Conflict Optimization for Binary CSP Applied to Minimum Partition into Plane Subgraphs and Graph Coloring

Loïc Crombez[1], Guilherme D. da Fonseca[2], Florian Fontan[3], Yan Gerard[1], Aldo Gonzalez-Lorenzo[2], Pascal Lafourcade[1], Luc Libralesso[1], Benjamin Momège[3], Jack Spalding-Jamieso[4], Brandon Zhang[3], and Da Wei Zheng[5]

[1]LIMOS, Université Clermont Auvergne, France
[2]LIS, Aix-Marseille Université, France
[3]Independent Researcher
[4]David R. Cheriton School of Computer Science, University of Waterloo, Canada
[5]Department of Computer Science, University of Illinois at Urbana-Champaign, USA

Abstract

CG:SHOP is an annual geometric optimization challenge and the 2022 edition proposed the problem of coloring a certain geometric graph defined by line segments. Surprisingly, the top three teams used the same technique, called conflict optimization. This technique has been introduced in the 2021 edition of the challenge, to solve a coordinated motion planning problem. In this paper, we present the technique in the more general framework of binary constraint satisfaction problems (binary CSP). Then, the top three teams describe their different implementations of the same underlying strategy. We evaluate the performance of those implementations to vertex color not only geometric graphs, but also other types of graphs.

1 Introduction

The CG:SHOP challenge (Computational Geometry: Solving Hard Optimization Problems) is an annual geometric optimization competition, whose first edition took place in 2019. The 2022 edition proposed a problem called *minimum partition into plane subgraphs*. The input is a graph G embedded in the plane with edges drawn as straight line segments, and the goal is to partition the set of edges into a small number of plane graphs (Fig. 1) [6]. This goal can be formulated as a vertex coloring problem on a graph G' defined as follows. The vertices of G' are the segments defining the edges of G, and the edges of G' correspond to pairs of *crossing* segments (segments that intersect only at a common endpoint are not considered crossing).

The three top-ranking teams (Lasa, Gitastrophe, and Shadoks) on the CG:SHOP 2022 challenge all used a common approach called *conflict optimization* [7, 26, 3] while the fourth team used a SAT-Boosted Tabu Search [25]. Conflict optimization is a technique used by Shadoks to obtain the first place in the CG:SHOP 2021 challenge for low-makespan coordinated motion planning [4], and the main ideas of the technique lent themselves well to the 2022 challenge. Next, we describe the conflict optimizer as a metaheuristic to solve constraint satisfaction problems (CSP) [29]. We start by describing a CSP.

A CSP is a triple of

- *variables* $X = (x_1, \ldots, x_n)$,

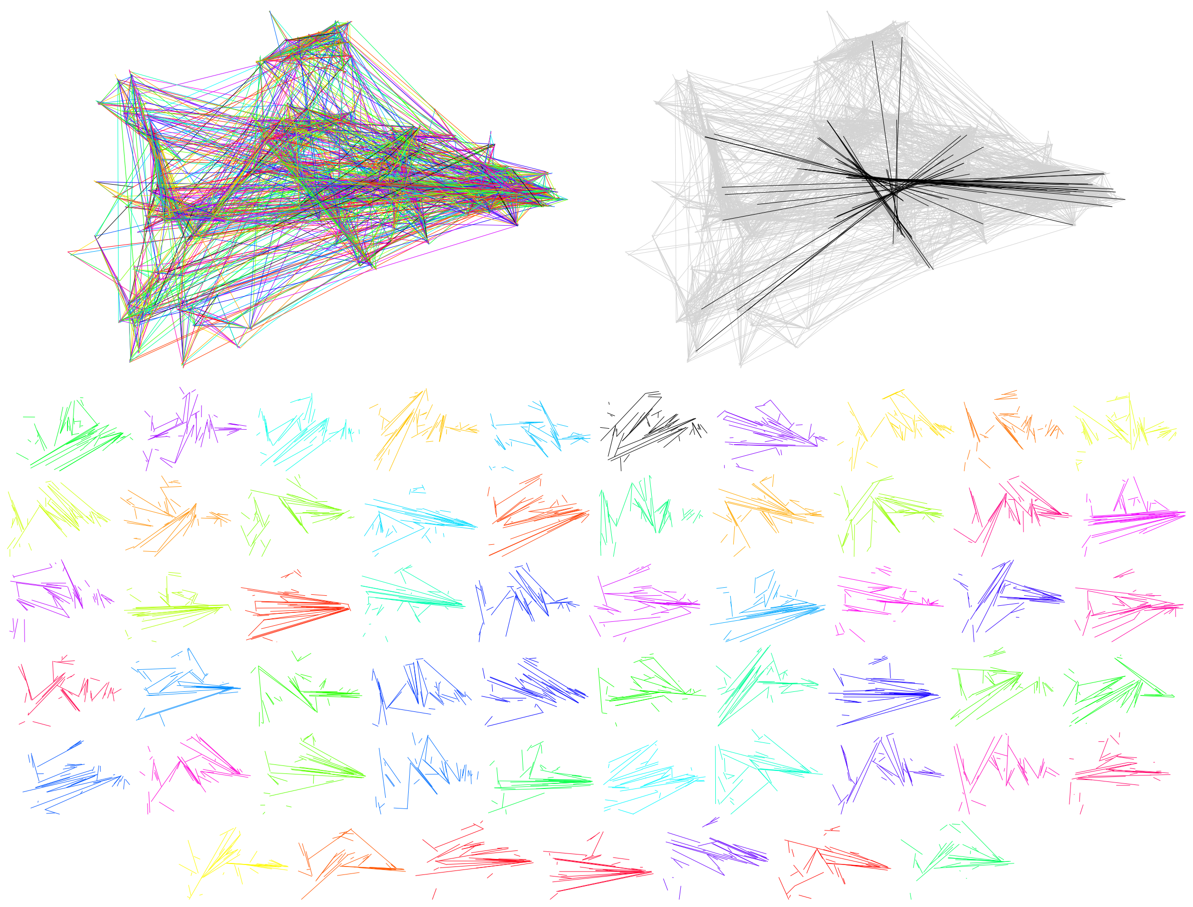
Figure 1: A partition of the input graph of the CG:SHOP2022 instance vispecn2518 into 57 plane graphs. It is the smallest instance of the challenge with 2518 segments. On top left, you see all 57 colors together. On top right, you see a clique of size 57, hence the solution is optimal. Each of the 57 colors is then presented in small figures.

