CS301

2023-2024 Spring

Project Report

Group 41

::Group Members::

Ayça Elif Aktaş 27802

Mehmet Talha Güven 27829

Table of Contents

1. Problem Description

1.1 Overview

The Graph Coloring Problem (GCP) is a well-known NP-Complete problem. Graph coloring includes both vertex coloring and edge coloring. However, the term graph coloring usually refers to vertex coloring rather than edge coloring. The graph coloring issue requires assigning colors to specific elements of a graph while adhering to some limitations and restrictions. In other words, graph coloring is the process of assigning colors to vertices so that no two adjacent vertices have the same color.

1.2 Decision Problem

A appropriate vertex coloring problem for a given graph G is to color all the vertices with different colors in such a way that any two adjacent vertices are given different colors. In graph theory, a vertex coloring with k colors is a mapping f: V (G) → N such that:
$\forall vi,vj \in V\ (G), i \neq j \exists (ei,ej) \Rightarrow f(i) \neq f(j)$.[1]

1.3 Optimization Problem

Considering an undirected graph G=(V,E), determine an appropriate coloring function χ:V→C that minimizes the amount of colors |C| subject to the constraint that no two neighboring vertices can have the same color. In this context, the Graph Coloring Problem consists of determining the minimum number k of different colors in order to carry out a coloring of G itself. This number k is known as the chromatic number of G, denoted by **X(G).**

[1] FormanowiczPiotr and TanaśKrzysztof. "A survey of graph coloring - its types, methods and applications" *Foundations of Computing and Decision Sciences* 37, no.3 (2012): 223-238. https://doi.org/10.2478/v10209-011-0012-y

1.4 Example Illustration

Figure 1 illustrates simple graph with 4 vertices.

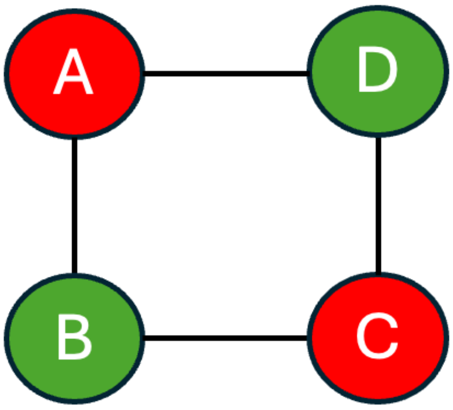


Figure 1: Illustration of Simple Graph

This graph shows a correct 3-coloring. To ensure that no two adjacent vertices have the same color, one potential color scheme could be: Vertex A is red, Vertex B is green, Vertex C is blue, and vertex D is green. Although B and D share the same color (Green), as seen in the graph, they are not adjacent. On the other hand, vertices B and C are connected by an edge, and they have different colors (Green and Blue) to satisfy the coloring constraint.

Figure 2 illustrates complete graph with 4 vertices.

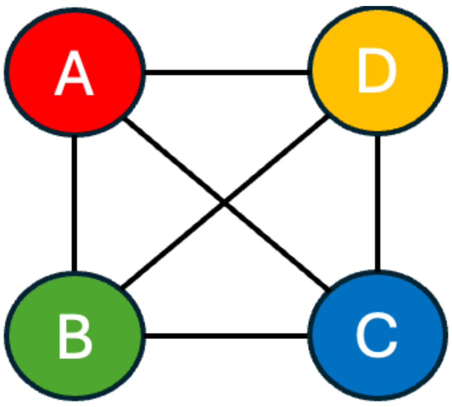


Figure 2: Illustration of Complete Graph

A complete graph has edges that connect every vertex to every other vertex. Here, there are edges connecting each of the four vertices (A, B, C, and D) to the other three. Consider attempting to paint a whole graph with V colors (in this example, V = 4). Since each vertex is linked to every other vertex, giving any vertex the same color would lead it to have near neighbors that share the same color, which would be against the coloring constraints. As a result, in order to avoid violating the coloring rule, each vertex in a complete graph must have a distinct color. In the example with four vertices, Red, Blue, Green, and Yellow were chosen to represent the four different colors.

Figure 3 illustrates bipartite graph with 6 vertices.

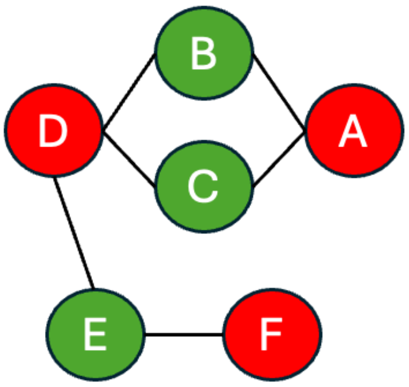


Figure 3: Illustration of Bipartite Graph

A bipartite graph is one in which the vertices are separated into two distinct sets with no edges linking them. In other words, all edges connect vertices from different sets. In this case, the graph is separated into two sets: Set 1: {A, D, F} (in red) Second set: {B, C, E} (in green). This characteristic allows us to always color a bipartite graph in two colors. Since the graph is bipartite, we can separate the vertices into two sets with no edges in between. By design, this coloring assures that no two adjacent vertices share the same color since edges only connect vertices from different sets with the different colors given.

1.5 Real World Applications

1.5.1 Map Coloring

Assigning colors to regions on a map such that adjacent regions do not share the same color is known as the classic map coloring issue. Applications of this topic include resource distribution in geographical areas, political border separation, and mapping.

1.5.2 Scheduling

Tasks or events are frequently allocated to resources or time slots in scheduling challenges while adhering to certain requirements. Graph coloring is an effective way to model and solve scheduling issues. For instance, graph coloring can be used to help assign time slots to examinations in exam scheduling, where exams must be planned so that no student has conflict exams simultaneously.

1.5.3 Wireless Frequency Assignment

In order to prevent interference in wireless communication networks, different devices must be given different frequencies or channels. Using graph coloring, one may simulate the limitations on device interference and determine the best way to allocate frequencies in order to reduce interference.

1.5.4 Register Allocation in Compilers

Compilers assign registers to variables when converting high-level programming languages into machine code. Graph coloring methods are used to figure out which variables may be assigned to the same register without causing conflicts. This enhances the efficiency of generated code and maximizes the utilization of hardware resources.

1.6 Hardness of the Problem

In their seminal work "Computers and Intractability: A Guide to the Theory of NP-Completeness" (1979), Garey and Johnson proved that the graph coloring problem is NP-complete.

In this book, Garey and Johnson provided a comprehensive proof that the graph coloring problem belongs to the class of NP-complete problems. They demonstrated this by reducing a well-known NP-complete issue, such as the Boolean satisfiability problem (SAT), into a graph coloring problem. This reduction implies that if we can solve the graph coloring issue effectively, we can solve SAT efficiently as well, indicating that graph coloring is NP-complete.

1.6.1 Proving NP

Since we have a coloring function χ:V→C and a graph G=(V,E) with a set of colors C, we can use polynomial time to check if χ is a suitable coloring for G by making sure that no two adjacent vertices have the same color. This verification takes $O(|V|+|E|)$ time, which is polynomial in input size. As a result, the graph coloring problem is in NP.

1.6.2 Proving NP-Hard

In order to demonstrate the NP-hardness of the graph coloring issue, we usually convert a recognized NP-complete problem to an instance of the graph coloring problem. The NP-complete Boolean satisfiability problem (SAT) is one of the most commonly used problems for this reduction.[2] Given an instance of SAT with n Boolean variables and m clauses, we create a graph G in which each variable corresponds to a vertex and each clause to a set of vertices connected by edges.

1.6.3 Conclusion to NP-Complete

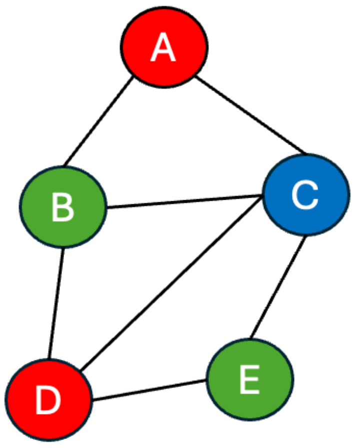
NP-completeness of the graph coloring issue is determined by combining the NP and NP-hardness arguments. This indicates that it is both NP-hard, hard as the toughest problems in NP, and in NP since solutions can be verified in polynomial time. As a result, the graph coloring problem is one of the most computationally challenging problems in the class of NP.

2. Algorithm Description (Brute Force)

2.1 Overview

The brute force technique for the graph coloring problem goes over all potential graph colorings in order to find a valid coloring with the minimum number of colors. This approach is inefficient since it considers every possible color combination for the vertices, resulting in exponential time complexity. However, if given enough time, it will find an optimal solution.

[2] Garey, M. R. and Johnson, D. S.. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. First Edition : W. H. Freeman, 1979.



A	B	C	D	E
Red	Green	Blue	Red	Green

Figure 4: An Example Coloring Table (below) and Its Visual Representation (above)

The first step in the brute force graph coloring algorithm is to initialize an empty list to hold all possible colorings. After that, it considers every potential combination of colors and creates every possible permutation of colors for the vertices. For each permutation, the algorithm assigns colors to the vertices and checks that the coloring is proper, thereby guaranteeing that no adjacent vertices share the same color. If an appropriate coloring is found, it is added to the list of possible colors. After analyzing all permutations, the algorithm chooses the coloring with the minimum number of colors, ensuring an optimum result.

