| CS301  2023-2024 Spring |
| --- |

Project Final Report

Group 187

İbrahim Mahir Akbaş 29492

Ufuk Ulaş Tokat 28914

[**1. Problem Description 2**](#_heading=h.gjdgxs)

[1.1 Overview 2](#_heading=h.30j0zll)

[1.2 Decision Problem 2](#_heading=h.3znysh7)

[1.3 Optimization Problem 3](#_heading=h.2et92p0)

[1.4 Example Illustration 3](#_heading=h.tyjcwt)

[1.5 Real World Applications 4](#_heading=h.3dy6vkm)

[1.6 Hardness of the Problem 6](#_heading=h.3rdcrjn)

[**2. Algorithm Description 7**](#_heading=h.1ksv4uv)

[2.1 Brute Force Algorithm 7](#_heading=h.44sinio)

[2.1.1 Overview 7](#_heading=h.2jxsxqh)

[2.1.2 Pseudocode 8](#_heading=h.z337ya)

[2.2 Heuristic Algorithm 9](#_heading=h.3j2qqm3)

[2.2.1 Overview 9](#_heading=h.1y810tw)

[2.2.2 Pseudocode](#_heading=h.4i7ojhp) 10

[**3. Algorithm Analysis 12**](#_heading=h.2xcytpi)

[3.1 Brute Force Algorithm 12](#_heading=h.1ci93xb)

[3.1.1 Correctness Analysis 12](#_heading=h.3whwml4)

[3.1.2 Time Complexity 13](#_heading=h.2bn6wsx)

[3.1.3 Space Complexity 14](#_heading=h.qsh70q)

[3.2 Heuristic Algorithm 15](#_heading=h.3as4poj)

[3.2.1 Correctness Analysis 15](#_heading=h.1pxezwc)

[3.2.2 Time Complexity 17](#_heading=h.49x2ik5)

[3.2.3 Space Complexity 17](#_heading=h.2p2csry)

[**4. Sample Generation (Random Instance Generator) 18**](#_heading=h.147n2zr)

[**5. Algorithm Implementation 19**](#_heading=h.3o7alnk)

[5.1 Brute Force Algorithm 19](#_heading=h.23ckvvd)

[5.1.1 Initial Testing of the Algorithm 21](#_heading=h.ihv636)

[5.2 Heuristic Algorithm 22](#_heading=h.32hioqz)

[**6. Experimental Analysis of the Performance (Performance Testing) 26**](#_heading=h.1hmsyys)

[**7. Experimental Analysis of the Quality 28**](#_heading=h.41mghml)

[**8. Experimental Analysis of the Correctness (Functional Testing) 32**](#_heading=h.vx1227)

[**9. Discussion**](#_heading=h.3l18frh) **39**

[**References 40**](#_heading=h.206ipza)

# **Problem Description**

| *Name:* | *Graph Coloring* |
| --- | --- |
| *Input:* | *An undirected graph with nodes and node set , and edge set ; a positive integer where* |
| *Question:* | *Does G have a proper vertex coloring with k number of colors?* |

## **Overview**

The Graph Coloring problem is about choosing proper colors for all nodes of the graph such that there are no adjacent nodes that have the same color. In other perspectives any subset of E must not have the same color on both sides. Formally, given G (V, E) is an undirected graph and k is the number of colors available. The function colors the G is f: V-> {1, 2, …, k} assigns a color to each vertex such that f(u) ≠ f(v) if u, v ∈ V and e = {u, v} ∈ E.

## **1.2 Decision Problem**

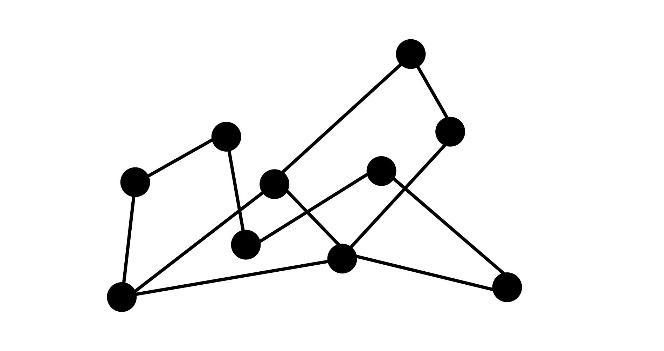
Given an undirected graph and a positive integer k, does there exist a proper coloring with k colors such that two adjacent vertices are not colored with the same color.

## **1.3 Optimization Problem**

In the context of an undirected graph minimize the number of colors k that satisfy f(u) ≠ f(v) if u, v ∈ V and e = {u, v} ∈ E.

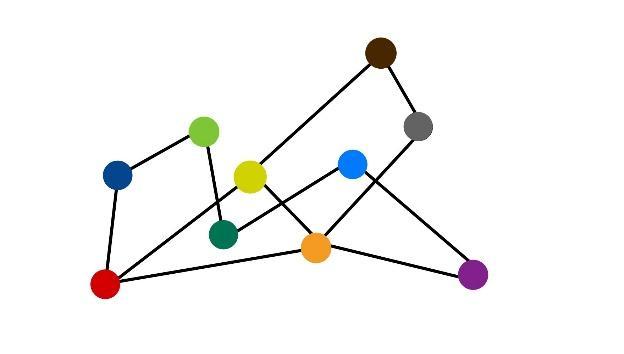
## **1.4 Example Illustration**

Figure 1 illustrates an undirected graph with 10 nodes and 13 edges.



