SAT-boosted Tabu Search for Coloring Massive Graphs

ANDRÉ SCHIDLER and STEFAN SZEIDER, TU Wien, Austria

Graph coloring is the problem of coloring the vertices of a graph with as few colors as possible, avoiding monochromatic edges. It is one of the most fundamental NP-hard computational problems. For decades researchers have developed exact and heuristic methods for graph coloring. While methods based on propositional satisfiability (SAT) feature prominently among these exact methods, the encoding size is prohibitive for large graphs. For such graphs, heuristic methods have been proposed, with tabu search among the most successful ones.

In this article, we enhance tabu search for graph coloring within the SAT-based local improvement (SLIM) framework. Our hybrid algorithm incrementally improves a candidate solution by repeatedly selecting small subgraphs and coloring them optimally with a SAT solver. This approach scales to dense graphs with several hundred thousand vertices and over 1.5 billion edges. Our experimental evaluation shows that our hybrid algorithm beats state-of-the-art methods on large dense graphs.

CCS Concepts: • **Theory of computation** → **Tabu search**; **Constraint and logic programming;**

Additional Key Words and Phrases: Graph coloring, SAT encoding, tabu search, SAT-based local improvement, massive graphs, experiments, cg:shop

ACM Reference format:
André Schidler and Stefan Szeider. 2023. SAT-boosted Tabu Search for Coloring Massive Graphs. *ACM J. Exp. Algor.* 28, 1, Article 1.5 (July 2023), 19 pages.
https://doi.org/10.1145/3603112

1 INTRODUCTION

Graph coloring is the fundamental computational problem of coloring the vertices of a given undirected graph with as few colors as possible, avoiding monochromatic edges, or, equivalently, partitioning the graph's vertex set into as few independent sets as possible. Graph coloring arises naturally in many applications, including scheduling, register allocation, pattern matching, and computational geometry. The decision version of the problem—where the number of colors is given, and one asks whether a coloring exists—can be naturally cast as a constraint satisfaction problem: The graph's vertices are variables that range over a finite domain of colors, and each edge represents a binary inequality constraint. Graph coloring is one of Karp's 21 fundamental NP-hardproblems [20].

We acknowledge the support from the Austrian Science Fund (FWF), projects P32441, P36420, and W1255; and from the Vienna Science and Technology Fund (WWTF), project ICT19-065.
Authors' address: A. Schidler and S. Szeider, TU Wien, Favoritenstrasse 9-11, Vienna, Austria, 1040; emails: {aschidler, sz}@ac.tuwien.ac.at.

1084-6654/2023/07-ART1.5 $15.00
https://doi.org/10.1145/3603112

For decades, much research has been devoted to developing algorithmic methods for graph coloring. One can distinguish between *exact methods* that search for a coloring with the smallest number of colors possible and *heuristic methods* that possibly yield suboptimal colorings.

Exact methods for graph coloring include**constraint programming (CP)**, **propositional satisfiability (SAT)**, and **integer linear programming (ILP)** formulations [6, 16, 18]. Here, the problem is expressed in terms of constraints, propositional logic, or linear constraints over integer domains, respectively, and then solved by a general solver. Generally, these exact methods do not scale to graphs with more than a few thousand vertices, as these encodings become prohibitively large. In our experiments, the largest graph successfully colored by a SAT encoding had around 14,000 vertices and was comparatively easy to color due to the graph's sparsity.

Heuristic graph coloring methods include various forms of greedy colorings combined with local search, especially tabu search, reducing the number of colors used by the greedy coloring [4, 5, 17]. Such heuristic methods scale to very large graphs and find good colorings for sparse graphs but struggle with large, dense graphs.

The CG:SHOP Challenge 2022[1] posed the ***Minimum Partition into Plane Subgraphs Problem (MPPS)***: the problem of finding the smallest number of classes we can partition a given set of line segments into, such that line segments within the same class do not intersect. This problem is reducible to graph coloring (see Section 2.2) and the competition instances were crafted such that they are noticeably different from well-known graph coloring instances [10] and yield graph coloring instances that are comparatively large and dense graphs. Since the aforementioned methods do not perform well on them,new approaches for graph coloring were developed, one of which is this article's subject.

In this article, we propose a hybrid approach between exact and heuristic techniques, following the general framework of **SAT-based Local Improvement Method (*SLIM*)** that has recently been successfully customized for various problems [13, 15, 21, 26, 27, 30, 31, 33]. Our idea is to enhance tabu search by applying SAT encodings locally. Our hybrid algorithm *GC-SLIM* incrementally improves a candidate coloring by repeatedly selecting small subgraphs (local instances) and coloring them optimally with a SAT solver. The problem solved by the SAT solver is a list coloring problem, where each vertex has a list of available colors. The lists ensure that the subgraph's coloring is consistent with the colors of the vertices outside the selected subgraph. GC-SLIM's most essential ingredients include strategies for *local instance selection*, the *SAT-based solution* of the local instance, and a technique called *chain propagation*.

GC-SLIM scales to dense graphs with several hundred thousand vertices and over 1.5 billion edges. Our experimental evaluation shows that our hybrid algorithm beats state-of-the-art methods on large, dense graphs.

1.1 Related Work

Since the work on graph coloring is extensive, we discuss only the most relevant work for this article. We refer to Sun's dissertation [35] for a more exhaustive survey on graph coloring algorithms.

Greedy colorings are the most common and easy heuristics for graph coloring. Given an ordering of the vertices, each vertex gets assigned the smallest color that avoids monochromatic edges in the given order. Different heuristics use different orderings. *DSatur* [5] is one of the most successful greedy heuristics, and we use it in our approach. DSatur always chooses as the next vertex one that is most constrained, i.e., one with the fewest colors available.

Tabu search has been successfully used for graph coloring. Most relevant to this article is *Partialcol* [4], which we discuss in more detail in Section 2.3.

[1]https://cgshop.ibr.cs.tu-bs.de/competition/cg-shop-2022/.

Iterated-DSatur (I-DSatur) [19] is a SAT-based extension of DSatur that combines DSatur with extensive pre-processing and SAT-solving to a new method that can compute optimal colorings for small graphs. Used as a heuristic, it scales to sparse graphs with several million vertices. I-DSatur adds a reordering mechanism to DSatur invoked whenever the current uncolored vertex v cannot be colored with any of the existing colors, i.e., the current partial k-coloring would become a partial $(k+1)$-coloring. At this point, I-DSatur tries to find a better coloring for all the vertices colored so far and v. If successful, then no new color is required; if unsuccessful, then the best lower bound on the number of required colors known to I-DSatur can be increased. The main difference between GC-SLIM and I-DSatur is that GC-SLIM tries to reduce the number of colors by improving several smaller local instances, while I-DSatur tries to find improvements for a single local instance that is as large as possible. The former scales better on dense graphs, while the latter performs better on sparse graphs, as we will further discuss in our experimental evaluation.

Large graphs yield a prohibitively large encoding size when the standard SAT encodings for graph coloring are used. Recently, a new approach based on clause learning has been proposed [16, 18], which can circumvent the size issue for many instances. Here, only those clauses required for a correct solution are added iteratively. This approach is also used in I-DSatur [19].

Much research in recent years focused on very large and sparse graphs. The advantage of sparse graphs is that they can often be colored with a small number of colors relative to their size and are easily reducible to smaller graphs. State-of-the-art approaches use these and other structural properties of sparse graphs to scale to graphs with millions of vertices [25, 32, 37]. We compare GC-SLIM to the most recent such algorithms FastColor [25] and I-DSatur [19].

The top three submissions to the CG:SHOP Challenge 2022 used different variations of the same idea. They perform local search guided by a conflict score, i.e., how often a vertex has been recolored [8, 9, 14, 34]. This strategy performed better on the competition instances than other established local search strategies. We will further discuss this strategy in Section 2.3.

2 PRELIMINARIES

2.1 Graphs and Colorings

We consider a connected simple graph G with the set of vertices $V(G)$ and set of edges $E(G)$. We will often assume without loss of generality that $V(G) = \{1, \dots, |V|\}$. We denote the edge between vertices $v, w \in V(G)$ by vw or equivalently wv. For $X \subseteq V(G)$, we denote by $N_G(X) = \{\, u \mid uv \in E(G) \text{ with } v \in X \text{ and } u \notin X \,\}$ the *neighborhood* of X. We write $N_G(v)$ instead of $N_G(\{v\})$ and drop the subscript if G is clear from the context. $G - X$ denotes the graph G' with $V(G') = V(G) \setminus X$ and $E(G') = \{\, uv \in E(G) \mid u, v \in V(G') \,\}$.

For an integer $k \geq 1$, we denote the set $\{1, \dots, k\}$ by $[k]$. A *partial k-coloring* of a graph G is a mapping $c : V(c) \to [k]$ defined on a set $V(c) \subseteq V(G)$ such that $c(u) \neq c(v)$ for every $uv \in E(G)$ with $u, v \in V(c)$. If $V(c) = V(G)$, then c is a *full k-coloring* or simply a *k-coloring* of G.[2] The *chromatic number* $\chi(G)$ of a graph G is the smallest k such that G has a k-coloring. We say a k-coloring is *optimal* if $k = \chi(G)$.