2.2 Pseudocode

```
-----------------------------------------------------------------------------------------
Step 1:  Initialize an empty list to store the final coloring with minimum number of color.
Also Initialize a variable to store the minimum color.
Step 2:  From one color up to n (number of vertex) colors generate all permutations of
colorings for the vertices.
Step 2.1:  For each permutation in all permutations:
Step 2.2:  Assign colors to vertices based on the current permutation and form the current
graph.
Step 2.3:    Check if the coloring is proper (no adjacent vertices have the same color).
Step 2.3.1:  For each edge in the graph:
Step 2.3.2:    If the colors of the edge's endpoints are the same:
```

```
Step 2.3.3:    Mark the coloring as improper.
Step 2.4:    If the coloring is proper:
Step 2.4.1:    If the number of colors used is less than the minimum:
Step 2.4.2:    Update the minimum number of colors and the corresponding coloring.
Step 3:    Return the coloring with the minimum number of colors.
```

2.2 Heuristic Algorithm

2.2.1 Overview

For the graph coloring problem, we propose applying the Greedy Coloring Algorithm as a heuristic. This approach is efficient, running in polynomial time, and offers a fair estimate to find the minimum coloring of the graph. Utilizing an iterative method, the Greedy Coloring Algorithm colors each vertex separately, selecting the lowest color that doesn't clash with the colors of nearby vertices. At each step, it chooses a vertex and evaluates its neighbors' colors. It then assigns the smallest available color that has not been used by any nearby vertex. This process repeats until all vertices are colored.

The Greedy Coloring Algorithm uses at most $\Delta + 1$ colors, where Δ is the graph's maximum degree. This was demonstrated by Matula and Marble in their work "Graph coloring algorithms" (1983). The evidence demonstrates that the Greedy Coloring Algorithm generates a coloring that is at most $\Delta + 1$ times the optimal coloring. As a result, the Greedy Coloring Algorithm produces a coloring that is within a specific factor of the optimal solution, making it an efficient solution for the graph coloring issue.

2.2.1 Pseudocode

```
Step 1: Initialize an empty dictionary 'coloring' to store the colors assigned to vertices.
Step 2: Initialize a list 'available_colors' to store the available colors.
Step 3: Initialize a variable 'max_color' to store the maximum color used.
Step 4: Sort the vertices of the graph in descending order of degree.
Step 5: For each vertex v in sorted_vertices:
Step 5.1: neighbors = GetNeighbors(v) # Get the neighbors of vertex v.
Step 5.2: used_colors = {coloring[n] for n in neighbors if coloring[n] is not None}
# Get the colors used by neighbors.
```

Step 5.3: available_colors = set(range(max_color + 2)) - used_colors # Get the available colors for vertex v.

Step 5.4: coloring[v] = min(available_colors) # Assign the smallest available color to vertex v.

Step 5.5: max_color = max(max_color, coloring[v]) # Update the maximum color used.

Step 6: Return coloring, max_color # Return the coloring and the maximum color used.

----

3. Algorithm Analysis

3.1 Brute Force Algorithm

3.1.1 Correctness Analysis

***Claim**:* Let G = (V, E) be a graph where V represents the vertices and E represents the edges. A coloring of G is defined as a function c : V → {1, 2, . . . , k}, where k is the number of colors. The coloring is proper if for every edge (u, v) ∈ E, c(u)/= c(v).[3] The chromatic number X(G) is defined as the smallest number of colors required to achieve a proper coloring of G. The brute force graph coloring algorithm finds a proper coloring of G using exactly X(G) colors, which is the minimum number necessary.

Proof (by contradiction): Consider an undirected graph G = (V, E) with n vertices and m edges. The goal of the brute force graph coloring algorithm is to find a minimal coloring of G such that no two adjacent vertices share the same color, utilizing the smallest number of colors possible. This number is denoted as X(G), the chromatic number of G. The algorithm's procedure is as follows:

1. Iterate over all possible colorings of the vertices, from one color up to n colors.

2. For each coloring, verify if it is proper; i.e., ensure no two adjacent vertices share the same color.

[3] Aslan, Murat, and Nurdan Akhan Baykan. "A Performance Comparison of Graph Coloring Algorithms". International Journal of Intelligent Systems and Applications in Engineering 4, no. Special Issue-1 (December 2016): 1-7. https://doi.org/10.18201/ijisae.273053.

3. If the coloring is proper and uses fewer colors than any previously found proper coloring, record it as the current optimal solution.

To demonstrate that the algorithm works correctly, we focus on two aspects:

(1) The algorithm always finds a proper coloring.

(2) The algorithm finds the optimal (minimal) coloring.

During its execution, the algorithm considers each possible assignment of colors to vertices. If a particular coloring satisfies the condition where no two adjacent vertices share the same color, it is deemed proper. In the worst case, the algorithm will consider assigning a unique color to each vertex, which is trivially a proper coloring. Thus, the algorithm is guaranteed to find at least one proper coloring.

Suppose for the sake of contradiction, that the algorithm returns a proper coloring C using m colors, where m > X(G). Let C′ be a proper coloring of G that uses X(G) colors. Since C is not minimal, we have m > |C′|. Consider the iteration where the algorithm checks colorings using |C′| colors. Since C′ is a proper coloring by definition, this subset of colors must also result in a proper coloring. Therefore, the algorithm should have recognized C′ as a proper coloring using fewer colors than m and recorded C′ as the optimal solution. This contradicts our assumption that m colors were necessary. Hence, the assumption that the algorithm finds a non-optimal solution m > X(G) leads to a contradiction. Therefore, the brute force algorithm must find the chromatic number X(G) and achieve the optimal coloring.

This proof by contradiction confirms that the brute force algorithm not only finds a proper coloring but also ensures that this coloring uses the fewest possible colors, thus solving the graph coloring problem optimally.

3.1.2 Time Complexity

Let n be the number of vertices in the graph. Let m be the number of edges. For k colors, there are k^n possible ways to assign these colors to n vertices. The algorithm checks color assignments starting from one color up to n colors. The complexity for each k involves generating k^n colorings and checking each coloring against m edges to ensure it is proper. The algorithm will have to check every edge and compare the vertices that are connected by that edge. That would give us (m+n) steps to do. Remember that we need to repeat this step for every possible vertex-color mapping. This leads to the following summation for total operations:

$$\sum_{k=1}^{n} k^n \, x \, (\mathrm{m} + \mathrm{n})$$

Given that k^n grows exponentially with increasing k, the dominant term in the sum is when k= n. Thus, the worst-case time complexity can be simplified to: O(n^n × (m +n)). This represents the exponential growth in the number of colorings multiplied by the linear factor from checking the validity of each coloring.

3.1.3 Space Complexity

The space complexity of the brute force graph coloring algorithm primarily depends on the storage requirements for: the graph itself, the storage of colors for each vertex, the space needed to maintain a list of valid colorings found during execution (if storing all valid solutions is required). In our case the algorithm stores the graph using an adjacency list which takes O(V+E) space, where V is the number of vertices and E is the number of edges. The current coloring of vertices at any time during the algorithm's execution requires an array of size V, where each entry stores the color of a vertex. This takes O(V) space. If the algorithm opts to store every valid coloring it finds (to later determine the one with the minimum number of colors), this can significantly increase the space requirement. In the worst case, where nearly all colorings are valid, this would require space proportional to the number of colorings multiplied by the number of vertices. However, in our case the algorithm stores only the best coloring found so far and the current number of colors that are used. In such a case, the space complexity remains O(V). The primary space usage is from the graph storage and the array of current vertex colors. Thus, the space complexity is O(V+E) when using an adjacency list, with an additional O(V) for the current vertex colors, not significantly altering the overall complexity.

3.2 Heuristic Algorithm

3.2.1 Correctness Analysis

Claim: The graph is correctly colored by the heuristic algorithm for the graph coloring issue.

Theorem: Let G = (V, E) denote an undirected graph. The heuristic algorithm for graph coloring results in a valid coloring of G.

Proof:

1. The algorithm iterates through the graph's vertices in descending order of degrees. This ensures that vertices with higher degrees are prioritized, possibly preventing conflicts with adjacent vertices.
2. The algorithm assigns the smallest color that is available and not in use by any of the vertex v's neighbors. This greedy method seeks to reduce conflicts by choosing colors that are unlikely to cause adjacent vertices to share the same color.
3. The program always selects the smallest color that meets the coloring rules, therefore by allocating colors to each vertex in this way, it guarantees that no two adjacent vertices share the same color.
4. The algorithm ensures that the final coloring is correct and does not violate any coloring rules because it traverses all of the vertices one by one and assigns colors depending on how adjacent vertices are colored.
5. Thus, the heuristic algorithm generates a correct graph coloring.

3.2.2 Time Complexity

The Graph Coloring Problem heuristic algorithm's worst-case scenario occurs if the graph is fully connected, which means that each pair of distinct vertices is connected by an edge. Let's examine further why the heuristic algorithm's worst-case time complexity occurs:

1. In color assignment step, each vertex has to be assigned a color. The algorithm chooses the minimum color that is available by examining the colors of its

adjacent vertices. In the worst-case scenario, each vertex may need to check all of its neighbors before assigning its color. This process requires O(Δ) time per vertex (where Δ is the maximum degree of the graph) because in an undirected graph with n vertices, a vertex can have at most n−1 neighbors.