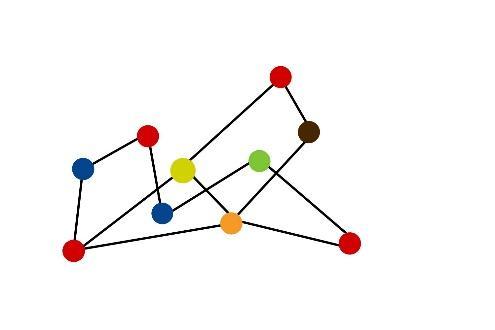
*Figure 1: Illustration of an undirected graph*

If the task is to find the minimum number of colors k that properly covers the graph, the trivial way is coloring all nodes with different colors demonstrated in Figure 2. This way of solving the problem covers all the graphs properly with 10 colors (k = n). It is the naive approach that does not give the best solution for the question.



*Figure 2: Trivial approach for graph coloring*

Figure 3 shows a better solution for the question with 6 colors. Also figure 4 shows an even better solution than figure 3 with only 3 colors. There are other colorings with 3 colors that properly cover all the graph. 3 is the best solution for this graph.

 → çizgi, daire içeren bir resim

Açıklama otomatik olarak oluşturuldu

*Figure 3: Illustration of graph coloring Figure 4: Illustration of graph coloring*

*with 6 colors with 3 colors*

## **1.5 Real World Applications**

The graph coloring problem is frequently used in various real-world applications, helping problem-solving across multiple domains.

### **1.5.1 Event Scheduling**

In schedule planning the graph coloring problem can be seen. Nodes can be designed to show events and an edge can show conflict between events. Finding the minimum number of colors needed to cover all graphs properly gives us the minimum number of places to make these events.

### **1.5.2 Team Building**

If we are planning to form different teams with a group of people. Some of these people are friends with each other. If we want to form groups in such a way that people will be grouped with the people, they don’t know. We can express people as nodes and edges as friendships. When we colored the graph properly and put people with the same color into the same groups our condition will be satisfied.

### **1.5.3 Wireless Communication**

Graph coloring is used for channel assignment in wireless communication. Devices that will communicate are the vertices and interference with each other are edges. Graph coloring can be used to assign frequencies in a way that a minimum number of frequencies are used and the interference between wireless networks is avoided.

### **1.5.4 Additional Applications**

Graph coloring is also used for design optimum register allocations while designing a compiler. Another example is solving sudoku. Sudoku can be represented as a graph, and we can assign digits as they are colors. Also, we can use graph coloring to create informative maps.

## **1.6 Hardness of the Problem**

The exploration of graph coloring as an algorithmic challenge dates back to the early 1970s, with the chromatic number problem being among the 21 NP-complete problems identified by Karp in 1972[[1]](#footnote-0). According to many papers[[2]](#footnote-1), graph coloring presents computational challenges. Determining whether a given graph can be colored with a specified number of colors (except for very small values) is an NP-complete problem. Specifically, computing the chromatic number, which is the minimum number of colors required to color the vertices of a graph, is NP-hard. Even for 4-regular planar graphs, solving the 3-coloring problem remains NP-complete.

# **Algorithm Description**

## 2.1 Brute Force Algorithm

### 2.1.1 Overview

The algorithm below is a greedy/brute-force approach to the problem. It works in exponential complexity. It moves through all possible colorings of the graph. For each coloring it checks if it is a proper coloring and the number of colors used for coloring is less than given number k. It is a direct approach that gives the solution but is very inefficient. If there exists a proper solution it gives it as a result at the end. If it cannot find a proper solution, which has a number of colors less than or equal to k, it gives null as an output.

### 2.1.2 Pseudocode

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

Step 1: Initialize the vertices.

Step 2: For current\_coloring in all\_collorings

Step 2.1: Assume current\_combination is valid coloring

Step 2.2: Initialize is\_valid to True.

Step 2.3: For current\_edge(u,v) in all\_edges

Step 2.3.1: If color u equal to color v

Step 2.3.2: Set is\_valid to false.

Step 2.4: If is\_colloring is True and number of colors is smaller than k

Step 2.5: Return current\_colloring

Step 3: Return None

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

## 2.2 Heuristic Algorithm

## 2.2.1 Overview

The algorithm presented is a heuristic approach to determining the minimum number of colors needed to properly color a graph. A heuristic algorithm provides a practical and efficient solution, albeit not necessarily optimal or exact. This algorithm, known as Welsh-Powell, employs a straightforward yet effective heuristic strategy.

Input: The algorithm takes an undirected graph G = (V, E) as input, where V is the set of vertices and E is the set of edges.

