CS301 – Term Project

Project Report

Group Members:

Zeynep Tandoğan -25200
Mert Gürsu Gökçen - 26760
Ulaş Eraslan - 25058
Sena Yapsu - 24874

Semester: 2020 Fall
Instructor: Hüsnü Yenigün
Sabancı University
Faculty of Engineering and Natural Sciences

Table of Contents

1)Problem Description 3

i. Definition 3

ii.Examples 4

iii. Edge coloring problem is NP Complete 6

2) Algorithm Description 14

i.Define the Algorithm 14

ii.Implementation 15

3) Algorithm Analysis 18

i. Correctness of the algorithm 18

ii. Complexity analysis 19

4)Experimental Analysis 21

i. Running time analysis 21

ii. Correctness 27

5) Testing 28

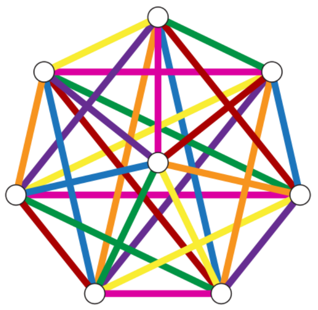
6) Discussion 33

1)Problem Description

i. Definition

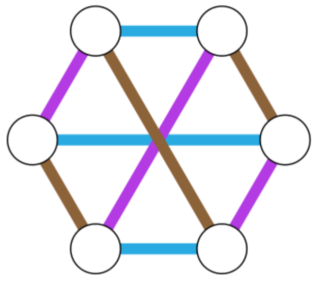
The edge coloring problem is a problem that aims to color all edges of a given graph with the minimum number of colors so that no two adjacent edges receive the same colour. Two edges are adjacent to each when they are connected to the same vertex.[1]

More formally, this problem can also be described like in the following statement. Let G = (V, E) be a graph. A k-edge-coloring of G is a function f : E(G) → C, for some set S with cardinality |S| = k. What is the minimum number of k that satisfy a function f : E(G) → C such that f(e) ≠ f(e') whenever edges e and e' are adjacent in G?[2]



[3]

Geometric construction of a 7-edge-coloring of the complete graph *K*8. Each of the seven color classes has one edge from the center to a polygon vertex, and three edges perpendicular to it.



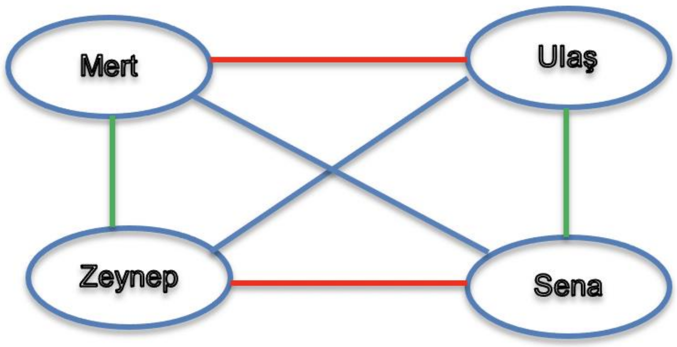
A complete biparite graph $K_{3,3}$ with each of its color classes drawn as parallel line segments on distinct lines.

[1] Cai, L., & Ellis, J. (1988). *Discrete applied mathematics*. Amsterdam: North Holland.
[2] Beseri, T. (n.d.). Edge Coloring of a Graph.
[3] https://www.wikiwand.com/en/Edge_coloring

ii.Examples

The edge-coloring problem, which is also called the chromatic index problem is a problem that is used and encountered in real life frequently. The most encountered cases from our daily lives are scheduling problems, below we have explained two such examples of scheduling problems.

Example 1: A **round-robin tournament** (or **all-play-all tournament**) is a competition in which each contestant meets all other contestants in turn.[4] Assume a chess tournament where Mert, Zeynep, Ulaş and Sena are the players and they all have to play with each other but a player can only play a single game in a given round. Then the minimum number of rounds becomes an edge coloring problem where nodes represent players and edges represent rounds and each same colored edge is the same round.

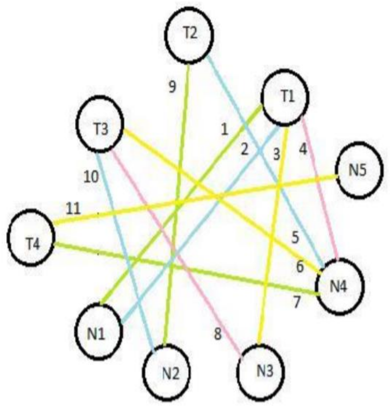


Example 2: [5] Let's consider another time table scheduling problem. Suppose that we have n many teachers in a school and there are m subjects that should be taught. Let's put two constraints before matching the subjects with professors. At one period, a teacher can only give one course and that each subject can be taught by a maximum one teacher. Let's take the number of teachers as 4 and subjects as 5 for simplicity and suppose that we have the information that which teacher can give which subjects in a table.

[4] https://en.wikipedia.org/wiki/Edge_coloring#Applications
[5] Ganguli, R., & Roy, S. (2017). A Study on Course Timetable Scheduling using Graph Coloring Approach. *International Journal of Computational and Applied Mathematics*.

Periods P	N1	N2	N3	N4	N5
T1	2	0	1	1	0
T2	0	1	0	1	0
T3	0	1	1	1	0
T4	0	0	0	1	1

The number of colors that will be used will show the number of time slots that are needed for teaching activities and our aim is to minimize the number of time slots. For this, first the table will be converted to a graph and then according to the table, a proper edge coloring will be done.



PINK	GREEN	BLUE	YELLOW
V4	V1	V2	V3
V8	V7	V5	V6
	V9	V10	V11

As it is mentioned above, the colors represent the periods. It is seen that teachers give maximum 3 courses simultaneously in this case.
The resulting table:

Period 1	Period 2	Period 3	Period 4
T1-N4	T1-N1	T1-N1	T1-N3
T3-N3	T4-N4	T2-N4	T3-N4
	T2-N2	T3-N2	T4-N5

iii. Edge coloring problem is NP Complete

Decision version of edge coloring problem is: “Given a graph G(V,E) and an integer N, is it possible to find a number of different colors smaller than or equal to N used for edges such that each adjacent edge is colored with no same color? ”

1. Edge coloring is NP
2. An NP Complete problem can be reduced to Edge Coloring in polynomial time.

1. Edge Coloring is NP

If any problem is in NP, then, given a solution to the problem and an instance of the problem (a graph G and a positive integer N), we will be able to verify (check whether the solution given is correct or not) the guess in polynomial time.

```
verifyGuess( G(V,E),N){
colors_used = []
    for each vertex in V{
        colors_used_for_vertex = []
        for each edge in E {
            if v is endpoint of edge{
                if edge.color in colors_used_for_vertex{
                    print(“Wrong solution”);
                    end the algorithm;
                }
                else{
                    insert edge.color to colors_used_for_vertex;
                    if edge.color not in colors_used then insert edge.color
                to colors_used;
                }}}}
```

```
          //for loop ends here

          if colors_used.size() <=N

                    print("Correct Solution");

          else print("Wrong Solution");

}
```

This algorithm is O(EV) time so edge coloring problem is polynomial time verifiable, hence edge coloring problem is in NP.

2. An NP Complete problem can be reduced to Edge Coloring in polynomial time.

We will show that it is NP Complete to find if there is a number of colors smaller than or equal to N to color the edges of the given graph such that no adjacent edges are colored with the same color.

We will first introduce Vizing's Theorem:The chromatic index is either k or k+1, where k is the maximal degree of the vertices of the graph.[6]

By acknowledging the correctness of Vizing's Theorem we can conclude that:

1. If the given number N <k, then it is not possible to color the edges of the given graph with N distinct colors.
2. If the given number N >=k+1, then it is definitely possible to color the edges of the given graph with N distinct colors.

The challenging part of this problem is to decide for when the given integer N is equal to k, it is NP complete to decide whether the number of different colors used is equal to N(=k).

Since we already acknowledged Vizing's Theorem, we know that for a given graph G with maximal degree of vertices is k, we must show that it is NP complete to decide whether the chromatic index is k or k+1.

<u>Theorem:</u> For any fixed k, the problem of deciding whether the chromatic index of a regular graph of degree k is k or k+1 is NP complete.

[6] https://en.wikipedia.org/wiki/Vizing%27s_theorem

In order to prove this theorem, we will reduce the well-known 3-SAT problem to our edge coloring problem in polynomial time.

A set of clauses C={c_1,,.........,c_n} in variables u_1,,.........,u_m is given each clause c_j consist of three literals l_{j1}, l_{j2},,l_{j3} where a literal l_{jk} is either a variable u_i or u_i'.

Proof of the theorem for k=3 (Holyer's proof): [7]

Let a,b be two edges and let a=b denote the fact that they are colored in the same colors, otherwise will be denoted as a!=b.

"Lemma 1:

In any 3 coloring of graph H, these hold:

1. Either a=b or c=d.
2. a=b(c=d) is equivalent to c!=d ∩ c!=e ∩ d!=e (a!=b ∩ a!=e ∩ b!=e)

B) Any coloring of a, b,c,d,e in 3 colors which satisfies 1 and 2 above completed to a 3 coloring of H."[8]

For the proof of this theorem, Holyer a Graph G consisting of 3 components:

1. Inverting Component

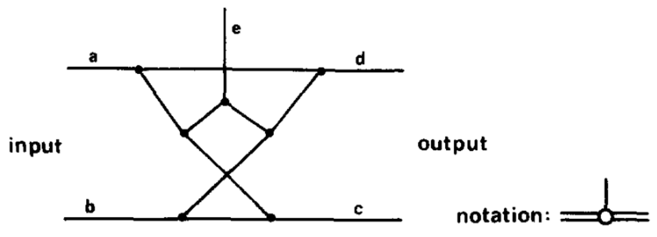


FIG. 1. An inverting component.

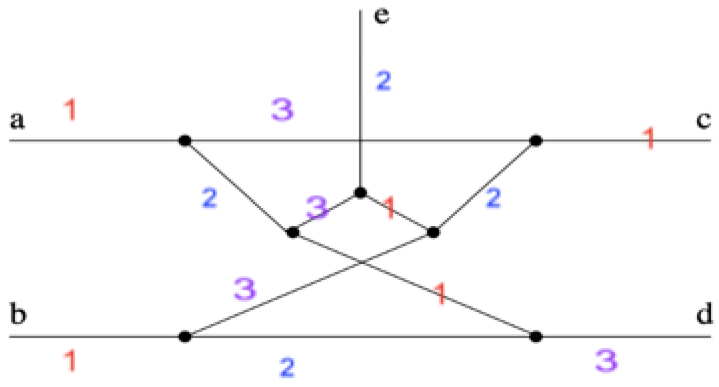
The inverting component has 2 input edges and 3 output edges, in which we have the following properties by the lemma 1:

1. If the color of input a is equal to input b, then colors of outputs c,d,e are distinct.
2. If the color of input a is not equal to input b, then colors of outputs c and d are equal, and the color of output e is distinct from the inputs and outputs.

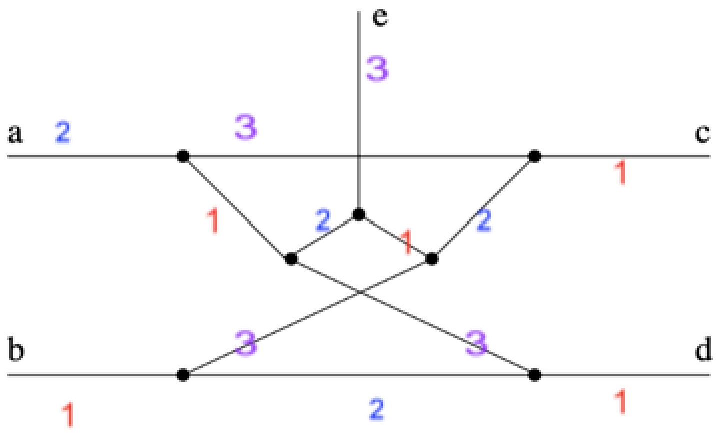
Example 1: When a=b, we will assign the adjacent edges of a and be different colors from them. Then the same will be done to the adjacent edges of the newly colored

[7] Holyer, I., 1981. The NP-Completeness of Edge-Coloring. *SIAM Journal on Computing*, 10(4), pp.718-720.
[8] https://moodle.umontpellier.fr/pluginfile.php/326326/course/section/68010/LevenGalil.pdf

edges, Then the remaining edges are colored in the same manner until there is no edge left uncolored. One can see that in order to preserve the rule of not coloring the adjacent edges with the same color; edges c, d and e must have distinct colors.



Example 2: When a!=b, we will assign the adjacent edges of a and be different colors from them. Then the same will be done to the adjacent edges of the newly colored edges, Then the remaining edges are colored in the same manner until there is no edge left uncolored. One can see that in order to preserve the rule of not coloring the adjacent edges with the same color, edges c and d must be assigned with the same color and edge e must have a distinct color then a, b.



3. Variable Setting Component

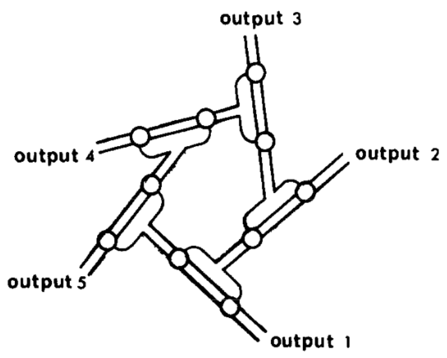
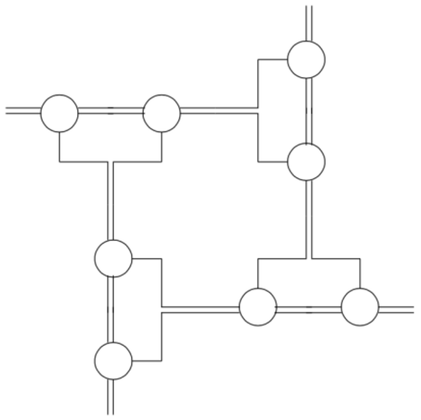


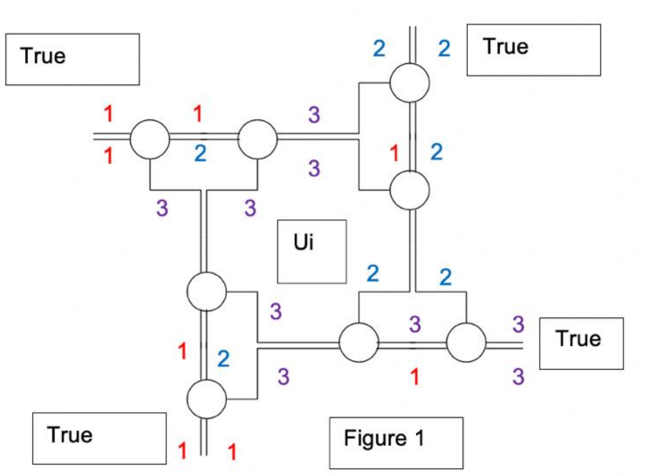
FIG. 2. A variable-setting component with five outputs.

