Dynamic Symmetry Breaking by Simulating Zykov Contraction

Bas Schaafsma, Marijn Heule* and Hans van Maaren

Department of Software Technology, Delft University of Technology
schaafsma@ch.tudelft.nl, marijn@heule.nl, h.vanmaaren@tudelft.nl

Abstract. We present a new method to break symmetry in graph coloring problems. While most alternative techniques add symmetry breaking predicates in a pre-processing step, we developed a learning scheme that translates each encountered conflict into *one* conflict clause which covers equivalent conflicts arising from *any* permutation of the colors.
Our technique introduces new Boolean variables during the search. For many problems the size of the resolution refutation can be significantly reduced by this technique. Although this is shown for various hand-made refutations, it is rarely used in practice, because it is hard to determine which variables to introduce defining useful predicates. In case of graph coloring, the reason for each conflicting coloring can be expressed as a node in the Zykov-tree, that stems from merging some vertices and adding some edges. So, we focus on variables that represent the Boolean expression that two vertices can be merged (if set to true), or that an edge can be placed between them (if set to false). Further, our algorithm reduces the number of introduced variables by reusing them.
We implemented our technique in the state-of-the-art solver minisat. It is competitive with alternative SAT based techniques for graph coloring problems. Moreover, our technique can be used on top of other symmetry breaking techniques. In fact, combined with adding symmetry breaking predicates, huge performance gains are realized.

1 Introduction

Satisfiability (SAT) solvers have become very powerful in recent years. Especially conflict-driven clause learning SAT solvers can effective tackle certain huge problems. Crucial to strong performance is learning *conflict clauses* that ensure that the same search space is not explored multiple times. However, in the presence of symmetry the effectiveness of conflict clauses is highly reduced: search spaces could be visited that are symmetric to already refuted areas.

This paper focusses on symmetry in graph coloring problems. In particulary, we want to break the symmetry that arises by permuting the colors. This can be broken *statically*, as a preprocessing step, or *dynamically*, during the search. A frequently used static technique assigns a different color to all vertices in a large

* Supported by the Dutch Organization for Scientific Research (NWO) under grant 617.023.611

clique [19]. Although effective and cheap (a large clique is easy to find), it only breaks the symmetry partially [15]. A dynamic symmetry breaking technique [10] adds, besides the conflict clause expressing the conflict, all symmetric conflict clauses. Yet the number of symmetric conflict clauses grows exponentially with the number of colors. Here, we present an alternative dynamic technique.

For each conflicting assignment of k colors in the DPLL-tree there exists $k!$ symmetric conflicting assignments that can be obtained by a permutation of the colors. At the core of our algorithm is the observation that all these symmetric conflicting assignments correspond to the same node in the Zykov-tree: a binary search tree that selects in each node two nonadjacent vertices of the graph being colored. One branch explores the search space by merging these vertices (the same color), while the other branch examines the space created by placing an edge between them (not the same color). We transform each conflicting DPLL-node to the corresponding Zykov-node and translate the latter back to SAT.

Since the Zykov algorithm branches on merging two nonadjacent vertices or placing an edge between, new variables are introduced – called *merge variables*: these variables represent that two vertices must have the same color (a merge step) if set to true, while they must be colored differently (adding an edge) if set to false. The proposed technique converts the original variables in conflict clauses to merge variables.

The outline of this paper is as follows: Section 2 deals with encoding graph coloring problems into SAT. Transformation of conflict clauses is explained in Section 3. Section 4 offers experimental results. Finally, in Section 5 we draw some conclusions and provide suggestions for future research.

2 Preliminaries

2.1 The Satisfiability problem

The Satisfiability problem (in short SAT) asks whether there exists an assignment for a given Boolean formula such that it evaluates to *true*. If such an assignment does exist, we call the problem satisfiable else the problem is qualified as unsatisfiable. In a more formal setting a formula $\mathcal{F} = \{C_1 \wedge \ldots \wedge C_m\}$ consists of conjunction of clauses C_i, while each clause $C_i = (l_{i,1} \vee \ldots \vee l_{i,j})$ consists of disjunction of literals. A literal l refers either to a Boolean variable x_i or to its negation $\neg x_i$. A clause is satisfied when at least one of its literals evaluates to *true*. Finally, a satisfying assignment satisfies all clauses.

2.2 The k-coloring problem

The k-coloring problem deals with the question whether the vertices of a graph can be colored with k colors such that two connected vertices have a different color. Or more formal, let φ_{color} be a mapping of vertices $v \in V$ onto an integer in $\{1, \ldots, k\}$. A graph $G = (V, E)$ is k-colorable, when there exists a φ_{color} to *all* vertices such that for every $(v, w) \in E$, $\varphi_{\text{color}}(v) \neq \varphi_{\text{color}}(w)$. The smallest k for which G is still k-colorable is known as the chromatic number of G or $\mathcal{X}(G)$.