1. Initialization:
   * The algorithm starts by sorting the vertices of the graph in descending order based on their degrees (number of incident edges).
   * A list of colors is initialized.
2. Coloring Process:
   * The algorithm assigns colors to vertices based on a greedy strategy:
     + Starting with the vertex of highest degree, each vertex is assigned the smallest possible color not already assigned to any adjacent vertex.
     + If all adjacent colors are taken, a new color is assigned.
     + This process continues until all vertices are colored.
3. Output:
   * The maximum color used represents the minimum number of colors needed to properly color the graph.

Explanation:

* The algorithm aims to minimize the number of colors needed while ensuring that adjacent vertices are not assigned the same color. By prioritizing vertices with higher degrees, it reduces the likelihood of conflicts and efficiently assigns colors to vertices.
* While the solution provided by the algorithm may not always be optimal, it offers a practical and efficient approach for coloring graphs.

### 2.2.2 Pseudocode

Input: A Graph G = (V, E)

Output: A proper coloring of the Graph

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

Step 1: Sort vertices by degree in descending order

Step 2: Initialize colors and assign the first vertex color 1

Step 3: For each vertex in the sorted list starting from the second vertex:

Step 3.1: Get the colors of neighboring vertices of the current vertex

Step 3.2: Determine available colors by subtracting neighboring colors from all possible colors

Step 3.3: If available colors exist:

Step 3.1.1: Assign the minimum available color to the current vertex

Step 3.4: If no available colors exist:

Step 3.4.1: Assign a new color to the current vertex, one greater than the maximum color used so far.

Step 4: Return the dictionary containing proper coloring

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

The algorithm described below is called the Welsh Powell algorithm. It is a heuristic algorithm that follows a greedy approach to solve the problem.

It first starts by sorting the vertex in decreasing order with respect to their number of connections. Then it chooses a color for the vertex with highest number connections. Then gives the same color to the other vertices which are not connected to that vertex. Then does the same thing with the remaining uncolored vertices.

However, this algorithm does not guarantee that the resulting coloring of the graph will have a minimum number of colors. It gives guarantee that coloring will be true.

# **Algorithm Analysis**

## 3.1 Brute Force Algorithm

### 3.1.1 Correctness Analysis

***Claim****:* Given the graph G = (E,V) and an integer k, Graph coloring algorithm finds the minimum number of colors (at most k) needed to color the graph properly such that f(u) ≠ f(v) if u, v ∈ V and e = {u, v} ∈ E.

***Proof (by contradiction):*** To prove the algorithm first we need to prove that it always finds a proper coloring. Then we need to prove that it finds the proper coloring with a minimum number of colors. Let G(V, E) an undirected graph that has n nodes.

**(1) The algorithm always finds a proper coloring:**

At each iteration the algorithm will try a coloring. If it is a proper one algorithm will stop and return the coloring. If it is not, the algorithm will continue to iterate. Even if none of these iterations satisfy the conditions of proper coloring there will be one case where k = n. In this iteration all of the nodes will be colored differently. So, this iteration will always satisfy the proper coloring criteria.

**(2) The algorithm finds the coloring with the minimum number of colors.**

Suppose we have our algorithm returns a coloring with more colors than the actual minimum number of colors. Let’s call the number of colors used in this coloring K. Actual minimum number of colors is K’. So, we know that K is bigger than K’. Algorithm goes through possible colorings in an increasing order of number of colors. Therefore it would stop at K’ and return it as result. This contradicts with the first assumption. This means our algorithm always finds the coloring with the minimum number of colors.

With this proof we can also say that the algorithm will stop at k when the K’ is bigger than k. So our algorithm either finds the minimum number of coloring that is less than or equal to k or returns null.

### 3.1.2 Time Complexity

Let’s consider an undirected graph G(V, E) that has n nodes and m edges. Maximum number of colors we want is k. Then the worst-case time complexity of the algorithm is O(x m). Worst case is the time when there is no proper coloring than coloring all nodes differently. It can happen when all nodes are connected to every other node in the graph. So, Algorithm will try all possible colorings. Outerloop will iterate O() times. Then in the inner loop it will check all edges in the graph which will iterate O(m) times.

### 3.1.3 Space Complexity

The algorithm uses a data structure with O(n) space complexity of storing assigned colors of nodes. It is O(n) in the worst and best cases because it must show colors of each node for all cases. Therefore the worst-case space complexity of the algorithm is O(n). There may be some other spaces used in the program but they are insignificant when we consider O(n).

## 3.2 Heuristic Algorithm

### 3.2.1 Correctness Analysis

**Theorem**: The proposed algorithm correctly constructs a colored graph where each node and their neighbors have different colors for an undirected graph G.

**Proof:** To prove the correctness of the algorithm, we need to show two properties:

1. **Property:** The resulting set C is a colored graph as explained above.
2. **Property:** The algorithm terminates and colors all vertices in the graph.

**1. Property:** The Resulting Set C is a Colored Graph

To prove that the resulting set C (where C consists of colored vertices and edges in an undirected graph G) is valid colored graph with no adjacent vertices sharing the same color, we can use the following reasoning:

**Sorting by Degree:** Sort the vertices of G in non-increasing order based on their degrees. Meaning that vertices with more connections (higher degree) are considered first in the process.