- *domains* $\mathcal{D} = (D_1, \ldots, D_n)$, and
- *constraints* $\mathcal{R}$.

Each variable x_i must be assigned a *value* in the corresponding domain D_i such that all constraints are satisfied. In general, the constraints may forbid arbitrary subsets of values. We restrict our attention to a particular type of constraints (*binary CSP*), which only involve pairs of assignments. A *partial evaluation* is an assignment of a subset of the variables, called *evaluated*, with the remaining variables called *non-evaluated*. All constraints involving a non-evaluated variable are satisfied by default. We only consider assignments and partial assignments that satisfy all constraints.

The conflict optimizer iteratively modifies a partial evaluation with the goal of emptying the set S of non-evaluated variables, at which point it stops. At each step, a variable x_i is removed from S. If there exists a value $x \in D_i$ that satisfies all constraints, then we assign the value x to the variable x_i. Otherwise, we proceed as follows. For each possible value $x \in D_i$, we consider the set $K(i, x)$ of variables (other than x_i) that are part of constraints violated by the assignment $x_i = x$. We assign to x_i the value x that minimizes

$$\sum_{x_j \in K(i,x)} w(j),$$

where $w(j)$ is a weight function to be described later. The variables $x_j \in K(i,x)$ become non-evaluated and added to S.

The weight function should be such that $w(j)$ increases each time x_j is added to S, in order to avoid loops that keep moving the same variables back and forth from S. Let $q(j)$ be the number of times x_j became non-evaluated. A possible weight function is $w(j) = q(j)$. More generally, we can have $w(j) = q(j)^p$ for some exponent p (typically between 1 and 2). Of course, several details of the conflict optimizer are left open. For example, which element to choose from S, whether some random noise should be added to w, and the decision to restart the procedure from scratch after a certain time.

The CSP as is, does not apply to optimization problems. However, we can, impose a maximum value k of the objective function in order to obtain a CSP. The conflict optimizer was introduced in a low makespan coordinated motion planning setting. In that setting, the variables are the robots, the domains are their paths (of length at most k) and the constraints forbid collisions between two paths. In the graph coloring setting, the domains are the k colors of the vertices and the constraints forbid adjacent vertices from having the same color.

The conflict optimizer can be adapted to non-binary CSP, but in that case multiple variables may be unassigned for a single violated constraint. The strategy has some resemblance to the similarly named *min-conflicts algorithm* [21], but notable differences are that a partial evaluation is kept instead of an invalid evaluation and the weight function that changes over time.

While the conflict optimization strategy is simple, there are different ways to apply it to the graph coloring problem. The goal of the paper is to present how the top three teams applied it or complemented it with additional strategies. We compare the relative benefits of each variant on the instances given in the CG:SHOP 2022 challenge. We also compare them to baselines on some instances issued from graph coloring benchmarks.

The paper is organized as follows. Section 2 presents the details of the conflict optimization strategy applied to graph coloring. In the three sections that follow, the three teams Lasa, Gitastrophe, and Shadoks present the different parameters and modified strategies that they used to make the algorithm more efficient for the CG:SHOP 2022 challenge. The last section is devoted to the experimental results.

1.1 Literature Review

The study of graph coloring goes back to the 4-color problem (1852) and it has been intensively studied since the 1970s (see [14, 17] for surveys). Many heuristics have been proposed [10, 13, 19, 23], as well as exact algorithms [5, 12, 18]. We briefly present two classes of algorithms: greedy algorithms and exact algorithms.

Greedy algorithms. These algorithms are used to find good quality initial solutions in a short amount of time. The classic greedy heuristic considers the vertices in arbitrary order and colors each vertex with the smallest non-conflicting color. The two most famous modern greedy heuristics are *DSATUR* [2] and *Recursive Largest First* (*RLF*) [16]. At each step (until all vertices are colored), DSATUR selects the vertex v that has the largest number of different colors in its neighbourhood. Ties are broken by selecting a vertex with maximum degree. The vertex v is colored with the smallest non-conflicting color. *RLF* searches for a large independent set I, assigns the vertices I the same color, removes I from G', and repeats until all vertices are colored.

Exact algorithms. Some exact methods use a branch-and-bound strategy, for example extending the DSATUR heuristic by allowing it to backtrack [24, 8]. Another type of exact method (branch-and-cut-and-price) decomposes the vertex coloring problem into an iterative resolution of two sub-problems [20, 12, 9]. The "master problem" maintains a small set of valid colors using a set-covering formulation. The "pricing problem" finds a new valid coloring that is promising by solving a maximum weight independent set problem. Exact algorithms are usually able to find the optimal coloring for graphs with a few hundred vertices. However, even the smallest CG:SHOP 2022 competition instances involve at least a few thousands vertices.