For a (partial) k-coloring c of G, we call the integers $[k]$ *colors* and the sets $c_\ell(G) = \{\, v \in V(G) \mid c(v) = \ell \,\}$, $\ell \in [k]$, the *color classes* of c. Observe that each color class is an independent set of G and that color classes are pairwise disjoint. We also write $c_0(G) = V(G) \setminus V(c)$ for the set of *uncolored* vertices and write $c(v) = 0$ for a vertex $v \in V(G) \setminus V(c)$. We write c_ℓ, instead of $c_\ell(G)$, if G is clear from the context. Since its color classes uniquely determine a partial k-coloring, we will often specify a k-coloring this way. Further, we write $N_{G,c,\ell}(X) = N_G(X) \cap c_\ell(G)$, the

[2] Some authors use the term *k-coloring* to refer to a mapping $c : V(G) \to [k]$ that allows monochromatic edges (edges $uv \in E(G)$ with $c(u) = c(v)$) and call c a *proper* k-coloring if it has no monochromatic edges.

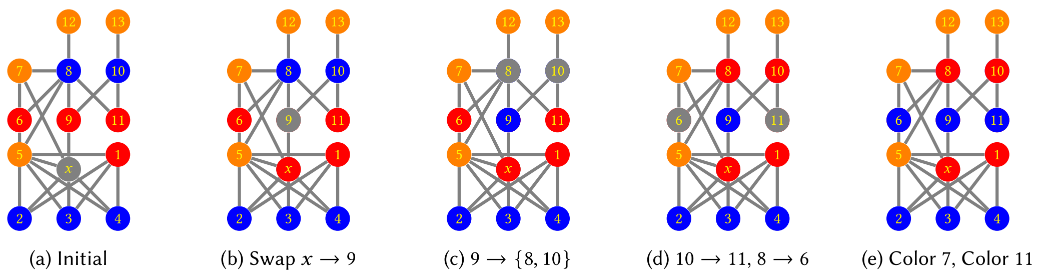


Fig. 1. Tabu search example that shows how vertex x is colored through a series of swaps. Note that for the last two graphs, two swaps are performed at once.

ℓ-colored neighborhood. Whenever G and c are clear from context, we drop the subscript and use $N_\ell(X)$. The *prevalence* of a color ℓ is $|c_\ell|$, and the prevalence of a color ℓ with respect to a vertex v is $|N_{G,c,\ell}(v)|$. Therefore, a least prevalent color of a k-coloring c in the neighborhood of v is $\arg\min_{\ell \in [k]} |N_{G,c,\ell}(v)|$.

The *graph coloring problem* takes an undirected graph G as input; the task is to produce a coloring of G that uses the least possible number of colors. The decision version of the problem takes as input G and an integer k; the task is to decide whether G admits a k-coloring.

2.2 Minimum Partition into Plane Subgraphs Problem (MPPS)

The ***Minimum Partition into Plane Subgraphs Problem (MPPS)*** takes as an instance a geometric graph G, with vertices $V(G)$ represented by points in the plane, and edges $E(G)$ by straight-line connections between vertices. The task is to finda partitioning of E into as few classes $E_1, \ldots, E_k$ as possible, such that each subgraph G_i, with $V(G_i) = V(G)$ and $E(G_i) = E_i$, is plane.

In this article, we consider the MPPS problem in terms of graph coloring. There is a natural reduction from the MPPS problem to graph coloring, which reduces an MPPS instance G to the conflict graph G', containing a vertex for each line segment and where two vertices are adjacent if the corresponding line segments intersect. Evidently, G admits a partitioning into k plane subgraphs if and only if G' has a k-coloring.

2.3 Tabu Search

Tabu search is a very successful local search approach to graph coloring. We use *Partialcol*'s search strategy [4]. Starting from a (non-optimal) $(k+1)$-coloring c of the given graph G, Partialcol selects a color $e \in [k+1]$ to eliminate. The vertices in c_e are then removed from c and considered uncolored, making c a partial k-coloring. Partialcol now tries to complete c and color the vertices in c_0 by performing *swaps*: For a partial k-coloring c, a vertex $v^* \in c_0$, and a color $\ell \in [k]$, a (color) *swap* of v^* to ℓ is obtained from c by setting $c(v^*) := \ell$, and $c(w) := 0$ for all $w \in N_\ell(v^*)$. The swap is a *p-swap* if $|N_\ell(v^*)| = p$. Let $u = |c_0|$ be the number of uncolored vertices before the swap, then $|c_0| = u + p - 1$ after a p-swap.

In each iteration, the algorithm performs a p-swap with smallest p. The choice of color ℓ for the p-swap is restricted by a *tabu* list for vertex v^*: a list of the colors assigned to v^* in the last few iterations. This mechanism ensures that vertices do not get re-assigned the same colors within a certain number of iterations and forces the algorithm to explore more of the search space. Figure 1 shows how a series of swaps can empty c_0.

Partialcol terminates if $c_0 = \emptyset$, in which case c is now a full k-coloring, or when it reaches a prescribed number of iterations. Usually, tabu search is run repeatedly, choosing different colors to eliminate.

Conflict Scores. The winning submissions [8, 9, 14, 34] to the CG:SHOP Challenge are based on heuristic algorithms that utilize a different selection criterion based on the *conflict count.* For a vertex v, the conflict count $q(v)$ measures how often v has been removed from c_0 by a swap, i.e., how often v has been colored. Initially, we set $q(v) = 0$ for all vertices v. The conflict count is then used to calculate a conflict score that is used for picking the next swap. The different submissions calculate the conflict score differently. We follow the approach by Spalding-Jamieson et al. [34] due to its simplicity: For the next swap, the solver picks a random vertex $v \in c_0$ and swaps it to the color ℓ that minimizes $\sum_{u \in N_\ell(v)}(1 + q(u)^2)$.

3 SAT-BASED LOCAL IMPROVEMENT FOR GRAPH COLORING

The propositional **satisfiability problem (SAT)** asks whether a given propositional formula is satisfiable. As the first problem to be shown to be NP-complete [7, 24], it forms a cornerstone in computational complexity. In contrast to its theoretical hardness, SAT provides an important framework for solving hard combinatorial problems in practice by encoding instances in propositional logic and solving them with a SAT solver [12]. Today's SAT solvers are extremely efficient, robust, and can routinely solve instances that encode real-world problems with hundreds of thousands of variables. The progress achieved by algorithm engineering for SAT is "*nothing short of spectacular*" [36]. SAT-based methods automatically benefit from further improvements to SAT solvers, making them even more attractive.

SAT-based Local Improvement (SLIM) is an *anytime* meta-heuristic that embeds SAT encodings into heuristic algorithms. It improves a given (sub-optimal) global solution through a series of local improvements accomplished by a SAT solver. SLIM has been successfully utilized in several applications [13, 26, 27, 30, 33] and allows us to apply the solving power of SAT to instances that are too large to be encoded as a whole to SAT. Instead, we repeatedly choose smaller *local instances* that can be quickly encoded and solved. SLIM is a special case of Large Neighborhood Search [29], distinguishing itself by combining a structurally constrained notion of a neighborhood with a complete method (SAT).

Our new SLIM approach to graph coloring, *GC-SLIM*, tries to eliminate one color at a time in a fashion similar to Partialcol. Starting from a heuristically computed $(k + 1)$-coloring, GC-SLIM selects a color $e \in [k + 1]$, removes e from c, and tries to iteratively recolor subgraphs using a SAT solver until all vertices are colored, and c gives rise to a k-coloring.

We first discuss the core of every SLIM algorithm: a method to extract local instances such that their improvement eventually translates to an overall improvement. First, we discuss how we define local instances, i.e., we show how we can color subgraphs of G with a SAT solver while maintaining consistency with the coloring of the remaining graph. Then, we discuss how we find good local instances. We also discuss further additions to GC-SLIM that enhance its performance.

3.1 Local Instances and SAT

Let G be the input graph and c a partial k-coloring of G. Since G is too large to be encoded as a whole to SAT, we select a subset $X \subseteq V(G)$, based on a process described in the next subsection, limiting the size of X in terms of a budget parameter b. The goal is now to find a partial k-coloring for the induced subgraph G', with $V(G') = X$ and $E(G') = \{ uv \in E(G) \mid u, v \in X \}$.

