Graph Chromatic Index

Mário Francisco Costa Silva, nMec: 93430

***Abstract* –The chromatic index problem, also known as the edge 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 Index, Edge Coloring, Time Complexity, Space Complexity, Greedy, Exhaustive, Heuristic**

I. Introduction

The chromatic index of G is the smallest number of colors needed to properly color the edges of a graph, such that no two edges sharing the same vertex have the same color.

In the following examples, we can visualize a simple graph with 6 edges and with a chromatic index of 5, and another graph that also has a chromatic index of 5 despite having 19 edges.

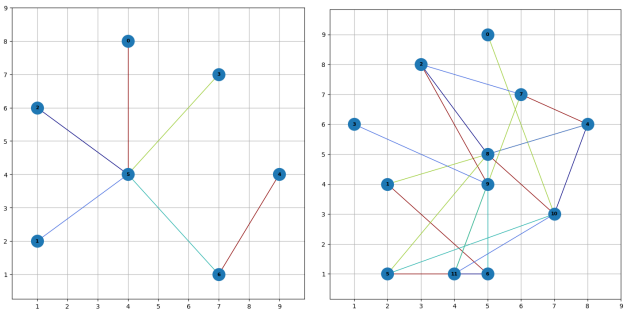

Fig. 1
Graphs with Chromatic Index of 5

Edge coloring may be used to schedule a round-robin tournament into as few rounds as possible so that each pair of competitors plays each other in one of the rounds. Or in fiber-optic communication, the path coloring problem is the problem of assigning colors (frequencies of light) to pairs of nodes that wish to communicate with each other, and paths through a fiber-optic communications network for each pair, subject to the restriction that no two paths that share a segment of fiber use the same frequency as each other. Paths that pass through the same communication switch but not through any segment of fiber are allowed to use the same frequency.

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>
        -v <Vizing Theorem>
        -p <Permutations>
        -gr <Greedy Search>
        -o <Heuristic to order Edges>
        -r <Number of Times to Recolor Graph: int>
        -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 edge 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 edges 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 edges stored in the attributes of the Graph object.

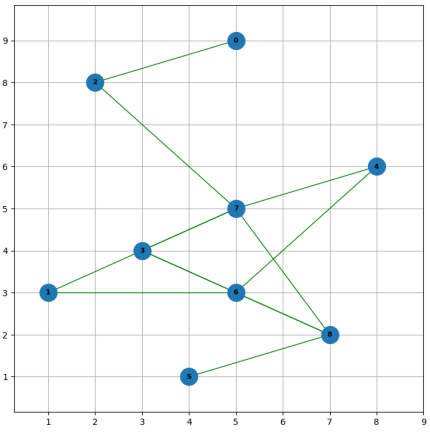

Fig. 3
Generated Graph

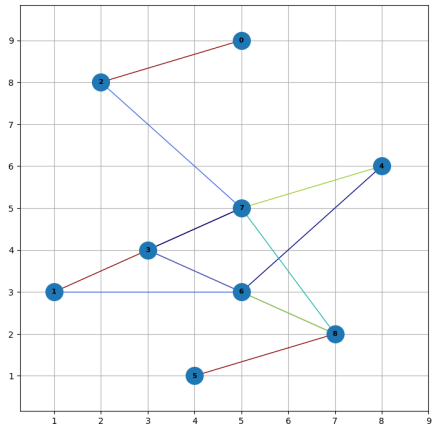

Fig. 4
Colored Graph

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 edge 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 edges starting with m equal to 1, and tries to assign the m colors one by one to different edges, starting from a random edge, and before assigning a color, it checks for safety by considering already assigned colors to connected edges (edges that share a node), if there is a color available then assign it and continue to assign edges 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 edge 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($m \cdot m^E$) or O(m^{E+1}) considering m to be the chromatic index, because every other number of colors smaller than the chromatic index 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 edges colored on each configuration it tries.

C.2 Vizing Theorem

In graph theory, Vizing's theorem states that every simple undirected graph may be edge colored using a number of colors that is at most one larger than the maximum degree $\triangle$ of the graph. At least $\triangle$ colors are always necessary, so the undirected graphs may be partitioned into two classes: "class one" graphs for which $\triangle$ colors suffice, and "class two" graphs for which $\triangle + 1$ colors are necessary. The theorem is named for Vadim G. Vizing who published it in 1964 [6].

By using this theorem, it is possible to start trying to color the graph with m equal to the maximum degree $\triangle$ of the graph and going for $\triangle$+1. This way it reduces heavily the search time, because it no longer tries to color with the previous numbers to $\triangle$.

The time complexity just like the previous one, but it only tries with 2 number of colors, so it will be O(2*m^E), which means it will be O(m^E), with. m being the chromatic index.

