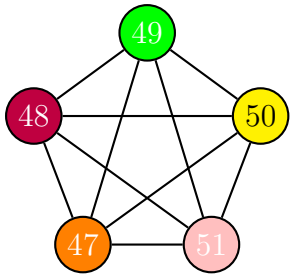
Delft University of Technology

MiniMerge

Symmetry-Free Learning in Combinatorial Problems



Bas Schaafsma

May 29, 2009

Committee:
dr. H. van Maaren
dr.ir. M.J.H. Heule
prof.dr. C. Witteveen
ir. H.J.A.M. Geers

MiniMerge

Symmetry-Free Learning in Combinatorial Problems

Master thesis by Bas Schaafsma, studentnr: 1150553.

Committee:
dr. H. van Maaren
dr.ir. M.J.H. Heule
prof.dr. C. Witteveen
ir. H.J.A.M. Geers

Acknowledgements

Dear Reader,

Before you start reading this thesis, allow me to express my thanks to my supervisors Marijn Heule and Hans van Maaren. Their positive support has been invaluable during my research and their input has been critical in order to keep me focused on the big picture and not get lost in details. I would also like to especially thank my daily supervisor Marijn for his patience in reading and criticizing my drafts, even if I sent them on short notice and his help in learning me to write a proper scientific article.

Bas Schaafsma

Abstract

We present a new method to break symmetry in graph coloring problems, which can be applied in many classes of combinatorial 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 combines Extended Resolution with domain specific knowledge. Although the Extended Resolution proof system is powerful in theory, 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 as much as possible.

We implemented our technique in the state-of-the-art solver MiniSat2. 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.

Contents

Chapter 1

Introduction

Satisfiability (SAT) solvers have become very powerful in recent years. Especially conflict-driven clause learning SAT solvers can effective tackle 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 decreased effectiveness reduces the performance of conflict-driven solvers on many classes of combinatorial problems, such as graph coloring which are known for their highly symmetric SAT encodings.

Fortunately, symmetry can be broken *statically*, as a preprocessing step, or *dynamically*, during the search. A frequently used static technique for graph coloring is to assign a different color to all vertices in a large clique [20]. Although quite effective and cheap (a large clique is easy to find), it only breaks the symmetry partially [20]. Even state-of-the-art static symmetry calculating programs can only break symmetries partially [16]. A dynamic symmetry breaking technique [10] adds, besides the conflict clause expressing the conflict, all symmetric conflict clauses. Yet, in case of many colors, the number of symmetric conflict clauses will increase dramatically. 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. One branch explores the search space by merging these vertices, while the other branch examines the space created by placing an edge between them. We translate 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.

Although the primary focus of the research presented in this thesis is improving the performance of conflict-driven solvers on graph coloring instances, the underlying principles of the Zykov algorithm can easily be abstracted and applied to many other classes of combinatorial problems such as computing Van der Waerden numbers [29].

The outline of the rest this thesis is as follows: Chapter 2 introduces the satisfiability problem and current solving techniques. Chapter 3 introduces the graph coloring problem and presents a survey of non-satisfiability approaches presented in the literature to solve the problem. Chapter 4 discusses solving a graph coloring instance as a satisfiability instance. Extended Resolution and it's application to the graph coloring problem is discussed in Chapter 5. Embedding the proposed

conversion of conflict clauses is explained in Chapter 6. Chapter 7 introduces Van der Waerden numbers as an example of how symmetry-free clauses can be used outside the scope of the graph coloring problem. Chapter 8 describes the implementation of MiniMerge our Minisat2 implementation which uses symmetry-free conflict clauses to store conflicts. Chapter 9 discusses MiniMerge2, which uses an experimental method to reduce time spend propagating assignments. Chapter 10 offers experimental results which compare the performance of MiniMerge to the performance of MiniSat2 on graph coloring instances. Chapter 11 evaluates the performance between MiniMerge and MiniMerge2. Finally, in Section 12 we draw some conclusions and provide suggestions for future research.

Chapter 2

The satisfiability problem

The satisfiability (SAT) problem 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 the conjunction of clauses C_i, where 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 must satisfy all clauses. The SAT problem has been shown to be $\mathcal{NP}$-complete [2].

1. Formula simplification and iterative unit propagation

Consider the following example SAT instance $\mathcal{F}_{\text{example}}$:

$$\mathcal{F}_{\text{example}} = C_1 \wedge C_2 \wedge C_3 \wedge C_4$$

$$\begin{aligned} C_1 &= (x_1) \\ C_2 &= (x_1 \vee \neg x_2) \\ C_3 &= (\neg x_1 \vee \neg x_2) \\ C_4 &= (\neg x_1 \vee x_2 \vee x_3 \vee x_4) \end{aligned}$$

Obviously if $\mathcal{F}_{\text{example}}$ is to be satisfied, x_1 must be assigned `true`, since this is the only way to satisfy the *unit clause* (a clause of size one) C_1. Once we have assigned $x_1 =$ `true` (x_1 for short) we can simplify $\mathcal{F}_{\text{example}}$: clauses C_1 and C_2, can be removed from $\mathcal{F}_{\text{example}}$ since they are already satisfied, the literal $\neg x_1$ can be removed from clauses C_3 and C_4, since these clauses can no longer be satisfied by those literals. The simplification of $\mathcal{F}$ after an assignment is referred to as the *propagation* of that assignment. Propagating x_1 results in:

$$\mathcal{F}_{\text{example}} \cup \{x_1\} = (\neg x_2) \wedge (x_2 \vee x_3 \vee x_4)$$

Apparently, the propagation of x_1 caused the clause C_3 to become unit and propagate $\neg x_2$. A clause that propagates a value, such as C_3, is referred to as the *reason clause* of that assignment. After propagating $\neg x_2$, $\mathcal{F}_{\text{example}}$ becomes:

$$\mathcal{F}_{\text{example}} \cup \{x_1, \neg x_2\} = (x_3 \vee x_4)$$

After this propagation there are no more unit clauses that force assignments. This process of satisfying unit clauses, until there are no more unit clauses is known as iterative unit propagation.

Besides unit propagation, formula simplification can also be done by the the removal of tautological and subsumed clauses. Clause C_i is tautological when it contains both a positive and negative

literal x_i. Since C_i is always always satisfied it can be removed from $\mathcal{F}$. Clause C_i is subsumed by clause C_j, if C_j consists of a subset of literals contained in C_i. Since any assignment which satisfies C_j also satisfies C_i, C_i can be removed from $\mathcal{F}$.

2. The DPLL procedure

To prove that a given formula $\mathcal{F}$ is satisfiable requires a satisfying assignment, or model, of the formula. Proving the unsatisfiability of a formula must be done through a refutation. In most modern SAT solvers the search for either a refutation or a model is done by variations of the DPLL procedure [6]. The 'pure' DPLL procedure is a depth-first backtracking procedure, it first applies iterative unit propagation, then it picks and assigns a free variable x_i and propagates that assignment (branching). This process is repeated until no clauses remain in $\mathcal{F}$ indicating that we have found a solution, or when an empty clause is detected. When an empty clause, is detected it means that $\mathcal{F}$ cannot be satisfied under the current assignment and the procedure backtracks and tries another assignment of x_i. The pseudo code of the DPLL procedure is presented in Algorithm 2.1.

Algorithm 2.1 DPLL($\mathcal{F}$)

```
1: F ← propagate(F) /*apply iterative unit propagation*/
2: if F contains no more clauses then
3:    return SAT
4: else if F contains an empty clause then
5:    return UNSAT
6: end if
7: x ← pickBranchVariable() /*choose the next variable to branch on*/
8: if DPLL(F ∪ x) = UNSAT then
9:    return DPLL(F ∪ ¬x)
10: end if
11: return SAT
```

3. Lookahead Solving

There are two main approaches used by SAT solvers these days. The first one, the lookahead approach is an almost direct implementation of DPLL procedure. This approach aims to minimize runtime by choosing the optimal branch variable in each step in the search tree. Obviously, the effectiveness of this approach depends heavily on the implementation of the *pickBranchVariable* procedure, which uses a procedure called *lookahead* to determine the most successful branch variable.

This lookahead procedure evaluates for every variable x_i in $\mathcal{F}$ both $propagate(\mathcal{F} \cup \{x_i\})$ and $propagate(\mathcal{F} \cup \{\neg x_i\})$. If the lookahead on either the positive or the negative literal results in a conflict, this literal is called a failed literal. This literal needs to be fixed on complement of its current assignment. When the lookahead on both x_i and $\neg x_i$ results in a conflict, then $\mathcal{F}$ is unsatisfiable under that assignment. When both lookaheads on a variable do not result in a conflict, the lookahead will be evaluated and a value will be assigned to the literal, which indicates its potential. This value can be based on a number of different heuristics, such as the number of created binary clauses by this assignment. As the lookahead procedure is computationally heavy, lookahead is therefore usually only performed on a subset of the literals in $\mathcal{F}$. The lookahead aproach is used in successful solvers such as `March_dl` [32].

4. Conflict-Driven Solving

The second main approach and currently the most popular variant of the DPLL procedure is known as conflict-driven solving, first presented in [11] and currently used in successful solvers such as MiniSat2 [8]. Instead of simply backtracking when a conflict is detected, this variant uses the *analyze* procedure to determine the source and level -in the search tree- of the current conflict. After determining this, a learnt conflict clause C_{conflict} is added to $\mathcal{F}$, which represents the state of the solver when the conflict was encountered. Such a learnt clause C_{conflict} is said to be implied by $\mathcal{F}$. The procedure then backtracks to the level of the current conflict, after which the learnt clause becomes an unit clause and is propagated and the process of assigning variables continues. When a conflict is detected at level 0 this means that every possible assignment of the variables in $\mathcal{F}$ results in UNSAT and hence $\mathcal{F}$ is UNSAT. The pseudo code of this variant is presented in Algorithm 2.2.

Algorithm 2.2 ConflictDrivenDPLL($\mathcal{F}$)

```
1:  while true do
2:    F ← propagate(F) /*apply iterative unit propagation*/
3:    if not conflict then
4:      if all variables assigned then
5:        return SAT
6:      end if
7:      decide() /*pick a new variable and assign it*/
8:    else
9:      C_conflict ← analyze() /*analyze conflict and add a conflict clause*/
10:     if top level conflict found then
11:       return UNSAT
12:     end if
13:     F ← backtrack(C_conflict) /*undo assignments until conflict clause is unit*/
14:   end if
15: end while
```

4.1 Conflict clause construction

Conflict clauses are constructed through the use of an implication graph, which reconstructs the reason for the encountered conflict. An implication graph is a directed acyclic graph, in which a vertex either corresponds to the current assignment of a variable or to the empty clause which caused the current conflict. The incident edges of each vertex comprise the reason that lead to that assignment or conflict. Vertices that have no incident edges are referred to as decision variables.

In an implication graph, vertex v is said to dominate vertex w iff any path from the decision variables of the level of v to w needs to go through v. An Unique Implication Point(UIP) is a vertex at the current decision level that dominates the vertex corresponding to the conflict [24]. A reason for the current conflict is defined as a bipartite cut in the implication graph which has all decision variables and one UIP on one side (called reason side), and the conflict in the other side (the conflict side). The vertices on the reason side that have at least one edge to the conflict side, the *reason set*, comprise the assignments which led to current the conflict [24]. The corresponding conflict clause is the negation of that assignment.

Since an implication graph may contain multiple UIPs there are multiple cuts possible, each resulting in a different conflict clause. Currently the strategy of choosing the UIP closest to the conflict, known as the 1-UIP strategy, is considered on average to be the most robust strategy for choosing UIPs, but other strategies, such as choosing the last IUP (LAST-UIP), can have better results in specific cases [24].

Figure 2.1 provides an example of a conflict, which occurred while solving $\mathcal{F}_{\text{uip}}$, and the corresponding implication graph.

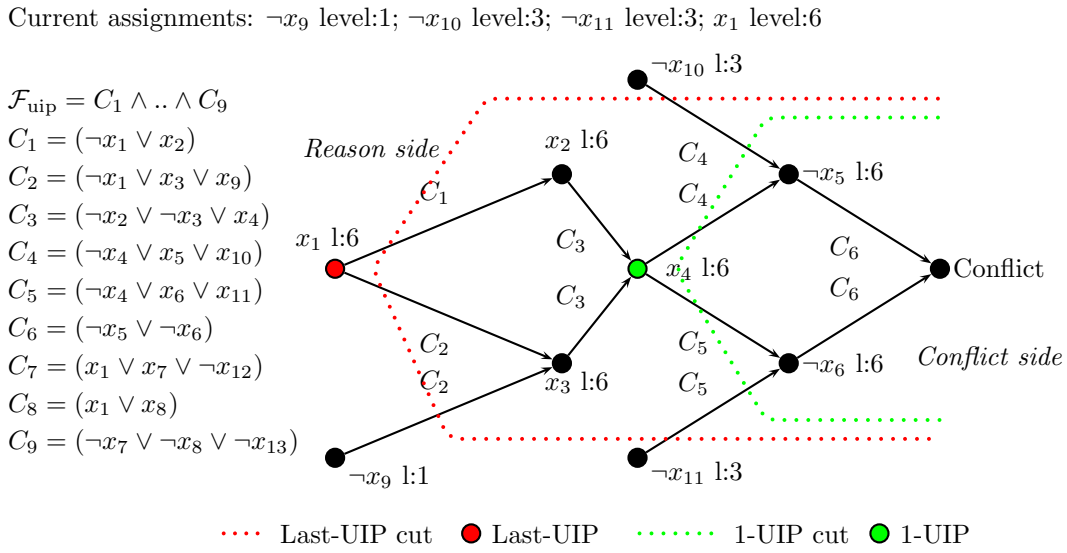


Figure 2.1. An implication graph example.

In this example, if the solver uses the 1-UIP cut, the *analyze* procedure would return the clause:

$$(\neg x_4 \vee x_{10} \vee x_{11})$$

On the other hand if the LAST-UIP cut would be used the procedure would return:

$$(\neg x_6 \vee x_{10} \vee x_{11} \vee x_9)$$

4.2 The VSIDS activity heuristic

Unlike the lookahead approach which spends considerable time on calculating the optimal branch variable, most conflict-driven solvers use a very simple heuristic, known as the Variable State Independent Decaying Sum (VSIDS) heuristic [12]. In the VSIDS heuristic an activity counter is kept for each variable x_i. This counter is increased each time x_i is present in the implication graph during the conflict analysis. Periodically all counters are multiplied with a constant between 0 and 1, thus decaying the activity of variables over time. When a new branch variable is needed the variable with the highest counter is selected.

4.3 Assignment propagation

The majority of time during solving is spend propagating assignments, therefore a good *propagate* implementation is key to an efficient SAT Solver [12]. In most modern conflict-driven solvers assignment propagation is done through a lazy algorithm known as the 'two watched literals' algorithm [12]. For each clause C_i we have two pointers *left* and *right* that point to two literals x_{left} and x_{right}, which either evaluate to `true` or are unassigned. When either x_{left} or x_{right} becomes falsified a new literal which is either satisfied or still free must be found to point to. If such a literal cannot be found we know that C_i can only be satisfied by the other watched literal and thus that assignment must be propagated. Another important property of this algorithm is that no pointers need to be updated during backtracking, making backtracking relatively cheap. A detailed example of the two

watched literals algorithm is depicted in Figure 2.2, which shows how the watched literal algorithm behaves with respect to a single clause $(\neg x_1 \vee x_4 \vee \neg x_7 \vee x_{12} \vee x_{15})$ under a number of assignments.

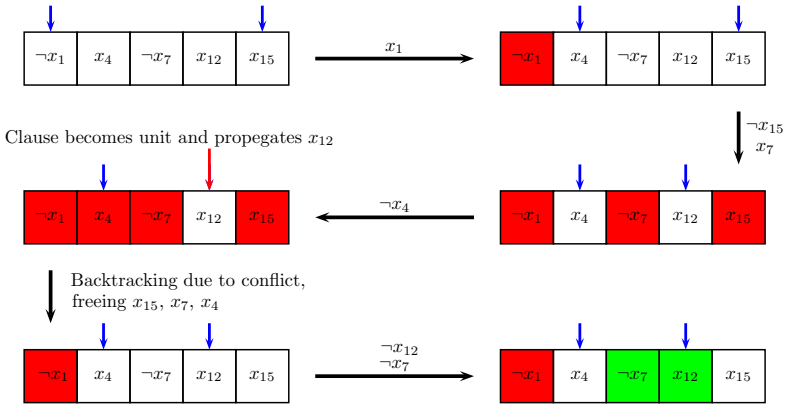


Figure 2.2. Two watched literals example, black arrows indicate successive assignments of variables, watched literals are represented by blue arrows, which turn red when propagating an assignment. Falsified literals are colored red, satisfied literals are colored green and free literals are colored white.

