Graph Coloring - SOTA and Benchmark Instances

Apetrii Radu, Baranceanu Vlad, Brezuleanu Alex

IAO 1

1 Graph Coloring Problem

The problem of graph coloring refers to assigning colors/numbers to vertices in such a way that no two adjacent vertices have the same color, while minimizing the total number of colors used. This dates back to the *19-th* century and was later classified as a *NP-complete* problem.

There are multiple techniques that have been used throughout the time, many of them providing really good results. In the next chapter, we plan on getting into details on some of these methods.

2 SOTA Approaches

Considering that the Graph Coloring problem is such a well-known one, there have been numerous attempts at solving it, each method consisting of specific ideas, mechanisms, resources. Knowing this, we have decided to tackle only the methods that can be divided/grouped into three main categories: *heuristic* approaches, *meta-heuristic* approaches, *hybrid* approaches. In terms of meaning, they consist of the following:

- *heuristic*: a practical rule or strategy used to make decisions during the coloring process without guaranteeing optimality. It is mainly based on intuitive or empirical principles rather that rigorous mathematical proof.
- *meta-heuristic*: a higher-level strategy used to efficiently explore the solution space, aiming to find near-optimal solutions without guaranteeing optimality.
- *hybrid*: a strategy that combines multiple techniques, often incorporating both heuristic and meta-heuristic methods, to explore their strengths and overcome their limitations.

2.1 Heuristic approaches

2.1.1 DSatur

A greedy algorithm mainly used for its simplicity and effectiveness in producing reasonably good solutions for the graph coloring problem. It focuses on computing saturation degrees for nodes, which translates into computing the number of different colors used by their neighbors. In short terms, this algorithm runs until all the nodes have been colored, each step consisting of selecting the node with the highest saturation degree that has not been colored yet, color it with the smallest possible color that is not used by its neighbours, and update the saturation degree of the neighbours. [1]

DSatur has been observed to perform well on a wide range of graph instances, including both sparse and dense graphs. It tends to excel when there are vertices with high degrees or high degrees of connectivity, as it prioritizes coloring these vertices early in the process, potentially reducing the overall number of colors needed.

However, DSatur may encounter difficulties with certain types of graphs, such as highly irregular or structured graphs, where the distribution of vertex degrees is uneven or where there are many vertices with similar saturation degrees. In such cases, DSatur may not be able to exploit the available structure effectively, leading to sub-optimal solutions.

2.1.2 Recursive largest first (RLF)

A strategy that intends to build upon DSatur by adding one more phase at each step. Basically, after coloring the selected node, there is a recursive refinement step added into the process. This recursive step considers for each color class, the vertices in non-decreasing order of degree, and then it tries to recolor the current vertex with the color that minimizes conflicts. If conflicts cannot be reduced, revert the coloring back to the previous state. [2]

RLF produces good solutions on tests that have a relatively small number of colors. By iteratively refining the coloring through a recursive process and considering the degree of vertices in the refinement step, RLF aims to reduce the number of conflicts and improve the overall quality of the coloring.

On the other hand, RLF may encounter difficulties with certain types of graphs, such as highly regular or structured graphs, where the recursive refinement step may not yield substantial improvements.

2.2 Meta-heuristic approaches

2.2.1 Genetic Algorithms

A strategy used due to its ability to efficiently explore the solution space and find good-quality solutions. It relies on three key processes during each generation: [3]

- *Selection*: Select individuals from the population for reproduction based on their fitness value. Individuals with higher fitness values are more likely to be selected.
- *Mutation*: Process applied in order to introduce small changes and maintain diversity in the population. This might involve randomly changing the color of a vertex, swapping colors between vertices, or other modifications.
- *Crossover*: Method used for generating offsprings. This involves exchanging genetic material (colors assigned to vertices) between parent solutions to produce new candidate solutions.

Genetic algorithms are capable of finding solutions with relatively few colors compared to the greedy algorithms, particularly for large and complex problem instances. Not only that, but this approach is well-suited for solving large-scale instances, as it can effectively explore the solution space.

However, genetic algorithms can be computationally expensive. Also, if the instance presents characteristics of slow converging, the quality of the solutions obtained by GAs may not always be optimal.