The k-coloring problem can be naturally translated to SAT. We focus on the widely used *direct encoding* [14]. It uses Boolean variables $x_{v,i} \leftrightarrow \varphi_{\text{color}}(v) = i$, which we refer to as *color variables*. The property that all vertices must be colored is encoded by the *at-least-one* clauses, which are of the form:

$$\bigwedge_{v \in V} (x_{v,1} \vee x_{v,2} \vee \cdots \vee x_{v,k}) \tag{1}$$

Further, for each $(v, w) \in E$, k *conflicting clauses* encode $\varphi_{\text{color}}(v) \neq \varphi_{\text{color}}(w)$:

$$\bigwedge_{1 \leq i \leq k} \bigwedge_{(v,w) \in E} (\neg x_{v,i} \vee \neg x_{w,i}) \tag{2}$$

The above is known as the *minimum encoding* [9]. The *extended encoding* adds redundant clauses which encode that vertices must have *at-most-one* color:

$$\bigwedge_{1 \leq i < j \leq k} \bigwedge_{v \in V} (\neg x_{v,i} \vee \neg x_{v,j}) \tag{3}$$

Although optional, most complete solvers perform better on instances where these clauses have been added [14]. Yet for our technique they are not required.

2.3 Zykov Contraction algorithms

One of the main family of algorithms which determines $\mathcal{X}(G)$ for a graph G, or approximates $\mathcal{X}(G)$ is known as Contraction. This family of algorithms is based on a theorem due to Zykov [20], which states:

$$\mathcal{X}(G) = min(\mathcal{X}(G/(v, w)), \mathcal{X}(G + (v, w))) \tag{4}$$

In this theorem, $G/(v, w)$ denotes the graph with vertex v and w contracted, meaning that vertex w is deleted and all its edges are transferred to v. $G+(v, w)$ means that an edge is added between vertex v and w, as shown in Figure 1.

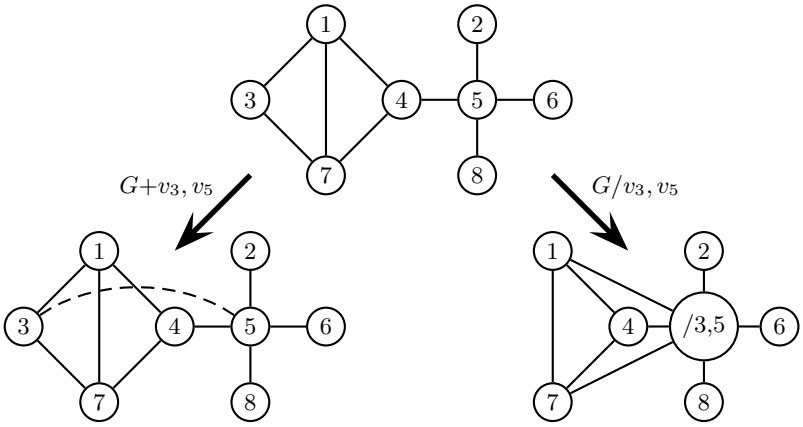

Fig. 1. A Zykov-tree example. The numbers in the vertices refer to their index v_i.

Repeated steps of applying this theorem to a graph G result in a binary tree. The leaves of this tree are fully connected graphs, which each have a chromatic number equal to their number of vertices. The chromatic number of G is then equal to the chromatic number of the graph with the least amount of vertices.

Zykov Contraction can be simulated in a SAT solver by adding redundant variables and clauses to the CNF translation of a graph coloring problem. Adding redundant variables and clauses was introduced by Tseitin and is known as *Extended Resolution* (ER) [17]. ER is shown to be very powerful in theory [3].

Each step of the Contraction algorithm can be simulated by introducing a Boolean variable $e_{v,w}$, referred to as *merge variables*, which expresses:

$$e_{v,w} \leftrightarrow \varphi_{\text{color}}(v) = \varphi_{\text{color}}(w) \tag{5}$$

This relation can be translated to CNF using the following clauses:

$$\bigwedge_{1 \leq i \leq k} (e_{v,w} \vee \neg x_{v,i} \vee \neg x_{w,i}) \wedge (\neg e_{v,w} \vee x_{v,i} \vee \neg x_{w,i}) \wedge (\neg e_{v,w} \vee \neg x_{v,i} \vee x_{w,i}) \tag{6}$$

These clauses will propagate the fact that vertices v and w have equal or unequal colors when $e_{v,w}$ is set. If set to true the clauses simulate merging two vertices, while setting $e_{v,w}$ to false represents placing an edge between them.

Initially, we studied the use of adding merge variables and the corresponding clauses to a given formula as a preprocessing step. This turned out to merely decrease the performance. However, we observed that one could capitalize on the expressive power of merge variables by strengthening conflict clauses. Therefore, instead of ER, only the clauses are added which are required for the Tseitin translation [17] of these learnt clauses.

3 Merge clauses

A powerful application of simulating Contraction lies in strengthening conflict clauses in conflict-driven algorithms [11] for graph coloring instances. Simply put, conflict-driven solvers continue to assign variables until a conflict is detected. When a conflict is detected, the solver determines an assignment responsible for this conflict and adds a conflict clause C_{conflict} to $\mathcal{F}$, where C_{conflict} is the negation of the assignments which led to the conflict.

