Shadoks Approach to Minimum Partition into Plane Subgraphs

Loïc Crombez
LIMOS, Université Clermont Auvergne, Aubière, France

Guilherme D. da Fonseca
LIS, Aix-Marseille Université, France

Yan Gerard
LIMOS, Université Clermont Auvergne, Aubière, France

Aldo Gonzalez-Lorenzo
LIS, Aix-Marseille Université, France

Abstract

We explain the heuristics used by the `Shadoks` team to win first place in the CG:SHOP 2022 challenge that considers the minimum partition into plane subgraphs. The goal is to partition a set of segments into as few subsets as possible such that segments in the same subset do not cross each other. The challenge has given 225 instances containing between 2500 and 75000 segments. For every instance, our solution was the best among all 32 participating teams.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases Plane graphs, graph coloring, intersection graph, conflict optimizer, line segments, computational geometry

Digital Object Identifier 10.4230/LIPIcs.SoCG.2022.71

Category CG Challenge

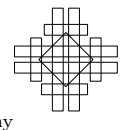
Supplementary Material *Software (Source Code)*: `https://github.com/gfonsecabr/shadoks-CGSHOP2022`; archived at `swh:1:dir:ec88e5b901c034d5a91aa133e824d65cff3788a3`

Funding *Guilherme D. da Fonseca*: This work is supported by the French ANR PRC grant ADDS (ANR-19-CE48-0005).
Yan Gerard: This work is supported by the French ANR PRC grant ADDS (ANR-19-CE48-0005).
Aldo Gonzalez-Lorenzo: This work is supported by the French ANR PRC grant COHERENCE4D (ANR-20-CE10-0002).

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

1 Introduction

This paper presents our strategy to win first place in the CG:SHOP 2022 geometric optimization challenge. This edition proposed a problem called *minimum partition into plane subgraphs*. The goal is to partition the set of the edges of a given graph G embedded in the plane (with line segments as edges) into a small number k of plane graphs. The problem reduces to graph coloring a conflict graph G_C where the vertices of G_C are the segments of G and two vertices of G_C are connected by an edge if the corresponding segments cross each other (for details on the definition of *cross*, see [4]).



The study of graph coloring goes back to the 4-color problem (1852) and the problem has been intensively studied since the 1970s [9]. Many heuristics have been proposed [6, 8, 13, 14], as well as exact algorithms [3, 7, 12] (see for instance the book [11]). In this paper we present the ideas we used in the competition. The main element is a *Conflict Optimizer*, that does not use any geometry. It is based on the same approach we used to solve low-makespan coordinated motion planning in the CG:SHOP 2021 challenge [2]. Our initial solutions, however, make extensive use of geometry. The code is available on github.

The paper is organized as follows. Section 2 presents some heuristics that we used to compute initial solutions. In Section 3 we describe the technique used to improve a solution. Section 4 details our implementation of the algorithm and a parameter analysis. Section 5 describes the results we obtained.

2 Initial Solutions

The most simple method to produce a (reasonably small) coloring of a graph is the classic greedy heuristic: for each segment e, we color e using the first color *available*, i.e. that is not already used by any of the segments that cross e. If necessary, we create a new color. The order by which we consider the segments influences the quality of the solution. We refer to the greedy heuristic using the order by which the segments appear on the instance files as `Greedy` when comparing the results. Sorting the segments in angular order (and trying different starting angles) produces good solutions to the challenge instances. We refer to this simple heuristic as `Angle`.

The *squeaky wheel* paradigm has been widely applied to graph coloring [10]. The idea is to run a heuristic, detect elements that have been solved poorly, and run the same heuristic again handling these elements earlier this time. The procedure is repeated several times and the best solution found is returned. We use this paradigm together with `Angle` as follows. Throughout the algorithm, the segments are partitioned into two lists *Good*, *Bad*, both kept sorted by angle. Initially, all segments are in *Good*. At each step we apply the greedy coloring first treating the segments in *Bad* and then in *Good*. Then, the segments that have been assigned the last color are added to *Bad* and we repeat the procedure. The number of colors used may eventually increase (since both lists are kept sorted by angle). We stop after a certain time or number of steps and return the solution with the smallest number of colors found. We refer to this heuristic as `Bad`.

A classic variation of the greedy coloring is the `DSatur` heuristic [1]. It does not use any geometric information. At each step, we color the segment e that crosses the largest number of different colors, breaking ties by the total number of segments that cross e. As in the standard greedy heuristic, the color assigned to e is the first color that is available.

