Computers & Operations Research 51 (2014) 282–301

Contents lists available at ScienceDirect

Computers & Operations Research

journal homepage: www.elsevier.com/locate/caor

An exact algorithm with learning for the graph coloring problem ☆

Zhaoyang Zhou [a,c], Chu-Min Li [a,b], Chong Huang [a], Ruchu Xu [a,*]

[a] School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
[b] MIS, Université de Picardie Jules Verne, 33 Rue St. Leu 80039 Amiens Cedex, France
[c] Library, Hubei University, Wuhan, China

ARTICLE INFO

Available online 24 May 2014

Keywords:
Backtracking
Clause learning
Graph Coloring
SAT

ABSTRACT

Given an undirected graph $G=(V,E)$, the Graph Coloring Problem (GCP) consists in assigning a color to each vertex of the graph G in such a way that any two adjacent vertices are assigned different colors, and the number of different colors used is minimized. State-of-the-art algorithms generally deal with the explicit constraints in GCP: any two adjacent vertices should be assigned different colors, but do not specially deal with the implicit constraints between non-adjacent vertices implied by the explicit constraints. In this paper, we propose an exact algorithm with learning for GCP which exploits the implicit constraints using propositional logic. Our algorithm is compared with several exact algorithms among the best in the literature. The experimental results show that our algorithm outperforms other algorithms on many instances. Specifically, our algorithm allows to close the open DIMACS instance 4-Fullins_5.

© 2014 Published by Elsevier Ltd.

1. Introduction

Given an undirected graph $G=(V, E)$, where V is a set of vertices and E a set of unordered pairs of vertices called edges $\{(v_i, v_j)|v_i \in V, v_j \in V\}$, the Graph Coloring Problem (GCP) consists in assigning a color to each vertex of the graph G in such a way that any two adjacent vertices are assigned different colors, and the number of different colors used is minimized. The minimum number of colors needed to color G is called the *chromatic number* of G, denoted by $\chi(G)$. Deciding whether a graph can be colored using at most k colors is referred as *k-colorability* problem, which is the decision version of GCP.

GCP is one of the most studied NP-hard combinatorial optimization problems. It is interesting not only for studying the computational complexity but also for practical applications. Many real world problems can be naturally encoded into GCP and solved using a GCP algorithm. Examples of such problems include scheduling [1,2], timetable [3,4], register allocation [5], frequency assignment [6], and communication networks [7].

Algorithms for GCP generally fall into two categories: exact algorithms (see e.g., [8–12]) and approximate algorithms. Approximate algorithms include construction method [13], local search [14], evolutionary algorithms based on populations [15,16], hybrid algorithms [17], and other heuristic methods [18]. Because of its hardness, most algorithms for GCP in the literature are approximate and few are exact. Indeed, a GCP encoding a real application problem is often very large and can be out of the reach of exact algorithms, because exact coloring algorithms generally need to enumerate the search space of a problem and have difficulties to solve large instances. However, approximate algorithms are usually unable to prove the optimality of their solutions. Especially, they are usually unable to prove the unfeasibility of a k-colorability problem. So exact algorithms remain very useful and indispensable. In this paper, we focus on exact algorithms for GCP.

The constraints in a GCP are very easy to express and understand, saying that any two adjacent vertices cannot be assigned the same color. However, these simple explicit constraints between adjacent vertices imply many complex constraints among non-adjacent vertices. It appears that most state-of-the-art GCP algorithms do not specially detect and exploit these implicit constraints. One way to discover and exploit the implicit constraints is to encode a GCP using k colors into the Boolean satisfiability problem (SAT) by expressing the explicit constraints in CNF (Conjunctive Normal Form) clauses. Then modern SAT algorithms discover and exploit the implicit constraints among non-adjacent vertices using Conflict Driven Clause Learning (CDCL). However, the SAT instance encoding the GCP generally is very large, limiting the size of the GCP that can be solved in this way.

☆This work is supported by National Natural Science Foundation of China (NSFC). (Grant No. 61070235)

* Corresponding author.
E-mail addresses: authar@163.com (Z. Zhou), chu-min.li@u-picardie.fr (C.-M. Li), jason_hch@hotmail.com (C. Huang), xrcy0315@sina.com (R. Xu).

http://dx.doi.org/10.1016/j.cor.2014.05.017
0305-0548/© 2014 Published by Elsevier Ltd.

In this paper, we propose a new exact algorithm for GCP based on a backtracking schema. The explicit constraints of a GCP are not encoded into SAT. At the beginning, every vertex has the set of k available colors. Vertices are assigned a color from their sets of available colors one by one. When a color c is assigned to a vertex v, c is removed from the set of available colors of all vertices adjacent to v. Each time a dead-end is encountered, i.e., when the set of available colors of a vertex becomes empty, the Conflict Driven Clause Learning (CDCL) schema in modern SAT solvers is adapted to discover an implicit constraint among vertices. This implicit constraint is used in subsequent search to reduce the search space.

The paper is organized as follows. Section 2 presents the related work. Section 3 presents the motivation of our approach. Section 4 presents the new backtracking algorithm and discusses in detail the clause learning schema in our algorithm adapted from modern SAT solvers. Section 5 empirically studies the clause management strategies and the distribution and impact of the learnt clauses in the real coloring process, and shows the effectiveness of CDCL by comparing our algorithms (with and without clause learning) with other algorithms on standard GCP benchmarks. Section 6 concludes the paper.

2. Related work

GCP can be expressed using integer programming and then solved using a software such as CPLEX. For each vertex v and each color c, let v_c be a binary variable that evaluates to 1 if and only if (iff) v is assigned color c. The decision version of GCP, i.e., the k-colorability problem, can be naturally expressed using $n*k$ binary variables as follows:

$$\sum_c v_c = 1, \quad \forall\ v \in V \tag{1}$$

$$u_c + v_c \le 1, \quad \forall\ (u,v) \in E \text{ and } \forall\ c \tag{2}$$

Eq. (1) says that each vertex is assigned exactly one color, and Eq. (2) says that two adjacent vertices cannot be assigned the same color. In order to minimize the number of colors, one can consider n colors and use n additional binary variables y_c $(1 \le c \le n)$ that evaluate to 1 iff color c is used. The integer programming problem becomes

$$\min \sum_c y_c$$

subject to Eq. (1) and

$$u_c + v_c \le y_c, \quad \forall\ (u,v) \in E \text{ and } \forall\ c \tag{3}$$

The number of binary variables in the above integer programming formulations of GCP is in $O(n^2)$. Mehrotra and Trick proposed a formulation using up to an exponential number of variables in [19]. Let S be the set of all independent sets of G, a binary variable x_s is associated to each independent set s that evaluates to 1 iff all vertices in s are assigned the same color. The corresponding integer programming formulation of GCP becomes

$$\min \sum_{s \in S} x_s$$

subject to

$$\sum_{v \in s, s \in S} x_s \ge 1, \quad \forall\ v \in V \tag{4}$$

An alternative solving approach of GCP is to enumerate all possible vertex colorings of G following a branch-and-bound schema. Let G be a graph of n vertices such that each vertex v is associated with a set $C_v = \{1, 2, \ldots, n\}$ of n available colors, $G \backslash v$ be G after removing the vertex v and all incident edges of v, and let function max(a, b) denote the largest value between a and b. Algorithm 1 returns the chromatic number $\chi(G)$ after enumerating all possible colorings of G in a backtracking tree.

Algorithm 1. backGCP(G, k, UB), a backtracking algorithm for GCP.

```
Input: A graph $G=(V, E)$, the largest color $k$ used so far in the current partial coloring solution, and the smallest number $UB$ of colors
    used in a complete coloring solution of $G$ so far
Output: the chromatic number $\chi(G)$ of $G$
begin
  if $|V| = 0$ then
    return $k$;
  if there is a vertex $v$ such that $C_v$ does not contain any color less than $UB$ then
    return $UB$;
  select a vertex $v$ from $G$; /* branching */
  for each color $c$ in $C_v$ such that $c < UB$ do
    remove $c$ from $C_u$ of each vertex $u$ adjacent to $v$;
    $UB = \text{backGCP}(G \backslash v, \max(k, c), UB)$;
    insert $c$ to $C_u$ of each vertex $u$ adjacent to $v$;
  return $UB$
end
```

Algorithm 1 should be called with backGCP(G, 0, $n+1$) to search for the chromatic number of G. At the beginning, $k=0$, i.e., the largest color used is 0 (no color is used at the beginning), the smallest number UB of colors used in a complete coloring solution so far is initialized to $n+1$, and all the vertices have the same set of n available colors. The algorithm selects a vertex v, and for each available color c of v smaller than UB, assigns c to v, i.e., removes c from the adjacent vertices of v, and recursively calls the algorithm to color the remaining vertices of G. If G does not contain any remaining vertex (i.e., $G\backslash v$ is empty), a complete coloring solution is obtained and the largest color used in the solution becomes the new *UB*. Otherwise, if there exists a vertex containing no available color smaller than *UB*, the current partial coloring solution cannot be completed, i.e., *UB* cannot be improved, and the algorithm backtracks to the last vertex having at least one available color smaller than *UB*.

Observe that the algorithm backGCP can be easily simplified or adapted to solve the k-colorability problem. It suffices to call the algorithm using backGCP(G, 0, $k+1$) and stop the **For** loop as soon as a recursive call returns a *UB* smaller than or equal to k.

The choice of vertex v for recursive calls is very important for the performance of backGCP, because it determines the total number of recursive calls (i.e., the size of the search tree). A famous heuristic called DSATUR [20] selects the vertex v with the smallest available set C_v of colors less than *UB*, breaking ties in favor of the vertex with the most adjacent non-colored vertices. In CSP (Constraint Satisfaction Problem) terms, DSATUR corresponds to the heuristic dom+deg used in a backtracking algorithm for CSP that branches on the variable with the smallest value domain, breaking ties in favor of the variable with the maximum degree in the constraint graph. Obviously the number of available colors less than *UB* of the branching vertex in backGCP is the number of subproblems to solve. The intuition of DSATUR is to generate as few subproblems as possible and to remove as much available colors as possible from the non-colored vertices.

3. Motivation of our approach

GCP is defined only using explicit constraints between adjacent vertices. However, these explicit constraints may imply many implicit constraints among non-adjacent vertices. For example, when coloring the simple graph in Fig. 1 using two colors, v_1 and v_4 cannot be assigned the same color although they are non-adjacent. This constraint between non-adjacent vertices is only implicit and is implied by explicit constraints between adjacent vertices. Implicit constraints can be very useful to prune the search space. For example, in Fig. 1, when vertex v_1 is assigned a color, this color should be removed not only from v_2, but also from v_4 according to the implicit constraint. However, it appears that most state-of-the-art GCP algorithms do not specifically detect and exploit these implicit constraints.

One way to discover the relevant implicit constraints in the k-colorability problem is to encode it into a propositional satisfiability (SAT) problem, since the popular Conflict Driven Clause Learning (CDCL) in modern SAT solvers such as Minisat [21] allows to discover these constraints. Let G have n vertices and m edges, and the k colors be numbered 1, 2, …, k. The usual encoding uses $n*k$ Boolean variables, each variable being named v_c which takes the truth value 1 iff the vertex v is assigned the color c $(1 \le c \le k)$. A literal is either v_c or $\overline{v}_c$. A positive (negative) literal such as v_c $(\overline{v}_c)$ is satisfied iff $v_c=1(0)$. For each vertex v, a CNF clause of length k, $v_1 \vee v_2 \vee \ldots \vee v_k$, is defined to say that v should be assigned a color, and $k \times (k-1)/2$ binary clauses $\overline{v}_i \vee \overline{v}_j$ $(1 \le i < j \le k)$ are defined to say that at most one color is assigned to v. For each edge (u, v) and each color c, k binary clauses $\overline{u}_c \vee \overline{v}_c$ $(1 \le c \le k)$ are defined to say u and v cannot both be assigned color c. The $m \times k + (k \times (k-1)/2+1) \times n$ clauses constitute a SAT instance encoding the k-colorability instance. A CNF clause is satisfied iff at least one of its literals is satisfied. A truth value assignment of the Boolean variables satisfying all the clauses clearly corresponds to a complete coloring solution of G.

Observe that in a hard GCP instance, m may be of $O(n^2)$. So, the number of clauses in the SAT instance generated using the usual encoding may be huge for hard GCP with large n and large k.

Note that one might use a CDCL SAT solver to directly solve the SAT instance encoding a GCP instance, since the CDCL technique in the SAT solver allows to discover and exploit the implicit constraints. Nevertheless, the explicit and implicit constraints in GCP have very complex relationships among them due to the NP-hardness of GCP, and a CDCL SAT solver implements a general-purpose SAT algorithm that is not aware of the precise meaning of any boolean variable and of any clause, so that the effective DSATUR branching heuristic cannot be easily used in the SAT solver. In addition, the learning may not be as precise as possible and the exploitation of the learnt implicit constraints may not be as effective as possible. Moreover, there are a huge number of clauses in the SAT instance and learnt during search, which are hard to manage. The most effective way to solve GCP might be then to adapt CDCL into a dedicated backtracking algorithm for GCP such as backGCP. Example 1 suggests how backGCP reinforced by CDCL works.