To illustrate the benefits of simulating Contraction, consider the example conflict, in the 3-coloring instance presented in Figure 2. In the corresponding SAT instance, a conflict clause for this conflict would be:

$$(\neg x_{1,1} \vee \neg x_{2,1} \vee \neg x_{3,2} \vee \neg x_{4,2} \vee \neg x_{5,3}) \tag{7}$$

Yet, due to the inherent symmetries of a k-coloring instance, any permutation of colors in a conflicting assignment is also a conflicting assignment. Thus for the corresponding SAT instance, the following clause is also logically implied by $\mathcal{F}$:

$$(\neg x_{1,3} \vee \neg x_{2,3} \vee \neg x_{3,1} \vee \neg x_{4,1} \vee \neg x_{5,2}) \tag{8}$$

Unfortunately, with a maximum of $k!$ possible permutations it is impractical to add each implied clause, for almost any k larger than four [10].

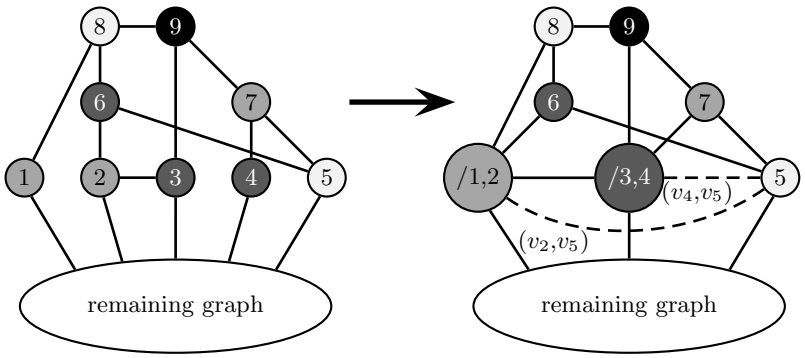

Fig. 2. A Zykov Contraction example. The numbers in the vertices refer to the index v_i. The added edges are shown as dashed lines. Vertex v_9 is in conflict because it cannot be colored. The example focusses on the assignment to v_1, v_2, v_3, v_4, and v_5.

3.1 Transforming conflict clauses

Any conflict clause, which consists of negative color literals, such as the clauses depicted in (7) and (8), encodes a conflicting coloring φ_{color} of a subset of vertices in G. This encoded coloring corresponds to some node in a Zykov-tree with G as root. Vertices in this subset that are equally colored in φ_{color} are contracted into a single vertex and edges are added to induce a clique among these contracted vertices. This relation exists, because once the vertices are contracted and the clique is induced, any two vertices equally colored in φ_{color}, will be equally colored in any coloring of our created clique, because they have been merged. Furthermore, any two vertices that were not equally colored in φ_{color}, will be unequally colored in any coloring of our clique, because there exists an edge between them. Thus any coloring of the created clique corresponds to a permutation of φ_{color} and therefore will, just like φ_{color}, result in a conflict. Therefore, any conflict clause consisting out of negative color literals can be converted in a *corresponding merge clause*, denoted by C_{merge}, which is a conflict clause consisting out of merge variables.

Back to the example, consider the conflict depicted in Figure 2 as a node in a Zykov-tree, in which v_1 and v_2 are merged, v_3 and v_4 are merged, and the edges (v_2, v_5), (v_4, v_5) are added. This is represented using merge variables as:

$$(\neg e_{1,2} \vee \neg e_{3,4} \vee e_{2,5} \vee e_{4,5}) \tag{9}$$

In any merge clause C_{merge}, negative literals correspond to contractions of the equally colored vertices in φ_{color}. For each set of n equally colored vertices in φ_{color} we will need $n-1$ negative merge literals. The positive literals C_{merge} correspond to the edges added to induce a clique. Of course no edges need to be added between contracted vertices v and w, if such an edge already exists in G.

Unfortunately, in most cases one could choose from many merge variables to construct a merge conflict clause. In the example, instead of using $e_{2,5}$ (or $e_{4,5}$), one could select $e_{1,5}$ (or $e_{3,5}$). The choice of the merge variables influences

the performance, therefore one would prefer to select the "optimal" candidates. Heuristics for this selection are discussed in Section 3.2.