**Color Assignment:**

* Iterate through the sorted list of vertices.
* For each vertex v, assign the smallest possible color that is not used by any of its adjacent vertices.
* This ensures that adjacent vertices do not share the same color.

**Correctness:** By using for loops(iteration) assigning colors based on vertex ordering and checking neighboring colors, the algorithm guarantees that no adjacent vertices in the resulting graph C share the same color.

**2. Property:** Algorithm Termination and Coloring of All Vertices

**Finite Execution:** The algorithm runs iteratively coloring the vertices until all vertices are processed.

**Coloring Process:**

* For each vertex, assign a color that is different from its neighbors.
* The algorithm ensures termination because each vertex eventually receives a color during the iteration.

**Termination:**

* Since the number of vertices is finite in G, and the algorithm assigns colors in a systematic way (iterates all the vertices in ordered form), it is guaranteed to terminate after a finite number of steps.

**Completeness:**

* At the end of the algorithm's execution, every vertex in G will have a color assigned to it. The reason behind it is that the algorithm iterates all the vertices and colors all of them.

**Conclusion**

The Welsh-Powell algorithm correctly constructs colored graph where no two adjacent vertices have the same color. It achieves this by systematically assigning colors and neighbor checks while ensuring termination and completeness of the coloring.

### 3.2.2 Time Complexity

The algorithm will pass through all vertices to sort them. This will take O(Vlg(V)) if we use Quicksort with a proper pivot algorithm. Then it will go over all of the vertices in the graph. For all vertices it will look at their neighbors. In the worst case a vertex will have V-1 vertices. So it is O(V^2). Overall complexity of the program will be O(Vlg(V)+V^2). V^2 term will dominate VlgV. Therefore the time complexity of the algorithm is O(V^2).

### 3.2.3 Space Complexity

In the algorithm graph is represented as a dictionary where keys are vertices and values are lists of vertices connected to key vertex. So, it needs O(V + E) space to store graph information.

The algorithm uses no extra space for sorting if quicksort algorithm is used because quicksort algorithm is an in-place sorting algorithm.

The algorithm needs O(V) space for the colors array which keeps track of colors assigned to the vertices.

# **Sample Generation (Random Instance Generator)**

We made a program that will create random instances of a graph given the number of vertices. It will take one integer input as a parameter.

* numNodes: Determine how many nodes(vertices) the graph will have.

First, the algorithm will create an empty instance of a graph. Second, it will add vertices that don’t have any neighbors to the graph. Number of these vertices will be equal to numNodes. Third, for all nodes it will go through all nodes and randomly decide whether to add an edge or not. Then if the algorithm decides to add an edge it will add node u to connection list of v and v to connection list u. If it decides to not add an edge it will continue without doing nothing. Finally it will return the graph.

An example instance is below:

G = {

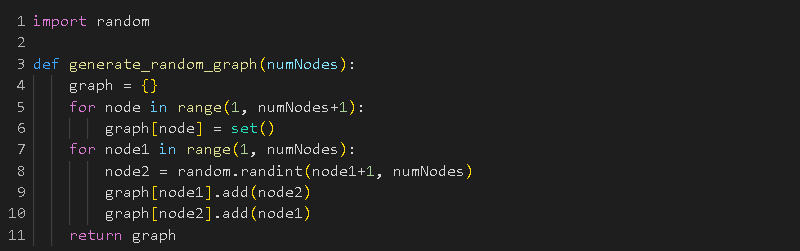
1:[2],

2:[1,3]

3:[2,4]

4:[3]

}



*Figure 8: Implementation of random sample generation*

# **Algorithm Implementation**

## 5.1 Brute Force Algorithm

Function Definition: ***graph\_coloring(graph, n)***

Input parameters to the algorithm are as follows:

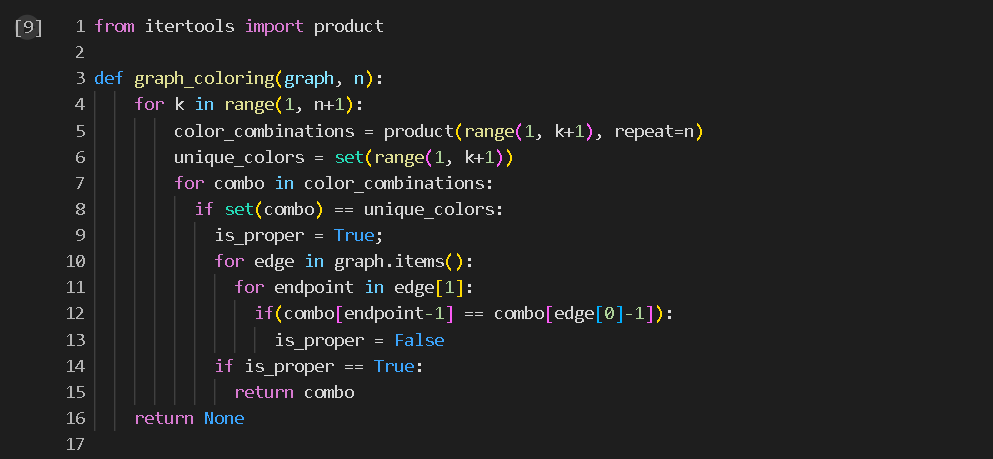
**graph**: A dictionary representing the graph, where keys represent vertices and values represent sets of adjacent vertices.

