Sabancı University

Spring 2018

CS301 - Algorithms

Project Report

3-COL

BORA İKİZOĞLU, EMİR ALAATTİN YILMAZ, SELİN SEZER, CAN BERK GÜZGEREN

PROBLEM DESCRIPTION

A proper graph-colouring is the assignment of colors to the vertices of a graph under the condition that no adjacent vertices joined by an edge have the same color. More formally,
An n-node undirected graph G(V,E) with node set V and edge set E, an assignment of a color ϕ_v, $\phi_v \neq \phi_w$ for every $(v,w) \in E$.

A graph colouring that properly uses k colors is called k-colouring. The k-colouring decision problem asks, for each graph, G, is G k-colourable. Call this problem k-COLOR.

In this project, 3-COLOR will be examined, therefore our problem can be expressed as follows:

On an n-node undirected graph G(V,E) with node set V and edge set E, can each node of G(V,E) be assigned exactly one of three colors - Red, Blue, Green - in such a way that no two nodes which are joined by an edge, are assigned the same color?

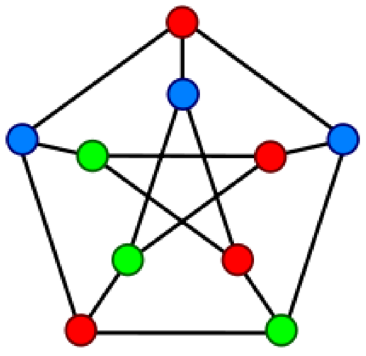


Fig.1. An example of 3 colourable graph

3-COL is NP-complete.

Proof.

In order to prove a problem is NP-complete

1- Problem needs to be shown in NP.

2- It needs to be reduced a NP-Complete problem in polynomial time.

1. In order to prove that the 3-COL is in NP, our verifier takes a graph G(V;E) and a colouring c, and checks in $O(n^2)$ time whether c is a proper colouring by checking if the end points of every edge $e \in E$ have different colors.

To prove that 3-COL is NP-hard, we will give a reduction from 3-SAT to 3-COL in polynomial time.

1. Let's say that Φ be a 3-SAT instance, and $C_1, \ldots, C_k$ are the clauses of Φ defined with the variables $\{x_1,\ldots,x_n\}$ (for example: C_2: x_1 v x_2' v x_3)

2. The graph G(V,E) that we will implement has two rules:
   It will establish the truth assignment for $x_1, x_2, \ldots, x_n$ via the colors of the vertices of G.
   It needs to capture the satisfiability of every clause Ci in 3-SAT instance.

3. To satisfy the rules, we will create a triangle in G with three vertices {T, F, B} that is T for True, F for False, B for Base.

4. We can also think T, B, F as the set of colors we will use to colour the vertices of G.

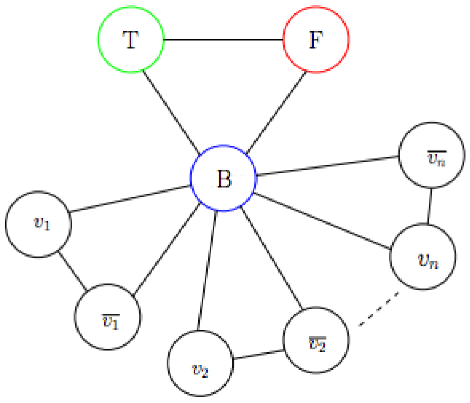


Fig.2. Truth assignment of the literals in G with 3-color

5. We said that {T, B, F} is a part of G, so we can color G with 3 colors. Then we added v_i and v_i' as vertices for every x_i to x_n and form triangles with every (v_i,v_i') pairs between B as shown above.
(ex: for x_1: v_1 and v_1' … for x_n: v_n and v_n')

6. In Fig2. assignment of colors to vertices v_i are representing truth assignments to them, in other words, for instance, if we assign color T to the v_i, we will assign F to the v_i' and it will be 3-coloured as T,F,B for vertices v_i, v_i' and B respectively.

7. After truth assignment, we also need to add constraints (by edges and extra vertices). To adding constraints to G, we will get the satisfiability of the clauses (C_1,…,C_n) of 3-SAT instance by adding OR-gadget.

8. For a clause C_i = (a v b v c), we will connect the literals (a,b,c) of the clause and express them by our colors {T,F,B} as can be seen in Fig.3 below.

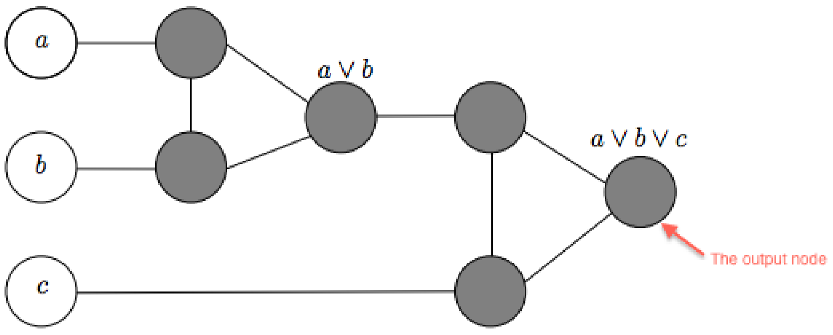


Fig.3. OR-gadget that satisfies the output node

9. We basically want this output node to be coloured T if C_i is satisfied and F otherwise.

10. This is a two-step construction:
    a. The node labeled a ∨ b captures the output of (a ∨ b)
    b. Repeat the same operation for ((a ∨ b) ∨ c).

11. If we examine a, b, c and make some assignments to them, we can notice that

a. If a, b, c are all coloured as F in a 3-colouring, then the output node of the OR-gate has to be coloured as F. (F ∨ F ∨ F = F). Therefore, it brings the unsatisfiability of the clause C_i = (a ∨ b ∨ c).

b. If one of a, b, c is coloured T, <u>then there exists</u> a valid 3-colouring of the OR-gadget where the output node is colored T. Therefore, again it brings the satisfiability of the clause.