Besides C_{merge}, one also needs to add the clauses $M(C_{\text{merge}})$, which arise from the Tseitin translation [17], to $\mathcal{F}$. Theoretically, for each introduced merge variable we could add the full set of clauses described in Section 2.3. Yet, in practice it suffices to add only the clauses that contain the negation of the literal of our introduced variable. Only adding these clauses is good practice as it saves resources [13]. For any C_{merge}, $M(C_{\text{merge}})$ equals to:

$$\bigwedge_{1\leq i\leq k}\Big(\bigwedge_{e_{v,w}\in C_{\text{merge}}}((\neg e_{v,w}\vee x_{v,i}\vee\neg x_{w,i})\wedge(\neg e_{v,w}\vee\neg x_{v,i}\vee x_{w,i}))\wedge\bigwedge_{\neg e_{v,w}\in C_{\text{merge}}}(e_{v,w}\vee\neg x_{v,i}\vee\neg x_{w,i})\Big) \tag{10}$$

Thus $M(C_{\text{merge}})$ for our example is:

$$\bigwedge_{1\leq i\leq k}\begin{pmatrix}(e_{1,2}\vee\neg x_{1,i}\vee\neg x_{2,i})\wedge(\neg e_{2,5}\vee x_{2,i}\vee\neg x_{5,i})\wedge(\neg e_{2,5}\vee\neg x_{2,i}\vee x_{5,i})\wedge\\(e_{3,4}\vee\neg x_{3,i}\vee\neg x_{4,i})\wedge(\neg e_{4,5}\vee x_{4,i}\vee\neg x_{5,i})\wedge(\neg e_{4,5}\vee\neg x_{4,i}\vee x_{5,i})\end{pmatrix} \tag{11}$$

3.2 Implementation

We have applied the principal of merge conflict clauses in the conflict-driven clause learning (CDCL) SAT-solver architecture which we refer to as the CDCLMERGE algorithm. CDCLMERGE is specialized for the k-coloring problem and uses merge conflict clauses to store conflicts. Its most important feature is the TRANSFORMCONFLICT procedure, which transforms the color literals in a conflict clause to merge literals. In order to make the TRANSFORMCONFLICT function properly, we also had to adapt the DECIDE procedure. Algorithm 1 gives a detailed overview of the CDCLMERGE algorithm.

The DECIDE procedure
The proposed transformation to merge clauses requires that all conflicts can be expressed as a disjunctions of negative color literals and merge literals. This cannot be guaranteed if the solver branches on negative literals. E.g. consider the perfect graph of size three. Assigning $x_{1,1}$ to false, $x_{2,1}$ to false, and $x_{3,2}$ to true results in a conflict which can be expressed as $(x_{1,1} \vee x_{2,1} \vee \neg x_{3,2})$. Notice that this conflict clause cannot be translated to a merge clause is a meaningful way. Therefore, DECIDE is adapted such that it assigns each decision variable to true. This heuristic is similar to the one used in minisat which assigns all decision variables to false [7].

Algorithm 1 CDCLMERGE($\mathcal{F}$)

```
1: while true do
2:    PROPAGATE() /* propagate unit clauses */
3:    if not conflict then
4:       if all variables assigned then
5:          return satisfiable
6:       end if
7:       DECIDE() /*select decision variable. ADAPTED*/
8:    else
9:       C_conflict ← ANALYZE() /*analyze the conflict*/
10:      C_merge ← TRANSFORMCONFLICT(C_conflict) /*ADDED*/
11:      if top level conflict found then
12:         return unsatisfiable
13:      end if
14:      BACKTRACK(C_conflict) /*backtrack while C_conflict remains unit or falsified*/
15:   end if
16: end while
```

The TRANSFORMINGCONFLICT procedure
The input C_{conflict} is transformed into C_{merge} using the following steps:

1. Positive color literals in C_{conflict} are replaced by merge and negative color literals by expanding them into their reason literals. For instance, say the example conflict clause would have been $(x_{1,3} \vee x_{2,2} \vee x_{3,3} \vee x_{4,2} \vee \neg x_{5,3})$. Assume that the same conflicting coloring was its reason. In that case $\neg x_{1,2}$ will be the reason literal for $x_{1,3}$. Therefore, we can replace the latter by the former. This process is iterated while C_{conflict} contains positive literals.
2. Redundant literals (see Theorem 2 and 3) are removed from C_{conflict}.
3. C_{conflict} is split into C_{color}, which consist out of all negative color literals in C_{conflict} and C_{extra}, which consists of all merge literals in C_{conflict}.
4. Transform C_{color} into a merge clause C_{zykov} by computing the corresponding node in the Zykov-tree. Preliminary tests showed that the performance improved if the number of introduced variables were kept to a minimum and introduced variables were reused whenever possible. Therefore, in case of choice between possible merge literals to use in the transformation to C_{zykov}, the merge literal is selected which is most frequently used in conflict clauses. Ties are broken pseudo randomly.
5. Return the union of C_{zykov} and C_{extra} as the transformed clause C_{merge}.

The BACKTRACK procedure
Conflict-driven clause learning SAT solvers backtrack (also known as backjump) to the lowest decision level where the latest conflict clause is still a unit clause. In CDCLMERGE this aspect of the solving algorithm is not changed. However, if a conflict clause C_{conflict} is unit, a corresponding merge clause C_{merge} may not be unit.