However, a newly found k-coloring of G' will, in general, not be compatible with the coloring c of the vertices outside X. We consider the vertices adjacent to X as extra constraints by defining the local instance in terms of the *list coloring* problem: Let L be a mapping that assigns each vertex $v \in X$ a set $L(v) \subseteq [k]$, called the *list* of v. Here in particular, we let

$$L(v) = [k] \setminus \{ c(u) \mid u \in N_G(v) \setminus X \}.$$

A *partial list coloring* of (G', L) is partial k-coloring c' of G' with the additional property that $c'(v) \in L(v)$ for each $v \in V(c')$. Let $c \cup c'$ denote the partial k-coloring obtained by composing c and c':

$$(c \cup c')(v) = \begin{cases} c(v) & \text{if } v \in V(c) \setminus X; \\ c'(v) & \text{if } v \in V(c'). \end{cases}$$

The following lemma provides an important link between colorings and list colorings:

Lemma 3.1. *Given a graph G and $X \subseteq V(G)$, let c be a partial k-coloring of G, (G', L) be the local instance for X, and c' be a partial list coloring of (G', L). Then, $c \cup c'$ is a partial k-coloring of G.*

Proof. Consider an edge $uv \in E(G)$. If $u, v \in V(c) \setminus X$, then $(c \cup c')(u) = c(u) \neq c(v) = (c \cup c')(v)$. If $u, v \in X \cap V(c')$, then $(c \cup c')(u) = c'(u) \neq c'(v) = (c \cup c')(v)$. If $u \in V(c) \setminus X$ and $v \in V(c')$, then $c(u) \notin L(v)$, since $u \in N_G(v)$, hence $(c \cup c')(u) = c(u) \neq c'(v) = (c \cup c')(v)$. □

We note in passing that the list coloring problem is a proper generalization of the graph coloring problem. For instance, graph coloring is fixed-parameter tractable in the graph's treewidth, while list coloring is W[1]-hard when parameterized by treewidth [11].

Our general aim is to increase the number of colored vertices. Ideally, we would find a full k-coloring for (G', L). While this is often not possible, it turns out that it is still useful to obtain a partial list coloring c' of (G', L), which colors all previously uncolored vertices and minimizes the number of newly introduced uncolored vertices.

We achieve this by a slight tweak of the local instance. For all $v \in X \setminus c_0$, we add 0 to $L(v)$ and thereby allow them to become uncolored. The problem is now a minimization problem: find a partial list coloring c' for (G', L) that minimizes $|c'_0|$.

We encode the existence of a partial list coloring of G' that minimizes the number of uncolored vertices. To this end, for $r \leq |X|$, we define a propositional formula $F(G', L, r)$ that is satisfiable if and only if (G', L) has a partial list coloring c' where $|c'_0| \leq r$. We can minimize the number of uncolored vertices by solving $F(G', L, r)$ for different values of r.

The encoding requires one set of variables and two sets of clauses. For each $v \in X$ and $\ell \in L(v)$, the variable $c_{v,\ell}$ is true if and only if $v \in V(c')$ and $c'(v) = \ell$ or $v \notin V(c')$ and $\ell = 0$.

Hence, the first set of clauses encodes that each vertex $v \in X$ is assigned at least one color $\ell \in L(v)$ or is set to 0:

$$\bigwedge_{v \in X} \bigvee_{\ell \in L(v)} c_{v,\ell}.$$

$c_{v,0}$ is true if and only if $v \notin V(c')$.

The second set of clauses encodes that adjacent vertices in G' must not have the same color:

$$\bigwedge_{\substack{vw \in E(G'), v < w, \\ \ell \in L(u) \cap L(v), \ell \neq 0}} \neg c_{v,\ell} \vee \neg c_{w,\ell}.$$

Note that $c_{v,0}$ and $c_{w,0}$ can both be true even if $vw \in E(G)$. Finally, we use a *totalizer encoding* [2] to express the cardinality constraint

$$|\{\, v \in X \mid c_{v,0} = \text{true} \,\}| \leq r.$$

The constraint adds $\Theta(|X| \cdot \log |X|)$ many variables and $\Theta(|X|^2)$ many clauses.

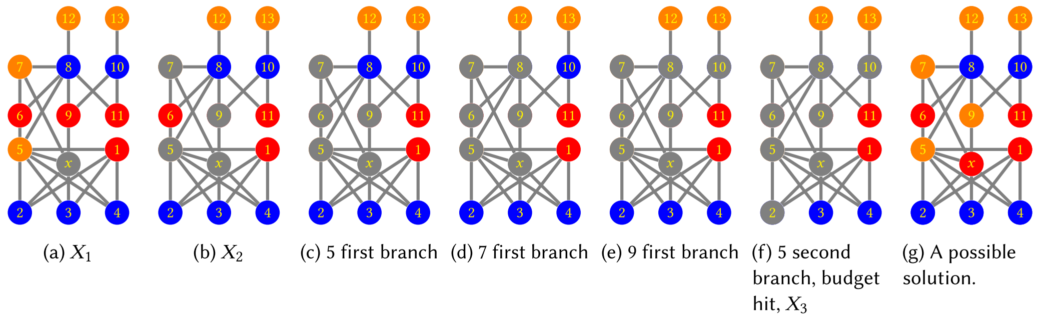


Fig. 2. Example for local instance selection with branching factor 2 and a budget of 8. The selected component is indicated by discolored (gray) vertices.

3.2 Local Instance Selection

In this section, we describe how GC-SLIM constructs local instances for the SAT encoding described in the previous section. Let G be the input graph and c a partial k-coloring of G. Our goal is to select a suitable subset $X \subseteq V(G)$ that defines our local instance. The overall approach is to start at a single uncolored vertex and perform a breadth-first search among the least prevalent colors in the neighborhoods where the size of X is limited by a *budget* b and the breadth by a *branching factor* f.

We first select an uncolored vertex $v^* \in c_0$. We initially put $X_0 = \emptyset$, $X_1 = \{v^*\}$ and continue computing a chain of sets $X_0 \subsetneq X_1 \subsetneq \cdots \subsetneq X_s$ as long as $|X_s| \leq b$. If no further addition is possible, then we stop, as we have found the set $X = X_s$.

Assume we have constructed X_i, $i \geq 1$. We now construct X_{i+1} by starting from $X_{i+1} := X_i$ and incrementally extending X_{i+1}. Let $S = X_i \setminus X_{i-1}$. For each $w \in S$, we find the smallest non-empty set $N_\ell(w)$, $\ell \in [k]$ such that $N_\ell(w) \cap X_{i+1} = \emptyset$. If $|N_\ell(w)| + |X_{i+1}| \leq b$, we add $N_\ell(w)$ to X_{i+1} and in any case, we proceed to the next vertex in S. We repeat this step at most f times, i.e., for each vertex in S, we add at most f colors from the neighborhood of the vertex to finish constructing X_{i+1}.

We observe that $X \setminus V(c) = \{v^*\}$, i.e., v^* is the only uncolored vertex in X. Figure 2 illustrates local instance selection on a simple graph.

The goal of the budget b is to keep the size of the local instance small enough such that the SAT solver can solve it within an expected timeout. In practice, the best budget varies greatly with the instance, so we automatically adjust it. Whenever a specified number of consecutive SAT solver calls time out, the budget is decreased, and conversely, whenever the same number of consecutive SAT solver calls return a result, the budget is increased.

We described the process such that we always expand X_{i+1} using the color ℓ such that $N_\ell(w)$ is minimal. Alternatively, we can also use the conflict score discussed in Section 2.3. We discuss both options in our experimental section.

3.3 Chain Propagation

In this section, we describe *chain propagation*, which is a powerful technique that allows us to determine whether we can quickly color a given uncolored vertex v^* by using a *chain*, or sequence, of swaps and propagating the impact of the swaps in the chain until hopefully finding a 0-swap. This concept is inspired by *s-chain* tabu search [28], where chains up to a length s are explored, and by the consideration of a single *flat chain* in I-DSatur [19], where a chain of 1-swaps is applied within a single iteration whenever available. Another way to view chain propagation is as a lookahead for the actions that Partialcol would perform.

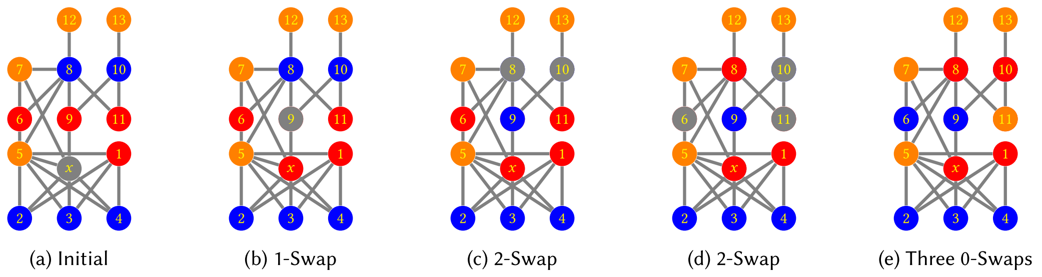


Fig. 3. Example of a chain propagation sequence coloring vertex x.

We start with the set of uncolored vertices $U = \{v^*\}$ and try to empty this set by applying the following rules until either $U = \emptyset$ or no rule is applicable. Whenever we find a chain of swaps that empties U, we have found a chain that successfully colors v^*. Figure 3 illustrates these rules using our running example.