Example 1. Refer to Fig. 1, $C_v=\{1, 2\}$ for all the vertices at the beginning. As backGCP proceeds, if a vertex v_i is assigned a color c, the corresponding boolean variable v_{ic} is assigned 1, and if a color c is removed from a vertex v_i, the corresponding boolean variable v_{ic} is assigned 0. Let v_1 be selected for branching in Algorithm backGCP and assigned color 1 $(v_{11}=1)$, a consequence is $v_{21}=0$. Then let v_4 be selected for branching and assigned color 1 $(v_{41}=1)$, a consequence is $v_{31}=0$. Then v_2 should be assigned color 2 $(v_{22}=1)$, because color 1 has been removed from it after its adjacent vertex v_1 is assigned color 1. Now no color is available for v_3, i.e., $v_{31}=0$ and $v_{32}=0$. In SAT term, we have an empty clause derived from $v_{31} \vee v_{32}$ representing a dead-end. Driven by this dead-end, the CDCL technique analyzes all reasons of the dead-end to find the most relevant reasons, which can roughly be illustrated as follows.

The last removed color in v_3 is color 2, i.e., the last falsified literal in $v_{31} \vee v_{32}$ is v_{32}. The literal v_{32} is falsified because the adjacent vertex v_2 of v_3 is assigned color 2 $(v_{22}=1)$, which is in turn because $v_{21}=0$. We re-write the empty clause as $v_{31} \vee v_{21}$, replacing v_{32} with its falsification reason $v_{21}=0$.

In the new empty clause $v_{31} \vee v_{21}$, the last falsified literal is v_{31}, which is because $v_{41}=1$. The assignment $v_{41}=1$ is a branching decision and does not have reason. We re-write the empty clause as $\overline{v}_{41} \vee v_{21}$, where the last falsified literal is v_{21} and it is falsified by $v_{11}=1$, a branching decision. Finally we have the empty clause $\overline{v}_{41} \vee \overline{v}_{11}$, meaning that $v_{41}=1 \wedge v_{11}=1$ results in a dead-end in the coloring process. Similarly, we have $\overline{v}_{42} \vee \overline{v}_{12}$. The two clauses say that the two non-adjacent vertices v_1 and v_4 cannot have the same color.

4. A conflict driven clause learning algorithm for GCP

We focus on backGCP for the k-colorability problem and reinforce it with Conflict Driven Clause Learning (CDCL) to obtain a new algorithm called cdclGCP. Algorithm 2 gives the pseudo-code of cdclGCP. In cdclGCP, explicit constraints between adjacent vertices are not encoded into SAT and are always represented in its original form as in backGCP. When a dead-end is encountered, CDCL is called to deduce

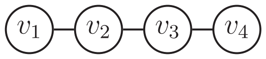

Fig. 1. A simple graph.

an implicit constraint from the dead-end using the explicit constraints (in their original form) and possibly previous implicit constraints. All implicit constraints discovered during search using CDCL are represented as CNF clauses. We present the general structure of cdclGCP in the first subsection, and some detailed aspects of cdclGCP in the remaining subsections.

4.1. General structure of cdclGCP

Given an input graph G and k colors, each vertex v of G is associated with a set C_v of k available colors, and cdclGCP is executed with cdclGCP(G, $\emptyset$, 0). Differently from backGCP, cdclGCP constructs during search the set of implicit constraints *implicitConstraints* and uses it to prune the search space.

Algorithm 2. cdclGCP(G, $\mathcal{A}$, *level*), a CDCL algorithm for GCP.

```
Input: A graph $G=(V, E)$, the current partial coloring solution $\mathcal{A}$, the current recursive level level
Output: $\mathcal{A}$, a coloring solution of $G$ if it exists, $\emptyset$ otherwise
begin
  if $|V|=0$ then
    return $\mathcal{A}$;
  unitPropagation($G$, $\mathcal{A}$, level);
  if there is a vertex $v$ such that $C_v$ is empty or there is an empty clause then
    reason:=analyzeConflict();
    if reasonisemptythen
      return $\emptyset$;
    else
      implicitConstraints ← implicitConstraints $\cup$ {reason};
      backtracking(secondLargestLevel(reason));
  select a vertex $v$ from $G$ using the dom+deg (or DSATUR) heuristic;
  for each color $c$ in $C_v$ do
    remove $c$ from $C_u$ of each vertex $u$ adjacent to$v$;
    record $\{v \leftarrow c\}$ as the reason of each removal in level $level+1$;
    $\mathcal{A}$:=cdclGCP($G \backslash v$, $\mathcal{A} \cup \{v \leftarrow c\}$, $level+1$);
    if $\mathcal{A} \neq \emptyset$ then
      return $\mathcal{A}$; /*A coloring solution is found */
    insert $c$ to $C_u$ of each vertex $u$ adjacent to $v$;
  return $\emptyset$;
end
```

Algorithm 3. unitPropagation(G, $\mathcal{A}$, *level*).

```
Input: A graph $G=(V, E)$, the current partial coloring solution $\mathcal{A}$, the current recursive level level
begin
  while(there is a unit clause or a unit vertex and there is neither empty clause nor empty vertex) do
    if there is a unit clause $l$ then
      remove $\neg l$ from all clauses containing it;
      remove all clauses containing $l$;
      if $l$ is a positive literal $v_c$ then
        remove $c$ from all vertices adjacent to $v$;
        $\mathcal{A}$:=$\mathcal{A} \cup \{v \leftarrow c\}$;
      else
        remove color $c$ from $C_v$; /* $l$ is a negative literal $\neg v_c$ */
      record the clause $l$ as the reason of the falsification of literal $\neg l$ and all color removals in the level level;
    if there is a unit vertex $v$ with the unique color $c$ then
      remove $c$from all vertices adjacent to $v$;
      remove $\neg v_c$ from all clauses containing it;
      remove all clauses containing $v_c$;
      record $\{v \leftarrow c\}$ as the reason of all the removals in the level level;
      $\mathcal{A}$:=$\mathcal{A} \cup \{v \leftarrow c\}$;
  return;
end
```

During search, at every search tree node, cdclGCP tries to extend the current partial coloring solution $\mathcal{A}$ by adding new colored vertices into $\mathcal{A}$. All extensions realized in the current tree node are associated with the current recursive level, that roughly corresponds to the number of branchings from the root of the tree to the current tree node. Note that the DSATUR branching heuristic is always used to select the branching vertex as in backGCP, which would be hardly possible in a SAT solver. Compared with backGCP, cdclGCP uses three new procedures: unitPropagation, analyzeConflict, and backtracking, in order to discover implicit constraints added into *implicitConstraints* and to exploit them for pruning the search space.

The available color set C_v of a non-colored vertex v becomes smaller and smaller, as the partial coloring solution of G is extended, because colors assigned to adjacent vertices of v are removed from C_v. When C_v contains only one color, v is called a *unit vertex*; and if C_v becomes empty, v is called an *empty vertex* and represents a conflict, meaning that v cannot be colored. Similarly, when a literal is assigned false, it is also removed from the clauses containing it, because it cannot satisfy these clauses. When a clause contains only one literal, it is called a *unit clause*; and when it contains no literal (i.e., all its literals become false and have been removed), it is called an *empty clause* and also represents a conflict.

When a clause or a vertex becomes unit, there is only one possibility to satisfy the clause or to color the vertex. This possibility is used and propagated to possibly generate other unit clauses and/or unit vertices, using the procedure *unit propagation* (see Algorithm 3). When a conflict occurs, i.e., an empty vertex or an empty clause is produced, an analysis of the conflict is done to derive a CNF clause representing the reason of the conflict, using the *analyzeConflict* procedure. The derived clause is added into *implicitConstraints*. See the next subsection for details of the conflict analysis.

The derived clause is a set of literals assigned the value false (0). Each false literal is associated with the recursive level where it was assigned false. The reason of the current conflict only lies in these levels: it is the falsification of these literals that results in a conflict. So, the backtracking procedure backjumps to the second largest recursive level indicated in the derived clause. For example, if the derived clause is $l_1 \vee l_2 \vee l_3 \vee l_4$, in which the four literals are falsified respectively in levels 2, 4, 6, and 10, the backtracking procedure should backjump to level 6 (without undoing any assignment of level 6), so that the derived clause becomes a unit clause l_4 (other literals in the clause remain false after backjumping), pruning subtrees in levels 7, 8, 9, since literals assigned or vertices colored in levels 7, 8 and 9 have nothing to do with the current conflict. After backjumping, the unit clause l_4 allows to assign true to l_4 to resolve the current conflict before further branching. Note the derived clause is inserted into *implicitConstraints* and will prevent cdclGCP from falling into the same conflict: as soon as three of these literals become false, the remaining literal will be assigned true by unit propagation.

4.2. Conflict-driven clause learning (CDCL)

A conflict is either an empty clause or an empty vertex. An empty vertex v_i can be also considered as an empty clause $v_{i1} \vee v_{i2} \vee \ldots \vee v_{ik}$ in which all literals are false (recall that v_{ic} is true iff vertex v_i is assigned color c). In what follows, we will represent a conflict as a set of falsified literals $\{l_1, l_2, \ldots, l_s\}$ for simplicity.

Let l_i $(1 \le i \le s)$ be the most recently falsified literal in the conflict. The analyzeConflict() procedure implements the first UIP (Unique Implication Point) learning schema [22], by repeatedly applying one of the following rules until the conflict contains only one literal falsified in the current recursive level.

`unitVertex` : if l_i is a positive literal v_c, and the reason of the falsification of l_i is $\{u \leftarrow c\}$, where u was a unit vertex (any c' such that $c' \neq c$ was removed from u, forcing $u \leftarrow c$), replace l_i with the set of falsified literals $\{u_{c'} | c' \neq c\}$,

`branchingVertex` : if l_i is a positive literal v_c, and the reason of the falsification of l_i is a branching vertex u assigned color c, replace l_i with $\overline{u}_c$,

`unitClause` : if the reason of the falsification of l_i is a unit clause cl, replace l_i with the set of all falsified literals in cl.

See Example 1 for an illustration of this procedure. Observe that CDCL in modern SAT solvers realizes the unitVertex rule by two resolution steps: Without loss of generality, let $i=1$, i.e., $l_1 = v_c$. CDCL in a SAT solver makes the first resolution step from $\{v_c, l_2, \ldots, l_s\}$ and $\{\overline{u}_c \vee \overline{v}_c\}$ to obtain $\{\overline{u}_c, l_2, \ldots, l_s\}$, and then the second resolution step between this new clause and $\{u_1 \vee u_2 \vee \ldots \vee u_k\}$ to obtain $\{u_{c'} | c' \neq c\} \cup \{l_2, \ldots, l_s\}$. The two steps are realized in analyzeConflict() in only one step using the unitVertex rule.

The analyzeConflict() procedure stops when a clause containing only one literal falsified in the current recursive level is derived. This literal will be the only non-assigned literal in the clause after backjumping to the second largest level in the clause, so that the clause becomes a unit clause that should be propagated after backjumping.

4.3. Learnt clause management

The learnt clauses allow to prune subtrees by preventing cdclGCP from visiting the same conflicts. Unfortunately, too many clauses consume much memory and can slow down the algorithm.

A common method used in modern SAT solvers to remedy this defect is to periodically delete half of the learnt clauses selected using some heuristic. For example, a heuristic called Literal Block Distance (LBD) was reported in [27] to measure the quality of learnt clauses. The clauses with the largest LBD are considered as less relevant in pruning subtrees and are deleted.

We have implemented in our cdclGCP algorithm several heuristics used in modern SAT solvers to manage the learnt clauses, including LBD. The results are not very good. It appears that clauses learnt in an algorithm for GCP have some special properties to be studied and taken into account in their management. Namely, special heuristics should be defined to measure the quality of these clauses. We plan to investigate these issues in the future. In this paper, we use the following simple strategy: delete the oldest half of clauses every N conflicts, which has the following interesting property and appears to be good enough when coloring most DIMACS graphs.

Proposition 1. *If cdclGCP deletes half clauses every N conflicts, then before the* k*th deletion, the maximum number of learnt clauses in the memory is* $(2^k - 1)/2^{k-1}N$.

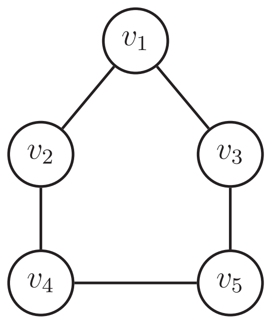

Fig. 2. A simple cycle graph.

Proof. We prove the proposition by induction on k. Recall that cdclGCP learns one new clause per conflict. When $k=1$, i.e., before the first deletion, the maximum number of learnt clauses in the memory is $N(=(2^1-1)/2^{1-1}N)$. Assume that before the kth deletion, the maximum number of learnt clauses is $(2^k-1)/2^{k-1}N$. The kth deletion removes $(2^k-1)/2^kN$ clauses and there remain $(2^k-1)/2^kN$ clauses. Upon the $(k+1)$th deletion, there are N additional clauses because there are N new conflicts. So, the maximum number of learnt clauses in the memory before the $(k+1)$th deletion is $(2^k-1)/2^kN+N=(2^{k+1}-1)/2^{k+1-1}N$. □

Proposition 1 means that the number of learnt clauses in the memory is always smaller than $2N$. N being a parameter of cdclGCP, we will provide in Section 5.2 an empirical justification for the precise value of N used in cdclGCP.

