Constraint and Satisfiability Reasoning for Graph Coloring

Emmanuel Hebrard   HEBRARD@LAAS.FR
LAAS-CNRS, ANITI, Université de Toulouse, CNRS, France

George Katsirelos   GKATSI@GMAIL.COM
MIA Paris, INRAE, AgroParisTech and
ANITI, Université de Toulouse, France

Abstract

Graph coloring is an important problem in combinatorial optimization and a major component of numerous allocation and scheduling problems.

In this paper we introduce a hybrid CP/SAT approach to graph coloring based on the addition-contraction recurrence of Zykov. Decisions correspond to either adding an edge between two non-adjacent vertices or contracting these two vertices, hence enforcing inequality or equality, respectively. This scheme yields a symmetry-free tree and makes learnt clauses stronger by not committing to a particular color.

We introduce a new lower bound for this problem based on Mycielskian graphs; a method to produce a clausal explanation of this bound for use in a CDCL algorithm; a branching heuristic emulating Brélaz' heuristic on the Zykov tree; and dedicated pruning techniques relying on marginal costs with respect to the bound and on reasoning about transitivity when unit propagating learnt clauses.

The combination of these techniques in both a branch-and-bound and in a bottom-up search outperforms other SAT-based approaches and Dsatur on standard benchmarks both for finding upper bounds and for proving lower bounds.

1. Introduction

Many applications require to partition items into a minimum number of subsets of compatible elements. For instance, when allocating frequencies, we may want to partition devices so that the physical distance between any two devices in a subset is large enough to share a frequency without interference (Aardal et al., 2007; Park & Lee, 1996). Alternatively, in compilers, we may want to partition frequently-accessed variables into registers so that two variables with overlapping live ranges are not given the same register (Chaitin et al., 1981).

Many such combinatorial problems can be seen as a *Graph Coloring Problem*. The compatibility structure is represented by a graph $G = (V, E)$ where V (the vertices) is a set of items and E (the edges) is a binary relation on V representing incompatibility. This problem is NP-hard, and in fact deciding if the chromatic number of a graph is lower than a given constant is one of Karp's 21 NP-complete problems (Karp, 1972).

Although local search and meta heuristics can be extremely efficient for that problem, there are cases where proving optimality is important. For instance, in 2016, the U.S. Federal Communications Commission (FCC) conducted a reverse auction to acquire radio spectrum from TV broadcasters. At each iteration of the auction, the quote given by the FCC decrease and some potential sellers may opt out. The auction ends when it is not possible to allocate frequency bands (i.e., colors) to the broadcasters who opted out

of the auctions (i.e., vertices) so that two broadcasters that can potentially interfere (i.e., adjacent) are allocated distinct frequency bands. This problem was solved by Newman et al. (2017) using a portfolio of mainly SAT-based solvers. Indeed, having a *guarantee* that the spectrum was acquired at the best price for the taxpayer was extremely important given the magnitude of the transaction.

Contributions. In this work[1], we propose a novel approach to solving the vertex coloring problem. Our method is a hybrid of *Conflict-Driven Clause Learning* (Marques-Silva & Sakallah, 1999; Moskewicz et al., 2001) with techniques tailored to graph coloring. We use the idea of integrating constraint programming into clause learning satisfiability solvers by having each propagator label each pruning or failure by a clausal *explanation* (Katsirelos & Bacchus, 2005; Ohrimenko et al., 2007).

One of the main characteristics of the method we propose is to use a branching scheme proposed by Zykov (1949) whereby a choice point consists in either contracting two non-adjacent vertices, or adding an edge between them. This is not the first modern algorithm to use this technique. The branch and price algorithm proposed by Mehrotra and Trick (1995) branched using Zykov recurrence. Schaafsma et al. (2009) introduced edge variables, i.e. variables whose truth value indicates whether two vertices get the same color, in order to learn more general conflict clauses than the standard color-based SAT model permits. However, whereas the former method is based on an implicit exponential set of variables for computing a lower bound and the latter still requires standard variables standing for color assignment in order to branch and propagate, the method we propose relies entirely on edge variables.

There are two significant advantages to using Zykov recurrence. Firstly, color symmetries are completely eliminated, at no significant computational cost. Secondly, when exploring a branch of the search tree, the density of the graph increases (through vertex contractions and edge additions) and computing a maximal clique of this graph gives an effective lower bound on the chromatic number.

Moreover, the use of a hybrid with constraint programming helps alleviate the potential drawbacks of this search scheme. Most importantly, edge variables can be made consistent at a reasonable computational cost via a dedicated propagator, whereas a clausal decomposition would be too costly and propagation using a partial coloring would be weaker. Moreover, it is possible to emulate Brélaz' DSATUR heuristic without maintaining a partial coloring, and experimental evaluation of the different approaches show that this is effective.

The method we propose also uses a novel lower bound, stronger than cliques, based on finding a generalization of Mycielskian graphs (Mycielski, 1955) embedded in the current graph. Mycielskian graphs can have large chromatic number without necessarily containing large cliques. For example, the graphs $M(k)$ described by Mycielski have chromatic number k and are triangle-free, i.e., they contain no clique of size 3. Therefore, the Mycielskian bound can potentially be stronger than the clique number. Since the two bounds (cliques and Mycielskians) rely on the same principle of isolating subgraphs with large chromatic number, failures have similar clausal explanations: at least one pair of vertices for which an edge was added must be contracted in order to find an improving coloring.

---

1. This article is an extended version of previously published work (Hebrard & Katsirelos, 2018).

Then, we show how to detect vertex contractions or edge additions whose marginal cost with respect to either of these bounds violates the upper bound. In that case the domains can be pruned accordingly. Thanks to this inference, our method is provably stronger than the standard branch and bound method in the sense that the search tree explored by the latter is larger.

As a final contribution, we describe two additional new inference techniques. First we show how the propagation of learnt nogoods can be improved by taking into account the current state of the addition/contraction graph. This technique yields stronger, more general conflict clauses. Second, we introduce a preprocessing method whereby we extract an arbitrary independent set of the graph and replace it by constraints, thus yielding a model with fewer variables and which is stronger with respect to constraint propagation. These techniques can sometimes improve the performance of the proposed approach. In particular, the former significantly improves the results on two classes of benchmarks. However, on average over the whole data set we used, the impact is nearly null for the former and slightly negative for the latter.

2. Background

Graphs. A graph G is a pair (V, E) where V is a set of vertices and E is a set of edges. An edge is a binary set of distinct vertices, i.e., $E \subseteq V^2 \setminus \{(vv) \mid v \in V\}$. When $(uv) \in E$, we say that u and v are adjacent. We denote $N_G(v)$ the neighborhood $\{u \mid (uv) \in E\}$ of v in G (or $N(v)$ when there is no ambiguity about the graph).

A *coloring* c of a graph is a labeling of its vertices such that adjacent vertices have distinct labels, and its cardinality is the total number of distinct labels. The *chromatic number* $\chi(G)$ of a graph G is the minimum cardinality of a coloring of G. The Graph Coloring Problem asks for a minimum coloring of a graph and the decision problem of whether there exists a coloring with at most $k \geq 3$ colors is NP-complete.

A clique (independent set) is a set of vertices $C \subseteq V$, such that all pairs (no pairs) of vertices u, v of C are adjacent. The clique number (stability number) of a graph G, denoted $\omega(G)$ ($\alpha(G)$), is the size of the largest clique (independent set) contained in G.

Since no two vertices in a clique can take the same color, we have $\omega(G) \leq \chi(G)$. Therefore, the size of a clique (maximum or not) can be used as lower bound. A coloring can be seen as a partition of a graph into independent sets (since no adjacent vertices may have the same color) and finding the optimum coloring is a set covering problem over all independent sets.

Constraint Satisfaction Problem. A constraint satisfaction problem is a triple $(\mathbf{X}, \mathbf{D}, \mathbf{C})$ where $\mathbf{X}$ is a set of variables, $\mathbf{D}$ is the domain function and $\mathbf{C}$ is a set of constraints. The domain function maps each variable $X \in \mathbf{X}$ to the set of its possible *values*. A (partial) assignment to $\mathbf{X}' \subseteq \mathbf{X}$ is a mapping from each $X \in \mathbf{X}'$ to a value in $\mathbf{D}(X)$. A complete assignment is a partial assignment to $\mathbf{X}$. A constraint c is a pair $(scope(c), P_c)$ where $scope(c) \subseteq \mathbf{X}$ and P_c is a polynomial time predicate over assignments to $scope(c)$. A constraint is satisfied by an assignment A if $P_c(A)$ is satisfied. A CSP is satisfied by a complete assignment if every constraint c is satisfied by the restriction of that assignment to $scope(c)$. Solving a CSP means finding a satisfying assignment and determining whether that exists is an NP-complete problem.

A value $v \in \mathbf{D}(X)$ is domain consistent (also called generalized arc consistent or GAC) with respect to a constraint $C = (S, P)$ with $X \in S$ there exists an assignment A to S that satisfies C and such that $A(X) = v$, otherwise it is domain inconsistent. A constraint is domain consistent if for all variables $X \in scope(c)$ and all values $v \in \mathbf{D}(X)$, v is domain consistent with respect to c. A CSP is domain consistent if all its constraints are domain consistent. Domain consistency can be enforced by removing values from domains that are domain inconsistent, called pruning, until the unique domain consistent fixpoint is reached, or a constraint has no remaining satisfying assignments. Depending on the predicate of a constraint, the complexity of checking that a value is domain consistent may range anywhere from polynomial time to NP-complete.

CSP solvers that we care about in this paper solve an instance with backtracking search. Branching is typically binary, for example by assigning a variable to one of the values in its domain in one branch and removing that value in the other branch. Other schemes exist, but are not relevant to this work. At each node of the search, the solver executes a propagator for each constraint, which tightens the domains of the variables in its scope in order to enforce domain consistency, or potentially some weaker form of consistency. In the latter case, we say that the propagator is incomplete. Crucially for the success of constraint programming in general, propagators can execute any arbitrary algorithm, in particular algorithms that take advantage of domain knowledge and whose propagation cannot be replicated by simpler constraints (Bessiere et al., 2009).

Boolean Satisfiability. In satisfiability (SAT), we express problems with Boolean variables $\mathbf{X}$. We say that a literal l is either a variable x or its negation $\neg x$ and we write $x = var(l)$. Moreover, we write $\bar{l} = \neg l$. Clauses are disjunctions of literals, written interchangeably as sets of literals or as disjunctions, which are satisfied by an assignment if it assigns at least one literal to true. A formula is a conjunction of clauses, written also as a set of clauses and is satisfied by an assignment if it satisfies all clauses. The SAT problem is determining whether such a satisfying assignment exists and is NP-hard (Cook, 1971).

A SAT instance is closed under unit propagation if it contains no unit clauses. It is failed if it contains the empty clause. In order to compute the unit closure of an instance, for each unit clause (l) we remove all clauses that contain l and remove $\bar{l}$ from other clauses. We call this procedure *performing unit propagation*.

The state of the art in solving SAT is the Conflict-Driven Clause Learning (CDCL) algorithm (Marques-Silva & Sakallah, 1999; Moskewicz et al., 2001). The basic techniques used in CDCL are a variant of backtracking search with unit propagation at each node, frequent restarting and *far-backjumping*, learning implied constraints from conflicts in the backtracking search which empower unit propagation (clause learning) and activity based branching (VSIDS).

The techniques of CDCL can be readily adapted in CSP solvers, except for clause learning. It has been shown (Katsirelos & Bacchus, 2005) that it is possible to do this as well by using an appropriate propositional encoding of domains and generating clausal explanations for each pruning and each constraint failure. This can be further improved for some highly useful classes of constraints by using a domain encoding that takes domain order into account (Ohrimenko et al., 2007). We shall abuse CSP notations when considering

Boolean literals. In particular, for a constraint involving a Boolean variable x, we shall say that x (resp. $\neg x$) is GAC if and only if the assignment $x = 1$ (resp. $x = 0$) is GAC.

3. Related Work

Complete approaches for the graph coloring problem can be broadly divided into three categories: branch and bound, branch and price, and satisfiability-based.

Branch & Bound. One of the oldest and most successful techniques is a backtracking algorithm using Brélaz' DSATUR heuristic (Brélaz, 1979): when branching, the vertex with highest *degree of saturation* is chosen and colored with the lexicographically least candidate. The degree of saturation of a vertex v is the number of assigned colors within its neighborhood $N(v)$ in G. In case of a tie, the vertex with largest number of uncolored neighbors is chosen among the tied vertices. This heuristic is often used within a branch-and-bound algorithm with one variable per vertex whose domain is the set of possible colors. It is known as `dom+deg` in the CSP literature (Frost & Dechter, 1995). More sophisticated tie-breaking heuristics have been shown to improve DSATUR (Segundo, 2012).

One common caveat of DSATUR-based branch and bound algorithms, however, is that the standard lower bound, which is the size of a maximal (not necessarily maximum) clique, is computed only once at the root of the search tree. Indeed, the algorithm branches on the coloring, but the structure of the graph does not change, hence there is no point in computing another clique. Consequently, Furini et al. (2016) proposed three lower bound techniques taking advantage of the coloring decisions. The first introduces an auxiliary graph where the vertices sharing the same color are merged on which cliques are easier to compute and are potentially larger. The second relies on computing the *Lovász Theta number* (Lovász, 1979), which lies between the clique number and chromatic number of a graph (Grötschel et al., 1981) and which can be computed in polynomial time using semi-definite programming. Finally, the third bound uses a different auxiliary graph whose independent sets can be mapped to coloring of the original graph (Cornaz & Jost, 2008).