Rule 1 (0-swap). *Take a vertex $w \in U$ and a color $\ell \in [k]$ such that $N_\ell(w) = \emptyset$. Swap the color of w to ℓ and remove w from U.*

The immediate goal of local search is finding a 0-swap, as a 0-swap decreases the number of uncolored vertices. The problem with 0-swaps is that they only consider the immediate neighborhood of the vertex.

Therefore, local search may miss possible 0-swaps if they are not included in our local instance or hidden behind larger swaps. We remedy this issue by extending chain propagation beyond 0-swaps and exploring all chains of limited complexity with the goal of coloring v^*.

A slightly more elaborate case prevails when we apply the following rule multiple times, keeping the number of uncolored vertices constant, completed by a final application of Rule 1:

Rule 2 (1-swap). *Take a vertex $w \in U$, such that for a color $\ell \in [k]$ and a vertex u, we have $N_\ell(w) = \{u\}$. Swap the color of w to ℓ, make u uncolored, and replace w with u in set U.*

We call such a sequence of rule applications a *1-swap chain*, sometimes called a flat chain [19].

Even more powerful but also more costly is a *p-swap chain*, where $p > 1$ is a fixed constant. It uses the following generalization of Rule 2:

Rule 3 (p-swap). *Take a vertex $w \in X$, such that for a color $\ell \in [k]$, we have $|N_\ell(v)| \leq p$. Swap the color of w to ℓ, make all the vertices in $N_\ell(w)$ uncolored, and replace w with $N_\ell(w)$ in X.*

Chain propagation explores the possible chains exhaustively. Bookkeeping is necessary to avoid re-applying the same series of swaps, as this leads to cycles, and consequently, chain propagation may not terminate. Further, we apply the rules in order, as it is faster to explore chains without p-swaps.

Two hyperparameters regulate the complexity of the chains. Since Rule 3 increases the number of uncolored vertices, it is the main factor for the complexity of the chains explored and, therefore, the main factor on the runtime of chain propagation. We limit the applications of the rule in two ways: (i) We limit p and thereby how much the number of uncolored vertices can increase within one rule application, and (ii) we limit how often Rule 3 can be applied within one chain for the same reason. Together, the two hyperparameters regulate how much the number of uncolored vertices can increase within a single chain.

3.4 Putting It All Together

Algorithm 1 combines the main ingredients of GC-SLIM that we have discussed. Initially, we compute a k-coloring c with a heuristic like DSatur and then repeatedly call GC-SLIM with different colors as the elimination goal. Each call either succeeds, reducing the number of colors, or fails, in which case, we restore c to the state it had before a color was removed. For each call, we pick the least prevalent color we have not yet tried as the elimination goal, breaking ties arbitrarily.

ALGORITHM 1: GC-SLIM

```
1:  iteration ← 1
2:  Remove color $e$ from $c$.
3:  **while** $c_0 \neq \emptyset$ and iteration < iteration_limit **do**
4:      Update tabu list.
5:      Pick vertex $v^* \in c_0$ to color.
6:      **if** *chain propagation* for $v^*$ is not successful **then**
7:          $m \leftarrow \min_{\ell \in [k]} |N_\ell(v)|$
8:          $\mathcal{I} \leftarrow$ CONSTRUCT_LOCAL_INSTANCE($v^*$)
9:          Changes ← CALL_SAT($\mathcal{I}$, $m$, sat_timeout)
10:         **if** finding a list coloring $\mathcal{I}$ with at most $m$ uncolored vertices fails **then**
11:             Perform $m$-swap of $v^*$.
12:             Check if budget should decrease.
13:         **else**
14:             Check if budget should increase.
15:         **end if**
16:     **end if**
17: **end while**
18: **if** $c_0 = \emptyset$ **then**
19:     **return** $c$
20: **else**
21:     Restore $c$.
22:     **return** Failed
23: **end if**
```

In Algorithm 1, GC-SLIM starts with adjusting c according to the given elimination goal and then tries to complete c for a prescribed number of iterations. In each iteration, it picks a vertex v^* and first tries to color it using chain propagation. If this fails, then the algorithm creates a local instance based on v^* and tries to color it using the SAT encoding. The number of uncolored vertices in the solution to the local instance is limited to m, where m is the prevalence of the least prevalent color in the neighborhood of v^*. This limit of m ensures that GC-SLIM will not perform worse than a p-swap. If the local instance is (partially) colored successfully, then GC-SLIM proceeds to the next vertex; otherwise, it defaults to a p-swap as Partialcol would perform.

The algorithm contains several hyperparameters, which we will discuss next.

3.5 Hyperparameters

The hyperparameters controlling Algorithm 1 are the *iteration limit*, the *timeout for the SAT solver*, and the *choice of SAT solver*.

The iteration limit controls how much time the algorithm spends on eliminating a single color. Lower iteration limits cause shorter runtimes. Therefore, one can try to eliminate more colors in the same amount of time at the price of possibly missing some improvements: Sometimes GC-SLIM will run many iterations with very few uncolored vertices until eventually finding the 0-swaps that complete the coloring. A low iteration limit will miss these improvements. Omitted in the listing is a mechanism that grants GC-SLIM an extra 10% of the iteration limit whenever the number of uncolored vertices decreases. Thus, GC-SLIM runs as long as it reduces the number of uncolored vertices, no matter the iteration limit.

The timeout for the SAT solver follows a similar tradeoff. Lower values lead to quicker search space exploration by trying many different local instances, while larger values may discover new improvements. While the iteration limit regulates how often GC-SLIM generates a local instance, the timeout for the SAT solver strongly influences the budget for the local instances.

The SAT solver can also impact the performance of GC-SLIM, both in terms of memory usage and speed, and different solvers may perform very differently for different instances.

The hyperparameters from this section, together with the branching factor, budget, and p-limit for chain propagation as discussed above, control GC-SLIM. As we will discuss next, some further options can severely impact GC-SLIM's performance. We will further explore this impact in our experiments.

3.6 Further Options

In this section, we discuss several minor options that can affect GC-SLIM's efficiency positively or negatively, depending on the instance. We will explore their effects further in the next section.

Prerun Tabu Search. Partialcol iterations are much faster than GC-SLIM iterations and can often reduce the number of colors quicker, while GC-SLIM can find improvements that Partialcol misses. Running Partialcol for several iterations before starting GC-SLIM tries to take the best from both worlds.

Flexible Vertices. We say that a vertex $v \in V(G)$ is *flexible* with respect to a (partial) k-coloring c if $v \in V(c)$ and there is a color $\ell \in [k] \setminus \{c(v)\}$ such that $N_\ell(v) = \emptyset$. Thus, we can change the color of a flexible vertex and still have a (partial) k-coloring. We let $F_c \subseteq V(G)$ be the set of all vertices that are flexible w.r.t. c. Flexible vertices provide an additional option when choosing a color: Instead of simply choosing the least prevalent color, we redefine the prevalence of a color ℓ as $|c_\ell \setminus F_c|$ and the prevalence w.r.t. the neighborhood of a vertex analogously. This can lead to a more accurate estimation, since flexible vertices allow immediate 0-swaps. This calculation is heuristical, as adjacent flexible vertices can block each other's color options, so we actually can only change the color of one of them.

Parallelization. GC-SLIM can run in parallel with minimal synchronization: Each thread runs GC-SLIM, and whenever one thread successfully eliminates a color, GC-SLIM is restarted in each thread with the improved coloring. This introduces the new hyperparameter *thread count*. Generally, more threads are better, as they enable faster search space exploration. Threads can either try to eliminate different colors, the same colors with different hyperparameter settings, or a mixture of both.

4 EXPERIMENTS

The aim of this article is not to determine the fastest graph coloring method but to investigate how SAT/CP methods can be utilized for the coloring of large, dense graphs.

We conduct three sets of experiments, one that evaluates the impact of the different hyperparameter settings and, by extension, the different features of GC-SLIM. The second experiment compares GC-SLIM to the state-of-the-art graph coloring methods FastColor and I-DSatur.[3] In the last experiment, we look at GC-SLIM's performance on the whole set of CG:SHOP instances.

Setup. We ran our experiments on a cluster where each server had two Intel Xeon E5-2640 v4 CPUs with 10 cores to 2.4 GHz for the first and second experiment, and two AMD EPYC 7402 CPUs, each with 24 cores running at 2.8 GHz, for the last experiment. The servers ran on Ubuntu 18.04 and used gcc 7.5.0. The runs were limited to 64 GB of memory. We implemented our approach in C++ and used Glucose 3[4] [1] and Cadical 1.5.0[5] [3] as SAT solvers.

Our implementation of DSatur [5] computes the initial colorings. We compare GC-SLIM against FastColor[6] [25] and I-DSatur[7] [19], representing state-of-the-art methods for coloring massive graphs. FastColor and I-DSatur runs were limited to 128 GB, as lower memory limits were insufficient for large and dense graphs.