Recall the example at the start of this section:

$$C_{\text{conflict}} \Leftrightarrow C_{\text{merge}}$$
$$(\neg x_{1,1} \vee \neg x_{2,1} \vee \neg x_{3,2} \vee \neg x_{4,2} \vee \neg x_{5,3}) \Leftrightarrow (\neg e_{1,2} \vee \neg e_{3,4} \vee e_{2,5} \vee e_{4,5})$$

Say that variable $x_{v,i}$ is assigned at level v. The BACKTRACK procedure will jump to level 4. At this level C_{conflict} is reduced to $(\neg x_{5,3})$, while C_{merge} is reduced to $(e_{2,5} \vee e_{4,5})$. The reason is that two merge literals refer to vertex v_5. Currently, this problem is solved by changing the DECIDE procedure in such a way that if the latest merge clause consists of multiple unassigned literals one of these literals is assigned to false. This is repeated until the merge clause becomes unit.

Although C_{conflict} is satisfiability equivalent to $C_{\text{merge}} \wedge M(C_{\text{merge}})$ (see Theorem 4), the transformation is not *arc-consistent* under unit propagation [8]. As soon as a merge clause contains multiple literals that refer to the same vertex, the merge clause will not become unit when the original conflict clause would be unit. In the example a similar problem would arise in case v_2 (or v_4) was the last assigned vertex, because both $\neg e_{1,2}$ and $e_{2,5}$ (or both $\neg e_{3,4}$ and $e_{4,5}$) occur in C_{merge}.

The lack of arc-consistency is a serious weakness of the current implementation. We study various options to deal with this weakness. An interesting partial solution is adding a second merge clause. Back to the example: besides C_{merge}, we could also add $(\neg e_{1,2} \vee \neg e_{3,4} \vee e_{1,5} \vee e_{3,5})$. This solves arc-consistency for vertex v_2 and v_4. However, the problem is still unsolved for vertex v_5. In general, a second merge clause can fix arc-consistency for all vertices that are colored the same as another vertex in the conflict clause.

3.3 Optimizations

Variable selection heuristics
Although merge variables are useful to create merge conflict clauses, they seem rather weak as decision variables. For instance, if a clique of size $k+1$ arises by assigning some merge variables (i.e. a conflicting assignment), one may not detect this at the CNF level (no empty clause). Therefore, we only branch on color (original) variables. This choice is also supported by some experiments.

Finally, we propose a specialized version of the VSIDS activity heuristic [12]. Since merge variables will not be selected as decision variables, it does not make sense to maintain an activity for them. If a merge variable should have been increased, we want to bump the activity of the corresponding color variables instead. This idea have been implemented using an activity counter for vertices too. Each time a merge variable contributes to a conflict, the activity heuristic of both corresponding vertices is increased. The selection of decision variables is narrowed by choosing a variable from the most active vertex. This variant of VSIDS is inspired by [2].

Symmetry breaking in the presence of unit clauses
In the presence of symmetry, it is good practice to add *symmetry breaking predicates* [15]. In case of graph coloring problems, one can search for a large clique and force all vertices in that clique to a different color – by adding unit clauses to the formula. Cliques in a graph can be cheaply detected using the algorithm by M. Trick [21]. In the more general context of CNF formulae, shatter [1] can be used to compute symmetry breaking predicates.

Apart from symmetry breaking predicates, many structured graph coloring problems, such as quasi-group instances [9], contain unit clauses. In case the symmetry is already partially broken by some unit clauses, it does not make sense to introduce merge variables.

Regarding the implementation: if unit clause $(x_{u,i}) \in \mathcal{F}$ and $\neg x_{u,i} \in C_{\text{conflict}}$, then none of the literals $\neg x_{v,i} \in C_{\text{conflict}}$ are replaced by merge literals. Further, if unit clause $(x_{u,i}) \in \mathcal{F}$, then for all positive merge literals $e_{u,w}$ that would have added, the positive color literal $x_{w,i}$ is added instead.

3.4 Proof of correctness of merge conflict clauses

Definition 1. *Let $\pi : (1, \ldots, k) \to (1, \ldots, k)$ be a function that is one to one and onto.*

Definition 2. *Let $\mathcal{P}_\pi$ be a function for which holds (with $l_{h,i}$ as literals of C_h):*

$$\begin{aligned}
\mathcal{P}_\pi(C_h) &= (\mathcal{P}_\pi(l_{h,1}) \vee \ldots \vee \mathcal{P}_\pi(l_{h,i})) \\
\mathcal{P}_\pi(\neg l_{h,i}) &= \neg \mathcal{P}_\pi(l_{h,i}) \\
\mathcal{P}_\pi(x_{v,i}) &= x_{v,\pi(i)} \\
\mathcal{P}_\pi(e_{v,w}) &= e_{v,w}
\end{aligned}$$

Theorem 1. *If Boolean function $\mathcal{F}$ represents a k-coloring problem and clause C_h is logically implied by $\mathcal{F}$, then for any π, $\mathcal{P}_\pi(C_h)$ is logically implied by $\mathcal{F}$.*

Proof. Every satisfying assignment makes C_h true. Applying π to the satisfying assignments yields a permutation of them. So, these assignments satisfy $\mathcal{P}_\pi(C_h)$.