2 Conflict Optimization for Graph Coloring

Henceforth, we will only refer to the intersection conflict graph G' induced by the instance. Vertices will refer to the vertices $V(G')$, and edges will refer to the edges $E(G')$. Our goal is to partition the vertices using a minimum set of k color classes $\mathcal{C} = \{C_1, \ldots, C_k\}$, where no two vertices in the same color class C_i are incident to a common edge.

2.1 Conflict Optimization

We consider the classical problem of coloring the vertices of a graph $G' = (V(G'), E(G'))$. We assume that an initial solution $\mathcal{C} = \{C_1, \ldots, C_k\}$ has been previously computed (the choice of the initial solution does not seem to impact the quality of the final solution produced by the conflict optimizer). The goal of the conflict optimizer is to reduce the number of colors of $\mathcal{C}$ by one. When (and if) the conflict optimizer terminates, it will give such a solution. However, after a certain amount of time or when a certain situation arrives, we may decide to abort the execution of the conflict optimizer without any solution, and perhaps try again.

Throughout the execution, we maintain a partial coloring, which is a valid coloring for a subset of the vertices. The complementary subset of uncolored vertices is called the *conflict set* and denoted S. The conflict optimizer proceeds as follows:

1. Pick a color class C_i to be eliminated. Uncolor all vertices in C_i and make $S \leftarrow C_i$. A valid vertex-coloring is maintained for the set $V(G') \setminus S$. If S is empty, we have a valid vertex coloring of G' which uses one fewer color.

2. Pick and remove an element v from S. For each color class, compute the *conflict score* with v. The conflict score of a color class C_j is

$$score(C_j) = f(C_j) \sum_{\substack{u \in C_j \\ (u,v) \in E(G')}} w(u) \tag{1}$$

   where the weight $w(u)$ is a variable depending on the the number of times that u has been removed from the conflict set S in previous iterations, and where $f(C_j)$ is a random variable adding randomness in the process.

3. Pick the color class C_j with the lowest conflict score. Uncolor all vertices in C_j which are adjacent to v and add those vertices to S. This step is slightly modified when the BDFS option detailed in the later is activated. In this case, the algorithm does not put in the conflict set S all the vertices in conflict with S. Some of them are recolored easily so that they do not enter in the conflict set S. Insert v into C_j.

4. Repeat steps 2 and 3 until the set S is empty.

The three teams provided different variants of the algorithm by playing with different options of the optimizer.

(a) The first option is the choice of the initial color C_i to be eliminated at the first step of the loop. It is random for Gitastrophe, and the smallest color class for Shadoks and Lasa variants.

(b) The second option is the way to choose the element v from S in step 2. Random for Gitastrophe, a fifo queue for Shadoks, and the element that provides the least total conflict score after its removal for Lasa.

(c) The third option is the choice of the weight function $w(\cdot)$ defined on the vertices. Different functions can be used, all depending on the parameter $q(u)$ that is defined as the number of times that a vertex u has been removed from S. Lasa uses $w(u) = 1 + q(u)$. Gitastrophe uses $w(u) = 1 + q(u)^2$. Shadoks uses $w(u) = 1 + q(u)^p$ with $p \in [1, 2]$. Shadoks also add a threshold $q_{\max}$ with $w(u) = \infty$ if $q(u) > q_{\max}$. Gitastrophe also has such a threshold, but instead uses it as a heuristic to abort the execution and start again.

(d) The fourth option is the choice of $f(C_i)$. Lasa and Gitastrophe simply set $f(C_i) = 1$, while Shadoks use a Gaussian random variable with average 1 for $f(C_i)$. The right amount of randomness, controlled by the variance σ, has a significant impact on the search time.

(e) The fifth option is that Shadoks add a Bounded Depth-First Search (BDFS) option which detects vertices that can be recolored easily. These vertices are recolored immediately, instead of entering S, and consequently does not suffer an increase in the value of $q(\cdot)$.

Some extra options are useful in order to drive the computation.

- Restart: The computation is restarted from step 2 if the size of the conflict set S becomes too large because the coloring of $V(G') \setminus S$ has deteriorated too much to come back to a valid coloring.
- Multistart: Shadoks also use a multistart option to restart from step 1 with a random eliminated color C_i and a color shuffle.

The different parameters, options and complementary strategies used by each team are described in the next three sections.

3 Lasa Team

3.1 Finding Initial Solutions

Lasa team used two approaches to find initial solutions:

1. **DSATUR** is the classical graph coloring algorithm presented in Section 1.
2. **Orientation greedy** is almost the only algorithm where the geometry of the segments is used. If segments are almost parallel, it is likely that they do not intersect (thus forming an independent set). This greedy algorithm first sorts the segments by orientation, ranging from $-\frac{\pi}{2}$ to $\frac{\pi}{2}$. For each segment in this order, the algorithm tries to color it using the first available color. If no color has been found, a new color is created for coloring the considered segment. This algorithm is efficient, produces interesting initial solutions and takes into account the specificities of the competition.