2. The heuristic algorithm iterates over all the graph's vertices. In order to assign colors, iteratively traverses over each vertex's neighbors. In the worst-case scenario, this would mean iterating over all vertices in the graph and then iterating over each vertex's neighbors. This step increases the overall complexity by O(n·Δ) since there are n vertices and Δ is the maximum degree.
3. We find that the worst-case time complexity is Θ(n·Δ) by combining the difficulties of the heuristic algorithm and the color assignment step.

3.2.3 Space Complexity

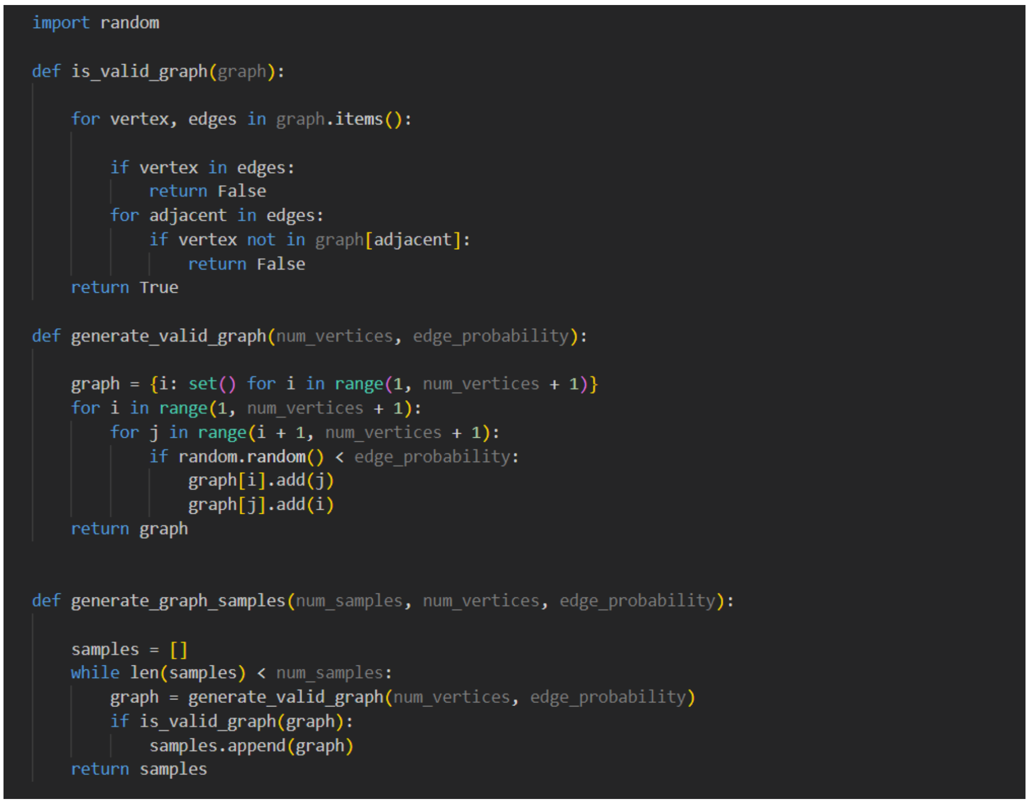
The colors that are assigned to each vertex needs to be stored by the algorithm. Since each vertex is assigned a color, the space needed for storing the colors is proportional to the number of vertices in the graph, which leads to a space complexity of O(n), where n is the number of vertices. While the algorithm might use temporary data structures like arrays to track color usage during the coloring process, these temporary structures have a negligible impact on space complexity in fully connected graphs.

4. Sample Generation (Random Instance Generator)

The random instance generator algorithm is designed to create random instances of graphs for testing the graph coloring algorithm. It produces the graph as a representation of an adjacency list. It takes three arguments: the sample size and the number of vertex and edge probability.

1. *Function is_valid_graph(graph):* This function checks if a graph (represented as a dictionary where keys are vertices and values are sets of adjacent vertices) is valid. It iterates through each vertex and its edges to ensure two conditions: There are no self-loops: A vertex should not be connected to itself. The graph is undirected: If vertex A is connected to vertex B, then B must also be connected to A.

2. *Function generate_valid_graph(num_vertices, edge_probability):* Initializes a graph with ”num_vertices” vertices as a dictionary with vertex keys and each value is an empty set. It then iterates over pairs of vertices (i, j). For each pair, it uses the edge_probability to decide whether to create an edge between them. If the random condition is met, an edge is added by updating the sets of both vertices (i and j) to include each other, maintaining the undirected nature of the graph.
3. *Function generate_graph_samples(num_samples, num_vertices, edge_probability):* This is the main function that generates the desired number of graph samples (num_samples) with each graph having ”num_vertices” vertices.It initializes an empty list (samples) to hold the generated graphs. It enters a loop that continues until the list has the required number of samples. Within the loop: It calls generate_valid_graph to create a graph graph having” num_vertices” vertices. It checks the validity of the graph with is_valid_graph. If valid, it adds the graph to the samples list.



```
import random

def is_valid_graph(graph):

    for vertex, edges in graph.items():

        if vertex in edges:
            return False
        for adjacent in edges:
            if vertex not in graph[adjacent]:
                return False
    return True

def generate_valid_graph(num_vertices, edge_probability):

    graph = {i: set() for i in range(1, num_vertices + 1)}
    for i in range(1, num_vertices + 1):
        for j in range(i + 1, num_vertices + 1):
            if random.random() < edge_probability:
                graph[i].add(j)
                graph[j].add(i)
    return graph


def generate_graph_samples(num_samples, num_vertices, edge_probability):

    samples = []
    while len(samples) < num_samples:
        graph = generate_valid_graph(num_vertices, edge_probability)
        if is_valid_graph(graph):
            samples.append(graph)
    return samples
```

Figure 5: Implementation of random sample generation

5. Algorithm Implementation

5.1 Algorithm

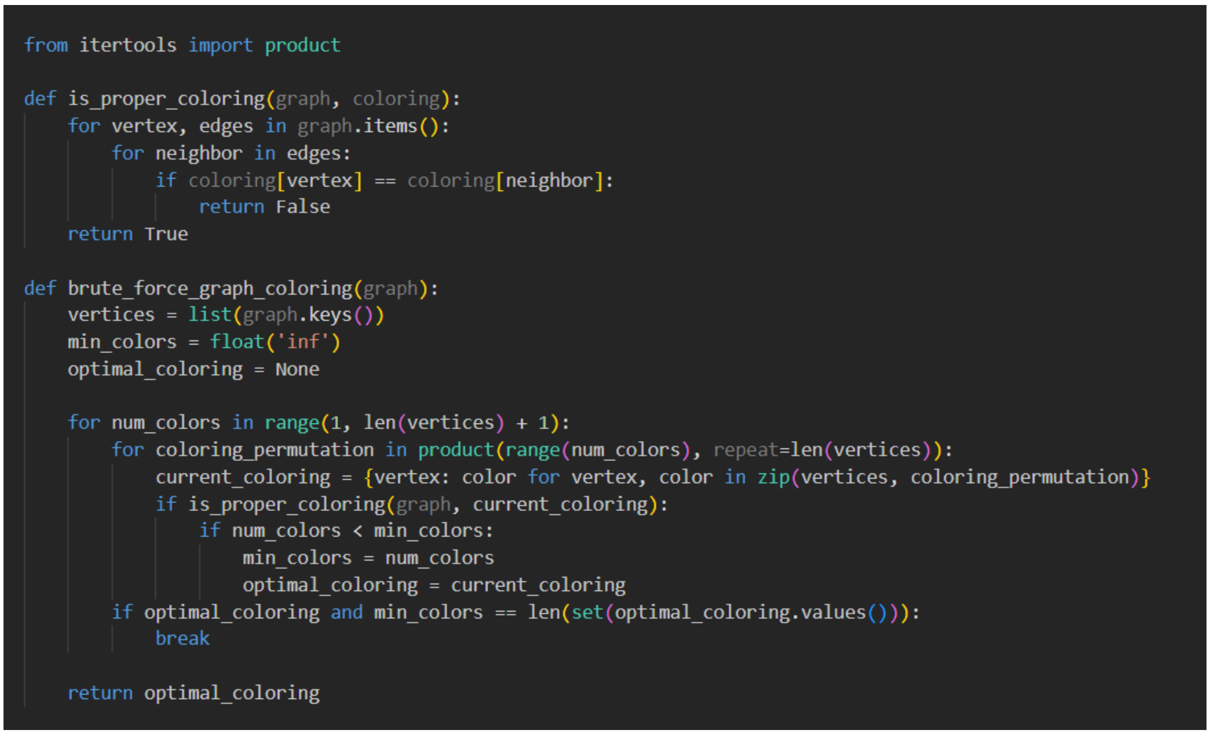
1. *is_proper_coloring(graph, coloring):* This function checks whether a given coloring of a graph is valid, where no two adjacent vertices have the same color.