4.4 Optimizing techniques used in conflict-driven solving

- *Conflict clause minimization.* It is possible that a learnt clause contains redundant literals, which can be removed. A literal x_i in clause C_{conflict} is redundant, when $\neg x_i$ is implied by the negation of the other literals in C_{conflict}, as C_{conflict} can never be satisfied solely by x_i. The removal of redundant literals has been shown to significantly shorten conflict clauses which in turn reduces the time spend looking for new watch literals when a literal is falsified. Improving the performance of the *propagate* procedure.

- *Conflict clause deletion.* Obviously, conflict clauses are essential in the conflict-driven approach, but each added clause slows the propagating of assignments, while the clause may never become unit again. Therefore all state-of-the-art conflict-driven solvers periodically delete conflict clauses. Criteria for deleting such clauses vary from solver to solver, but usually the heuristic is based on the "relevance" of the clause (i.e. how often it became unit in a given time span).

- *Solver restarts.* It is a well established principle that while a SAT problem maybe easy, a solver can get stuck in a difficult part of the problem, which may have no bearing on a simple refutation which exists for the problem or is solved quickly when the simpler part of the solution has been found. To counter this problem conflict-driven solvers employ a restart strategy. After a preset number of conflicts, the solver unassigns all variables and starts in another part of the problem. Learnt clauses are kept to ensure that excluded search space is not visited. Completeness (the guarantee that it will return an answer) of the algorithm is ensured by iteratively increasing the the restart threshold after a restart. Like clause deletion the actual strategies for restarting vary from solver to solver.

Chapter 3

The graph coloring problem

The graph coloring problem (GCP) deals with the question, given a graph G what is the lowest number of colors such that the vertices of G can be colored sp that two connected vertices are differently colored. For instance the graph presented in Figure 3.1 can be colored using 3 colors, but not using 2 colors.

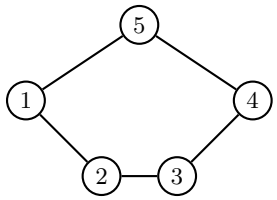


Figure 3.1. A graph that is 3-colorable but not 2-colorable. The numbers in the vertices refer to their index v_i.

More formally , a graph $G = (V, E)$ is k-colorable, when there exists a mapping φ_{color} from each vertex $v \in V$ onto an integer $\{1 \,..\, k\}$ such that for every $(v, w) \in E$, $\varphi_{\text{color}}(v) \neq \varphi_{\text{color}}(w)$. When φ_{color} does not map each $v \in V$, then we refer to φ_{color} as a partial coloring of G.

The smallest k for which G is still k-colorable is known as the chromatic number of G or $\mathcal{X}(G)$. Like the SAT problem, the GCP has been shown to be $\mathcal{NP}$-complete [2].

Although this thesis focuses on solving GCP instances through a SAT approach, there exist many other possible approaches, such as contraction algorithms, sequential algorithms and Integer Programming which we will discuss in this chapter. First however, we will discusses how to efficiently obtain a lower bound for the chromatic number of G through finding a *clique* in G.

1. Cliques as Lower Bounds

A fully connected subgraph of G containing n vertices is referred to as a clique of size n. Since each vertex in a clique must be colored differently, cliques of size n provides a solid proof of a lower bound of n for the chromatic number of G and a good starting point in a search for a n-coloring of G.

As useful as cliques are, it should be noted that there exists no relation between the largest clique in G and the chromatic number of G. For example, k-Mycielski graphs have a chromatic number of $k + 1$, while only having cliques of size 2 [40].

Although the problem whether a graph contains a clique of size n has been proven to be $\mathcal{NP}$-complete [2], there still exists effective algorithms to find reasonably large cliques in graphs. In our experiments we used, unless stated otherwise, a recursive clique finding algorithm implemented by Trick and available at [22]. The pseudo code of this algorithm is presented in Algorithm 3.1 and 3.2.

This algorithm uses a basic greedy algorithm to find a clique in G, it then tries to improve on this clique by choosing a vertex with maximum degree (the number of connected vertices to this vertex) not included in the returned clique and tries to find a clique that contains that vertex, which is bigger than the previously found clique. This process is repeated until the largest clique has been found or the number of tries exceeds a stated number of maximum allowable tries. The first procedure call of this algorithm is made as $maximum\_weighted\_clique(G, 1, size(G))$, the global variables $tries$ and $maximum\_number\_of\_tries$ are set to 0 and 10000 respectively.

In our experience the performance of this algorithm is moderate to good, execution of the algorithms takes only one or two seconds. We have done no extensive experiments on the quality of the cliques it returns, but we believe it is reasonable. For example, for one random graph with 125 nodes and 6961 edges it returned a clique of 27 vertices, while the largest clique in the graph has size 33 vertices.

Algorithm 3.1 GreedyClique(G)

add the vertex with the largest degree to the chosen clique.
while G contains vertices that are connected to every vertex in the chosen clique **do**
    add the vertex with the largest degree to the clique.
end while
return chosen clique.

Algorithm 3.2 maximum_weighted_clique(G, lower, target)

if ($size(G) <$ target) **OR** (tries $>$ maximum_number_of_tries) **then**
    return $\emptyset$ /* return the empty graph */
end if
$tries \leftarrow tries + 1$
$G_{\text{found\_clique}} \leftarrow GreedyClique(G)$
if $size(G_{\text{found\_clique}}) \geq target$ **then**
    return $G_{\text{found\_clique}}$
end if
sort all $v_i \in G$ by vertex degree
for all $v_i \in G$ and $\notin G_{\text{found\_clique}}$ **do**
    $G' \leftarrow$ the subgraph of G which contains v_i and all vertices connected to v_i
    $G_{\text{candidate\_clique}} \leftarrow maximum\_weighted\_clique(G', size(G_{\text{found\_clique}}), \text{target - 1})$
    if $size(G_{\text{candidate\_clique}}) + 1) > (size(G_{\text{found\_clique}}))$ **then**
        $G_{\text{found\_clique}} \leftarrow G_{\text{candidate\_clique}} \cup v_i$
    end if
end for
return $G_{\text{found\_clique}}$

2. Contraction algorithms

The concept of contraction algorithms is based on a theorem due to Zykov [21], which states:

$$\mathcal{X}(G) = min(\mathcal{X}(G/(v,w)), \mathcal{X}(G+(v,w))) \tag{3.1}$$

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 3.2.

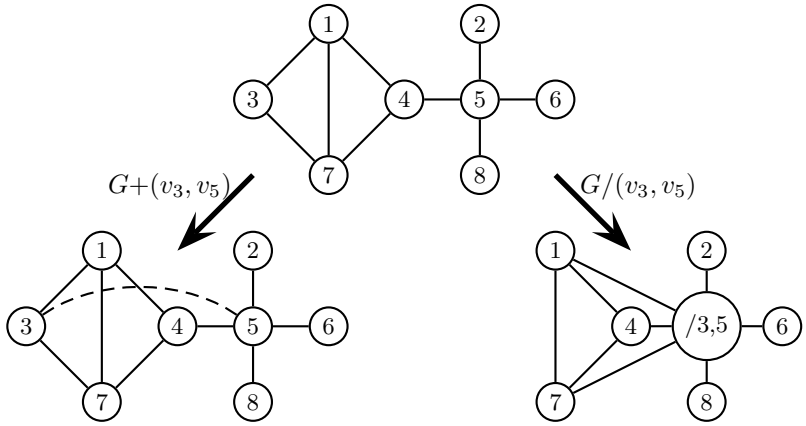


Figure 3.2. A Zykov-tree example. The numbers in the vertices refer to their index v_i. The added edge is shown as a dashed line.

Repeated steps of applying this theorem to a graph G result in a binary tree, known as the *Zykov-tree*. The leaves of this tree are fully connected graphs, which each have a chromatic number equal to their number of vertices. For example we can prove that the chromatic number of the graph in Figure 3.2 is at most 4 by exploring the follow branch of the Zykov-tree:

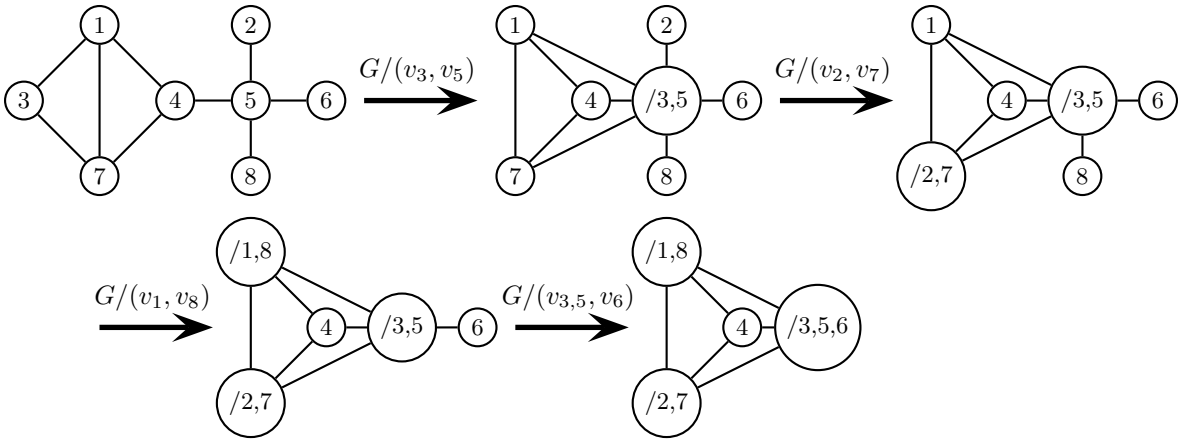


Figure 3.3. A branch of the Zykov-tree, which proves that the chromatic number of the graph presented in Figure 3.2 is at most 4.

2.1 Approximation contraction algorithms

Approximation contraction algorithms explore a single branch on the Zykov-tree, as shown in Figure 3.3 and return the number of vertices in the resulting leaf as an approximation of $\mathcal{X}(G)$. The choices

for the vertices to be merged are made on the observation that the number of vertices left in G is the upper bound of their approximation of $\mathcal{X}(G)$ and each merge reduces the number of vertices in G with one, therefore each merge should result in a graph containing as few edges as possible, which is the graph most likely to allow for the largest number of merges. The difference between the number of edges in G and the number of edges in $G/(v, w)$ is known as the common neighbors of v and w or $cn(v, w)$. For example in Figure 3.2, $cn(v_1, v_2) = 0$, $cn(v_2, v_4) = 1$ and $cn(v_3, v_4) = 2$

In the algorithm of Brigham and Dutton [27] $cn(v, w)$ is calculated for every pair of vertices $v, w \in V$, then iteratively two vertices v, w, with max $cn(v, w)$ are merged and the affected neighborhood scores updated, until the graph is fully connected.

In the better performing RLF (Recursive-Largest-First) algorithm of Leighton [28], after calculating $cn(v, w)$ for every pair of vertices $v, w \in V$ a vertex v with maximum degree is chosen and recursively merged with a vertex w with max $cn(v, w)$ and the affected neighborhood scores updated, until every vertex in G is connected with v at which point a new v is chosen, until the graph is fully connected.

2.2 Exact contraction algorithms

Exact contraction algorithms explore the entire Zykov-tree and return the exact chromatic number of G [31]. Obviously such algorithms cannot explore the entire tree as it is of super-exponential size. The search space is kept manageable by using a technique known as branch & bound. In the branch & bound method the search space is recursively divided into smaller subproblems (Branching). Subproblems for which we know that they will not yield an optimal solution are not considered (Bounding).

As stated, branching in the Zykov-tree is done by selecting, two vertices v, w and creating two new subproblems, one in which v and w are merged and one in which an edge is placed between v and w. For example Figure 3.2 shows the two resulting subproblems from a branch on the pair of vertices (v_3, v_4)

Bounding in the Zykov-tree is achieved by keeping track of an established upper bound k and discarding subproblems, which have a lower bound $\leq k$. A value of k for this upper bound represents a found leaf containing a fully connected graph containing k vertices. Thus the chromatic number of G is at most k and subproblems which contain a clique of size $k+1$ can be ignored. For example, the Zykov-tree branch explored in Figure 3.3 established an upper bound of 4 which means that branches which contain a clique of size 5 can be ignored. Detecting such cliques is non-trival and complicates the effectiveness of this algorithm.

2.3 Contraction algorithm performance

To our best knowledge, the family of contraction algorithms is currently considered to be the poorest performing family of coloring algorithms.

3. Sequential algorithms

3.1 The DSATUR algorithm

The DSATUR (Degree of Saturation) algorithm [26] is a sequential approximation algorithm which uses a dynamically established list of vertices. Given a partial coloring φ_{color}, the degree of saturation of vertex v, $degs(v)$, is equal to the number of differently colored neighbors of v. The DSATUR algorithm starts by assigning color 1 to vertex v with maximum degree. The vertex v to be colored next is the vertex with max $degs(v)$. If two vertices v, w have an equal degree of saturation, we chose the vertex with maximum degree. If v and w still have equal values, we choose randomly. The color used for v is the color with the lowest index that is available.

3.2 An exact backtracking algorithm

In [26] Brélaz proposed an exact sequential backtracking algorithm for the GCP. In this algorithm the vertices of G are stored in an array A. The order of the vertices in A is dynamically determined by the saturation degree of the vertices, vertices with equal saturation degree are ordered by vertex degree. When the vertices stored in $A[1]$ .. $A[n]$ have been colored by the partial coloring φ_{color}, the number of colors used in φ_{color} is defined as $color(n) = c_n$, the set of candidate colors $U[A[n+1]]$ for the vertex stored in $A[n+1]$ is then the subset of the colors $\{1, 2, .., c_n + 1\}$ which are not used to color vertices neighboring $A[n]$. At each iteration of the algorithm, an uncolored vertex with the largest saturation is colored. When an upperbound k for the chromatic number of G has been established, by a found coloring φ_{color}, all colors with an index $\geq k$ are removed from each U. If any U becomes empty, a backtrack must be performed. The pseudo code of this algorithm is shown in 3.3. M. Trick has a ready implementation of this algorithm available at [22]. Although currently considered to be sub-standard this algorithm is still often used to benchmark new algorithms.

Algorithm 3.3 ExactBacktrack(G)

find and color a clique of size r
$\varphi_{\text{return}} \leftarrow$ a coloring found using the DSATUR algorithm
$k \leftarrow colors(|V|)$
$start \leftarrow r + 1$
$v \leftarrow A[r + 1]$
$U[v] \leftarrow$ the non decreasing set of free colors for vertex v, which are $< k$
while $(start \geq r)$ **do**
  for $(i = start;\ i \leq |V|;\ i \leftarrow i + 1)$ **do**
    $v \leftarrow A[i]$
    if $(i > start)$ **then**
      $U[v] \leftarrow$ the non decreasing set of free colors for v, which are $< k$
    end if
    if $(|U[v]| > 0)$ **then**
      $\varphi_{\text{color}}(v) = U[v][0]$
      remove $U[v][0]$ from $U[v]$;
    else
      $start \leftarrow i - 1$
      $\varphi_{\text{color}}(A[start]) \leftarrow \emptyset$
    end if
  end for
  $k \leftarrow colors(|V|)$
  $\varphi_{\text{return}} \leftarrow \varphi_{\text{color}}$
  $i \leftarrow$ least i with $\varphi_{\text{color}}(A[i]) = \text{k}$
  $start \leftarrow i - 1$
  for $(i = start + 1;\ i \leq |V|;\ i \leftarrow i + 1)$ **do**
    $\varphi_{\text{color}}(A[i]) \leftarrow \emptyset$
  end for
  for $(i = 0;\ i \leq start;\ i \leftarrow i + 1)$ **do**
    remove from U[A[i]] all colors $\geq k$
  end for
end while
return φ_{return}

4. Solving graph coloring instances through Integer Programming

Like many other combinatorial problems, the GCP can be formulated as an Integer Programming (IP) problem. IP is a special case of Linear Programming (LP). LP problems are optimization problems which consists of an objective function and linear constraints. The objective function is defined as acquiring the maximum or the minimum of a specific function. The objective only holds under certain circumstances, which are represented in the problem constraints. Next to that, the variables are in most cases also bounded to non-negative values. In Linear Programming variables may be given any real value, in IP all variables must be given an integer value. The general problem LP has been proven to be in $\mathcal{P}$, while IP has been proven to be $\mathcal{NP}$-Complete.

