Finding the Chromatic Index of Large Graphs using an Improved Exhaustive Search Algorithm

Tomás Oliveira Costa <89016>, tomascosta@ua.pt

***Abstract* – This article's main objective is to analyse the problem of the chromatic index of a given graph, also known as the edge coloring problem. In this paper a solution is provided for the given problem on feasible execution time and number of basic operations. It is also provided a set of options for generation and visualization of the graphs under analysis.**

I. INTRODUCTION

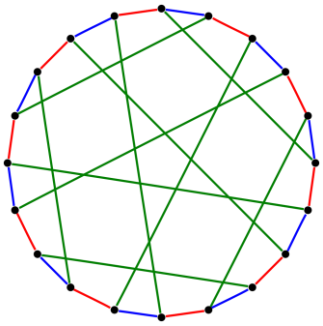
The problem of finding the chromatic index of a graph or also known as edge coloring problem, appeared with the four-color problem. The first paper dealing with edge coloring was written in 1880 and since then, many solutions have been given to this problem. It is an NP-hard problem, and for the general problem of coloring edges, the fastest known algorithms for it take exponential time.

On this work we take on an exhaustive search approach and reduce the load of the algorithm by not iterating certain elements of the created adjacency matrix of the graph.

II. DESCRIPTION OF THE PROBLEM

The edge coloring problem states that, for a given graph G, with n vertices and m edges, the chromatic index for that given graph is the minimum amount of colors needed to paint all edges, with the restriction that no adjacent edges can have the same color.

Given the information above, we start visualizing the problem with a simple solution, like the figure below:



As we can visualize, there is no single node with a repeated color for its adjacent edges. But, even for a problem with a dimension of 20 nodes, it is possible to paint every edge with just 3 distinct colors.

This problem must not be confused with the graph coloring problem, which instead of coloring the edges, you color the vertices, that leads to different solutions, and on this work we focused on solving the edge problem.

III. DESCRIPTION OF THE SOLUTION

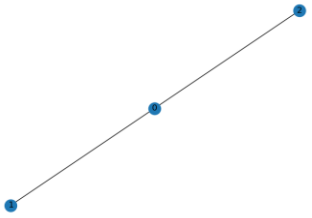
To better understand the solution, I advise a careful reading through the code, which is written in python3 and therefore, very easy to understand.

The graph is represented with an adjacency matrix, and therefore it was important to create a structure that would represent the graph in itself. For that, we created a graph class and assigned it a few methods. The methods **add_edge(edge1, edge2)** serves to create the connection from edge1 to edge2, and consequentially that's the same edge from edge2 to edge1, since the graph is undirected.

After the adjacency matrix has been created, I took a few important notes, such as:

- The diagonals have no value, since there is no edge connecting the vertice to itself
- Since the matrix is mirrored on the diagonal, we only need to iterate either the top side of the matrix, or the bottom one. That reduces the values for up to half.
- We only need to look at values where there is an edge, and therefore 0s in the adjacency matrix will also be 0s in the color matrix

After these notes were taken into consideration, we can simply iterate over the rows of the matrix, then iterate over the columns and only look at values where the adjacency matrix has a non-zero value.



So, for this graph above, we would have the vertice 0 with 2 edges connecting it to the other 2 vertices.

For this example, the adjacency matrix would look like this:

$$G = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

And after processing, the color matrix (let's call it C) should look something like this:

$$C = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 0 & 0 \\ 2 & 0 & 0 \end{bmatrix}$$

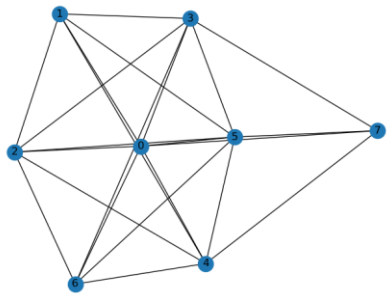
I treat the first row of the matrix with a special condition, because there is barely any restriction when you are at the first node, since your neighbours haven't been colored yet. So, for this node, we simply start coloring each of its edges with the next available color.

With the notes taken from above, for this simple example the algorithm would only iterate over 2 elements, which are indeed all number of edges present in the problem.