3.2 Conflict Optimization

TABUCOL inspired neighbourhood One classical approach for the vertex coloring involves allowing solutions with conflicting vertices (two adjacent vertices with the same color). It was introduced in 1987 [13] and called TABUCOL. It starts with an initial solution, removes a color (usually the one with the least number of vertices), and assigns uncolored vertices with a new color among the remaining ones. This is likely to lead to some conflicts (*i.e.* two adjacent vertices sharing a same color). The local search scheme selects a conflicting vertex, and tries to swap its color, choosing the new coloring that minimises the number of conflicts. If it reaches a state with no conflict, it provides a solution with one color less than the initial solution. The process is repeated until the stopping criterion is met. While the original TABUCOL algorithm includes a "tabu-list" mechanism to avoid cycling, it is not always sufficient, and requires some hyper-parameter tuning in order to obtain a good performance on a large variety of instances. To overcome this issue, we use a neighbourhood, but replace the "tabu-list" by the conflict optimizer scheme presented above.

PARTIALCOL inspired neighbourhood PARTIALCOL another local search algorithm solving the vertex coloring problem was introduced in 2008. This algorithm proposes a new local search scheme that allows partial coloring (thus allowing uncolored vertices). The goal is to minimize the number of uncolored vertices. Similarly to TABUCOL, PARTIALCOL starts with an initial solution, removes one color (unassigning its vertices), and performs local search iterations until no vertex is left uncolored. When coloring a vertex, the adjacent conflicting vertices are uncolored. Then, the algorithm repeats the process until all vertices are colored, or the stopping criterion is met. This neighbourhood was also introduced alongside a tabu-search procedure. The tabu-search scheme is also replaced by a conflict-optimization scheme. Note that this neighbourhood was predominantly used by the other teams.

4 Gitastrophe

4.1 Solution Initialization

The gitastrophe team uses the traditional greedy algorithm of Welsh and Powell [30] to obtain initial solutions: order the vertices in decreasing order of degree, and assign each vertex the minimum-label color not used by its neighbors. During the challenge Gitastrophe attempted to use different orderings for the greedy algorithm, such as sorting by the slope of the line segment associated with each vertex (as the orientation greedy initialization presented in Section 3), and also tried numerous other strategies. Ultimately, after running the solution optimizer for approximately the same amount of time, all initializations resulted in an equal number of colors.

4.2 Modifications to the Conflict Optimizer

Taking inspiration from memetic algorithms, which alternate between an intensification and a diversification stage, the algorithm continually switched between a phase using the above conflict score, and one minimizing only the number of conflicts. Thus during the conflict-minimization phase, the random variables $f(C_j)$ and $w(u)$ are both fixed equal to 1 leading to a conflict score

$$score(C_j) = \sum_{u \in C_j, (u,v) \in E(G')} 1.$$

Each phase lasted for 10^5 iterations. Adding the conflict-minimization phase gave minor improvements to some of the challenge instances.

5 Shadoks

In this section, we describe the choices used by the Shadoks team for the options described in Section 2.1.

Option (a) The Shadoks generally chose to eliminate the color with the smallest number of elements. However, if the multistart option is toggled on, then a random color is used each time.

Option (b) The conflict set S is stored in a queue. The Shadoks tried other strategies, but found that the queue gives the best results.

Option (c) The weight function used is $w(u) = 1 + q(u)^p$, mostly with $p = 1.2$. The effect of the parameter p is shown in Fig. 2. Notice that in all figures, the number of colors shown is the average of ten executions of the code using different random seeds.

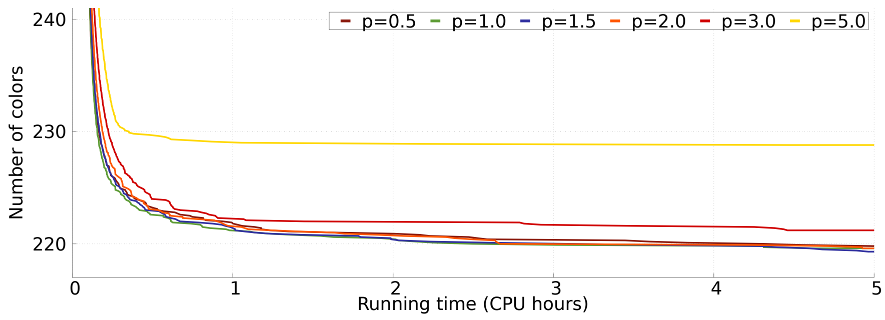

Figure 2: Number of colors over time for the instance `vispecn13806` using different values p. The algorithm uses $\sigma = 0.15$, easy vertices, $q_{\max} = 59022$, but does not use the BDFS nor any clique.

If $q(u)$ is larger than a threshold $q_{\max}$, the Shadoks set $w(u) = \infty$ so that the vertex u never reenters S. If at some point an uncolored vertex v is adjacent to some vertex u of infinite weight in every color class, then the conflict optimizer is restarted. When restarting, the initial coloring is shuffled by moving some vertices from their initial color class to a new one.

Looking at Fig. 3, the value of $q_{\max}$ does not seem to have much influence as long as it is not too small. Throughout the challenge the Shadoks almost exclusively used $q_{\max} = 2000 \cdot (75000/m)^2$, where m is the number of vertices. This value roughly ensures a restart every few hours.

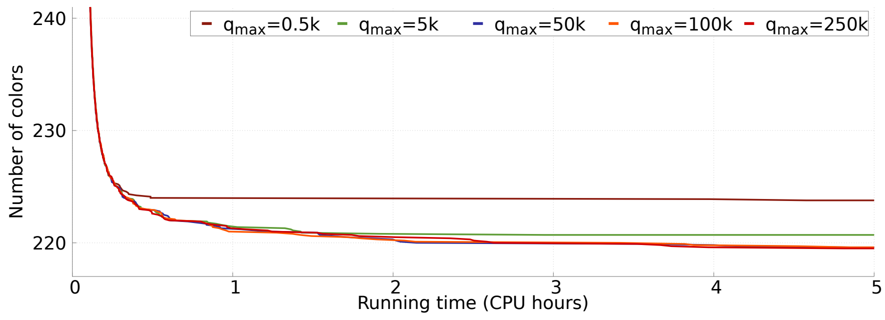

Figure 3: Number of colors over time with different values of $q_{\max}$ obtained on the instance `vispecn13806`. Parameters are $\sigma = 0.15$, $p = 1.2$, no clique knowledge, and no BDFS.