Although a general discussion about the solving techniques for IP problems falls outside the scope of this thesis, we will discuss a number of IP encodings for the GCP as IP encodings and SAT encodings of the GCP are very similar and suffer from similar drawbacks. For a detailed discussion on IP solving techniques we refer the reader to [37].

Let $x_{v,i}$ be a binary variable for which $x_{v,i} = 1 \leftrightarrow \varphi_{\text{color}}(v) = i$ holds, w_j be a binary variable for which $w_j \leftrightarrow \varphi_{\text{color}}(v) = j$ for some v holds, then the traditional IP encoding of the GCP is:

$$Minimize \sum_{j=1}^{|V|} w_j \tag{3.2}$$

$$Subject\ to \sum_{j=1}^{|V|} x_{v,j} = 1 \qquad v \in V \tag{3.3}$$

$$x_{v,j} + x_{w,j} \leq w_i \qquad (v,w) \in E, \qquad 1 \leq j \leq |V| \tag{3.4}$$

$$x_{v,j} \in \{0,1\} \qquad v \in V, \qquad 1 \leq j \leq |V| \qquad w_j \in \{0,1\} \qquad 1 \leq j \leq |V| \tag{3.5}$$

Where (3.2) is the objective function to be minimized, (3.3) are constraints which encode that every vertex of G must be given at least one color and (3.4) are constraints which encode that no two neighboring vertices are given the same coloring and that vertices may only use colors which corresponding variable has been set to one.

Since colors in the GCP are interchangeable, $(k!)$ symmetrical colorings exist for k given number of colors. Since also feasible solutions of an IP graph coloring formulation suffer from that symmetry drawback, IP algorithms, based on the above formulation, tends to behave poorly (even for small instances of graph coloring). The main reason for that is the fact that many subproblems in the search tree have the same optimal value. [33].

In the literature we have found two solution to the above mentioned problem: The first solution is to extend the above representation with constraints which (partly) break these symmetries, for example in [34] it was proposed that the the sets of equally colored vertices should be sorted by the minimum index of the vertices belonging to each set and only colorings are considered that assigns color j to the j^{th} independent set. All other permutations that define the same coloring do not lead to a feasible solution. This can be achieved by adding the following constraints:

$$x_{v,j} = 0 \qquad j \geq v + 1 \tag{3.6}$$

$$x_{v,j} \leq \sum_{w=j-1}^{i-1} x_{w,j-1} \qquad v \in V \backslash v_1, \qquad 2 \leq j \leq v - 1 \tag{3.7}$$

Here constraint (3.7) encodes that vertex v_i, with $i > 1$ can only be colored j if a vertex v_x, with $x < i$ has already be colored $j - 1$.

There exists a fine line between the number of broken symmetries and the actual effectiveness of the symmetry breaking constraints. The above constraints are too computational heavy and are

not used in [34]. Instead they use constraints which specify that the number of vertices colored by j must be greater or equal than the number of vertices colored by $j+1$, which is enforced by adding the following constraints :

$$w_j \leq \sum_{v \in V} x_{v,j} \qquad 1 \leq j \leq |V| \tag{3.8}$$

$$\sum_{v=1}^{|V|} x_{v,j} \leq \sum_{v=1}^{|V|} x_{v,j+1} \qquad 1 \leq j \leq |V| - 1 \tag{3.9}$$

These constraints only partially break the symmetries that arise form color permutations, permutation between vertex sets that have the same size are still possible, but enough symmetries are broken to make these constraints effective while not being to computationally heavy.

A second approach is to use an encoding which does not exhibit the symmetries present in the traditional encoding. An interesting example of such an approach is presented in [25], where an encoding is used based on the notion of maximum independent sets.

An independent set of G is a set of vertices such that there is no edge in E connecting any of these vertices. A maximal independent set is an independent set that is not strictly included in any other indepent set. Any coloring of G is comprised of a set of independent sets, thus the chromatic number of G is equal to the minimum number of maximal independent sets which include every vertex in G. For example we need 3 maximum independent sets to completely cover the graph presented in Figure 3.4.

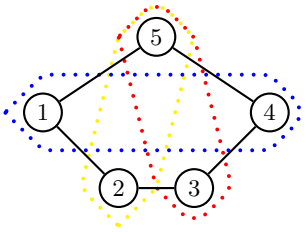


Figure 3.4. A complete coverage of G using three maximum independent sets. The numbers in the vertices refer to their index v_i.

A coverage of G by independent sets can be translated into a coloring of G by assigning a different color to each independent set, for vertices which are contained in multiple sets, such as v_5 in Figure 3.4, it suffices to chooses a color which corresponds to one of the sets they belong to.

Let S be the set of of all maximal independent sets in G, x_s be a binary variable with $x_s = 1 \leftrightarrow$ all vertices in the maximum independent set s are equally colored, then the Independent Set (IS) formulation of the GCP for G is:

$$Minimize \sum_{s} x_s \tag{3.10}$$

$$Subject\ to \sum_{s:i \in S} x_s \geq 1 \qquad i \in V \tag{3.11}$$

$$x_s \in \{0,1\} \quad s \in S \tag{3.12}$$

As any graph G can have an exponential number of maximum independent set, the number of variables in an IS formulation are far to great to add them all explicitly. Instead [25] uses a

technique known as column generation. In this technique variables are added at the point when they are required. To quote [25] as to how this is done with respect to (IS):

Begin with a subset S' of independent sets. Solve the linear relaxation (replace the integrality of constraints on x_s with nonnegativity) of (IS) restricted to $s \in S'$. This gives a feasible solution to the linear relaxation of (IS) and a dual value of π_i for each constraint in (IS). Now, determine if it would be useful to expand S'. This is done by solving the following maximum weighted independent set problem:

$$Maximize \sum_{v \in V} \pi_v z_v \tag{3.13}$$

$$Subject\ to\ z_v + z_w \leq 1 \qquad (v, w) \in E \tag{3.14}$$

$$z_v \in \{0, 1\} \qquad v \in V \tag{3.15}$$

If the optimal solution to this problem is more than 1, then the z_i with value 1 correspond to an independent set that should be added to S'. If the optimal value is less than or equal to 1, then there exist no improving independent sets: solving the linear relaxation of (IS) over the current S' is the same as solving it over S. Since the maximum weighted set problem itself is a difficult problem to solve the authors note that any set with size > 2 could be added if necessary.

4.1 Integer Programming algorithm performance

As far as we know IP algorithms are currently seen as the superior class of graph coloring algorithms [33]. However, the superiority of IP algorithms is not unchallenged.

For many benchmarks in the DIMACS benchmark set a SAT approach provides a better upper bound than the one presented in [33]. In [20] better upper bounds were presented for the benchmarks: `DSJC125_5`, `DSJC125_9` and `abb313GPIA`. Furthermore, in our own preliminary experiments we found that MiniSat2 provided better upper bounds for the benchmarks: `queen9_9`, `ash958GPIA` and any `le450` benchmark.

Establishing good lower bounds for a GCP instance using a SAT approach is more difficult as currently such an approach only supports partial symmetry breaking. However, by using our approach full symmetry breaking can be achieved, therefore we expected to find improved lower bounds for difficult DIMACS instances. In fact using our approach, we were able to calculate better lower bounds than presented in [33] for numerous benchmarks. Among these benchmarks are the `Mycielski` and the `DSJC125_5` benchmarks, for more details on the exact results we refer the reader to Chapter 10.

Furthermore the results presented in [33] indicate that the exact backtracking algorithm presented in Section 3.2 outperforms IP algorithms for graphs with small chromatic numbers. In our preliminary experiments MiniSat2 outperformed this backtracking algorithm on 3-coloring instances with 400 vertices. Extrapolating from the data presented in [33] it stands to reason that a SAT approach would outperform the IP approach for 3-coloring problems. This conclusion is further supported by the fact that during our literature surveys we did not find any detailed studies involving exact IP algorithms for 3-coloring instances with 400 or more vertices.

Chapter 4

Solving the graph coloring problem as SAT instances

The question whether a graph is k-colorable can be encoded as a SAT problem, therefore a GCP instance can be expressed as a series of SAT problems. Encoding the question whether a graph is k-colorable can be done through many different encodings. In the next section we will consider the most commonly used encodings discussed in [14, 20]. To illustrate these encodings we will use them to encode a graph containing two adjacent edges v and w and the colors $\{1, 2, 3\}$.

1. A survey of SAT encodings for the graph coloring problem

The direct encoding

In the direct encoding of a k-coloring problem a color variable $x_{v,i} \leftrightarrow \varphi_{\text{color}}(v) = i$. The direct encoding has three types of clauses. The property that each vertex must be colored with at least one color, is encoded by the *at-least-one* clauses, which are of the form:

$$(x_{v,1} \vee x_{v,2} \vee \cdots \vee x_{v,k}) \qquad v \in V \tag{4.1}$$

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

$$(\neg x_{v,i} \vee \neg x_{w,i}) \qquad (v, w) \in E, \quad 1 \leq i \leq k \tag{4.2}$$

The *at-most-one* clauses encode that each vertex can only be colored with one color:

$$(\neg x_{v,i} \vee \neg x_{v,j}) \qquad v \in V, \quad 1 \leq i < j \leq k \tag{4.3}$$

Encoding our example graph with the direct encoding results in:

$$\begin{array}{ccc} (x_{v,1} \vee x_{v,2} \vee x_{v,3}) & & (x_{w,1} \vee x_{w,2} \vee x_{w,3}) \\ \\ (\neg x_{v,1} \vee \neg x_{v,2}) & (\neg x_{v,1} \vee \neg x_{v,3}) & (\neg x_{v,2} \vee \neg x_{v,3}) \\ (\neg x_{w,1} \vee \neg x_{w,2}) & (\neg x_{w,1} \vee \neg x_{w,3}) & (\neg x_{w,2} \vee \neg x_{w,3}) \\ \\ (\neg x_{v,1} \vee \neg x_{w,1}) & (\neg x_{v,2} \vee \neg x_{w,2}) & (\neg x_{v,3} \vee \neg x_{w,3}) \end{array} \tag{4.4}$$

The multivalued encoding

The at-most-one clauses from the direct encoding are optional, in the multivalued encoding the at-most-one clause are omitted allowing each encoded vertex to be assigned more than one color

simultaneously. A coloring φ_{color} can be derived from a SAT solution by selecting any one $x_{v,i}$ assigned `true` for each vertex v. Encoding our example graph with the multivalued encoding results in:

$$\begin{array}{ccc} (x_{v,1} \vee x_{v,2} \vee x_{v,3}) & & (x_{w,1} \vee x_{w,2} \vee x_{w,3}) \\ (\neg x_{v,1} \vee \neg x_{w,1}) & (\neg x_{v,2} \vee \neg x_{w,2}) & (\neg x_{v,3} \vee \neg x_{w,3}) \end{array} \tag{4.5}$$

The log encoding

The log encoding aims to encode a graph coloring instance with as few SAT variables (logarithmic instead of linear) as possible. For each vertex $v \in V$ we add $\lceil \log_2(k) \rceil$ variables of the form x_v^i where $x_v^i = 1 \leftrightarrow$ bit i of the integer representation of $\varphi_{\text{color}}(v)$ is 1. For example in a 3-coloring instance assigning $\neg x_v^1,\ x_v^2$ encodes that $\varphi_{\text{color}}(v) = 2$. In the log encoding there are no at-least-one or at-most-one clauses, instead for each edge $(v, w) \in E$ k conflicting clauses are added that forbid combination of variables that encode that $\varphi_{\text{color}}(v) = \varphi_{\text{color}}(w)$. If k is not a power of two, prohibited-value clauses are added that prevent vertices being colored with an illegal coloring. Encoding our example graph with the multivalued encoding results in:

$$\begin{array}{ccccc} (x_v^1 \vee x_v^2 \vee x_w^1 \vee x_w^2) & (\neg x_v^1 \vee x_v^2 \vee \neg x_w^1 \vee x_w^2) & (x_v^1 \vee \neg x_v^2 \vee x_w^1 \vee \neg x_w^2) \\ (\neg x_v^1 \vee \neg x_v^2) & & (\neg x_w^1 \vee \neg x_w^2) \end{array} \tag{4.6}$$

The binary encoding

The log encoding without prohibited-value clauses is known as the binary encoding, Instead all bit combinations are allowed, each value encoded value $> k$ is interpreted as a value $\leq k$. For example $l > k$ can be mapped onto $l - k$. Obviously in the binary encoding contains no prohibited-value clauses, however the binary encoding may generate exponentially more clauses than the log encoding. Encoding our example graph wit the binary encoding results in:

$$\begin{array}{cc} (x_v^1 \vee x_v^2 \vee x_w^1 \vee x_w^2) & (\neg x_v^1 \vee \neg x_v^2 \vee \neg x_w^1 \vee \neg x_w^2) \\ (\neg x_v^1 \vee x_v^2 \vee \neg x_w^1 \vee x_w^2) & (x_v^1 \vee x_v^2 \vee \neg x_w^1 \vee \neg x_w^2) \\ (x_v^1 \vee \neg x_v^2 \vee x_w^1 \vee \neg x_w^2) & (\neg x_v^1 \vee \neg x_v^2 \vee x_w^1 \vee x_w^2) \end{array} \tag{4.7}$$

The support encoding

Like the direct encoding, the support encoding consists of the at-least-one and at-most-one clauses, however instead of conflicting conflicting clauses it uses supporting clauses. Which encode for each $(v, w) \in E$ that the value $\neg x_{w,i}$ must be propagated if $x_{v,i}$ would be propagated by an at-least-one clause. Encoding our example graph with the support encoding results in:

$$\begin{array}{ccc} (x_{v,1} \vee x_{v,2} \vee x_{v,3}) & & (x_{w,1} \vee x_{w,2} \vee x_{w,3}) \\ (\neg x_{v,1} \vee \neg x_{v,2}) & (\neg x_{v,1} \vee \neg x_{v,3}) & (\neg x_{v,2} \vee \neg x_{v,3}) \\ (\neg x_{w,1} \vee \neg x_{w,2}) & (\neg x_{w,1} \vee \neg x_{w,3}) & (\neg x_{w,2} \vee \neg x_{w,3}) \\ (x_{v,1} \vee x_{v,2} \vee \neg x_{w,3}) & & (x_{w,1} \vee x_{w,2} \vee \neg x_{v,3} \\ (x_{v,1} \vee x_{v,3} \vee \neg x_{w,2}) & & (x_{w,1} \vee x_{w,3} \vee \neg x_{v,2}) \\ (x_{v,2} \vee x_{v,3} \vee \neg x_{w,1}) & & (x_{w,2} \vee x_{w,3} \vee \neg x_{v,1}) \end{array} \tag{4.8}$$

In our experiments we focused on the direct encoding, as this encoding produces the best results for conflict-driven solvers [14, 20], yet our proposed technique can theoretically be extended to any presented encoding.

2. Symmetry breaking predicates in SAT encodings

Like the classic IP encoding, all colors are interchangeable in the direct encoding, making this encoding highly symmetric, thus greatly reducing the effectiveness of standard solving techniques on GCP instances. As far as we know, there is no SAT encoding for the GCP, like the Independent Set IP encoding, that does not contain such inherit symmetries. In the presence of such symmetry, it is good practice to add *symmetry breaking predicates* [16], which are additional clauses that break such symmetries, while not effecting the satisfiability of $\mathcal{F}$.

2.1 Clique forcing

In case of the GCP, a common technique for symmetry breaking is to search for a large clique and force all vertices in that clique to a different color – by adding unit clauses to the formula [20]. As stated, cliques in a graph can be easily detected thus making this a cheap and reasonable effective method. Unfortunately in most cases, this technique breaks the color symmetries only partially since:

1. The chromatic number of G may be higher than the largest clique in G, thus symmetries involving permutations of colors not present in the forced clique are not broken.