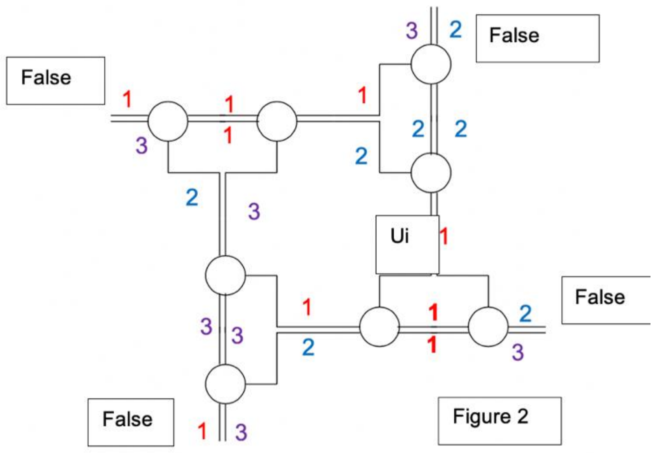


A variable setting component with 4 outputs.

A variable setting component is formed of 2n inverting components where n is the number of output pairs of the variable setting component, each variable setting component represents a literal in the 3-SAT problem and the number of outputs (n) is the number of the occurrence of the literal (including the occurrence of the negation of the literal). One may check that all the outputs of a given variable setting component is either true or false, an output pair is true if both edges have the same color, if they have different colors it's false. Without loss of generality, we will assume that each literal occurs more than once in the 3-SAT problem, if not one can generate another clause simply. (ex: if u1 occurs once in P, we can simply say P=P $\wedge$ (u1 V u1 V u1')).

Let us see by an example how a variable setting component works. Let the variable Ui occur 4 times in the 3-SAT formula we want to solve (including Ui'). If Ui =True, we will see the variable setting component in figure 1 and If Ui =False, we will see the variable setting component in figure 2.





4. Satisfaction Testing Component

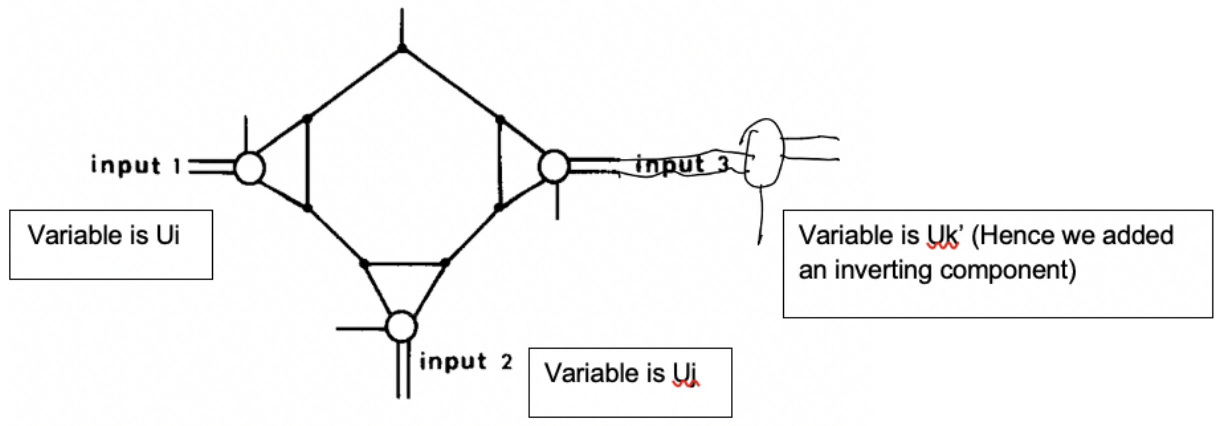


FIG. 3. A satisfaction-testing component.

A satisfaction setting component represents a conjunction clause in the 3-SAT problem, and we connect the outputs of the variable setting component to the inputs of the satisfaction testing component, if the associated literal with input is the inverted literal, then an inverting component is inserted between the variable setting component and the satisfaction setting component. The edges of satisfaction testing component can be colored with 3 colors if and only if the conjunction clause gives output as True. For example, let C1 = (u1’ V u2 V u1’), we will connect each with the variable with the satisfaction component. Here C1 is true (Hence 3 colorable) if u1=false and u2=true (figure 3), and C1=false (Hence not 3 colorable) if u1=true and u2=false (figure 4).

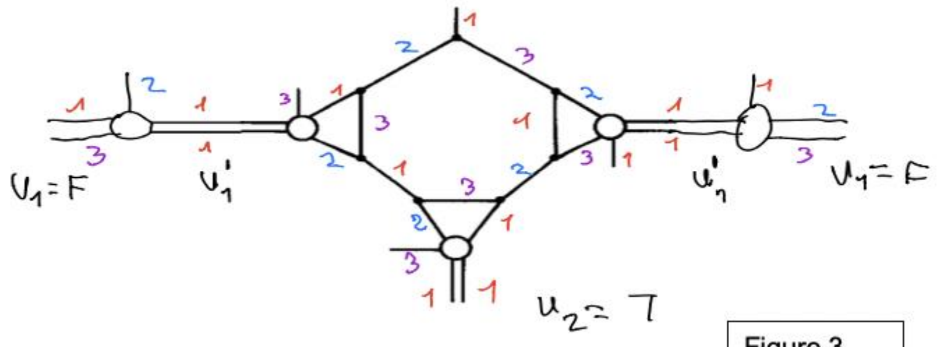


Figure 3

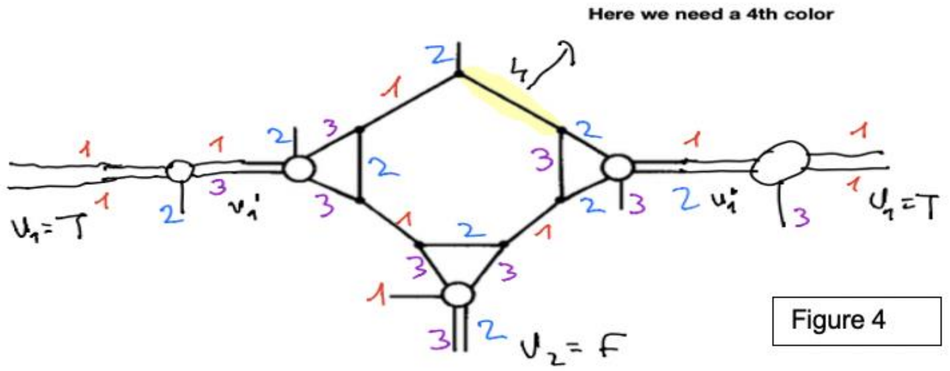


Figure 4

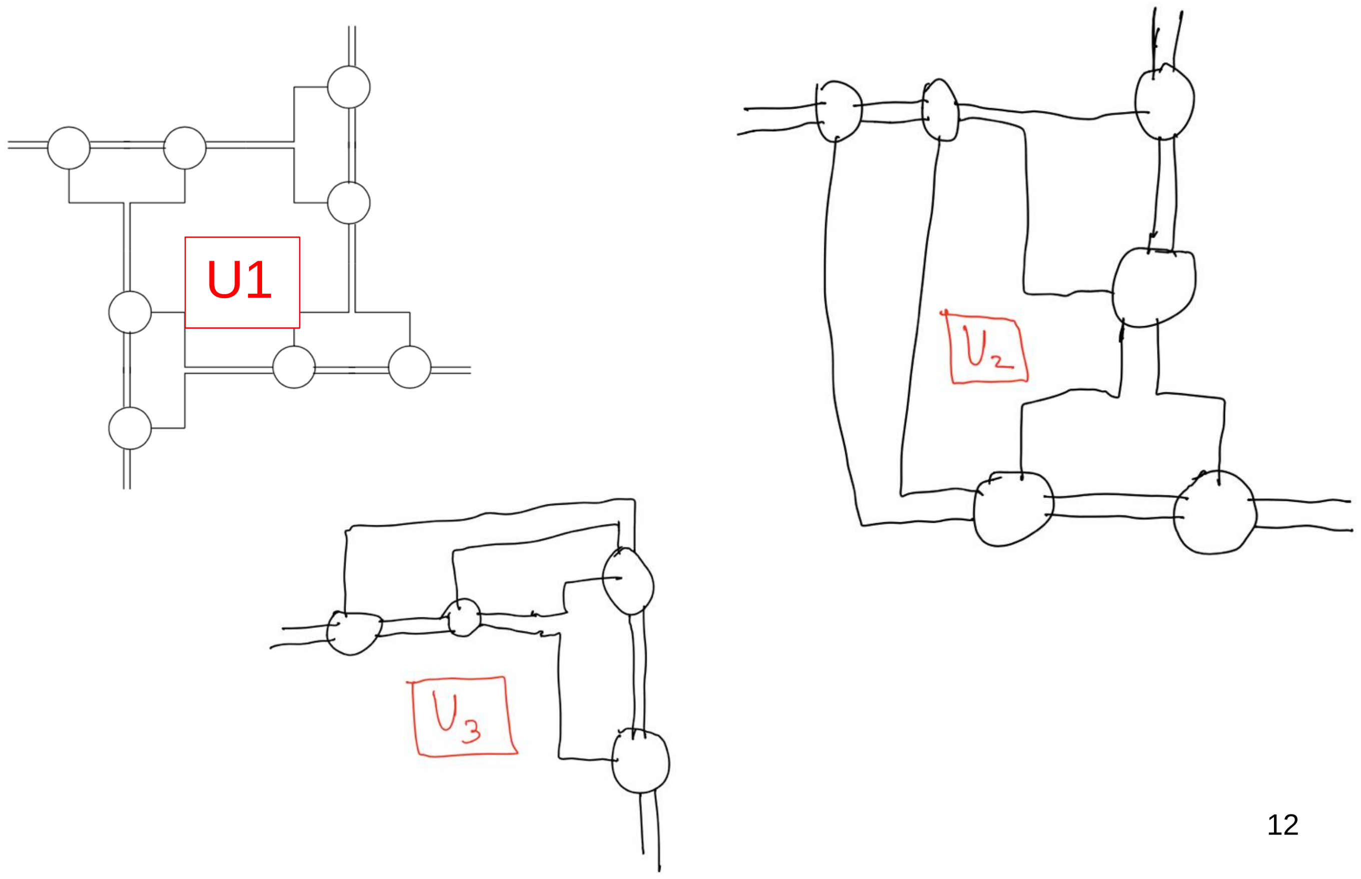
As seen from the figures, there are edges which remain unconnected, in order to deal with them we will simply duplicate the whole graph and connect corresponding edges.

Before continuing to the algorithm part, let us see an example of how a 3-SAT problem is reduced to our problem.

Let P = (u1 Vu3 Vu2)∧ (u1' Vu1 Vu2) ∧ (u1 Vu2 Vu3')

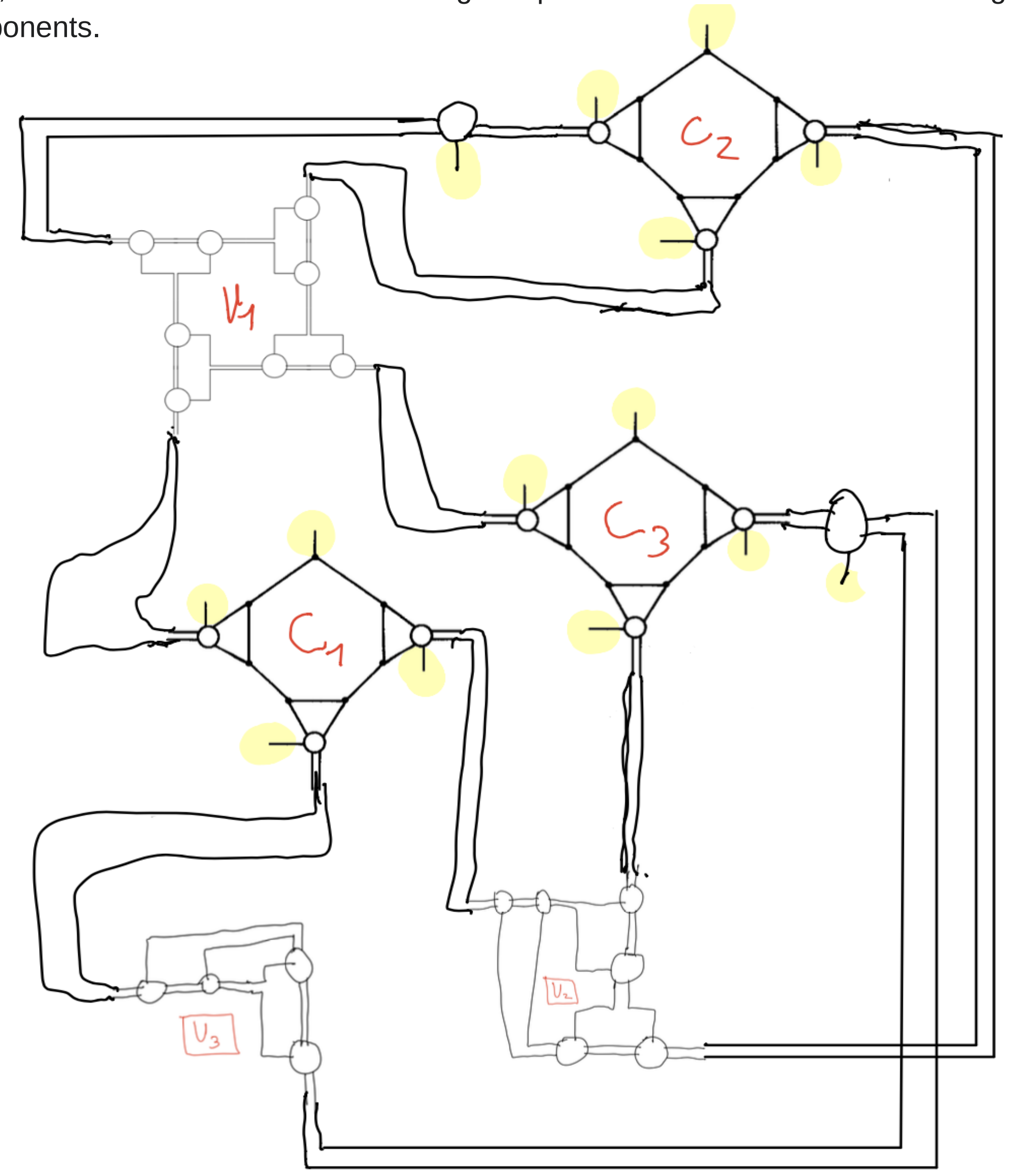
First we count the number of each ui (with it's complement) : #u1=4 #u2=3 #u3=2

Then we form appropriate variable setting component for each literal.
U1 will have 4 output pairs and hence 8 inverting components.
U2 will have 3 output pairs and hence 6 inverting components.
U3 will have 2 output pairs and hence 4 inverting components.