We modify the `DSatur` heuristic into the `DSHull` heuristic that uses geometric information. We color the segments following the same order criterion as `DSatur`. However, instead of assigning to e the first color available, we choose the color as follows. The segments that have the same color are kept in a set called a *color class*. For each color class C, let $w(C)$ be the area of the convex hull of the segments in C. When coloring a segment e, we choose, among the color classes C that are available for e, the one that minimizes $w(C \cup \{e\}) - w(C)$. Ties are broken arbitrarily and if no color class is available for e, then we create a new color class containing only e. The intuition is that a small increase in the convex hull areas corresponds to a compact packing of the segments, producing larger color classes.

A comparison of the heuristics on several challenge instances is presented in Table 1.

Table 1 Initial solutions produced by several heuristics compared against the best solution found.

instance	density	Greedy	Angle	Bad	DSatur	DSHull	Best
rsqrpecn8051	41%	342	205	203	213	**201**	175
vispecn13806	19%	427	308	300	289	**283**	218
rsqrp14364	50%	294	**139**	**139**	165	157	136
vispecn19370	13%	370	285	278	265	**248**	192
visp26405	7%	154	101	97	94	**92**	81
visp31334	5%	152	90	**88**	99	98	81
visp38574	14%	287	148	**146**	168	168	133
sqrpecn45700	47%	952	504	**500**	562	522	462
reecn51526	24%	642	361	**359**	388	360	310
vispecn58391	12%	789	607	594	499	**494**	367
vispecn65831	12%	916	647	637	578	**564**	439
sqrp72075	47%	609	**280**	**280**	363	337	269

3 Improving Solutions

In this section we describe our optimization approach that we call *Conflict Optimizer*. Section 3.1 describes the backbone of the *conflict optimizer*. Section 3.2 describes some improvements that were made to the conflict optimizer in order to get better solutions.

3.1 Conflict optimizer

The goal of the conflict optimizer is to remove one color from a given solution with k colors. Let C_0 be a color class. The conflict optimizer puts all segments of C_0 in a queue Q and deletes C_0. We now have a partial solution with $k-1$ colors and a queue Q that contains uncolored segments. The goal is to empty Q by coloring every segment in Q.

At each step until Q is empty, we pop a segment e from Q and color e as follows. If there exists a color class C such that no segment in C crosses e, then we add e to C. In most cases, such C does not exist and we choose C to minimize the following cost function. Let $q(e)$ be the number of times the segment e has been added to Q. The *penalty* for adding e to Q is $1 + q(e)^p$. The *cost* of each color class C is the product of a Gaussian random variable of mean 1 and variance σ with the sum of the penalties of the segments of C that cross e. The values of the parameters p, σ are analysed in Section 4 ($p = 1.2$ and $\sigma = 0.15$ are good default values).

3.2 Modifications to the conflict optimizer

In this section we describe several modifications that we made to the conflict optimizer described in Section 3.1. In our code, we developed several options that can be toggled on or off. The impact on the computation of solutions is discussed in Section 4.

Easy segments. Given an objective number of colors k, we call *easy segments* a list of segments S such that, if the remainder of the segments of S are colored using k colors, then we are guaranteed to be able to color all segments with k colors. To obtain S we iteratively remove from the graph a segment e that has at most $k-1$ crossings, appending e to S. We repeat until no other segment can be added to S. Notice that, once we color the remainder

of the graph with at least k colors, we can use a greedy coloring for S in order from last to first without increasing the number of colors used. Removing the easy segments reduces the total number of segments, making the conflict optimizer more effective.

Clique segments. A *clique* is a set of mutually crossing segments. We used several heuristics to produce large cliques. Let K be the largest clique we found for a given instance. Since the segments of K must have different colors, we forbid the segments in K from entering the queue by setting a infinite penalty.

Restarting. We implemented a strategy to restart the conflict optimizer. We set a hard limit $q_{\max}$ to how many times a segment can be queued. Once a segment e has been queued $q_{\max}$ times, the penalty of e becomes infinite. Once it becomes impossible to color a segment from the queue (that is, the minimum cost in infinite), the conflict optimizer aborts and restarts. When restarting, the coloring is shuffled by moving segments that fit multiple color classes.

Bounded Depth-First Search. The *bounded depth-first search* (BDFS) algorithm tries to improve the dequeuing process. The goal is to prevent a segment from being queued by locally recoloring a bounded number of segments in the current partial solution. To do so, we perform a local search into the tree of possible ways to color the segments.