Theorem 2. *Let C_{conflict} be a conflict clause consisting of merge literals and negative color literals. Let $\mathcal{C} = \{i : \neg x_{v,i} \in C_{\text{conflict}}\}$ denote the set of colors used in C_{conflict}. A literal $\neg x_{u,i} \in C_{\text{conflict}}$ is redundant, if C_{conflict} does not contain a literal $\neg x_{v,i}$ $(u \neq v)$ and for each $j \in \mathcal{C} (j \neq i)$ C_{conflict} contains a literal $\neg x_{w,j}$ $(u \neq w)$ such that $(u, w) \in E$ (the edge set).*

Proof. C_{conflict} with and without $\neg x_{u,i}$ correspond to the same node in the Zykov-tree, because $\neg x_{u,i}$ is the only literal assigned to i assures that no merge steps are required, while no edges have to be added, because for each $j \in \mathcal{C} (j \neq i)$, u is already connected to a vertex w with $\varphi_{\text{color}}(w) = j$.

Theorem 3. *Let C_{conflict} be a conflict clause consisting of merge literals and negative color literals. A literal $\neg e_{u,v} \in C_{\text{conflict}}$ is redundant, if $(\neg x_{u,i} \vee \neg x_{v,i}) \in C_{\text{conflict}}$, while a literal $e_{u,w} \in C_{\text{conflict}}$ is redundant, if $(\neg x_{u,i} \vee \neg x_{w,j}) \in C_{\text{conflict}}$.*

Proof. Any solution to a graph coloring problem assigns a Boolean value to *all* color variables. So, each solution will be a *full assignment*. Each full assignment which satisfies $\neg e_{u,v}$ also satisfies $(\neg x_{u,i} \vee \neg x_{v,i})$, while each full assignment which satisfies $e_{u,w}$ also falsifies $(\neg x_{u,i} \vee \neg x_{w,j})$.

Notice that based on this theorem, we can conclude that a formula is unsatisfiable if a conflict clause only consists of a negative color literal. We refer to a *reduced clause* if all redundant literals (based on Theorem 2 and 3) are removed.

Theorem 4. *Let Boolean function $\mathcal{F}$ represent a k-coloring problem and let C_h be a reduced clause logically implied by $\mathcal{F}$. If C_h consists of merge literals and negative color literals and C_{merge} is a corresponging merge clause of C_h, then*

$$\bigwedge_{\pi_1..\pi_{k!}} \mathcal{P}_{\pi_i}(C_{\text{h}}) \text{ is satisfiability equivalent to } C_{\text{merge}} \wedge M(C_{\text{merge}}) \tag{12}$$

Proof. Recall that any solution must be a full assignment. (UNSAT $\Rightarrow$ UNSAT) If a full assignment falsifies $\bigwedge \mathcal{P}_{\pi_i}(C_h)$, then there exists a π_i for which $\mathcal{P}_{\pi_i}(C_h)$ is falsified. Since $\mathcal{P}_{\pi_i}(C_{\text{merge}}) = C_{\text{merge}}$ also represents $\mathcal{P}_{\pi_i}(C_h)$, $C_{\text{merge}} \wedge M(C_{\text{merge}})$ is falsified as well. (SAT $\Rightarrow$ SAT) If a full assignment satisfies $\bigwedge \mathcal{P}_{\pi_i}(C_h)$ by merge literals in C_h, then C_{merge} is also satisfied because it contains all merge literals in C_h. Notice that because C_h is a reduced clause, it either contains zero negative color literals (in case the former case is applicable) or at least two negative color literals. A full assignment can only satisfy $\bigwedge \mathcal{P}_{\pi_i}(C_h)$ if two vertices are assigned a different color while the corresponding color literals in C_h have the same color index, or two vertices are assigned the same color while the corresponding color literals in C_h have the different color index. In both cases $C_{\text{merge}} \wedge M(C_{\text{merge}})$ is satisfied as well.

Thus once we have learnt C_{conflict}, we could add all clauses $\mathcal{P}_{\pi_i}(C_{\text{conflict}})$ to $\mathcal{F}$ (Theorem 1). Yet, based on Theorem 4, we add $C_{\text{merge}} \wedge M(C_{\text{merge}})$ instead. Furthermore, Theorem 4 implies that using merge conflict clauses requires that every conflict can be expressed into merge literals and negative color literals. In order to ensure this, the variables selection heuristics of the solver have to be adapted. This adaptation is described in Section 3.2.

4 Results

All experiments were performed on a 2.0 GHz Intel Core 2 Duo with 1 GB of DDR2 Memory. Instances were encoded using the extended direct encoding and we used the method of finding and forcing cliques as symmetry breaking method.

4.1 Medium sized random graphs

This experiment was performed to compare our CDCLMERGE implementation, referred to as MiniColor to the standard distribution of MiniSat2, branching on positive variables. In this experiment we generated 45 random graphs of 70 vertices, with varying edge probabilities (denoted by P_{edge}). Per graph, one SAT instance (G, $\mathcal{X}(G)$) and one UNSAT instance (G, $\mathcal{X}(G)$ -1) were created. Table 1 shows the average solving times and the number of solved instances.