- **Step 1:** Iterate over each vertex in the graph dictionary.
- **Step 2:** For each vertex, iterate over its neighbors.
- **Step 3:** Check if the vertex and its neighbor have the same color. If they do, return **False**, indicating the coloring is not proper.
- **Step 4:** If no conflicts are found throughout the entire graph, return **True**, indicating that the coloring is proper.

2. *brute_force_graph_coloring(graph):* This function finds an optimal coloring for the graph that uses the minimum number of colors, ensuring no two adjacent vertices share the same color.

- **Step 1:** Extract and list the vertices from the graph's keys.
- **Step 2:** Initialize **min_colors** with a very high value (**float('inf')**) to keep track of the smallest number of colors found that can properly color the graph.
- **Step 3:** Initialize **optimal_coloring** as **None** to store the best coloring configuration when found.
- **Step 4:** Loop through possible numbers of colors:
    - **Sub-step 4.1:** For each possible number of colors (**num_colors** from one to the number of vertices):
        - **Sub-step 4.2:** Generate all possible color assignments for the vertices using **product(range(num_colors), repeat=len(vertices))**. This creates every combination of colors for all vertices given the current **num_colors**.
        - **Sub-step 4.3:** For each color combination (**coloring_permutation**):

        - **Sub-step 4.4:** Create a dictionary **current_coloring** mapping each vertex to a color from the permutation.
        - **Sub-step 4.5:** Use **is_proper_coloring** to verify if **current_coloring** is a valid coloring. If it is:
            - **Sub-step 4.6:** Check if the current number of colors is less than **min_colors**. If so:
                - **Sub-step 4.7:** Update **min_colors** to the current number of colors.
                - **Sub-step 4.8:** Set **optimal_coloring** to this valid coloring configuration.
    - **Sub-step 4.9:** After checking all permutations for the current **num_colors**, if **optimal_coloring** uses exactly **min_colors** different colors, break the loop as no fewer colors will suffice, achieving the minimal coloring configuration.
- **Step 5:** Return **optimal_coloring**, which now contains the vertex-color mappings using the fewest possible colors to ensure no two adjacent vertices are colored the same.



```python
from itertools import product

def is_proper_coloring(graph, coloring):
    for vertex, edges in graph.items():
        for neighbor in edges:
            if coloring[vertex] == coloring[neighbor]:
                return False
    return True

def brute_force_graph_coloring(graph):
    vertices = list(graph.keys())
    min_colors = float('inf')
    optimal_coloring = None

    for num_colors in range(1, len(vertices) + 1):
        for coloring_permutation in product(range(num_colors), repeat=len(vertices)):
            current_coloring = {vertex: color for vertex, color in zip(vertices, coloring_permutation)}
            if is_proper_coloring(graph, current_coloring):
                if num_colors < min_colors:
                    min_colors = num_colors
                    optimal_coloring = current_coloring
        if optimal_coloring and min_colors == len(set(optimal_coloring.values())):
            break

    return optimal_coloring
```

Figure 6: Implementation of brute-force graph coloring algorithm

5.1.1 Initial Testing of the Algorithm

15 number of instances tried, there were no issues, errors or failures. These are the results of the *brute-force graph coloring algorithm* for 15 graphs with different number of vertices all with edge_probability = 0.5.

1. Graph with 1 vertices: {1: ()}

   Optimal Coloring: {1: 0}

2. Graph with 2 vertices: {1: (), 2: ()}

Optimal Coloring: {1: 0, 2: 0}

3. Graph with 3 vertices: {1: {2}, 2: {1}, 3: ()}

Optimal Coloring: {1: 0, 2: 1, 3: 0}

4. Graph with 4 vertices: {1: {4}, 2: {3, 4}, 3: {2}, 4: {1, 2}}
   Optimal Coloring: {1: 0, 2: 0, 3: 1, 4: 1}

5. Graph with 5 vertices: {1: {3, 4, 5}, 2: {4, 5}, 3: {1, 5}, 4: {1, 2}, 5: {1, 2, 3}}
   Optimal Coloring: {1: 0, 2: 0, 3: 1, 4: 1, 5: 2}

6. Graph with 6 vertices: {1: {4}, 2: {4, 6}, 3: {4, 5}, 4: {1, 2, 3, 6}, 5: {3, 6}, 6: {2, 4, 5}}
   Optimal Coloring: {1: 0, 2: 0, 3: 0, 4: 1, 5: 1, 6: 2}

7. Graph with 7 vertices: {1: {3, 4, 5, 6}, 2: {3, 4, 5}, 3: {1, 2, 6, 7}, 4: {1, 2, 5}, 5: {1, 2, 4, 6}, 6: {1, 3, 5, 7}, 7: {3, 6}}
   Optimal Coloring: {1: 0, 2: 0, 3: 1, 4: 2, 5: 1, 6: 2, 7: 0}

8. Graph with 7 vertices: {1: {6, 7}, 2: {4, 5}, 3: {4, 7}, 4: {2, 3, 6, 7}, 5: {2}, 6: {1, 4}, 7: {1, 3, 4}}
   Optimal Coloring: {1: 0, 2: 0, 3: 0, 4: 1, 5: 1, 6: 2, 7: 2}

9. Graph with 8 vertices: {1: {2, 6, 7}, 2: {1, 3, 4, 5}, 3: {8, 2, 5, 6}, 4: {2, 5, 7}, 5: {2, 3, 4, 7}, 6: {1, 3}, 7: {8, 1, 4, 5}, 8: {3, 7}}
   Optimal Coloring: {1: 0, 2: 1, 3: 0, 4: 0, 5: 2, 6: 1, 7: 1, 8: 2

10. Graph with 8 vertices: {1: {6}, 2: {3, 5, 6, 7, 8}, 3: {2, 4, 5, 6, 7, 8}, 4: {3}, 5: {8, 2, 3}, 6: {8, 1, 2, 3}, 7: {2, 3}, 8: {2, 3, 5, 6}}
    Optimal Coloring: {1: 0, 2: 0, 3: 1, 4: 0, 5: 2, 6: 2, 7: 2, 8: 3}

11. Graph with 9 vertices: {1: {3, 6}, 2: {8, 9, 6, 7}, 3: {1, 4, 5, 6, 8}, 4: {9, 3, 7}, 5: {8, 3, 6, 7}, 6: {1, 2, 3, 5, 7, 9}, 7: {2, 4, 5, 6, 9}, 8: {2, 3, 5}, 9: {2, 4, 6, 7}}
    Optimal Coloring: {1: 0, 2: 0, 3: 1, 4: 0, 5: 0, 6: 2, 7: 1, 8: 2, 9: 3}

12. Graph with 10 vertices: {1: {5, 6, 7, 9, 10}, 2: {3, 4, 6}, 3: {8, 2}, 4: {8, 2, 5, 7}, 5: {1, 10, 4, 6}, 6: {1, 2, 10, 5}, 7: {8, 1, 4, 9}, 8: {3, 4, 7, 9, 10}, 9: {8, 1, 10, 7}, 10: {1, 5, 6, 8, 9}}
    Optimal Coloring: {1: 0, 2: 0, 3: 1, 4: 1, 5: 2, 6: 1, 7: 2, 8: 0, 9: 1, 10: 3}

13. Graph with 11 vertices: {1: {2, 3, 6, 7, 9, 10, 11}, 2: {1, 4, 5, 7, 10}, 3: {1, 6, 7, 8, 10, 11}, 4: {2, 5, 6, 7, 9}, 5: {2, 4, 6, 7, 8, 10, 11}, 6: {1, 3, 4, 5, 8}, 7: {1, 2, 3, 4, 5, 8, 10, 11}, 8: {3, 5, 6, 7}, 9: {1, 11, 4}, 10: {1, 2, 3, 5, 7, 11}, 11: {1, 3, 5, 7, 9, 10}}
    Optimal Coloring: {1: 0, 2: 1, 3: 1, 4: 2, 5: 0, 6: 3, 7: 3, 8: 2, 9: 1, 10: 2, 11: 4}

14. Graph with 12 vertices: {1: {8, 10, 6, 7}, 2: {4, 5, 7, 8, 9}, 3: {4, 5, 6, 7, 11}, 4: {2, 3, 6, 7, 10, 11}, 5: {2, 3, 7}, 6: {1, 3, 4, 9, 10}, 7: {1, 2, 3, 4, 5, 9, 10, 11, 12}, 8: {1, 2, 9}, 9: {2, 6, 7, 8, 10, 11}, 10: {1, 4, 6, 7, 9}, 11: {3, 4, 7, 9, 12}, 12: {11, 7}}
    Optimal Coloring: {1: 0, 2: 0, 3: 0, 4: 1, 5: 1, 6: 2, 7: 2, 8: 2, 9: 1, 10: 3, 11: 3, 12: 0}

15. Graph with 13 vertices: {1: {6, 10, 11, 12, 13}, 2: {8, 10, 13, 5}, 3: {4, 5, 6, 7, 11}, 4: {3, 8, 9, 10, 11, 12, 13}, 5: {8, 2, 3}, 6: {1, 3, 7, 8, 9, 10, 13}, 7: {3, 6, 8, 9, 12, 13}, 8: {2, 4, 5, 6, 7, 9, 10, 11, 13}, 9: {4, 6, 7, 8, 10, 11}, 10: {1, 2, 4, 6, 8, 9, 12, 13}, 11: {1, 3, 4, 8, 9, 12}, 12: {1, 4, 7, 10, 11}, 13: {1, 2, 4, 6, 7, 8, 10}}
    Optimal Coloring: {1: 0, 2: 1, 3: 0, 4: 1, 5: 2, 6: 1, 7: 2, 8: 0, 9: 3, 10: 2, 11: 2, 12: 3, 13: 3}

5.2 Heuristic Algorithm

A detailed description of the algorithm's implementation can be found in section 3.2., Figure 7 illustrates an implementation of the heuristic algorithm that was taken from the internet[4]. Twelve samples are used with 12 vertices for the first testing, and they are produced using the sample generator tool from section 4. No errors have been found. For 15 graphs with edge_probability = 0.5, these are the outcomes of the heuristic graph coloring algorithm.