12. By adding OR-gadget of C_i in instance, we connect the output node of every gadget to every vertex we will get the following:

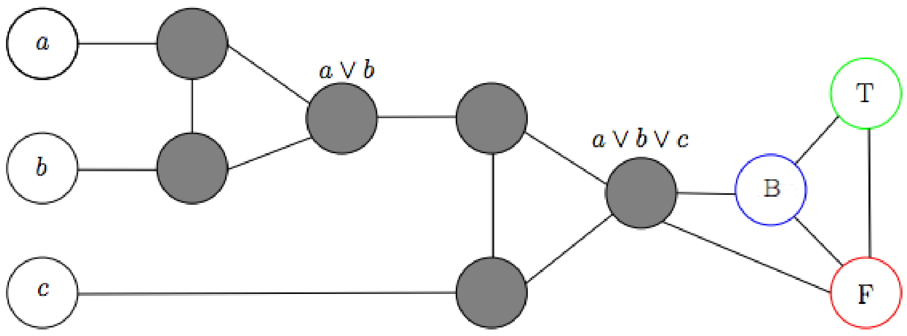


Fig.4. OR-gadgets and output node connected to G

13. In Fig4. we proved that our initial 3-SAT instance is satisfiable if and only the graph G constructed as above is 3-colourable.

14. Now we will suppose 3-SAT instance is satisfiable and let $(x_1^*, x_2^*, \ldots, x_n^*)$ be the satisfying assignment.

15. If x^*_i is assigned as True, we colour v_i with T and v_i' with F, that is connected to the Base vertex as colored as B so that it is a valid coloring.

16. Since 3-SAT instance is satisfiable, every clause $C_i = (a \vee b \vee c)$ must be satisfiable, i.e. at least of one of a, b, c is set to True.

17. Second property of the OR-gadget, we know that the gadget corresponding to C_i can be 3-coloured so that the output node is coloured as T. And because the output node is adjacent to the False and Base vertices of the initial triangle only, this is a proper 3-colouring.

18. Conversely, suppose G is 3-colourable. We construct an assignment of the literals of 3-SAT instance by setting x_i to True if v_i is coloured T and vice versa. Now suppose this assignment is not a satisfying assignment to 3-SAT instance, then this means there exists at least one clause $C_i = (a \vee b \vee c)$ that was not satisfiable. That is, all of a, b, c were set to False. But if this is the case, then the output node of corresponding OR-gadget of C_i must be coloured F (by

property (a)). But this output node is adjacent to the False vertex coloured F; thus, contradicting the 3-colourability of G!

To conclude, we shown that 3-COLOURING is in NP and that it is NP-hard by giving a reduction from 3-SAT. Therefore 3-COLOURING is NP-complete.

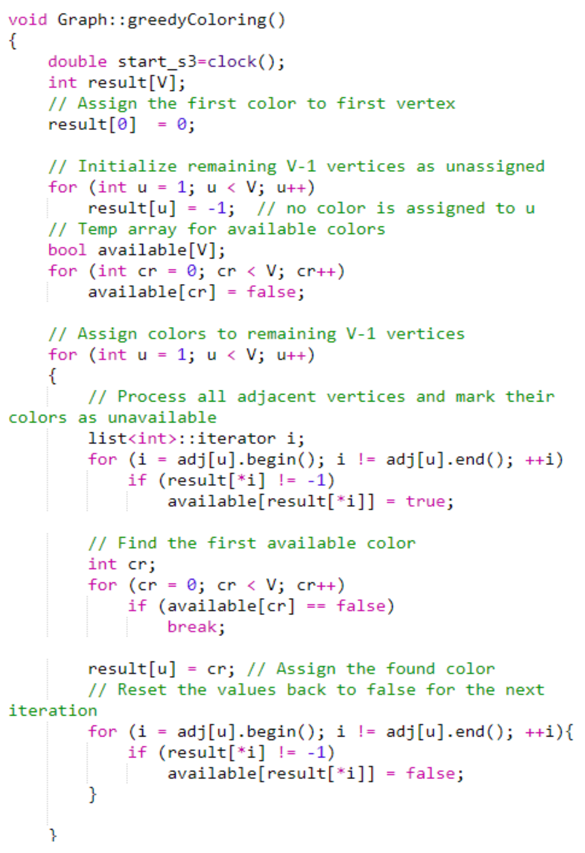
ALGORITHM DESCRIPTION

There is not an exact algorithm that solves 3-COL problem in polynomial time but there are some heuristics such as greedy coloring algorithm and a more efficient one 3-coloring in time $O(1.3446^n)$ algorithm by Richard Beigel and David Eppstein.

In this project, we will cover and analyze the performance of the greedy approach to 3-COL problem.

The algorithm basically does the following;

1. Initially, it colours the first vertex with the first color.
2. Then it checks the next vertex and color it with the next color which isn't used before on the vertices that are adjacent to current vertex.
3. If all colors which are used before is seen on adjacent vertices of the current vertex, it creates a new color and assign it.
4. Algorithms continues in this manner until there is no vertex that are not coloured.