If the clique option is toggled on, each vertex u in the largest known clique has $w(u) = \infty$. The impact of the clique option on the computation is shown in Fig. 4. The idea is that since each vertex of the clique must have a different color, it is useless to change their color. The algorithm works by recoloring the other vertices. During the challenge, the Shadoks used several methods to produce large cliques, including simulated annealing and mixed integer programming.

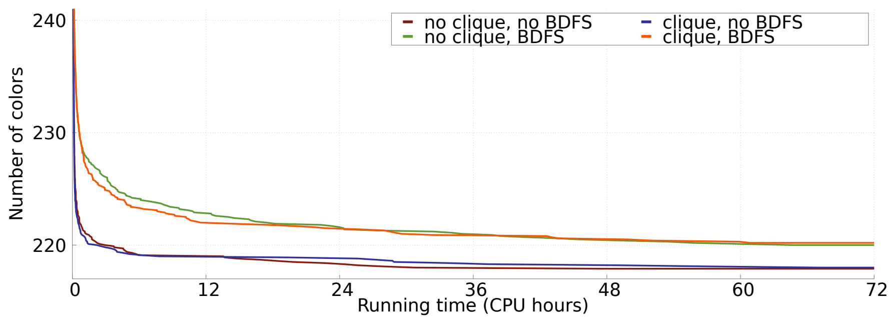

Figure 4: Number of colors over time with and without clique knowledge and BDFS obtained on the instance `vispecn13806`. Parameters are $\sigma = 0.15$, $p = 1.2$, and $q_{\max} = 1500000$.

Option (d) The Shadoks use the function f as a Gaussian random variable of mean 1 and variance σ. A good default value is $\sigma = 0.15$. The effect of the variance is shown in Fig. 5. Notice that setting $\sigma = 0$ gives much worse results.

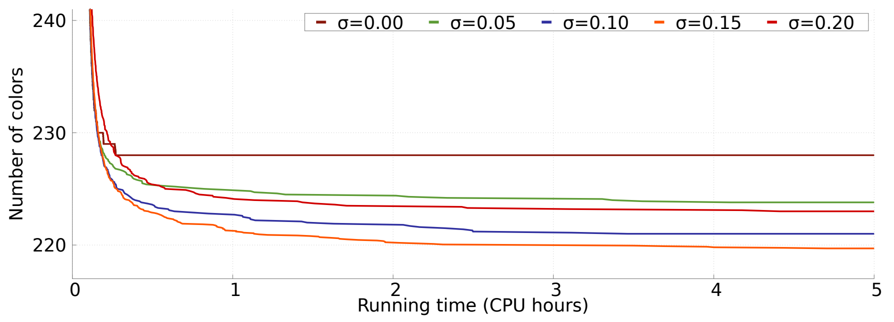

Figure 5: Number of colors over time for the instance `vispecn13806` for different values of σ. In both figures the algorithm uses $p = 1.2$, easy vertices, $q_{\max} = 59022$, but does not use the BDFS nor any clique. For $\sigma \geq 0.25$, no solution better than 248 colors is found.

Option (e) The goal of BDFS is to further optimize very good solutions that the conflict optimizer is not able to improve otherwise. Fig. 4 shows the influence of BDFS. While on this figure, the advantages of BDFS cannot be noticed, its use near the end of the challenge improved about 30 solutions.

The *bounded depth-first search* (BDFS) algorithm tries to improve the dequeuing process. The goal is to prevent a vertex in conflict with some adjacent colored vertices from entering in the conflict set. At the first level, the algorithm searches for a recoloring of some adjacent vertices which allows us to directly recolor the conflict vertex. If no solution is found, the algorithm could recolor some vertices at larger distances from the conflict vertex. To do so, a local search is performed by trying to recolor vertices at a bounded distance from the conflict vertex in the current partial solution.

The BDFS algorithm has two parameters: *adjacency bound* $a_{\max}$ and *depth* d. In order to recolor a vertex v, BDFS gets the set $\mathcal{C}$ of color classes with at most $a_{\max}$ neighbors of v. If a class in $\mathcal{C}$ has no neighbor of v, v is assigned to C. Otherwise, for each class $C \in \mathcal{C}$, BDFS tries to recolor the vertices in C which are adjacent to v by recursively calling itself with depth $d-1$. At depth $d = 0$ the algorithm stops trying to color the vertices.

During the challenge the Shadoks used BDFS with parameters $a_{\max} = 3$ and $d = 3$. The depth was increased to 5 (resp. 7) when the number of vertices in the queue was 2 (resp. 1).

Degeneracy order Given a target number of colors k, we call *easy vertices* a set of vertices Y such that, if the remainder of the vertices of G' are colored using k colors, then we are guaranteed to be able to color all vertices of G' with k colors. This is obtained using the degeneracy order Y. To obtain Y we iteratively remove from the graph a vertex v that has at most $k-1$ neighbors, appending v to the end of Y. We repeat until no other vertex can be added to Y. Notice that, once we color the remainder of the graph with at least k colors, we can use a greedy coloring for Y in order from last to first without increasing the number of colors used. Removing the easy vertices reduces the total number of vertices, making the conflict optimizer more effective. The Shadoks always toggle this option on (the challenge instances contain from 0 to 23% easy vertices).