Branch and price. A second type of approaches rely on the *column generation* method (Mehrotra & Trick, 1995; Malaguti et al., 2011). The problem is a seen as a set covering problem over all independent sets of the graph which are the 0-1 variables of the integer linear program. Independent sets, however, are dynamically generated and when no independent set that can potentially improve the master problem can be found, the current solution of the set covering is an optimal coloring. This method is significantly better than DSATUR-based approaches on some classes of graphs, however, it can be significantly outperformed on some other classes (Malaguti & Toth, 2010; Segundo, 2012).

Boolean Satisfiability. A third type of approaches rely on Boolean Satisfiability. In order to encode graph coloring with satisfiability, some approaches rely on *color* variables x_{vi}, where x_{vi} being true means vertex v takes color i. For every edge (uv), there is a binary clause $\overline{x}_{vi} \vee \overline{x}_{ui}$ for every color i. If k is the maximum number of colors, then there is a clause $\bigvee_{1 \leq i \leq k} x_{ui}$. This corresponds to the direct encoding (Walsh, 2000) of the constraint model where we have a color variable X_v for each vertex v, whose domain ranges from 1 to k and which indicates its assigned color and a constraint $X_v \neq X_u$ for all pairs of adjacent vertices. The CP and SAT models are exactly identical, so we refer to both by

the same name. Refinements to this encoding include Van Gelder's log encoding versions, where x_{vj} is true if the j-th bit of the binary encoding of the color taken by vertex v is 1 (Van Gelder, 2008). However, the use of modern SAT solving techniques like restarting (Gomes et al., 1998; Huang, 2007) and clause learning (Marques-Silva & Sakallah, 1999) are not straightforward to combine with symmetry breaking such as that of Van Hentenryck et al. (2003) since it relies on dedicated branching. They can only be easily combined with starting from an arbitrary coloring to a clique, but that is incomplete. The `Color6` solver (Zhou et al., 2014) uses symmetry breaking branching but forgoes restarting to maintain complete symmetry breaking.

Zykov's recurrence. In contrast to branching on colors, the search tree induced by the recurrence relation (1) below, due to Zykov (1949), has no color symmetry.

Let $G/(uv)$ be the graph obtained by *contracting* u and v: the two vertices are removed and replaced by a single new vertex $r(u) = r(v)$ and every edge (vw) or (uw) is replaced by $(r(v)w)$. Conversely, let $G + (uv)$ be the graph obtained by adding the edge (uv), in which case we say that the vertices are *separated*.

$$\chi(G) \quad = \quad \min\{\chi(G/(uv)), \chi(G + (uv))\} \tag{1}$$

Indeed, given a coloring of G, either the vertices u and v have distinct colors hence it is also a coloring of $G + (uv)$, or they have the same color and it is a coloring of $G/(uv)$.

Moreover, branching using this recurrence yields graphs with non-decreasing sets of edges. Therefore, using the clique number as lower bound is easy, and this lower bound can only increase when branching. In contrast, in the color variable formulation, the standard technique is to compute a clique at the root node only as discussed earlier.

Example 1. *Figure 1 illustrates the Zykov recurrence. From the graph G in Figure 1a, we obtain the graph $G + (cd)$ shown in Figure 1b by adding the edge (cd) and the graph $G/(cd)$ shown in Figure 1c. One of these two graphs has the same chromatic number as G.*

(a) G (b) $G + (cd)$ (c) $G/(cd)$
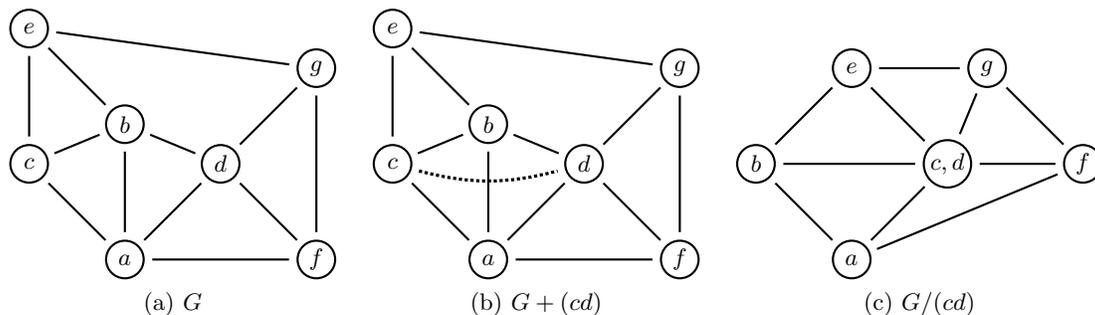
Figure 1: Zykov recurrence

This branching scheme was successfully used in a branch-and-price approach to coloring (Mehrotra & Trick, 1996). In the context of satisfiability, Schaafsma et al. (2009) introduced a novel and clever way of taking advantage of Zykov's idea: first, they introduce *edge* variables for every pair of non-adjacent vertices. When an edge variable is set to

true, the two vertices get the same color, and are assigned different colors otherwise. Then, when learning a clause involving color variables, they show that one can compactly encode all symmetric clauses using a single clause that only uses edge variables and propagates the same as if all the symmetric clauses were present. In their approach, they only introduce edge variables as needed and they have no lower bound computation except for that achieved by attempting to assign the color variables. They achieve good empirical results, but the scheme is quite complicated to implement.

4. Zykov Tree Search

In our approach, similar to that of Schaafsma et al., we use a model which leads to the exploration of the tree resulting from the Zykov recurrence. We have one Boolean edge variable e_{uv} for each non-edge of the input graph with the same semantics as those used for edge variables by Schaafsma et al. A complete assignment partitions the vertices of G into a set of independent sets, which correspond to colors. The number of independent sets is the size of the coloring that we have computed. We somewhat abuse notation in the sequel and write clauses using variables e_{uv} even when $(uv) \in E$ and assume that the variable is always false. Moreover, in this paper we consider only undirected graphs, hence e_{uv} and e_{vu} are the same variable.

The decision to assign the same color to two vertices is the same as following the branch that contracts the vertices in the Zykov recurrence, while assigning them different colors corresponds to the branch that adds an edge between them. Therefore, we can associate a graph G_X with a (partially assigned) set of edge variables X. G_X is the graph that results from contracting all vertices u, v of G for which e_{uv} is true and separating (adding an edge between) all pairs of vertices u, v of G for which e_{uv} is false. When e_{uv} and e_{vw} are both true, this means that we contract u and v and then contract w and $r(v)$. Similarly, if e_{uv} is true and e_{vw} is false, we contract u and v and add an edge between $r(v)$ and w. The operation of contracting vertices is associative and commutative, so we get the same graph G_X regardless of the order in which we process the variables in X. It is also a bijection, so each contracted graph can be mapped back to exactly one assignment. We can therefore use X and G_X interchangeably.

We now describe a propagation algorithm for the following constraint COLORING, for a graph $G = (V, E)$, an integer k, and the set of variables $X = \{e_{uv} \mid u, v \in V\}$:

$$\text{COLORING}(G, k, X) \equiv \quad \exists c : V \mapsto \mathbb{N} \quad \begin{aligned} c(u) = c(v) &\implies e_{uv} \\ \wedge \ |\{c(u) \mid u \in V\}| &\leq k \end{aligned} \quad \wedge \qquad (2)$$

$$e_{uu} \quad \forall u \in V \qquad \wedge \qquad (3)$$

$$\bar{e}_{uv} \quad \forall (uv) \in E \qquad (4)$$

The constraint COLORING is satisfied by any assignment of the edge variables that corresponds to a coloring with fewer than k colors, and is therefore clearly NP-hard. Equation 2 enforces that the equalities are consistent with a k-coloring, while constraints 3 and 4 ensure that the coloring is legal with respect to the edges of the graph.

Moreover, the property of having the same color is transitive, so if e_{uv} and e_{vw} are true, then so is e_{uw}. Transitivity also implies that if e_{uv} is true and e_{vw} is false, then e_{uw} must also be false. We enforce transitivity using the implied constraint TRANSITIVITY:

$$\text{TRANSITIVITY}(G, X) \equiv (e_{uv} \ \wedge \ e_{vw}) \implies e_{uw} \qquad \forall u, v, w \in V \qquad (5)$$

We can enforce GAC on this constraint using a clausal decomposition of size $O(|V|^3)$:

$$(\bar{e}_{uv} \vee \bar{e}_{vw} \vee e_{uw}) \qquad \forall u, v, w \in V \qquad (6)$$

Performing unit propagation on this decomposition takes $O(|V|^3)$ time cumulatively over a branch of the search tree. In our implementation, we have opted instead for a dedicated propagator described in Section 4.2, whose complexity over a branch is only $O(|V|^2)$.

Let $|G_X|$ be the number of vertices of the graph G_X resulting from the additions and contractions X. From Zykov (1949), we know that there exists a transitive assignment of X, compatible with the edges of G such that $|G_X| = \chi(G)$ (and trivially, $|G_X| \geq \chi(G)$ for any X). Therefore, the constraint COLORING can be rewritten:

$$
\begin{aligned}
\text{COLORING}(G, k, X) \equiv \quad & \text{TRANSITIVITY}(G, X) && \wedge \\
& |G_X| \leq k && \wedge \\
& e_{uu}, \ \forall u \in V && \wedge \\
& \bar{e}_{uv}, \ \forall (uv) \in E &&
\end{aligned}
$$

We describe two algorithms for computing lower bounds on $|G_X|$ in Sections 5.1 and 5.2. The first algorithm computes the well known clique lower bound (Section 5.1) on the graph G_X and the second a novel, stronger, bound (Section 5.2). If that bound meets or exceeds k, the propagator fails and produces an explanation. Moreover, we describe how to (incompletely) prune the domains of with respect to these bounds in Section 6.3.

Neither of the bounds that we use is cheap to compute, and computing marginal costs in order to prune domains can be expensive as well, hence the propagator for the bound runs at a lower priority than unit propagation and the TRANSITIVITY constraint.

Clearly, our approach is closely related to that of Schaafsma et al. However, there are some important differences. First, since we do not need the color variables to compute the size of the coloring, we completely eliminate the need for the clause rewriting scheme that they implement and get color symmetry-free search with no additional effort. In addition, since our model is a CP/SAT hybrid model, we can use a constraint to compute a lower bound at each node, thus avoiding a potentially large number of conflicts.

The approach of Schaafsma et al. of decomposing the constraint TRANSITIVITY using color variables (so that $X_v = X_u \iff e_{uv}$) does not enforce domain consistency on the constraint TRANSITIVITY. For instance, consider three vertices u, v and w such that X_v, X_u and X_w can all take values in $\{1, 2\}$ and we have the unit literals e_{uv} and $\bar{e}_{uw}$. The literal e_{vw} is not GAC for TRANSITIVITY, however it is GAC for the decomposition. In contrast,

the propagator described in Section 4.2 maintains GAC on this constraint, without having to encode channeling between the edge variables and the color variables.

The main drawback of this model is that we need a large number of variables. This is especially problematic for large, sparse graphs, where the number of non-edges is quadratic in the number of vertices and significantly larger than the number of edges. Indeed, in 4 of the 125 instances we used in our experimental evaluation, our solver exceeded the memory limit[2]. The approach of Schaafsma et al. does not have the same limitation, as they introduce variables only when they are needed to rewrite a learnt clause, in a way similar to lazy model expansion (De Cat et al., 2015). It is possible that this approach of lazily introducing variables can be adapted to our model, but this, as well as other ways of reducing the memory requirements, remains future work.

4.1 Optimization Strategy

Previous satisfiability-based approaches to coloring have mostly ignored the optimization problem of finding an optimal coloring of a graph and instead attack the decision problem of whether a graph is colorable with k colors. In our setting, we have the flexibility to do both. In particular, before starting search the solver computes initial bounds $\chi^{low} \leq \chi \leq \chi^{up}$ for the chromatic number, using the bound described in section 5.1 or 5.2 and a run of the Dsatur heuristic, respectively.

After that, it may use one of two search strategies: branch-and-bound or bottom-up. The former uses a single instance of a solver, uses χ^{up} to create the initial Coloring constraint and then iteratively finds improving solutions. For each solution, it adds to the model a new Coloring constraint with the new, tighter, upper bound[3]. This is similar to the top-down approach one would use when solving a series of decision problems, starting from a heuristic upper bound and decreasing that until we generate an unsatisfiable instance, in which case we have identified the optimum. The advantage of the branch-and-bound approach is that it does not discard accumulated information between solutions, namely learned clauses and heuristic scores for variables. Moreover, it more closely resembles the typical approach used in constraint programming systems.

The bottom-up approach temporarily sets $\chi^{up} = \chi^{low}$ and tries to solve the resulting instance, i.e., one in which $k = \chi^{up}$ in the Coloring constraint. If the instance is satisfiable, it terminates with $\chi = \chi^{low}$, otherwise it increases χ^{low} by one and iterates until the lower bound meets the true chromatic number. This has none of the advantages of the branch-and-bound approach, as it is not safe to reuse clauses from a more constrained problem in one that is less constrained. Moreover, it cannot generate upper bounds before it finds the optimum. But it gains from the fact that the more constrained problems it solves may be easier. One particular behavior we have observed is that sometimes the lower bound computed at the root coincides with the optimum and finding the matching upper bound is quite easy with the bottom-up strategy, while finding that solution with branch-and-bound can be very hard.