Given this theorem, it could also be said that without this theorem the time complexity is O($(\triangle + 1)^{E+1}$), and with it, O($(\triangle + 1)^E$).

The space complexity is the same as the previous one, O(E).

C.3 Permutations Based

The permutations based strategy is a naive approach to solving the edge coloring problem. It generates every possible order for the edges to be colored. Then it applies a simple algorithm for each order, like the greedy one, that it will be explained in the next section in more detail. After generating the permutations, it colors the graph and then checks if the new solution found uses a smaller amount of colors than the previous smallest one [7]. This way, it is guaranteed to find the best solution to the problem.

Here is a quick example on how the order of the edges to be colored have an impact.

Consider the following graph:

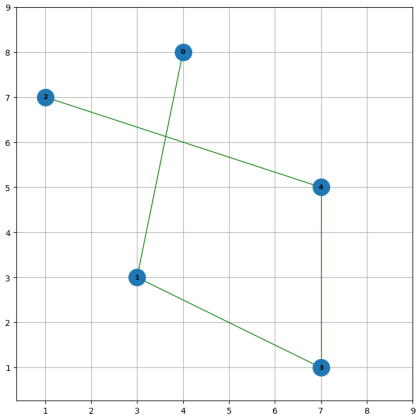

Fig. 5
Uncolored Graph Example

If we consider the order of edges the following: first the edge (0,1) and give it color 1, secondly (2,4) and give it color 1, then (3,4) and give it color 2 because (2,4) is connected and has color 1, and finally the edge (1,3) and give it color 3 because (0,1) and (3,4) are connected and have colors 1 and 2.

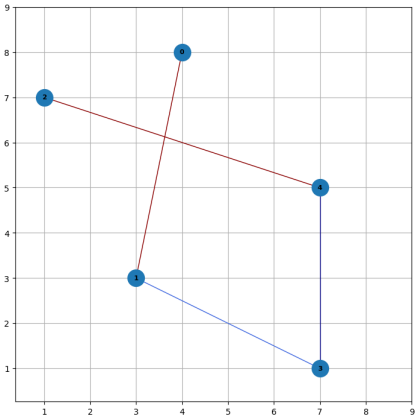

Fig. 6
Graph Colored with 3 Colors

In that order, the graph ends up with a chromatic index of 3, however this graph has actually a chromatic

index of 2. Let's try with other order of edges: first the edge (0,1) and give it color 1, secondly (1,3) and give it color 2, then (3,4) and give it color 1, and finally (2,4) and give it color 2.

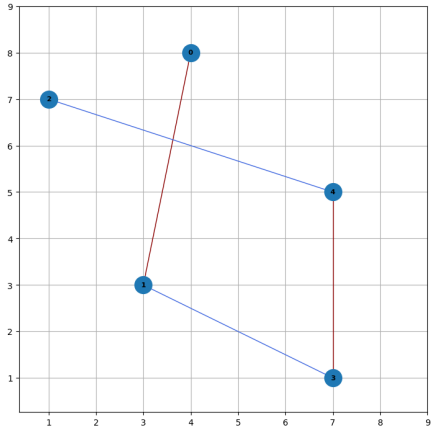

Fig. 7
Graph Colored with 2 Colors

This way, it is proved that the order in which the algorithm colors the edges has a big importance on the chromatic index.

It is possible to implement the Vizing Theorem on this strategy, whenever the solution found is equal to the maximum degree $\triangle$ of the graph, simply return it because there isn't any better solution. This will save some time, however for graphs of chromatic index equal $\triangle$+1, it wouldn't help at all.

The time complexity on this problem is O(E!·E), or just O(E!), since it does every possible permutation (order of edges) and colors them in a Greedy Strategy.

The space complexity is O(E), since it only colors each edge once, and it iterates over all possible orders of edges but doesn't store all of them in memory.

D. Greedy Search

The greedy algorithm, has mentioned before, only iterates over each edge once, and, therefore, only searches for one solution one time. When iterating each edge, checks for adjacent edges and colors the current edge with an available color, it simply repeats this process for each edge and in the end it is guaranteed a solution, however, not an optimal solution [8].

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

D.1 Order of Edges Heuristic

So the order in which the vertices are picked is important, as we have seen on the permutation based algorithm. Many people have suggested different ways to find an ordering that work better than the basic algorithm on average. In the node coloring problem, the most common approach is to consider nodes in descending order of their degrees, so, it was applied the same concept but for edges.

The heuristic used was the sorting of the edges from the one with the highest amount of connections to other edges to the lowest one. It still doesn't guarantee the best result, but the order of the edges have indeed an impact, as it is demonstrated on the results sections.

D.2 Recoloring N Times Heuristic

Another way to optimize the greedy algorithm is utilizing the greedy search with the heuristic and then perform a recoloring N times.