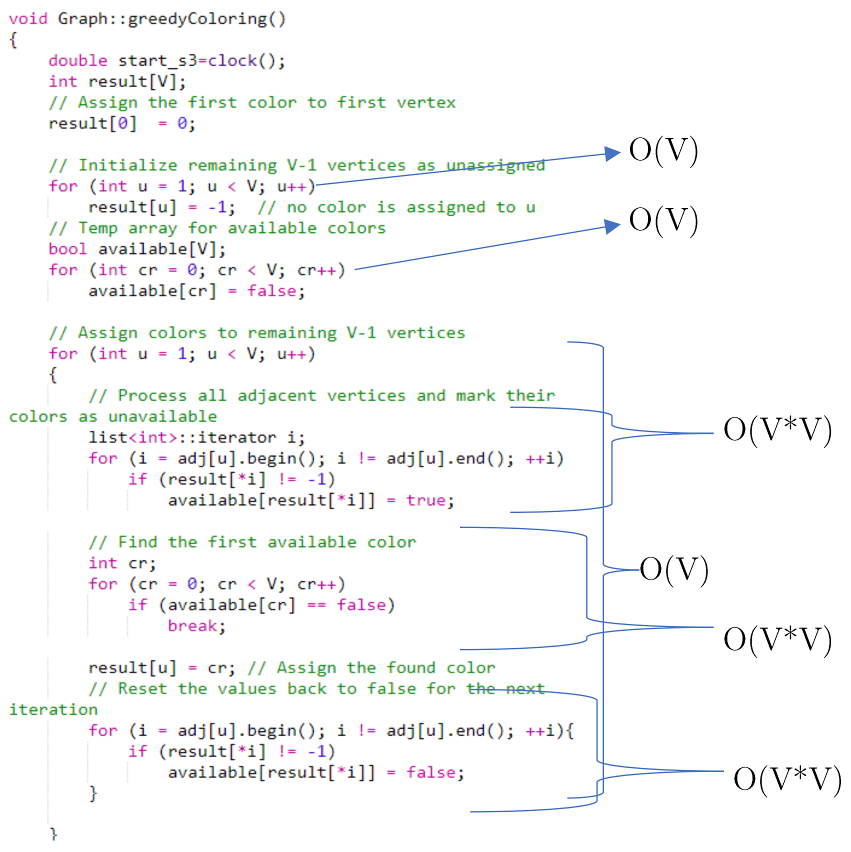


```
void Graph::greedyColoring()
{
    double start_s3=clock();
    int result[V];
    // Assign the first color to first vertex
    result[0]  = 0;

    // Initialize remaining V-1 vertices as unassigned
    for (int u = 1; u < V; u++)
        result[u] = -1;  // no color is assigned to u
    // Temp array for available colors
    bool available[V];
    for (int cr = 0; cr < V; cr++)
        available[cr] = false;

    // Assign colors to remaining V-1 vertices
    for (int u = 1; u < V; u++)
    {
        // Process all adjacent vertices and mark their
colors as unavailable
        list<int>::iterator i;
        for (i = adj[u].begin(); i != adj[u].end(); ++i)
            if (result[*i] != -1)
                available[result[*i]] = true;

        // Find the first available color
        int cr;
        for (cr = 0; cr < V; cr++)
            if (available[cr] == false)
                break;

        result[u] = cr; // Assign the found color
        // Reset the values back to false for the next
iteration
        for (i = adj[u].begin(); i != adj[u].end(); ++i){
            if (result[*i] != -1)
                available[result[*i]] = false;
        }

    }
```

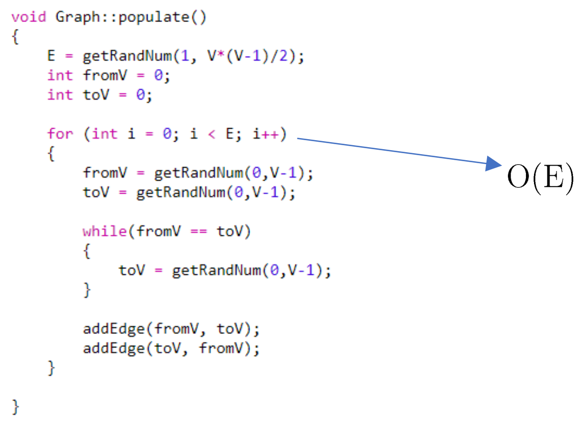
ALGORITHM ANALYSIS

The algorithm we suggest does not guarantee that it will find a colouring pattern with 3 colors, because if it can colour the graph with only use of 2 colors it does not care about the condition that requiring to have exactly 3 colors. Furthermore, if it gets stuck in 3 color, it just increases the number of colors to finish colouring all the vertices. We modified our code as if it encounters a new color more than 3, it is considered as a failure of that algorithm and we analyzed the success rate of our algorithm in this sense in the experimental analysis part of this report.

We can calculate the worst case running time of the algorithm as follows:



```
void Graph::greedyColoring()
{
    double start_s3=clock();
    int result[V];
    // Assign the first color to first vertex
    result[0]  = 0;

    // Initialize remaining V-1 vertices as unassigned
    for (int u = 1; u < V; u++)
        result[u] = -1;  // no color is assigned to u
    // Temp array for available colors
    bool available[V];
    for (int cr = 0; cr < V; cr++)
        available[cr] = false;

    // Assign colors to remaining V-1 vertices
    for (int u = 1; u < V; u++)
    {
        // Process all adjacent vertices and mark their
colors as unavailable
        list<int>::iterator i;
        for (i = adj[u].begin(); i != adj[u].end(); ++i)
            if (result[*i] != -1)
                available[result[*i]] = true;

        // Find the first available color
        int cr;
        for (cr = 0; cr < V; cr++)
            if (available[cr] == false)
                break;

        result[u] = cr; // Assign the found color
        // Reset the values back to false for the next
iteration
        for (i = adj[u].begin(); i != adj[u].end(); ++i){
            if (result[*i] != -1)
                available[result[*i]] = false;
        }

    }
```



```
void Graph::populate()
{
    E = getRandNum(1, V*(V-1)/2);
    int fromV = 0;
    int toV = 0;

    for (int i = 0; i < E; i++)
    {
        fromV = getRandNum(0,V-1);
        toV = getRandNum(0,V-1);

        while(fromV == toV)
        {
            toV = getRandNum(0,V-1);
        }

        addEdge(fromV, toV);
        addEdge(toV, fromV);
    }

}
```

Running Time: $O(V) + O(V) + O(V^2) + O(V^2) + O(V^2) + O(E) = O(V^2{+}E)$

EXPERIMENTAL ANALYSIS

For experimental analysis, we randomly generated several number of graphs that has 5,6,7,8,9,10,15,20 vertices as follows.