---

2. CPLEX exceeded the memory limit in 16 cases.
3. The actual implementation makes this modification in-place so that the number of Coloring constraints remains 1, regardless of the number of solutions found.

4.2 Transitivity Propagation

The propagator for the TRANSITIVITY (G, X) constraint works by maintaining a graph H that is updated to be equal G_X as the current partial assignment A grows and using that to compute the literals that must be propagated. For brevity, in the rest of this section, we always write $N(v)$ to mean $N_H(v)$ and abuse the neighborhood notation to write $N(S)$ for $\bigcup_{u \in S} N(u)$. Additionally, the propagator maintains for each vertex v a bag $b(v)$ to which it belongs. Each unique bag corresponds to a vertex in H. Initially, $b(v) = \{v\}$ for all v, since no contractions have taken place.

We set the propagator for TRANSITIVITY (G, X) to be invoked for every literal that becomes true. Call this literal p. When the propagator is invoked for $p = e_{uv}$, which corresponds to the contraction of u and v, then all vertices $v' \in b(v)$ and $u \in b(u)$ must be assigned the same color, so it sets $e_{u'v'}$ to true for all $v' \in b(v), u' \in b(u)$. The contraction implies that every vertex in $b(v)$ must be separated from all vertices in $N(b(u))$ with which it does not already have an edge (and symmetrically for vertices in $b(u)$). Hence the propagator also sets $e_{u'v'}$ to false for all $v' \in b(v)$ and $u' \in N(b(u) \setminus N(b(v)))$ and for all $v' \in N(b(v) \setminus N(b(u)))$. Finally, it sets $B = b(u) \cup b(v)$, updates $b(v') = B$ for all $v' \in B$ and contracts v and u in H. In the case where the propagator is invoked for $p = \overline{e_{uv}}$, it sets $e_{u'v'}$ to false for all $v' \in b(v), u' \in b(u)$ and adds an edge between v and u in H.

A small but important optimization is that if the propagator is invoked for e_{uv} becoming true (resp. false) but u and v are already in the same bag (resp. already separated) then it does nothing. This ensures that it touches each non-edge at most once, hence its complexity is quadratic over an entire branch. This is also optimal, since in the worst case every variable must be set either as a decision or by propagation. Notice also that the operation of merging and keeping track of $b(v)$ can be performed in amortized constant time using the union-find data structure (Tarjan & van Leeuwen, 1984), where e_{uv} becoming true entails the union operation Union(v, u) and $r(v) = $ Find(v). Of the two, the latter is important, as we can check whether two vertices v, u have been contracted by checking whether Find$(v) = $ Find(u), or $r(v) = r(u)$. On the other hand, amortized constant time Union is not important, because contracting two vertices u, v has complexity $O(|b(v)||b(u)|)$, which dominates even a naive implementation of Find. Indeed, since the algorithm has to iterate over all vertices in both bags anyway, our implementation simply updates $r(u)$ as it performs this iteration.

This propagator uses the clauses (6) as explanations. The way it chooses explanations for literals that it propagates is fairly straightforward, using the vertices involved in the literal that woke the propagator as pivots. For example, if $b(v) = \{v, v'\}$, $b(u) = \{u, u'\}$ and it is woken on the literal e_{uv}, it sets $e_{uv'}$ using $(\overline{e}_{vv'} \vee \overline{e}_{uv} \vee e_{uv'})$ as the reason, $e_{u'v}$ using $(\overline{e}_{uu'} \vee \overline{e}_{uv} \vee e_{u'v})$, and only then finally setting $e_{u'v'}$ using $(\overline{e}_{uv'} \vee \overline{e}_{uu'} \vee e_{u'v'})$ which relies on the fact that $e_{uv'}$ has been just set to true.

4.3 Transitivity-Aware Unit Propagation

Consider a (learned) clause $c = (F \vee e_{vu} \vee e_{vw})$, where F is a set of literals which are all false in the current node of the search tree and u, w are in the same bag. Notice that if we set e_{vw} to false, the transitivity propagator will also set e_{vu} to false, since v and w are in the same bag. This will violate clause c and hence we should set both e_{vw} and e_{vu} to

true. More generally, when $c = (F \vee B)$, all the literals in F are false, and B are the only unset literals, they have the same polarity and are over vertices that belong to exactly two bags, the literals in B will all necessarily get the same value: we will either merge the two bags, so all variables become true, or we will add an edge between them and all variables become false. So if the unset literals are negative, we must satisfy them all by adding an edge between the bags, while if they are positive, we must satisfy them all by merging the two bags.

Unit propagation is unable to make any inference when more than one literal is unset, while the transitivity constraint is unaware of the learned clauses, so this inference cannot be made by any single component of the solver. In order to detect this situation, the unit propagation procedure can be made aware of vertices and partitions. In particular, suppose we have a clause c, watched by l_1 with $var(l_1) = e_{uv}$ and l_2, and l_2 has just become false. The standard unit propagation algorithm finds another, non-false, literal $l_3 \neq l_1$ and uses it as the watch, or determines that the clause is unit or failed if no such literal exists. Instead of stopping on the first non-false literal, we keep on searching for a literal l_3 that is either a) true, b) has different polarity to l_1 or c) $var(l_3) = e_{wz}$ and $\{r(u), r(v)\} \neq \{r(w), r(z)\}$, i.e., u and v are not assigned to exactly the same partitions as w and z. If any of these conditions is true, we use l_3 as the new watch and continue. If not, then the clause has been reduced to a positive or negative clause over variables involving exactly two partitions and hence other literals must either all be true or all false because of the transitivity constraint. Since the clause must be satisfied, we must make one of them true.

We cannot use the clause c as the reason for this propagation, since it is not unit. To construct an explanation, let l be the literal that we have chosen to make true, with $var(l) = e_{uv}$, and let V_B be the vertices $\{v \mid l' \in B \wedge var(l') = e_{uv}\}$. In other words, the set B is the set of vertices which appear as a subscript of a literal e_{uv} or $\bar{e}_{uv}$ in B. Partition V_B into V_1 and V_2 such that $V_1 = \{w \mid w \in V_B \wedge r(w) = r(v)\}$ and $V_2 = V_b \setminus V_1$. Then, the explanation is $F \cup \{l\} \cup \{\bar{e}_{uu'} \mid u' \in V_1 \setminus \{u\}\} \cup \{\bar{e}_{vv'} \mid v' \in V_2 \setminus \{v\}\}$. In words, the clause states that because all literals in F are false, the vertices in V_1 are in the same bag as u, and the vertices in V_2 are in the same bag as v, we must make l true.

Example 2. *Consider again the clause $c = (F \vee e_{vu} \vee e_{vw})$ where all literals in F are false, e_{vu} and e_{vw} are unset and u, w are in the same bag. Since this clause meets the conditions defined above, we can make the literal $l = e_{vw}$ true. Since the clause c has two unset literals, it cannot be used as the clausal explanation for setting l to true. To construct the explanation, we proceed as above with $B = \{e_{vu}, e_{vw}\}$, $V_B = \{v, u, w\}$ and partition this into $V_1 = \{u, w\}$ and $V_2 = \{v\}$. The clausal explanation we compute is $F \cup \{e_{vw}\} \cup \{\bar{e}_{uw}\}$.*

To see why this a valid explanation, consider the case where the literal l is positive, i.e., $l = e_{vu}$. We trace back the steps that propagation of the TRANSITIVITY constraint would perform if we set l to false. For each literal $l' \in B$, $l' = e_{v'u'}$, $l \neq l'$, $u' \in V_1$, $v' \in V_2$, we resolve with the clauses $\{\bar{e}_{uu'}, \bar{e}_{u'v}, e_{uv}\}$ and $\{\bar{e}_{vv'}, \bar{e}_{u'v'}, e_{u'v}\}$, both of which are entailed by the TRANSITIVITY constraint. This eliminates the literal $e_{u'v'}$, introduces $\bar{e}_{uu'}$ and $\bar{e}_{vv'}$ as required and leaves e_{uv} in the clause. Once we repeat this for all literals $l' \neq l, l' \in B$, we get the explanation clause described above. Similar reasoning can be used when $v = v'$ or $u = u'$ and for the case where l is negative.

This feature makes unit propagation stronger at a moderate increase in runtime cost. In our experiments in Section 7, we show that while using this technique does not pay off overall, it improves performance in some classes.

4.4 Emulating Brélaz Heuristic in Zykov Search Tree

Schaafsma et al. have already shown that any particular partial coloring can be mapped to a node in the Zykov recurrence. We have alluded to the procedure for doing this earlier: we contract all vertices that get the same color and add edges between vertices that get different colors. The correspondence also works in the other direction, although it is not unique: from every node of the Zykov recurrence, we can construct several possible partial colorings. Consider a node n of the Zykov recurrence and the graph G_n at that node. We can choose any maximal clique in G_n and color its vertices arbitrarily to get a partial coloring. Since there are (exponentially) many cliques in a graph, this is clearly not a unique mapping. However, it allows us to replicate in the Zykov model any computation that requires a coloring to work with.

In particular, Brélaz' branching heuristic remains extremely competitive for finding good colorings, as evidenced by the performance of the `Dsatur` method in our experimental evaluation (Section 7). Moreover, Schaafsma et al. observed that branching on color variables was significantly better than branching on edge variables. We therefore use the observation above to emulate this heuristic without introducing color variables.

Indeed, adding color variables is not really desirable. First, it adds the overhead of propagating the reified equality constraints. Second, using these variables to follow DSATUR requires branching on them, which in turn requires some kind of symmetry breaking method, like the rewriting scheme of Schaafsma et al. So it would be preferable to get the benefit of the more effective branching heuristic without needing to introduce color variables.

As shown earlier, we can achieve behavior similar to that of DSATUR in the edge variable model, by exploiting the mapping from the current graph G_X back to a coloring of the original graph. To do this, we pick a maximal clique C in G_X. We pick the vertex v that maximizes $|N(v) \cap C|$, breaking ties by highest $|N(v) \setminus C|$, and an arbitrary vertex $u \in C \setminus N(v)$. Notice that u may not exist when the original graph is not connected. In this case we pick a different clique until we find a clique C with $N(v) \cap C \neq \emptyset$. We then set e_{uv} to true. This uses the current maximal clique to implicitly construct a coloring and uses that to choose the next vertex to color as DSATUR does. If the assignments e_{uv} are refuted for all $u \in C$, then v is adjacent to all vertices in C and so $C \cup \{v\}$ is a larger clique, which corresponds to using a new color in Brélaz' heuristic. This construction generalizes the construction we discussed earlier. From a node of the Zykov recurrence, we can construct not only a coloring, but also the state of the domains in the color variable model under that coloring, so that, for example, the current domain size of X_v is $\chi^{up} - |N(v) \cap C|$.

This branching strategy can be more flexible than committing to a coloring by assigning the color variables. For example, unit propagation on learned clauses as we explore a branch of the search tree can make it so that the maximal clique C' at some deep level is not an extension of the maximal clique C at the root of the tree, i.e., $C \nsubseteq C'$. This is further supported by our experiments juxtaposing the two algorithms for finding cliques that we describe in Section 5.1: the one that tries to extend a clique performs overall worse than the

computationally more costly algorithm which non-incrementally searches for cliques in the entire graph. The Brélaz heuristic on the color variables commits to using C at the root, hence cannot take advantage of the information that C' is a larger clique. The modification that we present here achieves this.

5. Lower Bounds

As we already discussed, an important advantage of the edge-variable based model is that computing a lower bound for the current subproblem is as easy as for the entire problem. For example, for X the set of partially assigned edge variables, the clique number of the graph G_X is a lower bound for the subproblem.

5.1 Clique-Based Lower Bound

In order to find a large clique we use the following greedy algorithm, called `GreedyCliqueFinder`. Let o be an ordering of the vertices, so we visit all vertices in the order $v_{o_1}, \ldots, v_{o_n}$. We maintain an initially empty list of cliques. Iterating over all vertices in the order o, we add each vertex to all the cliques which admit it and if no clique admits it we put it in a new singleton clique. When this finishes, we iterate over the vertices one more time and add them to all cliques which admit them, because in the first pass a vertex v was not evaluated against cliques which were created after we processed v. We then pick the largest among these cliques as our lower bound.

The number of bitwise operations performed by this algorithm is $O(|V|^2)$. Indeed, we construct at most $|V|$ cliques and for each vertex we check whether it can be added to each clique. The bitwise operations each have cost $O(|V|)$, so the complexity of the algorithm is $O(|V|^3)$, but we found it practical to think in terms of number of bitwise operations, as their cost if often near-constant time, at least for the graph sizes we dealt with.

We tried a few different heuristics for the ordering o of the vertices, including the inverse of the degeneracy order (Lick & White, 1970), which tends to produce large cliques (Eppstein et al., 2013; Jiang et al., 2017). We found that it works best to sort the vertices in order of decreasing bag size.

If the lower bound meets or exceeds the upper bound k, the propagator reports a conflict. We construct a clausal conflict as follows: each vertex v of the current graph is the result of the contraction of one or more vertices of the original graph. In keeping with the notation for the transitivity propagator, we call this the *bag* $b(v)$. We arbitrarily pick one vertex $r(v)$ from the bag of each vertex v in the largest clique C, and set the explanation to

$$\bigvee_{v,u \in C} e_{r(v)r(u)} \tag{7}$$

This clause contains $O(|C|^2)$ positive literals. When we falsify it, it means that the corresponding contracted graph contains edges between all pairs of vertices in C. Therefore, falsifying the clause entails violating the upper bound, so it is logically entailed.