This way, it will run first with the edges ordered by the number of connected edges, and on the next N times, passed as an argument to the program, it will shuffle the list of edges, and therefore, they will be colored in different orders, and it will be stored the solution that gave the smallest chromatic index.

Given this strategy, there is an even bigger chance of finding the optimal chromatic index or at least approximate to it even more.

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 edges colors and coloring the edge, the number of solutions/configurations searched were also noted, and finally the chromatic indexes from the solutions found.

The number of nodes is showed because the algorithm loads graphs based on the number of nodes passed as an argument to the program and therefore serves as an identifier, however, for the analysis of the computational costs, the number of nodes aren't relevant for the algorithms that were tested since they are never iterated over.

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 Index
2	1	0.0 s	3	1	1
3	2	0.0 s	9	5	2
4	3	0.0 s	27	15	3
5	4	0.0 s	17	9	2
6	5	0.0 s	64	5	3
7	6	0.001 s	907	9	5
8	7	0.0 s	175	105	3
9	12	0.026 s	57654	23140	5
10	13	3.20 s	6912505	2316452	6
11	18	16.9 s	30323505	10120635	6
12	19	0.836 s	1623334	649503	5

On the graph with 22 edges, the program timed out (over 30 minutes).

These results show that the time it takes does seem to grow exponentially, since with more than 22 edges, and also, with the chromatic index 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 edges 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, this can be seen on the last two cases of the table.

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

B. M-Coloring Exhaustive Search with Vizing Theorem

The same strategy, but with the usage of the Vizing Thereom, is able to find solutions way more efficiently for a much greater number of edges and of chromatic indexes.

TABLE II
Results for M-Coloring Exhaustive Search with Vizing Theorem Strategy

Nodes	Edges	Time	Basic Operations	Solutions Searched	Chromatic Index
2	1	0.0 s	4	1	1
3	2	0.0 s	7	2	2
4	3	0.0 s	11	3	3
5	4	0.0 s	15	6	2
6	5	0.0 s	16	5	3
7	6	0.0 s	24	6	5
8	7	0.0 s	65	29	3
9	12	0.0 s	43	12	5
10	13	0.0 s	49	13	6
11	18	0.0 s	105	28	6
12	19	0.001 s	3719	1235	5
13	22	0.0 s	87	22	7
			...		
81	638	0.213 s	111631	9770	22
			...		
84	690	36.86 s	26190676	1746238	29
85	678	17.13 s	14121631	1129842	24
86	726	3.79 s	2754213	204152	26

However, despite being able to find solutions with such high number of edges and chromatic indexes, the algorithm fails completely if the chromatic index is equal to the maximum degree of a node plus 1 ($\triangle + 1$) because it will still have to do $\triangle^E$ in order to reach m=$\triangle$ + 1, therefore, it will time out just like the previous algorithm. Essentially, the only reason this is able to do some chromatic indexes of such high values is because of luck and also most likely there are multiple solutions for it with m = $\triangle$, so it finds one rather quickly.

One example of this, is the graph with 81 nodes, that has 638 edges and a chromatic index of either 22 or 23, since the highest node degree is 22. I am not able to be sure what is the chromatic index, because there may be a solution with 22 colors, but the program timeouts and will never even get to try with 23 colors. This exhaustive search algorithms are the only way to find out, since the greedy approaches don't give optimal values all time.

C. Exhaustive Search with Permutations

TABLE III
Results for Exhaustive Search with Permutations Strategy

Nodes	Edges	Time	Basic Operations	Solutions Searched	Chromatic Index
2	1	0.0 s	3	1	1
3	2	0.0 s	10	2	2
4	3	0.0 s	44	6	3
5	4	0.0 s	242	24	2
6	5	0.002 s	1562	120	3
7	6	0.014 s	11522	24	5
8	7	0.07 s	95762	5040	3

On the graph with 9 nodes and 12 edges, the computer also timed out (+30 minutes), it has to find a solution for 12!, which is, almost 500 million different orders.

This exhaustive method, since it is factorial, is the fastest growing one in terms of operations executed, and a jump of 5 edges is too much for it to handle, therefore, the results were as expected.

D. Greedy Search

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

TABLE IV
Results for Greedy Search Strategy

Nodes	Edges	Time	Basic Operations	Solutions Searched	Chromatic Index
2	1	0.0 s	2	1	1
3	2	0.0 s	5	1	2
4	3	0.0 s	8	1	3
5	4	0.0 s	11	1	3
6	5	0.0 s	14	1	3
7	6	0.0 s	17	1	5
8	7	0.0 s	17	1	4
9	12	0.0 s	35	1	5
10	13	0.0 s	38	1	6
11	18	0.0 s	53	1	7
12	19	0.0 s	56	1	6
13	22	0.0 s	65	1	7
			...		
81	638	0.063 s	1913	1	25
			...		
500	21600	53.67 s	64799	1	130
501	21834	58.58 s	65501	1	135
			...		
600	31295	130.64 s	93884	1	162

From the table IV 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.