Then we will have 3 satisfaction components since we have 3 conjunctions. The first component will not have any inverting components inserted since no literal within is inverted, other satisfaction components have their first input inverted, hence an inverting component will be inserted to their first inputs.

Then, we will connect the variable setting components with the satisfaction testing components.



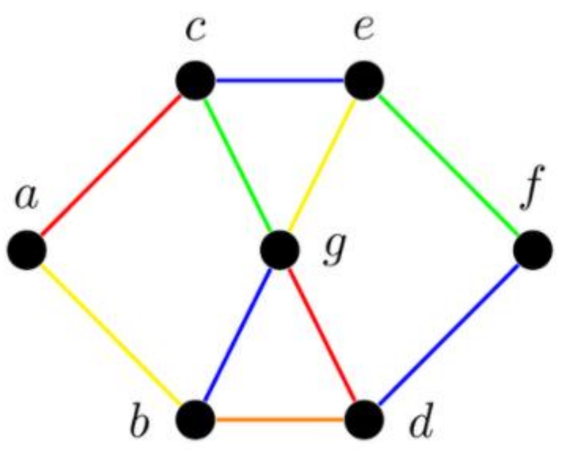
Finally, we will duplicate the above graph and connect the corresponding remaining edges (which are highlighted in the above figure).

Holyer's proof is generalized towards a general k value by Leven and Galil. For further information, Leven and Galil's article that named as "NP Completeness of Finding the Chromatic Index of Regular Graphs" [9]can be examined.

2) Algorithm Description

i.Define the Algorithm

No algorithm is found such that it can solve edge coloring in polynomial time. However, there are algorithms that are found for solving the problem. Some of them work for specific graph types such as bipartite graphs, while others are generalized but do not always give the optimal result.[10] In this problem, our focus is on simple graphs. This algorithm is a heuristic algorithm.



[11]

Algorithm Steps:

1. Use Depth First Search traversal to start traversing the graph.
2. Pick any vertex and start to give distinct color to its edges. While doing coloring, check whether the color is used before for one of its edges or the color is used by one of the edges of the vertex that the main vertex that we consider is connected with that specific edge.
3. Traverse one of its edges and get a new vertex.
4. Repeat all the process until all edges are colored.

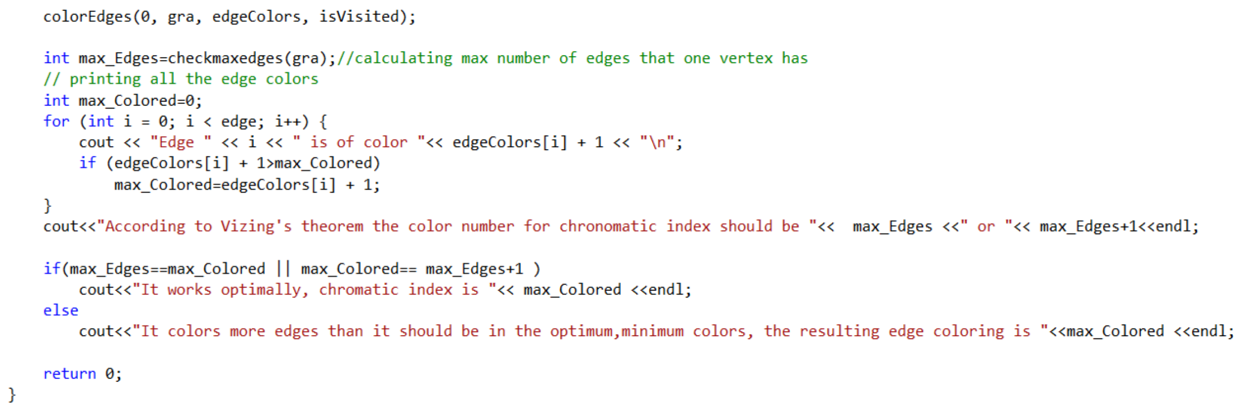
[9] Leven, D. and Galil, Z., 1983. NP completeness of finding the chromatic index of regular graphs. *Journal of Algorithms*, 4(1), pp.35-44.
[10] http://www.cs.umd.edu/class/fall2020/cmsc351-0301/files/bfs.pdf
[11] https://www.wikiwand.com/en/Misra_%26_Gries_edge_coloring_algorithm

ii.Implementation

```
int main()
{

    // declaring vector of vector of pairs, to define Graph
    vector<vector<pair<int, int> > > gra;

    vector<int> edgeColors;

    bool isVisited[100000] = { 0 };

    int ver = 4;
    int edge = 5;

    gra.resize(ver);
    edgeColors.resize(edge, -1);

    // Enter edge & vertices of edge
    // x--; y--;
    // Since graph is undirected, push both pairs
    // (x, y) and (y, x)
    // graph[x].push_back(make_pair(y, i));
    // graph[y].push_back(make_pair(x, i));
```

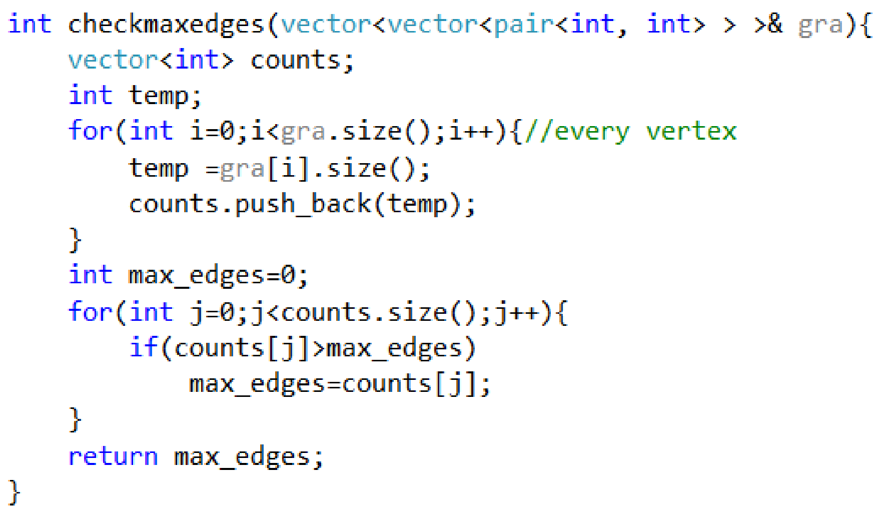
In the main part, we formed the graph that we want to calculate its chromatix index. In order to store information of the edges that each vertex has, a matrix which accumulate the pairs is formed. While the first element of these pairs denotes the other vertex that it is connected, the second element shows the edge number. This edge number will be used to show which edge is colored with which color. According to these, a sample graph is written below.

```
gra[0].push_back(make_pair(1, 0));
gra[1].push_back(make_pair(0, 0));

gra[1].push_back(make_pair(2, 1));
gra[2].push_back(make_pair(1, 1));

gra[2].push_back(make_pair(3, 2));
gra[3].push_back(make_pair(2, 2));

gra[3].push_back(make_pair(0, 3));
gra[0].push_back(make_pair(3, 3));

gra[0].push_back(make_pair(2, 4));
gra[2].push_back(make_pair(0, 4));
```



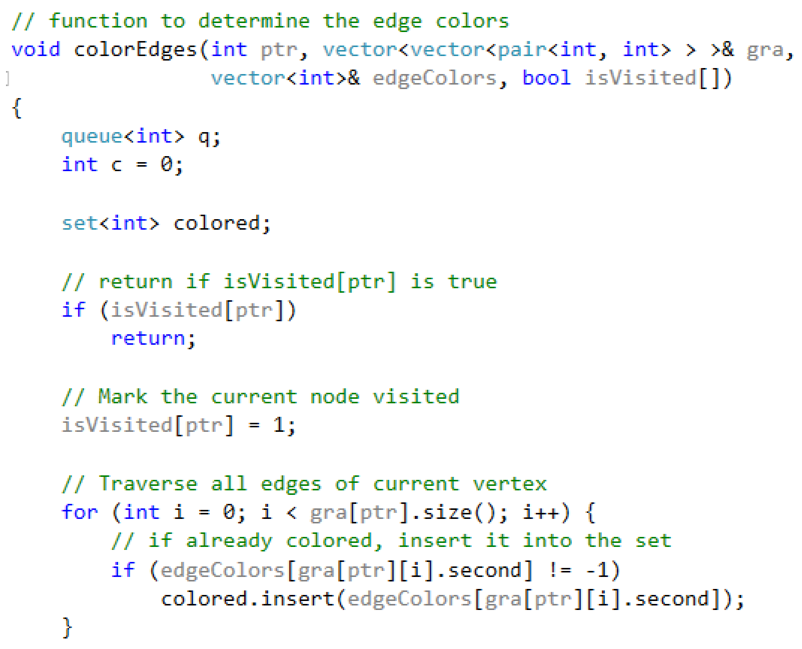
```
    colorEdges(0, gra, edgeColors, isVisited);

    int max_Edges=checkmaxedges(gra);//calculating max number of edges that one vertex has
    // printing all the edge colors
    int max_Colored=0;
    for (int i = 0; i < edge; i++) {
        cout << "Edge " << i << " is of color "<< edgeColors[i] + 1 << "\n";
        if (edgeColors[i] + 1>max_Colored)
            max_Colored=edgeColors[i] + 1;
    }
    cout<<"According to Vizing's theorem the color number for chronomatic index should be "<<  max_Edges <<" or "<< max_Edges+1<<endl;

    if(max_Edges==max_Colored || max_Colored== max_Edges+1 )
        cout<<"It works optimally, chromatic index is "<< max_Colored <<endl;
    else
        cout<<"It colors more edges than it should be in the optimum,minimum colors, the resulting edge coloring is "<<max_Colored <<endl;

    return 0;
}
```

After forming the graph, colorEdges function is called for the 0th vertex of the graph. After all evaluations; at the end of the main, the results are presented. While considering the optimal coloring; as Vizing's theorem states, we have evaluated the result of k and k+1, which k denotes the maximal degree of the vertices of the graph, as optimal. It should be noted that the optimal result can be concluded as k+1 whereas it is k in reality. These cases will be considered in the correctness part of the project.



```
int checkmaxedges(vector<vector<pair<int, int> > >& gra){
    vector<int> counts;
    int temp;
    for(int i=0;i<gra.size();i++){//every vertex
        temp =gra[i].size();
        counts.push_back(temp);
    }
    int max_edges=0;
    for(int j=0;j<counts.size();j++){
        if(counts[j]>max_edges)
            max_edges=counts[j];
    }
    return max_edges;
}
```

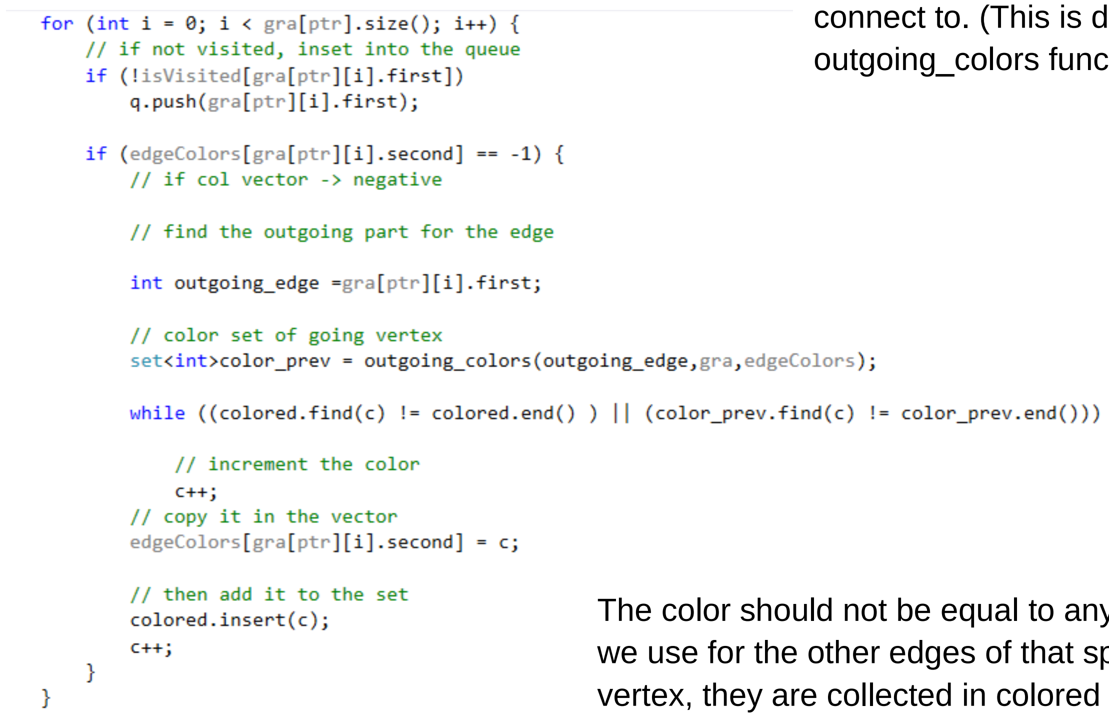
In this function, the maximal degree of the vertices of the graph is calculated and returned. The result of this is used to compare maximal degree and the algorithm's result for chromatic index.



```
// function to determine the edge colors
void colorEdges(int ptr, vector<vector<pair<int, int> > >& gra,
                  vector<int>& edgeColors, bool isVisited[])
{
    queue<int> q;
    int c = 0;

    set<int> colored;

    // return if isVisited[ptr] is true
    if (isVisited[ptr])
        return;

    // Mark the current node visited
    isVisited[ptr] = 1;

    // Traverse all edges of current vertex
    for (int i = 0; i < gra[ptr].size(); i++) {
        // if already colored, insert it into the set
        if (edgeColors[gra[ptr][i].second] != -1)
            colored.insert(edgeColors[gra[ptr][i].second]);
    }
```

At this part, edge colors are determined. The unvisited vertices are evaluated and the second element of every pair that a vertex has is checked to understand whether it is colored or not. edgeColors is used for this goal.