**size**: An integer representing the number of nodes in the graph.

Coloring Process:

1. *for k in range(1, n+1)*: Firstly, loop iterates over possible numbers of colors from 1 to ‘n’.
2. *color\_combinations = product(range(1, k+1), repeat=n):* Here, product generates all possible combinations of colors from 1 to ‘k’ for ‘n’ nodes.
3. *unique\_colors = set(range(1, k+1)):* This line creates a set of unique colors from 1 to ‘k’.
4. Then our code iterates each color combination and checks if the color combination uses all ‘k’ colors.
5. After that it checks if adjacent nodes have different colors. If any edge violates this condition, is\_proper is set to False.
6. If a proper coloring is found, it returns the color combination.
7. If no proper coloring is found within the given number of colors, it returns ‘None’.
8. It always finds the minimum number of colors necessary.

This brute force algorithm checks all possible ways to color the graph properly. It's good for small graphs or as a basic comparison for fancier methods. But, with bigger graphs, it might take a while because it has to check everything.



*Figure 9: Implementation of brute-force Graph Coloring algorithm*

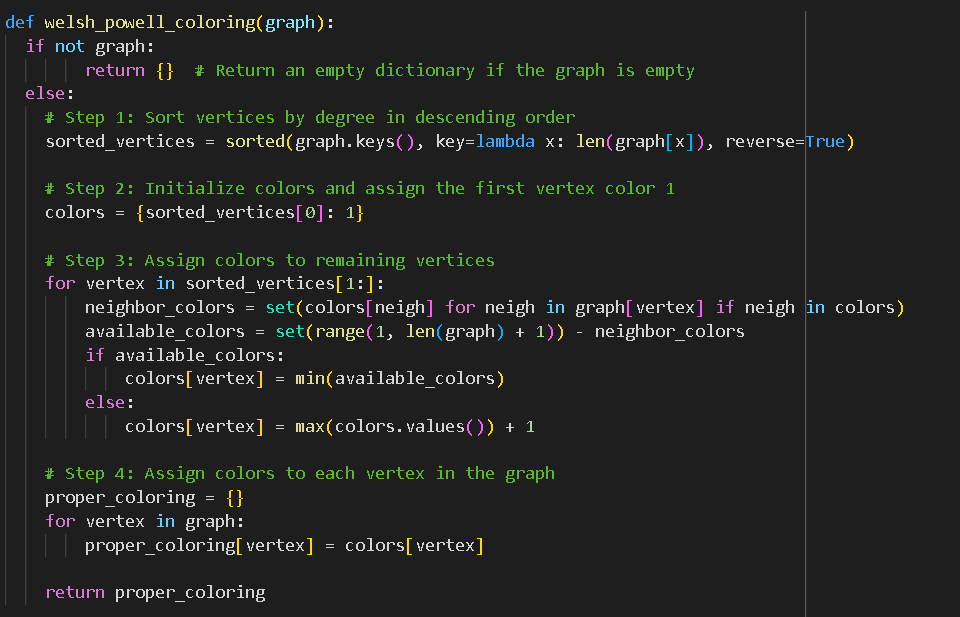
### 5.1.1 Initial Testing of the Algorithm

We successfully tested the code with 16 instances without encountering any problems. However, when we tested it with input sizes of 30 or more, we faced challenges because of the exponential complexities discussed earlier.

The size of the instances tried is as follows:

1. number of nodes in graph = 3
2. number of nodes in graph = 4
3. number of nodes in graph = 6
4. number of nodes in graph = 5
5. number of nodes in graph = 6
6. number of nodes in graph = 8
7. number of nodes in graph = 5
8. number of nodes in graph = 9
9. number of nodes in graph = 9
10. number of nodes in graph = 14
11. number of nodes in graph = 20
12. number of nodes in graph = 25
13. number of nodes in graph = 5
14. number of nodes in graph = 8
15. number of nodes in graph = 11
16. number of nodes in graph = 13

5.2 Heuristic Algorithm



*Figure 10: Implementation of Heuristic Algorithm*

Figure 10 shows an implementation of the welsh powell algorithm. Implementation for the algorithm is described in section 2.3 in detail.