1. Generated Graph: {1: {8, 9, 11, 6}, 2: {3, 7, 9, 10, 12}, 3: {8, 2, 5}, 4: {9, 10, 11, 12}, 5: {8, 9, 3, 6}, 6: {1, 11, 5, 7}, 7: {2, 6, 8, 9, 10, 12}, 8: {1, 3, 5, 7}, 9: {1, 2, 4, 5, 7, 10, 12}, 10: {2, 4, 7, 9, 11}, 11: {1, 4, 6, 10, 12}, 12: {2, 4, 7, 9, 11}, }
   Colors Assigned: {1: {1: 1}, 2: {2: 2}, 3: {3: 3}, 4: {4: 1}, 5: {5: 1}, 6: {6: 2}, 7: {7: 1}, 8: {8: 0}, 9: {9: 0}, 10: {10: 3}, 11: {11: 0}, 12: {12: 3}}

2. Generated Graph: {1: {11, 2, 10, 4}, 2: {1, 3, 4, 7, 8, 9, 11}, 3: {2, 4, 8, 10, 11, 12}, 4: {1, 2, 3, 5, 7, 9, 10, 11}, 5: {4, 6, 7, 8, 9, 10, 12}, 6: {5, 7, 9, 10, 12}, 7: {2, 4, 5, 6, 8, 12}, 8: {2, 3, 5, 7, 9, 10}, 9: {2, 4, 5, 6, 8, 10, 11, 12}, 10: {1, 3, 4, 5, 6, 8, 9, 12}, 11: {1, 2, 3, 4, 9, 12}, 12: {3, 5, 6, 7, 9, 10, 11}, }
   Colors Assigned: {1: 1, 2: 2, 3: 1, 4: 0, 5: 3, 6: 4, 7: 1, 8: 0, 9: 1, 10: 2, 11: 3, 12: 0}

3. Generated Graph: {1: {3, 5, 7, 11, 12}, 2: {8, 9, 3, 12}, 3: {1, 2, 4, 7, 8, 10, 11}, 4: {9, 10, 3, 7}, 5: {1, 10, 6}, 6: {10, 5, 7}, 7: {1, 3, 4, 6, 8, 9, 11}, 8: {11, 2, 3, 7}, 9: {2, 4, 7, 10, 12}, 10: {3, 4, 5, 6, 9}, 11: {1, 3, 7, 8, 12}, 12: {1, 2, 11, 9}, }
   Colors Assigned: {1: 2, 2: 1, 3: 0, 4: 2, 5: 0, 6: 2, 7: 1, 8: 2, 9: 0, 10: 1, 11: 3, 12: 4}

4. Generated Graph: {1: {3, 5, 7, 10, 12}, 2: {3, 7, 8, 9, 10}, 3: {1, 2, 5, 6}, 4: {5, 6, 7}, 5: {1, 3, 4, 7, 9, 12}, 6: {9, 3, 4, 12}, 7: {1, 2, 4, 5, 9, 10}, 8: {2, 11}, 9: {2, 5, 6, 7, 11, 12}, 10: {1, 2, 12, 7}, 11: {8, 9}, 12: {1, 5, 6, 9, 10}, }
   Colors Assigned: {1: 2, 2: 0, 3: 1, 4: 2, 5: 0, 6: 0, 7: 1, 8: 1, 9: 2, 10: 3, 11: 0, 12: 1}

[4] ALMARA'BEH, Hilal, and Amjad SULEIMAN. *Heuristic Algorithm for Graph Coloring Based On Maximum Independent Set*. Stefan cel Mare University of Suceava.

5. Generated Graph: {1: {2, 3, 5, 7, 8}, 2: {1, 4, 5, 6, 7, 10}, 3: {1, 4, 6, 7, 8, 11, 12}, 4: {2, 3, 5}, 5: {1, 2, 4, 7, 9, 11}, 6: {2, 3, 8, 9, 11}, 7: {1, 2, 3, 5, 9, 10}, 8: {1, 3, 6, 9, 10}, 9: {5, 6, 7, 8, 10, 11}, 10: {2, 7, 8, 9, 11, 12}, 11: {3, 5, 6, 9, 10}, 12: {10, 3}, }

   Colors Assigned: {1: 3, 2: 1, 3: 0, 4: 2, 5: 0, 6: 2, 7: 2, 8: 4, 9: 1, 10: 0, 11: 3, 12: 1}

6. Generated Graph: {1: {8, 11, 2, 10}, 2: {1, 3, 4, 5, 9, 12}, 3: {2, 5, 7, 10, 11}, 4: {2, 10, 5, 7}, 5: {2, 3, 4, 10, 11, 12}, 6: {9, 10, 11, 12}, 7: {3, 4, 9, 10, 11, 12}, 8: {1, 9}, 9: {2, 6, 7, 8, 10, 11}, 10: {1, 3, 4, 5, 6, 7, 9, 11, 12}, 11: {1, 3, 5, 6, 7, 9, 10}, 12: {2, 5, 6, 7, 10}, }

   Colors Assigned: {1: 3, 2: 2, 3: 4, 4: 1, 5: 3, 6: 2, 7: 2, 8: 0, 9: 3, 10: 0, 11: 1, 12: 1}

7. Generated Graph: {1: {4, 5, 7, 8, 9, 10}, 2: {7, 8, 9, 11, 12}, 3: {4, 5, 6, 8, 9, 10, 12}, 4: {1, 3, 5, 6, 7, 11}, 5: {1, 3, 4, 6, 9, 10}, 6: {8, 3, 4, 5}, 7: {8, 1, 2, 4}, 8: {1, 2, 3, 6, 7, 9, 10}, 9: {1, 2, 3, 5, 8, 10, 12}, 10: {1, 3, 5, 8, 9, 11}, 11: {2, 10, 4}, 12: {9, 2, 3}, }

   Colors Assigned: {1: 0, 2: 0, 3: 0, 4: 1, 5: 3, 6: 2, 7: 2, 8: 1, 9: 2, 10: 4, 11: 2, 12: 1}

8. Generated Graph: {1: {3, 5, 8, 10, 11, 12}, 2: {11, 12, 6}, 3: {1, 4, 5, 8, 9, 10}, 4: {3, 8, 9, 11, 12}, 5: {1, 3, 9, 10, 12}, 6: {9, 2, 11, 12}, 7: {8, 11, 12}, 8: {1, 3, 4, 7, 9}, 9: {3, 4, 5, 6, 8, 11, 12}, 10: {1, 3, 5}, 11: {1, 2, 4, 6, 7, 9, 12}, 12: {1, 2, 4, 5, 6, 7, 9, 11}, }

   Colors Assigned: {1: 1, 2: 1, 3: 0, 4: 3, 5: 2, 6: 3, 7: 1, 8: 2, 9: 1, 10: 3, 11: 2, 12: 0}

9. Generated Graph: {1: {10, 12, 5, 6}, 2: {8, 12, 5}, 3: {10, 11, 5}, 4: {11, 6}, 5: {1, 2, 3, 6, 10, 11}, 6: {1, 4, 5, 8, 10, 12}, 7: {8, 9, 11}, 8: {2, 6, 7, 9, 11}, 9: {8, 7}, 10: {1, 3, 5, 6, 12}, 11: {3, 4, 5, 7, 8}, 12: {1, 2, 10, 6}, }

   Colors Assigned: {1: 3, 2: 1, 3: 3, 4: 0, 5: 0, 6: 1, 7: 2, 8: 0, 9: 1, 10: 2, 11: 1, 12: 0}

10. Generated Graph: {1: {2, 3, 4, 6, 7, 8, 9, 10, 11}, 2: {1, 4, 5, 6, 7, 8, 9}, 3: {1, 4, 5, 6, 9, 10, 11}, 4: {1, 2, 3, 5, 8, 11}, 5: {2, 3, 4, 6, 8, 10}, 6: {1, 2, 3, 5, 7, 8, 9, 11, 12}, 7: {1, 2, 6, 8, 9, 10, 11, 12}, 8: {1, 2, 4, 5, 6, 7, 9, 11}, 9: {1, 2, 3, 6, 7, 8, 10, 11}, 10: {1, 3, 5, 7, 9, 11, 12}, 11: {1, 3, 4, 6, 7, 8, 9, 10, 12}, 12: {10, 11, 6, 7}, }
    Colors Assigned: {1: 0, 2: 2, 3: 3, 4: 1, 5: 0, 6: 1, 7: 3, 8: 4, 9: 5, 10: 1, 11: 2, 12: 0}