In the main part all of them is equalized to -1. If it is not -1, it means that it is colored. If it is not colored, first we need to check the colors that is used for the vertex that it will connect to. (This is done by outgoing_colors function.)



```
    for (int i = 0; i < gra[ptr].size(); i++) {
        // if not visited, inset into the queue
        if (!isVisited[gra[ptr][i].first])
            q.push(gra[ptr][i].first);

        if (edgeColors[gra[ptr][i].second] == -1) {
            // if col vector -> negative

            // find the outgoing part for the edge

            int outgoing_edge =gra[ptr][i].first;

            // color set of going vertex
            set<int>color_prev = outgoing_colors(outgoing_edge,gra,edgeColors);

            while ((colored.find(c) != colored.end() ) || (color_prev.find(c) != color_prev.end()))

                // increment the color
                c++;
            // copy it in the vector
            edgeColors[gra[ptr][i].second] = c;

            // then add it to the set
            colored.insert(c);
            c++;
        }
    }
```

The color should not be equal to any color that we use for the other edges of that specific vertex, they are collected in colored set. In addition to that, we have to check the colors that is used by the vertex that our specific vertex is connected to. The colors that are used by the other vertex is stored in color_prev. If the color is found in one of them we need to use another

```
    // while queue's not empty
    while (!q.empty()) {
        int temp = q.front();
        q.pop();

        colorEdges(temp, gra, edgeColors, isVisited);
    }

    return;
}
```

In the queue q, the vertices that is connected to main vertex is stored. It will visit the connected vertices and their edges again and again by calling the function inside of it. Therefore, at the end there will be no vertex that remain unvisited.

```
set<int> outgoing_colors (int outgoing_index,vector<vector<pair<int, int> > >& gra,vector<int>& edgeColors) {

    set<int> colors;

    for (int i=0; i<gra[outgoing_index].size(); i++) {

        int edge_color = edgeColors[gra[outgoing_index][i].second];

        if (edge_color !=-1 ) {
            colors.insert(edge_color);
        }
    }

    return colors;

}
```

This function is used to detect the the colors that is used by the other vertex for the colorEdges function.

The algorithm is inspired from a code in a website[12]. Since the code does not work correctly and insufficient for our problem, we have changed lots of detail in the code and added additional functions. Outgoing_colors and checkmaxedges and functions for generating random graphs are formed by us.

3) Algorithm Analysis

i. Correctness of the algorithm

Claim: Each edge is colored, and no two adjacent edges are colored with the same color.

[12] https://www.geeksforgeeks.org/edge-coloring-of-a-graph/

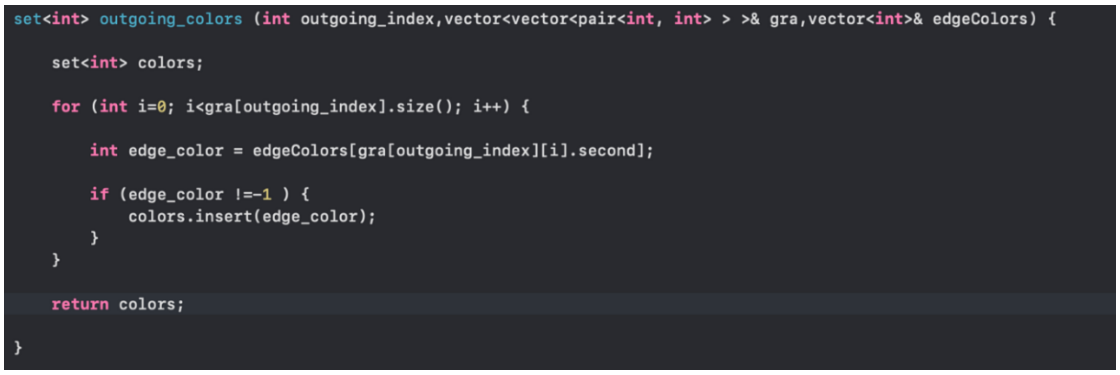
Loop invariants method:

Initialization: For a graph with 0 edge, it is trivially correct that each edge is colored and with a different color since there are no existing edges.

Maintenance: If for the given vertex with m th edge the claim holds, then for the m+1 th edge the claim holds. We check both endpoints of the given edge and assign it with a color such that no adjacent edge with it has the same assigned color. (Lines 69-88 in our code).

Termination: For the last edge the claim will hold since all edges connected to both endpoints of the given edge are checked and a color is assigned to the given edge such that it is not assigned before to any adjacent edge.

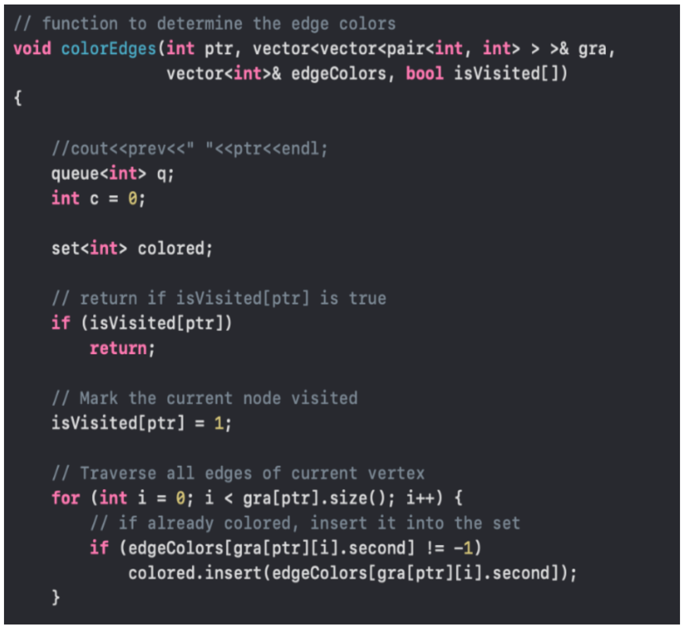
ii. Complexity analysis



```
set<int> outgoing_colors (int outgoing_index,vector<vector<pair<int, int> > >& gra,vector<int>& edgeColors) {

    set<int> colors;

    for (int i=0; i<gra[outgoing_index].size(); i++) {

        int edge_color = edgeColors[gra[outgoing_index][i].second];

        if (edge_color !=-1 ) {
            colors.insert(edge_color);
        }
    }

    return colors;

}
```

O(E)

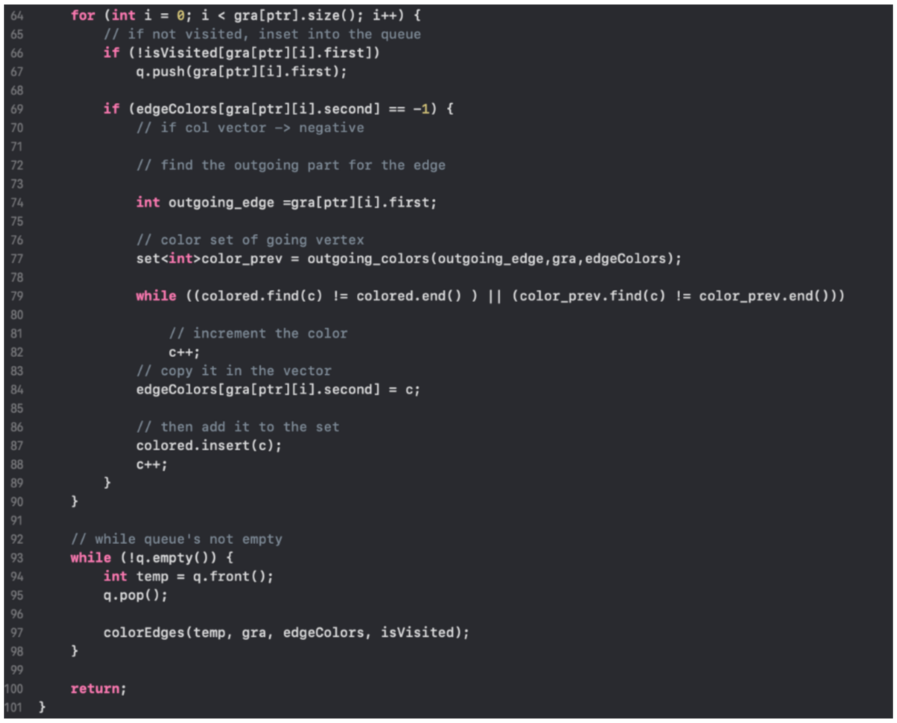
The function outgoing_colors iterates over the edges of a given vertex so the worst case behaviour is O(E).



```
// function to determine the edge colors
void colorEdges(int ptr, vector<vector<pair<int, int> > >& gra,
                vector<int>& edgeColors, bool isVisited[])
{

    //cout<<prev<<" "<<ptr<<endl;
    queue<int> q;
    int c = 0;

    set<int> colored;

    // return if isVisited[ptr] is true
    if (isVisited[ptr])
        return;

    // Mark the current node visited
    isVisited[ptr] = 1;

    // Traverse all edges of current vertex
    for (int i = 0; i < gra[ptr].size(); i++) {
        // if already colored, insert it into the set
        if (edgeColors[gra[ptr][i].second] != -1)
            colored.insert(edgeColors[gra[ptr][i].second]);
    }
```

Since we do constant time operations here, the time complexity is O(1).

The maximum number of edges for a given vertex can be E which is the total number of edges, hence time complexity of this for loop is O(E).



```
    for (int i = 0; i < gra[ptr].size(); i++) {
        // if not visited, inset into the queue
        if (!isVisited[gra[ptr][i].first])
            q.push(gra[ptr][i].first);

        if (edgeColors[gra[ptr][i].second] == -1) {
            // if col vector -> negative

            // find the outgoing part for the edge

            int outgoing_edge =gra[ptr][i].first;

            // color set of going vertex
            set<int>color_prev = outgoing_colors(outgoing_edge,gra,edgeColors);

            while ((colored.find(c) != colored.end() ) || (color_prev.find(c) != color_prev.end()))

                // increment the color
                c++;
            // copy it in the vector
            edgeColors[gra[ptr][i].second] = c;

            // then add it to the set
            colored.insert(c);
            c++;
        }
    }

    // while queue's not empty
    while (!q.empty()) {
        int temp = q.front();
        q.pop();

        colorEdges(temp, gra, edgeColors, isVisited);
    }

    return;
}
```

O(E^2)

In the above for loop we traverse all the edges and inside we call the outgoing_colors function which has a time complexity of O(E) (Shown in the previous page) and we have a while loop which iterates until a non-existing color is found, so it is O(color) and we know that the maximum number of different colors is actually the number of edges, so the overall time complexity of this for loop is O(E)*O(E)+ O(E)*O(E)= O(E^2).

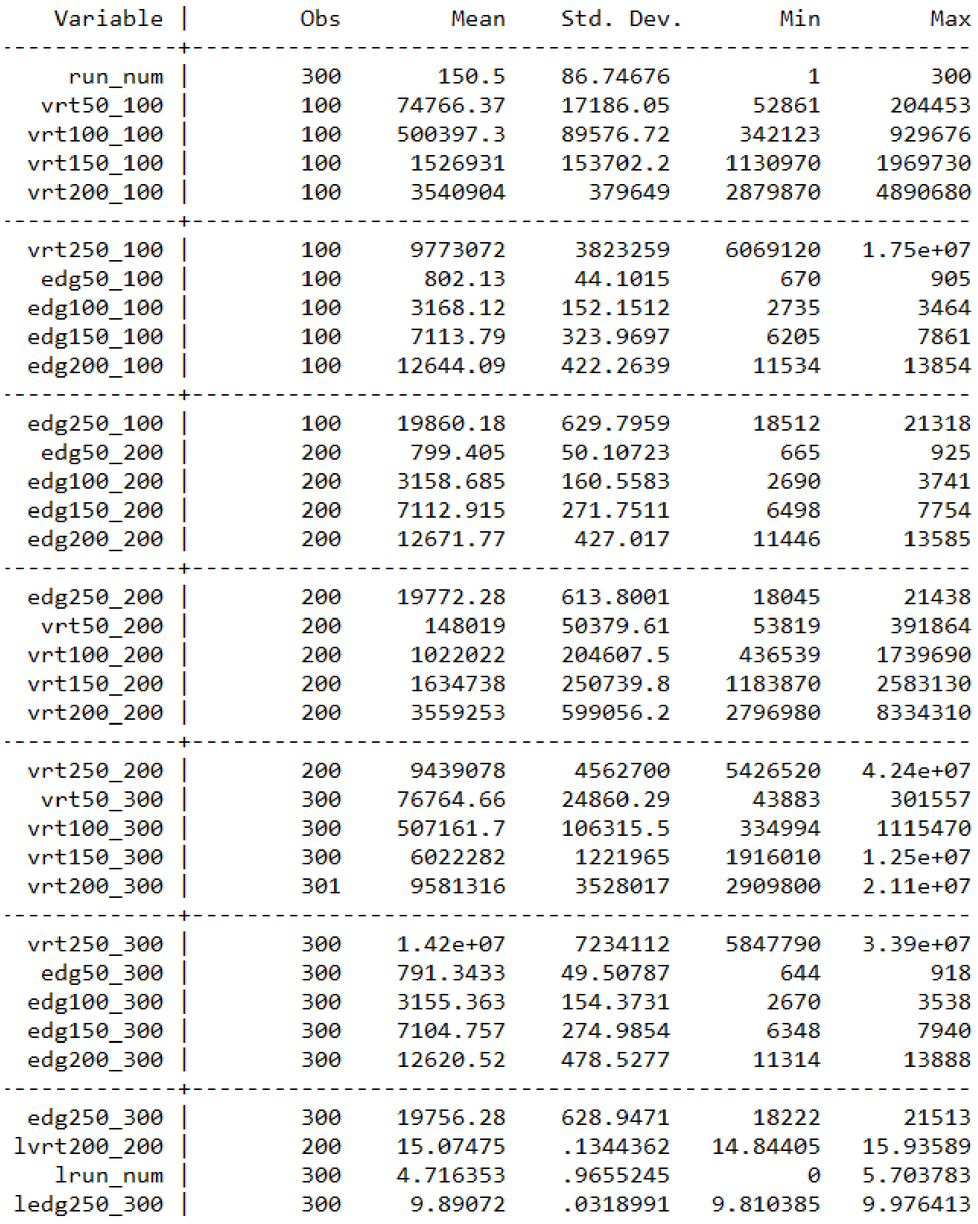
The while loop at the and searches all vertices with a Depth-first Search which has a time complexity of O(V+E). Although it seems like the whole algorithm is O(V+E)*O(E^2), one may observe that at the beginning we check if the current vertex is already visited or not. Since each vertex is not visited only for a single time and then it becomes visited the algorithm turns out to be $\underbrace{O(V)*O(E^2)}_{\text{Vertices which are visited for the first time.}} + \underbrace{O(E)*O(1)}_{\text{Vertices which are already visited.}} = O(V*E^2)$.

