Graph Chromatic Number

Renato Alexandre Lourenço Dias, nMec: 98380

***Abstract* - The chromatic number problem, also known as the vertice coloring problem, is one of the fundamental problems in graph theory, and it is linked to various other domains. In this paper, it is analyzed the computational complexity of some strategies of solving the given problem, as well as visualization of the experimental results from the implemented algorithms.**

Keywords – Graph Theory, Chromatic Number, Vertice Coloring, Time Complexity, Space Complexity, Greedy, Exhaustive, Heuristic

I. INTRODUCTION

The chromatic number of G is the smallest number of colors needed to properly color the vertices of a graph, such that no two adjacent vertices have the same color.

In the following examples, we can visualize a simple graph with 6 vertices and with a chromatic number of 3, and another graph that also has a chromatic number of 3 despite having 14 vertices.

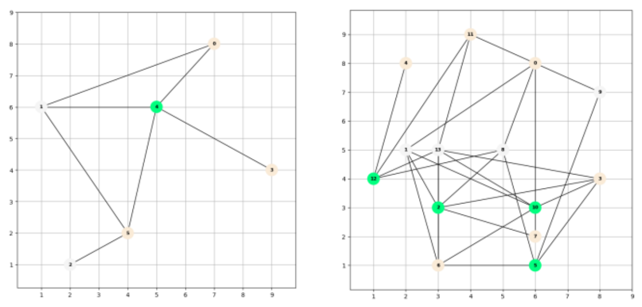


Fig. 1 - Graphs with Chromatic Number of 3.

II. PROBLEM DESCRIPTION

There is no efficient algorithm available for coloring a graph with minimum number of colors as the problem is a known NP Complete problem. There are approximate algorithms to solve the problem though, such as greedy algorithms to assign colors. It doesn't guarantee to use minimum colors, but it guarantees an upper bound on the number of colors [1].

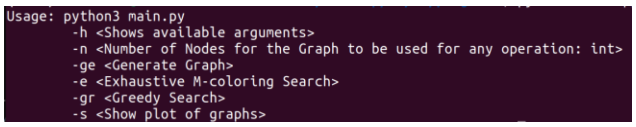
There are computational problems that can not be solved by algorithms even with unlimited time. For example Turing Halting problem. Alan Turing proved that a general algorithm to solve the halting problem for all possible program-input pairs cannot exist. A key part of the proof is, Turing machine was used as a mathematical definition of a computer and program (Source Halting Problem) [2]. Status of NP Complete problems is another failure story, NP complete problems are problems whose status is unknown. No polynomial time algorithm has yet been discovered for any NP complete problem, nor has anybody yet been able to prove that no polynomial-time algorithm exists for any of them. The interesting part is, if any one of the NP complete problems can be solved in polynomial time, then all of them can be solved.

III. APPROACH AND IMPLEMENTATION

A. Parameters Management

The program allows the user to pass some arguments that have different actions. The user can set a few parameters that allow for an easier use of the program and will be useful for allowing the operations that will be explained ahead.

It is important to note that the number of nodes is an identifier for the graph and will be used for the graph generation.



```
Usage: python3 main.py
        -h <Shows available arguments>
        -n <Number of Nodes for the Graph to be used for any operation: int>
        -ge <Generate Graph>
        -e <Exhaustive M-coloring Search>
        -gr <Greedy Search>
        -s <Show plot of graphs>
```

Fig. 2 - Program Arguments.

B. Graph Management

Before implementing a solution for the problem it is necessary to be able to manage graphs at will, for that, it was used external libraries to facilitate the process, NetworkX, along with matplotlib.

B.1 Generating Graphs

The program allows generating random undirected graphs given a number of nodes picked by the user, similarly to an Erdős-Rényi graph, [3], but enforcing that the resulting graph is connected. This is important because for this problem, unconnected graphs would be simply two graphs that would be solved individually. It also makes the graph undirected since considering directions the algorithm would have similar algorithmic complexity but with higher complexity to the visualization of results.

Created graphs also obey a few other rules:

• Vertices are 2D points on the X0Y plane, with integer valued.
• Coordinates must be between 1 and 9.
• Vertices should neither be coincident nor too close.
• The number of edges sharing a vertex is randomly determined.
• If the number of nodes is bigger than 81, a different plane must be used.

First it is generated all edges from a fully connected graph, and a random number that determines the edge probability, then iterates over all edges and if a random number is smaller than the edge probability, it adds the edge to the graph.

For the coordinates, it was used spring layout() function from NetworkX, which allows setting a repulsion force between nodes. Even with this force, there could be coincident coordinates, in that case, it is simply reassigned to a random position that isn't assigned on the plane.