Success Rates

Success Rates are determined as if our algorithm uses more than 3 color for graph-colouring, it is considered as a failure, if it uses exactly 3, it succeeds.

Number of Graphs Generated	5 Vertices	6 Vertices	7 Vertices	8 Vertices	9 Vertices	10 Vertices	15 Vertices	20 Vertices
100	0.478	0.48	0.40	0.36	0.33	0.24	0.13	0.08
1000	0.481	0.483	0.43	0.349	0.298	0.237	0.144	0.087
3000	0.4773	0.4893	0.427	0.375	0.304	0.2596	0.137	0.0923
5000	0.4694	0.488	0.4308	0.3688	0.3148	0.253	0.127	0.0966
10000	0.473	0.4901	0.4321	0.3559	0.3007	0.2562	0.1398	0.0997

Fig.5. Success rates table of randomly generated graphs with different vertices

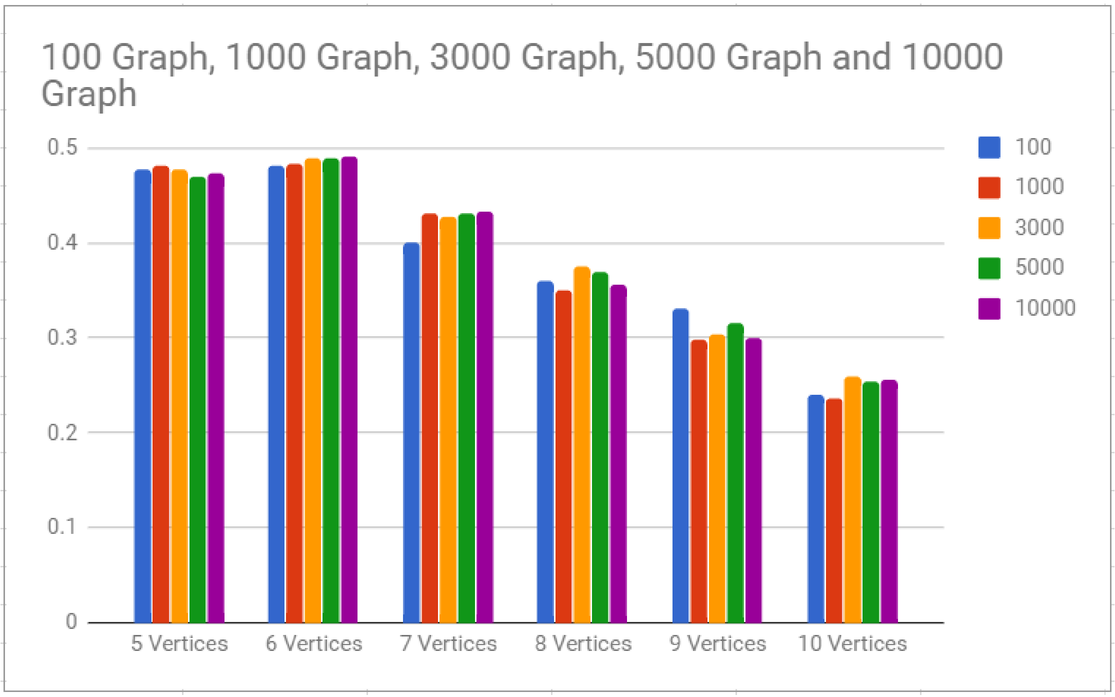


Fig.6. Success rate graph of randomly generated graphs

Number of Vertices	Success Rate
5	0.473
6	0.4901
7	0.4321
8	0.3559
9	0.3007
10	0.2562
15	0.1398
20	0.0997

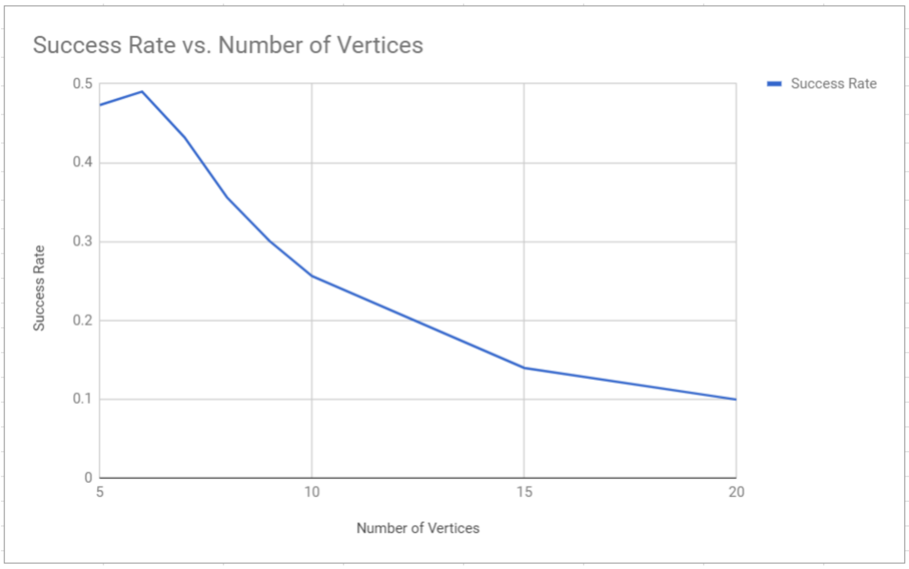


Fig.5. & Fig.6 Success rate table and graph according to number of vertices

It can be deduced that while the number of vertices are increasing, chance of finding solution to the 3-COL problem of our greedy algorithm is decreasing.