4)Experimental Analysis

i. Running time analysis

While analyzing the running time of the algorithm, these metrices will be used; standard deviation, standard error, sample mean, confidence interval and linear regression (OLS). The results are shown in the format of microseconds.

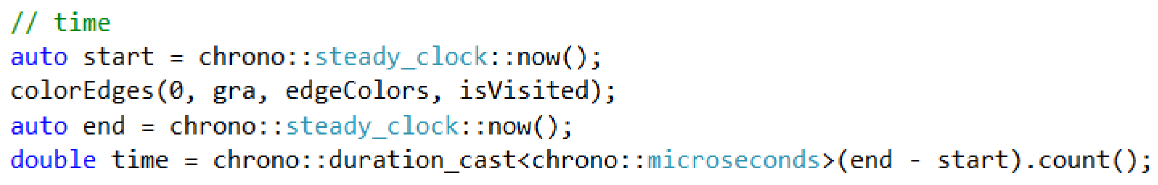
Below, summary statistics are shown for graphs with vertices 50,100,150,200 and 250; and edges of the graphs:



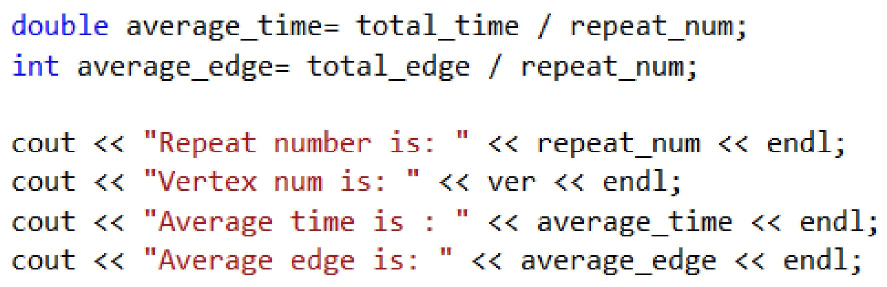
```
    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
     run_num |        300       150.5    86.74676          1        300
   vrt50_100 |        100    74766.37    17186.05      52861     204453
  vrt100_100 |        100    500397.3    89576.72     342123     929676
  vrt150_100 |        100     1526931    153702.2    1130970    1969730
  vrt200_100 |        100     3540904      379649    2879870    4890680
-------------+---------------------------------------------------------
  vrt250_100 |        100     9773072     3823259    6069120   1.75e+07
   edg50_100 |        100      802.13     44.1015        670        905
  edg100_100 |        100     3168.12    152.1512       2735       3464
  edg150_100 |        100     7113.79    323.9697       6205       7861
  edg200_100 |        100    12644.09    422.2639      11534      13854
-------------+---------------------------------------------------------
  edg250_100 |        100    19860.18    629.7959      18512      21318
   edg50_200 |        200     799.405    50.10723        665        925
  edg100_200 |        200    3158.685    160.5583       2690       3741
  edg150_200 |        200    7112.915    271.7511       6498       7754
  edg200_200 |        200    12671.77     427.017      11446      13585
-------------+---------------------------------------------------------
  edg250_200 |        200    19772.28    613.8001      18045      21438
   vrt50_200 |        200      148019    50379.61      53819     391864
  vrt100_200 |        200     1022022    204607.5     436539    1739690
  vrt150_200 |        200     1634738    250739.8    1183870    2583130
  vrt200_200 |        200     3559253    599056.2    2796980    8334310
-------------+---------------------------------------------------------
  vrt250_200 |        200     9439078     4562700    5426520   4.24e+07
   vrt50_300 |        300    76764.66    24860.29      43883     301557
  vrt100_300 |        300    507161.7    106315.5     334994    1115470
  vrt150_300 |        300     6022282     1221965    1916010   1.25e+07
  vrt200_300 |        301     9581316     3528017    2909800   2.11e+07
-------------+---------------------------------------------------------
  vrt250_300 |        300    1.42e+07     7234112    5847790   3.39e+07
   edg50_300 |        300    791.3433    49.50787        644        918
  edg100_300 |        300    3155.363    154.3731       2670       3538
  edg150_300 |        300    7104.757    274.9854       6348       7940
  edg200_300 |        300    12620.52    478.5277      11314      13888
-------------+---------------------------------------------------------
  edg250_300 |        300    19756.28    628.9471      18222      21513
 lvrt200_200 |        200    15.07475    .1344362   14.84405   15.93589
    lrun_num |        300    4.716353    .9655245          0   5.703783
 ledg250_300 |        300     9.89072    .0318991   9.810385   9.976413
```

verX_Y : X denotes number of vertices, Y denotes number of repetitions. This shows runtime.
edgX_Y: X denotes number of vertices, Y denotes number of repetitions. This shows the number of edges.
lvrtX_Y: log transformed version of verX_Y.
ledgX_Y: log transformed version of edgX_Y.

As it can be seen from the above chart, the relationship between average number of edges and vertices is approximately 0.32*V^2.



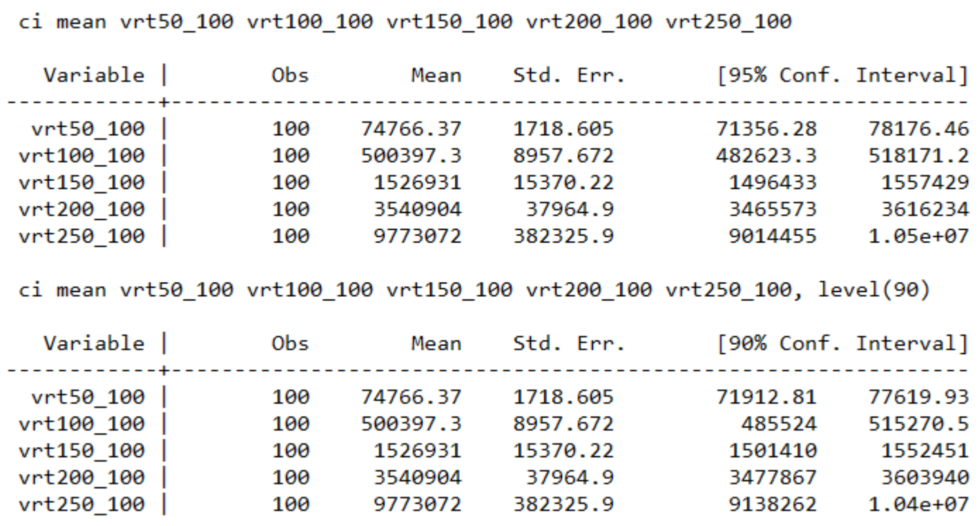
```
// time
auto start = chrono::steady_clock::now();
colorEdges(0, gra, edgeColors, isVisited);
auto end = chrono::steady_clock::now();
double time = chrono::duration_cast<chrono::microseconds>(end - start).count();
```



```
double average_time= total_time / repeat_num;
int average_edge= total_edge / repeat_num;

cout << "Repeat number is: " << repeat_num << endl;
cout << "Vertex num is: " << ver << endl;
cout << "Average time is : " << average_time << endl;
cout << "Average edge is: " << average_edge << endl;
```

In this piece of code, the necessary calculations for recording time are made. Results are stored in txt files. For statistical analysis, we utilize statistical analysis software Stata. Below you can see the mean and confidence interval of each graphs with different number of iterations. Standard deviations are located in the above chart.

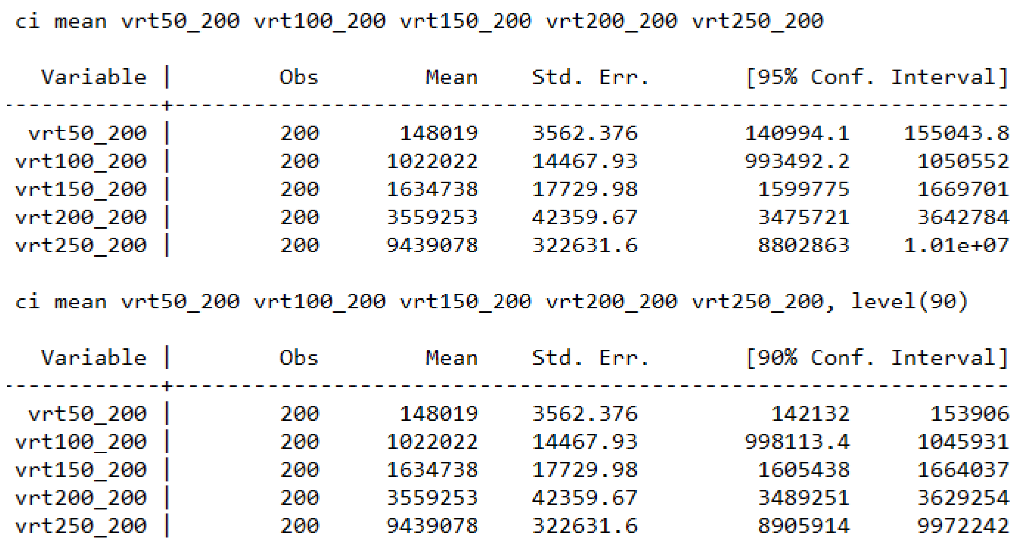
For 100 iterations mean, std error and confidence intervals (90% and 95%):



```
ci mean vrt50_100 vrt100_100 vrt150_100 vrt200_100 vrt250_100

    Variable |        Obs        Mean    Std. Err.       [95% Conf. Interval]
-------------+---------------------------------------------------------------
   vrt50_100 |        100    74766.37    1718.605        71356.28    78176.46
  vrt100_100 |        100    500397.3    8957.672        482623.3    518171.2
  vrt150_100 |        100     1526931    15370.22         1496433     1557429
  vrt200_100 |        100     3540904     37964.9         3465573     3616234
  vrt250_100 |        100     9773072    382325.9         9014455    1.05e+07

ci mean vrt50_100 vrt100_100 vrt150_100 vrt200_100 vrt250_100, level(90)

    Variable |        Obs        Mean    Std. Err.       [90% Conf. Interval]
-------------+---------------------------------------------------------------
   vrt50_100 |        100    74766.37    1718.605        71912.81    77619.93
  vrt100_100 |        100    500397.3    8957.672          485524    515270.5
  vrt150_100 |        100     1526931    15370.22         1501410     1552451
  vrt200_100 |        100     3540904     37964.9         3477867     3603940
  vrt250_100 |        100     9773072    382325.9         9138262    1.04e+07
```

<u>In 100 iteration case;</u> graph with 50 vertices terminates around 0.074 seconds on average whereas graph with 250 vertices terminates around 9.77 seconds on average. It can be said that with 95% probability graph with 50 vertices runs in the interval between 0.071 and 0.078 seconds; graph with 250 vertices runs in the interval between 9.01 and 10.5 seconds.

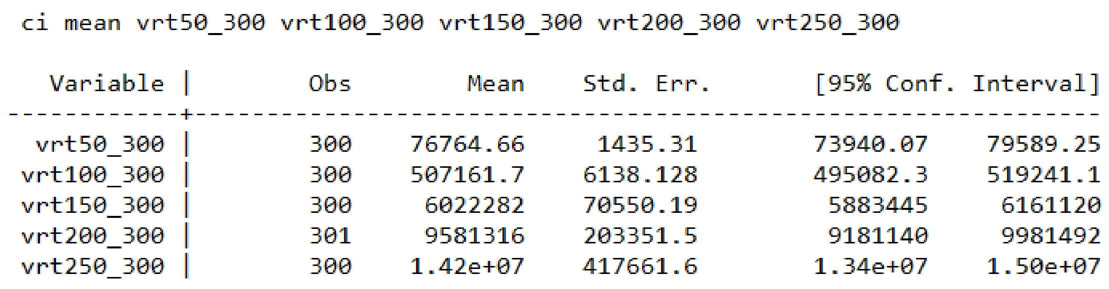
For 200 iterations mean, std error and confidence intervals (90% and 95%):



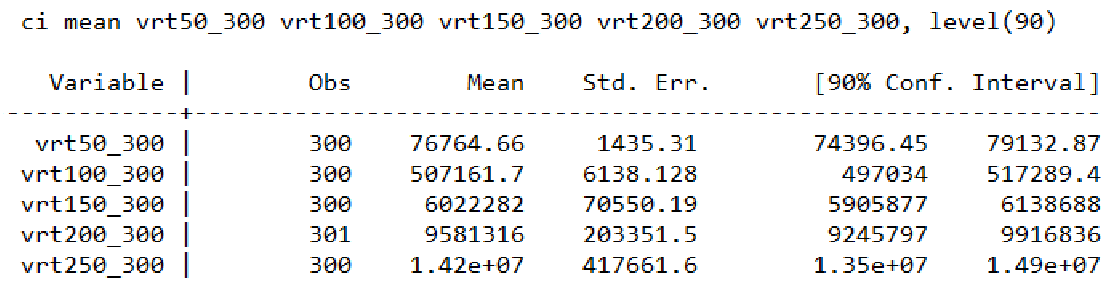
```
ci mean vrt50_200 vrt100_200 vrt150_200 vrt200_200 vrt250_200

    Variable |        Obs        Mean    Std. Err.       [95% Conf. Interval]
-------------+---------------------------------------------------------------
   vrt50_200 |        200      148019    3562.376        140994.1    155043.8
  vrt100_200 |        200     1022022    14467.93        993492.2     1050552
  vrt150_200 |        200     1634738    17729.98         1599775     1669701
  vrt200_200 |        200     3559253    42359.67         3475721     3642784
  vrt250_200 |        200     9439078    322631.6         8802863    1.01e+07

ci mean vrt50_200 vrt100_200 vrt150_200 vrt200_200 vrt250_200, level(90)

    Variable |        Obs        Mean    Std. Err.       [90% Conf. Interval]
-------------+---------------------------------------------------------------
   vrt50_200 |        200      148019    3562.376          142132      153906
  vrt100_200 |        200     1022022    14467.93        998113.4     1045931
  vrt150_200 |        200     1634738    17729.98         1605438     1664037
  vrt200_200 |        200     3559253    42359.67         3489251     3629254
  vrt250_200 |        200     9439078    322631.6         8905914     9972242
```

In 200 iteration case; graph with 50 vertices terminates around 0.148 seconds on average whereas graph with 250 vertices terminates around 9.43 seconds on average. It can be said that with 95% probability graph with 50 vertices runs in the interval between 0.140 and 0.155 seconds; graph with 250 vertices runs in the interval between 8.8 and 10.1 seconds.

