Shyni *et.al.* Edge Coloring Graph Implementation

[1]Eduardo Pinhata,
[2]Rafael P Cassetta
[1,2]CMCC, Federal University of ABC,
Santo André, São Paulo, Brazil

e-mail: [1] eduardo.pinhata@ufabc.edu.br,
[2] cassetta.rafael@ufabc.edu.br

18/07/2025

Abstract - This is a report of an implementation and execution results of the algorithm available in the article "Edge coloring of graph using edge adjacency matrix", by Shyni C., Ramachandran T. and Vijayalakshmi V.. [1].

***Keywords*:** graph, coloring, theory of computation

1 INTRODUCTION

One common problem in graph theory is the coloring problem. There are the vertices and edge coloring variants. The edge coloring problem is to determine the minimum number of colors to color the edges, so that the adjacent edges have different colors. An edge a_1 is adjacent to an edge a_2 if they are both linked to the same vertex. An edge a_3 is not adjacent to a_1 if the vertexes that they connect are different.

The Shyni *et al.* [1] article provided an algorithm to solve the problem. It was implemented and the results are described here.

2 IMPLEMENTATION

Kotlin was used to implement the algorithm and there were some structures used during the development of the work while we were optimizing the algorithm that would be interesting to cite here. The main challenges in this implementation was two: represent the adjacency matrix, and find the maximum null matrix.

2.1 Adjacency Matrix

During the implementation, the adjacency matrix was represented in three ways: two-dimensional array, HashMap with adjacent edges only, map of nodes to edges, and edges to nodes.

Two-dimensional array

The most intuitive way to represent the adjacency matrix was using a two-dimentional array data structure with n rows, and n columns, where n is the number of edges in the graph. The value 0 in row i and column j would represents that the edge e_i is not adjacent to the edge e_j, while the value 1 would represent that one edge is adjacent to the other.

Listing 1: Full matrix example

```
private var adjacentVertexes: Array<IntArray>
```

Given the following graph G_1, described by the figure 1.

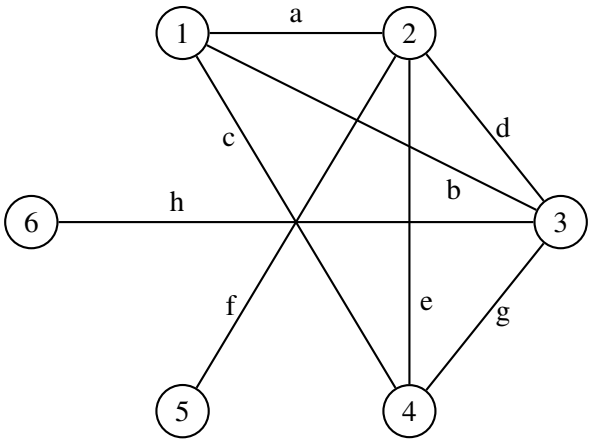

Figure 1: Representation of the undirected graph G_1 with 6 vertices.

The matrix M_1 describes this graph, and the matrix M_1a, describes the adjacency matrix of this graph.

$$M_1 = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 2 & 1 & 0 & 1 & 1 & 1 & 0 \\ 3 & 1 & 1 & 0 & 1 & 0 & 1 \\ 4 & 1 & 1 & 1 & 0 & 0 & 0 \\ 5 & 0 & 1 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 1 & 0 & 0 & 0 \end{array}$$

$$M_1a = \begin{array}{c|cccccccc} & a & b & c & d & e & f & g & h \\ \hline a & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ b & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ c & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ d & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ e & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ f & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ g & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ h & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \end{array}$$

By representing this whole matrix as a bidimensional array in Kotlin, it will require $n.n$ times the type of array size of memory. For small matrixes, this is no problem for the computer, but for the instances ran in this exercice, we can easily use over 30 Gb of RAM memory using this matrix.