This explanation is not unique. In particular, we can generate mixed-sign clauses as explanations, which are also shorter.

Example 3. *Suppose that under the current partial assignment to X, the graph G_X contains the clique $C = \{v, u_1, u_2, w_1, w_2\}$ such that v corresponds to the bag $\{u, w\}$ and $|C| \geq \chi^{up}$. Further suppose that $N_G(u) = \{u_1, u_2\}$ and $N_G(w) = \{w_1, w_2\}$. To generate an explanation for this according to (7) we arbitrarily pick $r(v) = u$ and the clause that we generate will be of the form $c \cup \{e_{uw_1}, e_{uw_2}\}$, where c contains the literals implied by (7) which involve only the vertices $\{u_1, u_2, w_1, w_2\}$. It also implicitly includes the literals $\{e_{uu_1}, e_{uu_2}\}$, but since the edges (u, u_1) and (u, u_2) are present in G, it is as if there exist unit clauses $(\bar{e}_{uu_1})$ and $(\bar{e}_{uu_2})$, so they can be resolved away from the explanation clause. However, this clause is not the only possible explanation. The clause $c \cup \{\bar{e}_{uw}\}$ is also a valid explanation. Indeed, falsifying it means that we make e_{uw} true, which means that we contract u and w and that the resulting vertex has neighborhood $\{u_1, u_2, w_1, w_2\}$. Moreover, this clause that contains both positive and negative literals is shorter than the clause generated by (7) by 1 literal, because the single contraction entails 4 edge additions $((u, w_1), (u, w_2), (w, u_1), (w, u_2))$, two of which $((u, w_1), (u, w_2))$ are needed for the explanation.*

This example can be extended to show that in some cases it is possible to generate an explanation of length $O(|C|)$ as opposed to $O(|C|^2)$. We have experimented with producing such mixed-sign explanations. However, it is computationally costly to find which choice of representative from each bag generates the shortest clause. Moreover, although mixed-sign explanations that we tried tend to be much shorter and speed up search in terms of number of conflicts per second, they also significantly increase the overall effort required, both in number of conflicts and runtime. Therefore, we use exclusively the positive clauses given by equation (7).

Incremental clique updates. The heuristic method we described above for finding cliques has the disadvantage that it examines the entire (contracted) graph each time that the propagator is invoked. Depending on the instance, this can end up a significant overhead. Therefore, we have also implemented a different method which sacrifices the quality of the bound to achieve significantly better performance per invocation.

This algorithm, which we call `IncrementalCliqueFinder`, proceeds as follows. When invoked at the root of the tree, it only forwards to `GreedyCliqueFinder` and then discards all cliques except those that have largest cardinality. For subsequent invocations, it requires access to a list of vertices which have gained edges as a result of contraction or separation of vertices in the graph. When we contract u and v as a result of e_{uv} becoming true, both vertices may gain new edges, as their neighborhoods are updated to all be identical $N(v) \cup N(u)$. Similarly, when we separate two vertices which were previously non-neighbors as a result of e_{vu} becoming false, both u and v gain an edge in their neighborhood. The neighborhood of all such vertices may now contain one of the cliques which we have stored (but when we contract two vertices v, u, only $r(v) = r(u)$ is examined). This list can be easily kept up to date as the propagator is woken for every literal that becomes true or false. Given this list, `IncrementalCliqueFinder` examines each vertex v in it and adds v to all stored cliques C such that $N(v) \supseteq C$. At the end, it again filters the set of stored cliques to keep only those with largest cardinality. `IncrementalCliqueFinder` returns this largest cardinality as the lower bound.

One complication is that on backtracking, the set of stored subgraphs are no longer cliques, as vertices are uncontracted and edges removed, hence they are invalid.

Rather than implement a trailing or copying scheme for these cliques, after a conflict `IncrementalCliqueFinder` again simply forwards to `GreedyCliqueFinder`, as it does at the root.

The advantage of `IncrementalCliqueFinder`, compared to `GreedyCliqueFinder`, is that it examines only the part of the graph that has been modified since the last invocation, and that only in relation to a small set of cliques. More concretely, if the neighborhood of p vertices has changed and it has stored q cliques, it performs $O(pq)$ bitwise operations to update the set of cliques and get a new bound, so its complexity is $O(pq|V|)$. Empirically, q is small (less than 5), so even if nearly all vertices have had their neighborhood changed, it is faster than `GreedyCliqueFinder` by a factor of $|V|$ and even faster in the more typical case where only a small subset of vertices have changed since the last invocation.

Despite the advantage of `IncrementalCliqueFinder` over `GreedyCliqueFinder` in terms of runtime complexity, we have discovered empirically that the cliques that cause bound violations may arise from regions of the graph that initially do not contain large cliques and will therefore be ignored by `IncrementalCliqueFinder`. This is somewhat mitigated by the fact that we store all cliques of largest cardinality and that we call `GreedyCliqueFinder` after every conflict. However, as we show in Section 7, using the `IncrementalCliqueFinder` algorithm in our solver degrades overall performance, despite the fact that it improves speed in terms of number of conflicts per second.

The empirical observation that finding diverse sets of cliques is important for performance is also informative in light of the correspondence between nodes of the Zykov tree and colorings of the graph, discussed in Section 4. Since a) finding diverse sets of cliques is important for performance, b) maintaining a single coloring is analogous to maintaining a single clique, and c) using color variables forces us to maintain a single coloring, we conclude that using color variables cannot replicate the results of using edge variables.

5.2 Mycielski-Based Bound

Although being a useful bound in practice, the clique number is both hard to compute and gives no guarantees on the quality of the bound. We propose here a new lower bound inspired by *Mycielskian graph*.

Definition 1 (Mycielskian graph (Mycielski, 1955)). *The Mycielskian graph $\mu(G)=(\mu(V),\mu(E))$ of $G =(V,E)$ is defined as follows:*

- *$\mu(V)$ contains every vertex in V, and $|V| + 1$ additional vertices, the vertices $U = \{u_i \mid v_i \in V\}$ and another distinct vertex w.*

- *For every edge $v_iv_j \in E$, $\mu(E)$ contains v_iv_j, v_iu_j and u_iv_j. Moreover, it contains all the edges between U and w.*

The Mycielskian $\mu(G)$ of a graph G has the same clique number, however its chromatic number is $\chi(G) + 1$. Indeed, consider a coloring of $\mu(G)$. For any vertex $v_i \in V$, we have $N(v_i) \subseteq N(u_i)$, and therefore we can safely use the same color v_i as for u_i. If can be shown that at least $\chi(G)$ colors are required for the vertices in U, and since $N(w) = U$, then w requires a $\chi(G) + 1$-th color. Mycielski introduced these graphs to demonstrate that triangle-free graphs can have arbitrarily large chromatic numbers, hence the clique number does not approximate the chromatic number.
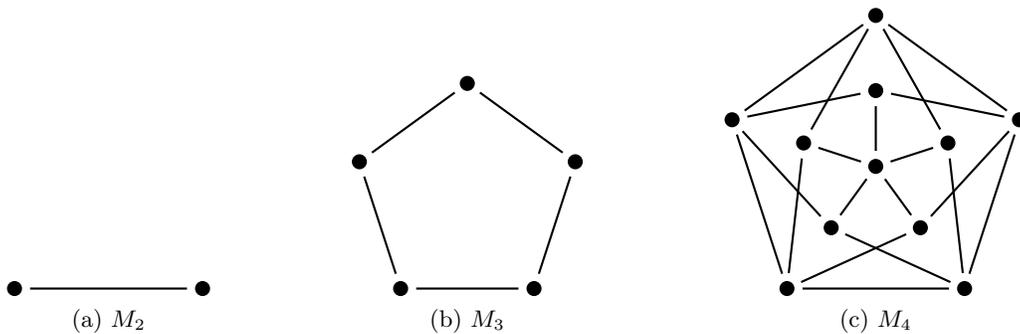
Figure 2: A 2-clique $M_2 = \mu(\emptyset)$, its Mycielskians $M_3 = \mu(M_2)$ and $M_4 = \mu(M_3)$

The principle of our bound is a greedy procedure that can discover embedded pseudo Mycielskians. Indeed, the class of embedded graphs that we look for is significantly broader than the set of pure Mycielskians $\{M_2, M_3, M_4, \ldots\}$. First, the witness subgraph may not necessarily be induced. Therefore, trivially, Mycielskians with extra edges also provide valid lower bounds. Moreover, rather than starting from a single edge, as in the construction of the pure Mycielskians, we construct Mycielskians of cliques. Finally, the method we propose can also find Mycielskians modulo some vertex contractions. Clearly, those are also valid lower bounds since contracting vertices is equivalent to adding equality constraints to the problem.

Given a subgraph $H = (V_H, E_H)$ of G, and $v \in V_H$, we define:

$$S_v = \{u \mid N_H(v) \subseteq N_G(u)\} \tag{8}$$

Suppose that there exists a vertex w with at least one neighbor in every set S_v:

$$w \in \cap_{v \in V_H} N_G(S_v) \tag{9}$$

and let $u(v)$ be any element of S_v such that $u(v) \in N_G(w)$ and $U = \{u(v) \mid v \in V\}$, then:

Lemma 1.

$$H' = (V \cup U \cup \{w\}, E \cup \bigcup_{v \in V} N_H(v) \times u(v) \cup \bigcup_{u \in U} \{(uw)\}) \implies \chi(H') \geq \chi(H) + 1$$

Proof. The proof follows from the facts that H' is the Mycielskian graph of H possibly with contracted vertices, and is embedded in G.

Suppose first that, for each $v \in V$, $u(v) \neq v$ and $w \notin V$. Then we have $H' = \mu(H)$ by using $u(v_i)$ for the vertex u_i, and w for itself, in Definition 1.

Suppose now that $H' \neq \mu(H)$. This can only be because either:

- For some vertex v_i of H, we have $u(v_i) = v_i$. In this case, consider the graph $\mu(H)$ and contract u_i and v_i. The resulting graph $\mu(H)/(u_i v_i)$ has a chromatic number at least as high as $\mu(H)$. However, it is isomorphic to H'.

- The vertex w is the vertex v_i from the original subgraph H. Here again contracting v_i and w in $\mu(H)$ yields H'.

Notice that there is not a third case where w is taken among U since, for any $v \in V_H$, we have $u(v) \notin \cap_{v \in V_H} N_G(S_v)$ because $u(v)$ is not a neighbor of itself.

H' is a subgraph of G since the edges added to H' are all edges of G □

Example 4. *Figure 3a shows the graph $G/(cd)$ obtained by contracting vertices c and d in the graph G of Figure 1. Let H be the clique $\{a, b, c\}$. We have $S_a = \{a, e\}$, $S_b = \{b, f\}$ and $S_c = \{c\}$. Furthermore, $N_G(\{a, e\}) \cap N_G(\{b, f\}) \cap N_G(\{c\}) = \{b, c, g\} \cap \{a, e, c, g\} \cap \{a, b, e, g, f\} = \{g\}$, from which we can conclude that this graph has chromatic number at least 4. As shown in Figure 3b, when called with $H = \{a, b, c\}$ Algorithm 1 will extend it with a first layer $U = \{e, c, f\}$ and an extra vertex $w = g$. Notice that the graph obtained by adding the edge (cd) has a 4-clique (see Figure 1). Therefore, the graph G in Figure 1a also has a chromatic number of at least 4.*

(a) $H = G/(cd)$ (b) Trace of Algorithm 1 on H
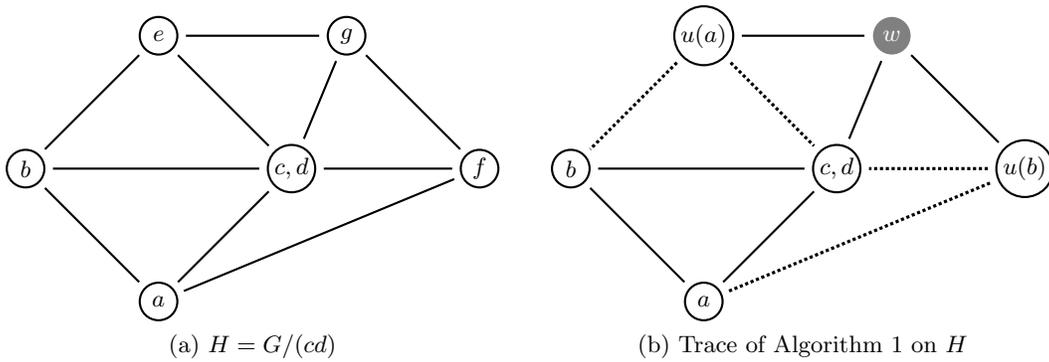
Figure 3: Embedded Mycielski

Algorithm 1 greedily extends a subgraph $H = (V_H, E_H)$ of the graph G (with $\chi(H) \geq k$) into a larger subgraph $H' = (V'_H, E'_H)$, following the above principles. As long as this succeeds, in the outermost loop, we replace H by H' and iterate. The computed bound k is equal to $\chi(H)$ plus the number of successful iterations.

We compute the sets S_v (Equation 8) and the set W of nodes with at least one neighbor in every S_v in Loop 1. Then, if it is possible to extend H (Line 4), we compute the pseudo Mycielskian (V'_H, E'_H) as shown in Lemma 1 and replaces H with it in Line 5 before starting another iteration.