We used an initial budget of 300 for the local instance. Whenever three consecutive SAT solver calls times out, we decrease the budget by 60 vertices; whenever three consecutive calls succeeds, we increase the budget by 60. In practice, the budget varied between 60 for very dense graphs and over 2,000 for sparse graphs.

Instances. We used four sets of instances: (i) instances from the CG:SHOP 2022 competition (CG),[8] (ii) large random graphs (Random), (iii) large graphs from the Snap repository (Snap)[9] [22, 23, 38], and (iv) hard DIMACS instances (DIMACS).[10] An overview of the number of vertices, edges, and densities is given in Table 2.

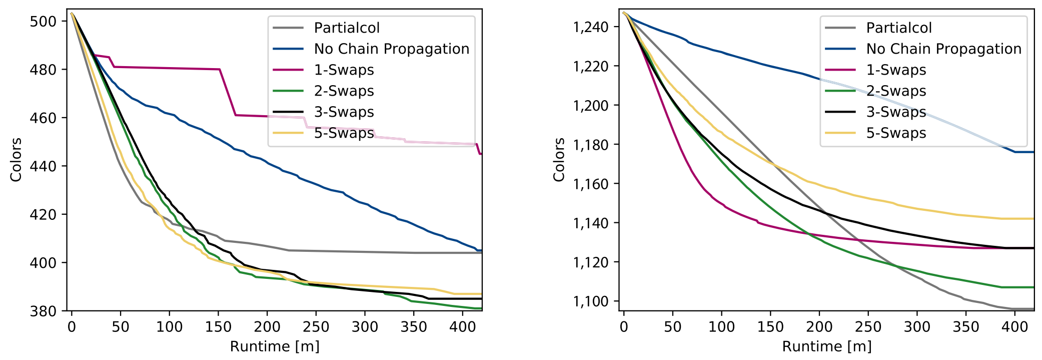
The CG:SHOP competition instances are very dense and have more structure than random graphs. The instances have up to 73,000 vertices and 1.5 billion edges. We picked 10 instances of the 225 used in the competition for our second experiment: Five instances are from the largest instances in the set with over 73,000 vertices; the other five were chosen such that density and size vary.

We used random graphs, as it was hard to find large benchmark instances that were not also very sparse. Therefore, we generated 14 Erdos-Renyi random graphs, which vary in size between 10,000 and 100,000 nodes and in density between 0.05 and 0.5.

The instances from the Snap repository and 10th DIMACS instances contain large graphs with up to several million vertices and have been used in related work [19, 25]. We preprocessed the instances by removing all vertices with degrees smaller than the lower bound on the chromatic number, determined in related work [19]. We picked the 11 instances with more than 10,000 and fewer than 280,000 vertices from these preprocessed instances. The 280,000 limit was due to memory constraints, since we focused on supporting dense graphs, and adjacency matrices become very memory-intensive for larger graphs. The mentioned sizes refer to preprocessed instances.

The DIMACS instances have been used in many papers for graph coloring and are included for reference, as GC-SLIM was not designed for such small instances. We used 10 instances that are considered hard [17].

[3]Code is available at https://github.com/ASchidler/coloring/ and https://doi.org/10.5281/zenodo.7947477; and results at https://doi.org/10.5281/zenodo.7787294.
[4]https://www.labri.fr/perso/lsimon/glucose/.
[5]http://fmv.jku.at/cadical/.
[6]https://lcs.ios.ac.cn/~caisw/Color.html.
[7]https://bitbucket.org/gkatsi/gc-cdcl/.
[8]https://cgshop.ibr.cs.tu-bs.de/competition/cg-shop-2022/.
[9]https://snap.stanford.edu/snap/.
[10]https://www.cc.gatech.edu/dimacs10/.



(a) CG instance vispecn74166 (b) Random instance rnd_100000_10

Fig. 4. Comparison of how many colors are eliminated over time with different swap limits for chain tracing. Time is in minutes on the x-axis and the number of colors is on the y-axis.

4.1 Hyperparameter Impact

We explore how the different hyperparameter settings change the results in the first set of experiments. We use a base configuration and vary the setting of one parameter at a time. Each run for each instance was limited to seven hours.

As the base configuration, we use a SAT solver timeout of 10 seconds, 300 iterations, no chain propagation, a branching factor for local instances of 3, no multithreading, and no prerun tabu search. We count the instances for which a hyperparameter value finds the best coloring. We also count the instances where it does so uniquely, i.e., no other setting for this hyperparameter found an equally good or better coloring.

SAT Solver Timeout. We try different values for the SAT solver timeout: 5, 10 (default), 30, and 60 seconds. A timeout of 5 seconds finds the best result for 37 of the 44 instances, where it reaches the unique best result on 15 of the instances. The results quickly deteriorate with higher timeouts: While a timeout of 10 seconds finds the unique best result for 4 instances, the higher timeouts achieve the same for only one instance.

A closer look at the number of SAT calls and the size of the local instances explains the results. While local instances contain 1,511 vertices on average with a 5 seconds timeout, they only increase to 1,610 vertices on average with a 60 seconds timeout. This is in stark contrast to the number of SAT calls, which decreases from 13,910 to 6,764. Depending on the instance, 25% to 50% of the SAT calls eventually time out, leading to a significant decrease in the number of possible SAT calls with higher timeouts. Therefore, higher timeouts should be reserved for later stages when lower values fail to find improvements.

Chain Propagation. We use different limits for the maximum size of the swaps in the chains we propagate. We use values of 0 (default), 1, 2, 3, and 5. A limit of 2 achieves the overall best result, reaching the best result on 34 of the 44 instances and the unique best on 18 instances. Limiting the chains to 1-swaps or using no chain propagation at all performs very poorly. Higher limits can be beneficial for some instances, as, for example, a limit of 5 performs slightly better on the Snap instances. This indicates that higher limits might be beneficial for large and very sparse graphs.

Figure 4 shows how chain propagation impacts GC-SLIM for large, dense instances. We can see that the number of improvements over time speeds up significantly and becomes comparable to Partialcol in terms of speed, sometimes surpassing it.

Impact of the SAT solver. We consider Glucose and Cadical as SAT solvers due to good results and their capability for incremental solving. Overall, Cadical achieves the unique best result on 19 of the 44 instances and Glucose on 10. This makes Cadical the best default choice, while Glucose may be better suited for individual instances. Glucose generally performs better on random instances and worse on the other instances in our experiments.

Flexible Vertices. Using flexible vertices does not give a clear advantage in the number of best results. Considering flexible vertices achieves the unique best result on 17 of the instances and not considering them on 14. While this does not seem like a clear advantage, the reduction in colors is significant, up to 100 colors for the instances where it performs better. For instances where it performs worse, the increase in the number of colors is never worse than 6. Considering flexible vertices performs consistently better for the CG instances.

Prerun tabu search. The benefits of running tabu search prior to GC-SLIM are very instance-specific. It achieves consistently better results on the DIMACS and Snap instances and worse results on the random and CG instances. This indicates that this configuration is beneficial for small and sparse graphs. One possible explanation is that the tabu search iterations become slower for dense graphs, reducing the efficiency gain over GC-SLIM.

Conflict Score. Another option is using a conflict score to determine swaps and selecting local instances instead of simply picking the smallest colored neighborhood. This achieves the unique best result for 16 instances, while not using this option gives the unique best result on 15, making its benefits very instance-specific. Using prerun tabu search with the conflict score gives the unique best result on 19 instances, in contrast to 17 instances, where the basic configuration finds the unique best result. It performs consistently bad for the random instances, mixed for the DIMACS instances, and consistently good on the CG and snap instances.

Iteration Limit. We try iteration limits of 100, 300 (default), 500, 1,000, and 5,000. None of the limits performs significantly better than the others, with 500 and higher performing better and 5,000 performing overall best with 7 uniquely best results. While 1,000 iterations is a good default setting, different settings may perform better for different instances. Furthermore, higher iteration limits may be necessary if the number of colors is close to optimal and it becomes harder to eliminate a color.

Branching. We try branching factors of 2, 3 (default), 5, 10, and 15. The overall best is a branching factor of 2, which achieves the best result on 31 of the 44 instances and finds 13 uniquely best instances. The results worsen with higher branching factors, except for the Snap instances, where a branching limit of 10 performs best. This matches the results for chain propagation, where higher limits also perform better for the Snap instances, suggesting that a search focused on breadth over depth may be a generally good strategy for sparse instances.

Initial Node Limit. When creating the local instance, we treat the initial vertex as a special case: Instead of limiting the number of colors we choose, we limit the number of neighbors we add for the initial vertex. This system chooses more different colors if there are many low-prevalence colors and fewer if not. We try limits of 10, 25, 50 (default), 75, and 100.