We tested the algorithm with some popular graphs and saw no error in the resulting colorings. They are proper colorings with no defects.

| **Graph** | **|V| (input size)** | **|C| (number of colors used)** |
| --- | --- | --- |
| Tetrahedron | 4 | 4 |
| Kuratowski | 6 | 2 |
| Octahedron | 6 | 3 |
| Wheel graph W8 | 8 | 4 |
| Cube | 8 | 2 |
| Petersen | 10 | 3 |
| Grötzsch | 11 | 4 |
| Herschel | 11 | 2 |
| Icosahedron | 12 | 5 |

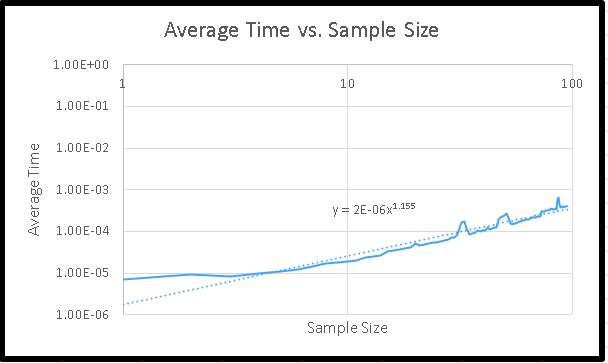
Table 1: Initial testing of heuristic algorithm with popular graphs.

# 

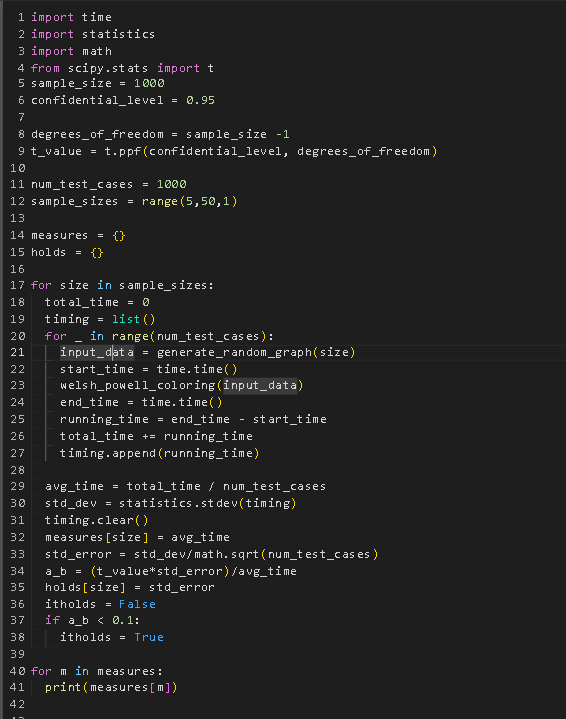
# **6. Experimental Analysis of the Performance (Performance Testing)**

Here we did some experiments on the algorithm to analyze its performance . This analysis was different from the upper bound analyses we did in section 3.2. We did actual tests to see its real performance on time. We created 1000 random instances for each of different graph sizes to represent input sizes properly.We choose graph sizes between 5 and 100. We Then performed the algorithm on each of these graphs while recording time. Then we plotted the resulting graph in log-log format as seen in figure 19. The polynomial fit of the graph is y =4E-06x^(1.155). So in practice the time complexity of our algorithm is O(n^1.155).

We used sample size 1000 and confidence level of 0.95. The mean values of each input sizes are in the interval mean +/- standard deviation \* t where t is for 0.95 confidence level and 1000 experiments. We made a code that will check that if standard deviation \* t will be less than 0.1so the interval will be narrow. Implementation is shown in figure 20.



*Figure 19*



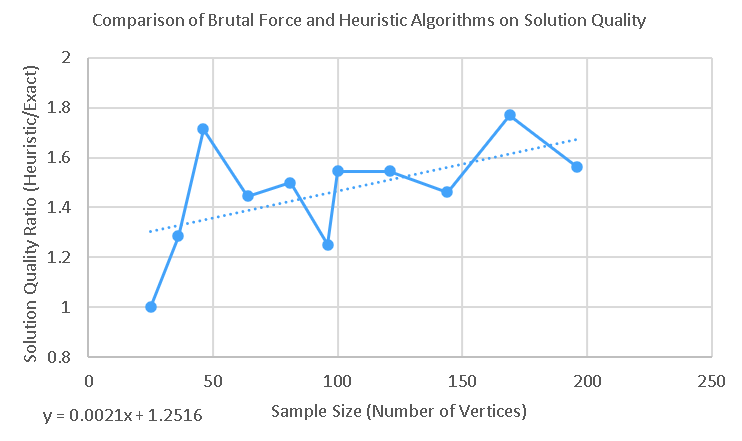
*Figure 20*

# **7. Experimental Analysis of the Quality**

*Table 2*

| Input Size | Exact Algorithm Minimum Number of Colors | Heuristic Algorithm Min Number of Colors | Ratio of Heuristic/Exact |
| --- | --- | --- | --- |
| 25 | 5 | 5 | 1 |
| 36 | 7 | 9 | 1.28571429 |
| 46 | 7 | 12 | 1.71428571 |
| 64 | 9 | 13 | 1.44444444 |
| 81 | 10 | 15 | 1.5 |
| 96 | 12 | 15 | 1.25 |
| 100 | 11 | 17 | 1.54545455 |
| 121 | 11 | 17 | 1.54545455 |
| 144 | 13 | 19 | 1.46153846 |
| 169 | 13 | 23 | 1.76923077 |
| 196 | 16 | 25 | 1.5625 |

We provided some graphs that we know their real minimum number of colors to the welsh powell algorithm.Table 2 shows that in some cases the heuristic algorithm does not provide the actual minimum number of colors needed.Then we plot the ratios with respect to sample size. Fit of the resulting graph has an equation of y = 0.0021x + 1,12516. We see that when sample size is increased error is increasing.