Complexity. An iteration of Algorithm 1 does $O(|V_H| \times |V|)$ bitset operations (Line 2 is 1 'AND' operation and Line 3 is $O(|V|)$ 'OR' operations and 1 'AND'). The second part of the loop, starting at Line 4, runs in $O(|V_H|^2)$ time. Typically, the number of iterations is very small. It cannot be larger than $\log |V|$ since the number of vertices in H is (more than) doubled at each iteration. It follows that Loop 1 is executed at most $2|V|$ times, and therefore, the worst case time complexity is $O(|V|^2)$ bitset operations (hence $O(|V|^3)$ time).

Algorithm 1: MYCIELSKIBOUND($k, H = (V_H, E_H), G = (V, E)$)

---

    while $|V_H| < |V|$ **do**

        $W \leftarrow V$;

        $\forall v \in V_H \; S_v \leftarrow \{v\}$;

1   **foreach** $v \in V_H$ **do**

            foreach $u \in V$ **do**

2   **if** $N_H(v) \subseteq N_G(u)$ **then** $S_v \leftarrow S_v \cup \{u\}$;

3 $W \leftarrow W \cap N_G(S_v)$;

4   **if** $W \neq \emptyset$ **then**

            $k \leftarrow k + 1$;

            $(V_H', E_H') \leftarrow (V_H, E_H)$;

            $w \leftarrow$ any element of W;

            foreach $v \in V_H$ **do**

                $V_H' \leftarrow V_H' \cup \{$ any element of $(N_G(w) \cap S_v)\}$;

                $E_H' \leftarrow E_H' \cup \{(wu)\} \cup \{u \times N_H(v)\}$;

5 $(V_H, E_H) \leftarrow (V_H', E_H')$;

        else break;

        return k;

Explanation. Similarly to the clique based lower bound, the explanations that we produce here correspond to the set of all edges in the graph H:

$$\bigvee_{(uv) \in E_H} e_{uv} \tag{10}$$

Adaptive application of the Mycielskian bound. In preliminary experiments, we found that trying to find a Mycielskian subgraph in every node of the search tree was too expensive and did not pay off in terms of total runtime. Therefore, we adapted a heuristic proposed by Stergiou (2008) which allows us to apply this stronger reasoning less often. In particular, we only compute the clique lower bound by default. But every time there is a conflict, whether by unit propagation or by bound computation, we compute the Mycielskian lower bound in the next node. If that causes a conflict, we keep computing this bound until we backtrack to a point where even the stronger bound does not detect a bound violation. This has the effect that we compute the cheaper clique lower bound most of the time, but learn clauses based on the stronger bound. This combination of stronger clause learning with faster propagation measurably outperforms the baseline of computing a bound based only on cliques, as we show in Section 8.

6. Preprocessing and Inference

In addition to the lower bound, we implemented three additional inference mechanisms: two preprocessing techniques and a method to extend the lower bounds we propose to marginal-

cost pruning. The former preprocessing is well-known, however, the latter is novel, and is made possible by the hybrid SAT/CP architecture of our approach.

6.1 Peeling

The so-called *peeling* procedure is an efficient scale reduction technique introduced by Abello et al. (1999) for the maximum clique problem. Since vertices of $(k+1)$-cliques have each at least k neighbors, one can ignore vertices of degree $k-1$ or less. As observed by Verma et al. (2015), this procedure corresponds to restricting search to the maximum χ^{low}-core of G where χ^{low} is some lower bound on $\omega(G)$:

Definition 2 (k-Core and degeneracy). *A subset of vertices S is a k-core of the the graph G if the minimum degree of any vertex in the subgraph of G induced by S is k. The maximum value of k for which G has a non-empty k-core is called the* degeneracy *of G.*

Verma et al. noted that the peeling technique can also be used for graph coloring, since low-degree vertices can be colored greedily.

Theorem 1 (Verma *et al.* 2015). *G is k-colorable if and only if the maximum k-core of G is k-colorable.*

Indeed, starting from a k-coloring of the maximum k-core of G, one can explore the vertices of G that do not belong to the core and add them back in an order (the inverse of the *degeneracy ordering*) such that any vertex is preceded by at most $k-1$ of its neighbors. It follows that these extra vertices can each be colored without introducing a $k+1$-th color.

Lin et al. (2017) recently proposed a similar reduction rule for graph coloring instances, which allowed them to reduce the size of large, sparse graphs.

Proposition 1 (Lin et al., 2017). *Let G be a graph with $\chi(G) \geq k$ and let I be an independent set of G such that for all $v \in I$, $d(v) < k$. Then, $k-1 \leq \chi(G \setminus I) \leq \chi(G)$ and if $\chi(G \setminus I) = k-1$ then $\chi(G) = k$.*

However, applying the rule of proposition 1 actually computes the maximum χ^{low}-core, hence we used a linear-time algorithm to compute the degeneracy order instead.

Besides the obvious advantage of trimming the graph this reduction might also improve the lower bound found by a heuristic maximal clique algorithm. The reason is that whatever heuristic we use for finding a maximal clique may make a suboptimal choice and this preprocessing step removes some obviously suboptimal choices from consideration.

We have used the following preprocessing: first we compute a lower bound χ^{low}, then we remove vertices that do not belong to the maximum χ^{low}-core. However, we observed very little benefit in our instance set, which comprises smaller and denser graphs than the one that Lin et al. used. We also used it, however, as initial ordering for the greedy clique-finding algorithm.

6.2 Independent Set Extraction

It has recently been shown (Jansen & Pieterse, 2018) that graph coloring is fixed parameter tractable with parameter $k = q + s$ where q is the size of the coloring the graph admits and

s is the cardinality of the minimum vertex cover. The idea of that algorithm is based on the observation that, similar to peeling, for any vertex v of G, a $q-$coloring of the graph induced by $V \setminus \{v\}$ (that we denote $G \mid_{V \setminus \{v\}}$) can be extended to a $q-$coloring of G, if at most $q - 1$ colors are used to color the vertices $N(v)$. Then, v is simply colored with any color not used in its neighborhood. This reasoning can be extended to any independent set, since being an independent set ensures that the coloring of the residual graph can be extended to all vertices in the independent set without interfering with each other. If G admits a vertex cover S of size s, then $V \setminus S$ is an independent set of G and so G admits a $q-$coloring if and only if $G \mid_S$ admits a $q-$coloring such that for every $v \in V \setminus S$, the number of colors in $N(v)$ is at most $q - 1$, i.e., a coloring must respect the *local independent set constraints*:

$$\chi(N(v)) \leq q - 1 \qquad \forall v \in S$$

We can test all possible s^q $q-$partitions of $G \mid_S$ to determine whether there exists one that is a coloring and satisfies the above constraints. This gives an algorithm with time complexity $O(s^q p(|V|))$ for a polynomial p, proving that the problem is FPT.

We can exploit the same observation in our solver, noting that during branch-and-bound, if the incumbent uses k colors, we set $q = k - 1$ for the constraints required by the FPT algorithm. We heuristically find an independent set after peeling, which makes it slightly stronger than what is proposed by Jansen and Pieterse. Indeed, the vertices removed during peeling do not need to be independent from other vertices that we remove, allowing for greater reduction.

During search, we enforce the constraints by checking whether any of the cliques we have stored are contained in $N(v)$ for any removed v. If so, we trigger a conflict as if the upper bound had been violated. There are two complications here: first, since we contract vertices during search, the neighbors of v may not be present in the current graph. Hence, as we descend a branch, we remap constraints so that each vertex v is replaced by its representative $r(v)$, so that the constraint $\chi(N(v)) \leq q - 1$ becomes $\chi(\{r(u) \mid u \in N(v)\}) \leq q - 1$. Second, explanations for a global bound violation can be generated by using an arbitrary vertex from each bag that participated in the subgraph that violates the bound. In the case of local independent set constraints, this is not true. Consider for example the case where $q = 4$, we have the bags $\{v_1, u_1\}$, $\{v_2, u_2\}$ and $\{v_3, u_3\}$ with $r(u_i) = r(v_i) = v_i$ for i in $\{1, 2, 3\}$ in a 3-clique and we have the local constraint $\chi(\{u_1, u_2, u_3\}) \leq 3$. In this configuration, the local constraint gets rewritten to $\chi(\{v_1, v_2, v_3\}) \leq 3$. Even though the global upper bound is not violated, the clique $\{v_1, v_2, v_3\}$ is contained in subgraph involved in the local constraint, hence the constraint is violated and we can backtrack. However, the explanation procedure will generate a clause explaining the edges between the vertices v_i, $i \in \{1, 2, 3\}$, but this clause is not a valid explanation for the violation of the local IS constraint. Indeed, if we simply add all the edges between the vertices v_i, the local constraint is not rewritten and not violated. Instead, we have to explicitly instruct the explanation procedure to pick representatives that are mentioned in the original local constraint. This is sufficient to produce valid explanations for local constraint violations.

To our surprise, using this technique resulted in worse performance overall, therefore it is not used by default in our solver. Part of the reason for this is the cost of checking

the local constraints, but the main issue was that the problem reduction did not result in a sufficient reduction of search effort. We are optimistic that this remains a promising direction.

6.3 Clique-Based Pruning

Since maximal cliques and embedded Mycielskian graphs provide non-trivial lower bounds, it is natural to prune the domain of variables based on the marginal cost of the values with respect to the bound.

In our case, however, the marginal cost cannot be greater than 1. Indeed, setting a variable e_{uv} to true corresponds to contracting the vertices u and v, whilst setting it to false corresponds to adding the edge (uv). Now, let u and v be two vertices of a graph $G = (V, E)$ with $(uv) \notin E$. From a $\chi(G)$-coloring of G we can derive a $\chi(G) + 1$-coloring of $G + (uv)$ or $G/(uv)$ by simply using a fresh color (e.g. $\chi(G) + 1$) for u, since all the new edges contain u.

Therefore, we can hope to get some pruning only when the bound is tight, that is, when we find a subgraph with chromatic number $\chi^{up} - 1$ with χ^{up} the current best upper bound. In this case, we can use the two following lemmas (whose proofs are trivial).

Lemma 2. *If the subgraph $H = (V', E')$ of $G = (V, E)$ has chromatic number k and there exists a vertex $v \in V \setminus V'$ such that $V' \setminus N(v) = \{u\}$ then $\chi(G + (uv)) = k + 1$.*

Lemma 3. *If the subgraph $H = (V', E')$ of $G = (V, E)$ has chromatic number k and there exist two vertices u and v such that $V' \subseteq (N(u) \cup N(v))$ then $\chi(G/(uv)) = k + 1$.*

From Lemma 2 and 3, we can design two pruning algorithms. Lemma 2 requires to count, for each subgraph $H = (V', E')$ witnessing for the lower bound (i.e., such that $|V'| = \chi^{low}$), the number of neighbors within V' of every vertex in $V \setminus V'$. This can be done in $O(|E|)$ time for every witness subgraph. There are at most $O(|V|)$ of them, although only the subgraphs of size $\chi^{up} - 1$ need to be checked and there are much fewer of them. In particular, on hard instances the gap between lower and upper bound is large even relatively deep in the search tree and therefore this pruning technique can be skipped altogether.

Moreover, if we incrementally maintain the cliques used as lower bound, as described in Section 5.1, then this complexity can be amortized down a branch of the search tree. We can maintain, for every vertex v and every maximal clique i, the number μ_v^i of neighbors of that vertex in that clique. When a new edge (uv) is added, the stored values μ_v^i, for v (u) and for every clique i that u (v) belongs to, are incremented. The contraction of two vertices u and v is treated similarly, as it corresponds to adding several edges.

Lemma 3, however, requires a similar count for each *pair* of vertices without edge in $V \setminus V'$ hence an extra linear factor to the time complexity. On the one hand, our experimental results (see Section 7.2) show that looking for implied contractions (Lemma 2) is sufficiently cheap in practice to have a positive impact on the overall method. On the other hand, looking for implied edge additions is too expensive.

Under some fair assumptions, it is possible to show that our approach including the clique-based lower bound and the pruning technique described above (denoted `gc-cdcl`) is strictly stronger than the standard color variable-based model (denoted `Dsatur`).

As observed in Section 4.4, there is a surjective (but non-injective) function from the partial coloring of a graph G to the graphs obtained by adding edges and contracting vertices in G. Furthermore, there is a similar function from *domains* of color variables to contraction-addition graphs of G. It is important to distinguish between partial colorings and domains, because when baktracking on the decision to assign the color i to vertex v, a domain can represent the fact that color i is now forbidden for v, even if no neighbor of v is currently colored with i.

Given a domain $\mathbf{D} : V \mapsto 2^{\{1,...,k\}}$ of $G = (V, E)$, let the *Zykov graph* of $(G, \mathbf{D})$ be the graph obtained by contracting the vertices $\{v \mid \mathbf{D}(v) = \{i\}\}$ for $1 \le i \le k$ (the representative vertex is denoted v_i) and adding an edge between every pair of vertices v, u in the contracted graph such that $\mathbf{D}(v) \cap \mathbf{D}(u) = \emptyset$. We make the following assumptions:

1. Both methods make the same decisions. The same mapping applies: $u = i$ for `Dsatur` can be emulated by $u = v_i$ for `gc-cdcl` if there is a vertex v_i already mapped to i, or by the set of decisions $\{u \ne v_j \mid j < i\}$ if i is a fresh color. Moreover, `Dsatur` branching on $u \ne i$ can be emulated by `gc-cdcl` branching on $u \ne v_i$.