Adjacency HashMap

An optimization made in the adjacency matrix was to eliminate the 0s, storing only the adjacent edges. HM_1a describes the HashMap that stores the adjacency information from graph G_1.

$$HM_1a = \begin{array}{c|c} a & b, c, d, e, f \\ b & a, c, d, g, h \\ c & a, b, e, g \\ d & a, b, e, f, g, h \\ e & a, c, d, f, g \\ f & a, d, e \\ g & b, c, d, e, h \\ h & b, d, g \end{array}$$

Although the two-dimensional matrix could be a binary matrix, in a sparce graph, the HashMap structure can save memory, allocating $n.m$ spaces of memory, where n is the number of edges and $m <= n$ is the number of edges that are adjacent to each vertice.

Despite the efficiency of the HashMap structure, some instances, for example *dsjc1000.9.col.txt*, required more than 30Gb of RAM memory. Due to the hardware restrictions, we needed to improve the structure to save the adjacency matrix.

Map of nodes to edges and edges to nodes

In the instances used in this exercise, while the edges amount were really high, the amount of vertices were not as big. So, to get the information of the adjacency, two data structures were combined: one matrix of nodes to edge and one HashMap of edges to nodes.

The first structure is a two-dimensional array that makes a mapping of each pair of nodes to its corresponding edge. The matrix M_2 contain the nodes to edge matrix for graph G_1.

$$M_2 = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & - & a & b & c & - & - \\ 2 & a & - & d & e & f & - \\ 3 & b & d & - & g & - & h \\ 4 & c & e & g & - & - & - \\ 5 & - & f & - & - & - & - \\ 6 & - & - & h & - & - & - \end{array}$$

The second structure is a mapping of each edge to the vertices it link. This structure is better than the Adjacency HashMap because we'll only have two element for each edge line. The Adjacency HashMap, in dense graphs, could have almost the same amount of edges of the graph, which in some tested instances consumed a lot of memory. The HashMap M_3 contains the list of nodes per edge of graph G_1.

$$M_3 = \begin{array}{c|c} a & 1, 2 \\ b & 1, 3 \\ c & 1, 4 \\ d & 2, 3 \\ e & 2, 4 \\ f & 2, 5 \\ g & 3, 4 \\ h & 3, 6 \end{array}$$

The way to find the adjacent edges of edge e_1 is to get, fromM_3, the vertices that e_1 connects. Then, for each vertex found, we get, from M_2, the list of edges connected to this vertex. We remove the duplicated edges and the e_1 if present. In this way, we have the list of e_1's adjacent edges, using less space than the previous solution.

Structure	**RAM memory used**
Two-dimensional array	Over 30Gb
Adjacency hashmap	Over 30Gb
Map of nodes to edges and edges to nodes	1,8Gb

Table 1: Comparison of graph representations by RAM memory to run the instance *dsjc1000.9.col.txt*

2.2 Find Max Null Matrix

An important step of the Shyni *et al.* algorithm is the max null matrix. They did not provide any method to find it, so we created an heuristic to find a null matrix that is not necessarily the max null matrix. This is one improvement point from the implementation that we have created.

Listing 2: Function to compute Null Matrix

```
fun getNullMatrixRows(row: Int, alreadyProcessedRows: HashMap<Int,
    Boolean>): ArrayList<Int> {
    var nullCols = getNullColumns(row)
    var nonNullColSizeSorted = getNonNullColumnsSizeSorted(nullCols)
    var acceptedRows = arrayListOf(row)
    var rejectedRows: MutableMap<Int, Boolean> =
        alreadyProcessedRows.toMap().toMutableMap()

    nonNullColSizeSorted.keys.forEach{ col ->
        if (rejectedRows[col] == false){
            acceptedRows.add(col)
            getAdjacentEdges(col).forEach{ c -> rejectedRows[c] = true}
        }
    }

    return acceptedRows
}
```

We create a null matrix related to a specific edge e_1. First we take all the edges E_j non-adjacent to e_1. Then we sort the E_j edges in ascending order of edge adjacency count. Starting from the first sorted edge e_{j1}, we add it to a list of accepted edges. For each e_{j1} adjacent edge, that is in $e_j \in E_j$ and is not in the rejected edge list, add it to the rejected edge list. Then add the next sorted edge $e_j \in E_j$, and it is processed following the steps above. The accepted edges will form the null matrix used in the algorithm.

