 63).

[SW05] Thomas Schank and Dorothea Wagner. "Finding, counting and listing all triangles in large graphs, an experimental study". In: *International workshop on experimental and efficient algorithms*. Springer. 2005, pp. 606–609 (cit. on p. 42).

[Tan11] O. Tange. "GNU Parallel - The Command-Line Power Tool". In: *;login: The USENIX Magazine* 36.1 (February 2011), pp. 42–47. URL: http://www.gnu.org/s/parallel.

[TC11] Olawale Titiloye and Alan Crispin. "Quantum annealing of the graph coloring problem". In: *Discrete Optimization* 8.2 (2011), pp. 376–384 (cit. on p. 19).

[Tri02] Michael A Trick. *Computational symposium: graph coloring and its generalization*. 2002. URL: https://mat.tepper.cmu.edu/COLOR04/ (visited on July 23, 2021) (cit. on p. 7).

[Van08] Allen Van Gelder. "Another look at graph coloring via propositional satisfiability". In: *Discrete Applied Mathematics* 156.2 (2008), pp. 230–243 (cit. on pp. 5, 7, 32).

[WCB18] Robert J Woodward, Berthe Y Choueiry, and Christian Bessiere. "A Reactive Strategy for High-Level Consistency During Search." In: *IJCAI*. Vol. 2018. 2018, pp. 1390–1397 (cit. on pp. 63, 77).

[WH15] Qinghua Wu and Jin-Kao Hao. "A review on algorithms for maximum clique problems". In: *European Journal of Operational Research* 242.3 (2015), pp. 693–709 (cit. on p. 14).

[Woo69] David C Wood. "A technique for colouring a graph applicable to large scale timetabling problems". In: *The Computer Journal* 12.4 (1969), pp. 317–319 (cit. on p. iii).

[Xia+21] Mingyu Xiao, Sen Huang, Yi Zhou, and Bolin Ding. "Efficient reductions and a fast algorithm of maximum weighted independent set". In: *Proceedings of the Web Conference 2021*. 2021, pp. 3930–3940 (cit. on p. 23).

[YWH20] Emre Yolcu, Xinyu Wu, and Marijn JH Heule. "Mycielski graphs and PR proofs". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2020, pp. 201–217 (cit. on pp. 15, 32).

[Zho+14] Zhaoyang Zhou, Chu-Min Li, Chong Huang, and Ruchu Xu. "An exact algorithm with learning for the graph coloring problem". In: *Computers & operations research* 51 (2014), pp. 282–301 (cit. on pp. 7, 78).

[Zuc06] David Zuckerman. "Linear degree extractors and the inapproximability of max clique and chromatic number". In: *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*. 2006, pp. 681–690 (cit. on p. iii).

[Zyk49] Alexander Aleksandrovich Zykov. "On some properties of linear complexes". In: *Matematicheskii sbornik* 66.2 (1949), pp. 163–188 (cit. on p. 34).