4.4. Symmetry breaking

In GCP, all colors are symmetric in the sense that a color assignment to vertices is a coloring solution if and only if any assignment obtained by permuting the colors is a solution. In cdclGCP as well as in backGCP, we consider that two colors are symmetric if they are available for all uncolored vertices. When a vertex is considered to be colored, all symmetric colors but one are removed from its available color set, so that the corresponding subproblems are pruned. As usual, before executing cdclGCP or backGCP, we search for a maximum clique in G using the highly efficient algorithm MaxCLQ for maxclique [23,24], breaking ties in favor of the maximum clique in which the sum of degrees of vertices is the maximum. We begin by coloring the vertices of this clique since they become unit after removing the symmetric colors. Then while there remain symmetric colors, we repeatedly select a vertex v using the dom+deg heuristic and remove all symmetric colors from v but one.

Example 2. Consider the graph in Fig. 2 that should be colored using 4 colors, i.e., the set C_{v_i} of available colors for vertex v_i $(1 \le i \le 5)$ is $\{1, 2, 3, 4\}$. We first find the maximum clique $\{v_1, v_2\}$. All colors are available to all vertices so that they are all symmetric. We remove the symmetric colors 2, 3, and 4 from C_{v_1} which becomes $\{1\}$. Then we remove colors 3 and 4 from C_{v_2} which becomes $\{2\}$, since colors 2, 3 and 4 are available to all remaining vertices after assigning color 1 to v_1 (color 1 being removed from v_2 and v_3). Now colors 3 and 4 are available to the remaining vertices v_3, v_4 and v_5. We select v_3 and remove the symmetric color 4 from C_{v_3}. Finally we call cdclGCP or backGCP for the subgraph induced by v_3, v_4 and v_5 with the following sets of available colors: $C_{v_3}=\{2,3\}$, $C_{v_4}=\{1,3,4\}$ and $C_{v_5}=\{1,2,3,4\}$.

5. Experimental results and analysis

We conducted several experimental investigations to evaluate our approach. In this section, we first describe the experiment settings. Then we empirically study the clause management strategies and the distribution and impact of the learnt clauses in the real coloring process. Finally we show the effectiveness of CDCL for GCP and compare our algorithm cdclGCP with other algorithms on standard benchmarks.

5.1. Experiment settings

We use random graphs and DIMACS benchmarks in our experiments. Each random graph $G(n,p)$ of n vertices is generated by including each of the $n(n-1)/2$ possible edges with probability p. DIMACS instances[1] are the standard benchmark for GCP, including crafted graphs with special structures such as *queen*, *myciel*, *Leighton*, *Insertion*, *FullIns* and *Latin_square*_10; register allocation graphs such as *musol*, *zeroin*, *inithx* and *fpsol*; random graphs (*DSJC*), geometric random graphs (*DSJR*), and so on.

In all the experiments, we use the decision version of backGCP and cdclGCP to solve the k-colorability problem, which are implemented in C in the same way (using the same architecture and the same data structures) and are compiled using GNU gcc. Although unit propagation in Algorithm 1 is not explicit, we note that the DSATUR or dom+deg branching heuristic in backGCP means that unit vertex is selected for branching in priority, implying unit propagation. So, the only difference between backGCP and cdclGCP is the CDCL in cdclGCP. The experiments are conducted on a personal computer with Intel Core 2 Duo CPU E7500@2.93 GHz, 4G RAM and Ubuntu Linux system. User times for the program *dfmax.c* on our computer are 0.04, 0.36, 2.28, 8.72 s for *r* 200.5, *r* 300.5, *r* 400.5 and *r* 500.5, respectively. The program *dfmax.c* is a solver that searches for a maximum clique in a graph, and is used to (roughly) compare the performance of different computers in the literature.

Before running backGCP or cdclGCP on a graph G, a maximum clique of G is computed using MaxCLQ to break symmetries. The search for the maximum clique is limited to 100 s. In most cases, MaxCLQ returns very quickly (i.e., in less than 0.1 s) the maximum clique.

[1] Available at http://mat.gsia.cmu.edu/COLOR04.

Table 1
Impact of *N*.

(*instance*,*k*)	*N*	*Runtime*	*Tree size*	*Rate*
(1-insertion_4,4)	100	96.43	1,820,259	0.437
	300	51.35	1,489,601	0.467
	500	50.89	1,347,790	0.478
	700	51.86	1,226,744	0.482
	900	60.56	1,226,710	0.486
	1000	56.53	1,152,462	0.486
	3000	115.11	1,017,568	0.493
	5000	207.56	1,135,502	0.494
	7000	344.57	1,146,565	0.490
(myciel6,6)	100	>7200		
	300	1871.04	14,224,637	0.399
	500	1235.36	13,100,245	0.415
	700	1008.04	12,458,397	0.425
	900	951.36	12,073,786	0.430
	1000	944.53	11,862,795	0.433
	3000	1371.47	11,063,368	0.455
	5000	1969.52	11,220,606	0.463
	7000	2886.22	11,307,041	0.468
(3-FullIns_4,6)	100	39.01	5,68,122	0.539
	300	13.81	2,78,859	0.573
	500	10.06	1,82,273	0.587
	700	9.66	1,56,565	0.589
	900	8.54	83,180	0.583
	1000	5.97	83,180	0.580
	3000	8.47	77,304	0.546
	5000	10.47	66,428	0.556
	7000	18.03	84,002	0.521
(abb313GPIA,9)	100	49.00	2,86,670	0.442
	300	26.63	1,23,169	0.409
	500	24.44	95,715	0.406
	700	24.47	84,098	0.423
	900	25.05	83,906	0.408
	1000	19.78	54,418	0.412
	3000	21.44	41,943	0.382
	5000	21.40	35,078	0.443
	7000	19.58	35,121	0.424

Exceptionally, MaxCLQ can fail to find a maximum clique (for a large dense graph such as DSJC1000.9) within 100 s. In this case, MaxCLQ returns the largest clique found within 100 s.

5.2. *Impact of parameter N in clause management*

As described in Section 4.3, since managing a huge number of clauses would slow down cdclGCP, we delete the oldest half of clauses every time we learn *N* new clauses. We conducted experiments to study the impact of the parameter *N* when solving several representative DIMACS instances, by varying *N* from 100 to 900 with step 200 and from 1000 to 7000 with step 2000.

Table 1 shows the results. For each *N*, we report the runtime and the search tree size of cdclGCP. The number *k* is equal to $\chi(G)-1$ for all graphs *G* except for *abb*313*GPIA* for which $k=\chi(G)$. Here *k* is chosen so that the *k*-colorability instance is the hardest to solve for the graph. The search tree size corresponds to the number of backtrackings or the number of conflicts detected. Recall that cdclGCP learns one clause for each conflict. So, the search tree size is equal to the total number of learnt clauses. A clause is effective in pruning the search space if it becomes unit at least one time during search. The *rate* column reports the proportion of the effective clauses to all clauses. The number of existing clauses in the memory is limited to $(2^{k+1})-1/2^kN$ by Proposition 1.

From Table 1, we observe that the parameter *N* does affect the performance of cdclGCP. When *N* is very small, there are few clauses in the memory and the search tree is large. When *N* becomes larger, the search tree generally becomes smaller. However, when $N>1000$, the additional clauses do not allow to significantly reduce the search trees, and cdclGCP is considerably slowed down because it has to spend much time to manage these clauses.

In the sequel, we fix $N=1000$ in cdclGCP for all instances.

5.3. *Distribution of learnt clauses and their impact*

In this subsection, we conducted experiments using several representative instances to compare the search trees of cdclGCP and backGCP and to study the distribution and the impact of learnt clauses. Recall that each conflict in cdclGCP and backGCP corresponds to a leaf of a search tree, and that each conflict in cdclGCP corresponds to a learnt clause. So the distribution of the conflicts in different levels in cdclGCP corresponds to the distribution of the learnt clauses in these levels.

Fig. 3 compares the average number of conflicts detected per level by cdclGCP and by backGCP when coloring random graphs $G(90,p)$. The graphs $G(90,p)$ have 90 vertices and *p* ranges from 0.4 to 0.8 with step 0.1. "$G(90,p)$, *k* |*backGCP* (*cdclGCP*)" means to color the graph $G(90,p)$ using *k* colors by backGCP (cdclGCP), where *k* is equal to $\chi(G)-1$ for each particular graph *G*. We use such *k* to make cdclGCP and backGCP construct

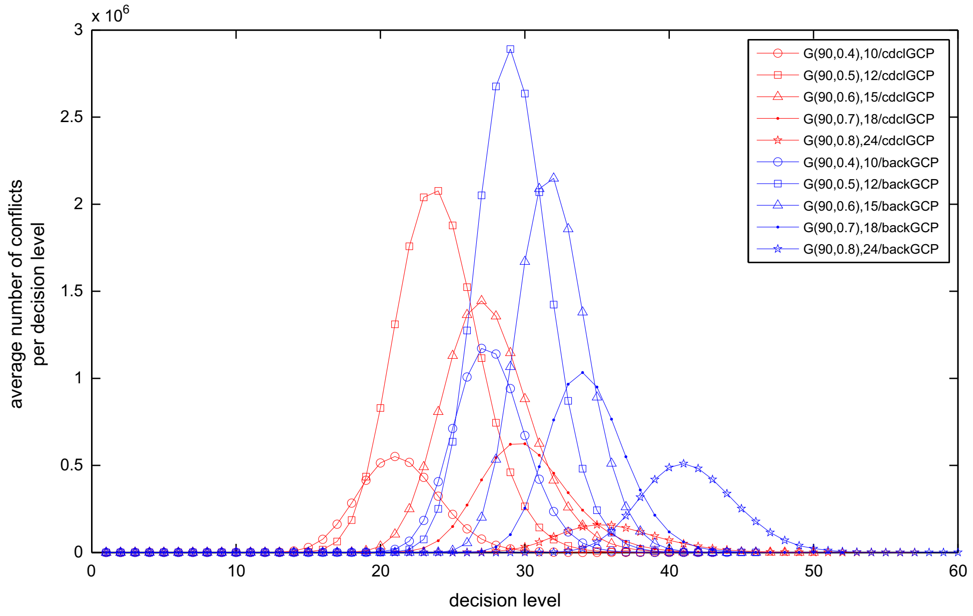

Fig. 3. Numbers of conflicts per level of cdclGCP and backGCP on random graphs ($G(90,p)$,k) with different p and k.

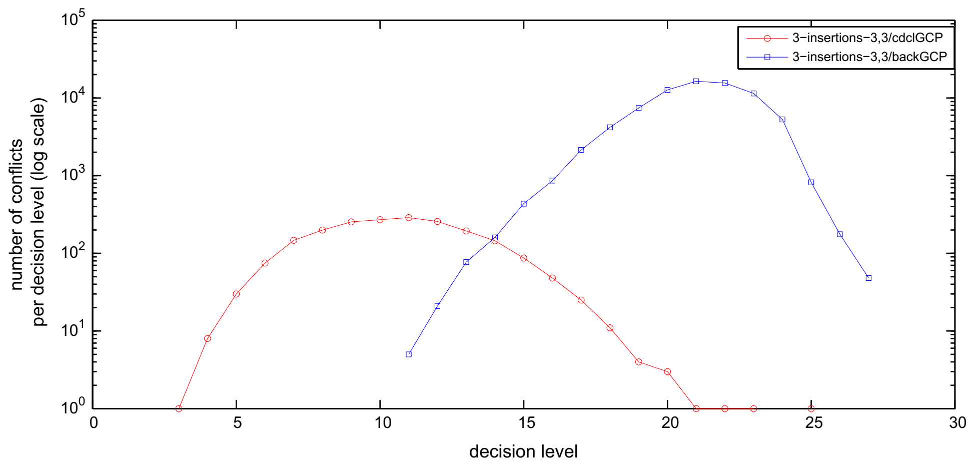

Fig. 4. Numbers of conflicts per level of cdclGCP and backGCP on 3-insertions_3.

a complete search tree for each instance, since an algorithm generally does not need a complete search tree for a colorable instance (search is stopped as soon as a solution is found). The number of conflicts at each decision level is averaged over 50 instances.

The results in Fig. 3 clearly show that the largest level (i.e., the level of the search tree containing the largest number of conflicts) of cdclGCP is significantly smaller than that of backGCP in two senses: the number of conflicts in the level is smaller and the value of the level is also smaller, showing the power of the learnt clauses in pruning search.

Figs. 4 and 5 compare the number of conflicts per level by cdclGCP and backGCP when coloring the DIMACS graphs 3-insertions_3 with 3 colors and myciel6 with 6 colors respectively. The number of colors is equal to $\chi(G)-1$ for each G. We can make similar observation in Figs. 4 and 5 as in Fig. 3, except that the difference between cdclGCP and backGCP is substantially larger, and the difference is so large that we have to use a log scale in the comparison, since otherwise the curves of cdclGCP would not be easily noticeable in the figures.

A final observation from Figs. 3, 4 and 5 is that most clauses are learnt at the middle levels of a search tree in cdclGCP.

5.4. Effectiveness of CDCL for GCP

In this subsection, we compare the real performance of cdclGCP with backGCP for coloring random and DIMACS graphs. Table 2 reports on the experimental results on random graphs. For each graph group $G(n,p)$, we compare the runtime and the search tree size of cdclGCP and backGCP to color each graph using k colors with k equal to $\chi-1$ (the Nonvalue), and to χ (the Yes value), respectively. To make the comparison clearer, we also give the reduction in search tree size of cdclGCP over backGCP, computed by *reduction*=(backGCP-cdclGCP)/backGCP. The runtimes and search tree sizes are averaged over 50 graphs for $n<90$ at each point. They are median of 51 graphs for $n \geq 90$: the 51 runtimes or search tree sizes of each point are sorted in increasing order, and the 26th value is the median. We use a cutoff time of 4 h. When $n<90$, backGCP and cdclGCP solved all instances within the cutoff time. When $n \geq 90$, the median can be computed at a point if at least 26 instances are solved within the cutoff time. If less than 26 instances are solved within 4 h, the entry is marked by ">4 h".