*, Figure 21*

As literature shows, the Welsh Powell algorithm gives exact minimum values in a very short time for CAR graphs and Register Allocation graphs but it works very badly at Queen graphs.

# **8. Experimental Analysis of the Correctness (Functional Testing)**

Black Box Testing for the Heuristic Graph Coloring Algorithm

We did some testing in this part by evaluating the functionality of the code in corner cases without considering the code itself.

Test case 1: Empty graph

* Description: Testing algorithm against empty graph.
* Graph: {}
* Expected output: {}
* Explanation: There are no vertices to color so it should output an empty dictionary.

Test case 2: Graph with no edges

* Description: Testing algorithm against graph without edges.
* Graph: {'1': [], '2': [], '3': [], ‘4': []}
* Expected output: {1: 1, 2: 1, 3: 1, 4: 1}
* Explanation: There are no edges so it should give the same color to all vertices.

Test case 3: Normal Graph

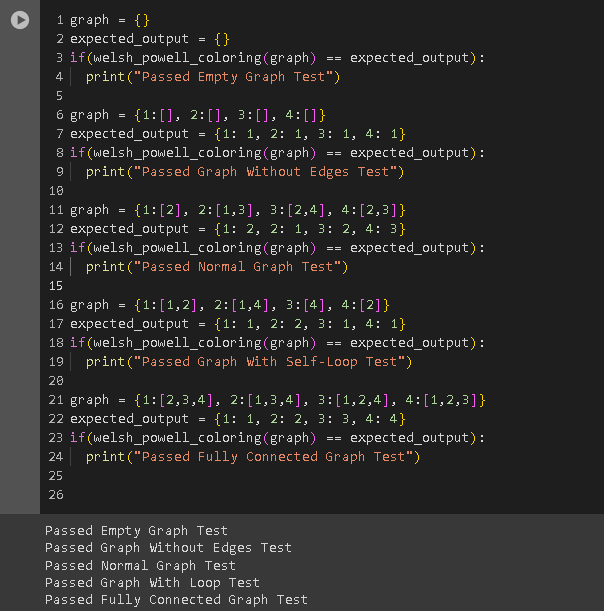
* Description:
* Graph: {1:[2], 2:[1,3], 3:[2,4], 4:[2,3]}
* Expected output: {1: 2, 2: 1, 3: 2, 4: 3}
* Explanation: It should give a proper coloring to the given graph.

Test case 4: Graph with self-loop

* Description: Testing algorithm against graph with self-loop
* Graph: {1:[1,2], 2:[1,4], 3:[4], 4:[2]}
* Expected output: {1: 1, 2: 2, 3: 1, 4: 1}
* Explanation: Algorithm should act like there is no self-loop.

Test case 5: Graph with fully connected components

* Description: Testing algorithm against a fully connected graph.
* Graph: {1:[2,3,4], 2:[1,3,4], 3:[1,2,4], 4:[1,2,3]}
* Expected output: {1: 1, 2: 2, 3: 3, 4: 4}
* Explanation: Since all vertices are connected to all other vertices it should give different colors to each vertex.



*Figure 23: Implementation of Black Box Test Cases*

Conclusion For the Black-box testing

As it can be seen in figure 23 algorithm passed all black box testings successfully. It supports our proof of the correctness of the algorithm by testing empty, not connected, normal, self-loop and fully connected graphs.

White Box Testing for the Heuristic Graph Coloring Algorithm

There are 5 if/else statements in our code. For white box testing we need to consider them in terms of statement coverage, decision coverage and path coverage.

Statement Coverage:

For statement coverage we need to make sure that all statements are executed in the test suite. Giving any input to the algorithm covers almost every statement in the algorithm. Test case 1 in our suite covers the statements in the first else statement which are executed if the graph is not empty. To cover the case where the if statement is inside the for loop of this else statement did not work, we need a graph without any edges.Test case 2 will cover that. Other if/else statements are covered in the test case 1 because it contains some vertices having the same color and some vertices having different colors.An empty graph would cover the first if statement. So, our test case 3 is an empty graph.