Running Time Experimental Measurement

In order to analyze running time of our algorithm experimentally, we created some functions to help us to interpret some statistic concepts such as standard deviation, standard error, sample mean and confidence level intervals.

```
float calculateSD(vector<float> & data)
{
    float sum = 0.0, mean, standardDeviation = 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++)
        standardDeviation += pow(data[j] - mean, 2);
    standardDeviation = sqrt(standardDeviation / data.size
());
    return standardDeviation;
}
```

$$s = \sqrt{\frac{\sum(x - \bar{x})^2}{n - 1}}$$

```
float calculateStandartError(float standartDeviation,int N)
{
  return standartDeviation/sqrt(N);
}
```

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

```
void getRunningTime(){
  float totalTime = 0.0;
  for (int i = 0; i < runningTimes.size(); i++){
    totalTime += runningTimes[i];
  }

  float standartDeviation = calculateSD(runningTimes);
  int N = runningTimes.size();
  float m = totalTime/N; // Sample Mean

  const float tval90 = 1.645; // t-value for %90 Confidence
Level
  const float tval95 = 1.96; // t-value for %95 Confidence
Level

  float sm = calculateStandartError(standartDeviation,N);

  float upperMean90 = m+tval90*sm;
  float lowerMean90 = m-tval90*sm;

  float upperMean95 = m+tval95*sm;
  float lowerMean95 = m-tval95*sm;

  cout << "Mean Time: " << m << " ms";

  cout << ", SD: " << standartDeviation << " Standart Error:
" << sm << ", " << ", %90 " << upperMean90 << " - " <<
lowerMean90 << ", %95 " << upperMean95 << " - " << lowerMean9
5;
    runningTimes.clear();
}
```

As can be understood from the code on the left, this function is used for calculating true mean in 90% and 95% confidence level intervals (t-values are used as fixed values for that confidence levels because after 100 samples they are so close to each other, and our samples are all higher than 100)

100 RUN PER INPUT SIZE

Size	Mean Time(s)	Standard Deviation	Standard Error	%90-CL	%95-CL
5	0.00424	0.00232861	0.000232861	0.00462305- 0.00385694	0.00469641- 0.00378359
10	0.00774	0.00437177	0.000437177	0.00845916- 0.00702085	0.00859687- 0.00688313
15	0.01295	0.00669533	0.000669533	0.0140514- 0.0118486	0.0142623- 0.0116377
20	0.02383	0.0165747	0.00165747	0.0265565- 0.0211035	0.0270786- 0.0205814
25	0.03359	0.0199445	0.00199445	0.0368709- 0.0303091	0.0374991- 0.0296809

Fig.7. 100 Run for 5,10,15,20,25 vertices and Mean times, Standard Deviation, Standard Error, Confidence Levels

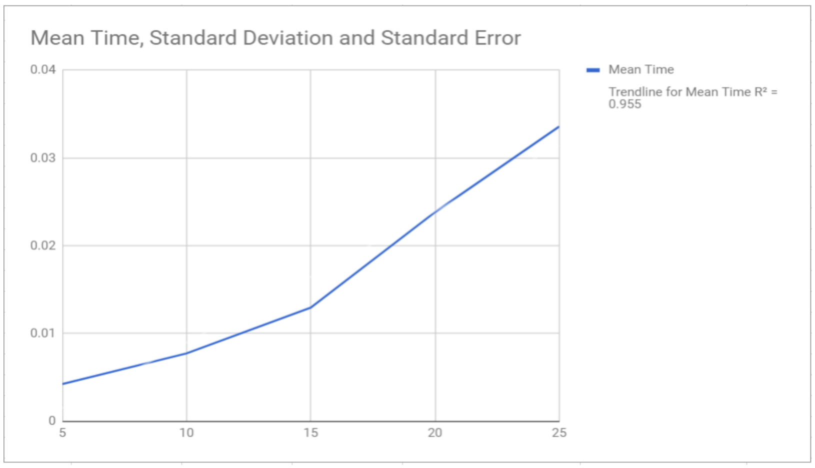


Fig.8. Running time Graph of 100 run according to input size (number of vertices)

1000 RUN PER INPUT SIZE

Size	Mean Time(s)	Standard Deviation	Standard Error	%90-CL	%95-CL
5	0.002614	0.00156812	4.96E-05	0.00269558- 0.00253243	0.0027112- 0.00251681
10	0.00626399	0.00287234	9.08E-05	0.00641341- 0.00611458	0.00644202- 0.00608597
15	0.011974	0.00590841	0.00018684	0.0122813- 0.0116666	0.0123402- 0.0116078
20	0.02023	0.011063	0.000349844	0.0208055- 0.0196545	0.0209157- 0.0195443
25	0.031714	0.0188678	0.000596651	0.0326955- 0.0307325	0.0328834- 0.0305445

Fig.9. 1000 Run Table for 5,10,15,20,25 vertices and Mean times, Standard Deviation, Standard Error, Confidence Levels