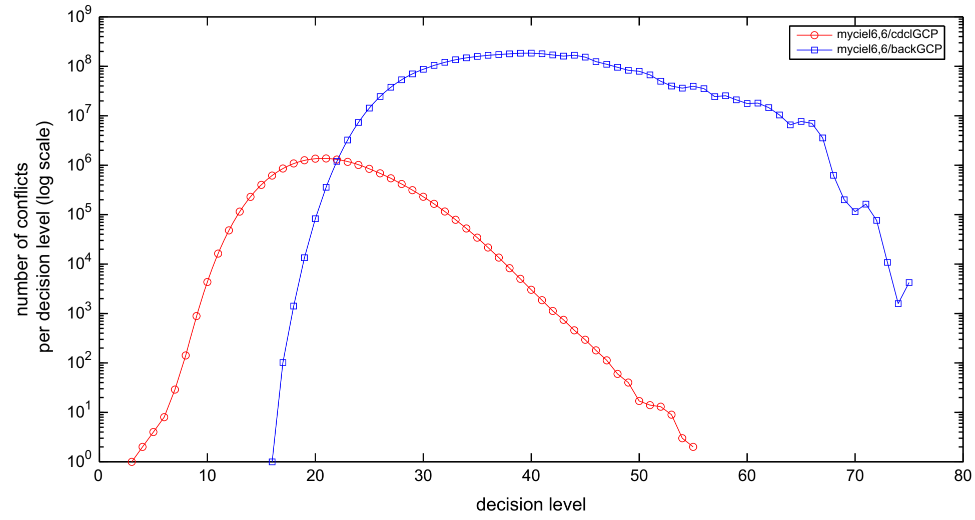

Fig. 5. Numbers of conflicts per level of cdclGCP and backGCP on myciel6.

Table 2
Comparison between cdclGCP and backGCP on random graph instances. Each row entry is averaged over 50 instances for graphs with fewer than 90 vertices, and is the median value of 51 instances for graphs with 90 vertices or more. The runtimes are in seconds. The reduction in the search tree size of cdclGCP w.r.t backGCP is computed as (backGCP-cdclGCP)/backGCP.

Name	*k*	Colorable	backGCP		cdclGCP		Reduction
			Tree size	Time	Tree size	Time	
G(70,0.1)	3.0	Non	1.4	0.0	0.0	0.0	1.00
	4.0	Yes	6.5	0.0	6.0	0.0	0.08
G(70,0.3)	6.8	Non	9898.5	0.06	9436.3	0.16	0.05
	7.8	Yes	9903.9	0.07	3714.1	0.06	0.62
G(70,0.5)	10.8	Non	578,408.6	4.7	538,155.4	14.60	0.07
	11.8	Yes	155,200.7	1.27	148,389.8	3.65	0.04
G(70,0.7)	16.2	Non	1729,093.7	16.27	1124,778.6	42.6	0.35
	17.2	Yes	953,216.2	8.77	899,515.3	33.74	0.06
G(70,0.9)	27.5	Non	9758.0	0.11	8826.1	0.34	0.11
	28.5	Yes	430,520.1	0.41	17,075.7	0.65	0.96
G(80,0.1)	3.3	Non	68.1	0.0	54.5	0.0	0.21
	4.3	Yes	41.0	0.0	20.5	0.0	0.50
G(80,0.3)	7.2	Non	520,190.1	4.1	258,250.7	6.88	0.50
	8.2	Yes	217,839.1	1.76	199,075.9	4.50	0.09
G(80,0.5)	11.9	Non	11,054,361.8	107.71	10,052,641.4	560.9	0.09
	12.9	Yes	930,621.2	8.98	670,427.1	49.7	0.30
G(80,0.7)	17.8	Non	10,738,928.6	121.74	9169,520.9	565.6	0.15
	18.8	Yes	6328,857.7	70.59	5244,035.9	303.46	0.17
G(80,0.9)	30.6	Non	1283,969.5	15.97	832,954.9	67.67	0.35
	31.6	Yes	636,522.2	6.70	374,071.8	22.26	0.41
G(90,0.1)	4.0	Non	272	0.0	163	0.0	0.40
	5.0	Yes	5	6.70	1	0.0	0.80
G(90,0.3)	8.0	Non	1052,376	8.0	933,650	43.32	0.11
	9.0	Yes	49,195	0.4	34,608	0.88	0.30
G(90,0.5)	12.6	Non	350,517,900	3314.85	–	> 4th	–
	13.6	Yes	178,067,376	1970.94	–	> 4th	–
G(90,0.7)	19.3	Non	224,872,693	2391.03	–	> 4th	–
	20.3	Yes	–	> 4th	–	> 4th	–
G(90,0.9)	33.2	Non	3,123,289	38.68	1,003,696	57.07	0.68
	34.2	Yes	1,119,597	11.82	250,723	12.56	0.78
G(95,0.1)	4.0	Non	162	0.0	79	0.0	0.51
	5.0	Yes	2	0.0	1	0.0	0.50
G(95,0.3)	8.0	Non	688,402	6.46	488,011	21.89	0.29
	9.0	Yes	6849,294	69.42	4065,282	423.73	0.41
G(95,0.5)	13.0	Non	264,878,003	3051.27	–	> 4th	–
	14.0	Yes	552,511,079	6711.75	–	> 4th	–
G(100,0.1)	4.0	Non	58	0.0	46	0.0	0.21
	5.0	Yes	1	0.0	1	0.0	0.00
G(100,0.3)	8.3	Non	435,242	4.04	263,691	10.19	0.40
	9.3	Yes	419,937,655	4411.32	–	> 4th	–
G(100,0.9)	35.7	Non	35,929,328	673.93	–	> 4th	–
	36.7	Yes	50,252,835	899.39	–	> 4th	–

Table 3
Comparison between cdclGCP and backGCP on DIMACS benchmarks. Runtime (in seconds) and search tree size for each *k*-colorability instance is reported with *k* equal to the Non and Yes values respectively. The time in bold face means the clearly best result. For backGCP, "-" means that the tree size is not available since the instance cannot be solved within the cutoff time.

Name	*k*	Colorable	backGCP		cdclGCP		Reduction
			Tree size	Time	Tree size	Time	
le450_5a	4	Non	0	0.0	0	0.03	0
	5	Yes	1310	0.07	581	0.07	0.55
le450_5b	4	Non	0	0.0	0	0.04	0
	5	Yes	356	0.04	267	0.05	0.25
le450_5c	4	Non	0	0.12	0	0.11	0
	5	Yes	10	0.12	10	0.12	0
le450_5d	4	Non	0	0.11	0	0.12	0
	5	Yes	27	0.11	28	0.12	−0.04
le450_15a	14	Non	0	0.08	0	0.08	0
	16	Yes	7	0.08	6	0.08	0.14
le450_15b	14	Non	0	0.08	0	0.08	0
	16	Yes	0	0.08	0	0.08	0
le450_15c	14	Non	0	0.34	0	0.34	0
	23	Yes	45	0.34	24	0.35	0.46
le450_15d	14	Non	0	0.34	0	0.34	0
	23	Yes	912	0.38	126	0.36	0.86
le450_25a	24	Non	0	0.08	0	0.08	0
	25	Yes	0	0.08	0	0.08	0
le450_25b	24	Non	0	0.08	0	0.08	0
	25	Yes	0	0.08	0	0.08	0
le450_25c	24	Non	0	0.37	0	0.37	0
	27	Yes	–	> 7200	246	**0.41**	–
le450_25d	24	Non	0	0.37	0	0.38	0
	27	Yes	8124	0.7	842	0.54	0.89
queen8_8	8	Non	122,528	**0.79**	88,390	3.46	0.28
	9	Yes	1934	**0.01**	1269	0.04	0.34
queen8_12	11	Non	0	0.0	0	0.0	0
	12	Yes	62	0.0	43	0.0	0.31
queen9_9	8	Non	0	0.0	0	0.0	0
	10	Yes	1,896,126	**14.56**	1,230,681	71.26	0.35
queen10_10	9	Non	0	0.0	0	0.0	0
	12	Yes	5666	**0.04**	2194	0.10	0.61
queen11_11	10	Non	0	0.0	0	0.0	0
	13	Yes	34,541	**0.3**	17,118	1.06	0.50
queen12_12	11	Non	0	0.01	0	0.01	0
	14	Yes	391,002	**4.17**	140,329	14.16	0.64
queen13_13	12	Non	0	0.02	0	0.02	0
	15	Yes	54,987,984	**71.21**	1,685,561	182.59	0.97
queen14_14	13	Non	0	0.04	0	0.04	0
	17	Yes	82,576	**1.06**	23,170	2.84	0.72
queen15_15	14	Non	0	0.06	0	0.07	0
	18	Yes	389	0.07	110	0.07	0.72
queen16_16	15	Non	0	0.1	0	0.1	0
	19	Yes	290,514	4.22	40,243	6.91	0.86
myciel6	6	Non	–	> 7200	11,862,795	**945.24**	–
	7	Yes	0	0.0	0	0.0	0
myciel7	5	Non	8,546,502	185.05	17,611	**1.34**	0.99
	8	Yes	0	0.0	0	0.01	0
1-insertions_4	4	Non	83,039,581	365.91	1,152,462	**56.30**	0.99
	5	Yes	0	0.0	0	0.0	0
1-insertions_5	4	Non	–	> 7200	21,374,521	**3535.32**	–
	6	Yes	0	0.0	0	0.01	0
1-insertions_6	3	Non	32	0.04	22	0.04	0.31
	7	Yes	0	0.05	0	0.05	0
2-insertions_4	3	Non	1453	0.01	274	0.01	0.81
	5	Yes	0	0.0	0	0.0	0
2-insertions_5	3	Non	879	0.05	143	**0.03**	0.84
	6	Yes	0	0.02	0	0.02	0
3-insertions_3	3	Non	430,897	0.84	1361	**0.02**	0.99
	4	Yes	0	0.0	0	0.0	0
3-insertions_4	3	Non	696,784	10.6	6951	**0.38**	0.99
	5	Yes	0	0.0	0	0.02	0
3-insertions_5	3	Non	902,031	85.09	1737	**0.43**	0.99
	6	Yes	0	0.13	0	**0.12**	0
4-insertions_3	3	Non	263,339,681	632.87	87,455	**3.01**	0.99
	4	Yes	0	0.0	0	0.0	0
4-insertions_4	3	Non	–	> 7200	301,891	**23.98**	–
	5	Yes	0	0.0	0	0.0	0
1-Fullins_4	4	Non	8	0.0	7	0.0	0.13
	5	Yes	0	0.0	0	0.0	0

Table 3 (*continued*)

Name	k	Colorable	backGCP		cdclGCP		Reduction
			Tree size	Time	Tree size	Time	
1-Fullins_5	5	Non	12,505	0.24	218	**0.04**	0.98
	6	Yes	0	0.01	0	0.01	0
2-Fullins_3	4	Non	1	0.0	1	0.0	0
	5	Yes	0	0.0	0	0.0	0
2-Fullins_4	5	Non	–	> 7200	740	**0.03**	–
	6	Yes	0	0.0	0	0.01	0
2-Fullins_5	6	Non	–	> 7200	606	**0.89**	–
	7	Yes	0	0.19	0	0.95	0
3-Fullins_3	5	Non	1	0.0	2	0.0	−1.0
	6	Yes	0	0.0	0	0.0	0
3-Fullins_4	6	Non	–	> 7200	83,180	**5.78**	–
	7	Yes	0	0.01	0	0.09	0
3-Fullins_5	7	Non	–	> 7200	1210	**10.84**	–
	8	Yes	0	1.46	0	1.46	0
4-Fullins_3	6	Non	1	0.0	3	0.0	−2.0
	7	Yes	0	0.0	0	0.0	0
4-Fullins_4	7	Non	26	0.05	26	0.06	0.0
	8	Yes	0	0.06	0	0.06	0
4-Fullins_5	8	Non	–	> 7200	1417	**97.35**	–
	9	Yes	0	8.0	0	8.3	0
5-Fullins_3	7	Non	1	0.0	4	0.0	−3.0
	8	Yes	0	0.0	0	0.0	0
5-Fullins_4	8	Non	–	> 7200	31	**1.56**	–
	9	Yes	0	0.17	0	1.58	0
ash331GPIA	3	Non	2	**0.02**	1	0.33	0.5
	4	Yes	7	**0.02**	7	0.32	0
ash608GPIA	3	Non	2	**0.07**	1	2.0	0.5
	4	Yes	715252	37.67	23	**2.32**	0.99
ash958GPIA	3	Non	2	**0.19**	1	7.81	0.5
	4	Yes	–	> 7200	52	**7.64**	–
abb313GPIA	8	Non	–	> 7200	76	**8.49**	–
	9	Yes	–	> 7200	54,418	**18.88**	–
qg.order30	29	Non	0	0.84	0	1.77	0
	30	Yes	3406	0.86	706	1.8	0.79
qg.order40	39	Non	0	4.76	0	9.21	0
	40	Yes	2457,688	**63.83**	124,184	69.22	0.95
qg.order60	59	Non	0	55.33	0	113.75	0
	61	Yes	11,924,880	**375.36**	576,395	1191.85	0.95
mug88_1	3	Non	1.431,467,008	1824.44	122	**0.0**	0.99
	4	Yes	0	0.0	0	0.0	0
mug88_25	3	Non	79,943,296	120.26	88	**0.0**	0.99
	4	Yes	0	0.0	0	0.0	0
mug100_1	3	Non	2,369,826,816	3516.49	139	**0.0**	0.99
	4	Yes	0	0.0	0	0.0	0
mug100_25	3	Non	3,629,146,112	5517.99	142	**0.0**	0.99
	4	Yes	0	0.0	0	0.0	0
flat300_26_0	14	Non	1,220,747	37.77	858,321	62.83	0.30
	40	Yes	9077	0.86	10	0.59	0.99
flat300_28_0	14	Non	420,183	13.37	414,025	30.02	0.01
	39	Yes	252,286	8.74	85,350	13.44	0.66
flat1000_50_0	17	Non	4,726,668	580.12	5,840,469	1708.45	−0.24
	112	Yes	243	73.77	5095	79.42	−19.96
flat1000_60_0	17	Non	32,724,663	3507.31	1,908,408	**498.65**	0.94
	110	Yes	95,933	87.54	1,789,739	1949.47	−17.65
wap03a	39	Non	1	99.13	0	101.51	1.0
	54	Yes	–	> 7200	0	107.57	–
wap04a	39	Non	34	104.24	26	109.34	0.23
	46	Yes	0	105.11	0	110.18	–
wap05a	49	Non	–	> 7200	0	2.27	–
	50	Yes	0	2.22	0	2028	–
wap06a	39	Non	462	2.29	174	2.36	0.62
	44	Yes	0	2.26	0	2.34	–
wap07a	39	Non	–	> 7200	0	13.07	–
	44	Yes	–	> 7200	25,172	35.08	–

The results in Table 2 clearly show that clause learning allows cdclGCP to effectively prune the search space for random graphs. However, this pruning is not enough to compensate the overhead of the clause learning and the management of the learnt clauses, so that cdclGCP is slower than backGCP for random graphs.