For the final rule, since it is only possible to fit 81 nodes on an integer plane from 1 to 9, the assignment of positions is made on a plane of real numbers between -1 and 1.

B.2 Saving Graphs

Graph's positions and vertice colors are set as attributes on the Graph object, and the program is able to store the object in a text file and plot the image of it.
Both the generated graphs, uncolored, and the solved graphs, with vertices colored, are saved to different directories with a name regarding the number of vertices and the strategy for coloring used if the graph is colored.

B.3 Loading Graphs

The program loads the saved graphs from the disk if a given graph with a certain number of nodes was saved on disk, otherwise, it generates a new one and stores it for later usages. This makes the program run faster, since generating graphs can take a bit of extra time, this way, it only generates it once.

B.4 Plotting Graphs

To visualize graphs, they are demonstrated in a grid on a certain plane with the help of matplotlib, and NetworkX draw() function, which uses the positions of nodes and the colors of vertices stored in the attributes of the Graph object.

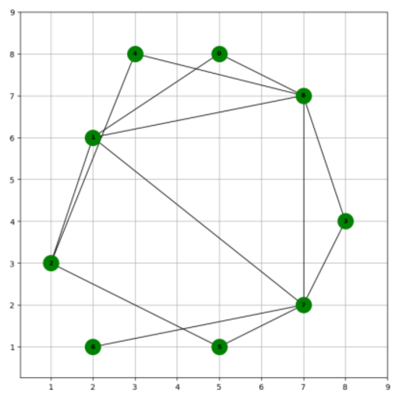


Fig. 3 - Generated Graph.

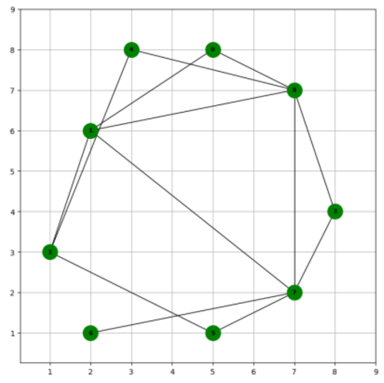


Fig. 4 - Colored Graph.

C. Exhaustive Search

Brute-force search or exhaustive search algorithms normally have a big associated computational cost. This because these strategies consist of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement [4].

They are usually simple to implement and will always find a solution if it exists, implementation costs are proportional to the number of candidate solutions, which in the vertice coloring problem tends to grow very quickly as the size of the problem increases.

There are, however, problem-specific heuristics that can be used to reduce the set of candidate solutions to a manageable size, as it will be demonstrated next.

C.1 M-Coloring

This strategy tries to color the vertices starting with m equal to 1, and tries to assign the m colors one by one to different vertices, starting from a random vertice, and before assigning a color, it checks for safety by considering already assigned colors to connected vertices (vertices that share an edge), if there is a color available then assign it and continue to assign vertices recursively, else if no assignment of color is possible, then it backtracks and returns false. If after backtracking and trying every possible color for the vertice it still fails, it adds up one to m until it finds a solution [5].

The time complexity for this algorithm is O(m^E) for each m, because there are total O(m^E) combination of colors for a given m. The total time complexity is O(m * m^E) or O(m^{E+1}) considering m to be the chromatic number, because every other number of colors smaller than the chromatic number was also tested. The upper bound time complexity remains the same as a naive approach (without backtracking and checking for safety of colors when assigning colors), but the average time taken will be much smaller.

This algorithm has a space complexity of O(E), because it only stores the vertices colored on each configuration it tries.

D. Greedy Search

The greedy algorithm, has mentioned before, only iterates over each vertice once, and, therefore, only searches for one solution one time. When iterating each vertice, checks for adjacent vertices and colors the current vertice with an available color, it simply repeats this process for each vertice and in the end it is guaranteed a solution, however, not an optimal solution [6].

So the time complexity for this algorithm is O(E) for iterating each vertice once, and the space complexity is O(E) for storing the color for each vertice.

IV. RESULTS AND DISCUSSION

For the results, several runs using different strategies were made and noted the execution time, the number of basic operations, which were operations considered to be similar between all algorithms, such as checking adjacent vertices colors and coloring the vertice, the number of solutions/configurations searched were also noted, and finally the chromatic numbers from the solutions found.

A. M-Coloring Exhaustive Search

For the m-coloring strategy, the results were as the table I shows.