After the coloring of the first row is done, it should already have some of the colors matrix filled out, or all of them if the problem was as the above. But, for more complex graphs, there is high connectedness between vertices, therefore it needs to keep iterating the matrix and for each edge it looks at, it analyses the vertices that create that edge and store the used colors by both of those vertices, since neither of those colors can be used again by the current edge. After it has those colours, it grabs the minimum value from the available colors.

After it is done filling up the colors matrix, it picks the maximum value from its matrix

For a slightly more complex graph, as the one below, I determined that the chromatic index is highly influenced by the connectedness of graphs and how they are connected, since we can have a 20 node graph with 3 colors, but also a 3 node graph with 3 colors.



This is a graph with 8 nodes/vertices and 23 edges that was randomly generated by the code, with the possibility to visualize the graph included (more of the generation on IV). The algorithm assigned a chromatic index of 7 to this graph and assigned it in 0.113 milliseconds.

```
(base) tom1k@DESKTOP-OJISSPF:/mnt/d/Coding/WSL/AA_Proj1/code$ python3 main.py -g 8 -t
Created Graph with 8 nodes and 23 edges in 0.152 ms
Cromatic index for given graph is: 7
Time elapsed for coloring: 0.113 ms
(base) tom1k@DESKTOP-OJISSPF:/mnt/d/Coding/WSL/AA_Proj1/code$
```

IV. TESTING TOOLS FOR ANALYSIS

To assist the process of testing the algorithm and visualize its results, a getopts approach similar to C getopts was implemented. These options provide two special parameters:

- g <int> - generate a random matrix with <int> nodes
- t - serves as a testing flag, has no argument

A. Getopt G

Serves as a generator for a random matrix. This option takes in an integer as argument which is the number of nodes, the edges are randomly created for each node.

This allows the user to create very large graphs in a matter of seconds, which is very useful for comparison between computational use of the algorithm.

B. Getopt T

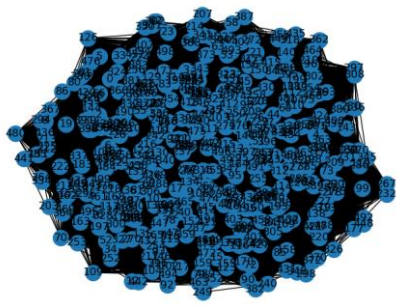
If this option is present, but not the generator option, the program conducts tests for three given graphs with known chromatic indexes, some other graphs were tested, but were of similar size and therefore not a great addition to the testing set.

This will output the given tests and output its number of basic operations and the time elapsed for coloring each one, as follows.

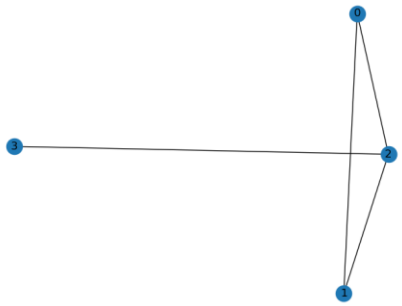
```
Conducting extra tests:

Cromatic index for given graph is: 4

Time elapsed for coloring: 0.032 ms
Number of basic operations: 7
Chromatic Index calculated correctly? True

Cromatic index for given graph is: 3

Time elapsed for coloring: 0.015 ms
Number of basic operations: 3
Chromatic Index calculated correctly? True
```

For extra testing, we can enable both the generator (g) and testing (t) options. This will lead to the testing of the generated matrix and, since we don't know the confirmed solution to the generated matrixes, I included a drawing of the graph to the folder **images**, in order to manually

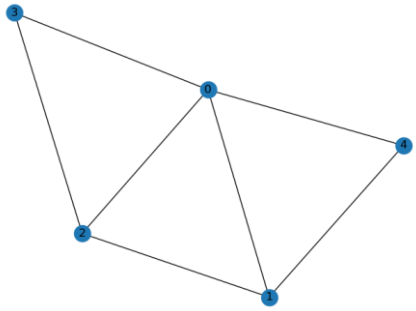
analyse simpler graphs and confirm the algorithm was working properly.
The drawing of the graph is only useful to manually determine the chromatic index of a given graph, or to determine if the graph is well connected. As for example, we see that from a graph with 500 nodes its impossible to extract valuable information.