The BDFS algorithm has two parameters: *crossing bound* $c_{\max}$ and *depth* d. In order to recolor a segment e, BDFS gets the set $\mathcal{C}$ of color classes with at most $c_{\max}$ crossings with e. If a class of $\mathcal{C}$ has no crossings with e, we assign e to C. Otherwise, for each class $C \in \mathcal{C}$, BDFS tries to recolor the list of segments in C that cross e by recursively calling itself with depth $d-1$. At depth $d=0$ the algorithm stops trying coloring the segments.

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

4 Implementation and Experiments

In this section, we describe the techniques we used to efficiently implement the conflict optimizer. We also analyze the influence of the different parameters and options.

4.1 Implementation

We implemented our algorithm in C++ using only the standard library. As the conflict optimizer spends most of its time testing crossings, we precompute the crossings. To save memory space, we stored the crossing state of each pair of segments using just one bit, which allows us to store the largest instances of the challenge on less than 800MB.

4.2 Parameter analysis

The two parameters of the conflict optimizer are the variance σ of the Gaussian noise and the exponent p of the penalty. The two others options BDFS and multistart can be activated to improve solutions that have already been optimized several times.

Parameters σ and p. Figure 1 shows the influence of both these parameters (the initial solutions used for the figure are computed using `Greedy`). In all figures, the number of colors shown is the average of multiple executions of the code using different random numbers.

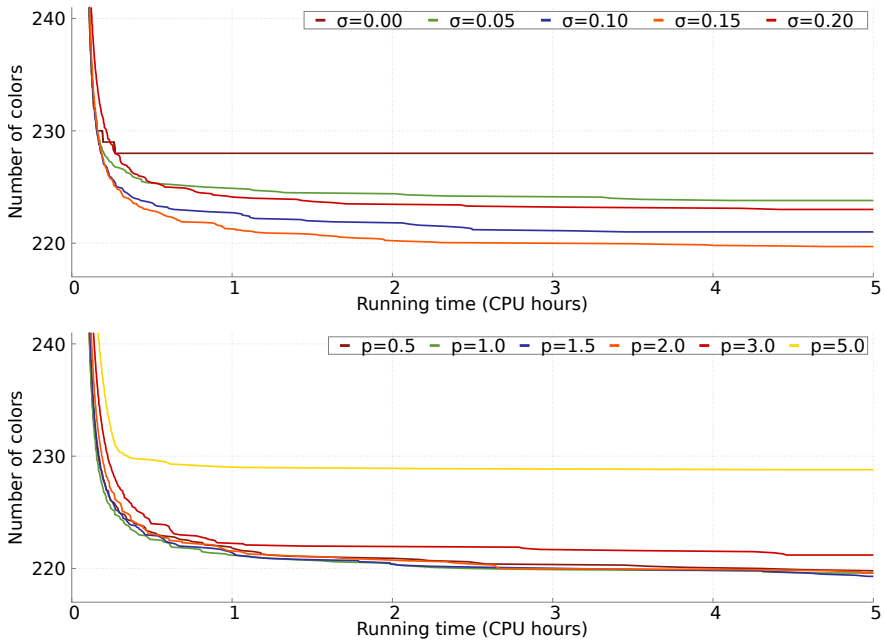


Figure 1 Number of colors over time for the instance `vispecn13806` using different parameters. In both figures the algorithm uses easy segments, $q_{\max} = 59022$, but does not use the BDFS nor any clique. The first plot shows results with different values of σ for $p = 1.2$. The second plot shows results with different values of p for $\sigma = 0.15$.

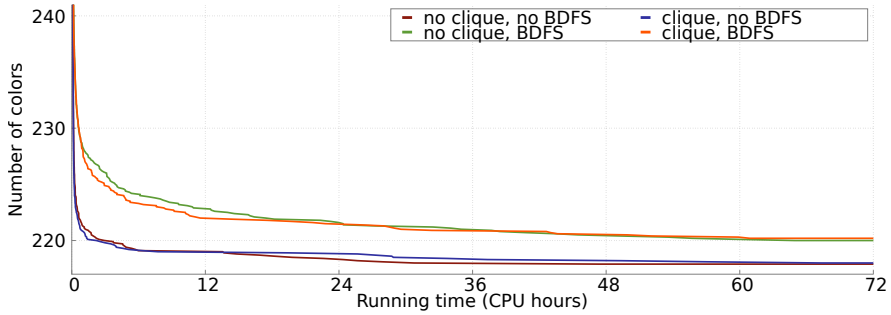


Figure 2 Number of colors over time with and without clique knowledge and BDFS obtained on the instance `vispecn13806`. Parameters are $\sigma = 0.15$, $p = 1.2$, and $q_{\max} = 1500000$).