For 300 iterations mean, std error and confidence intervals (90% and 95%):



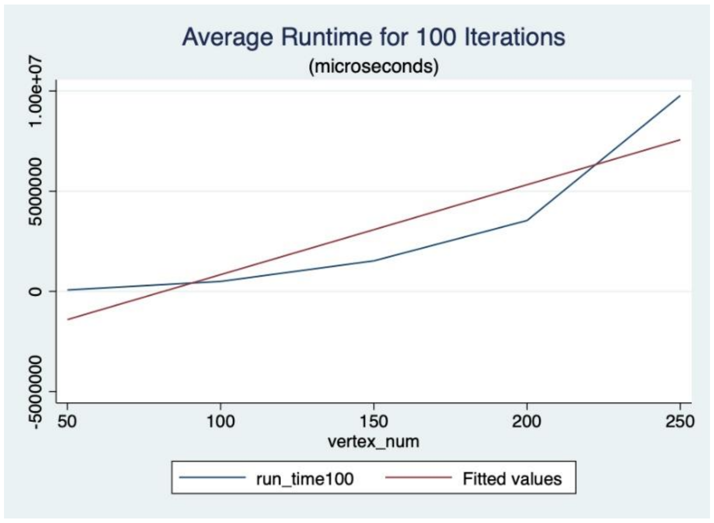
```
ci mean vrt50_300 vrt100_300 vrt150_300 vrt200_300 vrt250_300

    Variable |        Obs        Mean    Std. Err.       [95% Conf. Interval]
-------------+---------------------------------------------------------------
   vrt50_300 |        300    76764.66     1435.31        73940.07    79589.25
  vrt100_300 |        300    507161.7    6138.128        495082.3    519241.1
  vrt150_300 |        300     6022282    70550.19         5883445     6161120
  vrt200_300 |        301     9581316    203351.5         9181140     9981492
  vrt250_300 |        300    1.42e+07    417661.6        1.34e+07    1.50e+07
```

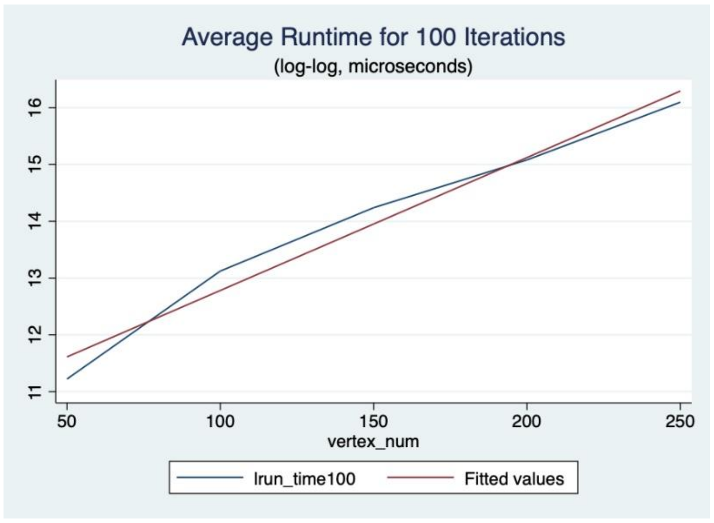


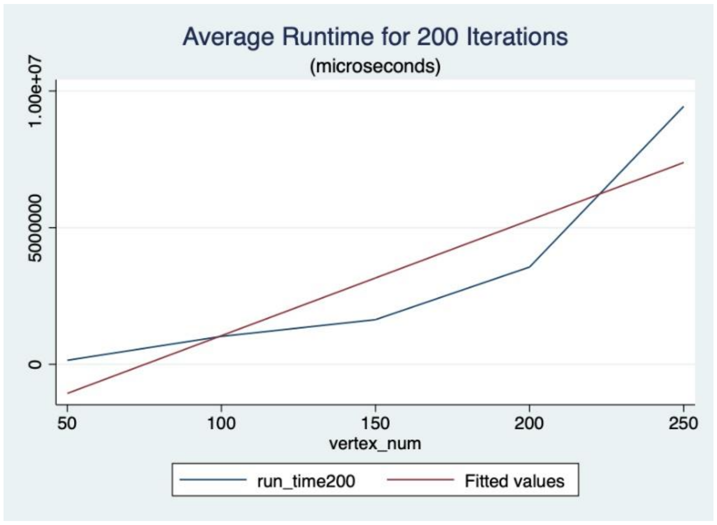
```
ci mean vrt50_300 vrt100_300 vrt150_300 vrt200_300 vrt250_300, level(90)

    Variable |        Obs        Mean    Std. Err.       [90% Conf. Interval]
-------------+---------------------------------------------------------------
   vrt50_300 |        300    76764.66     1435.31        74396.45    79132.87
  vrt100_300 |        300    507161.7    6138.128          497034    517289.4
  vrt150_300 |        300     6022282    70550.19         5905877     6138688
  vrt200_300 |        301     9581316    203351.5         9245797     9916836
  vrt250_300 |        300    1.42e+07    417661.6        1.35e+07    1.49e+07
```

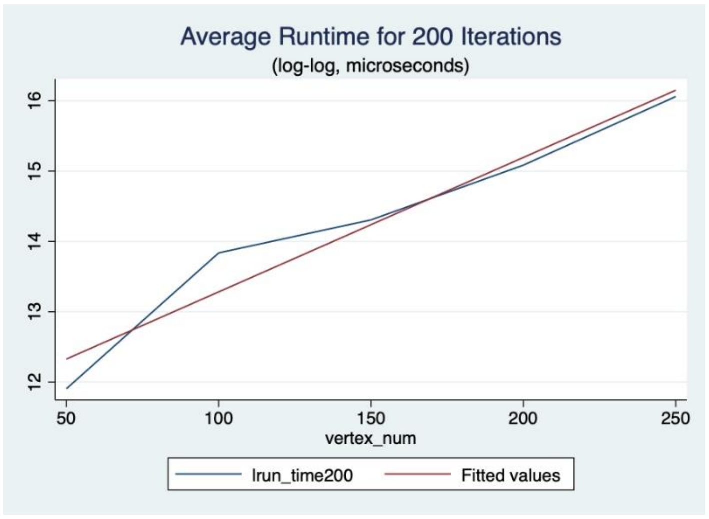
In 300 iteration case; graph with 50 vertices terminates around 0.076 seconds on average whereas graph with 250 vertices terminates around 14.2 seconds on average. It can be said that with 95% probability graph with 50 vertices runs in the interval between 0.074 and 0.079 seconds; graph with 250 vertices runs in the interval between 13.4 and 15 seconds.

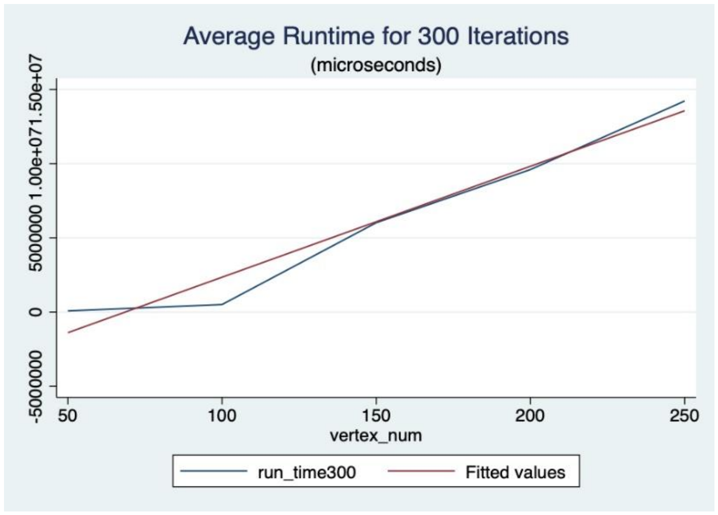
Graphs of Average Runtime:

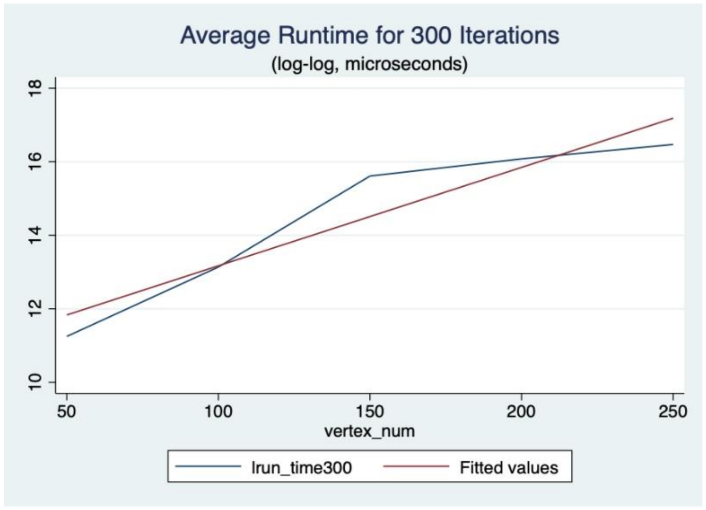




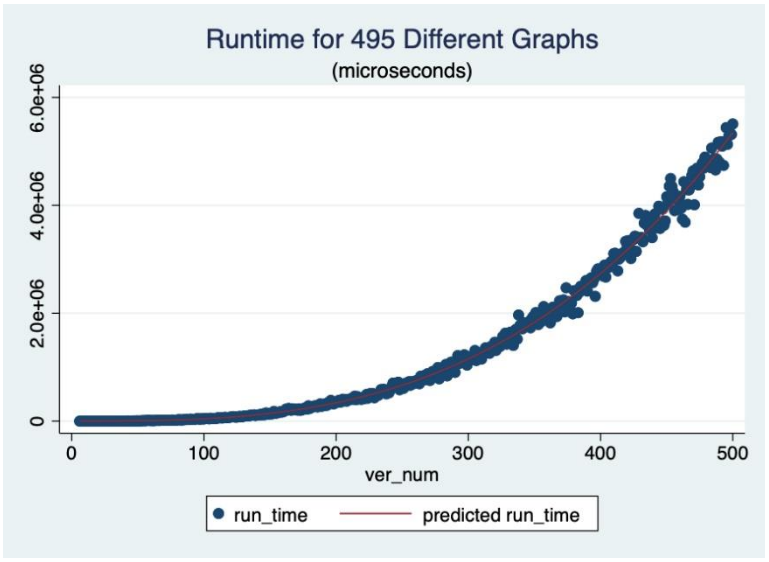


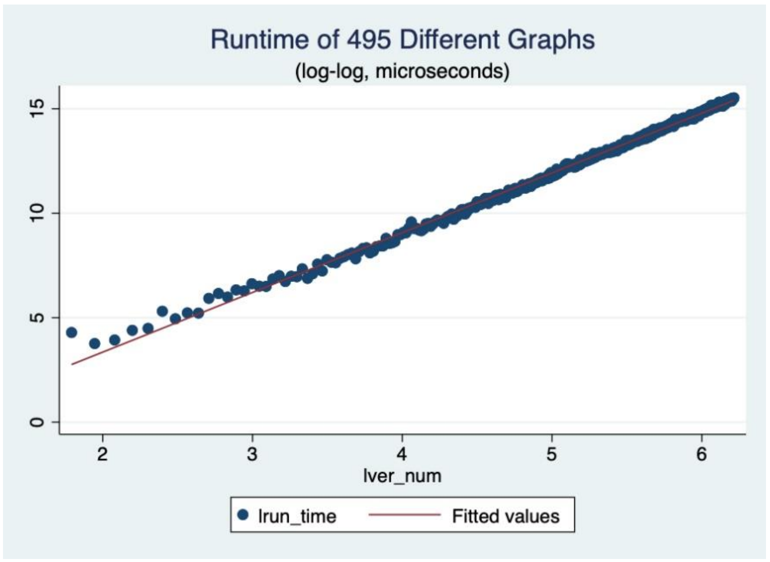


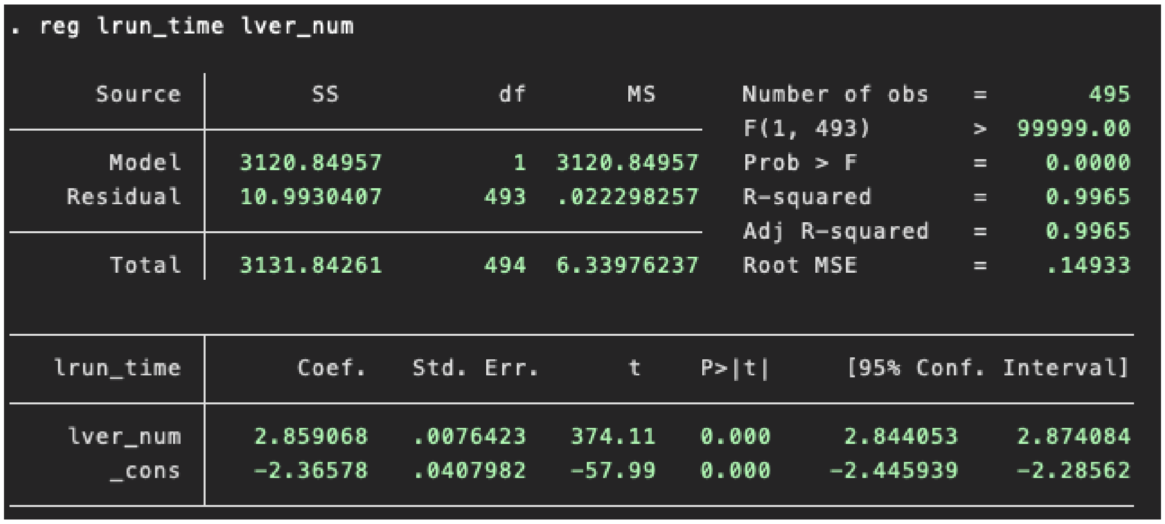




In order to compare the theoretical analysis (O(V*E^2)= O(V*(V^2)^2)= O(V^5)) result with the experimental analysis result; since the slope is not linear, the log-log plots are formed.