As can be seen in Table 1 the performances on satisfiable instances are on par, although MiniColor was able to solve one more instance. On the other hand, performance on unsatisfiable instances has significantly improved. Besides solving more instances, MiniColor was on average one order of magnitude faster.

Table 1. Average runtimes for medium sized graphs, with a 1200 (s) timeout.

				SAT instances				UNSAT instances			
				Minisat2		MiniColor		Minisat2		MiniColor	
$\|G\|$	$\|V\|$	P_{edge}	$\mathcal{X}(G)$	time (s)	#	time (s)	#	time (s)	#	time (s)	#
15	70	0.5	11-12	25.59	15	13.94	15	190.72	14	39.98	15
15	70	0.7	17-18	24.73	13	43.41	14	307.88	8	26.1	14
15	70	0.9	27-28	0.73	15	0.16	15	19.00	13	0.95	15

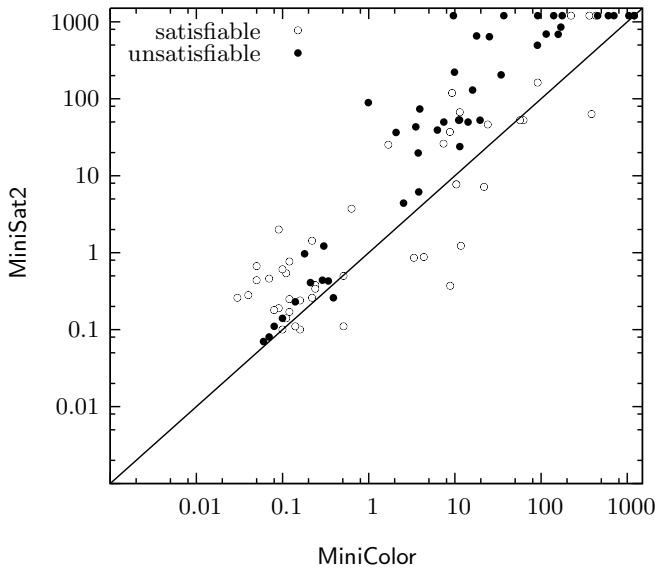

Fig. 3. Performance comparison between MiniColor and MiniSat2 on medium sized random graphs, with a 1200 (s) timeout.

4.2 DIMACS benchmarks

This experiment was executed to compare MiniColor to published results on graph coloring and to the unmodified MiniSat2 solver. As published benchmark performances we used the results published in [19] by Van Gelder which, to our knowledge, present the best broad overview of SAT based graph coloring results. Of the 27 graphs used in this benchmark set most are relatively easy. However, the five graphs presented in Table 2 were shown to be particularly difficult.

A comparison of MiniColor with best presented runtimes in [19], denoted by VanGelder and the runtimes of MiniSat2 on these graphs can be found in Table 3 and 4. For comparative purposes, we scaled the times presented in [19] to what they would have been if the instances were run on our platform[1].

Table 2. Difficult DIMACS instances.

instance	$\|V\|$	$\|E\|$	$\mathcal{X}$	found clique size
`Myciel6`	95	755	7	2
`Myciel7`	191	2360	8	2
`abb313GPIA`	1557	53356	9	6
`DSJC125.5`	125	3891	?	10
`DSJC125.9`	125	6961	?	33

Table 3. Runtimes on difficult satisfiable DIMACS in seconds.

	SAT instances			
instance	k	VanGelder	MiniSat2	MiniColor
`Myciel6`	7	0	0	0.01
`abb313GPIA`	9	1256	3.63	1.89
`DSJC125.5`	19	4446	43.46	18.51
`DSJC125.9`	46	19119	140	16.73

Table 4. Runtimes on difficult unsatisfiable DIMACS in seconds.

	UNSAT instances			
instance	k	VanGelder	MiniSat2	MiniColor
`Myciel6`	6	2113	3096	1726
`abb313GPIA`	8	5.63	0.73	0.72
`DSJC125.5`	12	488	5.85	4.08
`DSJC125.9`	37	4630	934	53.06

[1] The `dfmax` benchmark takes 12s for `r500.5.b` on our platform compared to 16.96 in [19]. For more information of the `dfmax` benchmark, please check [22].

As can be seen in Table 3 and 4, the runtimes of our implementation are vast improvements over the runtimes of MiniSat2 and those presented in [19]. After these encouraging results, we tried how our implementation would handle more difficult coloring of these graphs. As it turned out we could prove that `Myciel7` is not 6 colorable, `DSJC125.5` is not 13 colorable and `DSJC125.9` is not 38 colorable within reasonable time. The corresponding runtimes are shown in Table 5.

Table 5. Runtimes of MiniColor and MiniSat2 on harder versions of the DIMACS instances.