6 Results

We provide the results of the experiments performed with the code from the three teams on two classes of instances. First, we present the results on some selected CG:SHOP 2022 instances. These instances are intersection graphs of line segments. Second, we execute the code on graphs that are not intersection graphs, namely the classic DIMACS graphs [15], comparing the results of our conflict optimizer implementations to previous solutions. The source code for the three teams is available at:

- Lasa: `https://github.com/librallu/dogs-color`
- Gitastrophe: `https://github.com/jacketsj/cgshop2022-gitastrophe`
- Shadoks: `https://github.com/gfonsecabr/shadoks-CGSHOP2022`

6.1 CG:SHOP 2022 Instances

We selected 14 instances (out of 225) covering the different types of instances given in the CG:SHOP 2022 challenge. The results are presented in Table 1. For comparison, we executed the HEAD [22] code on some instances using the default parameters. The table shows the smallest number of colors for which HEAD found a solution. We ran HEAD for 1 hour of repetitions for each target number of colors on a single CPU core (the HEAD solver takes the target number of colors as a parameter and we increased this parameter one by one). At the end of the challenge, 8 colorings computed by Lasa, 11 colorings computed by Gitastrophe, and 23 colorings computed by Shadoks over 225 instances have been proved optimal (their number of colors is equal to the size of a clique).

In order to compare the efficiency of the algorithms, we executed the different implementations on the CG:SHOP instance `vispecn13806`. The edge density of this graph is 19%, the largest clique that we found has 177 vertices and the best coloring found during the challenge uses 218 colors. Notice that `vispecn13806` is the same instance used in other Shadoks experiments in Section 5. Notice also that HEAD algorithm provides 283 colors after one hour compared

Table 1: Several CG:SHOP 2022 results. We compare the size of the largest known clique to the smallest coloring found by each team on a selection of 14 CG:SHOP 2022 instances.

Instance	Clique	Best	HEAD [22]	Lasa	Gitastrophe	Shadoks
`rvisp5013`	46	**49**	59	50	**49**	**49**
`rsqrpecn8051`	173	**175**	207	177	176	**175**
`vispecn13806`	77	**218**	283	224	221	**218**
`rsqrp14364`	134	**136**	174	137	137	**136**
`vispecn19370`	169	**192**	266	197	194	**192**
`rvisp24116`	97	**104**	166	110	105	**104**
`visp26405`	78	**81**	112	83	**81**	**81**
`sqrp28863`	**190**	**190**	297	191	191	**190**
`visp38574`	118	**133**	199	138	134	**133**
`sqrpecn45700`	460	**462**		465	465	**462**
`reecn51526`	308	**310**		315	312	**310**
`vispecn58391`	305	**367**		380	369	**367**
`vispecn65831`	357	**439**		453	440	**439**
`sqrp72075`	264	**269**		272	271	**269**

to less than 240 colors for the conflict optimizers. We ran the three implementations on three different servers and compared the results shown in Figure 6. For each implementation, the x coordinate is the running time in hours, while the y coordinate is the smallest number of colors found at that time.

6.2 Results on DIMACS Graphs

We tested the implementation of each team on the DIMACS instances [15] to gauge the performance of the conflict optimizer on other classes of graphs. We compared our results to the best known bounds and to the state of the art coloring algorithms HEAD [22] and QACOL [27, 28].

The time limit for Lasa's algorithms is 1 hour. CWLS is Lasa's conflict optimizer with the neighbourhood presented in TABUCOL [13], while PWLS is the optimizer with the neighbourhood presented in PARTIALCOL [1]. Gitastrophe algorithm ran 10 minutes after which the number of colors no longer decreases. Shadoks algorithm ran for 1 hour without the BDFS option (results with BDFS are worse).

Results are presented in Table 2. We only kept the difficult DIMACS instances. For the other instances, all the results match the best known bounds. The DIMACS instances had comparatively few edges (on the order of thousands or millions); the largest intersection graphs considered in the CG:SHOP challenge had over 1.5 billion edges.

We notice that the conflict optimizer works extremely poorly on random graphs, but it is fast and appears to perform well on geometric graphs (r250.5, r1000.1c, r1000.5, dsjr500.1c and dsjr500.5), matching the best-known results [11]. Interestingly, these geometric graphs are not intersection graphs as in the CG:SHOP challenge, but are generated based on a distance threshold. On the DIMACS graphs, Lasa implementation shows better performance than the other implementations.

7 Acknowledgments

We would like to thank the challenge organizers and other competitors for their time, feedback, and making this whole event possible.