2. Symmetries involving color permutations over sets of vertices that don't contain vertices from the forced clique are not broken. This causes the technique to become less powerful as G contains more vertices and conflicts occur away from the forced clique.

It should be noted that clique forcing is good practice even if it only partially breaks symmetries or the symmetries are already broken some other way and should always be practiced. Rember, each vertex in a clique must be colored differently, thus adding unit clauses for a single clique does not change the satisfiability of $\mathcal{F}$, while it may significantly simplify $\mathcal{F}$.

2.2 Shatter

In the more general context of CNF formulae, the program Shatter [1] can be used to compute symmetry breaking predicates. Shatter is a CNF preprocessor based on the work of Crawford [35, 36], it computes the symmetries in $\mathcal{F}$ by reducing the calculation of symmetries in $\mathcal{F}$ to the graph automorphism problem.

Given two graphs, an isomorphism is a 1-to-1 mapping between the vertex sets of the two graphs that maps edges to edges. Given a graph, an automorphism is an isomorphism onto it self. The graph automorphism problem, is the problem given a graph G what are all automorphisms of G in terms of group generators [1]. Group generators are an important part of the mathematical field of Group Theory, which complex as it is falls outside the scope of this thesis, for more information about this field we refer the reader to [39].

Although the graph automorphism problem is in $\mathcal{NP}$ and currently no general polynomial algorithm has been found for this problem, it is not believed to be $\mathcal{NP}$-Complete and has many polynomial algorithms for bounded cases [38]. Therefore computing the symmetries of a CNF by mapping it onto a graph is computationally feasible in many cases. For na overview of possible mappings of a CNF to a graph, we refer the reader to [1, 16].

Since a CNF, such as our graph coloring instances, may have an exponential number of symmetries, full symmetry breaking using this approach is impractical [1]. In practice only breaking the symmetries encoded by the generators returned by the procedure which solves the graph automorphism problem achieves good results [16]. For an overview of possible translations from symmetry group generators to symmetry breaking predicates we refer the reader to [1, 16]

Chapter 5

Extended Resolution

It is a well known fact that any DPLL refutation can be polynomially translated into a resolution refutation, limiting the power of the DPLL procedure to the power of the resolution proof system. Resolution is a propositional proof system which consists of a single inference rule:

$$\frac{C_i \cup x \quad C_j \cup \neg x}{C_i \cup C_j} \tag{5.1}$$

A refutation in resolution is created, when the empty clause $\emptyset$ is derived using the above rule. For example using resolution we can refute:

$$\mathcal{F} = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2) \tag{5.2}$$

With the refutation:

$$\dfrac{\dfrac{x_1 \vee \neg x_2 \qquad x_1 \vee x_2}{x_1} \qquad \dfrac{\neg x_1 \vee \neg x_2 \qquad \neg x_1 \vee x_2}{\neg x_1}}{\emptyset}$$

The complexity of the length of resolution refutations has been shown to be exponential for many families of problems, even problems known to be in $\mathcal{P}$ [19], severely limiting the effectiveness of the DPLL procedure when applied to these problems.

Fortunately, proof systems more powerful than resolution exist. Of these proof systems, the *Extended Resolution* system (ER), proposed by Tseitin [18], is of particular interest to the SAT community, since ER is very similar to normal resolution, making it potentially easy to integrate in a DPLL based solver, while being much more powerful.

In ER the *extension rule* is added to the resolution system. This rule allows for the introduction of new clauses and variables to $\mathcal{F}$. These clauses and variables to $\mathcal{F}$ must express a relationship in the form:

$$e \leftrightarrow \mathcal{F}' \tag{5.3}$$

In this relationship e is a new variable not occurring in $\mathcal{F}$ and $\mathcal{F}'$ is an arbitrary Boolean formula of literals in $\mathcal{F}$. For instance, adding $e \leftrightarrow (x_1 \wedge x_2)$ is done by adding to $\mathcal{F}$:

$$(e \vee \neg x_1 \vee \neg x_2) \wedge (\neg e \vee x_1) \wedge (\neg e \vee x_2) \tag{5.4}$$

To our best knowledge, there are no SAT-solvers which use ER as a search strategy. This is due to the fact that there exist almost an infinite number of candidates for $\mathcal{F}'$ at any point in the DPLL search tree, if the extension rule should be used at all. Therefore, ER is currently mainly seen as a theoretical principle or used to compress found refutations of problems [17].

Yet, the power of ER is equal to the most powerful known proof systems [5]. Although the question whether there exists a short ER refutation for any unsatisfiable instance $\mathcal{F}$ is an open question,

ER has polynomial refutations for certain classes of problems where there only exist exponential refutations, using normal resolution. According to Cook [4], the key to creating such refutations seems to be finding constructive properties of the proof which can be simulated by ER.

1. ER example: The Pigeonhole Principle

One of the most well known classes of problems were there exists an short ER refutation, but no short resolution refutation is The Pigeonhole Principe (PHP). PHP is the assertion that there exists an one-on-one mapping φ of n onto $n-1$.

Cook [4] showed how a short ER refutation can be formulated. Informally the refutation of the assertion is given by assuming that if there exists an one-on-one mapping φ of n onto $n-1$, there also exists an one-on-one mapping φ' of $n-1$ onto $n-2$ of the form:

$$\varphi' = \begin{cases} \varphi(v) & \text{if} \quad \varphi(v) \neq n-1 \\ \varphi(n) & \text{if} \quad \varphi(v) = n-1 \end{cases} \qquad 1 \leq i \leq n-1 \tag{5.5}$$

This procedure will in the end assert the impossibility of an one-on-one mapping of 2 onto 1. The ER refutation of Cook follows the informal argument closely. The PHP assertion for n can be given using $x_{v,i} \leftrightarrow \varphi(v) = i$:

$$X_n \begin{cases} (x_{v,1} \quad \vee \quad x_{v,2} \vee \cdots \vee x_{v,n-1}) & 1 \leq v \leq n \\ (\neg x_{v,i} \quad \vee \quad \neg x_{w,i}) & 1 \leq v < w \leq n, \quad 1 \leq i \leq n-1 \end{cases} \tag{5.6}$$

The mapping of φ' can be described by introducing the following clauses, with the extension rule using $e_{i,k} \leftrightarrow \varphi'(i) = k$:

$$E_n \begin{cases} (e_{v,i} \quad \vee \quad \neg x_{v,i}) & \\ (e_{v,i} \quad \vee \quad \neg x_{n,i} \quad \vee \quad \neg x_{v,n-1}) & \\ (\neg e_{v,i} \quad \vee \quad x_{v,i} \quad \vee \quad x_{v,n-1}) & \\ (\neg e_{v,i} \quad \vee \quad x_{v,i} \quad \vee \quad x_{n,i}) & \end{cases} \qquad 1 \leq i \leq n-1, 1 \leq i \leq n-2 \tag{5.7}$$

Cook showed that from the union of E_n and X_n one can derive:

$$X_{n-1} \begin{cases} (e_{v,1} \quad \vee \quad e_{v,2} \vee \cdots \vee e_{v,n-2}) & 1 \leq v \leq n-1 \\ (\neg e_{v,i} \quad \vee \quad \neg e_{w,i}) & 1 \leq v < w \leq n-1, \quad 1 \leq i \leq n-2 \end{cases} \tag{5.8}$$

After $n-2$ dimension reduction steps, this results in X_2=$\{e^*_{1,1}$,$e^*_{2,1}$,$\neg e^*_{1,1} \vee \neg e^*_{2,1}\}$, from which the empty clause can be derived.

2. Emulating contraction algorithms through ER

ER can emulate a Zykov-step by introducing and setting a variable $e_{v,w}$ which expresses :

$$e_{v,w} \leftrightarrow \varphi_{\text{color}}(v) = \varphi_{\text{color}}(w) \tag{5.9}$$

This relation can be translated to CNF using the following *support clauses* -no relation to the support encoding for the graph coloring problem-:

$$(e_{v,w} \vee \neg x_{v,i} \vee \neg x_{w,i}) \wedge (\neg e_{v,w} \vee x_{v,i} \vee \neg x_{w,i}) \wedge (\neg e_{v,w} \vee \neg x_{v,i} \vee x_{w,i}) \quad 1 \leq i \leq k \tag{5.10}$$

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.

Chapter 6

Symmetry-free conflict clauses in graph coloring

A powerful application of emulating contraction lies in strengthening conflict clauses. To illustrate the benefits of emulating contraction, consider the example conflict, in the 3-coloring instance presented in Figure 6.1.

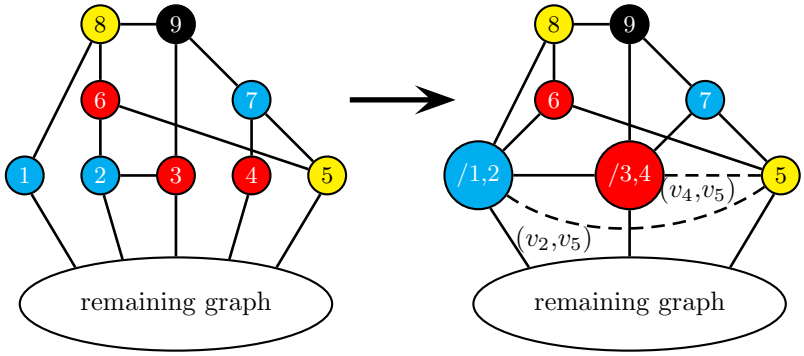


Figure 6.1. A Zykov contraction example. The numbers in the vertices refer to their index v_i. The added edges are shown as dashed lines. Vertex v_9 is in conflict because it cannot be colored. Although various reasons can be addressed for this conflict, the example focusses on the assignment to v_1, v_2, v_3, v_4, and v_5.

In the corresponding SAT instance, a conflict clause for this conflict would be:

$$(\neg x_{1,1} \vee \neg x_{2,1} \vee \neg x_{3,2} \vee \neg x_{4,2} \vee \neg x_{5,3}) \tag{6.1}$$

However, due to the inherit 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 implied by $\mathcal{F}$:

$$(\neg x_{1,3} \vee \neg x_{2,3} \vee \neg x_{3,1} \vee \neg x_{4,1} \vee \neg x_{5,2}) \tag{6.2}$$

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

1. Converting conflict clauses

Any conflict clause, which consists out of negative color literals, such as the clauses depicted in (6.1) and (6.2), encodes a conflicting coloring φ_{color} of a subset of vertices in G. This encoded coloring corresponds to a node in a Zykov-tree where all vertices which 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 6.1 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:

$$(\neg e_{1,2} \vee \neg e_{3,4} \vee e_{2,5} \vee e_{4,5}) \tag{6.3}$$

In any symmetry-free conflict 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. Unfortunately there exist many different combinations of merge literals which encode the necessary contractions. For example, encoding that vertices v_1, v_2 and v_3 should not be equally colored, can be done with $(\neg e_{1,2} \vee \neg e_{1,3})$ or $(\neg e_{1,2} \vee \neg e_{2,3})$. Different combinations result in different runtimes, therefore we need to select the "optimal" candidates. Heuristics for this selection are discussed in Section 1.

The positive literals C_{merge} correspond to the edges added to induce a clique. Again, in most cases, there are quite a lot of possible candidates from which we need to make a choice. We will use the same heuristics as for the negative merge literals. Of course no edges need to be added between contracted vertices v and w, if such an edge already exists in G.

Besides C_{merge}, one also needs to add the support clauses $M(C_{\text{merge}})$, which are the clauses required by the extension rule, to $\mathcal{F}$. Theoretically, for each introduced merge variable we could add the full set of clause described in Section 2. Yet, in practice it suffices to only add 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]. Formally, for any symmetry-free learnt clause C_{merge}, $M(C_{\text{merge}})$ equals to:

$$\bigwedge_{1\leq i\leq k} \Big(\bigwedge_{e_{v,w}\in C_{\text{merge}}} \big((\neg e_{v,w} \vee x_{v,i} \vee \neg x_{w,i}) \wedge (\neg e_{v,w} \vee \neg x_{v,i} \vee x_{w,i})\big) \wedge \bigwedge_{\neg e_{v,w}\in C_{\text{merge}}} (e_{v,w} \vee \neg x_{v,i} \vee \neg x_{w,i}) \Big) \tag{6.4}$$

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

$$\begin{array}{lll} e_{1,2}\{ & (e_{1,2}\vee\neg x_{1,i}\vee\neg x_{2,i}) & \\ e_{3,4}\{ & (e_{3,4}\vee\neg x_{3,i}\vee\neg x_{4,i}) & \\ e_{2,5}\{ & (\neg e_{2,5}\vee x_{2,i}\vee\neg x_{5,i}) \quad \wedge \quad (\neg e_{2,5}\vee\neg x_{2,i}\vee x_{5,i}) & \\ e_{4,5}\{ & (\neg e_{4,5}\vee x_{4,i}\vee\neg x_{5,i}) \quad \wedge \quad (\neg e_{4,5}\vee\neg x_{4,i}\vee x_{5,i}) & \end{array} \qquad 1\leq i\leq 3 \tag{6.5}$$

2. Proof of correctness of symmetry-free conflict clauses

Definition Let $\pi : (1..k) \rightarrow (1..k)$ be an function that is one to one and onto.

Definition Let $\mathcal{P}_\pi$ be a recursive function for which holds:

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

Theorem 2.1 *If $\mathcal{F}$ represents a k-coloring instance, encoded with the direct encoding and C_h is implied by $\mathcal{F}$, then for any π, $\mathcal{P}_\pi(C_h)$ is also implied by $\mathcal{F}$.*

Proof Given a resolution trail R, with no clauses being learnt clauses, from which the first clause C_h was learnt, the resolution trail R_π, from which $\mathcal{P}_\pi(C_h)$ would be learnt, can be be constructed by replacing for every literal $l_{h,i} \in R$ with $\mathcal{P}_\pi(l_{h,i})$. R_π is a correct resolution trail, since for every clause C_r represented in R it holds that $\mathcal{P}_\pi(C_r) \in \mathcal{F}$. Thus for every π, $\mathcal{P}_\pi(C_h)$ is implied by $\mathcal{F}$.

For any next learnt clause C_h^*, either R contains no learnt clauses, in which case our original proof applies or it has already been shown that for a clause in $\mathcal{F}$ which does correspond to a learnt clause C_h, for any π, $\mathcal{P}_\pi(C_h)$ is implied by $\mathcal{F}$, thus proving that for any possible π, $\mathcal{P}_\pi(C_h^*)$ is also implied by $\mathcal{F}$.

Theorem 2.2 *Let C_{conflict} be a conflict clause consisting of merge literals and negative color literals. A literal $\neg x_{v,i} \in C_{\text{conflict}}$ is redundant, if C_{conflict} does not contain a literal $\neg x_{w,i}$ $(v \neq w)$ and C_{conflict} contains for each color in the conflicting assignment $j \neq i$ a literal $\neg x_{w,j}$ $(v \neq w)$ such that $(v, w) \in E$.*

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

Theorem 2.3 *Let C_{conflict} be a conflict clause consisting of merge literals and negative color literals. A literal $\neg e_{v,w} \in C_{\text{conflict}}$ is redundant, if $(\neg x_{v,i} \vee \neg x_{w,i}) \in C_{\text{conflict}}$, while a literal $e_{v,w} \in C_{\text{conflict}}$ is redundant, if $(\neg x_{v,i} \vee \neg x_{w,j}) \in C_{\text{conflict}}$.*

Proof (by resolution) Let $\otimes$ denote the resolution operator. For $\neg e_{v,w}$, resolve $C_{\text{conflict}} \otimes (e_{v,w} \vee \neg x_{v,i} \vee \neg x_{w,i})$. For $e_{v,w}$, resolve $C_{\text{conflict}} \otimes \big((\neg e_{v,w} \vee \neg x_{v,i} \vee x_{w,i}) \otimes (\neg x_{w,i} \vee \neg x_{w,j})\big)$.

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 conflict clause* C_{conflict} if all redundant literals (based on Theorem 2.2 and 2.3) are removed.