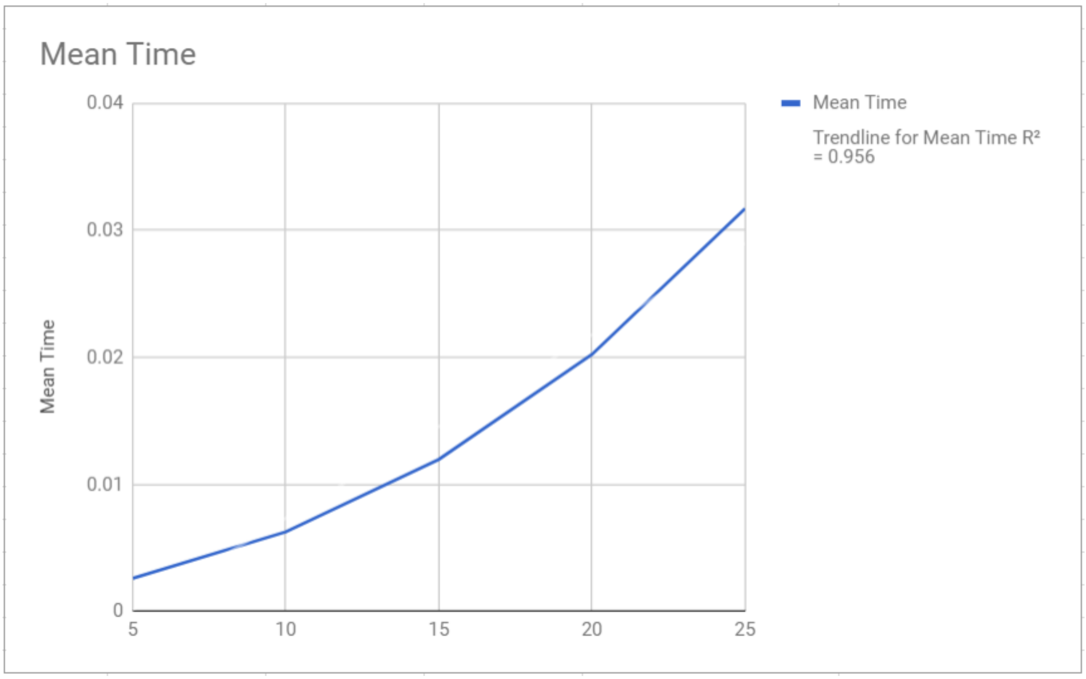


Fig.10. Running time Graph of 1000 run according to input size (number of vertices)

3000 RUN PER INPUT SIZE

Size	Mean Time(s)	Standard Deviation	Standard Error	%90-CL	%95-CL
5	0.00288363	0.00166417	3.04E-05	0.00293361- 0.00283365	0.00294318- 0.00282408
10	0.00697033	0.00352543	6.44E-05	0.00707621- 0.00686445	0.00709648- 0.00684417
15	0.0136743	0.00820747	0.000149847	0.0139208- 0.0134278	0.013968- 0.0133806
20	0.02177	0.0137025	0.000250172	0.0221815- 0.0213584	0.0222603- 0.0212796
25	0.036593	0.0252588	0.00046116	0.0373516- 0.0358344	0.0374969- 0.0356891

Fig.11. 3000 Run Table for 5,10,15,20,25 vertices and Mean times, Standard Deviation, Standard Error, Confidence

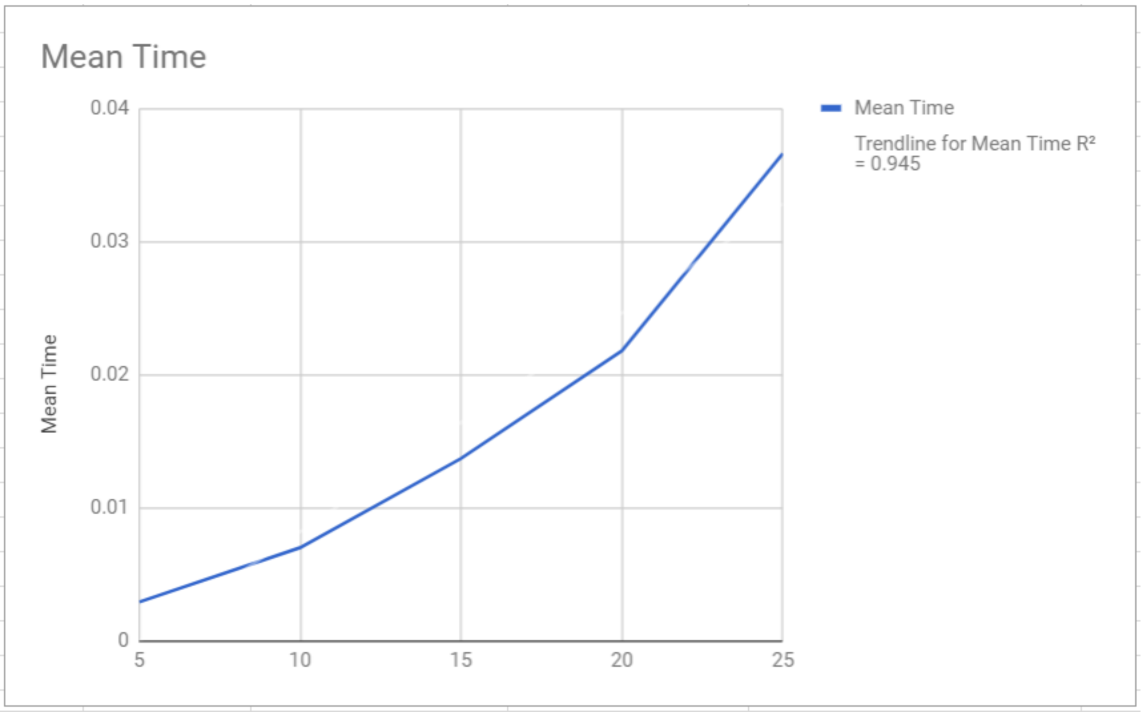


Fig.10. Running time Graph of 3000 run according to input size (number of vertices)

5000 RUN PER INPUT SIZE

Size	Mean Time(s)	Standard Deviation	Standard Error	%90-CL	%95-CL
5	0.00261599	0.00133301	1.89E-05	0.002647 - 0.00258498	0.00265294 - 0.00257904
10	0.00621641	0.00265368	3.75E-05	0.00627814 - 0.00615467	0.00628996 - 0.00614285
15	0.0126694	0.00687041	9.72E-05	0.0128292 - 0.0125095	0.0128598 - 0.0124789
20	0.0233198	0.0145339	0.000205541	0.0236579 - 0.0229817	0.0237227 - 0.022917
25	0.0353543	0.0237548	0.000335944	0.0359069 - 0.0348016	0.0360127 - 0.0346958