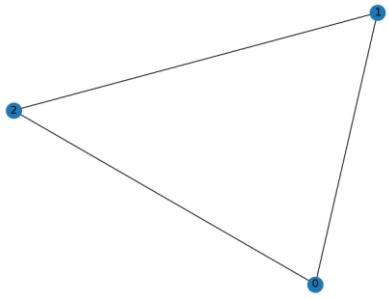
: C. Visualization of know graphs for testing



Chromatic index of 3



Chromatic index of 4



Chromatic index of 3

V. RESULTS AND DISCUSSION

The comparison between different graphs is more easily visualized through a table, so I created the following table:

Nodes	Edges	Basic Operations	Time (ms)
2	1	1	0.010
3	2	2	0.036
5	8	8	0.037
10	31	31	0.187
20	122	122	0.829
100	3603	3603	48.207
500	93155	93155	5244.014
1000	381339	381339	45800.912
2000	1506649	1506649	536486.104

As its observable from the results above, the algorithm is capable of some really good results, but since the complexity is bounded by a polynomial, after a given number of nodes we start getting execution times that are too demanding.

In my case, I was able to test up to **2000 nodes** and **1.5 Million edges**, but these results took a little under 10 minutes, which I think is way too much, and therefore I would say the program is able to give results to anything under 1000 nodes in a quick time.

This batch of tests is present in the following figure:

```
(base) tom1k@DESKTOP-QJISSPF:/mnt/d/Coding/WSL/AA_Proj1/code$ python3 main.py -g 5 -t
Created Graph with 5 nodes and 8 edges in 0.072 ms
Cromatic index for given graph is: 5

Time elapsed for coloring: 0.037 ms
Number of basic operations: 8
(base) tom1k@DESKTOP-QJISSPF:/mnt/d/Coding/WSL/AA_Proj1/code$
(base) tom1k@DESKTOP-QJISSPF:/mnt/d/Coding/WSL/AA_Proj1/code$ python3 main.py -g 10 -t
Created Graph with 10 nodes and 31 edges in 0.248 ms
Cromatic index for given graph is: 10

Time elapsed for coloring: 0.187 ms
Number of basic operations: 31
(base) tom1k@DESKTOP-QJISSPF:/mnt/d/Coding/WSL/AA_Proj1/code$ python3 main.py -g 20 -t
Created Graph with 20 nodes and 122 edges in 0.943 ms
Cromatic index for given graph is: 18

Time elapsed for coloring: 0.829 ms
Number of basic operations: 122
(base) tom1k@DESKTOP-QJISSPF:/mnt/d/Coding/WSL/AA_Proj1/code$ python3 main.py -g 100 -t
Created Graph with 100 nodes and 3603 edges in 63.183 ms
Cromatic index for given graph is: 111

Time elapsed for coloring: 48.207 ms
Number of basic operations: 3603
(base) tom1k@DESKTOP-QJISSPF:/mnt/d/Coding/WSL/AA_Proj1/code$ python3 main.py -g 500 -t
Created Graph with 500 nodes and 93155 edges in 7029.788 ms
Cromatic index for given graph is: 583

Time elapsed for coloring: 5244.014 ms
Number of basic operations: 93155
(base) tom1k@DESKTOP-QJISSPF:/mnt/d/Coding/WSL/AA_Proj1/code$ python3 main.py -g 1000
Created Graph with 1000 nodes and 381339 edges in 60503.992 ms
Cromatic index for given graph is: 1216

Time elapsed for coloring: 45800.912 ms
Number of basic operations: 381339
(base) tom1k@DESKTOP-QJISSPF:/mnt/d/Coding/WSL/AA_Proj1/code$ python3 main.py -g 2000
^[[D^[[CCreated Graph with 2000 nodes and 1506649 edges in 506017.541 ms
Cromatic index for given graph is: 2393

Time elapsed for coloring: 536486.104 ms
Number of basic operations: 1506649
```

VI. CONCLUSIONS

While doing a post-analysis for solutions to the edge coloring problem, I found several similar solutions that

didn’t have into account all of the considerations, such as [1], that I had. So I am really confident with the results and think the solution is given within a reasonable amount of basic operations and execution time.

REFERENCES

[1] F. Salama: New algorithm for calculating chromatic index of graphs and its applications, Journal of the Egyptian Mathematical Society, (2019) 27:18.

[2] Guantao Chen: The Graph Edge Coloring, Department of Mathematics and Statistics Georgia State University Atlanta, Georgia.

[3] GeeksForGeeks: Edge Coloring of a Graph, 01-11-2018