TABLE I
Results for M-Coloring Exhaustive Search Strategy

Nodes	Edges	Time	Basic Operations	Solutions Searched	Chromatic Number
2	1	0.000s	9	5	2
3	3	0.000s	27	15	3
4	4	0.000s	14	7	2
5	6	0.000s	58	33	3
6	7	0.000s	94	54	3
7	10	0.000s	67	37	3
8	12	0.000s	47	24	3
9	13	0.000s	215	119	3
10	11	0.000s	54	29	2
15	28	0.002s	7948	3978	4
20	52	0.035s	144855	69857	4
25	82	0.057s	228864	108540	4
30	114	10.25s	45048061	18018582	5
35	159	22.32s	88532767	35234219	5

These results show that the time it takes does seem to grow exponentially, since with more than 35 edges, and also, with the chromatic number greater to 7, the time increased exponentially compared to the ones on the table, which seems to confirm the time complexity cost of O(m^{E+1}).

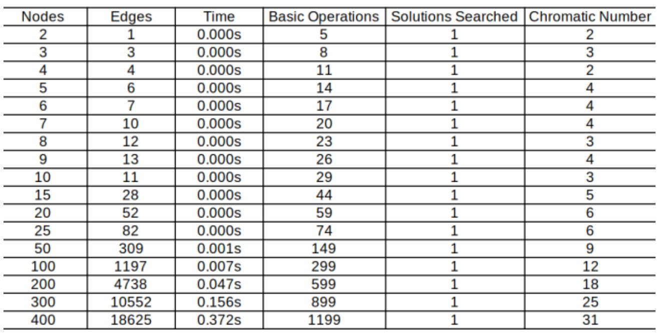
A big impact of executions times is the order in which the vertices are colored, if by chance, it follows an order that gives a valid solution, it may find a solution much faster than the worst-case scenario.

As for the number of basic operations and solutions/configurations searched, they grow as the number of vertices and the chromatic number of the graph increase, and are the obvious reason for the execution time to grow as it is observable.

B. Greedy Search

For greedy method, the algorithm complexity grows at a linear rate, therefore it is able to solve graphs with huge amount of vertices.

TABLE II
Results for Greedy Search Strategy



Nodes	Edges	Time	Basic Operations	Solutions Searched	Chromatic Number
2	1	0.000s	5	1	2
3	3	0.000s	8	1	3
4	4	0.000s	11	1	2
5	6	0.000s	14	1	4
6	7	0.000s	17	1	4
7	10	0.000s	20	1	4
8	12	0.000s	23	1	3
9	13	0.000s	26	1	4
10	11	0.000s	29	1	3
15	28	0.000s	44	1	5
20	52	0.000s	59	1	6
25	82	0.000s	74	1	6
50	309	0.001s	149	1	9
100	1197	0.007s	299	1	12
200	4738	0.047s	599	1	18
300	10552	0.156s	899	1	25
400	18625	0.372s	1199	1	31

From the table II we can clearly see a linear growth. The greedy method only tries one single solution/configuration, and therefore it is extremely faster and can solve problems with big inputs.

V. CONCLUSION

Overall, different algorithms were created in order to solve the given vertice coloring problem, and it was also produced a way to manage the graphs for testing purposes.

Analyzing the results, it was observed that they go hand-to-hand with the complexity analysis made prior to the testing phase, and they were simple to interpret after doing that analysis.

It can be concluded that for with NP-complete problems, the exhaustive algorithm will work on an acceptable amount of time for small inputs, but the time to find a specific solution, grows rapidly as the input size increases. The only way to solve these kinds of problems with big inputs, is with greedy method, however, they do not guarantee the optimal solution, but including heuristics in these algorithms helps to find solutions very close to the most optimal one.

Also note that I tried to implement one more exhaustive algorithm (Vizing Theorem) and 2 greedy algorithms (Order of Vertices Heuristic and Recoloring N Times Heuristic), as we can see in code, however I could not test them and get results due to some kind of problems.

REFERENCES

[1] Wikipedia contributors. (2022, September 15). Graph coloring. Wikipedia. https://en.wikipedia.org/wiki/Graph_coloring

[2] GeeksforGeeks. (2020, November 10). Graph Coloring. https://www.geeksforgeeks.org/graph-coloring-applications/

[3] Wikipedia contributors. (2022, May 31). Erdős–Rényi Model. Wikipedia. https://en.wikipedia.org/wiki/Erd%C5%91s%E2%80%93R%C3%A9nyi model

[4] Wikipedia contributors. (2021, November 16). Brute-force search. Wikipedia. https://en.wikipedia.org/wiki/Brute-force search

[5] GeeksforGeeks. (2022, September 23). m Coloring Problem — Backtracking-5. https://www.geeksforgeeks.org/m-coloring-problem-backtracking-5/

[6] GeeksforGeeks. (2022, July 13). Graph Coloring — Set 2 (Greedy Algorithm). https://www.geeksforgeeks.org/graph-coloring-set-2-greedy-algorithm/