Theorem 2.4 *Let C_{conflict} be a reduced conflict clause consisting of merge literals and negative color literals. If C_{merge} is a corresponding merge clause, by combining DP-resulotion [7] on merge variables in C_{merge} on $C_{\text{merge}} \wedge M(C_{\text{merge}})$ with resolution on at-most-one clauses, while removing tautological and subsumed clauses, will results in:*

$$\bigwedge_{\pi_1..\pi_{k!}} \mathcal{P}_{\pi_i}(C_{\text{conflict}}) \tag{6.6}$$

Proof First, we perform DP-resolution on all merge variables that occur as a negative literal in C_{merge} but not in C_{conflict}. Let k denote the number of colors and n the number of corresponding negative merge literals. So, the above DP-resolution results in kn resolvents. Yet many of them are redundant:

- In case both $\neg e_{u,v}$ and $\neg e_{v,w}$ occur in C_{merge} then all resolvents with the literals $\neg x_{u,i} \vee \neg x_{v,i} \vee \neg x_{v,j} \vee \neg x_{w,j}$ are redundant because they are subsumed by the at-most-one clauses $(\neg x_{v,i} \vee \neg x_{v,j})$ (see Section 1).
- In case $(v, w) \in E$ then all resolvents with $\neg x_{v,i} \vee \neg x_{w,i}$ are redundant because they are subsumed by the conflicting clauses $(\neg x_{v,i} \vee \neg x_{w,i})$.
- In case positive merge literal $e_{v,w} \in C_{\text{merge}}$ then resolvents with $(\neg x_{v,i} \vee \neg x_{w,i})$ are redundant because all resolvents after DP-resolution on $e_{v,w}$ will be either subsumed by the at-most one clauses or are tautological.

Second, we get rid of the merge variables that occur as a positive literal in C_{merge} but not in C_{conflict}. Notice that by performing resolution on $(\neg x_{w,i} \vee \neg x_{w,j}) \otimes (\neg e_{v,w} \vee \neg x_{v,i} \vee x_{w,i})$ we can deduce each clause $(\neg e_{v,w} \vee \neg x_{v,i} \vee \neg x_{w,j})$. Now, we apply resolution on all pairs of not redundant resolvents of the first step with all possible clauses $(\neg e_{v,w} \vee \neg x_{v,i} \vee \neg x_{w,j})$. After removing all clauses that are subsumed by other clauses, the only remaining clauses are $\mathcal{P}_{\pi_i}(C_{\text{conflict}})$. For an detailed example of DP-resolution we refer the reader to appendix 1.

Thus once we have learnt C_{conflict}, we could add all clauses $\mathcal{P}_{\pi_i}(C_{\text{conflict}})$ to $\mathcal{F}$ (Theorem 2.1). Yet, based on Theorem 2.4, we add $C_{\text{merge}} \wedge M(C_{\text{merge}})$ instead.

Chapter 7

Symmetry-free learning for Van der Waerden instances

In 1927 the Dutch mathematician Van der Waerden proved that for every positive integers r and l there is a positive integer n such that every partition of $\{1..n\}$ into r blocks has at least one block which contains a sequence of l number such that the difference of any two successive members of that sequence is a constant, called an arithmetic progression of l. The number n for a given r and l is called the Van der Waerden number $W(k, l)$. For example we can partition$\{1..8\}$ into two blocks $\{1, 4, 5, 8\}$ and $\{2, 3, 6, 7\}$ without creating an arithmetic progression of 3, while every partition of $\{1..9\}$ into 2 blocks will contain such a progression, thus $W(2, 3) = 9$. Calculating Van der Waerden numbers is a very hard combinatorial problem and currently only 6 such numbers are known [29]. The question whether the numbers $\{1..n\}$ can be partitioned in r blocks without an arithmetic progression of l is referred to as the Van der Waerden instance $W(r, l, n)$.

1. Translating a Van der Waerden instance into SAT

The encoding used for Van der Waerden instance $W(r, l, n)$ is the widely accepted encoding presented in [29]. Much like the k-coloring problem, this encoding consists of three types of clauses. The property that each number n must be placed in one block is encoded by the at-least-one clauses, which are of the form:

$$(x_{v,1} \vee x_{v,2} \vee \cdots \vee x_{v,r}) \qquad 1 \leq v \leq n \tag{7.1}$$

The conflicting clauses forbid each possible arithmetic progression:

$$(\neg x_{v,j} \vee \neg x_{v+d,j} \vee .. \vee \neg x_{v+d(l-1),j}) \qquad 1 \leq v, d \leq n \;\; v + (l-1)d \leq n, \;\; 1 \leq j \leq r \tag{7.2}$$

The at-most-one clauses encode that each number can only be placed in one block:

$$(\neg x_{v,i} \vee \neg x_{v,j}) \qquad 1 \leq v \leq n, \;\; 1 \leq i < j \leq r \tag{7.3}$$

2. Symmetry breaking predicates

Static partial symmetry breaking for the instance $W(r, l, n)$ can be done by limiting the valid blocks for $r-1$ selected numbers $\{n_1 \,..\, n_{r-1}\}$. These symmetry breaking predicates force that the n_1 must be placed in block 1, n_2 must be placed either in block 1 or 2 and n_3 number must be placed in either be placed in block 1,2 or 3 etc. Numbers $\{n_1 \,..\, n_{r-1}\}$ are the numbers which occur in the

largest number of arithmetic progressions. Forbidden blocks for numbers are encoded using negative unit clauses as by (7.4).

$$\neg x_{n,j} \qquad n \in \{n_1 \,..\, n_{r-1}\} \qquad j \geq n+1 \tag{7.4}$$

For example consider the Van der Waerden instance $W(4,3,10)$. For this instance the following arithmetic progressions are forbidden:

$$\begin{array}{lll} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \\ 4 & 5 & 6 \\ 5 & 6 & 7 \\ 6 & 7 & 8 \\ 7 & 8 & 9 \\ 8 & 9 & 10 \\ 1 & 3 & 5 \\ 2 & 4 & 6 \\ 3 & 5 & 7 \\ 4 & 6 & 8 \\ 5 & 7 & 9 \\ 6 & 8 & 10 \\ 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \\ 4 & 7 & 10 \\ 1 & 5 & 9 \\ 2 & 6 & 10 \end{array} \tag{7.5}$$

The three most occurring numbers in these progressions are: 5(8x) 6(8x) 4(7x). Thus our symmetry breaking predicates become:

$$\begin{array}{lll} (\neg x_{5,4}), & (\neg x_{5,3}), & (\neg x_{5,2}) \\ (\neg x_{6,4}), & (\neg x_{6,3}) & \\ (\neg x_{4,4}) & & \end{array} \tag{7.6}$$

Chapter 8

MiniMerge

We have implemented the principal of symmetry-free conflict clauses in the open source conflict-driven solver Minisat2 in the form of the CDCL_merge algorithm. CDCL_merge is a conflict-driven algorithm, specialized for combinatorial problems such as the k-coloring problem and Van der Waerden instances, which uses symmetry-free conflict clauses to store conflicts. Its most important feature is the procedure transformConflict, which transforms a normal conflict clause into a symmetry-free conflict clause. In order to make the procedure transformConflict function properly, we also had to adapt the decide procedure. Algorithm 8.1 gives a detailed overview of our CDCL_merge algorithm.

Algorithm 8.1 CDCL_merge($\mathcal{F}$)

```
1:  while true do
2:    F ← propagate(F) /* propagate unit clauses */
3:    if not conflict then
4:      if all variables assigned then
5:        return SAT
6:      end if
7:      decide() /*select decision variable see Section 2*/
8:    else
9:      C_conflict ← analyze() /*analyze the conflict*/
10:     C_merge ← transformConflict(C_conflict) /*transforms C_conflict See Section 1*/
11:     if top level conflict found then
12:       return UNSAT
13:     end if
14:     F ← backtrack(C_merge) /*undo assignments until conflict clause is unit*/
15:   end if
16: end while
```

1. The transformConflict procedure

The transformConflict procedure follows the following steps to convert C_{conflict} into C_{merge}:

1. All positive color literals in C_{conflict} are transformed into merge and negative color literals, by expending them into their reason literals.
2. All according to Theorem 2.2 and 2.3 redundant literals 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 out of all merge literals in C_{conflict}.

4. Transform C_{color} into the symmetry-free C_{zykov} by computing the the corresponding node in the Zykov-tree. Preliminary tests showed that our runtimes improve if we keep the number of introduced variables to a minimum and reuse already introduced variables whenever possible. Thus when we must make a choice between two possible merge literals that can both be used for C_{zykov}, we will always choose the literal which was added to the largest number of conflict clauses. Ties are broken pseudo randomly.

5. Return the union of C_{Zykov} and C_{extra} as the transformed clause C_{merge}.

2. The decide procedure

First and foremost, the variable selection heuristics must facilitate that any conflict clause can be extended in such a way that it contains no positive color literals (see Section 1). We ensure this by assigning each decision variable to `true`. This heuristic is similar to the one used in MiniSat2 which assigns all decision variables to `false` [8].

Although merge variables are useful to create symmetry-free 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). Section 10 seems to bear this out

Finally, we propose a specialized version of the VSIDS activity heuristic. 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 has 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 [3].

3. The backtrack procedure

An important property of clauses generated by the *transformConflict* procedure, which complicates backtracking, is that they are not necessarily clauses that become unit after backtracking. For example, consider the conflict depicted in Figure 8.1 from which $C_{merge} = (e_{1,2} \vee e_{1,3} \vee e_{2,3})$ is learnt.

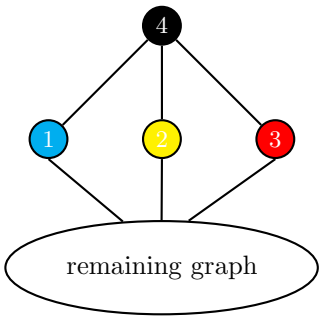


Figure 8.1. An example of a coloring that cannot be expressed with an asserting clause. The numbers in the vertices refer to their index v_i. Vertex v_4 is in conflict because it cannot be colored. Although various reasons can be addressed for this conflict, the example focusses on the assignment to v_1, v_2 and v_3, which were set at level 1, 2 and 3 respectively.

Since both $e_{1,3}$ and $e_{2,3}$ are set at level 3, after backtracking, both $e_{1,3}$ and $e_{2,3}$ become free, leaving no asserting literal. We refer to such clauses which have no asserting literal, as non-asserting

clauses. Although many different methods could be used to deal with non-asserting clauses, the following method showed the best results during preliminary tests. Given non-asserting clause C_l, which after backtracking, contains the free literals $l_1..l_n$, we branch on $\neg l_1, \neg l_2..\neg l_{n-1}$ until a conflict occurs or l_n is propagated. The literal l_n that is chosen to be propagated by the newly created clause, is the literal which we would have chosen to branch on, if $l_1..l_n$ would be the only available free literals. After either a conflict or the propagation of l_n, normal branching activity and conflict analysis is resumed.

The existence of non-asserting clause proves that although C_{conflict} is satisfiability equivalent to $C_{\text{merge}} \wedge M(C_{\text{merge}})$ (see Theorem 2.4), the transformation is not *arc-consistent* under unit propagation [41] for a non-asserting C_{merge}. 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_1 (or v_2) was the last assigned vertex.

The lack of arc-consistency is a serious (if not the largest) weakness of MiniMerge. The number of times such clauses are generated is very high in some cases larger than 50%, thus severely limiting the effectiveness of MiniMerge. We have studied various options to deal with this weakness, but to date no satisfactory solution has been found which guarantees arc-consistentcie under unit propagation.

4. Adding support clauses

Given a new clause C_{merge}, we must add the support clauses for each merge literal $e_{v,w} \in C_{\text{merge}}$ which have not yet been added to $\mathcal{F}$. Theoretically we only have to add the support clauses which propagate the negation of $e_{v,w}$, however in our current implementation we add all support clauses for $e_{v,w}$. This practice is considered to be a point for improvement in new version of MiniMerge, as these unnecessary clauses slow down assignment propagation.

Chapter 9

MiniMerge2

MiniMerge2 is an experimental version of MiniMerge, it has three additional features, mixed merge clauses, shortcutting and preferential merge literal selection, which we will describe in the following sections.

1. Mixed merge clauses and shortcutting

In the face of the partial symmetry breaking induced by a forced clique the following question arises: "Given that some symmetry is already broken, is it useful to convert a given conflict clause into merge clauses?" After all, converting conflict clauses takes time and the support clauses for the corresponding merge clause slow down assignment propagations.

To illustrate the different scenario's we might face, consider the following clause C_{conflict}, learnt in a 3-coloring instance:

$$(\neg x_{1,1} \vee \neg x_{2,1} \vee \neg x_{3,2} \vee \neg x_{4,2} \vee \neg x_{5,3}) \tag{9.1}$$

Suppose that in the encoded graph coloring instance G a forced clique forces $\varphi_{\text{color}}(v_1) = 1$, $\varphi_{\text{color}}(v_2) = 2$ and $\varphi_{\text{color}}(v_5) = 3$. Obviously in that case all symmetries of C_{conflict} are already broken through the forced clique. Since all symmetries of C_{conflict} are already broken, converting C_{conflict} into a corresponding merge does not break any addition symmetries. Note that all symmetries would also be broken if the forced clique would force only $\varphi_{\text{color}}(v_1) = 1$ and $\varphi_{\text{color}}(v_1) = 2$.

More generally, if in a k-coloring instance $\mathcal{F}$ a learnt clause C_{conflict} contains a literal that corresponds a forced vertex for every color in C_{conflict} or if C_{conflict} contains k colors and $k-1$ literals which encode forced vertices, directly adding C_{conflict} is equivalent to adding a corresponding merge clause is. Directly adding C_{conflict} to $\mathcal{F}$ instead of converting it into a corresponding merge clause is revered to as *shortcutting*.

Let us get back to our example clause C_{conflict}, suppose that the forced clique only forces $\varphi_{\text{color}}(v_1) = 1$. In this cases the forced clique does break some, but not all symmetries of C_{conflict}. The following symmetries of C_{conflict} are implied by $\mathcal{F}$, but not broken by the fixed clique:

$$\begin{array}{l}(\neg x_{1,1} \vee \neg x_{2,1} \vee \neg x_{3,3} \vee \neg x_{4,3} \vee \neg x_{5,2}) \\ (\neg x_{1,1} \vee \neg x_{2,1} \vee \neg x_{3,2} \vee \neg x_{4,2} \vee \neg x_{5,3})\end{array} \tag{9.2}$$

Using merge clauses, we could easily break these symmetries by learning the following merge clause:

$$(\neg e_{1,2} \vee \neg e_{3,4} \vee e_{1,3} \vee e_{1,5} \vee e_{3,5}) \tag{9.3}$$

Unfortunately this merge clause breaks symmetries of C_{conflict}, that are not implied by $\mathcal{F}$ such as:

$$(\neg x_{1,2} \vee \neg x_{2,2} \vee \neg x_{3,1} \vee \neg x_{4,1} \vee \neg x_{5,3}) \tag{9.4}$$

Breaking symmetries that are not implied is not useful and wastes resources. Fortunately we can change C_{merge} into a clause which breaks only the symmetries implied by $\mathcal{F}$ by using the following equalities:

$$e_{1,2} \leftrightarrow x_{2,1}\ , e_{1,3} \leftrightarrow x_{3,1}\ , e_{1,5} \leftrightarrow x_{5,1} \tag{9.5}$$

These equalities hold since the fixed clique forces $\varphi_{\text{color}}(v_1) = 1$. By replacing merge literals in C_{merge} by their corresponding color literals in the above equalities, we create a clause that breaks all symmetries of C_{conflict}, not broken by the fixed clique, while not breaking symmetries that are not implied by $\mathcal{F}$:

$$(\neg x_{2,1} \vee \neg e_{3,4} \vee x_{3,1} \vee x_{4,1} \vee \neg x_{5,3}) \tag{9.6}$$

We refer to clauses such as (9.6) as *mixed merge clauses* and the color literals in them as *pseudo merge literals*, we denoted such literals as $\bar{x}_{v,i}$ to distinguish them from normal color literals. Using this notation we write the above clause as:

$$(\neg \bar{x}_{2,1} \vee \neg e_{3,4} \vee \bar{x}_{3,1} \vee \bar{x}_{4,1} \vee \neg \bar{x}_{5,3}) \tag{9.7}$$

Mixed merge clauses have two advantages over "normal" merge clauses:

1. Mixed merge clauses are quicker constructed than merge clauses since there are far less options to consider when encoding the corresponding merge clause with color literals than through merge literals.

