SABANCI UNIVERSITY

FALL 2020

CS 301 ALGORITHMS PROJECT REPORT

Begüm Altunbaş 26824

Ülkü Eylül Şahin 26786

Boğaçhan Arslan 26569

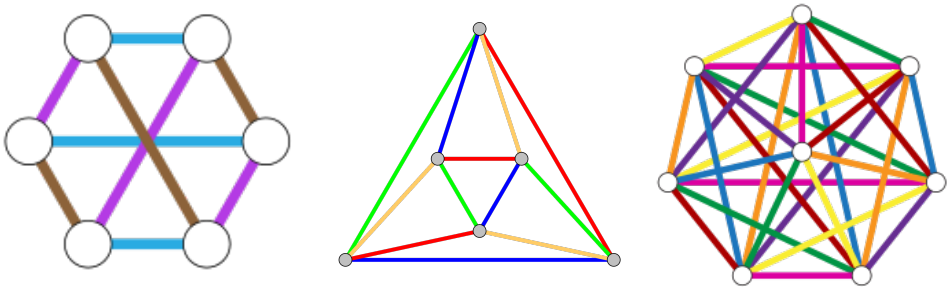
Berk Öztaş 25260

CHROMATIC INDEX-EDGE COLORING

1) PROBLEM DESCRIPTION:

Edge Coloring, also called as Chromatic Index, is a NP-complete problem finding the minimum number of colors necessary to color each edge of a graph in a way that no edges connected to the same vertex have the same color. Vizing's theorem states that the chromatic index can be either one of the two classes: **Δ** or **Δ+1** where **Δ** represents the largest vertex degree of the graph (1).

Here are three basic examples of graphs with colored edges;



Chromatic index: 3 *Chromatic index: 4* *Chromatic index: 6*

Chromatic index problem is also used in several real-life applications, especially in scheduling problems and frequency assignment for fiber optic networks. For example, when compilers of programming languages are generating codes, variables are being allocated in registers and deciding which register to store the variable is an edge coloring problem. The problem is also used in wavelength assignment in multi-fiber WDM networks (2) by generalizing the problem to edge coloring problem.

PROOF:

In order to prove a problem to be NP-complete it should satisfy the followings:

1-Problem needs to be in NP

2-A NP-complete problem is reducible to this problem in polynomial time

A- Edge Coloring is in NP:

To check whether a problem is contained within the NP set, we must demonstrate the existence of a verification algorithm (a solution checker) that, given an instance of the problem and a sample solution or guess, the algorithm would provide whether the solution is correct or not in polynomial time.

To exemplify such a solution verifier, we provide a sample solver that checks if the chromatic index given as the solution will be exceeded in a coloring attempt or not. The definition of the problem requires us to ensure that the given chromatic index value will provide enough coloring options that with an appropriate mechanism of coloring, no adjacent edge (edges that share a common vertex) will have the same color. The following algorithm inspects the mentioned problem on a graph G with the vertex and edge set V & E, with the chromatic index value k.

```
colors = [red,blue,yellow,green,...]
Func verify_chromatic_index(G,k):
     Generate Map m that stores <vertex,color> pairs
     For each vertex v in G.V:
          available = colors[:k]
          For each edge e that is connected to v:
               Bool added = false
               For color in available:
                    If <v,color> not exists in m:
                         m.add(<v,color>)
                         available.remove(color)
                         Added = true
               If not added:
                    print("The solution is false")
     print("All edges colored successfully")
```

When inspected, it is obvious that the checker algorithm has a polynomial time bound. It works for each Vertex; Edge times and goes through all available colors. Hence it is possible to conclude that verify_chromatic_index(G,k) = O(V*E*k). The problem is thus in NP.

B- Reducing a NP-Complete Problem to Edge Coloring:

We are required to perform reduction from the known NP-complete problem "3SAT" in order to prove Edge Coloring is NP-hard.

To briefly explain the 3SAT problem; Given a set S consisting of boolean variables in conjunctive normal form, where each clause contains three literals which are either an element of S, or its negation. The 3SAT problem is to decide whether there is a truth value assigned to each boolean variable to satisfy all conjuncts in the formula, and the formula evaluates to "true".

In order to satisfy the parity condition, we will use the lemma given in Isaacs (4);

Lemma: *Let G be a cubic, 3-edge-colored graph and V'⊆V (G) a set of vertices of G. Let E'⊆E(G) be the set of edges of G which connect V' to the remainder of the graph. If the number of edges of color i in E' is* k_i (i = 1, 2, 3)*, then*

$$k1 \equiv k2 \equiv k3 \pmod 2$$

Proof: *If E12 is the set of edges of G which are colored with color 1 or 2, then E12 consists of a collection of cycles. Thus, E12 meets E0 in an even number of edges, and so* $k1 + k2 \equiv 0 \pmod 2$ *which gives* $k1 \equiv k2 \pmod 2$*. Similarly,* $k2 \equiv k3 \pmod 2$*.*

Describing the components used in reduction to 3SAT: Constructing a graph G, we will show that it is possible to 3-edge-color G if and only if an instance of 3SAT is satisfiable. We will be using pairs of edges in order to carry information in the theorem. In graph G mentioned above, true (T) and false (F) values are assigned to pairs of edges, indicating that the edges have different colors if the value is F, and same color if the value is T.

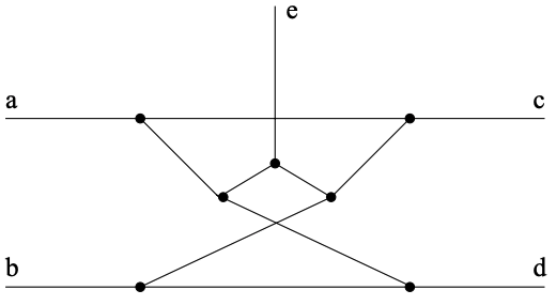


Figure 1

Loupekine (4) composed a large group of graphs with chromatic index 4 and during the process, he used the inverting component represented above in Figure 1. We can check if this component is 3-edge-colored using the parity condition shown before. The only restriction coloring this graph is; one of the edge pairs (such as {a,b} or {c,d}) must have the same colors and the remaining 3 edges must have distinct colors. This component will be representing a truth value if (a,b) is considered as the input pair and (c,d) is considered as the output pair. This truth value depends on the values of edge pairs.