11. Generated Graph: {1: {3, 4, 6, 7, 8, 12}, 2: {5, 8, 9, 10, 11, 12}, 3: {1, 4, 7, 8, 10, 11, 12}, 4: {1, 3, 5, 6, 7, 12}, 5: {2, 10, 4, 12}, 6: {1, 10, 4}, 7: {1, 3, 4, 8, 9, 10, 11}, 8: {1, 2, 3, 7, 10}, 9: {2, 10, 7}, 10: {2, 3, 5, 6, 7, 8, 9, 12}, 11: {2, 3, 12, 7}, 12: {1, 2, 3, 4, 5, 10, 11}, }
    Colors Assigned: {1: 0, 2: 1, 3: 1, 4: 3, 5: 4, 6: 1, 7: 2, 8: 3, 9: 3, 10: 0, 11: 0, 12: 2}

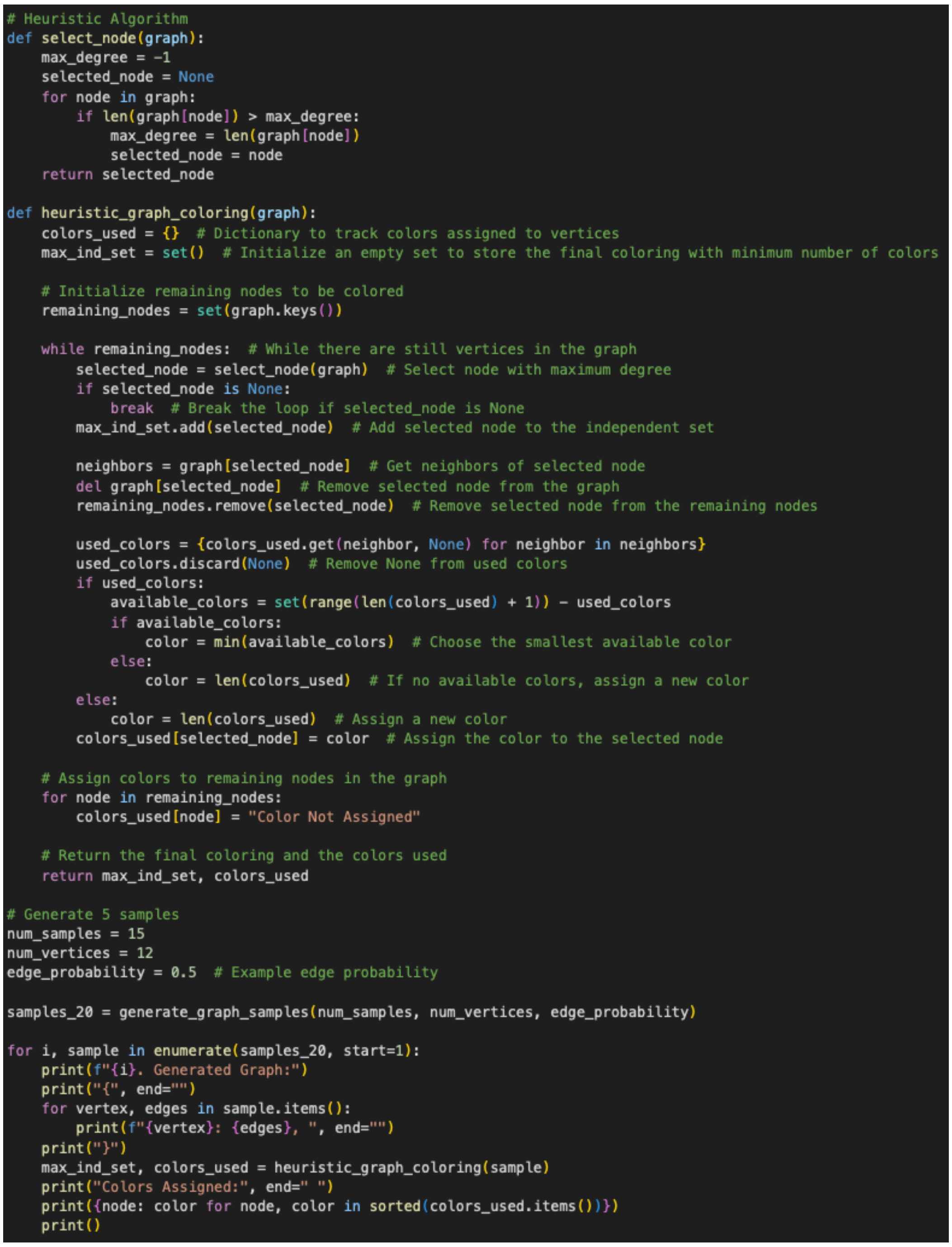
12. Generated Graph: {1: {4, 5, 8, 9, 12}, 2: {4, 5, 8, 9, 11}, 3: {4, 6, 9, 10, 11}, 4: {1, 2, 3, 5, 6, 8, 11}, 5: {1, 2, 4, 6, 7, 8, 9, 11, 12}, 6: {3, 4, 5, 10, 12}, 7: {5}, 8: {1, 2, 4, 5, 10, 11, 12}, 9: {1, 2, 3, 5, 10, 11}, 10: {3, 6, 8, 9, 11, 12}, 11: {2, 3, 4, 5, 8, 9, 10, 12}, 12: {1, 5, 6, 8, 10, 11}, }
    Colors Assigned: {1: 1, 2: 4, 3: 3, 4: 2, 5: 0, 6: 1, 7: 1, 8: 3, 9: 2, 10: 0, 11: 1, 12: 2}

13. Generated Graph: {1: {3, 5, 6, 7, 9}, 2: {9, 3, 6}, 3: {1, 2, 7, 8, 12}, 4: {8, 10, 11, 6}, 5: {1, 6, 7, 8, 10, 11, 12}, 6: {1, 2, 4, 5, 8, 10, 11}, 7: {1, 3, 5, 8, 10, 11}, 8: {3, 4, 5, 6, 7}, 9: {1, 2, 11, 12}, 10: {4, 5, 6, 7, 11}, 11: {4, 5, 6, 7, 9, 10, 12}, 12: {11, 9, 3, 5}, }
    Colors Assigned: {1: 2, 2: 2, 3: 0, 4: 0, 5: 0, 6: 1, 7: 1, 8: 2, 9: 0, 10: 3, 11: 2, 12: 1}

14. Generated Graph: {1: {3, 5, 7, 8, 10}, 2: {9, 11, 5}, 3: {1, 4, 5, 10, 11, 12}, 4: {8, 3, 12}, 5: {1, 2, 3, 7, 8, 9, 10}, 6: {8, 9, 7}, 7: {1, 5, 6, 8, 10, 11}, 8: {1, 4, 5, 6, 7, 11}, 9: {2, 11, 5, 6}, 10: {1, 3, 5, 7, 11, 12}, 11: {2, 3, 7, 8, 9, 10, 12}, 12: {11, 10, 3, 4}, }
    Colors Assigned: {1: 1, 2: 3, 3: 2, 4: 1, 5: 0, 6: 0, 7: 2, 8: 3, 9: 2, 10: 3, 11: 1, 12: 0}

15. Generated Graph: {1: {3, 4, 6, 8, 9, 11}, 2: {3, 5, 6, 8, 10, 11}, 3: {1, 2, 4, 9, 11}, 4: {1, 3, 6, 7, 8, 9, 12}, 5: {2, 7, 8, 9, 10, 11}, 6: {8, 1, 2, 4}, 7: {8, 10, 4, 5}, 8: {1, 2, 4, 5, 6, 7, 12}, 9: {1, 3, 4, 5}, 10: {2, 5, 7, 11, 12}, 11: {1, 2, 3, 5, 10}, 12: {8, 10, 4}, }

Colors Assigned: {1: 2, 2: 0, 3: 1, 4: 0, 5: 2, 6: 3, 7: 3, 8: 1, 9: 3, 10: 1, 11: 3, 12: 2}



```python
# Heuristic Algorithm
def select_node(graph):
    max_degree = -1
    selected_node = None
    for node in graph:
        if len(graph[node]) > max_degree:
            max_degree = len(graph[node])
            selected_node = node
    return selected_node

def heuristic_graph_coloring(graph):
    colors_used = {}  # Dictionary to track colors assigned to vertices
    max_ind_set = set()  # Initialize an empty set to store the final coloring with minimum number of colors

    # Initialize remaining nodes to be colored
    remaining_nodes = set(graph.keys())

    while remaining_nodes:  # While there are still vertices in the graph
        selected_node = select_node(graph)  # Select node with maximum degree
        if selected_node is None:
            break  # Break the loop if selected_node is None
        max_ind_set.add(selected_node)  # Add selected node to the independent set

        neighbors = graph[selected_node]  # Get neighbors of selected node
        del graph[selected_node]  # Remove selected node from the graph
        remaining_nodes.remove(selected_node)  # Remove selected node from the remaining nodes

        used_colors = {colors_used.get(neighbor, None) for neighbor in neighbors}
        used_colors.discard(None)  # Remove None from used colors
        if used_colors:
            available_colors = set(range(len(colors_used) + 1)) - used_colors
            if available_colors:
                color = min(available_colors)  # Choose the smallest available color
            else:
                color = len(colors_used)  # If no available colors, assign a new color
        else:
            color = len(colors_used)  # Assign a new color
        colors_used[selected_node] = color  # Assign the color to the selected node

    # Assign colors to remaining nodes in the graph
    for node in remaining_nodes:
        colors_used[node] = "Color Not Assigned"

    # Return the final coloring and the colors used
    return max_ind_set, colors_used

# Generate 5 samples
num_samples = 15
num_vertices = 12
edge_probability = 0.5  # Example edge probability

samples_20 = generate_graph_samples(num_samples, num_vertices, edge_probability)

for i, sample in enumerate(samples_20, start=1):
    print(f"{i}. Generated Graph:")
    print("{", end="")
    for vertex, edges in sample.items():
        print(f"{vertex}: {edges}, ", end="")
    print("}")
    max_ind_set, colors_used = heuristic_graph_coloring(sample)
    print("Colors Assigned:", end=" ")
    print({node: color for node, color in sorted(colors_used.items())})
    print()
```