Table 3 shows the experimental results on DIMACS instances, excluding the random instances and the very easy instances (solved by both cdclGCP and backGCP without backtracking) for clarity. For each graph, we report the best Nonvalue (the largest number of colors with which the graph is shown non-colorable), and the best Yes value (the smallest number of colors with which a coloring solution is found), obtained by cdclGCP or backGCP within the cutoff time (7200 s), and the runtime and search tree size (i.e., the number of backtrackings or number of conflicts) of the two algorithms to solve the k-colorability instance (k equal to the best Non or Yes values). When an algorithm fails to solve the instance within the cutoff time, the corresponding runtime entry is ">7200". We also give the reduction in the tree size of cdclGCP over backGCP, computed by *reduction*=(backGCP−cdclGCP)/backGCP.

Table 3 shows that clause learning in cdclGCP allows an effective reduction of the search tree size compared with backGCP for DIMACS graphs, and contrarily to random graphs, this pruning of the search space allows cdclGCP to be significantly more efficient than backGCP. In fact, cdclGCP solves 9 instances more than backGCP within the cutoff time (2-Fullins_4, 2-Fullins_5, 3-Fullins_4, 3-Fullins_5, 4-Fullins_5, 5-Fullins_4, *ash958GPIA*, *myciel6*, *abb313GPIA*). And there are three other instances (le450_25c, 1-insertion_5, 4-insertion_4) for which cdclGCP finds better *Non* or *Yes* values within the cutoff time. There is no instance for which backGCP can solve or find a *Non* or *Yes* value within the cutoff time but cdclGCP cannot. Especially, cdclGCP proves the chromatic number of the graph 4-Fullins_5, which is 9, for the first time in the literature to our knowledge.

We say a graph is structured, if its edges satisfy some special constraints. The structured graphs are to be compared with random graphs in which edges are generated independently with a probability and therefore do not have any intended relationship among them. Looking at Tables 2 and 3, the search tree size reduction of cdclGCP over backGCP is comparable for random and DIMACS graphs. However, DIMACS graphs are often structured, and the learnt clauses in cdclGCP are generally shorter for structured graphs than for random graphs (see below). Learning and managing longer clauses are very time-consuming, explaining why cdclGCP is slower than backGCP for random graphs, but significantly faster than backGCP for structured graphs, although the search tree reduction is similar in two cases.

When coloring a structured graph, the constraints between edges allow cdclGCP to derive short clauses. For example, a typical structured DIMACS graph family is *myciel* (*Insertion* and *FullIns* are two other DIMACS graph families generalizing the *myciel* graphs). A graph in the *myciel* family is triangle free. In other words, let v_1, v_2 and v_3 be three vertices, if edge (v_1, v_2) and edge (v_2, v_3) are in the graph, then edge (v_1, v_3) is not in the graph. Consequently, the graph contains many chains formed by 4 vertices as illustrated in Fig. 1. When 4 vertices in a chain have to be colored using 2 colors, Example 1 shows how cdclGCP derives 2 binary clauses saying v_1 and v_4 cannot have the same color. These binary clauses are powerful in pruning the search space, since once v_1 or v_4 is assigned a color, the color is immediately removed from the other vertex by the two clauses via unit clause propagation.

5.5. Comparison with state-of-the-art algorithms

In this subsection, we compare our approach with the four following exact algorithms for GCP among the best in the literature:

- `B&C`: A branch-and-cut algorithm proposed by Méndez-díaz and Zabala [10] in 2006. B&C is based on a new integer programming formulation of GCP.
- `MMT-BP`: A recent Branch-and-Price algorithm embedding an efficient metaheuristic procedure MMT described in [11]. It was proposed by Malaguti, Monaci and Toth in 2010 and is based on the Set Covering formulation of GCP.
- `PASS`: A very recent exact algorithm proposed by Segundo [12] in 2012 based on the well-known DSATUR heuristic. It uses a new tie-breaking strategy different from SEWELL [19] in that it restricts the application of the strategy to a particular set of vertices.
- `VCS`: An algorithm proposed in [34,35]. A *k-Vertex-Critical-Subgraph* (*k-VCS*) H of a graph G is a subgraph of G, such that the chromatic number of any subgraph of H obtained by removing a vertex from H is smaller than k. Given G, VCS determines a lower bound k of $\chi(G)$ by seeking a *k-VCS* of G and then proving that k is the chromatic number of the *k-VCS*. If k is equal to the best known upper bound of $\chi(G)$, then $k=\chi(G)$.

Given a graph G, B&C, MMT-BP and PASS find $\chi(G)$ or the tightest possible LB and UB such that LB $\le \chi(G) \le$ UB. In PASS, LB is always the size of a clique found within 5 s, while in B&C and MMT-BP, LB can be improved by solving the continuous relaxation problem at each step of the branching scheme. Our algorithms backGCP and cdclGCP solve the k-colorability problem. We adapt them as showed in Algorithm 4 to find the chromatic number $\chi(G)$ or the tightest UB of $\chi(G)$, so that they are comparable with B&C, MMT-BP and PASS.

Algorithm 4. algoGCP(G).

```
Input: A graph $G=(V, E)$
Output: The chromatic number $\chi(G)$
begin
    Find a maximum clique using MaxCLQ with a cutoff time 100 s;
    break symmetries using the clique;
    $k:=|V|-1$;
    while true do
        call backGCP or cdclGCP for coloring G with k colors;
        if A coloring solution is found then
            $k:=k-1$;
        else
            $\chi(G):=k+1$;
            return $\chi(G)$;
end
```

Given a graph G, Algorithm 4 first breaks symmetries using the clique found by MaxCLQ within 100 s and repeatedly solves the k-colorability problem by decrementing k from $|V|-1$ colors until G is not colorable.

5.5.1. *Comparison result on random graphs*

Table 4 compares the performance of backGCP and cdclGCP with PASS on random graphs $G(n,p)$ with $n \in \{60, 70, 75, 80\}$ and each p ranging from 0.1 to 0.9 with step 0.1, except for $n=80$ for which p steps only from 0.1 to 0.3. These are graphs used for testing PASS in [12]. The results of B&C and MMT-BP are not displayed in the table since PASS clearly outperforms them according to [12]. All entries are averaged over 50 instances for PASS and backGCP. The few instances that cdclGCP fails to solve(#fail) at some point are excluded when computing the average time of cdclGCP. Column χ gives the average chromatic number of these graphs.

The PASS runtimes were taken directly from [12] and were on an I7-CPU 920@2.67 GHz with 6 GB RAM(Windows O.S). The user times of *dfmax* on this machine are 0.031, 0.234, 1.419 and 5.336 s for r 200.5, r 300.5, r 400.5 and r 500.5, respectively. The cutoff time for PASS is 1200 s.

The displayed runtime of backGCP and cdclGCP for a graph G is the runtime of Algorithm 4, including the runtime of MaxCLQ and the runtimes of backGCP or cdclGCP for all k from $|V|-1$ to $\chi(G)-1$, which is different from Tables 2 and 3 where the displayed runtimes of backGCP and cdclGCP are for only one number of colors.

Since backGCP and cdclGCP run on a slower computer than PASS, their runtimes are calibrated according to the method usually used in the literature, by comparing the user times of *dfmax* for the two largest graphs r 400.5 and r 500.5 on our machine and on the machine of PASS. As *dfmax* spends 2.28 and 8.72 for the two graphs respectively, our computer is roughly 1.62 times slower ($=(2.28/1.419+8.72/5.336)/2$). So, we divide the runtimes of backGCP and cdclGCP by 1.62 to make a rough but fair comparison with PASS. In addition, since PASS uses a cutoff time of 1200 s, backGCP and cdclGCP use a cutoff time of 1800 s on our machine to favor PASS (the calibrated cutoff time would be $1200 \times 1.62 = 1944$ s).

According to Table 4, PASS and backGCP have comparable performance for random graphs. PASS is faster than backGCP for dense graphs ($p \geq 0.6$), and backGCP is faster than PASS for other random graphs. We conjecture that the clause learning CDCL approach integrated in PASS would have the same impact.

Table 5 shows the performance of backGCP and cdclGCP for large random graphs averaged over 50 graphs at each point. Since some of these graphs are hard, and Algorithm 4 just can give an upper bound of χ for them within a cutoff time of 4 h, we also use Algorithm 5 below to compute a lower bound LB of χ within 4 h. At each point, if all the 50 instances are solved (i.e., the chromatic number of all the 50 graphs are found) using Algorithm 4 within the cutoff time, then LB=UB, and the average runtime of Algorithm 4 is displayed; otherwise, Algorithm 5 is called to compute the LB for the graphs for which Algorithm 4 fails to find the chromatic number. In the latter case, the best LB and the best UB obtained within 4 h are averaged over 50 graphs and the runtime is marked ">4 h". We are not aware of the performance of other exact algorithms in the literature on random graphs of these sizes, especially when $0.5 \leq p \leq 0.8$.

Table 4
Comparison of backGCP, cdclGCP with PASS on random graphs. Time in seconds. Entries in bold face mean the clearly best results.

Name	χ	PASS		backGCP		cdclGCP	
		Time	*#fail*	*Time*	*#fail*	*Time*	*#fail*
G(60,0.1)	4.0	0.0	0	0.00	0	0.00	0
G(60,0.2)	5.5	0.0	0	0.00	0	0.00	0
G(60,0.3)	7.0	0.0	0	0.00	0	0.00	0
G(60,0.4)	8.9	0.1	0	**0.03**	0	0.06	0
G(60,0.5)	10.7	0.2	0	0.15	0	0.28	0
G(60,0.6)	12.9	0.3	0	0.21	0	0.50	0
G(60,0.7)	15.6	0.2	0	0.25	0	0.50	0
G(60,0.8)	19.1	0.1	0	0.11	0	0.23	0
G(60,0.9)	25.7	0.0	0	0.00	0	0.01	0
G(70,0.1)	4.0	0.0	0	0.00	0	0.00	0
G(70,0.2)	6.0	0.0	0	0.00	0	0.00	0
G(70,0.3)	7.8	0.2	0	**0.08**	0	0.14	0
G(70,0.4)	9.4	1.7	0	**0.90**	0	2.61	0
G(70,0.5)	11.8	5.5	0	**3.69**	0	11.33	0
G(70,0.6)	14.1	8.4	0	8.43	0	18.39	0
G(70,0.7)	17.2	14.3	0	15.87	0	48.10	0
G(70,0.8)	21.4	**2.6**	0	3.34	0	9.86	0
G(70,0.9)	28.5	**0.1**	0	0.34	0	0.63	0
G(75,0.1)	4.0	0.0	0	0.00	0	0.00	0
G(75,0.2)	6.0	0.0	0	0.00	0	0.01	0
G(75,0.3)	8.0	0.2	0	0.13	0	0.15	0
G(75,0.4)	10.0	6.8	0	**3.74**	0	14.24	0
G(75,0.5)	12.1	35.4	0	**25.02**	0	227.27	2
G(75,0.6)	14.9	72.2	0	60.39	0	395.53	6
G(75,0.7)	18.0	76.6	0	79.70	0	262.81	6
G(75,0.8)	22.4	**17.2**	0	24.66	0	64.60	0
G(75,0.9)	29.9	**1.6**	0	2.43	0	9.75	0
G(80,0.1)	4.3	0.0	0	0.00	0	0.00	0
G(80,0.2)	6.3	0.1	0	**0.04**	0	0.10	0
G(80,0.3)	8.2	7.3	0	**3.61**	0	7.03	0

Algorithm 5. algoGCPforLB(G).

```
Input: A graph $G=(V, E)$
Output: The chromatic number $\chi(G)$
begin
    LB := the cardinality of the clique found using MaxCLQ within 100 s;
    break symmetries using the clique;
    while true do
        call backGCP or cdclGCP for coloring $G$ with LB colors;
        if A coloring solution is found then
            $\chi(G)$:=LB;
            return $\chi(G)$;
        else
            $LB:=LB+1$;
end
```

Table 5
Comparison of backGCP and cdclGCP on large random graphs.