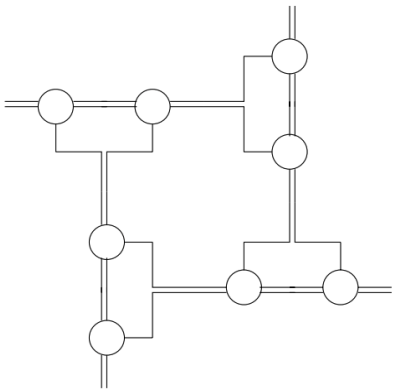


Figure 2

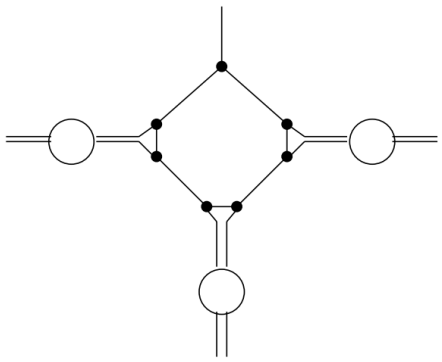


Figure 3

Figure 2 from Holyer's (6) paper shows the truth values of each variable ui being represented by a variable-setting component. The component shown in figure has 4 edge pairs as output (for 8 inverting components) therefore we can state that the figure is composed of *2n* inverting components and *n* output pairs. The component representing ui should have as many output pairs as there are appearances of ui among the clauses of a 3SAT problem. In any edge-coloring of a component, all the output pairs must represent the same truth value.

The truth of each clause cj will be tested by a satisfaction-testing component as shown in Figure 3 (6) This component can be edge-colored if and only if the three input pairs of edges do not all represent F.

4. **Conclusion of Proof**

Theorem: Chromatic index (edge coloring) is a NP-Complete problem.

Proof: We showed that this problem is in NP at section A. Later at section B, we showed a polynomial reduction from problem 3SAT. Using the reduction of 3SAT to cubic edge coloring of Hoyler (6): *Consider an instance C of 3SAT and construct from it a graph G as follows. For each variable ui take a variable-setting component Ui with one output pair of edges associated with each appearance of ui among the clauses of C. Take also a satisfaction testing component Cj for each clause cj . Suppose literal lj;k in clause cj is the variable ui. Then identify the kth input pair of Cj with the associated output pair of Ui . If, on the other hand, lj;k is ui, then insert an inverting component between the kth input pair of Cj and the associated output pair of Ui. The resulting graph H still has some connecting edges unaccounted for. The cubic graph G is formed from two copies of H by identifying the remaining connected edges in corresponding pairs. The graph G has an edge-coloring if and only if the collection C of clauses is satisfiable, as can be verified using the properties of the components developed above. Moreover, the graph G can be produced from C using a polynomial time algorithm.*

Chromatic index problem is definitely an NP-Complete problem since it has to decide whether the chromatic index is **Δ** or **Δ+1** which is reducible to the chromatic index of cubic graphs (decide whether the chromatic index is 3 or 4) therefore we can generalize this to chromatic index problem for all possible graphs.

2) ALGORITHM DESCRIPTION:

There is no polynomial time algorithm proven to solve the Chromatic Index problem for all kinds of graphs. However, there are several heuristic algorithms developed to solve this problem that can execute in polynomial or higher time complexities, but do not always yield to the optimal solution. In this step we will be analyzing a previously designed algorithm, extended for graph population and various cases, for Edge Coloring problem. The graphs (the input) are structured as a 2-dimensional integer array (matrix), each row representing an edge as an array, with the first two elements as the vertex numbers it is connected to, and the third element as the color value.

Here are the steps of the greedy algorithm:

1. Assign color to current edge as c=1
2. Traverse all edges and check if adjacent edges have the same color, if so, discard this color and go to the flag again and try the next color.

```
void EdgeColor(int ed[][3], int e) {
    int i, c, j;
    for(i = 0; i < e; i++) {
        if(ed[i][0] == -1 && ed[i][1] == -1) continue;
        c = 1; //assign color to current edge as c i.e. 1 initially.
        flag:
        ed[i][2] = c;
        //If the same color is occupied by any of the adjacent edges, then discard this color and go to flag
            again and try next color.
        for(j = 0; j < e; j++) {
            if(ed[i][0] == -1 && ed[i][1] == -1) continue;
            if(j == i)
                continue;
            if(ed[j][0] == ed[i][0] || ed[j][0] == ed[i][1] || ed[j][1] == ed[i][0] || ed[j][1] == ed[i][1])
                {
                if(ed[j][2] == ed[i][2]) {
                    c++;
                    goto flag;
                }
            }
        }
    }
}
```

Algorithm works in this manner:

1. **(Not part of the heuristics algorithm)** Take vertex number as input and create random number of edges within range [v,v(v-1)/2]. The graph matrix is allocated according to size parameters, but all values are initialized as -1 until it is populated.

2. **(Heuristics algorithm starts here)** The matrix is iterated for all of its edges.