The repository 25Q2-computer-theory/Colorize contains the implementation used in this exercise.

3 EXPERIMENTS

The experiments were run in a Macbook Air M3, 2024, with 16Gb of RAM and running the maxOS Sequoia version 15.5 as Operational System. Although there is no parallelization, the instances were run one per time, individually.

The instances used were dsjc250.5, dsjc500.1, dsjc500.5, dsjc500.9, dsjc1000.1, sjc1000.5, dsjc1000.9, r250.5, r1000.1c, r1000.5, dsjr500.1c, dsjr500.5, le450 25c, le450.25d, flat300 28 0, flat1000 50 0, flat1000 60 0, flat1000 76 0, latin square, C2000.5 e C4000.5, available in DIMACS Graphs: Benchmark Instances and Best Upper Bounds.

4 RESULTS AND DISCUSSION

The following table contains the number of colors required to colorize the graph of each instance mentioned in Experiments section.

Filename	**Colors Number**	**Execution Time (ms)**
dsjc250.5.col.txt	189	5.666
dsjc500.1.col.txt	88	1.122
dsjc500.5.col.txt	377	70.930
dsjc500.9.col.txt	659	8.792.908
dsjc1000.1.col.txt	159	14.621
dsjc1000.5.col.txt	735	2.006.689
dsjc1000.9.col.txt	1288	15.931.131
dsjr500.1c.col.txt	698	1.466.743
dsjr500.5.col.txt	461	42.520
flat300_28_0.col.txt	219	10.611
flat1000_50_0.col.txt	712	1.784.328
flat1000_60_0.col.txt	710	4.067.076
flat1000_76_0.col.txt	701	1.923.028
latin_square.col.txt	memory	–
le450_25c.col.txt	180	1.820
le450_25d.col.txt	167	2.170
r250.5.col.txt	222	3.667
r1000.1c.col.txt	memory	–
r1000.5.col.txt	918	882.594
c2000.5.col.txt	1481	35.879.083
c4000.5.col.txt	memory	–

Table 2: Execution times and color counts for various graph coloring instances

Each instance was executed only once. Since the algorithm used is deterministic and we were not really measuring the time to process it, only one execution was enough.

There are two instances that the algorithm couldn't get a solution. The algorithm implemented had problem dealing with the latin_square.col.txt instance. While it was not efficient enough to run the c4000.5.col.txt.

Further improvements in the algorithm would require a fix to be able to process the latin_square.col.txt, and a performance improvement to run c4000.5.col.txt in a viable time. Using parallel processing to get the null matrix is a candidate get this improvement.

5 CONCLUSIONS

We were able to implement one version of the Shyni *et. al.* algorithm. We had to think about the structure to store the adjacency matrix, so all instances were able to be executed in the available computer. An algorithm to find the maximum null matrix was not implemented, but instead, one that finds one null matrix. This led to solutions that were not as good as the one obtained when we use the max null matrix.

Further improvements in the implementation are needed in order to run one specific instance, and to improve the efficiency of the algorithm by using parallel processing.

REFERENCES

[1] C. P. Shyni, T. Ramachandran, and V. Vijayalakshmi, "Edge coloring of graph using edge adjacency matrix," *International Journal of Creative Research Thoughts*, vol. 11, no. 1, b128–b134, Jan. 2023, UGC-CARE Journal, Volume 11 Issue 1, January 2023, ISSN: 2320-2882. DOI: 10.1729/Journal.33068.