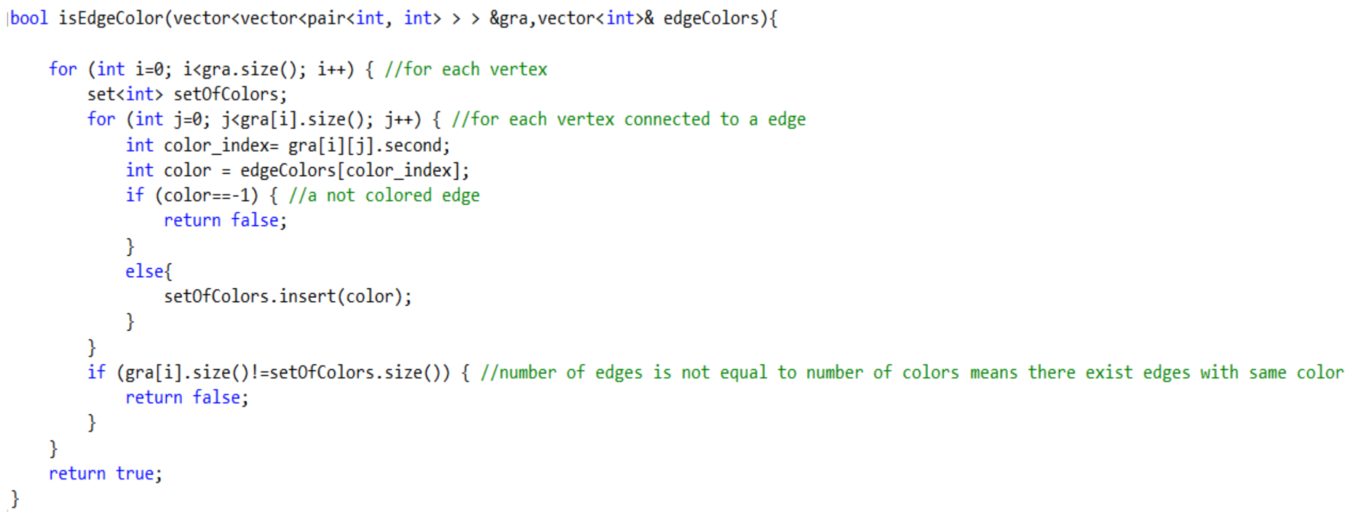


```
. reg lrun_time lver_num

      Source |       SS           df       MS      Number of obs   =       495
-------------+----------------------------------   F(1, 493)       >  99999.00
       Model |  3120.84957         1  3120.84957   Prob > F        =    0.0000
    Residual |  10.9930407       493  .022298257   R-squared       =    0.9965
-------------+----------------------------------   Adj R-squared   =    0.9965
       Total |  3131.84261       494  6.33976237   Root MSE        =    .14933

------------------------------------------------------------------------------
   lrun_time |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
    lver_num |   2.859068   .0076423   374.11   0.000     2.844053    2.874084
       _cons |   -2.36578   .0407982   -57.99   0.000    -2.445939    -2.28562
------------------------------------------------------------------------------
```

To validate our expectations, we created 495 different graphs that has vertices between 5 to 500. To find the experimental relationship between vertices and runtime, we run a simple linear regression (OLS). To be able to use this method, we need to do log-log transformation on the data. As a result of this, in practice the running time complexity can be written as approximately O(V^2.85). Since P value is around 0, result is statistically significant at 5% significance level.

ii. Correctness

This algorithm does not give optimal minimum coloring result for every graph. However, it works correctly to color the graph such as no adjacent edges colored with the same color.



```
bool isEdgeColor(vector<vector<pair<int, int> > > &gra,vector<int>& edgeColors){

    for (int i=0; i<gra.size(); i++) { //for each vertex
        set<int> setOfColors;
        for (int j=0; j<gra[i].size(); j++) { //for each vertex connected to a edge
            int color_index= gra[i][j].second;
            int color = edgeColors[color_index];
            if (color==-1) { //a not colored edge
                return false;
            }
            else{
                setOfColors.insert(color);
            }
        }
        if (gra[i].size()!=setOfColors.size()) { //number of edges is not equal to number of colors means there exist edges with same color
            return false;
        }
    }
    return true;
}
```

Coloring is controlled with isEdgeColor function. This function returns false when the edgeColors value is -1. This means that the edge has not been colored. In addition to this; for a given vertex it controls whether the number of colors that is used is equal or not equal to the number of edges that this vertex has. If they are not equal it means that the same color is used more than ones for the specific vertex. This process is done for every vertex in the graph.

While checking the running times of the random graphs for multiple iterations; this correctness is controlled. It is seen that in every time it comes true. This shows that correctness rate is 1. Every time it colors as no adjacent edges colored with the same color. However, as it is stated it might use number of colors which is more than the optimal value. Sample cases is located in the testing part.

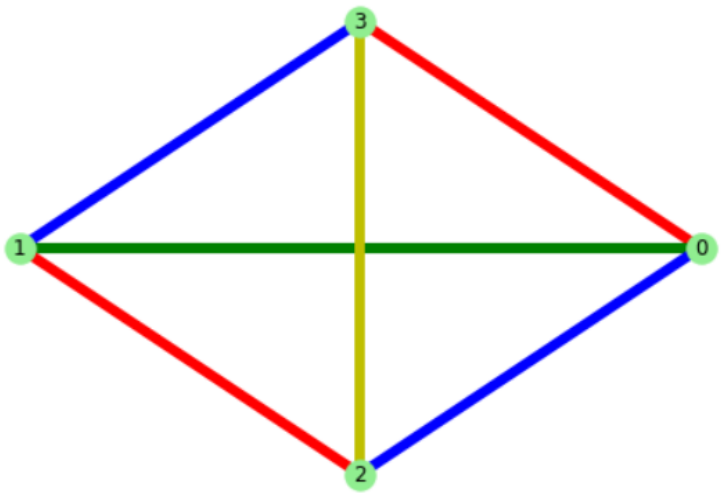
5) Testing

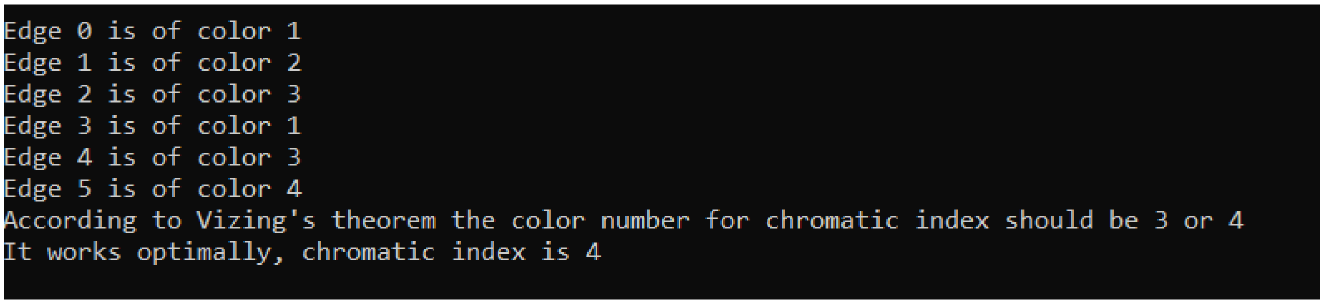
For testing the algorithm; black box testing will be used. In order to apply this testing, we created functions to generate random graphs for a user-entered vertex number.

```
void random_graph_gen (vector<vector<pair<int, int> > > &gra, int vertex_num,int &index) {

    int vertex_1=vertex_num;
    bool check_bigger= false;
    bool check_four = false;

    if (vertex_num>4) {
        check_bigger=true;
    }
    else if (vertex_num==4) {
        check_four=true;
    }
    for (int i=0; i<vertex_num; i++) {

        set<int>used_vertices = used_vertex(gra[i]);
        int  rand_edge_num=0;

        if (check_bigger) {
                vertex_1 = vertex_num-4;
                rand_edge_num = rand() % vertex_1 + 3; //Random number generator for edge.
           }
        else if (check_four) {

            rand_edge_num = 3; // if number of vertex is 4 then max and min edge number is 3;
        }
        int cur_edge_num = gra[i].size();
        rand_edge_num = rand_edge_num - cur_edge_num; // Discard the current edge number of vertex from random edge number assigned.

        for (int j=0; j< rand_edge_num;) {

            int  rand_vertex = (rand() % vertex_num ); //Random number generator to decide which vertex will be connected.

            if ((used_vertices.find(rand_vertex) == used_vertices.end()) && rand_vertex!=i) {

                gra[i].push_back(make_pair(rand_vertex, index));
                gra[rand_vertex].push_back(make_pair(i, index));

                cout << i << " " << rand_vertex << " index: "<< index << endl;

                index++;
                j++;
                used_vertices.insert(rand_vertex);
            }
        }
        cout << endl;
    }
}
```

In this function; we generate random graphs that has 4 vertices at least. This way is chosen because as it is mentioned in the NP complete proof, we stated that there is a mapping from 3-SAT to edge coloring problem. For this reason, we assume that one vertex has at least 3 edges. To have 3 edges for a one vertex, we have to have 4 vertices at least. For vertex number which is bigger than 3; random edge number is assigned. Before the for loop, we subtract the current edge number from rand edge number in order not to assign an edge that assigned already. In for loop, random vertex is chosen for connection as rand_vertex. At the end of for loop; after making the necessary increments for index and i, the vertex that is used is added to used_vertices set in order not to use the vertex again.

Test run-1 with 4 vertices:

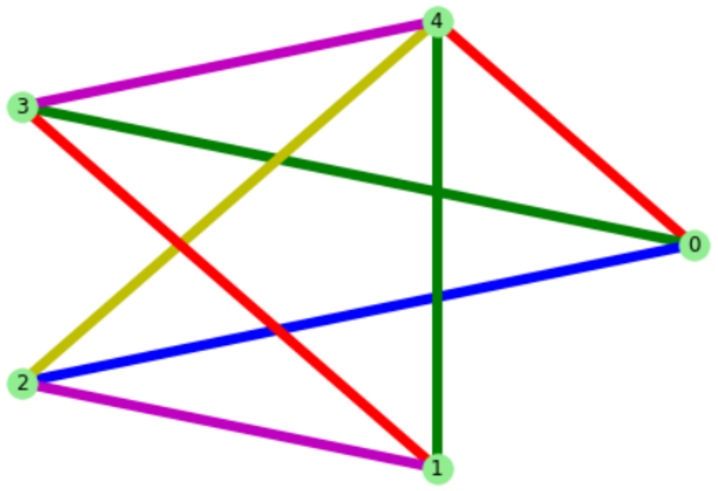


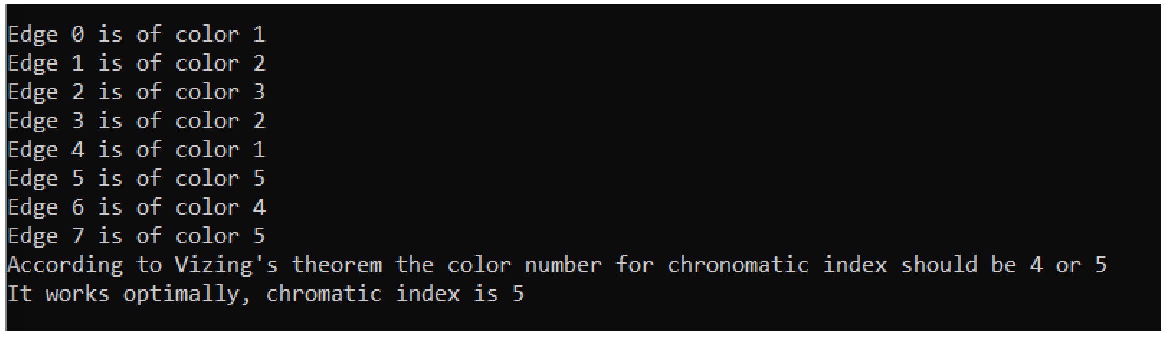


```
Edge 0 is of color 1
Edge 1 is of color 2
Edge 2 is of color 3
Edge 3 is of color 1
Edge 4 is of color 3
Edge 5 is of color 4
According to Vizing's theorem the color number for chromatic index should be 3 or 4
It works optimally, chromatic index is 4
```

FAILED: In the algorithm; according to Vizing’s theorem, optimal coloring number is specified as k and k+1, k is the maximal degree of the vertices of the graph. Although k+1 is accepted as optimal, sometimes a graph can be colored with k like in the above case. However, this can be controlled manually.