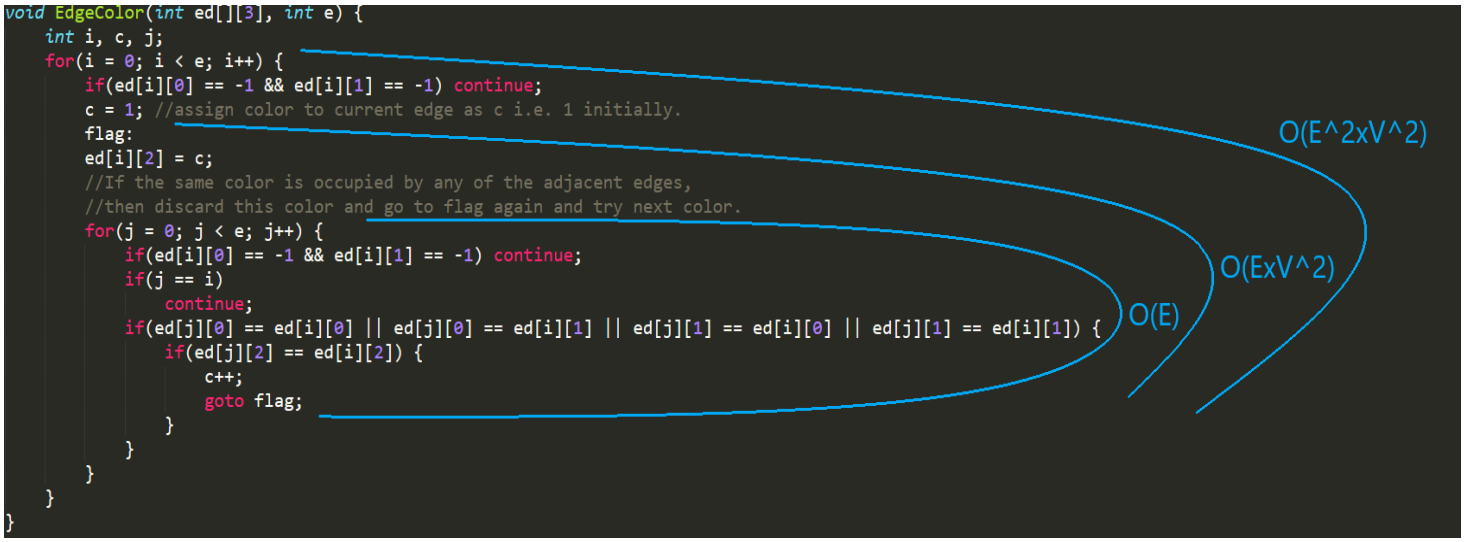
3. For each edge being iterated, we first set its color to 1 initially.

4. Then for that edge, we iterate all the edges again and check whether if any adjacent edges have the same color.

5. If so, we increment the color value of the current edge by 1 and then repeat step 4 until there are no adjacent edges that share the same color as this edge.

6. In the end, we are sure that each adjacent edge has a unique color value.

7. 3) ALGORITHM ANALYSIS:



$$\boldsymbol{O(E)} \times \boldsymbol{O(V^2)} \times \boldsymbol{O(E)} = \boldsymbol{O(E^2 \times V^2)}$$

- The inner most loop traverses all edges in the graph, resulting in $O(E)$ time complexity.
- The loop with $goto\ flag$; will start by 1 ite<n when no edge is colored yet and increase in count by the number of colors present in the graph. Potentially reaching V iterations. Resulting in: $O(\sum_{i=1}^{V} i) = O(\frac{V \times (V+1)}{2}) = O(V^2)$. Adding the inner loop into the equation makes it $O(E) \times O(V^2) = O(E \times V^2)$.
- Outer loop will also traverse all edges, resulting in $O(E)$ time complexity. Considering the inner loops makes the total: $O(E) \times O(E \times V^2) = O(E^2 \times V^2)$.
- Worst Case: Complete graph: E = V*(V-1)/2. **=>** $O(V^2 \times \big(V * (V-1)\big)^2 = O(V^6)$
- Best Case: E = V-1. Time complexity = $O(V^2 \times (V-1)^2) = O(V^4)$

4) EXPERIMENTAL ANALYSIS OF THE ALGORITHM:

In order to analyze our algorithm's **running time,** we need mathematical concepts such as STD, standard error, sample mean and confidence level intervals.

STANDARD DEVIATION:

$$SD = \sqrt{\frac{\sum |x - \bar{x}|^2}{n}}$$

n =number of elements in sample

x̄=mean of samples

Σ= summation sign-Sigma
S = standard deviation

```
float STD (vector<float>&data)
{
    float sum =0.0, mean,std=0.0 ;
    for (int i=0; i<data.size(); i++)
    {
        sum += data[i];
    }
    mean=sum/data.size();
    for (int j=0 ; j< data.size(); j++)
    {
        std += pow (data[j]-mean,2);
    }
    std = sqrt(std/data.size());


    return std;
}
```

Code implementation of Standard deviation calculation function.

STANDARD ERROR:

$$SE = \frac{\sigma}{\sqrt{n}}$$

← Standard deviation

← Number of samples

```
float StandardError(float std, int n)
{
    return std/ sqrt (n) ;
}
```

Code implementation and formula of Standard Error calculation

ERROR INTERVALS:

```
void RunningTime (vector <float> &all_times_observed)
{
    float total_time= 0.0 ;
    int size= all_times_observed.size();
    for (int i=0 ; i <size ; i++)
    {
        total_time+= all_times_observed[i];
    }
    float std = STD(all_times_observed);
    float sample_mean= total_time/size; // sample mean

    const float tval90=1.645; // t-value %90 confidence
    const float tval95= 1.96 ; //t-val %95

    float std_error = StandardError(std, size) ;

    float uppermean_90 = sample_mean+tval90*std_error ;
    float lowermean_90 = sample_mean-tval90*std_error ;

    float uppermean_95 = sample_mean+tval95*std_error ;
    float lowermean_95 = sample_mean-tval95*std_error ;



    cout<< "Mean time: "<< sample_mean<<"ms" <<endl;
    cout<< "SD: "<< std<<endl  ;
    cout<<"standard error"<< std_error<<endl ;
    cout<< "with %90 confidence level error interval "<< lowermean_90<< "-" <<uppermean_90<<endl ;
    cout<< "with %95 confidence level error interval "<< lowermean_95<< "-" <<uppermean_95<<endl ;

}
```