	SAT instances			UNSAT instances		
instance	k	MiniSat2	MiniColor	k	MiniSat2	MiniColor
`Myciel7`	8	0	0.03	6	6497	1381
`DSJC125.5`	18	> 19000	> 19000	13	> 19000	2931
`DSJC125.9`	45	> 19000	1008	38	> 19000	4683

5 Conclusions and Future Research

We showed how a SAT conflict-driven solver can be optimized for graph coloring problems by converting conflict clauses in such a way that they cover all permutations of the colors. This technique can be used in combination with other optimizations for graph coloring such as adding symmetry breaking predicates. In fact, the best performances are achieved by this combination.

We introduce new Boolean variables during the search. Although very powerful in theory, it is hardly used in practice. Regarding its practical application, we learnt two lessons. First, the introduced variables should be meaningful within the context of the problem – in this case, the branches in the Zykov-tree. Second, reusage of introduced (merge) variables is crucial. Recall that in each conversion step one can choose from many merge variables. Yet, heuristics that try to minimize the number of introduced variables were required to make the technique competitive.

Although the proposed technique is, as presented, only applicable to graph coloring problems, we have reason to believe that it can be generalized. Many multi-valued SAT problems seem fit for this purpose. In particular those consisting of constraints in which variables should either have the same value or a different one. Examples of such applications are computing Van der Waerden numbers and Schur numbers. The usefulness of our ideas will depend on whether they can be generalized successfully.

Acknowledgements

The authors thank the anonymous reviewers for their valuable comments that helped improving this paper.

References

1. Fadi A. Aloul, Karem A. Sakallah, and Igor L. Markov, *Efficient Symmetry Breaking for Boolean Satisfiability*. International Joint Conference on Artificial Intelligence (IJCAI), pp. 271–282, 2003.
2. Carlos Ansótegui, Jose Larrubia, Chu Min Li, and Felip Manyà, *Exploiting multivalued knowledge in variable selection heuristics for SAT solvers*. Annals of Mathematics and Artificial Intelligence **49**(1-4):191–205, 2007.
3. Stephen A. Cook, *A short proof of the pigeonhole principle using extended resolution. SIGACT News*. SIGACT News **8**(4):28–32, 1976.
4. Stephen A. Cook, *Feasibly constructive proofs and the propositional calculus*. In proceedings of STOC '75. pp. 83–97, 1975.
5. Martin Davis, G. Logemann, and D. Loveland, *A machine program for theorem proving*. Communications of the ACM, 5(7) pp. 394-397, July 1962.
6. Martin Davis and Hilary Putnam, *A Computing Procedure for Quantification Theory*. Journal of the ACM **7**(3):201–215, 1960.
7. Niklas Eén and Niklas Sörensson, *An Extensible SAT-solver*. In proceedings of SAT 2003, LNCS **2919**:502–518, 2003.
8. Ian P. Gent. *Arc Consistency in SAT*. In Proceedings of the Fifteenth European Conference on Artificial Intelligence (ECAI 2002), 2002.
9. Carla P. Gomes and David B. Shmoys, *Completing Quasigroups or Latin Squares: A Structured Graph Coloring Problem*. In proceedings of the Computational Symposium on Graph Coloring and Generalizations, pp. 22-39, Ithaca, USA, 2002.
10. Alexander Keur, Coen Stevens, and Mark Voortman, *Symmetry Breaking Options in Conflict Driven SAT Solving*. TU-delft technical report. Availiable at `http://www.st.ewi.tudelft.nl/sat/reports.php`
11. Joao P. Marques-Silva, Karem A. Sakallah, *GRASP – a new search algorithm for satisfiability*. In International Conference on Computer-Aided Design. pp. 220–227, 1996.
12. Matthew W. Moskewicz and Conor F. Madigan, *Chaff: engineering an efficient SAT solve*. In proceedings of DAC 2001. pp. 530–535, 2001.
13. David A. Plaisted and Steven Greenbaum *A structure-preserving clause form translation*. Journal of Symbolic Computation **2**(3):293–304, 1986.
14. Steven Prestwich, *Local Search on SAT-Encoded Colouring Problems*. In proceedings of SAT 2004. pp. 26–29, 2004.
15. Karem A. Sakallah, *Symmetry and Satisfiability*. Chapter 10 of Handbook of Satisfiability, pp 289–338, 2009.
16. Carsten Sinz and Armin Biere, *Extended Resolution Proofs for Conjoining BDDs*. In Proc. of CSR06, LNCS **3967**:600-611, 2006.
17. G. Tseitin, *On the complexity of derivation in propositional calculus*. Studies in Mathematics and Mathematical Logic, Part II. pp. 115-125, 1968.
18. Alasdair Urquhart, *Hard examples for resolution*. Journal of the ACM **34**(1):209–219, 1987.
19. Allen Van Gelder, *Another look at graph coloring via propositional satisfiability*. Discrete Applied Mathematics **156**(2):230–243, 2008.
20. A. A. Zykov, *On some properties of linear complexes*. Amer. Math. Soc.Translations 79 (1952), p. 81.
21. `http://mat.gsia.cmu.edu/COLOR/solvers/trick.c`
22. Computational Series: Graph Coloring and Its Generalizations. `http://mat.gsia.cmu.edu/COLOR04`.