2.2.2 Ant Colony Optimization

An algorithm designed to simulate the behavior of ants, ACO builds solutions iteratively by assigning colors to vertices based on probabilistic decision rules influenced by pheromone trails. In terms of the process, this approach consists of two key components: [4]

- *Ant*: An ant represents a solution that is constructed in an iterative mode by choosing colors in a probabilistic fashion. An ant begins its journey from a start node and goes along the graph until all the nodes have been colored.
- *Colony*: After all the ants of the current generation have finished their movement phase, the algorithm updates the pheromone levels from the edges based on the quality of the solutions found.

Ant Colony Optimization is able to find good solutions that use a few number of colors. Also, it is generally scalable and can handle large-scale instances of the graph coloring problem. However, this method may require a large number of iterations to converge to those good solutions.

2.3 Hybrid approaches

2.3.1 Genetic Algorithm with Simulated Annealing

Method that combines the strength of Genetic Algorithms and Simulated Annealing. It involves a phase for each algorithm, and can be described as such: [5]

- *Genetic Algorithm Phase*: Keep the ideas of this approach, meaning that there will be a Selection (based on fitness value), a Mutation, and a Crossover phase. This will keep the population evolving.

- *Simulated Annealing Phase*: After the termination condition has been met for the first phase, the Simulated Annealing kicks in. This way, it explores the solution space and further refines the candidate solutions obtained from the genetic algorithm part. It also applies moves that may worsen the solution with a certain probability determined by the current temperature. This allows the algorithm to escape local optima and explore new regions of the solution space.

2.3.2 Ant Colony Optimization with Tabu Search

As in the previous method, this approach consists of two phases: running the Ant Colony algorithm, and then optimizing the results using a Tabu Search: [6]

- *Ant Colony*: This phase is exactly as the one described in the meta-heuristic category, meaning that throughout the generations, each ant forms a solution, and then the algorithm updates the pheromone levels based on the goodness of the solution found.

- *Tabu Search*: This phase happens after the termination criteria is met for the first one. The method then performs local search operations such as swaps or perturbations to explore the neighborhood of the current solution, while also maintaining a tabu list to prevent revisiting previously explored solutions and encourage diversification in the search process.

3 Benchmark instances

In terms of instances that have been used over the years in order to test the quality of a solution, a few popular ones can be found here. [7] We will remark some of them that have unique features:

- *fpsol2.i.1* - Graph that is based on register allocation for variables in real code and contains 496 vertices, 11654 edges. The optimum solution for this instance is 65 colors.

- *inithx.i.1* - Graph which serves the same purpose as the instance before, containing 864 vertices and 18707 edges, and having an optimal solution of 54 colors.

- *latin_square_10* - Graph instance with 900 vertices and 307350 edges, constructed by following the pattern of the latin square.

- *le450_5c* - Leighton graph with 450 vertices, 9803 edges, and an optimal solution of 5 colors.

- *mulsol.i.5* - Graph with 186 vertices, 3973 edges, and an optimal solution of 31 colors.
- *school1_nsh* - Class scheduling graph with 352 vertices, 14612 edges.
- *homer* - Book graph containing 561 vertices, 1629 edges and an optimal solution of 13 colors.
- *miles750* - Miles graph containing 128 vertices, 2113 edges and an optimal solution of 31 colors.
- *queen16_16* - Queen graph containing 256 vertices and 12640 edges.
- *myciel7* - Graph based on the *Mycielski* transformation that contains 191 vertices, 2360 edges and an optimal solution of 8 colors.

References

[1] https://www.geeksforgeeks.org/dsatur-algorithm-for-graph-coloring/

[2] https://www.gerad.ca/~alainh/RLFPaper.pdf

[3] https://res.ijsrset.com/page.php?param=IJSRSET1841049

[4] https://www.sciencedirect.com/science/article/pii/S0166218X07001023

[5] Comparative Performance of Modified Simulated Annealing with Simple Simulated Annealing for Graph Coloring Problem - Anindya Jyoti Pala, Biman Rayb, Nordin Zakariaa, Samar Sen Sarmac

[6] https://ieeexplore.ieee.org/document/1631360

[7] https://mat.tepper.cmu.edu/COLOR/instances.html