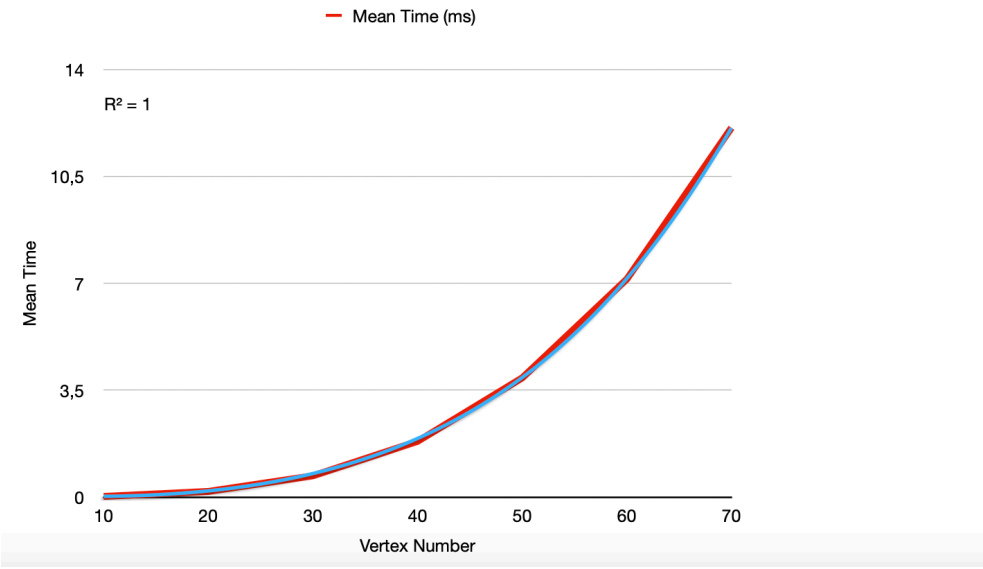
We choose confidence levels as %90, %95 since they are the most widely used ones. Their t-values are 1.645 and 1.96 respectively. Above code calculates mean time, standard error using std, upper and lower error rates for given confidence levels according to running time of the algorithm.

We know that our time complexity ($O(V^2E^2)$) depends on both the vertex number and the edge count of the graph. To experimentally prove the running time, we have illustrated the measured, actual running times by changing initially only the vertex count, then the edge count.

Iteration Count: 100, Edge count is fixed

measures

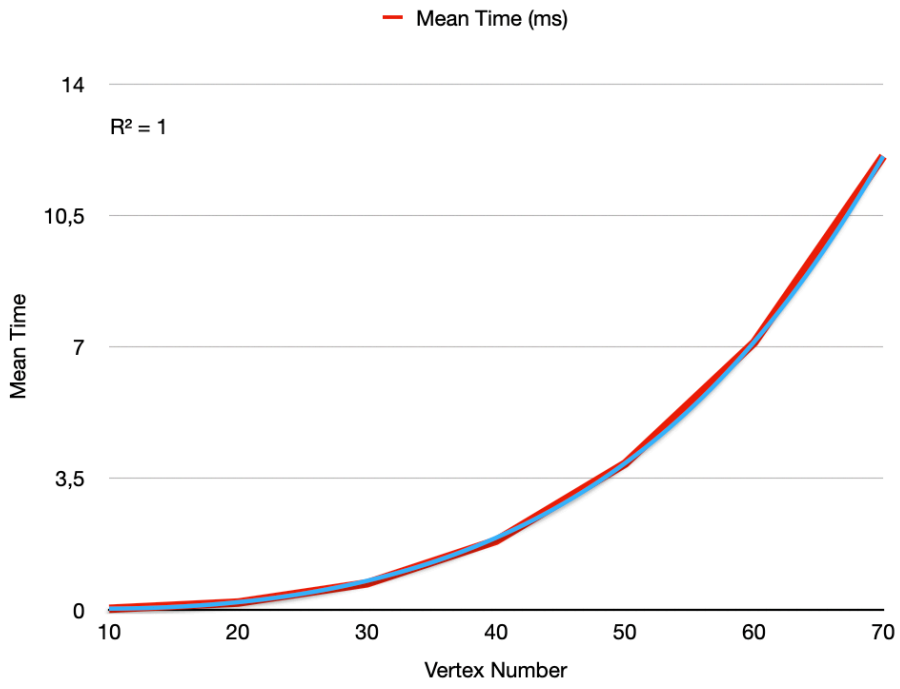
Size(V)	Mean Time (ms)	Standard Deviation	Standard Error	90% CL	%95 CL
10	0,02521	0,00501656	0,000501656	0,0243848-0,0260352	0,0242268-0,0261932
20	0,18853	0,00593036	0,000593036	0,187554-0,189506	0,187368-0,189692
30	0,70051	0,0237594	0,00237594	0,696602-0,704418	0,695853-0,705167
40	1,82735	0,0788272	0,00788272	1,81438-1,84032	1,8119-1,8428
50	3,9065	0,226839	0,0226839	3,86918-3,94381	3,86204-3,95096
60	7,13903	0,25459	0,025459	7,09715-7,18091	7,08913-7,18893
70	12,0936	0,419862	0,0419862	12,0245-12,1626	12,0113-12,1759