Decision Coverage

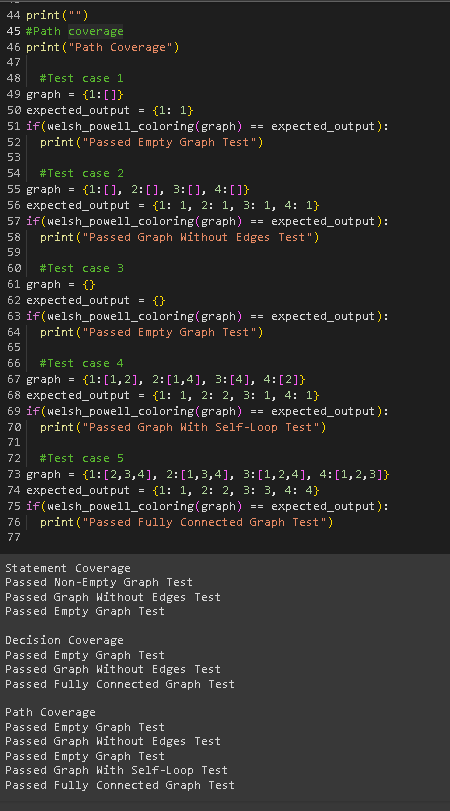
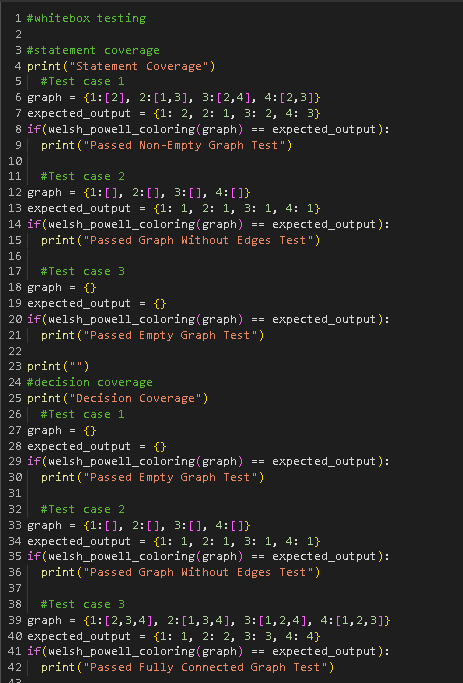
Our algorithm has 3 decisions. First it decides if the graph is empty or not.To do that we need an empty graph again.Second it checks for every vertex if they have edges connected to them or not. It can be tested by a graph without any edges. Finally it checks if the available colors list is empty or not. It can be checked by a fully connected graph.

Path Coverage

For path coverage we need to cover extreme cases such as:

* One vertex.
* Unconnected graph.
* Empty Graph.
* Self-Loop.
* Fully connected graph.

We cannot cover 100% of the paths but these cases will cover the corners.

******

*Figure 24: Implementation of White Box Testing*

# **9. Discussion**

When we compare brute force and welsh powell algorithms on graph coloring problems we see that welsh powell is more efficient to use in real life applications. This conclusion comes from the time complexity of the brute force algorithm. Its exponential time complexity makes the brute force algorithm unusable on problems with more than 20 vertices. Meanwhile the Welsh-Powell algorithm is way faster than the brute force algorithm but it has a tradeoff. It doesn't provide real optimal value. It works almost perfectly for some types of graph but works very bad for some others. It is still usable for real life applications because it has a small error ratio. It’s error is acceptable when we consider how fast it works.

Our whitebox and blackbox testing shows that our algorithm works without any error. Also our performance test shows that in real life our application works faster than the theoretical time complexity. It further proves that the Welsh-Powel algorithm works faster than the brute force algorithm.

# References

Dailey, D. P. (1980), "Uniqueness of colorability and colorability of planar 4-regular graphs are NP-complete", *Discrete Mathematics*, **30** (3): 289–293, [doi](https://en.wikipedia.org/wiki/Doi_(identifier)):[10.1016/0012-365X(80)90236-8](https://doi.org/10.1016%2F0012-365X%2880%2990236-8)

Garey, M. R.; Johnson, D. S.; Stockmeyer, L. (1974), "Some simplified NP-complete problems", *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, pp. 47–63, [doi](https://en.wikipedia.org/wiki/Doi_(identifier)):[10.1145/800119.803884](https://doi.org/10.1145%2F800119.803884)

Garey, M. R.; Johnson, D. S. (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, ISBN 0-7167-1045-5

Karp, R. M. (1972). *Reducibility among combinatorial problems*. In R. E. Miller & J. W. Thatcher (Eds.), Complexity of Computer Computations (pp. 85-103). Plenum Press.

Kettani, O., Ramdani, F., & Tadili, B. (2013). A Heuristic Approach for the Vertex Cover Problem. International Journal of Computer Applications, 82(4), 9.

M. Rajagaspar & S. Senthil (2022), “Applications of Graph Coloring Using Vertex Coloring”, *Journal of Algebraic Statics*, 13 (2): 3447-3454

T. Satish, B. Seema, S. Anand (2022), “Applications of graph coloring in various fields”, *Materials Today: Proceedings*, 66 (8): 3498-3501, <https://doi.org/10.1016/j.matpr.2022.06.392>.

Rhyd Lewis (2016) Graph Colouring: An Ancient Problem with Modern Applications, Impact, 2:1, 47-50, DOI: 10.1080/2058802X.2016.11963998

A., Murat (2016), “A Performance Comparison of Graph Coloring Algorithms”, *International Journal of Intelligent Systems and Applications in Engineering, 2147-67992*, <https://dergipark.org.tr/en/download/article-file/254140>

1. Karp, R. M. (1972) [↑](#footnote-ref-0)
2. Dailey, D. P. (1980), Garey, Johnson & Stockmeyer (1974), Garey & Johnson (1979). [↑](#footnote-ref-1)