Each limit leads to the best result on about 16 to 21 instances and a unique best result on 4 to 5 instances. Therefore, there is no discernible good default and choosing the right value always depends on the instance at hand. A pattern similar to chain propagation and the branching limit emerges here: The lower the density, the better a higher initial limit, i.e., more focus on breadth, works.

Table 1. Best Hyperparameter Settings for Different Instance Sets

Instance Set	TO	Chain	Flex	Iterations	Prerun TS	Conflict	Branching	Initial	Solver
CG	5	2	N	1,000	Y	Y	2	75	Cadical
DIMACS	5	2	N	1,000	Y	N	3	10	Cadical
Random	10	2	Y	1,000	N	N	3	50	Glucose
Snap	5	5	Y	5,000	Y	Y	10	100	Cadical

4.2 Comparison and Parallelism

We use the results from the first experiment to discern a *best configuration* for each of the four sets of instances; the configurations are shown in Table 1. In our comparison, we use different GC-SLIM configurations, FastColor, I-DSatur, and Partialcol. Each was run with a 24-hour time limit. We additionally run other methods and use their output as GC-SLIM's input, i.e., we *prerun* our own implementations of **I-DSatur (IS)** and/or **Partialcol (PC)**. Partialcol runs either 7 hours alone or 4 hours together with I-DSatur. We also apply multithreading with 4 threads and varying the configuration parameters using the values shown in Table 1. The multithreading runs for only 6 hours and has twice the memory. The results are in Table 2.

The results show how well GC-SLIM performs on large and dense graphs. On the CG and Random instances, it significantly outperforms FastColor and I-DSatur. Interestingly, for random instances, a Partialcol prerun performs consistently better than any other configuration. For the CG instances, using a portfolio of varied configurations, combining the configurations in Table 1, performs best, and the overall best configuration performs comparatively poorly.

While GC-SLIM also outperforms both algorithms on the small DIMACS instances, it does not come close to reaching the best-known value on almost all instances. This shows that specialized algorithms for these small instances work better.

FastColor and I-DSatur shine on the Snap instances where they benefit from the structure of sparse graphs, which GC-SLIM does not particularly exploit. Still, FastColor, which performs better than I-DSatur, finds the best solution for six of the instances, as does the varied configuration. Note that our goal was not to compare FastColor and I-DSatur, but to compare these approaches to GC-SLIM on various graphs.

The GC-SLIM configurations perform very differently. Figure 5 shows the progression over time for selected instances.

Overall, the results show that a varied configuration is better than a single, tuned configuration. This finding is strengthened by the fact that multithreading incurs an overhead, as each thread has to restart once a color has been eliminated.

4.3 CG:SHOP Instances

The competition used 225 instances in total, separated in different classes, where instances into each class are created by the same process but with different sizes and densities. The instances were created such that then-available graph coloring solvers failed to produce good results. This is supported by the fact that these solvers would have placed very low in the competition [10].

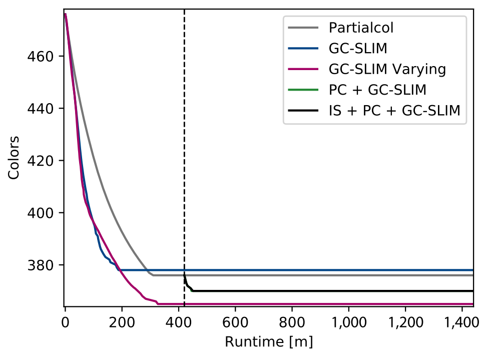
We started our submission by implementing Partialcol. This implementation was run for several days for each instance, and we only started implementing GC-SLIM, when Partialcol failed to find improvements. During that time, we varied and randomized every parameter and decision in our implementation. This gives us the possibility to compare GC-SLIM to these long Partialcol runs in Figure 6. The figure shows that GC-SLIM is able to significantly improve the colorings, even after the long Partialcol runs.

Table 2. Comparison of 24-hour Runs

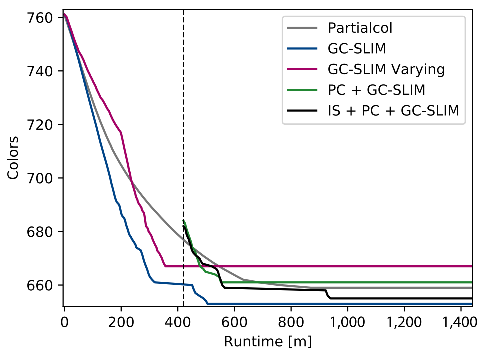
	Instance	Nodes	Edges	Density	DS	Best	Fast	IS	PC	G	GV	GP	GI	GPI
CG	reecn56737	56k	308m	0.19	316	258	314	307	270	266	**265**	**265**	271	268
	reecn73116	73k	610m	0.23	433	349	426	438	372	371	**362**	367	375	365
	rvisp20601	20k	19m	0.09	99	81	93	115	**85**	87	86	**85**	87	99
	sqrp49981	49k	621m	0.5	353	270	345	389	280	274	**272**	278	281	275
	sqrp73525	73k	1,560m	0.58	418	329	451	504	352	348	343	**341**	356	342
	sqrpecn71571	71k	1,272m	0.5	761	630	752	826	659	656	667	659	667	**655**
	visp40191	40k	53m	0.07	125	104	119	130	**110**	113	**110**	**110**	115	125
	visp70702	70k	192m	0.08	186	145	184	186	159	166	**158**	159	168	186
	vispecn26914	26k	30m	0.08	264	176	234	243	195	207	**190**	**190**	195	191
	vispecn74166	74k	205m	0.07	476	332	444	450	376	378	**365**	367	382	366
Random	rnd_100000_10	100k	499m	0.1	1247	–	1,256	1,247	1,089	1,091	1,112	**1,081**	1,096	1,097
	rnd_100000_5	100k	250m	0.05	665	–	676	663	580	577	584	**571**	581	573
	rnd_10000_10	10k	4,995k	0.1	169	–	171	170	145	144	145	**142**	145	**142**
	rnd_10000_25	10k	12m	0.25	399	–	398	396	333	337	340	**329**	339	330
	rnd_10000_5	10k	2,499k	0.05	93	–	94	93	80	79	93	**78**	80	**78**
	rnd_10000_50	10k	24m	0.5	844	–	837	840	703	715	722	**698**	715	699
	rnd_25000_10	25k	31m	0.1	371	–	377	371	319	317	321	**314**	321	315
	rnd_25000_25	25k	29m	0.1	247	–	234	234	246	**195**	197	196	196	196
	rnd_25000_5	25k	15m	0.05	202	–	204	200	173	172	173	**170**	173	**170**
	rnd_25000_50	25k	156m	0.5	1,896	–	1,877	1,899	1,613	1,630	1,646	**1,604**	1,636	1,605
	rnd_50000_10	50k	124m	0.1	679	–	687	677	585	586	592	**578**	590	579
	rnd_50000_5	50k	62m	0.05	363	–	371	364	314	314	316	**309**	316	310
	rnd_75000_10	75k	281m	0.1	968	–	975	968	839	842	852	**830**	847	834
	rnd_75000_5	75k	140m	0.05	517	–	525	517	449	447	452	**443**	450	444
Snap	G_n_pin_pout	99k	500k	0.0	6	5	6	6	6	**5**	**5**	6	**5**	6
	HR_edges	23k	328k	0.0	14	13	**13**	14	**13**	**13**	**13**	**13**	**13**	14
	WikiTalk	13k	728k	0.01	50	48	**48**	49	50	50	50	50	50	50
	artist_edges	18k	606k	0.0	23	19	**19**	21	20	20	20	20	20	23
	com-youtube	47k	670k	0.0	25	23	**23**	24	25	25	25	25	25	25
	gplus_combined	13k	6,766k	0.08	326	326	327	337	346	**326**	**326**	**326**	**326**	**326**
	kron_g500-simple-logn16	17k	1,495k	0.01	156	145	152	155	151	148	150	148	149	**147**
	p2p-Gnutella31	24k	100k	0.0	5	5	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**
	smallworld	100k	499k	0.0	8	6	7	8	**6**	**6**	**6**	**6**	**6**	8
	sx-stackoverflow	131k	10m	0.0	69	66	**66**	70	69	69	69	69	69	69
	wave	155k	1,057k	0.0	9	7	8	9	8	8	**7**	8	8	9
DIMACS	C2000.5	2,000	999k	0.5	208	145	205	207	168	170	173	**167**	170	**167**
	C2000.9	2,000	1,799k	0.9	555	408	534	547	429	425	441	429	**423**	428
	C4000.5	4,000	4,000k	0.5	377	259	376	376	312	313	317	**311**	312	**311**
	dsjc1000.1	1,000	49k	0.1	26	20	25	25	**22**	**22**	**22**	**22**	**22**	26
	dsjc1000.5	1,000	249k	0.5	116	83	112	114	93	93	97	**92**	93	**92**
	dsjc1000.9	1,000	449k	0.9	303	222	287	297	232	229	241	231	**228**	231
	flat1000_50_0	1,000	245k	0.49	114	50	111	112	**50**	69	88	**50**	**50**	114
	flat1000_60_0	1,000	245k	0.49	116	60	111	113	90	76	115	90	**60**	78
	flat1000_76_0	1,000	246k	0.49	114	81	111	112	92	93	94	92	**90**	114
	latin_square_10	900	307k	0.76	127	97	118	125	**102**	106	103	**102**	104	127
	r1000.1c	1,000	485k	0.97	102	98	103	103	99	100	**98**	99	99	102
	r1000.5	1,000	238k	0.48	242	234	**235**	240	245	236	**235**	239	236	236

DS shows the DSatur run used as input for GC-SLIM. *Best* shows the best known result for the instance from the literature. For comparison, we list *Fast* Color, Iterated DSatur (*IS*), and our Partialcol (*PC*) implementation. GC-SLIM configurations start with *G*, *V* denotes varying parameters, *I* indicates a I-DSatur Prerun, and *P* a Partialcol Prerun.