2. `Dsatur` model uses only binary inequalities.

3. `gc-cdcl` finds, in the Zykov graph of $(G, \mathbf{D})$, the k-clique corresponding to the set of representatives obtained from singleton domains: $\{v_i \mid \mathbf{D}(u) = \{i\}, u \in V\}$.

Theorem 2. *Under the three assumptions above, the search tree explored by* `Dsatur` *is not smaller than the search tree explored by* `gc-cdcl`.

Proof. Let $\mathbf{D}$ be a domain on the color variables of a graph $G = (V, E)$. Let $AC_{\texttt{Dsatur}}(G, \mathbf{D})$ be the domain consistent fixpoint of the domain $\mathbf{D}$ in the model used by `Dsatur`. Similarly, let $AC_{\texttt{gc-cdcl}}(G, \mathbf{D})$ be the graph corresponding to the domain consistent fixpoint of the Zykov graph of $(G, \mathbf{D})$ in the model used by `gc-cdcl`. We say that a graph H is *as tight as* a graph G if and only if H can be obtained by contracting vertices and adding edges in G.

Since we assume the same decisions, we prove the theorem by showing that given equivalent search states (i.e., respectively $(G, \mathbf{D})$ for `Dsatur` and the Zykov graph H of $(G, \mathbf{D})$ for `gc-cdcl`), the domain consistent fixpoint reached by `gc-cdcl` is strictly tighter than that reached by `Dsatur`. The fixpoint reached by `Dsatur` is defined by the domain $AC_{\texttt{Dsatur}}(G, \mathbf{D})$. We first show that the graph $AC_{\texttt{gc-cdcl}}(G, \mathbf{D})$ is at least as tight as the Zykov graph H' of $(G, AC_{\texttt{Dsatur}}(G, \mathbf{D}))$. Notice that failures in `Dsatur` may only happen when the total number of colors (distinct singleton domains in $\mathbf{D}$) is higher than or equal to the upper bound. Since we assume that `gc-cdcl` finds that clique, the same failure would happen in `gc-cdcl`, hence we consider non-empty arc-consistent fixpoints from now on.

Consider first an edge (uv) of H' that is not in H, that is, implied by domain consistency on `Dsatur`'s model. This edge is implied by the fact that the domains of the variables x_u and x_v are disjoint in the domain consistent fixpoint. However, a color i is removed from the domain of a variable x_u only if there is w a neighbor of u in G such that $\mathbf{D}(w) = \{i\}$ or if $x_u \ne i$ is the result of a previous branch $x_u \ne i$. In the latter case, the same branching in `gc-cdcl` has added an edge between u and a vertex w mapped to color i. Therefore, this new edge implies that the union of the neighborhoods of u and v include, for every possible

color, at least one vertex already mapped to that color. In this case the pruning rule from Lemma 3 would trigger and add the edge (uv) in $AC_{\texttt{gc-cdcl}}(G, \mathbf{D})$.

Now consider that the vertex v in H' is the contraction of v and u in H, i.e., $\mathbf{D}(x_v) = \mathbf{D}(x_v) = \{i\}$ for some color i. The same observation can be made as in the previous case, and thus we know that one of these vertices (say v w.l.o.g.) has at least one neighbor whose domain is the singleton $\{j\}$ for every possible color $j \neq i$. However, in this case Lemma 2 would trigger and u and v will be contracted in $AC_{\texttt{gc-cdcl}}(G, \mathbf{D})$.

To prove strictness, we simply need an example where $\texttt{gc-cdcl}$'s search tree is smaller. The easiest example is to use a Mycielskian graph, such as the one in Figure 2b, which $\texttt{gc-cdcl}$ will find at the root node whereas $\texttt{Dsatur}$ will be forced to branch. □

7. Experimental Evaluation

We implemented the techniques described in previous sections into a graph coloring solver called $\texttt{gc-cdcl}$. It runs in either of the two configurations described in section 4.1: branch-and-bound, denoted $\texttt{gc-cdcl}$, or bottom-up, denoted $\texttt{gc-cdcl}\uparrow$. We implemented $\texttt{gc-cdcl}$ with MINICSP[4] as the underlying CDCL CSP solver.[5] In both cases, we use adaptive application of the Mycielskian bound, as explained in Section 5.2. When computing the Mycielskian bound, we apply Algorithm 1 on all of the maximal cliques, and keep the best outcome. Moreover, both default approaches use the pruning technique described in Section 6.3 limited to Lemma 2 and the DSATUR-emulating heuristic described in Section 4.4. However, they do not use the independent set extraction technique described in Section 6.2, nor the incremental lower bound algorithm described in Section 5.1 since the results are slightly worse overall. Even though the results are extremely close, it does not use the transitivity-aware unit propagation described in Section 4.3 either. All these techniques are tested in Section 7.2.

We experimented primarily on the $\texttt{DIMACS}$ data set, composed of 125 instances[6] from Trick's graph coloring webpage and described in the proceedings of the COLOR02 workshop (Trick, 2002). In the subsequent tables, however, we omit 26 of these instances that were solved by every method to optimality. Further experiments were performed on a data set (denoted $\texttt{RCBII}$) of randomly generated graphs introduced in (Segundo, 2012).

All experiments were run on 7 cluster nodes, each with 36 Intel Xeon CPU E5-2695 v4 2.10GHz cores running Linux Ubuntu 16.04.4. Sources were compiled using g++8. Every algorithm was run ten times with the same set of distinct random seeds and a time limit of one hour and a memory limit of 3.5GB on every instance in the data sets.

7.1 Comparison with the State of the Art

We first compare with the state-of-art SAT-based solver $\texttt{Color6}$ (Zhou et al., 2014), a very efficient clause-learning algorithm for graph coloring. Similarly to our approach, it is based on a SAT solver, however, it uses the color-based formulation. It was shown to outperform the state of the art on many instances. As $\texttt{Color6}$ solves satisfiability instances

---

4. Sources available at: $\texttt{https://bitbucket.org/gkatsi/minicsp}$.
5. Sources available at: $\texttt{https://bitbucket.org/gkatsi/gc-cdcl/src/master/}$.
6. Available here: $\texttt{http://homepages.laas.fr/ehebrard/2ndDimacs.tgz}$

	#	gc-cdcl			Color6			Dsatur			Segundo			CPLEX		
		Opt.	χ^{up}	χ^{low}	Opt.	χ^{up}	χ^{low}	Opt.	χ^{up}	χ^{low}	Opt.	χ^{up}	χ^{low}	Opt.	χ^{up}	χ^{low}
DSJ	14	0.07	**76.05**	**32.50**	0.07	78.01	28.93	0.00	78.50	3.07	0.00	78.57	3.07	0.07	85.91	29.90
FullIns	14	**1.00**	6.79	**6.79**	0.21	6.79	5.14	0.00	6.79	4.86	0.00	6.79	4.86	0.81	6.90	6.38
Insertions	11	0.27	5.18	2.55	0.36	5.18	2.82	0.00	5.18	2.00	0.00	5.18	2.00	0.36	5.18	**3.73**
abb313GPIA	1	0.00	10.00	8.00	0.00	10.00	8.00	0.00	12.00	8.00	0.00	10.00	8.00	0.00	14.00	8.00
ash	3	1.00	4.00	4.00	1.00	4.00	4.00	0.00	5.00	3.00	0.00	4.37	3.00	1.00	4.00	4.00
flat	6	0.00	**73.92**	**12.50**	0.00	75.02	10.67	0.00	75.83	5.67	0.00	75.32	5.67	0.00	82.17	10.67
fpsol2	1	1.00	65.00	65.00	0.00	65.00	59.00	1.00	65.00	65.00	1.00	65.00	65.00	1.00	65.00	65.00
inithx	1	1.00	54.00	54.00	0.50	54.00	48.50	1.00	54.00	54.00	1.00	54.00	54.00	1.00	54.00	54.00
latin	1	0.00	**118.20**	90.00	0.00	132.50	90.00	0.00	130.00	0.00	0.00	128.40	0.00	0.00	159.00	90.00
le450	10	**0.59**	14.83	13.00	0.15	15.80	13.00	0.24	15.80	13.00	0.25	15.86	13.00	0.38	18.42	13.00
miles1000	1	1.00	42.00	42.00	0.00	42.00	40.00	1.00	42.00	42.00	1.00	42.00	42.00	1.00	42.00	42.00
mug	4	1.00	4.00	4.00	1.00	4.00	4.00	0.00	4.00	3.00	0.00	4.00	3.00	1.00	4.00	4.00
myciel	5	**1.00**	6.00	**6.00**	0.80	6.00	4.80	0.00	6.00	2.00	0.00	6.00	2.00	0.60	6.00	5.08
qg	4	**0.75**	**58.00**	57.50	0.28	63.38	57.50	0.25	62.00	57.50	0.40	58.47	57.50	0.35	70.78	57.50
queen	12	0.42	12.63	**11.33**	0.17	13.20	11.18	0.17	12.75	11.08	0.17	**12.57**	11.08	0.40	12.91	11.32
school1	2	1.00	14.00	14.00	0.50	20.00	14.00	0.00	19.00	0.00	0.00	14.00	0.00	1.00	14.00	14.00
wap0	8	0.12	**46.01**	41.38	0.00	48.08	40.00	0.12	48.12	41.38	0.12	48.24	41.38	0.00	51.12	40.00
will199GPIA	1	1.00	7.00	7.00	0.00	7.00	6.00	0.00	7.00	6.00	0.00	7.00	6.00	1.00	7.00	7.00

Table 1: Comparison with top-down methods – breakdown by classes of DIMACS instances

only (testing whether a coloring with a specific number of colors exists), we implemented a branch-and-bound wrapper on top of it, denoted Color6, as well as well as a wrapper that implements the bottom-up strategy, denoted Color6↑. We use the lower and upper bounds computed by our approach (respectively the maximal clique algorithm described in Section 5.1 and a greedy run of DSATUR) as initial bounds for Color6 and Color6↑. Moreover, we also compare with two implementations of DSATUR-based branch-and-bound by Furini et al., one denoted Dsatur and using Brélaz' original heuristic, and one denoted Segundo using an improved version (Segundo, 2012). Finally, we compare with an integer programming formulation in CPLEX. The model we use for CPLEX is the trivial one using binary color variables (one for each vertex and each color), and one binary inequality per edge. However, observe that CPLEX actually computes maximal cliques in its preprocessing, so providing it with clique inequalities is useless. Moreover, we initialize the upper bound with the same method as for Color6, and also arbitrarily fix the colors of one maximal clique in order to break symmetries. The branching selection of Color6, Dsatur and Segundo was randomized by replacing the lexicographical tie breaking by a uniform random choice so that we could make several runs on every instance.

Unfortunately, we could not compare our method to the method of Schaafsma et al. (Minicolor) directly. Indeed, its implementation, provided by the authors, is difficult to use in the type of extensive experiments of the type we performed. Firstly, the algorithm is restricted to instances with at most 32 colors. Secondly, it solves the satisfiability problem $\chi(G) \leq K$ and uses a file converter. Finally, the changes made to MiniSat's code do not seem to be robust and we experienced several occurrences of assertion failures.

The results in Tables 1 and 2 correspond to 10 random runs on every instance of the DIMACS data set. They are averaged over runs and over instances from the same class,

	#	gc-cdcl↑			Color6↑		
		Opt.	χ^{up}	χ^{low}	Opt.	χ^{up}	χ^{low}
DSJ	14	0.07	**78.69**	33.99	0.07	86.93	**35.88**
FullIns	14	**1.00**	**6.79**	**6.79**	0.21	7.29	5.43
Insertions	11	0.27	**5.18**	3.82	**0.36**	5.36	**4.09**
abb313GPIA	1	0.00	**11.00**	9.00	0.00	14.00	9.00
ash	3	1.00	4.00	4.00	1.00	4.00	4.00
flat	6	0.00	**77.52**	14.00	0.00	82.17	**16.90**
fpsol2	1	**1.00**	65.00	**65.00**	0.00	65.00	59.00
inithx	1	**1.00**	54.00	**54.00**	0.00	54.00	43.50
latin	1	0.00	**126.90**	90.00	0.00	159.00	90.00
le450	10	**0.60**	**15.20**	13.00	0.28	19.50	13.00
miles1000	1	**1.00**	**42.00**	**42.00**	0.00	45.00	40.00
mug	4	1.00	4.00	4.00	1.00	4.00	4.00
myciel	5	**1.00**	6.00	**6.00**	0.80	6.00	5.60
qg	4	**0.75**	**58.00**	57.50	0.38	70.75	57.50
queen	12	**0.42**	**13.36**	11.33	0.21	15.98	11.33
school1	2	1.00	15.50	**15.50**	1.00	**14.00**	14.00
wap0	8	**0.12**	**46.12**	**41.38**	0.00	51.12	40.00
will199GPIA	1	**1.00**	**7.00**	**7.00**	0.00	10.00	6.00

Table 2: Comparison with bottom-up methods – breakdown by classes of DIMACS instances

whose cardinality is given in the first column (denoted "#"). The same results on individual instances (averaged over the 10 runs) are shown in Appendix A.