Figure 7: Implementation of Heuristic Algorithm

6. Experimental Analysis of The Performance (Performance Testing)

This section of the study will provide an experimental examination of the algorithm's performance. The performance study carried out in section 3.2 is an upper bound on the algorithm's time and space complexity; it does not specify the performance in real-life applications. Now, we will provide the experimental results regarding the heuristic algorithm's time complexity. In order to do this, we will obtain k randomly generated distinct graphs for each input size, k = 200 in this example, as can be seen in figure 9. This will guarantee that the mean we obtain is representative of the population. Through empirical study, we are assessing the effectiveness of a heuristic approach for graph coloring.

First, we create randomly distributed, undirected graph with a given edge probability that range in size from 50 to 200 vertices. As can be seen in figure 8, equation for the fitted line is $y = 1.755682 * x + -14.530963$. The slope of the line is 1.755682, indicating that the average running time grows at a rate proportional to the number of vertices (x). In this case, the y-intercept of the line, which is -14.530963, is not practically significant because it reflects the baseline average running time when there are no vertices. Consider $T(n)=n^a +$ (lower order terms) as the theoretical time complexity function, where n represents the number of vertices and a is the slope in the log-log plot. Slop of the line can give insights into the algorithm's practical time complexity. Since a is 1.755682, the algorithm's time complexity is $O(n^{1.755682})$. As a result, in practice, algorithm performs better than the worst-case running time of $O(n \cdot \Delta)$ that is stated in section 3.2.

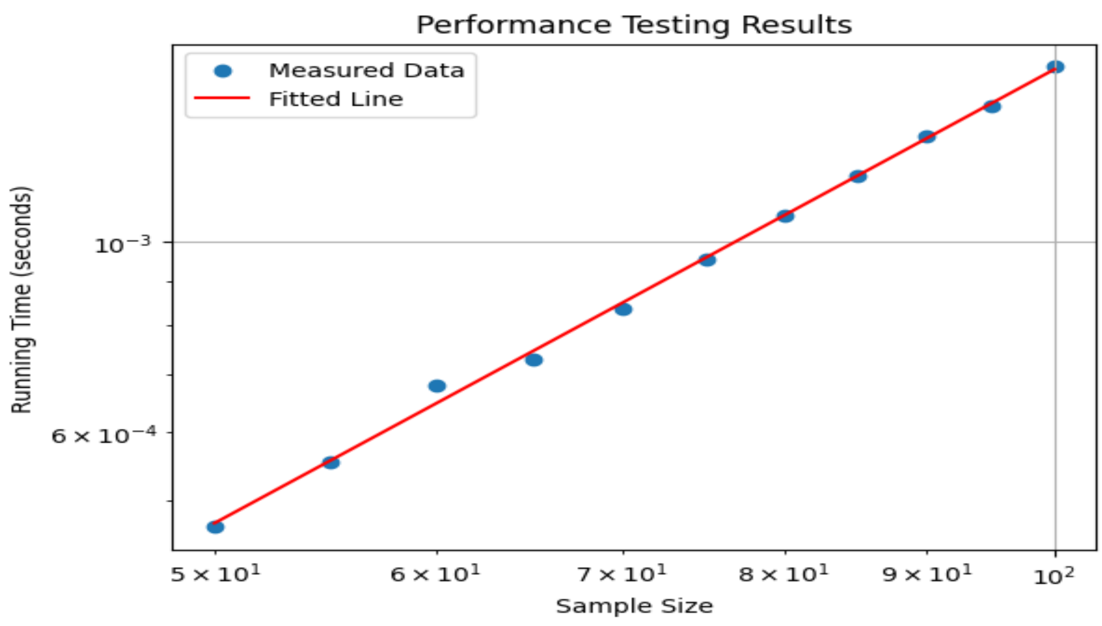
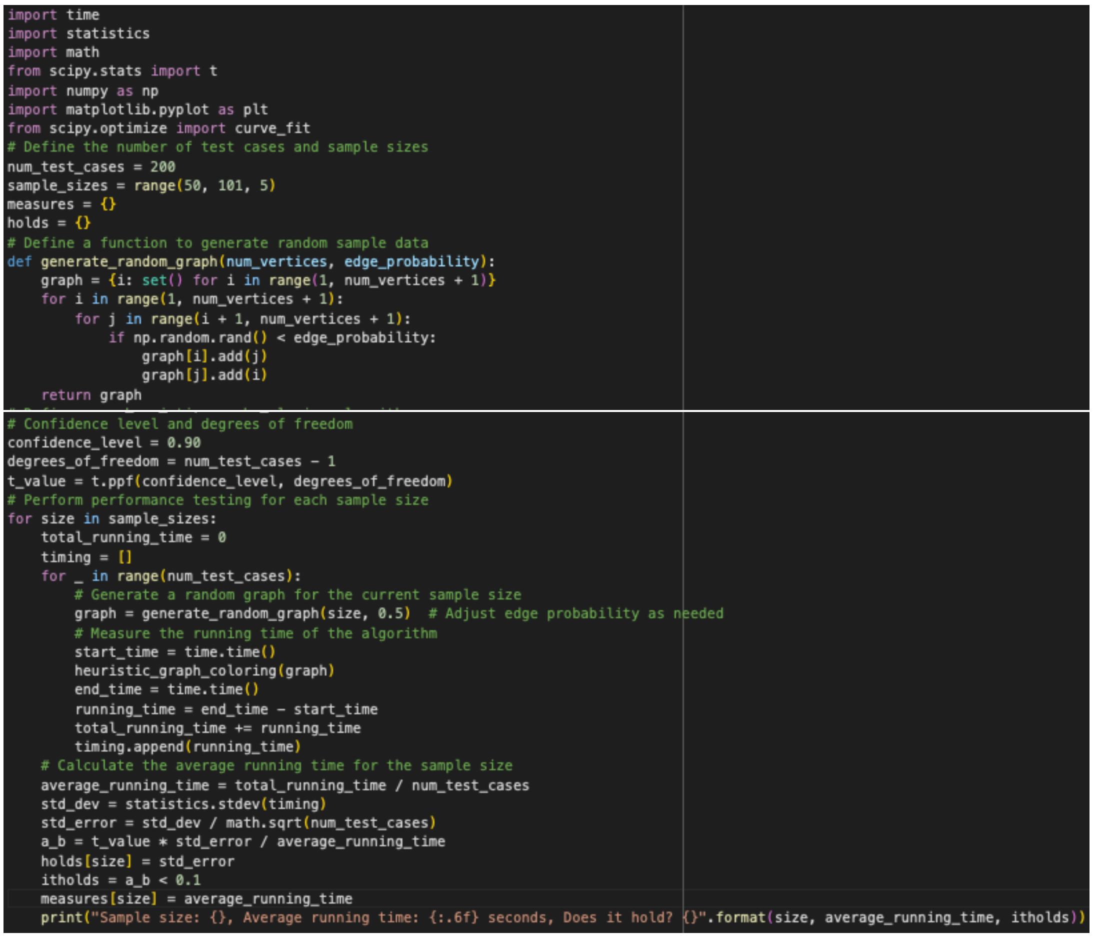


Figure 8: Performance Testing Result



```
import time
import statistics
import math
from scipy.stats import t
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
# Define the number of test cases and sample sizes
num_test_cases = 200
sample_sizes = range(50, 101, 5)
measures = {}
holds = {}
# Define a function to generate random sample data
def generate_random_graph(num_vertices, edge_probability):
    graph = {i: set() for i in range(1, num_vertices + 1)}
    for i in range(1, num_vertices + 1):
        for j in range(i + 1, num_vertices + 1):
            if np.random.rand() < edge_probability:
                graph[i].add(j)
                graph[j].add(i)
    return graph
```

```
# Confidence level and degrees of freedom
confidence_level = 0.90
degrees_of_freedom = num_test_cases - 1
t_value = t.ppf(confidence_level, degrees_of_freedom)
# Perform performance testing for each sample size
for size in sample_sizes:
    total_running_time = 0
    timing = []
    for _ in range(num_test_cases):
        # Generate a random graph for the current sample size
        graph = generate_random_graph(size, 0.5)  # Adjust edge probability as needed
        # Measure the running time of the algorithm
        start_time = time.time()
        heuristic_graph_coloring(graph)
        end_time = time.time()
        running_time = end_time - start_time
        total_running_time += running_time
        timing.append(running_time)
    # Calculate the average running time for the sample size
    average_running_time = total_running_time / num_test_cases
    std_dev = statistics.stdev(timing)
    std_error = std_dev / math.sqrt(num_test_cases)
    a_b = t_value * std_error / average_running_time
    holds[size] = std_error
    itholds = a_b < 0.1
    measures[size] = average_running_time
    print("Sample size: {}, Average running time: {:.6f} seconds, Does it hold? {}".format(size, average_running_time, itholds))
```