Fig.12. 3000 Run Table for 5,10,15,20,25 vertices and Mean times, Standard Deviation, Standard Error, Confidence Levels

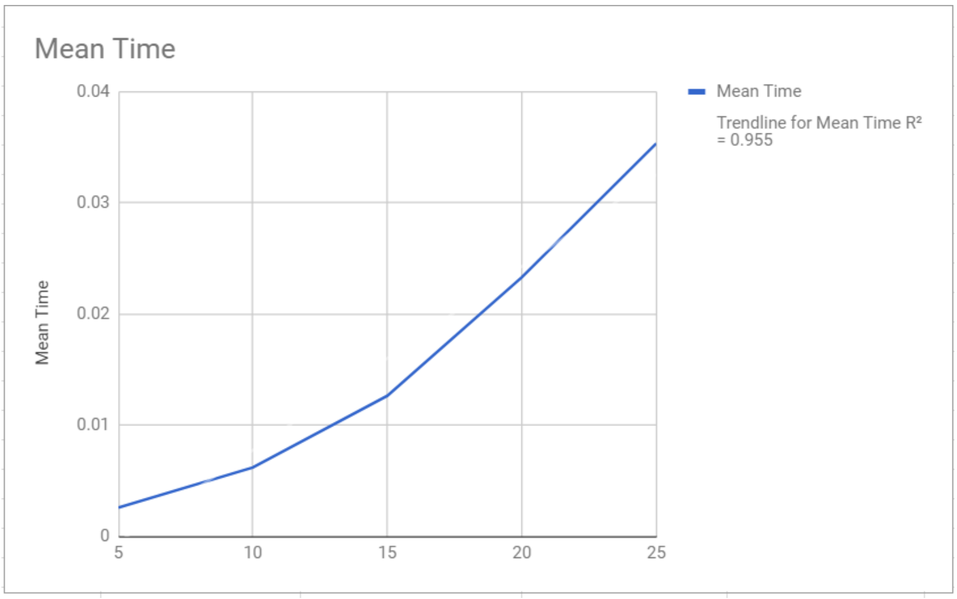


Fig.13. Running time Graph of 5000 run according to input size (number of vertices)

TESTING

For testing our algorithm with different inputs we used black-box testing technique, as we mentioned before, we randomly created graphs for different number of vertices by a Graph class member function called populate() as can be seen below:

```
void Graph::populate()
{
    E = getRandNum(1, V*(V-1)/2);
    int fromV = 0;
    int toV = 0;

    for (int i = 0; i < E; i++)
    {
        fromV = getRandNum(0,V-1);
        toV = getRandNum(0,V-1);

        while(fromV == toV)
        {
            toV = getRandNum(0,V-1);
        }

        addEdge(fromV, toV);
        addEdge(toV, fromV);
    }

}
```

Some Test Result Graphs

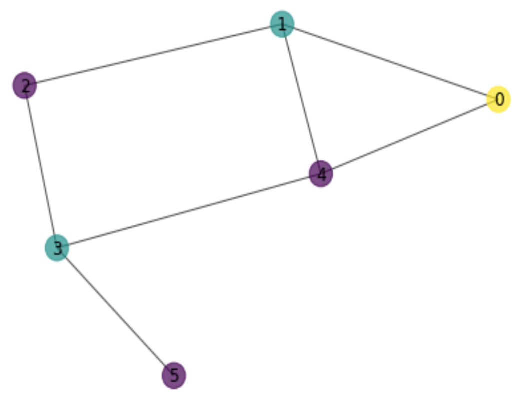


Fig.14. Graph with Vertices: {0,1,2,3,4,5}
Edges: (0,1),(0,4),(1,4),(1,2),(2,3),(3,4),(3,5)

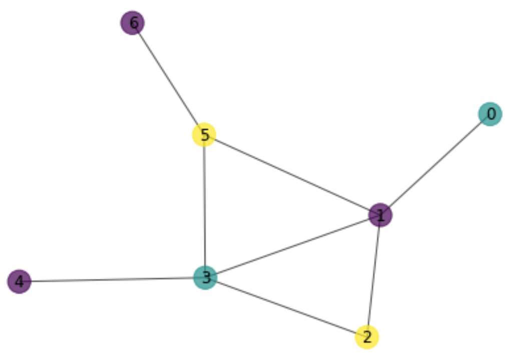


Fig.15. Graph with Vertices: {0,1,2,3,4,5,6}
Edges: (0,1),(1,2),(1,3),(1,5),(2,3),(3,5),(3,4),(5,6)

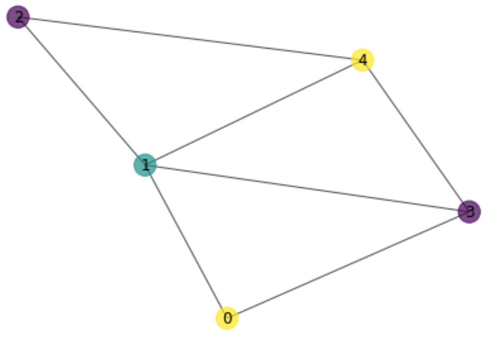


Fig.16. Graph with Vertices: {0,1,2,3,4,5}
Edges: (0,3),(0,1),(1,3),(1,2),(1,4),(3,4),(2,4)