Iteration Count= 250, Edge count is fixed

measures

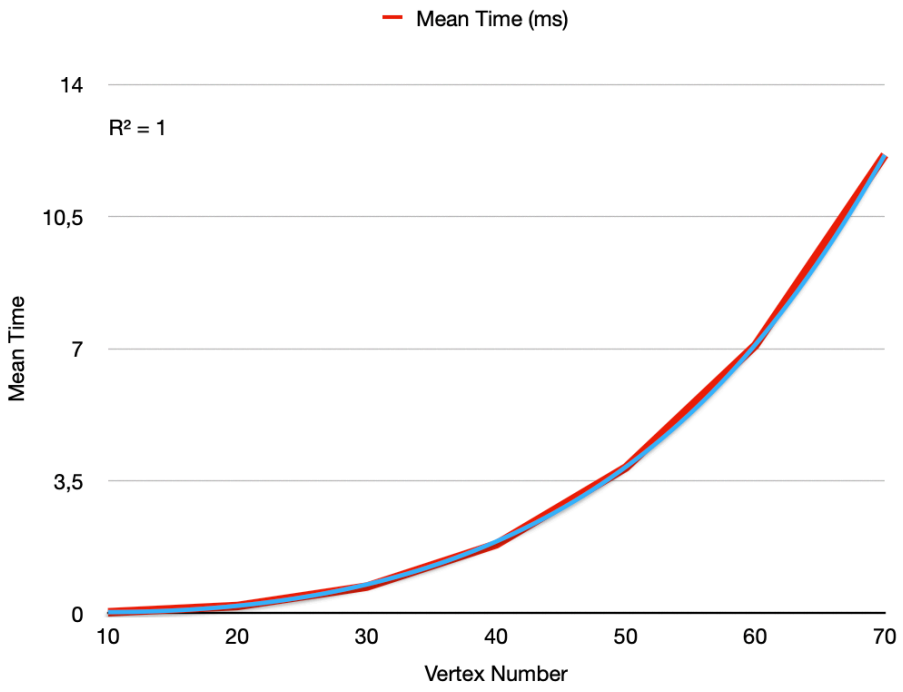
Size(V)	Mean Time (ms)	Standard Deviation	Standard Error	90% CL	%95 CL
10	0,02346	0,00364094	0,000230273	0,0230812-0,0238388	0,0230086-0,0239113
20	0,189188	0,00821125	0,000519325	0,188334-0,190042	0,18817-0,190206
30	0,701056	0,0281754	0,00178197	0,698125-0,703987	0,697563-0,704549
40	1,83438	0,100483	0,00635509	1,82392-1,84483	1,82192-1,84683
50	3,89069	0,161848	0,0102361	3,87385-3,90753	3,87063-3,91075
60	7,11277	0,276991	0,0175184	7,08395-7,14159	7,07843-7,14711
70	12,0729	0,565026	0,0357354	12,0141-12,1317	12,0028-12,1429



Iteration Count=500, Edge count is fixed

measures

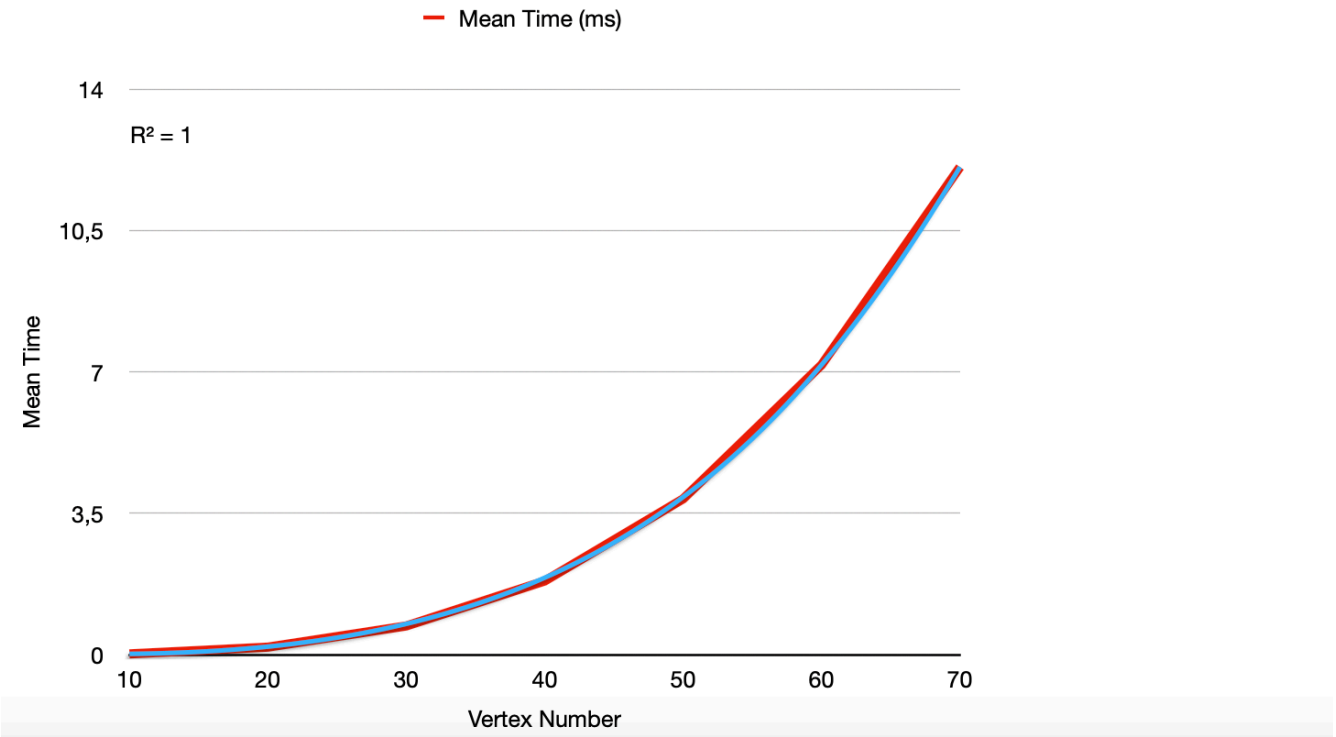
Size(V)	Mean Time (ms)	Standard Deviation	Standard Error	90% CL	%95 CL
10	0,023846	0,00439707	0,000196643	0,0235225-0,0241695	0,0234606-0,0242314
20	0,18995	0,0115784	0,000517802	0,189098-0,190802	0,188935-0,190965
30	0,703514	0,0432329	0,00193343	0,700333-0,706694	0,699724-0,707303
40	1,8227	0,0744069	0,00332758	1,81723-1,82818	1,81618-1,82922
50	3,85642	0,175302	0,00783976	3,84352-3,86931	3,84105-3,87178
60	7,0908	0,274233	0,0122641	7,07063-7,11098	7,06676-7,11484
70	12,1339	0,607636	0,0271743	12,0892-12,1786	12,0806-12,1871