Name	backGCP			cdclGCP		
	LB	*UB*	*Time*	*LB*	*UB*	*Time*
G(90,0.1)	5.0	5.0	0.00	5.0	5.0	0.0
G(90,0.2)	7.0	7.0	0.24	7.0	7.0	0.75
G(90,0.3)	9.0	9.0	8.43	9.0	9.0	44.34
G(90,0.4)	11.0	11.0	647.80	11.0	11.12	> 4 h
G(90,0.5)	13.5	13.9	> 4 h	13.0	14.0	> 4 h
G(90,0.6)	16.0	17.0	> 4 h	15.9	17.3	> 4 h
G(90,0.7)	20.0	20.9	> 4 h	19.0	21.5	> 4 h
G(90,0.8)	25.2	25.6	> 4 h	24.7	26.2	> 4 h
G(90,0.9)	34.2	34.2	50.50	34.2	34.3	> 4 h
G(95,0.1)	5.0	5.0	0.00	5.0	5.0	0.00
G(95,0.2)	7.0	7.0	0.24	7.0	7.0	0.58
G(95,0.3)	9.0	9.0	75.88	9.0	9.0	445.61
G(95,0.4)	11.5	11.9	> 4 h	11.0	12.0	> 4 h
G(95,0.5)	13.9	14.7	> 4 h	13.3	14.9	> 4 h
G(100,0.1)	5.0	5.0	0.00	5.0	5.0	0.00
G(100,0.2)	7.0	7.0	0.90	7.0	7.0	2.62
G(100,0.3)	9.3	9.4	> 4 h	9.0	9.8	> 4 h
G(110,0.1)	5	5	0.00	5	5	0.00
G(120,0.1)	5	5	0.14	5	5	0.21
G(130,0.1)	5.6	5.6	25.51	5.6	5.6	33.49
G(140,0.1)	6	6	1.74	6	6	4.08
G(150,0.1)	6	6	1.46	6	6	2.04
G(160,0.1)	6	6	12.6	6	6	11.92

Table 6
Comparison of backGCP, cdclGCP with VCS on random graphs.

Instance	VCS					backGCP		cdclGCP	
	k	*btk*	$\|V\|$	$\|E\|$	*btk'*	*k*	*btk*	*k*	*btk*
G(100,0.1)	5	113	47	178	128	5	78	5	72
G(110,0.1)	5	91	37	131	62	5	29	5	28
G(120,0.1)	5	2127	28	93	52	5	13	5	12
G(130,0.1)	5.5	5,54,302	70	389	477,382	5.5	863,720	5.5	699,196
G(140,0.1)	6	735,848	98	605	1,933,865	6	153,431	6	128,585
G(150,0.1)	6	303,302	90	556	446,554	6	63,526	6	52,604
G(160,0.1)	6	149,318	84	511	202,652	6	165,911	6	141,343
G(170,0.1)	6	39,328,744	79	471	113,852	6	14,188	6	10,844
G(180,0.1)	7	(45,084 s)	138	1151	(41,521 s)	7	(3522 s)	7	(42,422 s)
G(190,0.1)	7	(49,070 s)	148	1301	(40,927 s)	7	(3707 s)	7	(38,642 s)
G(200,0.1)	7	(48,259 s)	142	1239	(50,359 s)	7	(3798 s)	7	(38,941 s)

Table 7
Comparison between cdclGCP and other three exact algorithms on DIMACS benchmarks. The entry in bold face means the clearly best result. Entry "?" means that the chromatic number of graph instance is unknown so far. Entry "–" means that the corresponding value is not available from the original literature.

Instance	χ	B&C			MMT-BP			PASS			cdclGCP		
		LB	*UB*	*Time*	*LB*	*UB*	*Time*	*LB*	*UB*	*Time*	*LB*	*UB*	*Time*
le450_5a	5	5	9	*Out*	5	5	0.3	–	–	–	5	5	0.08
le450_5b	5	5	9	*Out*	5	5	0.2	–	–	–	5	5	0.07
le450_5c	5	5	5	*Init*	5	5	0.1	5	5	0.0	5	5	0.19
le450_5d	5	5	10	*Out*	5	5	0.2	5	5	98.1	5	5	0.2
le450_15a	15	15	17	*Out*	**15**	**15**	**0.4**	15	16	*Out*	15	16	*Out*
le450_15b	15	15	17	*Out*	**15**	**15**	**0.2**	15	16	*Out*	15	16	*Out*
le450_15c	15	15	24	*Out*	**15**	**15**	**3.1**	15	22	*Out*	15	23	*Out*
le450_15d	15	15	23	*Out*	**15**	**15**	**3.8**	15	23	*Out*	15	23	*Out*
le450_25a	25	25	25	*Init*	25	25	0.1	25	25	0.0	25	25	0.12
le450_25b	25	25	25	*Init*	25	25	0.1	25	25	0.0	25	25	0.13
le450_25c	25	25	28	*Out*	**25**	**25**	**1356.6**	–	–	–	25	27	*Out*
le450_25d	25	25	28	*Out*	**25**	**25**	**66.6**	–	–	–	25	27	*Out*
queen8_8	9	9	9	3.0	9	9	3.6	9	9	3.0	9	9	2.17
queen8_12	12	12	12	*Init*	12	12	0.2	12	12	0.0	12	12	0.0
queen9_9	10	9	11	*Out*	10	10	36.6	10	10	466.0	9	10	*Out*
queen10_10	11	10	12	*Out*	**11**	**11**	**686.9**	10	12	*Out*	10	12	*Out*
queen11_11	11	11	14	*Out*	**11**	**11**	**1865.7**	11	13	*Out*	11	13	*Out*
queen12_12	12	12	15	*Out*	12	**13**	*Out*	12	14	*Out*	12	14	*Out*
queen13_13	13	13	16	*Out*	13	**14**	*Out*	13	15	*Out*	13	15	*Out*
queen14_14	14	14	17	*Out*	14	**15**	*Out*	14	16	*Out*	14	17	*Out*
queen15_15	15	15	18	*Out*	15	**16**	*Out*	15	18	*Out*	15	18	*Out*
queen16_16	16	16	20	*Out*	16	**17**	*Out*	16	19	*Out*	16	19	*Out*
myciel6	7	5	7	*Out*	4	7	*Out*	2	7	*Out*	**7**	**7**	**585.77**
myciel7	8	5	8	*Out*	5	8	*Out*	2	8	*Out*	**6**	8	*Out*
1- insertions_4	5	5	5	2.0	3	5	*Out*	5	5	240.0	5	5	34.81
1-insertions_5	?	4	6	*Out*	3	6	*Out*	2	6	*Out*	4	6	*Out*
1-insertions_6	?	4	7	*Out*	3	7	*Out*	2	7	*Out*	4	7	*Out*
2-insertions_4	4	4	5	*Out*	3	5	*Out*	2	5	*Out*	4	5	*Out*
2-insertions_5	?	3	6	*Out*	2	6	*Out*	2	6	*Out*	**4**	6	*Out*
3-insertions_3	4	4	4	1.0	3	4	*Out*	4	4	0.6	4	4	0.02
3-insertions_4	?	3	5	*Out*	3	5	*Out*	2	5	*Out*	**4**	5	*Out*
3-insertions_5	?	3	6	*Out*	2	6	*Out*	2	6	*Out*	**4**	6	*Out*
4-insertions_3	4	3	4	*Out*	3	4	*Out*	4	4	351.0	**4**	**4**	**1.85**
4-insertions_4	?	3	5	*Out*	2	5	*Out*	2	5	*Out*	**4**	5	*Out*
1-Fullins_4	5	5	5	0.1	4	5	*Out*	5	5	0.2	5	5	0.0
1-Fullins_5	6	4	6	*Out*	4	6	*Out*	3	6	*Out*	**6**	**6**	**0.04**
2-Fullins_3	5	5	5	0.1	5	5	2.9	5	5	0.0	5	5	0.0
2-Fullins_4	6	5	6	*Out*	5	6	*Out*	4	6	*Out*	**6**	**6**	**0.03**
2-Fullins_5	7	5	7	*Out*	5	7	*Out*	4	7	*Out*	**7**	**7**	**1.14**
3-Fullins_3	6	6	6	0.1	6	6	2.9	5	6	*Out*	6	6	0.0
3-Fullins_4	7	6	7	*Out*	6	7	*Out*	5	7	*Out*	**7**	**7**	**3.68**
3-Fullins_5	8	6	8	*Out*	5	8	*Out*	5	8	*Out*	**8**	**8**	**8.27**
4-Fullins_3	7	7	7	3.0	7	7	3.4	6	7	*Out*	7	7	0.0
4-Fullins_4	8	7	8	*Out*	7	8	*Out*	6	8	*Out*	**8**	**8**	**0.11**
4-Fullins_5	9	6	9	*Out*	6	9	*Out*	6	9	*Out*	**9**	**9**	**67.71**
5-Fullins_3	8	8	8	2.0	8	8	4.6	7	8	*Out*	8	8	0.0
5-Fullins_4	9	8	9	*Out*	8	9	*Out*	7	9	*Out*	**9**	**9**	**2.43**
ash331GPIA	4	4	4	51.0	4	4	45.9	4	4	0.0	4	4	0.51
ash608GPIA	4	4	4	692.0	3	4	*Out*	4	4	0.1	4	4	2.72
ash958GPIA	4	4	5	*Out*	3	4	*Out*	4	4	0.4	4	4	10.5
abb313GPIA	9	8	10	*Out*	7	9	*Out*	8	10	*Out*	**9**	**9**	**18.10**
will199GPIA	7	–	–	–	7	7	80.7	6	7	*Out*	7	7	0.56
fpsol2.i.1	65	65	65	0.6	65	65	10.6	65	65	0.0	65	65	0.46
fpsol2.i.2	30	30	30	1.2	30	30	11.2	30	30	0.0	30	30	0.27

fpsol2.i.3	30	30	30	1.1	30	30	10.0	30	30	0.0	30	30	0.29
inithx.i.1	54	54	54	*Init*	54	54	21.0	54	54	0.0	54	54	1.54
inithx.i.2	31	31	31	*Init*	31	31	9.2	31	31	0.0	31	31	0.96
inithx.i.3	31	31	31	*Init*	31	31	9.9	31	31	0.0	31	31	0.94
miles250	8	8	8	*Init*	8	8	5.0	8	8	0.0	8	8	0.0
miles500	20	20	20	*Init*	20	20	3.7	20	20	0.0	20	20	0.0
miles750	31	31	31	*Init*	31	31	0.2	31	31	0.0	31	31	0.0
miles1000	42	42	42	0.2	42	42	0.2	42	42	0.0	42	42	0.04
miles1500	73	73	73	0.1	73	73	0.1	73	73	0.0	73	73	0.11
zeroin.i.1	49	49	49	*Init*	49	49	4.4	49	49	0.0	49	49	0.04
zeroin.i.2	30	30	30	*Init*	30	30	4.5	30	30	0.0	30	30	0.03
zeroin.i.3	30	30	30	*Init*	30	30	3.6	30	30	0.0	30	30	0.03
qg.order30	30	30	30	*Init*	30	30	0.2	30	30	0.0	30	30	2.36
qg.order40	40	40	42	*Out*	40	40	2.9	40	40	0.2	40	40	49.51
qg.order60	60	60	63	*Out*	**60**	**60**	**3.8**	60	61	*Out*	60	61	*Out*
anna	11	11	11	*Init*	11	11	3.6	11	11	0.0	11	11	0.0
huck	11	11	11	*Init*	11	11	0.2	11	11	0.0	11	11	0.0
jean	10	10	10	*Init*	10	10	0.2	10	10	0.0	10	10	0.0
david	11	11	11	*Init*	11	11	0.2	11	11	0.0	11	11	0.0
mug88_1	4	4	4	11.0	4	4	9.6	4	4	324.0	4	4	**0.0**
mug88_25	4	4	4	184.0	4	4	10.6	4	4	191.0	4	4	**0.0**
mug100_1	4	4	4	60.0	4	4	14.4	3	4	*Out*	4	4	**0.0**
mug100_25	4	4	4	60.0	4	4	12.0	3	4	*Out*	4	4	**0.0**
mulsol.i.1	49	49	49	*Init*	49	49	0.2	49	49	0.0	49	49	0.04
mulsol.i.2	31	31	31	*Init*	31	31	4.7	31	31	0.0	31	31	0.03
mulsol.i.3	31	31	31	*Init*	31	31	0.2	31	31	0.0	31	31	0.03
mulsol.i.4	31	31	31	*Init*	31	31	0.2	31	31	0.0	31	31	0.03
mulsol.i.5	31	31	31	*Init*	31	31	6.0	31	31	0.0	31	31	0.03
school1	14	14	14	*Init*	14	14	0.4	14	14	0.1	14	14	0.83
school1_nsh	14	14	14	*Init*	14	14	17.0	14	14	0.4	14	14	0.51
DSJC125.1	5	5	5	0.9	5	5	142.0	5	5	0.0	5	5	0.0
DSJC125.5	17	13	20	*Out*	**16**	**17**	*Out*	10	19	*Out*	14	19	*Out*
DSJC125.9	44	42	47	*Out*	**43**	**44**	*Out*	34	46	*Out*	39	48	*Out*
DSJC250.1	?	5	9	*Out*	5	**8**	*Out*	4	9	*Out*	**6**	9	*Out*
DSJC250.5	?	13	36	*Out*	15	**28**	*Out*	12	34	*Out*	**16**	35	*Out*
DSJC250.9	?	48	88	*Out*	**71**	**72**	*Out*	39	82	*Out*	48	86	*Out*
DSJC500.1	?	5	15	*Out*	5	**12**	*Out*	5	14	*Out*	**7**	15	*Out*
DSJC500.5	?	13	63	*Out*	16	**48**	*Out*	13	62	*Out*	**17**	61	*Out*
DSJC1000.1	?	6	26	*Out*	5	**20**	*Out*	6	25	*Out*	**7**	25	*Out*
DSJC1000.5	?	15	116	*Out*	13	**92**	*Out*	14	110	*Out*	**18**	115	*Out*
DSJC1000.9	?	**65**	301	*Out*	51	**226**	*Out*	57	300	*Out*	62	298	*Out*
DSJR500.1	12	12	12	*Init*	12	12	35.3	12	12	0.0	12	12	0.52
DSJR500.1c	85	78	88	*Out*	**85**	**85**	**288.5**	80	85	*Out*	84	85	*Out*
DSJR500.5	122	119	130	*Out*	**122**	**122**	**342.2**	120	130	*Out*	122	129	*Out*
latin_sq._10	?	90	129	*Out*	90	**108**	*Out*	90	130	*Out*	90	127	*Out*
games120	9	9	9	*Init*	9	9	0.2	9	9	0.0	9	9	0.0
wap01a	?	41	46	*Out*	40	**43**	*Out*	41	46	*Out*	41	46	*Out*
wap02a	?	40	45	*Out*	40	**42**	*Out*	40	46	*Out*	40	46	*Out*
wap03a	?	40	56	*Out*	40	**47**	*Out*	–	–	–	40	54	*Out*
wap04a	?	40	50	*Out*	40	**44**	*Out*	–	–	–	40	46	*Out*
wap05a	50	50	51	*Out*	50	50	293.2	50	50	0.0	50	50	4.96
wap06a	40	40	44	*Out*	**40**	**40**	**175.0**	40	47	*Out*	40	44	*Out*
wap07a	?	40	46	*Out*	40	**42**	*Out*	40	44	*Out*	40	44	*Out*
wap08a	?	40	47	*Out*	40	**42**	*Out*	40	45	*Out*	40	44	*Out*