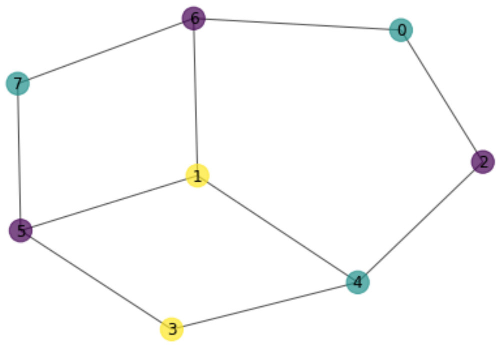


Fig.17. Graph with Vertices: {0,1,2,3,4,5,6,7}
Edges: (0,6),(0,2),(6,1),(2,4),(4,1),(4,3),(3,5),(1,5), (5,7), (6,7)

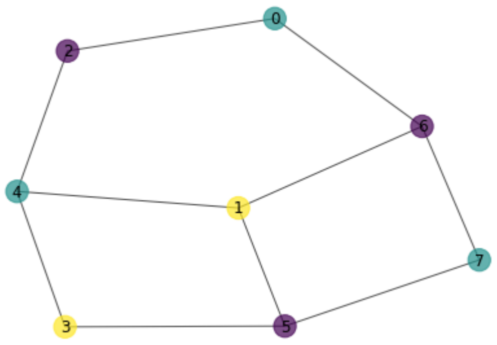


Fig.18. Graph with Vertices: {0,1,2,3,4,5,6,7}
Edges: (0,6),(0,2),(6,1),(2,4),(4,1),(4,3),(3,5),(1,5), (5,7), (6,7)

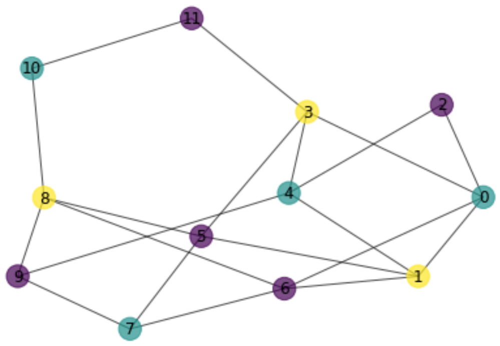


Fig.19. Graph with Vertices: {0,1,2,3,4,5,6,7}
Edges: (0,1),(0,3),(3,4),(3,11),(10,8),(9,4),(11,10),(7,9), (1,5),(6,7),(5,8),(6,8),(9,8),(3,5)

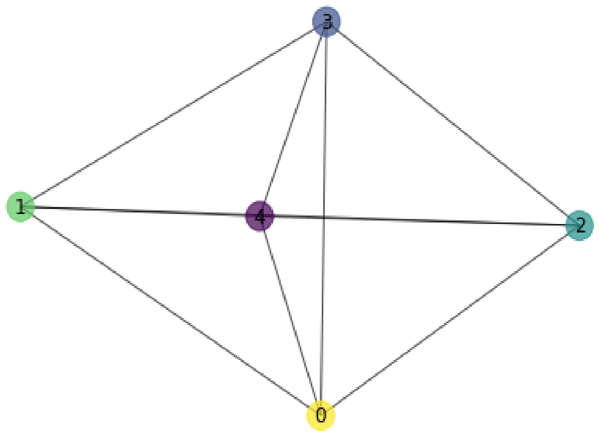


Fig.20. Graph with Vertices: {0,1,2,3,4} **FAILED**
Edges: (0,1),(0,2),(0,3),(0,4),(1,2),(1,3),(1,4),(2,3), (2,4), (3,4)

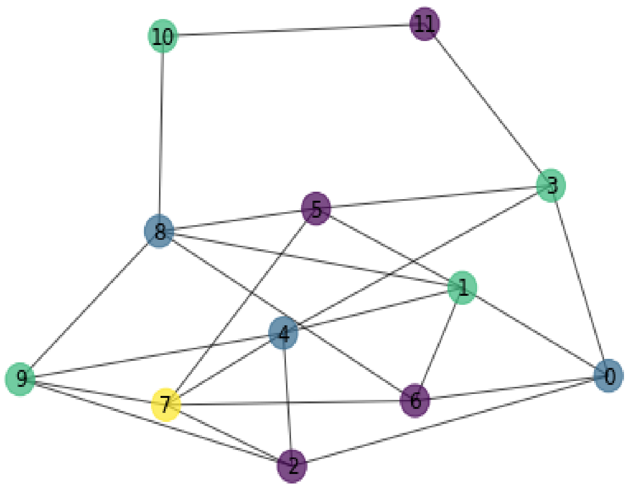


Fig.21. Graph with Vertices: {0,1,2,3,4,5,6,7,8,9,10,11} **FAILED**
Edges: (0,1),(0,3),(3,4),(3,11),(2,9),(2,7),(4,7),(4,3),(8,1),(10,8),(9,4), (11,10),(7,9),(1,5),(6,7),(5,8),(6,8),(9,8),(3,5)

CONCLUSION

To sum up, we have seen that 3-COLOR problem is a NP-Complete problem that is reduced from 3-SAT. Also, there is not such an exact efficient algorithm that solves that problem in polynomial time. There are just some approximate algorithms such as greedy-coloring. We examined and analyzed this algorithm in this report and have seen that this algorithm does not always find 3-color solution.

After experimental analysis of this algorithm, we deduced that it has 47% chance to find a 3-COLOR solution for a graph with 5 vertices, ignoring of some outliers, we can also say that chance of finding a 3-COLOR solution is decreasing inversely proportional to number of vertices of graph (it decreases to 43%, 35%, 30% and goes on) because greedy approach algorithm takes the easy way out, increases the number of colors when it gets stuck in 3 colors, thus causes failure. We think that, this algorithm can be improved to get higher chance of getting solutions. This can be done by focusing on solutions of algorithm with 2 colors, they can be forced to increase their number of colors.

If we consider the measurement of running time of the algorithm, the graph we get in the experimental analysis part shows a consistent trend (a quadratic type) with our result obtained from running time calculation in algorithm analysis part.