Iteration count 1000, Edge count is fixed

measures

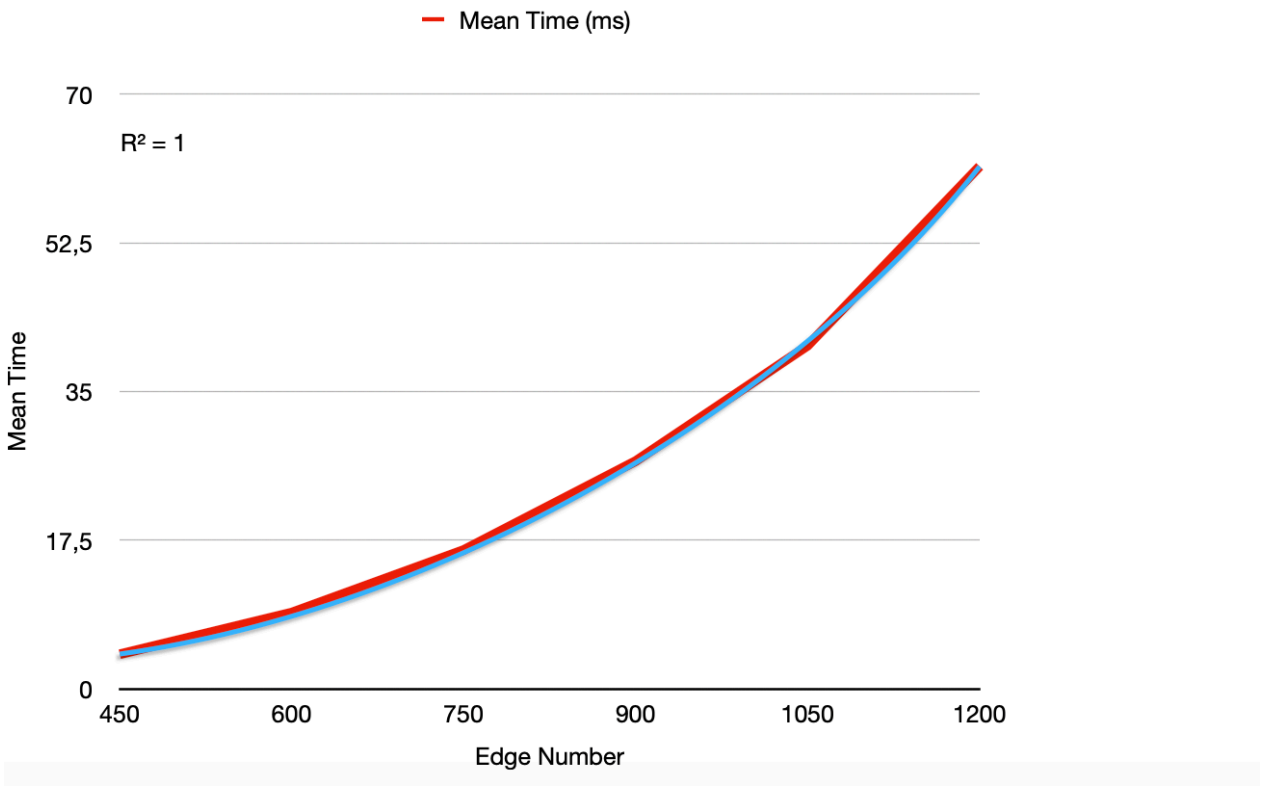
Size(V)	Mean Time (ms)	Standard Deviation	Standard Error	90% CL	%95 CL
10	0,023122	0,0038214	0,000120843	0,0229232-0,0233208	0,0228851-0,0233588
20	0,190676	0,0114463	0,000361963	0,190081-0,191272	0,189967-0,191386
30	0,714929	0,0441417	0,00139588	0,712633-0,717226	0,712194-0,717665
40	1,82686	0,0899421	0,00284422	1,82218-1,83154	1,82129-1,83244
50	3,85992	0,165795	0,00524289	3,8513-3,86855	3,84965-3,8702
60	7,17666	0,254905	0,00806081	7,1634-7,18992	7,16086-7,19246
70	12,0721	0,429832	0,0135925	12,0498-12,0945	12,0455-12,0988