Tables 2, 4 and 5 show that clause learning probably is not suitable for random graphs, which was expected, since SAT solvers based on clause learning are very inefficient for random SAT instances, as is shown in successive SAT competitions.[2] In fact, we are not aware of any exact solving approach highly efficient for large dense random graphs (of more than 100 vertices).

Table 6 compares backGCP and cdclGCP with VCS on random graphs for each pair $(n, 0.1)$ with $n \in \{100, 110, 120, \dots, 200\}$. The five columns for VCS give number k, the number of backtracks (*btk*) VCS needs to prove that k is the chromatic number of the corresponding graph, the number of vertices and the number of edges of the *k-VCS* detected by VCS, and the number of backtracks (*btk*') required to determine the chromatic number of the *k-VCS*. All values of VCS are directly averaged from the results for 4 random graphs in [34] at each point. The values enclosed in parentheses indicate that the chromatic number cannot be found within 250,000,000 backtracks and represent the runtime in seconds to exceed the 250,000,000 backtracks. The k and backtracks (*btk*) in other columns for backGCP and cdclGCP have the same signification. In particular, backGCP and cdclGCP are also limited by 250,000,000 backtracks. Note that the results of backGCP and cdclGCP are averaged over 4 random graphs at each point which are probably different from the 4 graphs used by VCS.

On the one hand, observe that the exact coloring algorithm used in VCS generally is not faster for coloring the detected critical subgraphs than for coloring the original graphs, although the critical subgraphs are significantly smaller, suggesting that the VCS approach, as well as clause learning, probably is not very effective for random graphs. On the other hand, the main overhead in cdclGCP is the (long) clause management, despite that the learnt clauses allow to reduce the search trees (see Table 2 and Fig. 3). Since the critical subgraphs are significantly smaller, cdclGCP might learn much fewer and shorter clauses when coloring them. Consequently the overhead caused by the clause management in cdclGCP for coloring the critical subgraphs might be significantly smaller than for coloring the original graphs, which suggests that it might be interesting to combine VCS and clause learning in the future to find the chromatic number of a graph.

5.5.2. *Comparison Result on DIMACS Graphs*

Table 7 compares cdclGCP, B&C, MMT-BP and PASS on DIMACS graphs. If an algorithm is able to find and prove the chromatic number of a graph (i.e., LB = UB) within a cutoff time (specified below), the runtime used is displayed. The entry “init” in the runtime column means that the chromatic number of a graph is found during the initialization phase of the corresponding algorithm, and the entry “out” means that the corresponding algorithm fails to achieve optimality within the cutoff time.

For cdclGCP, the entries in column LB are given by Algorithm 5 and the entries in column UB are given by Algorithm 4, except the graphs for which LB = UB because both Algorithm 4 and 5 achieve optimality within the cutoff time for these graphs. The displayed runtime of cdclGCP for a graph G is the runtime of Algorithm 4, including the runtime of MaxCLQ and the runtimes of cdclGCP for all k from $|V|-1$ to $\chi(G)-1$. The difference between Algorithms 4 and 5 is for the graphs they cannot achieve the optimality: for these graphs, Algorithm 4 improves UB, while Algorithm 5 improves LB. The cutoff time is the same for the two algorithms.

The B&C, MMT-BP and PASS entries are taken directly from their original sources [10–12] and have not been calibrated. The runtimes of B&C were obtained on a Sun ULTRA workstation @140 MHz with 288 MB of RAM, which took 24 s to compute *r* 500.5. The cutoff time used by B&C was 7200 s in the experiment. The results of MMT-BP were obtained from a PIV@2.4 GHz with 2GB RAM with windows XP system, of which the user times of *dfmax* for *r* 400.5 and *r* 500.5 are 2.0 s and 7.0 s respectively (while the user times of *dfmax* for these two graphs on our machine are 2.28 s and 8.72 s respectively). The cutoff time MMT-BP used was 2400 s. In [12], the cutoff time for PASS was 1500 s.

Since our machine is about 1.62 times slower than the machine of PASS, by comparing the runtimes of *dfmax* to solve *r* 400.5 and *r* 500.5, we set the cutoff time of cdclGCP to 2400 s on our machine. The displayed runtime of cdclGCP in Table 7 for a graph G has been divided by 1.62 to make a direct comparison with the newest PASS.

Comparing the results of cdclGCP with B&C, cdclGCP proves the chromatic number of 18 graphs (i.e., LB = UB) that B&C cannot solve. There is no instance for which B&C achieves optimality but cdclGCP does not. Furthermore, B&C finds tighter UB for 2 graphs, but cdclGCP finds tighter UB for 25 graphs.

Comparing the results of cdclGCP with MMT-BP, MMT-BP proves the optimal coloring for 13 graphs that cdclGCP cannot solve, while cdclGCP achieves optimality for 16 graphs that MMT-BP cannot solve. For other graphs that neither cdclGCP nor MMT-BP can find optimal coloring, MMT-BP tends to find better UB for the *queen*, *le*450, *DSJC* and *wap* families. Nevertheless, we believe that it is more important for an exact algorithm to find better LB because one often can use approximate algorithms to find better UB than exact algorithms. Using Algorithm 5 instead of Algorithm 4, cdclGCP also achieves optimality within the cutoff time for the 16 graphs that MMT-BP cannot solve, and finds better LB than MMT-BP for 16 other graphs (including the DSJC family) that neither MMT-BP nor cdclGCP can solve, while MMT-BP finds better LB only for 4 graphs. Recall that MMT-BP and cdclGCP are very different because MMT-BP is a branch-and-price algorithm based on column generation.

Comparing cdclGCP and the newest PASS, cdclGCP proves optimal coloring for 16 graphs that PASS cannot solve, while PASS achieves optimality for 1 graph that cdclGCP cannot solve. For graphs neither PASS nor cdclGCP can solve, PASS finds better UB for 7 graphs and cdclGCP finds better UB for 6 graphs. Note that the LB returned by PASS for a graph is simply the cardinality of a clique computed within 5 s during initialization. It is natural that cdclGCP via Algorithm 5 finds better LB for many graphs. Recall that PASS and cdclGCP are both based on DSATUR, and PASS might also be improved using clause learning.

Finally, we find that cdclGCP is quite competitive in *myciel*, *Insertions*, *FullIns*, *ash*, *abb* and *mug* families. For these graphs, cdclGCP outperforms significantly the three other exact state-of-the-art algorithms considered in Table 7. In particular, cdclGCP allows to close the open instance 4-Fullins_5.

In the next experiment, we compare VCS and cdclGCP on the DIMACS instances used in [34] for evaluating VCS. These instances are divided into three categories. The first category contains instances that are probably critical themselves. Table 8 displays the results of VCS taken from [34] and the results of cdclGCP for this category. The first 4 columns contain the name of the instances, their number of vertices, their number of edges, and the best known upper bound (k) of their chromatic number in the literature. The next three columns (for VCS) display the number of vertices and the number of edges of the *k-VCS*, which are the same as in the original graphs, and the number of backtracks (*btk*) VCS needs to show that k is the chromatic number of the *k-VCS*. The values enclosed in parentheses are the number of backtracks after reaching a 4 h CPU time limit, meaning that no value was obtained for $\chi(G)$ in this case. The last column for

[2] www.satcompetition.org.

Table 8
Comparison of cdclGCP with VCS on DIMACS graphs which are probably critical themselves. The entry k in bold face means the $k=\chi(G)$. The entry *btk* in bold face for cdclGCP means the clearly better result.

DIMACS				VCS			cdclGCP
Name	$\|V\|$	$\|E\|$	*k*	$\|V'\|$	$\|E'\|$	*btk*	*btk*
myciel5	47	236	**6**	47	236	30,998	**4899**
myciel6	95	755	**7**	95	755	(138,446,852)	**11,862,795**
myciel7	191	2360	**8**	191	2360	(77,223,695)	(34,750,697)
mug88_1	88	146	**4**	88	146	2,204,467	**122**
mug88_25	88	146	**4**	88	146	942,961	**88**
mug100_1	100	166	**4**	100	166	1,406,570	**139**
mug100_25	100	166	**4**	100	166	974,170	**142**
1_Insertion_4	67	232	**5**	67	232	104,296,036	**1,152,462**
1_Insertion_5	202	1227	6	202	1227	(133,727,661)	(36,217,000)
1_Insertion_6	607	6337	7	607	6337	(50,929,137)	(23,765,345)
2_Insertion_4	149	541	5	149	541	(154,902,785	(40,514,000)
2_Insertion_5	597	3936	6	597	3936	(48,458,541)	(27,741,481)
3_Insertion_3	56	110	**4**	56	110	723,616	**1361**
3_Insertion_4	281	1046	5	281	1046	(95,076,991)	(35,512,000)
3_Insertion_5	1406	96,957	6	1406	9695	(13,784,327)	(17,059,234)
4_Insertion_3	79	156	**4**	79	156	(228,367,528)	**87,455**
4_Insertion_4	475	1795	5	475	1795	(70,891,706)	(34,154,256)

Table 9
Comparison of cdclGCP with VCS on the third category of DIMACS graphs. The entry *UB* in bold face means the $UB=\chi(G)$. The entry in bold face for VCS or cdclGCP means the clearly better result.

DIMACS				VCS					cdclGCP	
Name	$\|V\|$	$\|E\|$	*UB*	*btk*	*k*	$\|V'\|$	$\|E'\|$	*btk'*	*LB*	*btk*
DSJC125.1	125	736	**5**	227	5	10	26	**1**	5	127
DSJC125.5	125	3891	**17**	(71,844,096)	14	70	1341	37,453,055	14	**9,27,177**
DSJC250.1	250	3218	8	(42,398,413)	6	64	362	2464	**7**	2,198,462
DSJC250.5	250	15,668	28	(32,205,501)	14	74	1505	22,670,005	**16**	7,081,951
DSJC500.1	500	12,458	12	(31,588,256)	6	65	369	6757	**7**	1,15,699
DSJR500.1	500	3555	**12**	(141,520,342)	12	12	66	0	12	0
DSJR500.1C	500	121,275	**85**	(6,401,403)	80	84	3477	1	**84**	1,799,713
DSJR500.5	500	58,862	**122**	(73,970,922)	90	90	4005	0	**122**	0
queen6_6	36	290	**7**	410	7	22	119	**45**	7	368
queen8_8	64	728	**9**	597,552	9	54	538	188,021	9	**1269**
queen9_9	81	2112	**10**	80,603,809	**10**	74	897	135,083,408	9	0
ash331GPIA	662	4185	**4**	14	4	9	16	2	4	8
1-FullIns_3	30	100	**4**	7	4	7	12	1	4	5
1-FullIns_4	93	593	**5**	5567	5	15	43	6	5	7
1-FullIns_5	282	3247	**6**	(106,523,508)	6	31	144	271	6	**218**
2-FullIns_3	52	201	**5**	1850	5	9	22	1	5	1
2-FullIns_4	212	1621	**6**	(209,999,176)	6	19	75	**8**	6	740
2-FullIns_5	852	12,201	**7**	(1,922,086)	7	39	244	715	7	606
3-FullIns_3	80	346	**6**	366,830	6	11	35	1	6	2
3-FullIns_4	405	3524	**7**	(164,058,937)	7	23	116	**10**	7	83,180
3-FullIns_5	2030	33,751	**8**	(34,366,333)	8	47	371	1675	8	**1210**
4-FullIns_3	114	541	**7**	80,247,163	7	13	51	1	7	3
4-FullIns_4	690	6650	**8**	(126,559,559)	8	27	166	**12**	8	26
5-FullIns_3	154	792	**8**	(448,858,523)	8	15	70	1	8	4

cdclGCP displays the number of backtracks cdclGCP needs to show that k is the chromatic number of the graphs. The values enclosed in parentheses are the number of backtracks after reaching a 2 h CPU time limit (our computer is about 2 times faster than the computer with a 1.6 GHz Athlon CPU and 512Mb of RAM used in [34]).