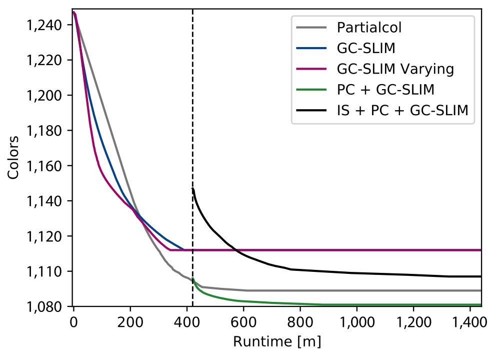
We also have results from a more controlled experiment, where GC-SLIM runs for 24 hours using the best configuration. Figure 7 shows that the final GC-SLIM implementation achieves almost the same results as the results from the long runs of Partialcol and GC-SLIM during development as described above. Our final implementation achieves these results in a fraction of the time. This



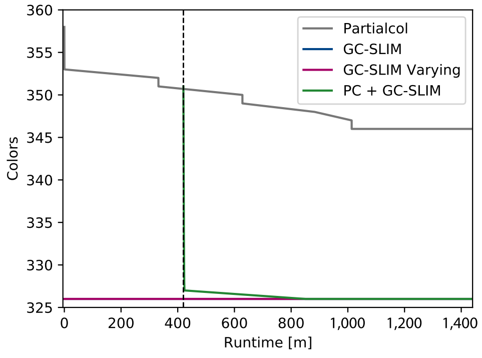
(a) CG instance vispecn74166



(b) CG instance sqrpecn71571



(c) Random instance rnd_100000_10



(d) Snap instance gplus_combined

Fig. 5. Comparison of how many colors are used by different configurations over time. The x-axis shows the time in minutes and the y-axis the number of colors. The vertical dotted line indicates when Partialcol preruns end and for multithreaded runs, the times are multiplied by the number of threads. PC refers to running Partialcol on the instance before GC-SLIM. IS refers to running I-DSatur to obtain the initial coloring.

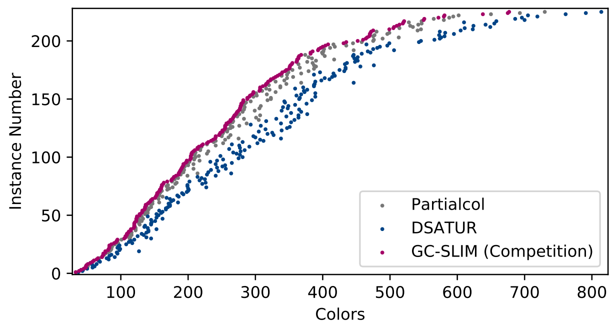


Fig. 6. The CG:SHOP 2022 instance results. For each instance, we show the initial solution using DSatur, the long-term improved coloring using Partialcol, and the eventually submitted solution obtained using GC-SLIM.

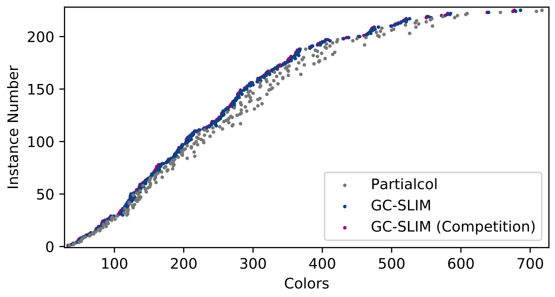


Fig. 7. The results from a 24-hour run of GC-SLIM compared to Partialcol and GC-SLIM as used in the competition and Figure 6.

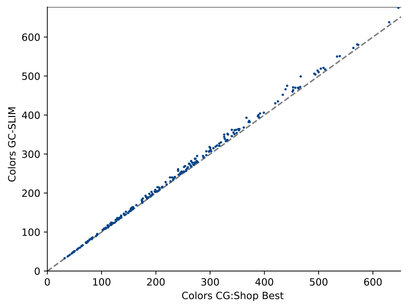


Fig. 8. Comparison between the best GC-SLIM and the best CG:SHOP 2022 colorings. Each mark is an instance, and the position indicates the number of colors used.

further shows how well GC-SLIM performs on these instances. Finally, Figure 8 shows the comparison between the best GC-SLIM run and the best results from the competition.

5 CONCLUSION

With GC-SLIM, we have presented a new hybrid approach to graph coloring that enhances tabu search with SAT-based local improvement. Key elements of this combination are the selection method for local instances, the SAT-based solution for local instances, and chain propagation. Further improvements are due to hyperparameter tuning, the prerunning of tabu search, and different metrics for selecting vertices for color elimination. We also proposed and tested a parallel version of GC-SLIM. Our experimental evaluation shows that GC-SLIM complements existing methods and can find colorings with significantly fewer colors than other methods on large dense graphs.

For future work, we see two main paths. The first one is improving the selection of local instances, as we expect a better criterion than using the least prevalent color. The other path is adapting GC-SLIM to more general graphs. We have seen that FastColor excels even for large random graphs and can handle even larger graphs. GC-SLIM can be adapted to handle large and sparse graphs. This would also require implementing features from FastColor and I-DSatur that exploit the structural properties of sparse graphs, as well as preprocessing. Integrating these features may also lead to a better method for local instance selection.

REFERENCES

[1] Gilles Audemard and Laurent Simon. 2009. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11–17, 2009*, Craig Boutilier (Ed.). 399–404. Retrieved from http://ijcai.org/Proceedings/09/Papers/074.pdf.

[2] Olivier Bailleux and Yacine Boufkhad. 2003. Efficient CNF encoding of Boolean cardinality constraints. In *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29–October 3, 2003, Proceedings (Lecture Notes in Computer Science)*, Francesca Rossi (Ed.), Vol. 2833. Springer, 108–122. DOI: https://doi.org/10.1007/978-3-540-45193-8_8

[3] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. 2020. CaDiCaL, KISSAT, PARACOOBA, PLINGELING and TREENGELING entering the SAT competition 2020. In *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions (Department of Computer Science Report Series B)*, Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda (Eds.), Vol. B-2020-1. University of Helsinki, 51–53.

[4] Ivo Blöchliger and Nicolas Zufferey. 2008. A graph coloring heuristic using partial solutions and a reactive tabu scheme. *Comput. Oper. Res.* 35, 3 (2008), 960–975. DOI: https://doi.org/10.1016/j.cor.2006.05.014

[5] Daniel Brélaz. 1979. New methods to color the vertices of a graph. *Commun. ACM* 22, 4 (Apr. 1979), 251–256. Retrieved fromhttps://dl.acm.org/doi/10.1145/359094.359101

[6] Edmund K. Burke, Jakub Marecek, Andrew J. Parkes, and Hana Rudová. 2010. A supernodal formulation of vertex colouring with applications in course timetabling. *Ann. Oper. Res.* 179, 1 (2010), 105–130. DOI: https://doi.org/10.1007/s10479-010-0716-z

[7] Stephen A. Cook. 1971. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3–5, 1971, Shaker Heights, Ohio, USA*, Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman (Eds.). ACM, 151–158. DOI: https://doi.org/10.1145/800157.805047

[8] Loïc Crombez, Guilherme Dias da Fonseca, Florian Fontan, Yan Gerard, Aldo Gonzalez-Lorenzo, Pascal Lafourcade, Luc Libralesso, Benjamin Momège, Jack Spalding-Jamieson, Brandon Zhang, and Da Wei Zheng. 2023. Conflict optimization for binary CSP applied to minimum partition into plane subgraphs and graph coloring. *J. Experim. Algor.* 28 (2023). This issue.