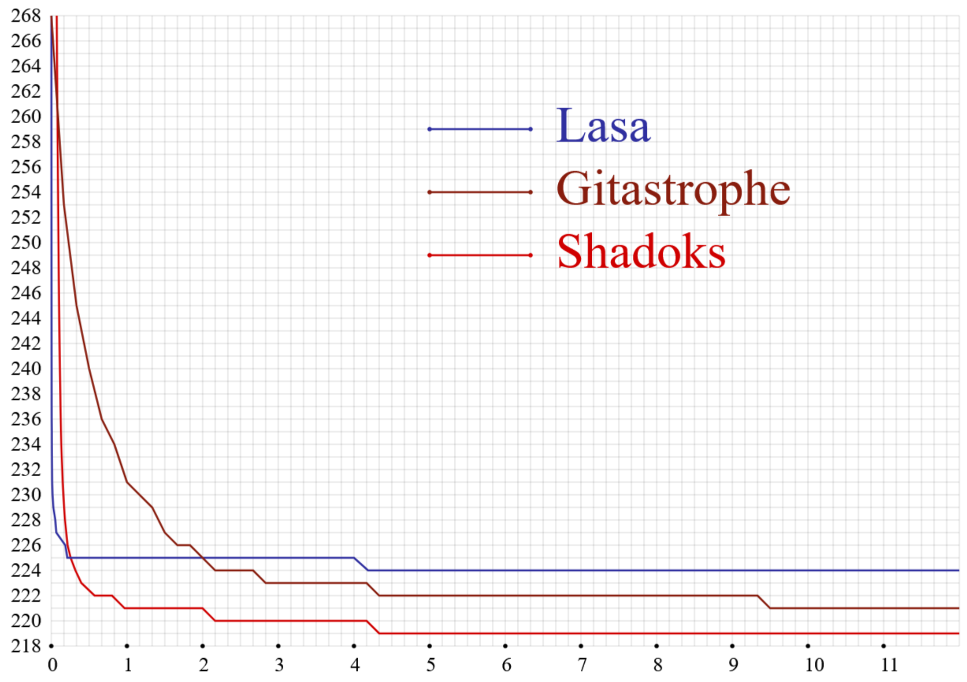

Figure 6: Number of colors over time (in hours) for the instance `vispecn13806`.

Table 2: Comparison of our method with state-of-the-art graph coloring algorithms. The conflict optimizer underperforms except on the geometric graphs `r*` and `dsjr*`.

Instance	Best	HEAD	QACOL	Lasa	Lasa	Gitastrophe	Shadoks
Instance		[22]	[27, 28]	CWLS	PWLS		
`dsjc250.5`	**28**	**28**	**28**	**28**	29	29	**28**
`dsjc500.1`	**12**	**12**	**12**	13	13	13	13
`dsjc500.5`	**47**	**47**	48	49	51	52	50
`dsjc500.9`	**126**	**126**	**126**	**126**	130	130	128
`dsjc1000.1`	**20**	**20**	**20**	21	22	21	21
`dsjc1000.5`	**82**	**82**	**82**	89	94	93	91
`dsjc1000.9`	**222**	**222**	**222**	223	240	235	231
`r250.5`	**65**	**65**	**65**	**65**	**65**	**65**	**65**
`r1000.1c`	**98**	**98**	**98**	**98**	**98**	**98**	**98**
`r1000.5`	**234**	245	238	**234**	**234**	**234**	237
`dsjr500.1c`	**84**	85	85	85	85	85	85
`dsjr500.5`	**122**	-	**122**	**122**	**122**	**122**	**122**
`le450_25c`	**25**	**25**	**25**	26	26	26	26
`le450_25d`	**25**	**25**	**25**	26	26	26	26
`flat300_28_0`	28	31	31	31	32	33	32
`flat1000_50_0`	**50**	**50**	-	**50**	**50**	91	54
`flat1000_60_0`	**60**	**60**	-	**60**	92	93	90
`flat1000_76_0`	**81**	**81**	**81**	88	93	92	90
`C2000.5`	**145**	146	**145**	165	173	173	168
`C4000.5`	**260**	266	259	311	320	317	312

The Shadoks would like to thank Hélène Toussaint, Raphaël Amato, Boris Lonjon, and William Guyot-Lénat from LIMOS, as well as the Qarma and TALEP teams and Manuel Bertrand from LIS, who continue to make the computational resources of the LIMOS and LIS clusters available to our research.

The work of Loïc Crombez has been sponsored by the French government research program "Investissements d'Avenir" through the IDEX-ISITE initiative 16-IDEX-0001 (CAP 20-25). The work of Guilherme D. da Fonseca is supported by the French ANR PRC grant ADDS (ANR-19-CE48-0005). The work of Yan Gerard is supported by the French ANR PRC grants ADDS (ANR-19-CE48-0005), ACTIVmap (ANR-19-CE19-0005) and by the French government IDEX-ISITE initiative 16-IDEX-0001 (CAP 20-25). The work of Aldo Gonzalez-Lorenzo is supported by the French ANR PRC grant COHERENCE4D (ANR-20-CE10-0002). The work of Pascal Lafourcade is supported by the French ANR PRC grant MobiS5 (ANR-18-CE39-0019), DECRYPT (ANR-18-CE39-0007), SEVERITAS (ANR-20-CE39-0005) and by the French government IDEX-ISITE initiative 16-IDEX-0001 (CAP 20-25). The work of Luc Libralesso is supported by the French ANR PRC grant DECRYPT (ANR-18-CE39-0007).

References

[1] Ivo Blöchliger and Nicolas Zufferey. A graph coloring heuristic using partial solutions and a reactive tabu scheme. *Computers & Operations Research*, 35(3):960–975, 2008. `doi:https://doi.org/10.1016/j.cor.2006.05.014`.

[2] Daniel Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256, 1979. `doi:https://doi.org/10.1145/359094.359101`.

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

[4] Loïc Crombez, Guilherme Dias da Fonseca, Yan Gerard, Aldo Gonzalez-Lorenzo, Pascal Lafourcade, and Luc Libralesso. Shadoks approach to low-makespan coordinated motion planning. *ACM J. Exp. Algorithmics*, 27:3.2:1–3.2:17, 2022. `doi:10.1145/3524133`.