Iteration Count=100, Vertex count = 50

measures

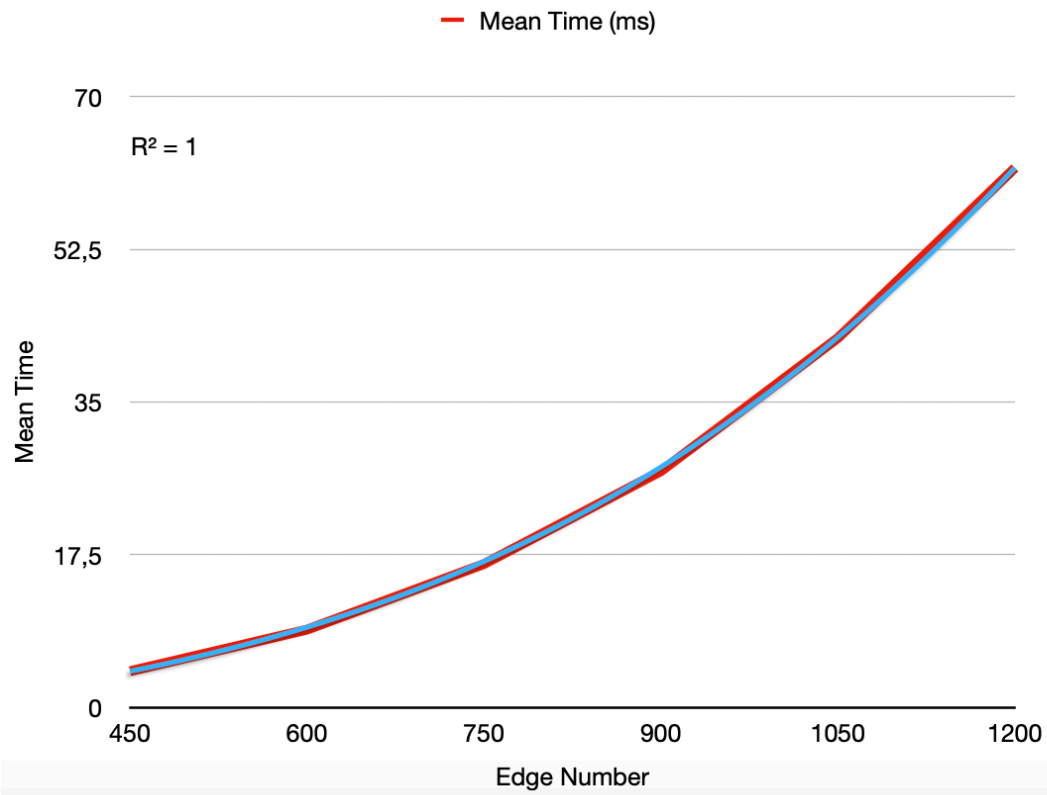
Size(E)	Mean Time (ms)	Standard Deviation	Standard Error	90% CL	%95 CL
450	4,07149	0,144302	0,0144302	4,04775-4,09523	4,04321-4,09977
600	8,98063	0,31458	0,031458	8,92888-9,03238	8,91897-9,04229
750	16,3526	0,477021	0,0477021	16,2742-16,4311	16,2592-16,4461
900	26,8238	0,800538	0,0800538	26,6922-26,9555	26,6669-26,9807
1050	40,4104	0,579925	0,0579925	40,315-40,5058	40,2967-40,5241
1200	61,5232	0,820354	0,0820354	61,3883-61,6582	61,3624-61,684



Iteration Count=250, Vertex count = 50

measures

Size(E)	Mean Time (ms)	Standard Deviation	Standard Error	90% CL	%95 CL
450	4,14458	0,182668	0,0115529	4,12558-4,16358	4,12194-4,16722
600	8,90895	0,51998	0,0328864	8,85485-8,96304	8,84449-8,9734
750	16,4327	0,772473	0,0488555	16,3524-16,5131	16,337-16,5285
900	27,0806	0,994181	0,0628775	26,9772-27,184	26,9574-27,2038
1050	42,3559	1,43512	0,0907651	42,2066-42,5052	42,178-42,5338
1200	61,8481	1,41925	0,0897614	61,7004-61,9957	61,6721-62,024



CORRECTNESS:

Our algorithm doesn't always give the optimal solution however to test its correctness and prove that it gives the right solution we created several functions. Here is the correctness function: This function checks whether the output graph has different colors for adjacent edges.

```
void checkSolution2(int ed[][3]){
    for(i = 0; i < e; i++) {
        if(ed[i][0] == -1 && ed[i][1] == -1) continue;
        for(j = 0; j < e; j++) {
            if(ed[i][0] == -1 && ed[i][1] == -1) continue;
            if(j == i)
                continue;
            if(ed[j][0] == ed[i][0] || ed[j][0] == ed[i][1] || ed[j][1] == ed[i][0] || ed[j][1] == ed[i][1])
                {
                if(ed[j][2] == ed[i][2]) {
                    cout << "****" <<endl;
                    cout << "The solution is INCORRECT !!!!"<< endl;
                    cout << "****" <<endl;
                    return;
                }
            }
        }
    }
    cout << "****" <<endl;
    cout << "The solution is Correct !!!!"<< endl;
    cout << "****" <<endl;
}
```

```
int maxDegree(int ed[][3],int v,int randE){
    vector<int> edgeCount(v,0);

    for(i = 0; i < randE; i++) {
        edgeCount[ed[i][0]]++;
        edgeCount[ed[i][1]]++;
    }
    int max = 0;
    for (int k = 0; k < v; ++k) {
        if(edgeCount[k]>max){
            max = edgeCount[k];
        }
    }
    return max;
}

int maxColor(int ed[][3], int randE){
    int max = 0;
    for(i = 0; i < randE; i++) {
        if(ed[i][2]>max){
            max = ed[i][2];
        }
    }
    return max;
}
```

Functions on the left returns the max degree of the output graph and max number of colors used in execution. According to Vizing's theorem minimum numbers of colors are either **Δ** or **Δ+1** where **Δ** represents the largest vertex degree of the graph. The difference between the max degree and the max color count identifies if the generated solution is optimal or not.

```
****
Correctness ratio is: %100
Optimality ratio is: %85.4
Program ended with exit code: 0
```

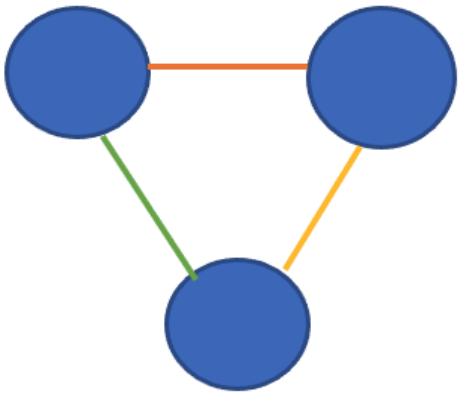
```
void measure_correctness(vector<int> & correct){
    int sumcorrect=0;
    for (int k = 0; k < correct.size(); ++k) {
        if(correct[k]) sumcorrect++;
    }
    cout << "Correctness ratio is: %" << double(sumcorrect*100)/correct.size()<< endl;
}

void measure_optimality(vector<int> & degreeDifferences){
    int sumcorrect=0;
    for (int k = 0; k < degreeDifferences.size(); ++k) {
        if(degreeDifferences[k]<=1) sumcorrect++;
    }
    cout << "Optimality ratio is: %" << double(sumcorrect*100)/degreeDifferences.size()<< endl;
}
```

5) TESTING

In order to apply blackbox testing, we implemented a random graph generating option with a given vertex as input in the main function of our program. Number of edges and the nodes they are connected to are assigned randomly. Some of the cases which failed or succeeded to find the chromatic index are mentioned below.