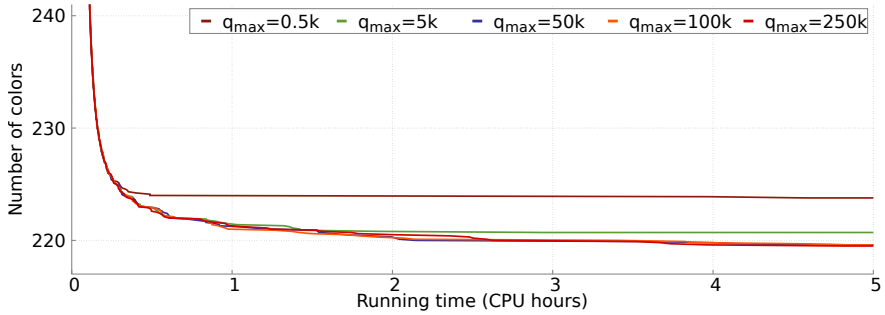


Figure 3 Number of colors over time with different values of $q_{\max}$ obtained on the instance `vispecn13806`. Parameters are $\sigma = 0.15$, $p = 1.2$, no clique knowledge, and no BDFS.

Options multistart and BDFS. The goal of multistart and BDFS is to further optimize very good solutions that the conflict optimizer is not able to improve otherwise. Figure 2 shows the influence of large clique knowledge and BDFS. While on this figure, the advantages of BDFS cannot be noticed, its use near the end of the challenge improved about 30 solutions.

Looking at Figure 3, the maximal number of times a segment can be queued does not seem to have much influence as long as its value is not too small. Throughout the challenge we almost exclusively used $q_{\max} = 2000 \cdot (75000/m)^2$, where m is the number of segments. This value roughly ensures a restart every few hours.

5 Challenge Results

We won first place in the challenge with the best solution among all 32 participating teams for all 225 instances. We also showed that 23 of those solutions are optimal by identifying a clique as large as the number of colors.

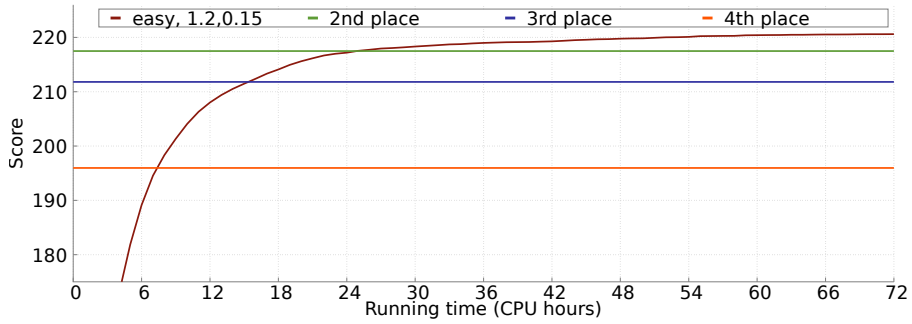


Figure 4 Evolution of the score over time compared to the scores of second to fourth place. The same parameters are used on all instances ($p = 1.2, \sigma = 0.15$, and easy segments are computed).

After generating initial solutions we ran our conflict optimizer with various parameters. The clique knowledge and the easy segments reduction were always used. Most of the time we used $\sigma = 0.15 \pm 0.05$ and $p = 1.2 \pm 0.1$. The BDFS strategy was used in the last couple of weeks of the challenge. We estimate that on average, we spent two to three weeks of single core of an Intel Xeon E5-2670 CPU per instance. However, despite the large amount of computing power used during the challenge, and the varying parameters of our algorithms, we note that after 25 hours of computation on each file, starting from the `Greedy` solution, using only the easy segments optimization and parameters $p = 1.2, \sigma = 0.15$, our conflict optimizer reaches a score of 217.64 on the CG:SHOP 2022 instances, which is better than the second place score (see Figures 4, 5). We note that the second and third team [16, 5] also use a conflict optimizer heuristic, while the forth team [15] uses instead a SAT solver coupled with tabu search. Despite several parameters that allow for increased diversity in order to find really good solutions, our conflict optimizer still performs well with default parameters. Finally, as the optimizer does not make use of any geometric property, it might be interesting in the future to test its performance on other classes of graphs.