[9] Loïc Crombez, Guilherme Dias da Fonseca, Yan Gerard, and Aldo Gonzalez-Lorenzo. 2022. Shadoks approach to minimum partition into plane subgraphs (CG challenge). In *38th International Symposium on Computational Geometry, SoCG 2022, June 7–10, 2022, Berlin, Germany (LIPIcs)*, Xavier Goaoc and Michael Kerber (Eds.), Vol. 224. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 71:1–71:8. DOI: https://doi.org/10.4230/LIPIcs.SoCG.2022.71

[10] Sándor P. Fekete, Phillip Keldenich, Dominik Krupke, and Stefan Schirra. 2023. Minimum partition into plane subgraphs: The CG:SHOP challenge 2022. *J. Experim. Algor.* 28 (2023). This issue.

[11] Michael R. Fellows, Fedor V. Fomin, Daniel Lokshtanov, Frances A. Rosamond, Saket Saurabh, Stefan Szeider, and Carsten Thomassen. 2011. On the complexity of some colorful problems parameterized by treewidth. *Inf. Comput.* 209, 2 (2011), 143–153. DOI: https://doi.org/10.1016/j.ic.2010.11.026

[12] Johannes K. Fichte, Markus Hecher, Daniel Le Berre, and Stefan Szeider. 2023. The silent (r)evolution of SAT. *Commun. ACM* 66, 6 (2023), 64–72. https://doi.org/10.1145/3560469

[13] Johannes Klaus Fichte, Neha Lodha, and Stefan Szeider. 2017. SAT-based local improvement for finding tree decompositions of small width. In *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28–September 1, 2017, Proceedings (Lecture Notes in Computer Science)*, Serge Gaspers and Toby Walsh (Eds.), Vol. 10491. Springer, 401–411. DOI: https://doi.org/10.1007/978-3-319-66263-3_25

[14] Florian Fontan, Pascal Lafourcade, Luc Libralesso, and Benjamin Momège. 2022. Local search with weighting schemes for the CG: SHOP 2022 competition (CG challenge). In *38th International Symposium on Computational Geometry, SoCG 2022, June 7–10, 2022, Berlin, Germany (LIPIcs)*, Xavier Goaoc and Michael Kerber (Eds.), Vol. 224. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 73:1–73:6. DOI: https://doi.org/10.4230/LIPIcs.SoCG.2022.73

[15] Robert Ganian, Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. 2019. SAT-encodings for treecut width and treedepth. In *Proceedings of ALENEX 2019, the 21st Workshop on Algorithm Engineering and Experiments*, Stephen G. Kobourov and Henning Meyerhenke (Eds.). SIAM, 117–129. DOI: https://doi.org/10.1137/1.9781611975499.10

[16] Gael Glorian, Jean-Marie Lagniez, Valentin Montmirail, and Nicolas Szczepanski. 2019. An incremental SAT-based approach to the graph colouring problem. In *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30–October 4, 2019, Proceedings (Lecture Notes in Computer Science)*, Thomas Schiex and Simon de Givry (Eds.), Vol. 11802. Springer, 213–231. DOI: https://doi.org/10.1007/978-3-030-30048-7_13

[17] Jin-Kao Hao and Qinghua Wu. 2012. Improving the extraction and expansion method for large graph coloring. *Discret. Appl. Math.* 160, 16-17 (2012), 2397–2407. DOI: https://doi.org/10.1016/j.dam.2012.06.007

[18] Emmanuel Hebrard and George Katsirelos. 2019. Clause learning and new bounds for graph coloring. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10–16, 2019*, Sarit Kraus (Ed.). ijcai.org, 6166–6170. DOI: https://doi.org/10.24963/ijcai.2019/856

[19] Emmanuel Hebrard and George Katsirelos. 2019. A hybrid approach for exact coloring of massive graphs. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4–7, 2019, Proceedings (Lecture Notes in Computer Science)*, Louis-Martin Rousseau and Kostas Stergiou (Eds.), Vol. 11494. Springer, 374–390. DOI: https://doi.org/10.1007/978-3-030-19212-9_25

[20] Richard M. Karp. 1972. Reducibility among combinatorial problems. In *Proceedings of a Symposium on the Complexity of Computer Computations, Held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA (The IBM Research Symposia Series)*, Raymond E. Miller and James W. Thatcher (Eds.). Plenum Press, New York, 85–103. DOI: https://doi.org/10.1007/978-1-4684-2001-2_9

[21] Alexander S. Kulikov, Danila Pechenev, and Nikita Slezkin. 2022. SAT-based circuit local improvement. In *47th International Symposium on Mathematical Foundations of Computer Science, MFCS 2022, August 22–26, 2022, Vienna, Austria (LIPIcs)*, Stefan Szeider, Robert Ganian, and Alexandra Silva (Eds.), Vol. 241. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 67:1–67:15. DOI: https://doi.org/10.4230/LIPIcs.MFCS.2022.67

[22] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. Retrieved from http://snap.stanford.edu/data.

[23] Jure Leskovec and Rok Sosic. 2016. SNAP: A general-purpose network analysis and graph-mining library. *ACM Trans. Intell. Syst. Technol.* 8, 1 (2016), 1:1–1:20. DOI: https://doi.org/10.1145/2898361

[24] Leonid Levin. 1973. Universal sequential search problems. *Prob. Inf. Transmiss.* 9, 3 (1973), 265–266.

[25] Jinkun Lin, Shaowei Cai, Chuan Luo, and Kaile Su. 2017. A reduction based method for coloring very large graphs. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19–25, 2017*, Carles Sierra (Ed.). ijcai.org, 517–523. DOI: https://doi.org/10.24963/ijcai.2017/73

[26] Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. 2017. SAT-encodings for special treewidth and pathwidth. In *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28–September 1, 2017, Proceedings (Lecture Notes in Computer Science)*, Serge Gaspers and Toby Walsh (Eds.), Vol. 10491. Springer, 429–445. DOI: https://doi.org/10.1007/978-3-319-66263-3_27

[27] Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. 2019. A SAT approach to branchwidth. *ACM Trans. Comput. Log.* 20, 3 (2019), 15:1–15:24. DOI: https://doi.org/10.1145/3326159

[28] Craig A. Morgenstern. 1993. Distributed coloration neighborhood search. In *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11–13, 1993 (DIMACS Series in Discrete Mathematics and Theoretical Computer Science)*, David S. Johnson and Michael A. Trick (Eds.), Vol. 26. DIMACS/AMS, 335–357. DOI: https://doi.org/10.1090/dimacs/026/16

[29] David Pisinger and Stefan Ropke. 2010. Large neighborhood search. In *Handbook of Metaheuristics*. Springer Verlag, 399–419. DOI: https://doi.org/10.1007/978-1-4419-1665-5_13

[30] Vaidyanathan Peruvemba Ramaswamy and Stefan Szeider. 2021. Turbocharging treewidth-bounded Bayesian network structure learning. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, the Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2–9, 2021.* AAAI Press, 3895–3903.Retrieved from https://ojs.aaai.org/index.php/AAAI/article/view/16508.

[31] Franz-Xaver Reichl, Friedrich Slivovsky, and Stefan Szeider. 2023. Circuit minimization with QBF-based exact synthesis. In *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023.* AAAI Press. To appear.

[32] Ryan A. Rossi and Nesreen K. Ahmed. 2014. Coloring large complex networks. *Soc. Netw. Anal. Min.* 4, 1 (2014), 228. DOI: https://doi.org/10.1007/s13278-014-0228-y

[33] André Schidler and Stefan Szeider. 2021. SAT-based decision tree learning for large data sets. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, the Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2–9, 2021.* AAAI Press, 3904–3912. Retrieved from https://ojs.aaai.org/index.php/AAAI/article/view/16509.

[34] Jack Spalding-Jamieson, Brandon Zhang, and Da Wei Zheng. 2022. Conflict-based local search for minimum partition into plane subgraphs (CG challenge). In *38th International Symposium on Computational Geometry, SoCG 2022, June 7–10, 2022, Berlin, Germany (LIPIcs)*, Xavier Goaoc and Michael Kerber (Eds.), Vol. 224. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 72:1–72:6. DOI: https://doi.org/10.4230/LIPIcs.SoCG.2022.72

[35] Wen Sun. 2018. *Heuristic Algorithms for Graph Coloring Problems. (Algorithmes heuristiques pour des problèmes de coloration de graphes).* Ph.D. Dissertation. University of Angers, France. Retrieved from https://tel.archives-ouvertes.fr/tel-02136810.

[36] Moshe Y. Vardi. 2014. Boolean satisfiability: Theory and engineering. *Commun. ACM* 57, 3 (2014), 5. DOI: https://doi.org/10.1145/2578043

[37] Anurag Verma, Austin Buchanan, and Sergiy Butenko. 2015. Solving the Maximum Clique and Vertex Coloring Problems on very large sparse networks. *INFORMS J. Comput.* 27, 1 (2015), 164–177. DOI: https://doi.org/10.1287/ijoc.2014.0618

[38] Marinka Zitnik, Rok Sosič, Sagar Maheshwari, and Jure Leskovec. 2018. BioSNAP Datasets: Stanford Biomedical Network Dataset Collection. Retrieved from http://snap.stanford.edu/biodata.

Received 25 November 2022; revised 28 April 2023; accepted 13 May 2023