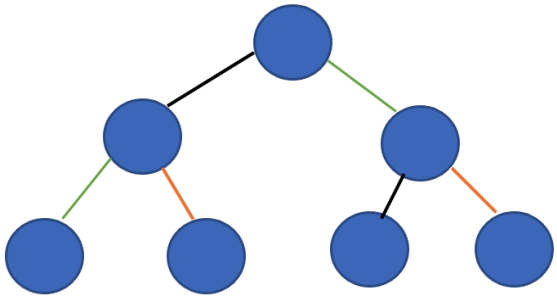
We also tried some outstanding graphs manually in order to test every edge case we can think of. We worked our algorithm for tree structured, connected, disconnected, cyclic, acyclic, graphs and also graphs containing a large sample of vertices. Some of the graphs we tested are shown below.

```
int randE=random.RandInt(n,n*(n-1)/2);
int ed[randE][3];

int cnt=0;
for(i = 1; i <= n; i++) {
    for(j = i+1; j <= n; j++) {
        ed[cnt][0] = -1;
        ed[cnt][1] = -1;
        ed[cnt][2] = -1;
        cnt++;
        if(cnt == randE) break;
    }
    if(cnt == randE) break;
}

for(i=0;i<randE;i++)
{
    cout<<"\nEnter edge vertices of edge "<<i+1<<" :";
    //cin>>x>>y;
    int x= random.RandInt(0,n-1);
    int y= random.RandInt(0,n-1);
    while (x==y || coordinateVisited(storedcoordinates, x, y) || coordinateVisited(storedcoordinates, y, x))
    {
        x= random.RandInt(0,n-1);
        y= random.RandInt(0,n-1);
    }
    cell.x = x;
    cell.y = y;

    ed[i][0]=x;
    ed[i][1]=y;
```



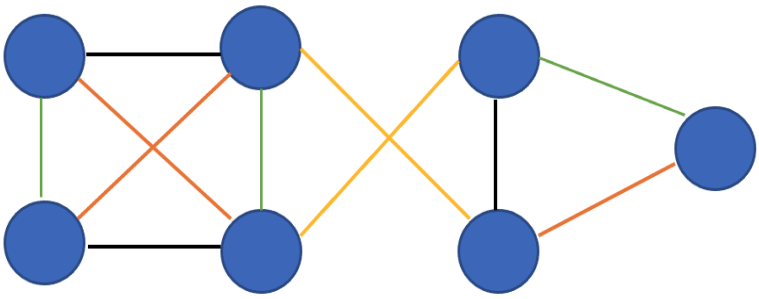
(a)

Succeeded: Chromatic index of this graph is 3, degree is 2.



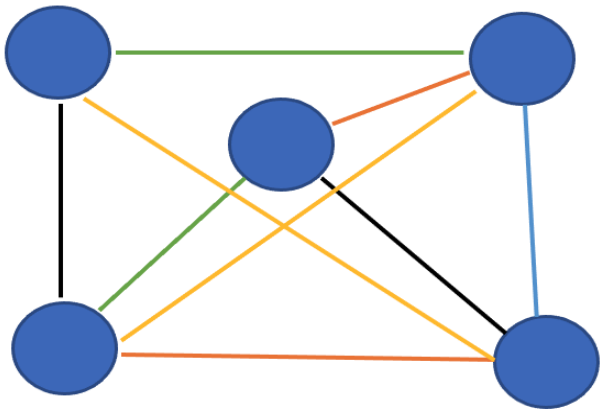
(b)

Succeeded: Chromatic index of this graph is 3, degree is 3.



(c)

Succeeded: Chromatic index of this graph is 4, degree is 4.



(d)

Failed: Maximum degree of graph is 4, correct chromatic index is 5 but our algorithm returned 6.

6) DISCUSSION

To conclude, Chromatic Index (Edge Coloring) is an NP-Complete problem reduced from 3SAT. There is no polynomial time algorithm proven to solve the Chromatic Index problem for all kinds of graphs. There isn't even any polynomial time algorithm proven to solve this problem for cubic graphs (decide between 3 or 4). There are some heuristics to solve this algorithm using different approaches. In this project, we analyzed a brute force algorithm with time complexity $O(E^2 \times V^2)$. This algorithm guarantees to color the edges correctly but doesn't always yield the optimal solution (find the minimum number of colors needed).

Correctness of the algorithm is measured by the *measure_correctness* function which returns whether the edges were colored correctly. By trying this algorithm with many different randomized graphs, we always obtained a correct solution.
In order to check if the algorithm yields the optimal solution; we created 2 functions maxColor and maxDegree, since by Vizing's Theorem it is known that the chromatic index is whether the max degree of the graph (**Δ),** or **Δ+1** and by the function *measure_optimality* we returned whether the solution was optimal. Approximately 85% of the time, our polynomial heuristics approach provides us an optimal value, solving the initial problem at hand. It is not perfect, yet when inspected, we often generate only 1 or 2 additional colors when compared to the optimal chromatic index.

FOOTNOTES:

1) https://www.sciencedirect.com/science/article/pii/S0012365X14000752

2) http://www.optimization-online.org/DB_FILE/2005/03/1096.pdf

3)https://www.geeksforgeeks.org/edge-coloring-of-a-graph/?ref=rp

4) R. Isaacs, Innite families of non-trivial trivalent graphs which are not Tait colorable, Amer. Math. Monthly, 82 (1975), pp. 221-239.

(5) Loupekine's snarks: A bifamily of non-Tait-colorable graphs, unpublished.

(6)https://pdfs.semanticscholar.org/37c9/2599655b9f66850f1e4e0e48ad045abad8f6.pdf

(7) https://www.geeksforgeeks.org/np-completeness-set-1/

(8)https://www.tutorialspoint.com/cplusplus-program-to-perform-edge-coloring-on-complete-graph