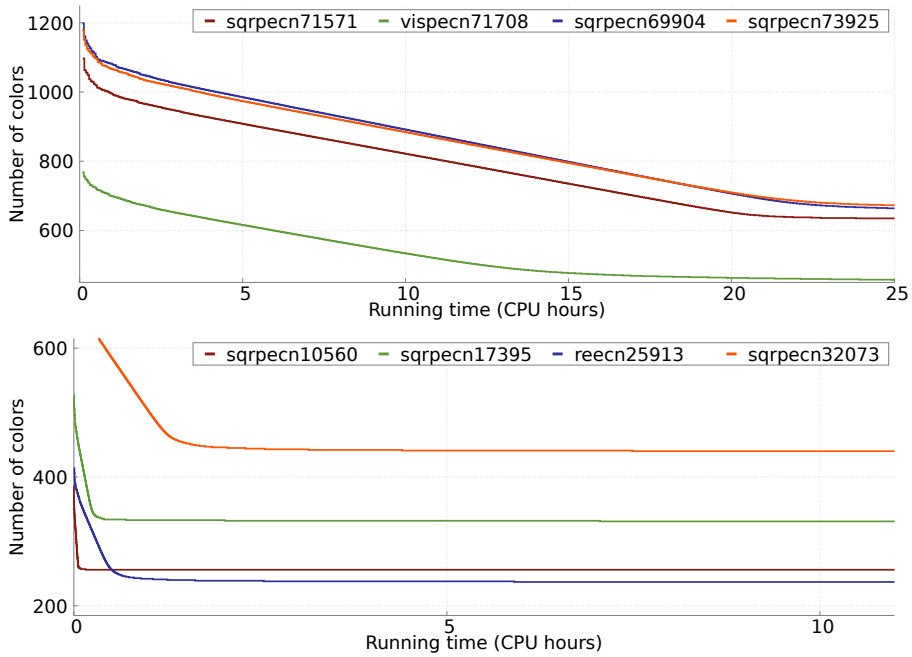


Figure 5 Challenge scores over time for several instances.

References

1. Daniel Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256, 1979.
2. Loïc Crombez, Guilherme D. da Fonseca, Yan Gerard, Aldo Gonzalez-Lorenzo, Pascal Lafourcade, and Luc Libralesso. Shadoks approach to low-makespan coordinated motion planning (CG challenge). In *37th International Symposium on Computational Geometry, SoCG 2021*, pages 63:1–63:9, 2021.
3. David Eppstein. Small maximal independent sets and faster exact graph coloring. *J. Graph Algorithms Appl*, 7(2):131–140, 2002.
4. 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. `arXiv:2203.07444`.
5. Florian Fontan, Pascal Lafourcade, Luc Libralesso, and Benjamin Momège. Local search with weighting schemes for the CG:SHOP 2022 competition. In *Symposium on Computational Geometry (SoCG)*, pages 73:1–73:7, 2022.
6. Philippe Galinier and Jin-Kao Hao. Hybrid evolutionary algorithms for graph coloring. *Journal of combinatorial optimization*, 3(4):379–397, 1999.
7. 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.
8. Alain Hertz and Dominique de Werra. Using tabu search techniques for graph coloring. *Computing*, 39(4):345–351, 1987.
9. Tommy R. Jensen and Bjarne Toft. *Graph coloring problems*. John Wiley & Sons, 2011.
10. David E. Joslin and David P. Clements. Squeaky wheel optimization. *Journal of Artificial Intelligence Research*, 10:353–373, 1999.
11. R. M. R. Lewis. *A Guide to Graph Colouring: Algorithms and Applications*. Springer Publishing Company, Incorporated, 1st edition, 2015.

12 Corinne Lucet, Florence Mendes, and Aziz Moukrim. An exact method for graph coloring. *Computers & Operations Research*, 33(8):2189–2207, 2006.

13 David W. Matula, George Marble, and Joel D. Isaacson. Graph coloring algorithms. In *Graph theory and computing*, pages 109–122. Elsevier, 1972.

14 Isabel Méndez-Díaz and Paula Zabala. A branch-and-cut algorithm for graph coloring. *Discrete Applied Mathematics*, 154(5):826–847, 2006.

15 André Schidler. SAT-based local search for plane subgraph partitions. In *Symposium on Computational Geometry (SoCG)*, pages 74:1–74:8, 2022.

16 Jack Spalding-Jamieson, Brandon Zhang, and Da Wei Zheng. Conflict-based local search for minimum partition into plane subgraphs. In *Symposium on Computational Geometry (SoCG)*, pages 72:1–72:6, 2022.