2. Less support clauses are added to $\mathcal{F}$ since less merge variables are used in conflict clauses, thus decreasing time spent propagating assignments.

1.1 Implementing mixed merge clauses

Although mixed merge clauses eliminate overhead caused by redundant merge variables in MiniMerge, they also significantly increase the complexity of MiniMerge2. This increased complexity is caused by the fact that mixed merge clauses are asymmetric constraints that are only valid given the specific assignment of our forced clique.

Adding asymmetric constrains to $\mathcal{F}$ makes it impossible to directly transform C_{conflict} into C_{merge} since permutations of the reason set for C_{conflict} may not lead to a conflict due the asymmetric nature of $\mathcal{F}$.

The problem of the asymmetric nature of $\mathcal{F}$ can be solved by calculating the reason set for the current conflict which would be generated if no mixed merge clauses, but 'normal' mixed clauses were used and converting the C_{conflict} corresponding to that reason into C_{merge}. Calculating this reason set requires two steps:

1. First, while constructing the implication graph we must add implicit vertices. Implicit vertices are vertices which would occur in the implication graph if normal mixed merge clause are used but don't occur if mixed merge clauses are used.

   For example, consider the literal $x_{v,i}$ which is falsified by the propagation of $\neg\bar{x}_{v,i}$. If normal mixed merge clauses were used, not $\neg\bar{x}_{v,i}$ but $\neg e_{v,w}$, where w is a vertex with a forced coloring i would have been propagated by the clause which propagated $\neg\bar{x}_{v,i}$. The propagation of $\neg e_{v,w}$ would have caused the supporting clause $(e_{v,w} \vee \neg x_{v,i} \vee \neg x_{w,i})$ to be become unit, which in turn would have propagated $\neg x_{v,i}$ falsifying $x_{v,i}$. Thus if we wish to create a reason set which is equivalent to the 'normal' reason set, we must add a vertex $\neg x_{w,i}$ to the implication graph with a directed edge to the same vertex as $\neg x_{v,i}$ is connected to.

   Fortunately, implicit vertices always correspond to a variable set at level 0, therefore if we detect an implicit vertex, we can directly add it's variable to the reason set and don't need to expand the vertex.

Table 9.1 shows an overview of the possible scenarios which might arise during the implication graph construction and whether or not to add an implicit vertex (our example corresponds to case 3). Note, that we consider a branch on negative color literal $\neg x_{v,i}$, a branch on a merge literal $\neg e_{v,w}$, where w is some vertex forcibly colored i. This is done because due to the way we deal with non-asserting clauses we may need to branch on negative color literals to falsify pseudo merge literals in a C_{merge}. By treating branches on negative color literals as a branch on merge literals we still can express any conflict in terms of (pseudo) merge literals and color literals.

2. Second, pseudo merge literals in the reason set must be treated as merge literals and not as color literals with respect to merge clause construction. For example the corresponding mixed merge clause for the reason set $\{x_{1,1},\ x_{2,1},\ \bar{x}_{3,2}\}$ should be $(\neg e_{1,2} \vee \neg \bar{x}_{3,2})$ instead of: $(\neg e_{1,2} \vee e_{2,3})$ as the reason set corresponds to the reason set $\{x_{1,1},\ x_{2,1},\ e_{3,v}\}$, where v is some vertex with a forced coloring.

Table 9.1. Lookup table for deciding when to add implicit vertices to the implication graph.

case	falsified literal	in which type of clause	reason clause	add implicit vertex?
1	positive color literal	normal clause	branch variable	yes
2	negative color literal	normal clause	branch variable	no
3	positive color literal	normal clause	learnt clause	yes
4	negative color literal	normal clause	learnt clause	yes
5	positive color literal	normal clause	normal clause	no
6	negative color literal	normal clause	normal clause	no
7	positive color literal	learnt clause	branch variable	no
8	negative color literal	learnt clause	branch variable	yes
9	positive color literal	learnt clause	learnt clause	no
10	negative color literal	learnt clause	learnt clause	no
11	positive color literal	learnt clause	normal clause	yes
12	negative color literal	learnt clause	normal clause	yes
13	any merge literal	any	any	no

2. Preferential merge literals selection

As stated when we must choose between two merge literal candidates for C_{merge} we take the literal that occurs in the most conflict clauses and ties are broken pseudo randomly. However when the *preferential merge literal selection* optimization is used, ties between unused merge literals are broken by an user determined score given to each merge literal.

For graph coloring instances we gave literal $e_{v,w}$ a score equal to the number of common neighbors between vertices v, w, which is a commonly used heuristic to determine the most effective vertex pair to branch on in Zykov algorithms as can be seen in Chapter 2.1.

For a Van der Waerden instances we choose to give literal $e_{v,w}$ a score equal to the number of progressions in which they both occur.

Chapter 10

Results

In this chapter we will present the performance of the MiniMerge solver on different graph coloring instances and Van der Waerden instances, benchmarked against the normal distribution of MiniSat2. For our experiments we used the following groups of instances:

Graphs with small chromatic numbers

In [10] experiments were done with a different dynamic symmetry breaking technique. In this technique symmetry-free learning was implemented by adding all symmetries of C_{conflict} to $\mathcal{F}$. This approach was very successful for graphs with a chromatic number of 3 or 4 but failed for graphs with for a chromatic number greater than 4.

We have repeated this experiment using our technique. We generated 500 graphs around the phase transition density, when graphs have a critical vertex / edge ratio so they are just k-colorable or just not k-colorable. For 3-, 4-, and 5-colouring experiments, we generated graphs with 400, 140 and 90 vertices respectively. We have run these instances twice, once without symmetry breaking predicates and once with symmetry breaking predicates.

The experiments on the instances without symmetry breaking predicates were done with a beta version of bMiniMerge, which did not support our special VSIDS version, even though we used only a beta version of bMiniMerge, the results are still substantial.

Medium sized graphs with a large chromatic number

Like [25] we have benchmarked our solver on medium graphs with relative large chromatic numbers. We generated a number of graphs containing 70 vertices with different edge probabilities. An edge probability of p means that there is a chance of p that two vertices v, w have an edge between them in G. In total we generated 15 graphs with an edge probability of 0.5, 0.7 and 0.9. We ran each graphs twice, once with k being $\mathcal{X}(G)$ and once k being $\mathcal{X}(G) - 1$. Preliminary tests indicated that these instances can only be effectively solved using symmetry breaking predicates, therefore we show no results without symmetry breaking predicates for these graphs.

DIMACS benchmark graphs

An important overview of the overall performance of SAT solvers on the GCP was presented in[20]. It describes a number of graphs from the DIMACS benchmarking set [23] where a SAT based approaches have great difficulty in solving them. We chose these graphs to benchmark the performance of our solver. Table 10.1 presents an overview of these graphs. The runtimes presented in this section are scaled to a workstation which solves the dfmax benchmark in 12s for r500.5.b on our platform compared to the 16.96s in [20]. For more information of the dfmax benchmark, please check [23].

Table 10.1. Difficult DIMACS instances.

instance	$\|V\|$	$\|E\|$	$\mathcal{X}(G)$	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

Van der Waerden instances

Benchmarking the performance of our solver on the Van der Waerden instances turned out to be very difficult. The available UNSAT instances were either to simple (W(3,3)), had not enough symmetries for our technique to be useful (all W(2,r) instances) or were too hard (W(4,3)) (We ran the UNSAT Van der Waerden instance W(4,3,76) for two days straight without results). The most difficult satisfiable instance available, W(4,3,75), was not solved by either MiniSat2, MiniMerge or MiniMerge2 within 2.5 hours. Results for easier instances than W(4,3,75) seemed to indicate that MiniSat2 outperforms MiniMerge but no clear pattern emerged. Therefore we draw no conclusion as to the effectiveness of our technique on Van der Waerden instances.

MiniSat2 parameters

By default MiniSat2 branches on negative variables but as branching on positive variables is superior to branching on negative variables in respect to GCP instances, in the interest of fairness MiniSat2 branches on positive variables in these experiments.

1. The results for graphs with small chromatic numbers

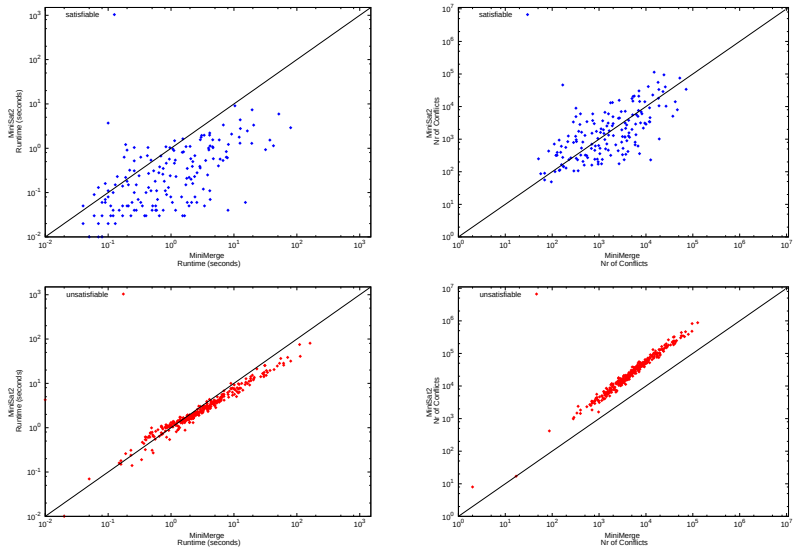


Figure 10.1. A comparison between bMiniMerge and MiniSat2 on random 3-coloring instances with 400 vertices without using symmetry breaking predicates.

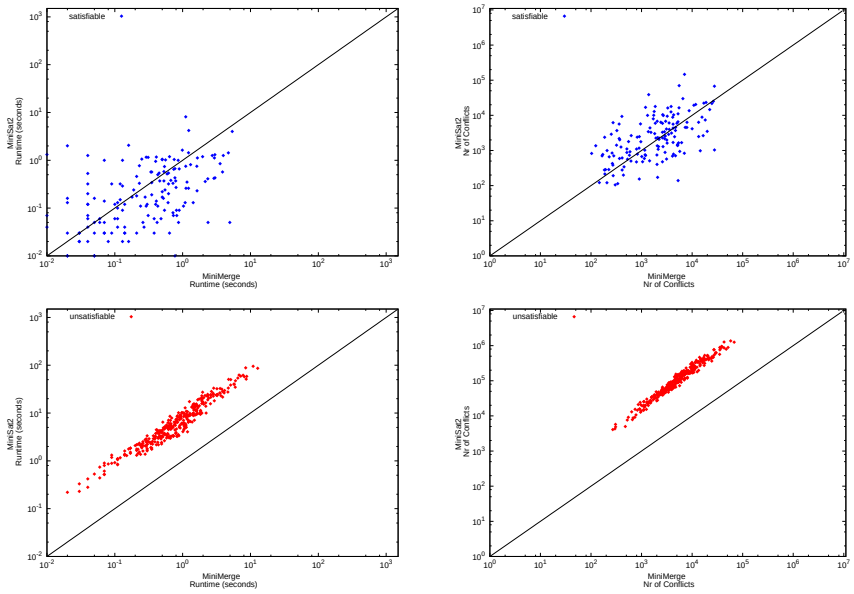


Figure 10.2. A comparison between bMiniMerge and MiniSat2 on random 4-coloring instances with 140 vertices without using symmetry breaking predicates.

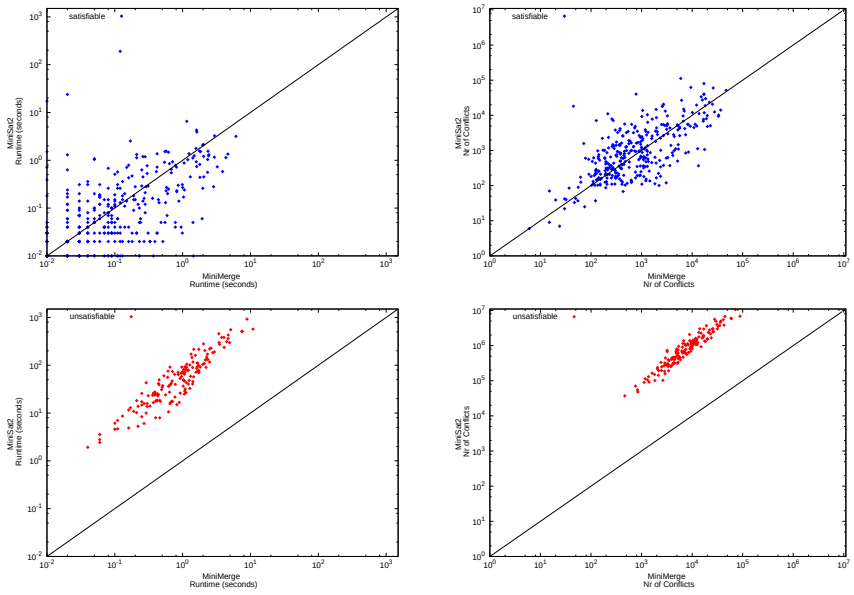


Figure 10.3. A comparison between bMiniMerge and MiniSat2 on random 5-coloring instances with 90 vertices without using symmetry breaking predicates.

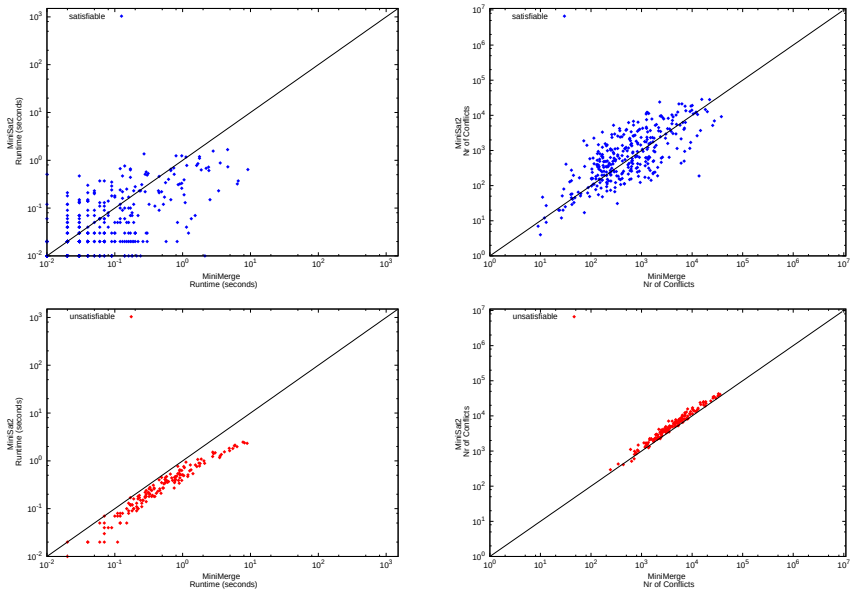


Figure 10.4. A comparison between MiniMerge and MiniSat2 on random 5-coloring instances with 90 vertices using symmetry breaking predicates and our specialized VSIDS.

The result of the unsatisfiable instances shown in Figures 10.1, 10.2 and 10.3 clearly shows the exponential speedup we expected to achieve through our symmetry breaking technique. These unsatisfiable instances are relatively easy, thus the deteriorating performance of MiniSat2, as the chromatic number increases can only be caused by the increasing number of symmetries present in $\mathcal{F}$, which our technique breaks and thus is unaffected by.

After adding symmetry breaking predicates to $\mathcal{F}$, the performance of MiniSat2 rapidly catches up with the performance of MiniMerge as shown in Figure 10.4. It should be noted though, that the number of conflicts for MiniMerge consistently stays significantly below the number of conflicts of MiniSat2 showing that after adding symmetry breaking clauses, there all still symmetries present in $\mathcal{F}$, which are not broken by the forced clique but are broken by our technique. This would seem to indicate that for larger graphs with the same chromatic number, the performance of MiniMerge as compared to MiniSat2 would be better.

Furthermore Figure 10.1 shows that even though the number of conflicts for MiniMerge is less than the number of conflicts for MiniSat2 for 3-coloring instances the runtime is a lot worse. This indicates that the symmetry breaking approach presented in [10] will outperform our symmetry breaking technique for 3-coloring instances and we suspect it might be on par with our technique for 4-coloring instances.