D.1 Order by Number of Adjacent Edges Heuristic

This algorithm iterates the edges in a descendant order of number of adjacent edges.

TABLE V
RESULTS FOR GREEDY WITH ORDER OF EDGES HEURISTIC SEARCH STRATEGY

Nodes	Edges	Time	Basic Operations	Solutions Searched	Chromatic Index
2	1	0.0 s	3	1	1
3	2	0.0 s	6	1	2
4	3	0.0 s	9	1	3
5	4	0.0 s	12	1	2
6	5	0.001 s	15	1	3
7	6	0.0 s	18	1	5
8	7	0.0 s	21	1	3
9	12	0.0 s	36	1	5
10	13	0.0 s	39	1	6
11	18	0.0 s	54	1	6
12	19	0.0 s	57	1	5
13	22	0.0 s	66	1	7
			...		
81	638	0.064 s	1914	1	23
			...		
500	21600	55.10 s	64800	1	120
501	21834	55.65 s	65502	1	121
			...		
600	31295	131.27 s	93885	1	146

From these results it is possible to observe that, just like the previous one, the search time has a linear growth.

Also, the time and basic operations are very similar, since this one has one extra initial step of sorting the edges to be iterated.

Most importantly, it can be confirmed that it does get better results. For example, for the graph with 4 edges, this heuristic manages to find a smaller chromatic index than the greedy search without it, and for bigger graphs, with more edges, it finds chromatic indexes much smaller (16 fewer colors for graph with 600 edges).

D.2 Recolor N Times Heuristic

To test if this heuristic actually optimizes the results, it was executed the greedy search with the order heuristic (previous one) and each graph was recolored with N equal to 10, which means it will store the best result, the least amount of colors used from the solutions found with 10 different order of edges.

TABLE VI
RESULTS FOR GREEDY WITH ORDER OF EDGES AND RECOLORING 10 TIMES HEURISTICS SEARCH STRATEGY

Nodes	Edges	Time	Basic Operations	Solutions Searched	Chromatic Index
2	1	0.0 s	13	11	1
3	2	0.0 s	46	11	2
4	3	0.0 s	79	11	3
5	4	0.0 s	112	11	2
6	5	0.0 s	145	11	3
7	6	0.0 s	178	11	5
8	7	0.0 s	211	11	3
9	12	0.0 s	376	11	5
10	13	0.0001 s	409	11	6
11	18	0.0005 s	574	11	6
12	19	0.001 s	607	11	5
13	22	0.001 s	706	11	7
			...		
81	638	0.120 s	21034	11	22
			...		
500	21600	66.87 s	712780	11	108
501	21834	70.49 s	720502	11	110
			...		
600	31295	142.35 s	1032715	11	140

From these results, for the graphs with higher number of edges, it is clear that the solutions found were more optimized than without this heuristic.

The graph, with 81 nodes and 648 edges, which has a chromatic index of 22, that can be confirmed that it is the best solution from the results II of the exhaustive m-coloring with Vizing Theorem algorithm, has 25 with the greedy algorithm without any heuristics, and 23 with the ordering of edges heuristic, and with both the ordering and re-coloring heuristic it has a chromatic index of 22.

It can be confirmed that running the algorithm with different order of edges, increases the probability of finding a better solution, that can approximate the most to the optimal one.

V. CONCLUSION

Overall, different algorithms were created in order to solve the given edge 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 algorithms 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 methods, 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.

References

[1] Wikipedia contributors. (2021, August 15). Edge coloring. Wikipedia.
https://en.wikipedia.org/wiki/Edge_coloring

[2] GeeksforGeeks. (2021, August 11). Edge Coloring of a Graph.
https://www.geeksforgeeks.org/edge-coloring-of-a-graph/

[3] Wikipedia contributors. (2019, August 2). 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. (2021c, September 24). m Coloring Problem — Backtracking-5.
https://www.geeksforgeeks.org/m-coloring-problem-backtracking-5/

[6] Wikipedia contributors. (2021a, July 6). Vizing's theorem. Wikipedia.
https://en.wikipedia.org/wiki/Vizing%27s_theorem

[7] Greedy coloring in descending degree ordering is not always optimal. (2020, March 6). Mathematics Stack Exchange.
https://math.stackexchange.com/questions/3571196/greedy-coloring-in-descending-degree-ordering-is-not-always-optimal

[8] GeeksforGeeks. (2021a, July 29). Graph Coloring — Set 2 (Greedy Algorithm).
https://www.geeksforgeeks.org/graph-coloring-set-2-greedy-algorithm/