Test run-2 with 5 vertices:

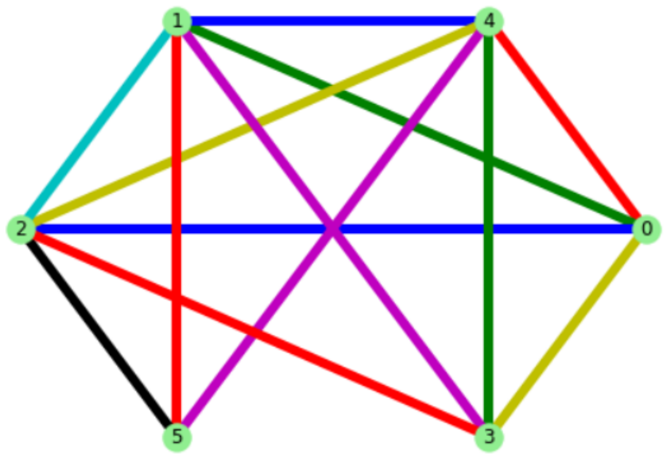


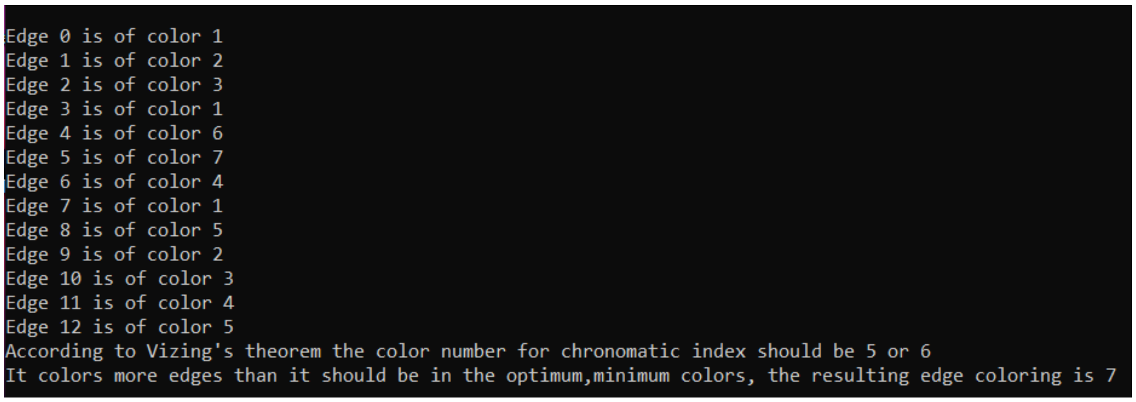


```
Edge 0 is of color 1
Edge 1 is of color 2
Edge 2 is of color 3
Edge 3 is of color 2
Edge 4 is of color 1
Edge 5 is of color 5
Edge 6 is of color 4
Edge 7 is of color 5
According to Vizing's theorem the color number for chronomatic index should be 4 or 5
It works optimally, chromatic index is 5
```

SUCCEEDED: The algorithm produced the optimal result.

Test run-3 with 6 vertices:

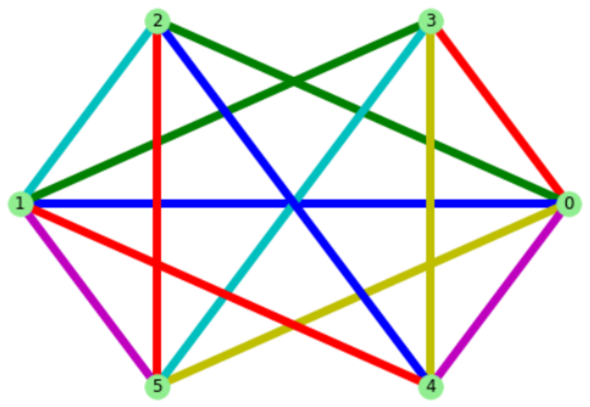


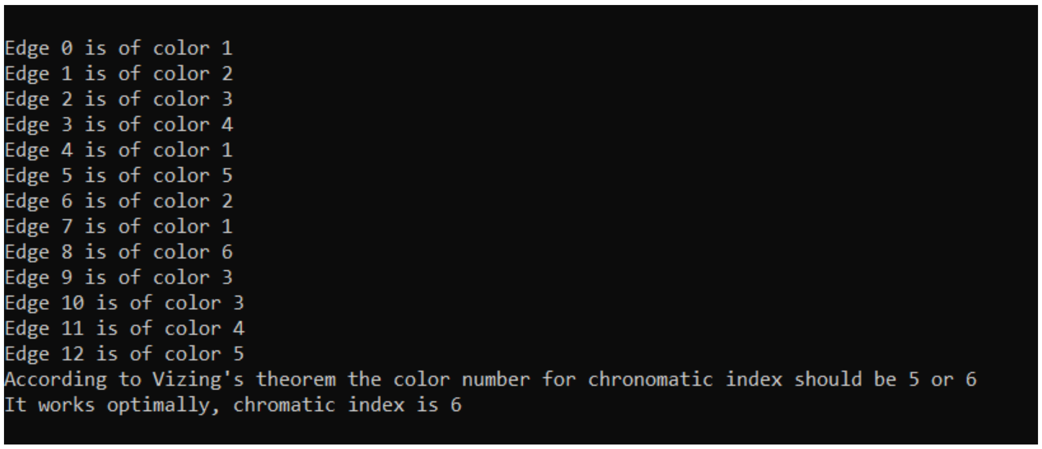


```
Edge 0 is of color 1
Edge 1 is of color 2
Edge 2 is of color 3
Edge 3 is of color 1
Edge 4 is of color 6
Edge 5 is of color 7
Edge 6 is of color 4
Edge 7 is of color 1
Edge 8 is of color 5
Edge 9 is of color 2
Edge 10 is of color 3
Edge 11 is of color 4
Edge 12 is of color 5
According to Vizing's theorem the color number for chronomatic index should be 5 or 6
It colors more edges than it should be in the optimum,minimum colors, the resulting edge coloring is 7
```

FAILED: The result is more than the optimal value, the result is k+2.

Test run-4 with 6 vertices:

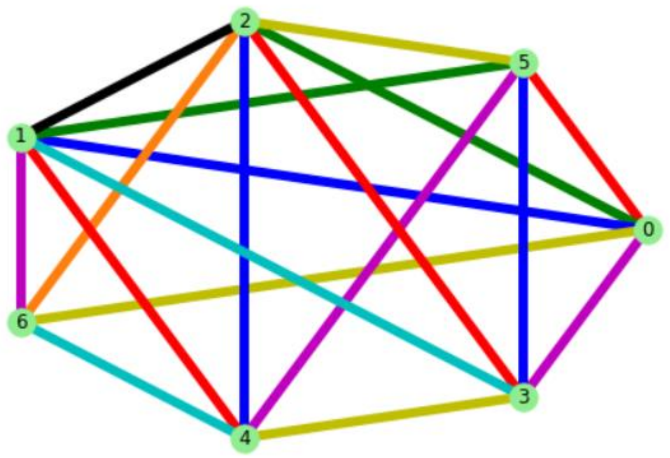


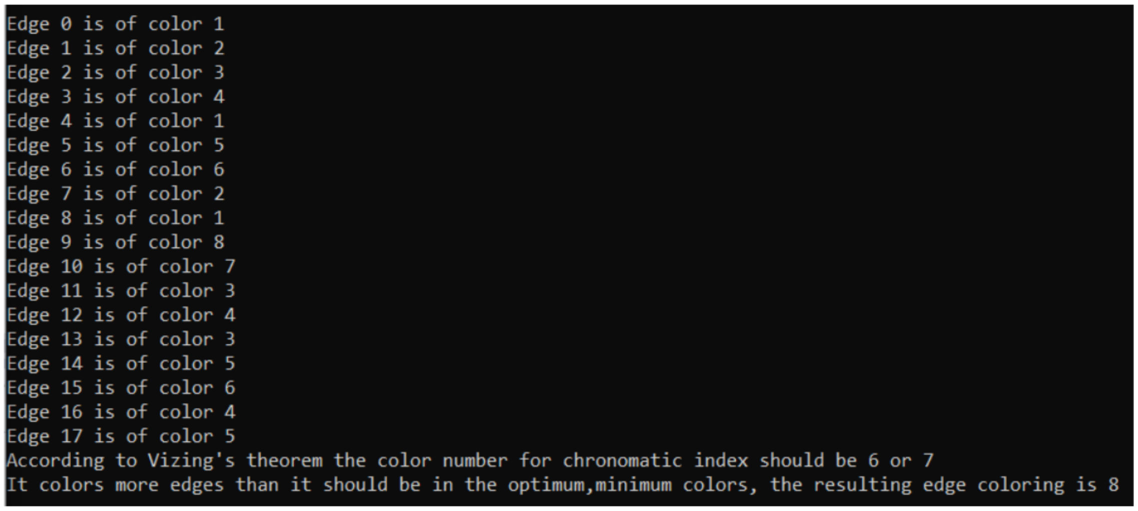


```
Edge 0 is of color 1
Edge 1 is of color 2
Edge 2 is of color 3
Edge 3 is of color 4
Edge 4 is of color 1
Edge 5 is of color 5
Edge 6 is of color 2
Edge 7 is of color 1
Edge 8 is of color 6
Edge 9 is of color 3
Edge 10 is of color 3
Edge 11 is of color 4
Edge 12 is of color 5
According to Vizing's theorem the color number for chronomatic index should be 5 or 6
It works optimally, chromatic index is 6
```

SUCCEEDED: The algorithm produced the optimal result.

Test run-5 with 7 vertices:

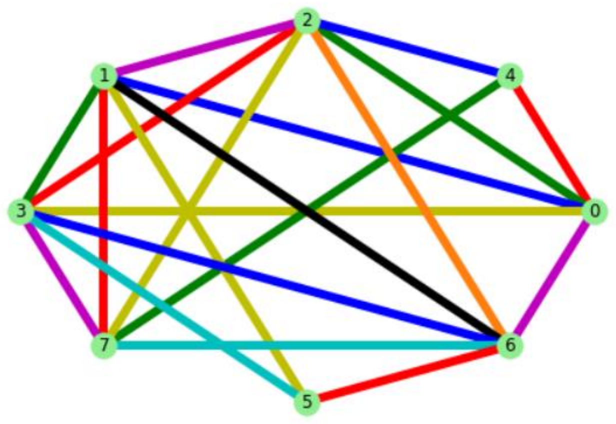


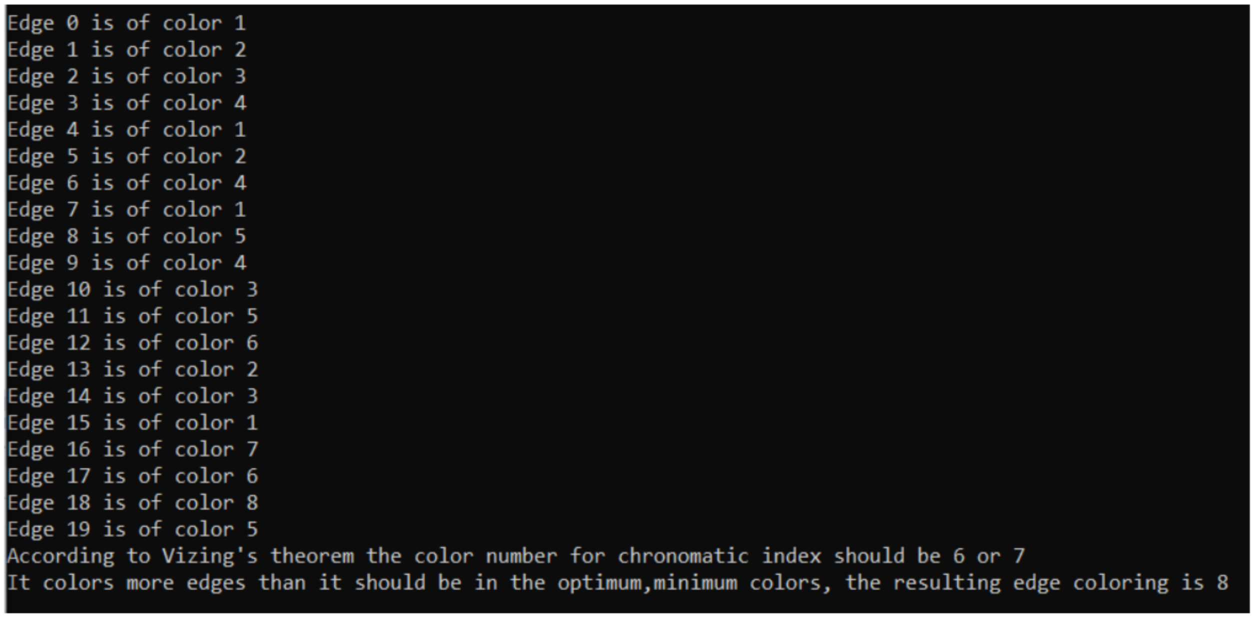


```
Edge 0 is of color 1
Edge 1 is of color 2
Edge 2 is of color 3
Edge 3 is of color 4
Edge 4 is of color 1
Edge 5 is of color 5
Edge 6 is of color 6
Edge 7 is of color 2
Edge 8 is of color 1
Edge 9 is of color 8
Edge 10 is of color 7
Edge 11 is of color 3
Edge 12 is of color 4
Edge 13 is of color 3
Edge 14 is of color 5
Edge 15 is of color 6
Edge 16 is of color 4
Edge 17 is of color 5
According to Vizing's theorem the color number for chronomatic index should be 6 or 7
It colors more edges than it should be in the optimum,minimum colors, the resulting edge coloring is 8
```

FAILED: The result is more than the optimal value, the result is k+2.

Test run-6 with 8 vertices:





```
Edge 0 is of color 1
Edge 1 is of color 2
Edge 2 is of color 3
Edge 3 is of color 4
Edge 4 is of color 1
Edge 5 is of color 2
Edge 6 is of color 4
Edge 7 is of color 1
Edge 8 is of color 5
Edge 9 is of color 4
Edge 10 is of color 3
Edge 11 is of color 5
Edge 12 is of color 6
Edge 13 is of color 2
Edge 14 is of color 3
Edge 15 is of color 1
Edge 16 is of color 7
Edge 17 is of color 6
Edge 18 is of color 8
Edge 19 is of color 5
According to Vizing's theorem the color number for chronomatic index should be 6 or 7
It colors more edges than it should be in the optimum,minimum colors, the resulting edge coloring is 8
```

FAILED: The result is more than the optimal value, the result is k+2.

6) Discussion

Edge coloring problem is a NP complete problem which is reduced from 3-SAT. No algorithm is found such that it can solve edge coloring in polynomial time. However, there are polynomial time algorithms that are found for solving the problem that does not guarantee the optimal minimum coloring result in each iteration. The algorithm that we are using is a heuristic algorithm. In this algorithm, we mainly used depth first search for traversing the graph. It might not find the optimal minimum edge coloring in some cases.

The running time of the algorithm comes as **O(V*E^2).** Although it seems like the whole algorithm is O(V+E) *O(E^2), it can be observed that since we check every time the current vertex is visited or not and decide to go over other processes then the algorithm turns out to be O(V)*O(E^2) +O(E)*O(1) = O(V*E^2).

The heuristic algorithm has performed edge coloring for lots of random graphs with multiple iterations. Its correctness is checked every random graph for its each iteration. It never returns false. In the performance analysis part, we see that the number of edges is equal to approximately 0.32V^2 on average in practice. In addition to this; by using log-log plots it is seen that the experimental running time is O(V^2.85) instead of O(V^5). O(V^5) is found as taking E=V^2 in worst case. There is a significant difference between experimental and theoretical running times especially when the graph has lots of vertices. With the outcomes of this project further analysis and experimental studies might be conducted to find more efficient results.