The varying effectiveness of our technique on satisfiable is attributed to the fact that solving individual random satisfiable instances depends for a large part on "luck" and the used random seed and less on the employed solving method.

Besides using our VSIDS scheme, we also experimented with a branching strategy in which we branch on merge variables used in merge clauses. The results of this strategy are comparable with the results presented in Figure 10.4.

2. The results for medium sized graphs with a large chromatic number

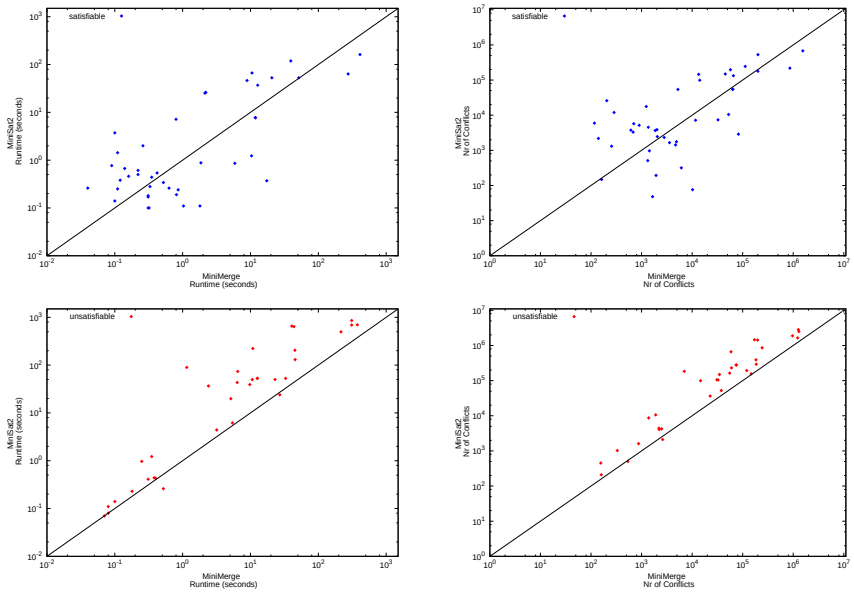


Figure 10.5. A comparison between Minisat2 and MiniMerge on random graphs of 70 vertices with different densities using symmetry breaking predicates and our specialized VSIDS.

Table 10.2. Average runtimes for medium sized graphs for MiniSat2 and MiniMerge, with a 1200 (s) timeout.

				SAT instances				UNSAT instances			
				Minisat2		MiniMerge		Minisat2		MiniMerge	
$\|G\|$	$\|V\|$	P_{edge}	$\mathcal{X}(G)$	time (s)	#	time (s)	#	time (s)	#	time (s)	#
15	70	0.5	11-12	25.59	15	33.4	15	190.72	14	75.85	14
15	70	0.7	17-18	24.73	13	33.15	14	307.88	8	69.43	12
15	70	0.9	27-28	0.73	15	0.36	15	19.00	13	1.4	15

Table 10.2 shows the average solving times of instances solved by both MiniMerge and MiniSat2 and the number of solved instances by each solver per edge probability. As can be seen in this table and Figure 10.5 the performance of MiniMerge is on par with MiniSat2 on satisfiable instances, although MiniColor was able to solve one more instance. On the other hand, performance on unsatisfiable instances has significantly improved. Besides solving more instances, MiniMerge is noticeably quicker especially on graphs with the highest chromatic numbers. Again besides using our VSIDS scheme, we experimented with a branching strategy in which we branch on merge variables used in merge clauses. The results of this strategy were terrible as most benchmarks were no longer solved. This seems to indicate that branching on merge variables is a bad strategy for graphs with large chromatic numbers.

3. The results for the DIMACS benchmarks

Table 10.3. Runtimes on difficult satisfiable DIMACS runtimes in seconds. VanGelder denotes the best results presented in [20]

	SAT instances			
instance	k	VanGelder	MiniSat2	MiniMerge
`Myciel6`	7	0	0	0.01
`abb313GPIA`	9	1256	3.63	2.71
`DSJC125.5`	19	4446	43.46	644
`DSJC125.9`	46	26958	140	$\geq$ 19000

Table 10.4. Runtimes on difficult unsatisfiable DIMACS runtimes in seconds. VanGelder denotes the best results presented in [20]

	UNSAT instances			
instance	k	VanGelder	MiniSat2	MiniMerge
`Myciel6`	6	2113	3096	1872
`abb313GPIA`	8	5.63	0.73	0.75
`DSJC125.5`	12	488	5.85	6.00
`DSJC125.9`	37	4630	934	119

As can be seen in Table 10.3 and 10.4, the runtimes of our implementation on unsatisfiable instances are vast improvements over the runtimes of MiniSat2 and those presented in [20], although we seemed to do worse on the satisfiable instances.

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 10.4. With respect to the instances DSJC125.5 and DSJC125.9, we would like to point out that no SAT based approach has been able to prove these bounds! Like in the medium graph instances, branching strategies which branch on merge variables yield poor results for the DIMACS instances.

Table 10.5. Runtimes of MiniMerge on harder versions of the DIMACS instances.

	SAT instances			UNSAT instances		
instance	k	MiniSat2	MiniMerge	k	MiniSat2	MiniMerge
`Myciel7`	8	0	0.03	6	6497	1339
`DSJC125.5`	18	$>$ 19000	$>$ 19000	13	$>$ 19000	4018
`DSJC125.9`	45	$>$ 19000	$>$ 19000	38	$>$ 19000	10091

Chapter 11

Results: A comparison between MiniMerge and MiniMerge2

This chapter offers a comparison between the performance of our experimental solver MiniMerge2 and MiniMerge on the instances from the previous chapter, to show the effect of mixed merge clauses. In order to provide a clear comparison between MiniMerge and MiniMerge2 the version of MiniMerge2 used for these experiments only supports mixed merge clauses and no shortcutting or preferential merge variable selection.

1. The results for graphs with small chromatic numbers

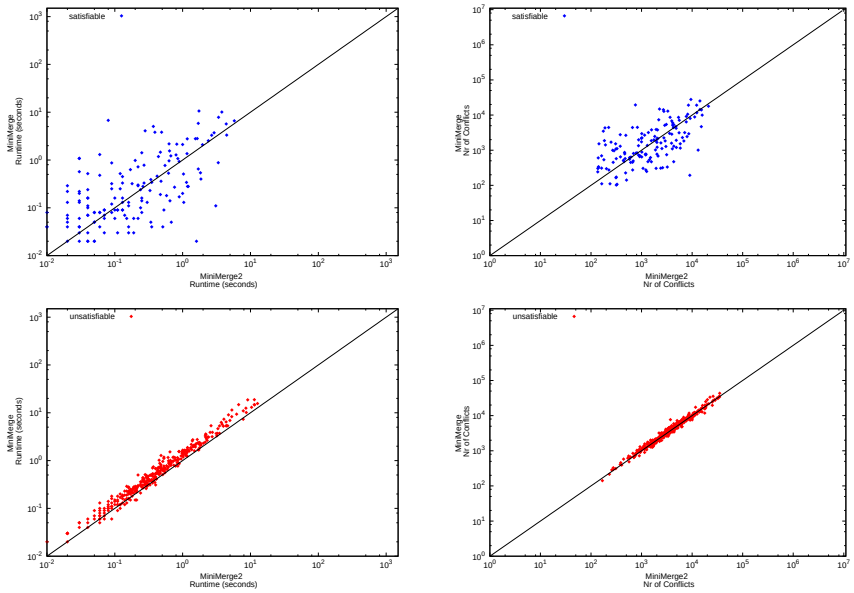


Figure 11.1. A comparison between MiniMerge and MiniMerge2 on satisfiable random 5-coloring instances with 90 vertices using symmetry breaking predicates and our specialized VSIDS.

2. The results for medium sized graphs with a large chromatic number

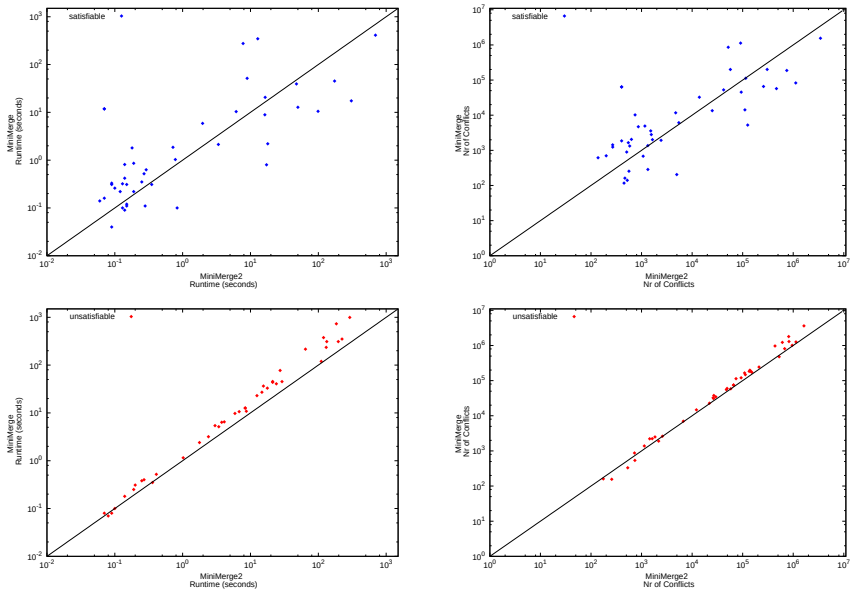


Figure 11.2. A comparison between MiniMerge and MiniMerge2 on random graphs of 70 vertices with different densities using symmetry breaking predicates and our specialized VSIDS.

Table 11.1. Average runtimes for medium sized graphs, with a 1200 (s) timeout.

				SAT instances				UNSAT instances			
				MiniMerge		MiniMerge2		MiniMerge		MiniMerge2	
$\|G\|$	$\|V\|$	P_{edge}	$\mathcal{X}(G)$	time (s)	#	time (s)	#	time (s)	#	time (s)	#
15	70	0.5	11-12	33.4	15	52.12	15	141.49	14	55.37	14
15	70	0.7	17-18	56.49	14	50.13	14	166.08	12	72.23	14
15	70	0.9	27-28	0.364	15	0.20	15	8.8	15	3.694	15

3. The results for the DIMACS benchmarks

Table 11.2. Runtimes of MiniMerge on difficult of the DIMACS instances.

	SAT instances		UNSAT instances	
instance	k	MiniMerge2	k	MiniMerge2
`Myciel7`	8	0	6	1056
`DSJC125.5`	18	263	12	4.25
`DSJC125.9`	45	28.24	38	95.92

Table 11.3. Runtimes of MiniMerge on harder versions of the DIMACS instances.

	SAT instances		UNSAT instances	
instance	k	MiniMerge	k	MiniMerge
`Myciel7`	8	0.03	6	892
`DSJC125.5`	18	> 19000	13	2054
`DSJC125.9`	45	> 19000	38	8604

4. A runtime comparison between MiniMerge2 and Minimerge

The results presented in sections 1, 2 and 3 of this Chapter indicate that MiniMerge2 is a factor 1.5 till 2 times faster than MiniMerge while needing around the same number of conflicts to solve an instance.

5. The impact of shortcutting in MiniMerge2

For the small chromatic number graphs the number of times shortcutting could be applied was significantly larger than we expected. If used, up to 70% of the generated conflicts were generated using shortcutting. We attribute this to the fact that these graphs turned out to be relatively small and the sizes of the forced cliques were equal to chromatic numbers of the involved graphs. Thus the majority of the conflicts involved the forced clique and since the forced clique contained all available colors, there was a large chance that all symmetries were already broken.

However since the mixed merge clauses used by MiniMerge2 only contain merge literals if there are unbroken symmetries, in most cases applying shortcutting or using mixed merge clauses result in the same generated learnt clause. Thus, although creating a mixed merge clause results in a small increase in conflict clause construction time, no additional time was spent unnecessary propagating assignments. Therefore the performance increase, even though shortcutting was widely applied, was only minimal for this class of graphs.

With respect to the other used instances, the medium graphs and the DIMACS instances, as there was a gap between the chromatic number and the found clique size, shortcutting was rarely used. Even if conflicts would occur 'near' the forced clique there would always be large numbers of symmetries which would not be broken as the number of colors involved in the conflict were larger than the set clique.

All considered, shortcutting as an independent addition to MiniMerge2 did not prove useful.

6. The impact of preferential merge literal selection in MiniMerge2

We have performed tests to benchmark the effectiveness of preferential merge literal selection. These tests seem to indicated that there is no evidence that choosing merge variables which have the highest neighborhood scores as starting merge literals yield better results than not doing so. However especially on SAT instances the performance on individual instances is hugely effected, either positive or negative, instances could suddenly take as 10 times as long or be solved in an instance.

7. On previously published results of MiniMerge2

In [42] we published results of the performance of our solver MiniMerge2 on the same medium graphs and DIMACS benchmarks under the name MiniColor, these results do not match the results presented in this Chapter. This difference is due to the fact that we used both the shortcutting and preferential merge literal selection options when running this solver, therefore the results differ

but are of the same order, although the MiniColor distribution was able to prove that DSJC125.9 is 45-colorable, a result that we have been unable to prove in this thesis.

Chapter 12

Conclusions and Future Work

1. Conclusions

In this thesis we showed how a SAT conflict-driven algorithm can be adapted for the graph coloring problem by converting conflict clauses in such a way that they cover all permutations of the colors in the conflicting assignment, achieving full symmetry breaking in that respect. Our technique is flexible and can be used in combination with other optimizations for graph coloring such as adding symmetry breaking predicates through the forcing of cliques. In fact, the best performances of our technique are achieved by this combination.

Our conversion is loosely based on extended resolution. Although the extended resolution proof system is 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 our case, the branches in the Zykov-tree. Second, reusage of introduced (merge) variables is crucial to contain an explosion of useless variables. 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 keep down the time spent propagating assignments to make the technique competitive.

The performance increases we can theoretically achieve for the SAT based approaches for the GCP are enormous ($k!$) and on simple instances without symmetry breaking we indeed achieve this theoretical performance increase. Our technique does not show this exponential speedup for most other instances, even the Mycielski graphs that only contain cliques of size 2 are not solved exponentially quicker than by MiniSat2. We believe the main reason for this gap is that clauses generated by our technique are not always arc-consistent. Even though we do not gain the full theoretical speedup we still achieve large speedups, provided that is k is larger than 3, otherwise the technique presented in [10] is more appropriate.

We used our technique to solve Van der Waerden instances however these instances proved to be either too simple or too difficult to prove the effectiveness of our technique for this class of problems.

Although the mixed merge clauses used in MiniMerge2 yielded positive results in the form of a speedup between a factor 1.5 and 2, we believe this optimization is too artificial, as it significantly alters solver operations, to be useful in the long term. We believe it can easily be made obsolete with other improvements such as only adding adding the necessary support clauses or other ways of clause manipulation.

2. Future Work

2.1 Solving the lack of arc-consistenty in merge clauses

The lack of arc-consistency of the clause transformation by MiniMerge is it's largest point of weakness and should be the first point to be addressed in future research. As stated it can occur that more than 50% of the clauses generated by MiniMerge are arc-inconsistent and we believe that an enormous potential for the improvement of MiniMerge exists in solving this problem.

2.2 Finding new problems for which our technique can be successfully applied

At this point, our presented technique only yields positive results for GCP instances, but we have shown that we can encode and solve other combinatorial problems, such as the Van der Waerden instances. We expect that many multi-valued SAT problems can be attacked using our technique. In particular those consisting of constraints in which variables should either have the same value. A second avenue of research should be finding new problems in which we can apply our approach for example computing Schur Numbers. The general usefulness of our approach will depend heavily on whether it can be applied successfully for a large class of problem.

2.3 Properly implementing adding support clauses

Our current implementation of MiniMerge does not support only adding the strictly necessary support clauses to $\mathcal{F}$, as shown in Chapter 6. We believe that we can gain a speedup of the same size as the speedup achieved by the mixed merge clauses in MiniMerge2 by proper implementing the adding of support clauses.