[5] David Eppstein. Small maximal independent sets and faster exact graph coloring. *J. Graph Algorithms Appl*, 7(2):131–140, 2002. `arXiv:https://doi.org/10.48550/arXiv.cs/0011009`.

[6] Sándor P. Fekete, Phillip Keldenich, Dominik Krupke, and Stefan Schirra. Minimum partition into plane subgraphs: The CG: SHOP Challenge 2022. *CoRR*, abs/2203.07444, 2022. URL: `https://arxiv.org/abs/2203.07444`, `arXiv:2203.07444`.

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

[8] Fabio Furini, Virginie Gabrel, and Ian-Christopher Ternier. An improved dsatur-based branch-and-bound algorithm for the vertex coloring problem. *Networks*, 69(1):124–141, 2017. doi:10.1002/net.21716.

[9] Fabio Furini and Enrico Malaguti. Exact weighted vertex coloring via branch-and-price. *Discrete Optimization*, 9(2):130–136, 2012. URL: https://www.sciencedirect.com/science/article/pii/S1572528612000205, doi:https://doi.org/10.1016/j.disopt.2012.03.002.

[10] Philippe Galinier and Jin-Kao Hao. Hybrid evolutionary algorithms for graph coloring. *Journal of combinatorial optimization*, 3(4):379–397, 1999. doi:https://doi.org/10.1023/A:1009823419804.

[11] Olivier Goudet, Cyril Grelier, and Jin-Kao Hao. A deep learning guided memetic framework for graph coloring problems, 2021. arXiv:2109.05948.

[12] Stefano Gualandi and Federico Malucelli. Exact solution of graph coloring problems via constraint programming and column generation. *INFORMS Journal on Computing*, 24(1):81–100, 2012. doi:https://doi.org/10.1287/ijoc.1100.0436.

[13] Alain Hertz and Dominique de Werra. Using tabu search techniques for graph coloring. *Computing*, 39(4):345–351, 1987. doi:https://doi.org/10.1007/BF02239976.

[14] Tommy R. Jensen and Bjarne Toft. *Graph coloring problems.* John Wiley & Sons, 2011.

[15] David S Johnson and Michael A Trick. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*, volume 26. American Mathematical Society, 1996.

[16] Frank Thomson Leighton. A graph coloring algorithm for large scheduling problems. *Journal of research of the national bureau of standards*, 84(6):489, 1979. doi:10.6028/jres.084.024.

[17] R. M. R. Lewis. *A Guide to Graph Colouring: Algorithms and Applications.* Springer Publishing Company, Incorporated, 1st edition, 2015.

[18] C. Lucet, F. Mendes, and A. Moukrim. An exact method for graph coloring. *Computers & Operations Research*, 33(8):2189–2207, 2006. URL: https://www.sciencedirect.com/science/article/pii/S0305054805000080, doi:https://doi.org/10.1016/j.cor.2005.01.008.

[19] David W. Matula, George Marble, and Joel D. Isaacson. Graph coloring algorithms. In Ronald C. Read, editor, *Graph Theory and Computing*, pages 109–122. Academic Press, 1972. URL: https://www.sciencedirect.com/science/article/pii/B9781483231877500155, doi:https://doi.org/10.1016/B978-1-4832-3187-7.50015-5.

[20] Anuj Mehrotra and Michael A Trick. A column generation approach for graph coloring. *INFORMS Journal on Computing*, 8(4):344–354, 1996. doi:https://doi.org/10.1287/ijoc.8.4.344.

[21] Steven Minton, Mark D Johnston, Andrew B Philips, and Philip Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial intelligence*, 58(1-3):161–205, 1992. doi:https://doi.org/10.1016/0004-3702(92)90007-K.

[22] Laurent Moalic and Alexandre Gondran. Variations on memetic algorithms for graph coloring problems. *Journal of Heuristics*, 24(1):1–24, 2018. arXiv:https://doi.org/10.48550/arXiv.1401.2184.

[23] Isabel Méndez-Díaz and Paula Zabala. A branch-and-cut algorithm for graph coloring. *Discrete Applied Mathematics*, 154(5):826–847, 2006. URL: https://www.sciencedirect.com/science/article/pii/S0166218X05003094, doi:https://doi.org/10.1016/j.dam.2005.05.022.

[24] Pablo San Segundo. A new dsatur-based algorithm for exact vertex coloring. *Computers & Operations Research*, 39(7):1724–1733, 2012. doi:https://doi.org/10.1016/j.cor.2011.10.008.

[25] André Schidler. Sat-based local search for plane subgraph partitions (CG challenge). In Xavier Goaoc and Michael Kerber, editors, *38th International Symposium on Computational Geometry, SoCG 2022, June 7-10, 2022, Berlin, Germany*, volume 224 of *LIPIcs*, pages 74:1–74:8. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.SoCG.2022.74.

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

[27] Olawale Titiloye and Alan Crispin. Quantum annealing of the graph coloring problem. *Discrete Optimization*, 8:376–384, 2011. doi:https://doi.org/10.1016/j.disopt.2010.12.001.

[28] Olawale Titiloye and Alan Crispin. Parameter tuning patterns for random graph coloring with quantum annealing. *PloS one*, 7(11), 2012. doi:https://doi.org/10.1371/journal.pone.0050060.

[29] Edward P. K. Tsang. *Foundations of constraint satisfaction.* 1993.

[30] D. J. A. Welsh and M. B. Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1):85–86, 01 1967. arXiv:https://academic.oup.com/comjnl/article-pdf/10/1/85/1069035/100085.pdf, doi:10.1093/comjnl/10.1.85.