In Table 8, each instance solved by VCS is also solved by cdclGCP using a substantially smaller subtree. In addition, cdclGCP solved two more instances (myciel6, 4_insertion_3) that VCS did not solve within the cutoff time.

The second category in [34] contains instances which have maximum cliques as minimum *k-VCS* for $k=\chi(G)$, including graphs le450*, fpsol*, inithx*, mulsol*, zeroin*, school1*, miles*, anna, david, homer, huck, jean, and some queen* graphs. Combined with the best known upper bound of the chromatic number of G in the literature, these cliques allow to immediately establish the chromatic number of G. Since VCS and cdclGCP both easily find these maximum cliques, we do not display these graphs to better emphasize the two other categories.

The third category contains the instances that fall in none of the above two categories. Table 9 displays the results of VCS taken from [34] and the results of cdclGCP for this category. As in Table 8, the first 4 columns contains the name of the instances, their number of vertices, their number of edges, and the best known upper bound (*UB*) of their chromatic number gathered from the literature. The next five columns (for VCS) display the number of backtracks (*btk*) VCS needs to solve these DIMACS graphs, the lower bound k of $\chi(G)$ used by

Table 10
UB of a subset of hard DIMACS graphs obtained by cdclGCP (via Algorithm 4) and state-of-the-art approximate algorithms. LB obtained by cdclGCP (via Algorithm 5) are also reported. The cutoff time used by cdclGCP for both Algorithm 4 and 5 is 7200 s. Entry "?" means that the chromatic number of graph instance is unknown so far.

Instance	χ	cdclGCP		UB_{best}	References
		LB	*UB*		
le450_15c	15	15	23	15	[28–33]
le450_15d	15	15	23	15	[28–33]
le450_25c	25	25	27	25	[29–31,33]
le450_25d	25	25	27	25	[29–31,33]
DSJC125.1	5	5	5	5	[28–33]
DSJC125.5	17	14	19	17	[28–33]
DSJC125.9	44	39	48	44	[28–33]
DSJC250.1	?	6	9	8	[28–33]
DSJC250.5	?	16	35	28	[29–31]
DSJC250.9	?	48	86	72	[28–33]
DSJC500.1	?	7	15	12	[29–31,33]
DSJC500.5	?	17	61	48	[29–31,33]
DSJC1000.1	?	7	25	20	[29–31,33]
DSJC1000.5	?	18	115	83	[29,31–33]
DSJC1000.9	?	62	298	222	[32,33]
DSJR500.1	12	12	12	12	[28–33]
DSJR500.1c	85	84	85	85	[31]
DSJR500.5	122	122	129	122	[28,29,31,33]
latin_sq._10	?	90	127	97	[33]
R125.1	5	5	5	5	[28–33]
R125.1c	46	46	46	46	[28–33]
R125.5	36	36	36	36	[28–33]
R250.1	8	8	8	8	[28–33]
R250.1c	64	64	64	64	[28–33]
R250.5	65	65	65	65	[29,31,33]
R1000.1	20	20	20	20	[28–33]
R1000.1c	?	86	105	98	[29–31,33]
R1000.5	234	234	241	234	[28,29]
flat300_20_0	20	15	38	20	[28–33]
flat300_26_0	26	15	40	26	[28–33]
flat300_28_0	28	15	39	28	[30,31]
flat1000_50_0	50	18	112	50	[29,31,32]
flat1000_60_0	60	18	110	60	[29,31,32]
flat1000_76_0	76	18	111	82	[29,31–33]
C2000.5	?	19	205	146	[32]
C4000.5	?	19	391	260	[32]

VCS to search for a *k-VCS* of these graphs, the number of vertices and the number of edges of these *k-VCS*s, and the number of backtracks (*btk*') VCS needs to prove that *k* is the chromatic number of these *k-VCS*s. The values enclosed in parenthesis mean that VCS cannot determine the chromatic number of the corresponding DIMACS graphs within a 4 h CPU cutoff time. The last 2 columns (for cdclGCP) display the best lower bound (*LB*) cdclGCP found for the chromatic number of the original graphs within a 2 h CPU cutoff time using our computer, as well as the number of backtracks (*btk*) of cdclGCP to find these lower bounds.

From Table 9, the critical subgraphs detected by VCS are significantly smaller than the original graphs, but cdclGCP finds better lower bound than VCS for 5 graphs, while VCS finds better *LB* than cdclGCP for 1 graph(queen9_9).

It is well-known that approximate algorithms for GCP are often better than exact algorithms for computing UB, while they usually are unable to compute LB. We refer to [25,26] for a comprehensive survey of approximate algorithms for GCP and a wide-ranging computational comparison of high-performance GCP algorithms. Table 10 compares the UB of a subset of hard DIMACS graphs computed by cdclGCP within 7200 s with the best known UB reported by an approximate algorithm in the literature. We also give in Table 10 the references reporting the best UB.

Table 10 confirms that approximate algorithms are better than cdclGCP for computing UB of random and quasi-random graphs such as *DSJC*, *flat*, and *C*. Nevertheless, approximate algorithms and exact algorithms may be complementary: the LB obtained by cdclGCP may be combined with the UB obtained by approximate algorithms to prove the chromatic number of 6 graphs (le450_15c, le450_15d, le450_25c, le450_25d, *DSJR* 500.5, *R* 1000.5) in Table 10. We note that cdclGCP computes better LB for 21 DIMACS graphs than all the three state-of-the-art exact algorithms B&C, MMT-BP and PASS according to Table 7, and that among these 21 graphs, cdclGCP solves 10 graphs (i.e., Algorithm 5 finds the chromatic number of these graphs within the cutoff time) that B&C, MMT-BP and PASS cannot solve.

6. Conclusions

We proposed a new exact algorithm with clause learning for the graph coloring problem. The proposed algorithm cdclGCP extends a backtracking algorithm backGCP by applying the CDCL technology to discover the implicit constraints among vertices. The implicit constraints are used to prune the search space. Experimental results show that the search tree size is significantly reduced. For random graphs, the learnt implicit constraints are very complex, so that the overhead caused by the CDCL technology is too important and slows down backGCP. However, for structured graphs, the learnt implicit constraints capture well the structures in the graphs and the algorithm cdclGCP resulted from combining backGCP with CDCL is very efficient. In particular, cdclGCP is able to close the open DIMACS instance 4-Fullins_5. In fact, to our knowledge, it is the first time that an exact algorithm for GCP reports the chromatic number of this instance in the literature.

In the future, we plan to study better clause management strategies for GCP to remove useless clauses and to keep relevant clauses for pruning the search space. We also plan to combine the approach VCS and the CDCL technology in cdclGCP to solve some large GCP problems.

References

[1] Gamache M, Hertz A, Ouellet JO. A graph coloring model for a feasibility problem in monthly crew scheduling with preferential bidding. Comput Oper Res 2007;34(8): 2384–95.
[2] Zufferey N, Amstutz P, Giaccari P. Graph colouring approaches for a satellite range scheduling problem. J Schedul 2008;11(4):263–77.
[3] Werra DD. An introduction to timetabling. Eur J Oper Res 1985;19:151–62.
[4] Burke EK, McCollum B, Meisels A, Petrovic S, Qu R. A graph-based hyper heuristic for timetabling problems. Eur J Oper Res 2007;176:177–92.
[5] Werra DD, Eisenbeis C, Lelait S, Marmol B. On a graph-theoretical model for cyclic register allocation. Discret Appl Math 1999;93(2–3):191–203.
[6] Smith DH, Hurley S, Thiel SU. Improving heuristics for the frequency assignment problem. Eur J Oper Res 1998;107(1):76–86.
[7] Woo TK, Su SYW, Wolfe RN. Resource allocation in a dynamically partitionable bus network using a graph coloring algorithm. IEEE Trans Commun 2002;39:1794–801.
[8] Méndez-Díaz I, Zabala P. A cutting plane algorithm for graph coloring. Discret Appl Math 2008;156(2):159–79.
[9] Lucet C, Mendes F, Moukrim A. An exact method for graph coloring. Comput Oper Res 2006;33(8):2189–207.
[10] Méndez-Díaz I, Zabala P. A branch-and-cut algorithm for graph coloring. Discret Appl Math 2006;154:826–47.
[11] Malaguti E, Monaci M, Toth P. An exact approach for the Vertex Coloring Problem. Discret Optim 2011;8(2):174–90.
[12] Segundo PS. A new DSATUR-based algorithm for exact vertex coloring. Comput Oper Res 2012;39:1724–33.
[13] Chaitin GJ. Register allocation and spilling via graph coloring. SIGPLAN Not 2004;39(4):66–74.
[14] Caramia M, DellOlmo P. Coloring graphs by iterated local search traversing feasible and infeasible solutions. Discret Appl Math 2008;156(2):201–17.
[15] Porumbel DC, Hao JK, Kuntz P. Diversity control and multi-parent recombination for evolutionary graph coloring algorithms. In: 9th European conference on evolutionary computation in combinatorial optimisation (EVOCOP2009). Tbingen, Germany; 2009. p. 121–32.
[16] Dowsland KA, Thompson JM. An improved ant colony optimisation heuristic for graph coloring. Discret Appl Math 2008;156(3):313–24.
[17] Galinier P, Hertz A, Zufferey N. An adaptive memory algorithm for the k-coloring problem. Discret Appl Math 2008;156(2):267–79.
[18] Prestwich SD. Generalized graph coloring by a hybrid of local search and constraint programming. Discret Appl Math 2008;156(2):148–58.
[19] Mehrotra A, Trick MA. A column generation approach for graph coloring. INFORMS J Comput 1996;8:344–54.
[20] Brélatz D. New methods to color the vertices of a graph. Commun ACM 1979;22:251–6.
[21] Eén N, Sorensson N. An extensible SAT solver. In: 6th international conference on theory and applications of satisfiability testing, Lecture Notes in Computer Science, vol. 2919; 2003. p. 502–18.
[22] Zhang L, Madigan CF, Moskewicz MH, Malik S. Efficient conflict driven learning in a Boolean satisfiability solver. Proc ICCAD 2001:279–85.
[23] Li CM, Quan Z. An efficient branch-and-bound algorithm based on maxsat for the maximum clique problem. In: Proceedings of the 24th AAAI conference on artificial intelligence, AAAI 2010. Atlanta, Georgia, USA: AAAI Press; 2010. p. 128–33.
[24] Li CM, Quan Z. Combining graph structure exploitation and propositional reasoning for the maximum clique problem. In: Proceedings of the 22th IEEE international conference on tools with artificial intelligence, ICTAI 2010. Arras, France; 2010. p. 344–51.
[25] Galinier P, Hertz A. A survey of local search methods for graph coloring. Comput Oper Res 2006;9:2547–62.
[26] Lewis R, Thompson J, Mumford C, Gillard J. A wide-ranging computational comparison of hight-performance graph coloring algorithms. Comput Oper Res 2012;39:1933–50.
[27] Audemard G, Simon L. Predicting learnt clauses quality in modern SAT solvers. In: Proceedings of the 21st international joint conference on Artifical intelligence; 2009. p. 399–404.
[28] Prestwich S. Coloration neighbourhood search with forward checking. Ann Math Artif Intell 2002;34(4):327–40.
[29] Malaguti E, Monaci M, Toth P. A metaheuristic approach for the vertex coloring problem. INFORMS J Comput 2008;20(2):302.
[30] Porumbel D, Hao JK, Kuntz P. A search space "Cartography" for guiding graph coloring heuristics. Comput Oper Res 2010;37(4):769–78.
[31] Lü Zhipeng, Hao Jin-Kao. A memetic algorithm for graph coloring. Eur J Oper Res 2010;203(1):241–50.
[32] Wu Qinghua, Hao Jin-Kao. Coloring large graphs based on independent set extraction. Comput Oper Res 2012;39:283–90.
[33] Titiloye Olawale, Crispin Alan. Graph coloring with a distributed hybrid quantum annealing algorithm. In: O'Shea J, Nguyen N, Crockett K, Howlett R, Jain L, editors. Agent and multi-agent systems: technologies and applications, lecture notes in computer science, vol. 6682. Berlin/Heidelberg: Springer; 2011. p. 553–62, ISBN 978-3-642-21999-3.
[34] Desrosiers C, Galinier P, Hertz A. Efficient algorithms for finding critical subgraphs. Discret Appl Math 2008;156(2):244–66.
[35] Herrmann F, Hertz A. Finding the chromatic number by means of critical graphs. ACM J Exp Algorithm 2002;7(10):1–9.