Bibliography

[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] Sipser, Michael, *Introduction to the Theory of Computation*. International Thomson Publishing, 1996.

[3] 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.

[4] Stephen A. Cook, *A short proof of the pigeonhole principle using extended resolution. SIGACT News*. SIGACT News **8**(4):28–32, 1976.

[5] Stephen A. Cook, *Feasibly constructive proofs and the propositional calculus*. In proceedings of STOC '75. pp. 83–97, 1975.

[6] Martin Davis, G. Logemann, and D. Loveland, *A machine program for theorem proving*. Communications of the ACM, 5(7) pp. 394-397, July 1962.

[7] Martin Davis and Hilary Putnam, *A Computing Procedure for Quantification Theory*. Journal of the ACM **7**(3):201–215, 1960.

[8] Niklas Eén and Niklas Sörensson, *An Extensible SAT-solver*. In proceedings of SAT 2003, LNCS **2919**:502–518, 2003.

[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] J. A. Robinson, *A machine-oriented logic based on the resolution principle*. Journal of the ACM **12**(1):23–41, 1965.

[16] Karem A. Sakallah, *Symmetry and Satisfiability*. Chapter 10 of Handbook of Satisfiability, pp 289–338, 2009.

[17] Carsten Sinz and Armin Biere, *Extended Resolution Proofs for Conjoining BDDs*. In Proc. of CSR06, LNCS **3967**:600-611, 2006.

[18] G. Tseitin, *On the complexity of derivation in propositional calculus*. Studies in Mathematics and Mathematical Logic, Part II. pp. 115-125, 1968.

[19] Alasdair Urquhart, *Hard examples for resolution*. Journal of the ACM **34**(1):209–219, 1987.

[20] Allen Van Gelder, *Another look at graph coloring via propositional satisfiability*. Discrete Applied Mathematics **156**(2):230–243, 2008.

[21] A. A. Zykov, *On some properties of linear complexes*. Amer. Math. Soc.Translations 79 p. 81, 1952.

[22] Availiable at http://mat.gsia.cmu.edu/COLOR/solvers/trick.c.

[23] Computational Series: Graph Coloring and Its Generalizations.
http://mat.gsia.cmu.edu/COLOR04.

[24] Zhang, Lintao and Madigan, Conor F. and Moskewicz, Matthew H. and Malik, Sharad, *Efficient conflict driven learning in a boolean satisfiability solver*. In proceedings of ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design pp. 279–285

[25] Anuj Mehrotra and Michael A. Trick, *A column generation approach for graph coloring*. INFORMS Journal on Computing Volume 8 Number 4 pp 344–354, 1996.

[26] Brélaz, Daniel, *New methods to color the vertices of a graph*. Commun. ACM Volume 22 Number 4 pp. 251–256, 1979.

[27] Brigham, R. D. and Dutton, R. D., *A new graph coloring algorithm*. The Computer Journal 24 pp 85-86, 1981.

[28] Leighton, F. T., *A graph coloring algorithm for large scheduling problems*. Journal of Research of the National Bureau of Standards 84, 1979,

[29] M.R. Dransfield, L. Liu, V. Marek, M. Truszczynski, *Satisfiability and Computing van der Waerden numbers*. The Electronic Journal of Combinatorics, vol. 11 (1), pp 489-503, 2004,

[30] N. Sörensson and N. Eén, *MiniSat vl.13 - a SAT solver with conflict-clause minimization*. In Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT 2005), St. Andrews, UK, June 2005. Springer-Verlag. LNCS 3569.

[31] D. G. Corneil and B. Graham *An Algorithm for Determining the Chromatic Number of a Graph*. SIAM J. Comput. Volume 2, Issue 4, pp. 311-318, 1973.

[32] Marijn J.H. Heule, *SmArT solving: Tools and techniques for satisfiability solvers* Availiable at http://www.st.ewi.tudelft.nl/sat/theses/heule_phd.pdf.

[33] Méndez-Díaz,, Isabel and Zabala,, Paula, *A branch-and-cut algorithm for graph coloring*. Discrete Appl. Math. Volume 154 Issue 5, pp. 826-847, 2006.

[34] Méndez-Díaz,, Isabel and Zabala,, Paula, *A cutting plane algorithm for graph coloring* Discrete Appl. Math. Volume 156, Issue 2, pp. 159-179, 2008.

[35] J. Crawford, *A theoretical analysis of reasoning by symmetry in first order logic*. in Proc. AAAI Workshop Tractable Reasoning, 10th Nat. Conf. Artif. Intell., San Jose, CA, 1992.

[36] J. Crawford, M. Ginsberg, E. Luks, and A. Roy, *Symmetry-breaking predicates for search problems*. in Proc. 5th Int. Conf. Principles Knowledge Representation Reasoning, Cambridge, MA, pp.148159, 1996.

[37] Hillier, Frederick S. and Lieberman, Gerald J, *Introduction to Operations Research Eight Edition*. McGraw-Hill Science/Engineering/Math, 2004.

[38] J. Köbler, U. Schöning, and J. Torán, *Graph isomorphism is low for PP Computat*. Complex., vol. 2, no. 4, pp. 301330, 1992.

[39] M. Hall Jr., *The Theory of Groups*. New York: Macmillan, 1959.

[40] http://mathworld.wolfram.com/MycielskiGraph.html.

[41] Ian P. Gent. *Arc Consistency in SAT*. In Proceedings of the Fifteenth European Conference on Artificial Intelligence (ECAI 2002), 2002.

[42] Bas Schaafsma and Marijn J.H. Heule and Hans van Maaren, *Dynamic Symmetry Breaking by Simulating Zykov Contraction*, In Proceedings of SAT 2009, Twelfth International Conference on Theory and Applications of Satisfiability Testing, 30 June - 3 July 2009. Lecture Notes in Computer Science, vol. 5584, pp. 223-236, 2009.

Appendices

1. DP-Resolution example 1

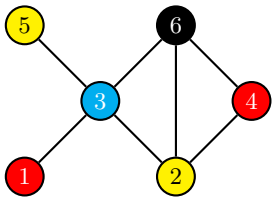


Figure 1. A graph coloring instance. The numbers in the vertices refer to their index v_i. Vertex v_6 is in conflict because it cannot be colored. Although various reasons can be addressed for this conflict, the example focusses on the assignmen.t to v_1, v_4 and v_5.

In the corresponding SAT instance of Figure 1, a conflict clause C_{conflict} for this conflict would be:

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

Converting C_{conflict} results in C_{merge}:

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

$M(C_{\text{merge}})$:

$$\begin{array}{ll}
(e_{1,4} \vee \neg x_{1,1} \vee \neg x_{4,1}) & \wedge \\
(e_{1,4} \vee \neg x_{1,2} \vee \neg x_{4,2}) & \wedge \\
(e_{1,4} \vee \neg x_{1,3} \vee \neg x_{4,3}) & \wedge \\
 & \\
(\neg e_{1,5} \vee \neg x_{1,1} \vee x_{5,1}) & \wedge \\
(\neg e_{1,5} \vee \neg x_{1,2} \vee x_{5,2}) & \wedge \\
(\neg e_{1,5} \vee \neg x_{1,3} \vee x_{5,3}) & \wedge \\
 & \\
(\neg e_{1,5} \vee x_{1,1} \vee \neg x_{5,1}) & \wedge \\
(\neg e_{1,5} \vee x_{1,2} \vee \neg x_{5,2}) & \wedge \\
(\neg e_{1,5} \vee x_{1,3} \vee \neg x_{5,3}) &
\end{array} \tag{3}$$

Applying DP-Resolution on the literal $e_{1,4}$ on the set of clauses $M(C_{\text{merge}})$ and C_{merge} results in the new clauses:

$$\begin{array}{ll}
(e_{1,5} \vee \neg x_{1,1} \vee \neg x_{4,1}) & \wedge \\
(e_{1,5} \vee \neg x_{1,2} \vee \neg x_{4,2}) & \wedge \\
(e_{1,5} \vee \neg x_{1,3} \vee \neg x_{4,3}) &
\end{array} \tag{4}$$

Applying resolution to create all possible $e_{1,5} \vee \neg x_{1,i} \vee \neg x_{5,j}$, $i \neq j$ results in:

$$\begin{array}{ll}
(\neg e_{1,5} \vee \neg x_{1,1} \vee \neg x_{5,2}) & \wedge \\
(\neg e_{1,5} \vee \neg x_{1,1} \vee \neg x_{5,3}) & \wedge \\
(\neg e_{1,5} \vee \neg x_{1,2} \vee \neg x_{5,1}) & \wedge \\
 & \\
(\neg e_{1,5} \vee \neg x_{1,2} \vee \neg x_{5,3}) & \wedge \\
(\neg e_{1,5} \vee \neg x_{1,3} \vee \neg x_{5,1}) & \wedge \\
(\neg e_{1,5} \vee \neg x_{1,3} \vee \neg x_{5,2}) &
\end{array} \tag{5}$$

Applying DP-Resolution on the positive literal $e_{1,5}$ on the clauses in (4) and (5) results in:

$$
\begin{array}{ll}
(\neg x_{1,1} \vee \neg x_{4,1} \vee \neg x_{5,2}) & \wedge \\
(\neg x_{1,1} \vee \neg x_{4,1} \vee \neg x_{5,3}) & \wedge \\
(\neg x_{1,1} \vee \neg x_{4,1} \vee \neg x_{1,2} \vee \neg x_{5,1}) & \wedge \\
(\neg x_{1,1} \vee \neg x_{4,1} \vee \neg x_{1,2} \vee \neg x_{5,3}) & \wedge \\
(\neg x_{1,1} \vee \neg x_{4,1} \vee \neg x_{1,3} \vee \neg x_{5,1}) & \wedge \\
(\neg x_{1,1} \vee \neg x_{4,1} \vee \neg x_{1,3} \vee \neg x_{5,2}) & \wedge \\
 & \\
(\neg x_{1,2} \vee \neg x_{4,2} \vee \neg x_{1,1} \vee \neg x_{5,2}) & \wedge \\
(\neg x_{1,2} \vee \neg x_{4,2} \vee \neg x_{1,1} \vee \neg x_{5,3}) & \wedge \\
(\neg x_{1,2} \vee \neg x_{4,2} \vee \neg x_{5,1}) & \wedge \\
(\neg x_{1,2} \vee \neg x_{4,2} \vee \neg x_{5,3}) & \wedge \\
(\neg x_{1,2} \vee \neg x_{4,2} \vee \neg x_{1,3} \vee \neg x_{5,1}) & \wedge \\
(\neg x_{1,2} \vee \neg x_{4,2} \vee \neg x_{1,3} \vee \neg x_{5,2}) & \wedge \\
 & \\
(\neg x_{1,3} \vee \neg x_{4,3} \vee \neg x_{1,1} \vee \neg x_{5,2}) & \wedge \\
(\neg x_{1,3} \vee \neg x_{4,3} \vee \neg x_{1,1} \vee \neg x_{5,3}) & \wedge \\
(\neg x_{1,3} \vee \neg x_{4,3} \vee \neg x_{1,2} \vee \neg x_{5,1}) & \wedge \\
(\neg x_{1,3} \vee \neg x_{4,3} \vee \neg x_{1,2} \vee \neg x_{5,3}) & \wedge \\
(\neg x_{1,3} \vee \neg x_{4,3} \vee \neg x_{5,1}) & \wedge \\
(\neg x_{1,3} \vee \neg x_{4,3} \vee \neg x_{5,2}) &
\end{array}
\tag{6}
$$

The removal of the subsumed clauses in (6) results in:

$$
\begin{array}{ll}
(\neg x_{1,1} \vee \neg x_{4,1} \vee \neg x_{5,2}) & \wedge \\
(\neg x_{1,1} \vee \neg x_{4,1} \vee \neg x_{5,3}) & \wedge \\
(\neg x_{1,2} \vee \neg x_{4,2} \vee \neg x_{5,1}) & \wedge \\
(\neg x_{1,2} \vee \neg x_{4,2} \vee \neg x_{5,3}) & \wedge \\
(\neg x_{1,3} \vee \neg x_{4,3} \vee \neg x_{5,1}) & \wedge \\
(\neg x_{1,3} \vee \neg x_{4,3} \vee \neg x_{5,2}) &
\end{array}
\tag{7}
$$

The clauses in (7) forbid all permutation of the conflicting assignment shown in Figure (1):

2. List of Symbols

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.

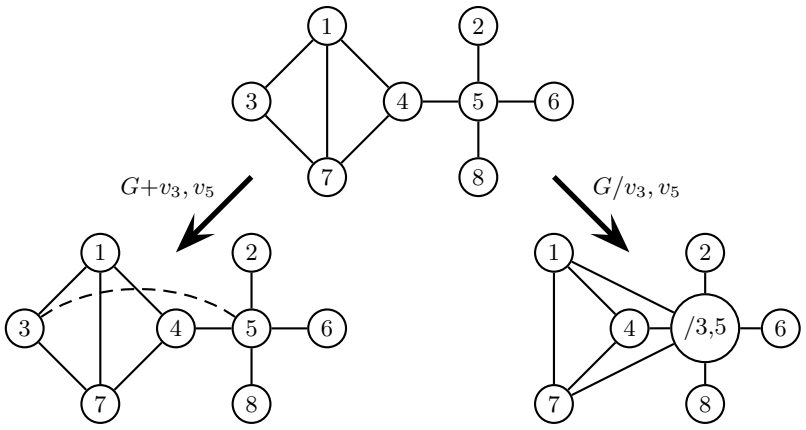


Fig. 1. A Zykov-tree example. The numbers in the vertices refer to their index v_i.

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 Tsietin 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].

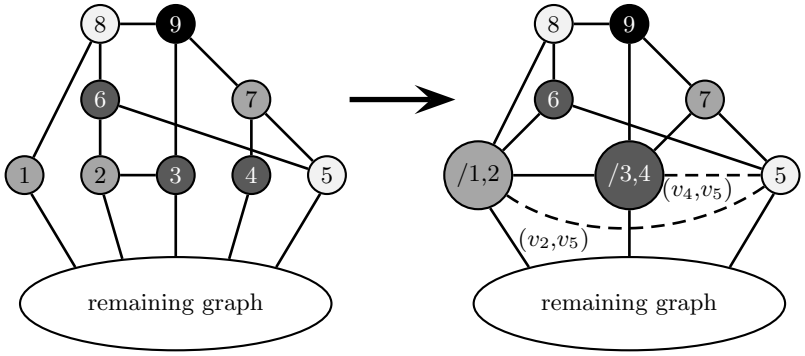


Fig. 2. A Zykov Contraction example. The numbers in the vertices refer to the index v_i. The added edges are shown as dashed lines. Vertex v_9 is in conflict because it cannot be colored. The example focusses on the assignment to v_1, v_2, v_3, v_4, and v_5.

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:

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

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 wekness. 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 $\pi : (1, \ldots, k) \rightarrow (1, \ldots, k)$ 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

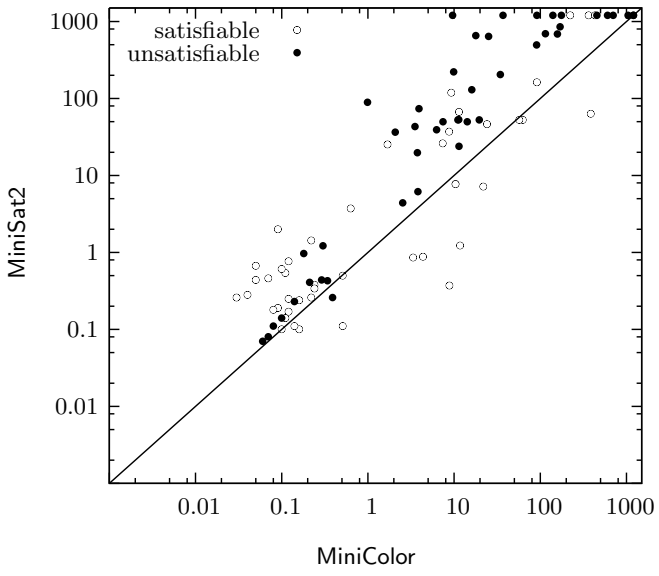


Fig. 3. Performance comparison between MiniColor and MiniSat2 on medium sized random graphs, with a 1200 (s) timeout.

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 runtimes 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 runtimes 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 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`.