We show the ratio of instances for which a proof of optimality was found ('Opt.'), as well as the average upper bound (χ^{up}) and lower bound (χ^{low}), for every method. The best results for each criterion are highlighted using colors, and bold font when the result is not matched by any other method in the table. Table 1 focuses on top-down methods. gc-cdcl is better on all but two classes of instances: Insertions and queen. Moreover, it finds the same coloring as the other methods in the Insertions class for which the CPLEX model is best, and computes strictly more proofs of optimality than other solvers in the queen classes, although Segundo finds better colorings. Finally, on 8 classes it is strictly better than the second best solver (considering at least one criterion).

Table 2 focuses on the two bottom-up methods. Here again there are far more classes where gc-cdcl↑ is better than classes (such as Insertions again) where the opposite is true. Moreover, although gc-cdcl↑ finds better lower bounds than gc-cdcl on two large classes (DSJ and flat), this does not translate to a higher proof ratio.

Table 3 shows results aggregated across all instances. We report the average ratio of instances proven optimal ('optimal') in the first column. Then in the second to the fifth columns, we report the geometric ('avg (G)') and arithmetic averages ('avg') for both the lower and upper bounds. Finally, we report the *mean normalized gap to the best upper bound*, and to the best lower bound. Let b (resp. w) be the value found by best (resp. worst) method. In the case of the lower bound, b will be the maximum, while it will be the minimum for the upper bound. The normalized gap $g(x)$ of the outcome x is:

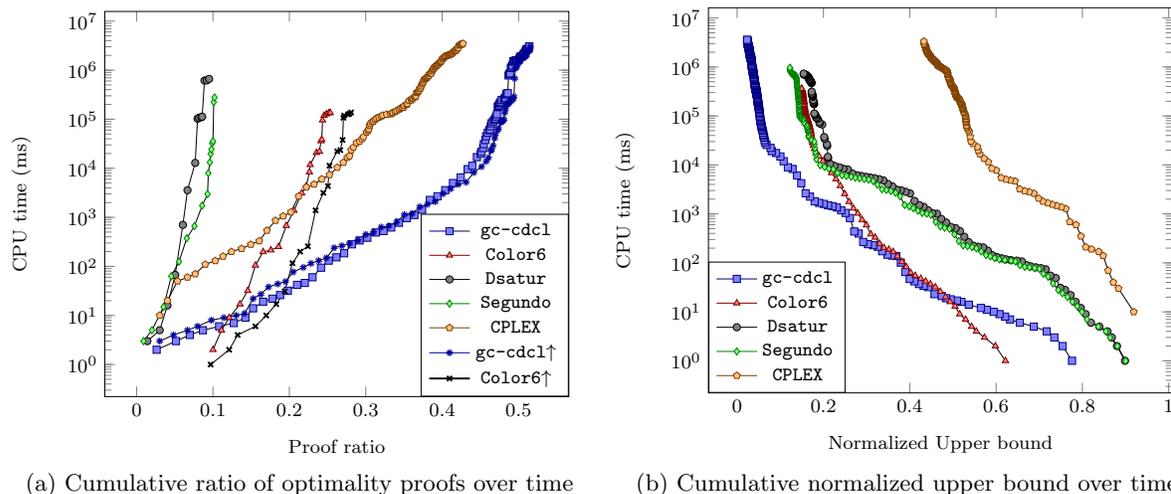$$g(x) = \begin{cases} 0 & \text{if } b = w \\ (b-x)/(b-w) & \text{otherwise} \end{cases}$$

(a) Cumulative ratio of optimality proofs over time

(b) Cumulative normalized upper bound over time

Figure 4: Comparison with the state of the art – anytime results on the `DIMACS` data set

method	Optimal	χ^{up}		χ^{low}		gap (ub)	gap (lb)
	avg	avg (G)	avg	avg (G)	avg	avg	avg
`gc-cdcl↑`	0.51515	15.335	30.575	11.508	18.989	0.1380	0.0821
`gc-cdcl`	0.51414	15.023	29.721	10.671	18.505	0.0766	0.2666
`CPLEX`	0.42727	16.111	33.410	10.787	17.940	0.3327	0.2177
`Color6↑`	0.28081	17.028	34.170	11.413	18.911	0.4879	0.1776
`Color6`	0.25354	15.550	30.881	9.854	17.351	0.1710	0.4876
`Segundo`	0.10202	15.370	30.573	5.721	12.091	0.1474	0.6880
`Dsatur`	0.09495	15.605	30.899	5.721	12.091	0.1820	0.6880

Table 3: Comparison with the state of the art – global results

A mean normalized gap of 0 (resp. 1) therefore indicates that the method systematically has the best (resp. worst) outcome.

Figure 4 gives an anytime view of these aggregated results. The left plot (Figure 4a) shows the ratio of instances for which optimality is proven within a given amount of CPU time. The right plot (Firgure 4b) shows the evolution of the upper bound normalized to $[0, 1]$ with respect to best and worst upper bounds found by any top-down method. We can see that when given less than a second, `Color6` seems to start up more quickly and finds better colorings and close more instances than `gc-cdcl`. It is not clear if this result can be attributed to the fact that `gc-cdcl` computes χ^{low} and χ^{up} at start up, while these values are given as arguments to `Color6`. In any case, given more time, `gc-cdcl` clearly dominates all other methods on this data set.

Overall, the variants of `gc-cdcl` are best for all criteria. `CPLEX` is third best for the number of optimality proofs. Although it requires a lot of memory, and is very poor in

		gc-cdcl			Color6			Dsatur			Segundo			CPLEX		
	#	Opt.	χ^{up}	χ^{low}	Opt.	χ^{up}	χ^{low}	Opt.	χ^{up}	χ^{low}	Opt.	χ^{up}	χ^{low}	Opt.	χ^{up}	χ^{low}
$|V|$=60	471	**1.00**	11.99	**11.99**	0.88	12.14	11.65	0.00	11.99	9.70	0.00	11.99	9.70	0.97	11.99	11.96
$|V|$=70	417	0.76	14.10	13.07	**0.85**	14.30	13.53	0.00	14.07	11.05	0.00	14.07	11.05	0.67	14.11	**13.75**
$|V|$=75	410	0.55	14.79	12.70	**0.85**	14.90	14.00	0.00	14.66	11.33	0.00	14.66	11.33	0.50	14.88	**14.15**
$|V|$=80	422	0.38	15.62	12.71	**0.95**	15.54	**15.28**	0.00	15.45	11.87	0.00	15.45	11.87	0.41	15.79	14.67
$|V|$=85	60	0.26	31.24	27.25	0.65	31.63	29.35	0.00	**31.00**	26.40	0.00	31.02	26.40	**0.82**	31.07	**30.72**
$|V|$=90	90	0.16	25.41	20.63	**0.63**	25.73	22.97	0.00	25.04	20.32	0.00	**25.03**	20.32	0.61	25.26	**24.18**
$|V|$=95	15	0.00	28.49	21.13	**0.27**	28.53	22.60	0.00	28.07	21.13	0.00	**28.00**	21.13	0.27	28.67	**25.80**
$|V|$=100	15	0.00	29.98	21.33	0.07	29.87	21.87	0.00	**29.60**	21.47	0.00	29.60	21.47	0.07	30.60	**26.27**
$|V|$=105	15	0.00	31.11	22.40	**0.20**	30.87	23.60	0.00	30.87	22.60	0.00	**30.80**	22.60	0.07	31.87	**27.00**
$|V|$=110	15	0.00	32.08	22.67	**0.07**	32.53	23.13	0.00	32.20	22.67	0.00	**31.93**	22.67	0.00	33.67	**27.00**
$|V|$=120	15	0.00	34.53	23.13	0.00	34.93	23.13	0.00	34.53	21.20	0.00	34.73	21.20	0.00	37.27	**28.40**
$|V|$=130	15	0.00	**37.39**	23.87	0.00	38.00	23.87	0.00	37.93	12.60	0.00	38.13	12.60	0.00	43.20	**27.93**
$|V|$=140	5	0.00	**50.60**	35.40	0.00	51.40	35.40	0.00	51.20	0.00	0.00	51.20	0.00	0.00	56.40	**39.80**

Table 4: Comparison with top-down methods – breakdown by classes of `RCBII` instances

terms of solution quality, `CPLEX` often gives good lower bounds. This is not so surprising since the linear relaxation is quite potent on this formulation. For instance at the root node, since we fix the variables of a maximal clique, the lower bound from the linear relaxation can only be higher than that of clique-based methods. It should be noted, however, that in many cases it was not able to improve on the initial bounds provided to the model, even when memory was not an issue. `Color6`↑ is second best for the lower bound, but notice that the much larger mean normalized gap to the best lower bound than `gc-cdcl`↑ indicates that it is better mostly when both methods are far better than the worst method since the marginal gain is smaller. We observe that `Dsatur` (and in particular the improved version `Segundo`), even though extremely simple, is still a very good method to actually find small colorings and is a close second best for the upper bound.

Finally, we performed further experiments on the `RCBII` data introduced in (Segundo, 2012) and containing random instances ranging from 60 to 140 vertices and edge density ranging from 0.1 to 0.9. We show the results in table 4. The results are averaged over instances with same number of vertices. Here again, we keep only the instances that at least one solver could not solve to optimality (there are 1965 of them). The same results on individual instances (averaged over the 10 runs) are shown in Appendix B.

The results of `gc-cdcl` are not as impressive on this data set. However, interestingly, it is robust with respect to the different criteria. For instance, `CPLEX` almost always finds the highest lower bounds but upper bounds are not as good and actually really poor for the largest instances. The two DSATUR variants find the best upper bounds but poor lower bounds and cannot solve any instance to optimality. Finally, `Color6` can compute far more proofs than other techniques, although the upper bounds it finds are generally the worst, except on large instances. The results of `gc-cdcl`, on the other hand, are not impressive, but relatively good on all criteria. Moreover, we observe that it is the only method that can close the class of smallest instances, and it finds the best colorings on the largest instances.

Notice that the largest classes (≥ 100 vertices) have much fewer instances, and those all have a high density (≥ 0.7). Higher densities tend to favour our approach since the Zykov search tree is shallower.
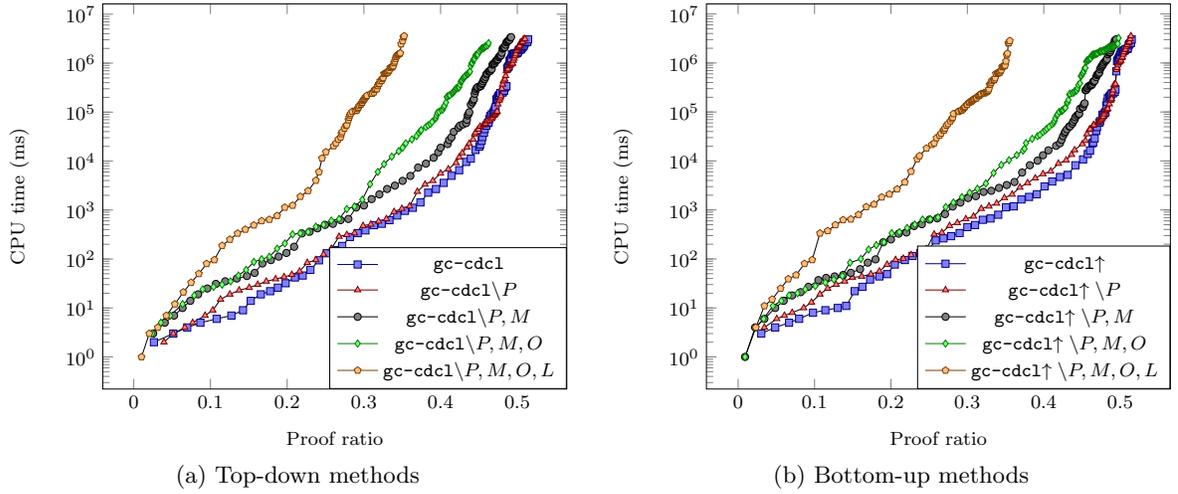
(a) Top-down methods   (b) Bottom-up methods

Figure 5: Factor analysis – ratio of optimality proofs over time on the `DIMACS` data set

7.2 Factor Analysis

Next, we analyse the impact of the new bounds, of the marginal cost pruning, and of learning using six variants of `gc-cdcl`. Let P denote pruning, L the usage of clause learning, M the Mycielskian-based lower bound and O the partition-based vertex ordering used to find maximal cliques. Then, $X \setminus S$ stands for the solver X where the options in S are turned off. The results reported in Table 5 and Figure 5 show the impact of each factor.

method	Optimal	χ^{up}		χ^{low}		gap (ub)	gap (lb)
	avg	avg (G)	avg	avg (G)	avg	avg	avg
`gc-cdcl`↑	0.51515	15.335	30.575	11.508	18.989	0.1380	0.0821
`gc-cdcl`	0.51414	15.023	29.721	10.671	18.505	0.0766	0.2666
`gc-cdcl`↑ $\setminus P$	0.51414	15.335	30.575	11.491	18.972	0.1380	0.0857
`gc-cdcl`$\setminus P$	0.51010	15.043	29.723	10.668	18.503	0.0776	0.2676
`gc-cdcl`↑ $\setminus P, M, O$	0.49798	15.185	30.381	11.316	18.915	0.1261	0.1335
`gc-cdcl`↑ $\setminus P, M$	0.49495	15.335	30.575	11.423	18.866	0.1380	0.0953
`gc-cdcl`$\setminus P, M$	0.49192	15.044	29.727	10.388	18.384	0.0783	0.2915
`gc-cdcl`$\setminus P, M, O$	0.46263	15.209	30.084	10.280	18.339	0.1027	0.3283
`gc-cdcl`↑ $\setminus P, M, O, L$	0.35556	15.516	30.666	11.053	18.747	0.1560	0.1545
`gc-cdcl`$\setminus P, M, O, L$	0.35253	15.265	30.114	10.029	18.212	0.1065	0.3597

Table 5: Factor analysis – detailed results on the `DIMACS` data set

There is an almost perfect correlation between turning off a feature, and moving down the ranking for any criterion. In particular, clause learning has clearly a very high impact as turning it off systematically and significantly degrades the performances on every criterion. Moreover, using the partition-based vertex ordering also has a very significant impact for such a simple technique. Finally the Mycielskian-based lower bound also helps. However, its impact is limited. For instance, with respect to `gc-cdcl`$\setminus P, M$, `gc-cdcl`$\setminus P$ increases the

	#	gc-cdcl		gc-cdcl^{col}		gc-cdcl$^{col}_{vsids}$		gc-cdcl^{inc}		gc-cdcl^{pru}		gc-cdcl^{IS}		gc-cdcl^{tr}	
		χ^{up}	χ^{low}	χ^{up}	χ^{low}	χ^{up}	χ^{low}	χ^{up}	χ^{low}	χ^{up}	χ^{low}	χ^{up}	χ^{low}	χ^{up}	χ^{low}
DSJ	14	76.0	32.5	76.6	32.5	78.4	32.5	**76.0**	32.5	76.7	32.5	76.3	32.4	76.1	32.5
FullIns	14	6.8	6.8	6.8	6.8	6.8	6.7	6.8	6.8	6.8	6.7	6.8	6.5	6.8	6.8
Insertions	11	5.2	2.5	5.2	2.5	5.2	2.5	5.2	2.5	5.2	2.5	5.2	2.5	5.2	2.5
abb313GPIA	1	10.0	8.0	10.0	8.0	10.0	8.0	10.0	8.0	10.1	8.0	**9.9**	**8.1**	10.0	8.0
ash	3	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.3	3.7	4.0	4.0
flat	6	73.9	12.5	74.3	12.5	**43.3**	12.5	74.0	12.5	74.9	12.5	74.3	12.5	74.0	12.5
fpsol2	1	65.0	65.0	65.0	65.0	65.0	65.0	65.0	65.0	65.0	65.0	65.0	65.0	65.0	65.0
inithx	1	54.0	54.0	54.0	54.0	54.0	54.0	54.0	54.0	54.0	54.0	54.0	54.0	54.0	54.0
latin	1	118.2	90.0	122.0	90.0	126.9	90.0	121.2	90.0	120.2	90.0	119.2	90.0	**118.0**	90.0
le450	10	14.8	13.0	15.9	13.0	15.2	13.0	14.9	13.0	15.4	13.0	15.0	13.0	**14.8**	13.0
miles1000	1	42.0	42.0	42.0	42.0	42.0	42.0	42.0	42.0	42.0	42.0	42.0	42.0	42.0	42.0
mug	4	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0
myciel	5	6.0	6.0	6.0	6.0	6.0	6.0	6.0	6.0	6.0	6.0	6.0	5.0	6.0	6.0
qg	4	58.0	57.5	58.0	57.5	58.4	57.5	58.0	57.5	58.2	57.5	58.1	57.5	58.0	57.5
queen	12	**12.6**	11.3	12.7	11.3	12.8	11.3	12.7	11.3	12.7	11.3	12.7	11.3	12.7	11.3
school1	2	14.0	14.0	14.0	14.0	14.0	14.0	14.0	14.0	14.0	14.0	14.0	14.0	14.0	14.0
wap0	8	46.0	41.4	**45.8**	41.4	46.0	41.4	45.9	41.4	46.1	41.4	46.0	41.4	46.0	41.4
will199GPIA	1	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0

Table 6: Variants of the method – breakdown by class of DIMACS instances

proof ratio by 3.7%, the mean lower bound by 0.6%, and decreases the mean upper bound by a tiny fraction. Finally, even though the impact is very slight, pruning by contracting vertices when the marginal cost of adding the edge is too high seems to help. In particular, we can see in Figure 5 that the improvement is larger for lower CPU time. In other words, this technique only slightly improve the results, but it helps achieve similar results significantly earlier (CPU time is log scaled). This phenomenon is not usual with relatively weak pruning mechanisms.

Finally, the results in Table 6 compare different variants of our approach.

In particular, we show the results of using color variables and channelling constraints. The variants denoted gc-cdcl^{col} and gc-cdcl$^{col}_{vsids}$ augment the model used in gc-cdcl with a color variable x_v for every vertex v of the graph. Moreover, for every pair u,v of vertices we add the constraint $x_v \neq x_u$ if the edge (uv) is in the graph, and otherwise we add the constraint $e_{uv} \iff x_v = x_u$. The former uses DSATUR as branching heuristic, whereas the latter uses VSIDS (Moskewicz et al., 2001).

We can observe that the methods using color variables are overall not as efficient. There are few exceptions, for instance gc-cdcl$^{col}_{vsids}$ obtains extremely good results on the flat class. However, this result cannot be attributed to VSIDS as the best upper bounds are found almost backtrack-free. Rather this seems to be the consequence of the input order and the pruning provided by the channelling constraints being lucky on these instances.

Then we show the impact of: updating the cliques incrementally for the lower bound as described in Section 5.1 (gc-cdcl^{inc}); pruning both implied contractions and edge additions as described in Section 6.3 (gc-cdcl^{pru}); the independent set extraction preprocessing

method described in Section 6.2 ($\texttt{gc-cdcl}^{IS}$); and the transitivity-aware adaptation of unit propagation described in Section 4.3 ($\texttt{gc-cdcl}^{tr}$).

A virtual best method would use almost all of these techniques, except the additional pruning which seems to be systematically too costly to be useful. In particular, transisivity-aware unit propagation significantly helps on the large $\texttt{latin}$ instance and on the class $\texttt{le450}$ while almost matching the overall results of the chosen default: it solves the same number of instances and the mean upper bound is only 0.04% higher. The other variants however, are slightly worse than the default on average.

8. Conclusion

We have presented a CP/SAT hybrid approach to graph coloring. The approach uses a new, sophisticated, lower bound that generalizes the clique bound and is inspired by Mycielskian graphs. We combined it with clause learning, effective primal heuristics for coloring, pruning, and preprocessing, to get a solver that in both its configurations outperforms the previous state of the art in satisfiability-based coloring, constraint programming-based coloring, as well as a MIP model of the problem. The main disadvantage of the approach is that it requires one Boolean variable for each non-edge of the graph and hence cannot scale to large sparse graphs.

Acknowledgements

References

Aardal, K. I., Van Hoesel, S. P., Koster, A. M., Mannino, C., & Sassano, A. (2007). Models and solution techniques for frequency assignment problems. *Annals of Operations Research*, *153*(1), 79–129.

Abello, J., Pardalos, P., & Resende, M. G. C. (1999). External memory algorithms. In Abello, J. M., & Vitter, J. S. (Eds.), *In External Memory Algorithms*, chap. On Maximum Clique Problems in Very Large Graphs, pp. 119–130. American Mathematical Society, Boston, MA, USA.

Bessiere, C., Katsirelos, G., Narodytska, N., & Walsh, T. (2009). Circuit Complexity and Decompositions of Global Constraints. In *Proceedings of IJCAI 2009*.

Brélaz, D. (1979). New Methods to Color the Vertices of a Graph. *Commun. ACM*, *22*(4), 251–256.

Chaitin, G. J., Auslander, M. A., Chandra, A. K., Cocke, J., Hopkins, M. E., & Markstein, P. W. (1981). Register allocation via coloring. *Comput. Lang.*, *6*(1), 47–57.

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pp. 151–158.

Cornaz, D., & Jost, V. (2008). A one-to-one correspondence between colorings and stable sets. *Operations Research Letters*, *36*(0), 673–676.

De Cat, B., Denecker, M., Bruynooghe, M., & Stuckey, P. (2015). Lazy model expansion: Interleaving grounding with search. *Journal of Artificial Intelligence Research*, *52*, 235–286.

Eppstein, D., Löffler, M., & Strash, D. (2013). Listing All Maximal Cliques in Large Sparse Real-World Graphs. *ACM Journal of Experimental Algorithmics*, *18*(3.1–3.21).

Frost, D., & Dechter, R. (1995). Look-ahead Value Ordering for Constraint Satisfaction Problems. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-1995)*, pp. 572–578.

Furini, F., Gabrel, V., & Ternier, I.-C. (2016). Lower bounding techniques for dsatur-based branch and bound. *Electronic Notes in Discrete Mathematics*, *52*, 149 – 156. INOC 2015 – 7th International Network Optimization Conference.

Gomes, C., Selman, B., & Kautz, H. (1998). Boosting Combinatorial Search Through Randomization. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-1998)*, pp. 431–438.

Grötschel, M., Lovász, L., & Schrijver, A. (1981). The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, *1*, 169—197.

Hebrard, E., & Katsirelos, G. (2018). Clause Learning and New Bounds for Graph Coloring. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP-2018)*, pp. 179–194.

Huang, J. (2007). The Effect of Restarts on the Efficiency of Clause Learning. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-2007)*.

Jansen, B. M., & Pieterse, A. (2018). Optimal data reduction for graph coloring using low-degree polynomials. *CoRR*, *abs/1802.02050*.

Jiang, H., Li, C., & Manyà, F. (2017). An Exact Algorithm for the Maximum Weight Clique Problem in Large Graphs. In *Proceedings of the 31st Conference on Artificial Intelligence (AAAI-2017)*, pp. 830–838.

Karp, R. M. (1972). Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, pp. 85–103.

Katsirelos, G., & Bacchus, F. (2005). Generalized Nogoods in CSPs. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-2005)*, pp. 390–396.

Lick, D. R., & White, A. T. (1970). k-degenerate graphs. *Canadian Journal of Mathematics*, *22*, 1082–1096.

Lin, J., Cai, S., Luo, C., & Su, K. (2017). A Reduction based Method for Coloring Very Large Graphs. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI-2017)*, pp. 517–523.

Lovász, L. (1979). On the shannon capacity of a graph. *IEEE Transactions on Information Theory*, *25*, 1–7.

Malaguti, E., Monaci, M., & Toth, P. (2011). An exact approach for the vertex coloring problem. *Discrete Optimization*, *8*(2), 174 – 190.

Malaguti, E., & Toth, P. (2010). A survey on vertex coloring problems. *ITOR*, *17*(1), 1–34.

Marques-Silva, J. P., & Sakallah, K. A. (1999). GRASP—a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, *48*(5), 506–521.

Mehrotra, A., & Trick, M. A. (1995). A column generation approach for graph coloring. *INFORMS Journal on Computing*, *8*, 344–354.

Mehrotra, A., & Trick, M. A. (1996). A column generation approach for graph coloring. *INFORMS Journal on Computing*, *8*(4), 344–354.

Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., & Malik, S. (2001). Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 39th Design Automation Conference (DAC-2001)*, pp. 530–535.

Mycielski, J. (1955). Sur le coloriage des graphes. In *Colloq. Math*, Vol. 3, pp. 161–162.

Newman, N., Fréchette, A., & Leyton-Brown, K. (2017). Deep optimization for spectrum repacking. *Commun. ACM*, *61*(1), 97–104.

Ohrimenko, O., Stuckey, P. J., & Codish, M. (2007). Propagation = lazy clause generation. In Bessiere, C. (Ed.), *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP-2007)*, Vol. 4741 of *LNCS*, pp. 544–558. Springer-Verlag.

Park, T., & Lee, C. Y. (1996). Application of the graph coloring algorithm to the frequency assignment problem. *Journal of the Operations Research society of Japan*, *39*(2), 258–265.

Schaafsma, B., Heule, M., & van Maaren, H. (2009). Dynamic Symmetry Breaking by Simulating Zykov Contraction. In *12th International Conference on Theory and Applications of Satisfiability Testing (SAT-2009)*, pp. 223–236.

Segundo, P. S. (2012). A new dsatur-based algorithm for exact vertex coloring. *Computers & Operations Research*, *39*(7), 1724 – 1733.

Stergiou, K. (2008). Heuristics for Dynamically Adapting Propagation. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI-2008)*, pp. 485–489.

Tarjan, R. E., & van Leeuwen, J. (1984). Worst-case analysis of set union algorithms. *Journal of the ACM*, *31*(2), 245–281.

Trick, M. A. (Ed.). (2002). *Computational Symposium on Graph Coloring and its Generalizations (COLOR-2002)*.

Van Gelder, A. (2008). Another Look at Graph Coloring via Propositional Satisfiability. *Discrete Appl. Math.*, *156*(2), 230–243.

Van Hentenryck, P., Ågren, M., Flener, P., & Pearson, J. (2003). Tractable Symmetry Breaking for CSPs with Interchangeable Values. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-2003)*, pp. 277–282.

Verma, A., Buchanan, A., & Butenko, S. (2015). Solving the maximum clique and vertex coloring problems on very large sparse networks. *INFORMS J. on Computing, 27*(1), 164–177.

Walsh, T. (2000). SAT v CSP. In *Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Singapore, September 18-21, 2000, Proceedings*, pp. 441–456.

Zhou, Z., Li, C.-M., Huang, C., & Xu, R. (2014). An exact algorithm with learning for the graph coloring problem. *Computers & Operations Research, 51*, 282–301.

Zykov, A. A. (1949). On some properties of linear complexes. *Mat. Sb. (N.S.), 24(66)*(2), 163–188.