Figure 9: Implementation of the Algorithm

7. Experimental Analysis of The Quality

INPUT SIZE	NUMBER OF COLORS NEEDED FOR THE BRUTE FORCE	NUMBER OF COLORS NEEDED FOR THE HEURISTIC	RATIO OF HEURISTIC/BRUTE FORCE
1	1	1	1.0
2	1	2	2.0
3	2	3	1.5
4	2	3	1.5
5	2	3	1.5
6	2	4	2.0
7	3	3	1.0
8	3	3	1.0
9	3	4	1.33
10	3	3	1.0
11	4	4	1.0
12	4	4	1.0
13	4	4	1.0
14	4	4	1.0
15	3	5	1.66

Table 1

Table 1 displays the results of a comparison of heuristic and brute force algorithms. As we suspected, the heuristic approach does not always produce the correct solution. Correct means whether it gives the minimum number of colors required to color the graph (the graph's chromatic number). The quality of the approaches also gets measure with the number of colors they use to color the graphs. Minimum number of colors means better quality. Table 1 shows the outcomes of providing random graphs to both algorithms, with input size, chromatic number, and ratio of results.

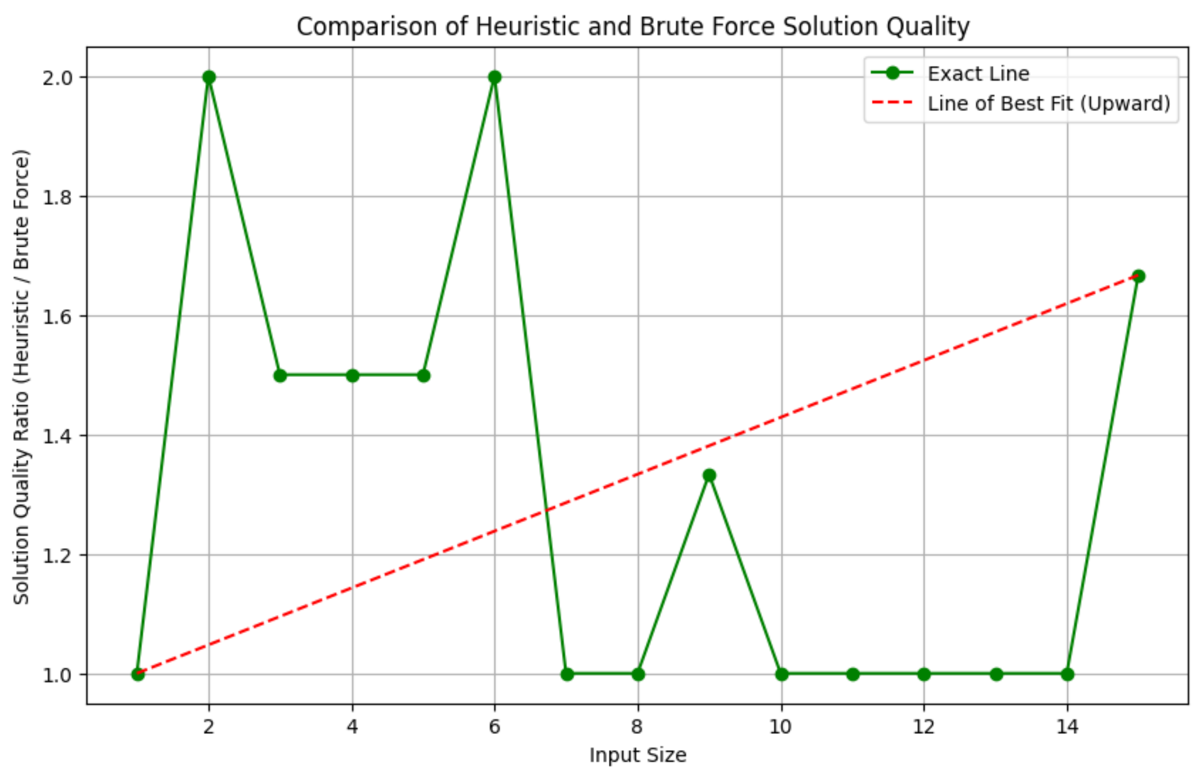


Figure 10: Results of the experiments for the size

The graph at Figure 10 shows the relationship between input size and the heuristic solution quality versus brute force solution quality. The red line (line of best fit) summarizes the statistical trend in the data. It is useful to analyze the overall behavior of the heuristic solution's quality in comparison to the brute force solution as the input size increases, smoothing out particular deviations. The line of best fit indicates a general upward trend in the ratio as input size grows. This suggests that when the problem grows larger, the heuristic approach tends to produce a lower quality (higher number of colors needed to color the graph) than the brute force. The increasing trend in the ratio may be useful in cases where the brute force approach is preferred over heuristic due to constraints such as time or computational power, particularly as input size increases. The graph shows that the heuristic does not maintains its performance relative to the brute force solution, but actually get worse as the problem's complexity increases. The accuracy of the heuristic algorithm depends on the shape of the graph as well as the size of it. Additionally, providing graphs with only one edge to heuristic and brute force algorithms leads to higher error rates. As only one edge is needed to color a graph, Figure 11 shows the results of both techniques.

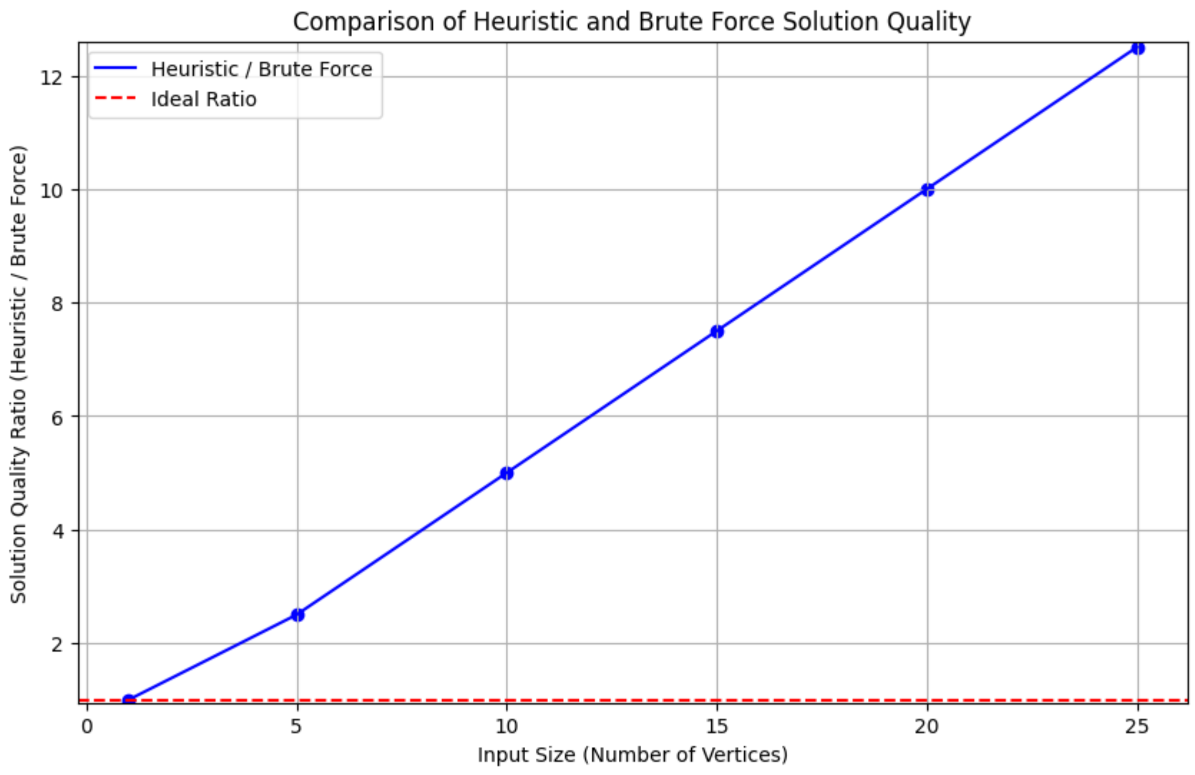


Figure 11:Results of the experiments for the shape and size

The graph at Figure 11 shows the ratio of heuristic to brute force solution quality as a function of the input size, specifically the number of vertices in the graph, under the constraint that all graphs have one edge. The blue line indicates that the ratio of the heuristic solution quality to the brute force solution quality decreases linearly with the number of vertices. This suggests that as the number of vertices increases, the heuristic solution's quality reduces (higher number of colors needed to color the graph) more significantly compared to the brute force solution. Essentially, the error rate is increasing. This could imply that the brute force algorithm is more scalable or more efficiently handles larger graphs with minimal connectivity (since there's only one edge). Essentially, the heuristic is less efficient or effective at solving problems with more vertices, compared to the brute force method at graphs with minimal connectivity. The red dashed line represents an "ideal" constant ratio that the heuristic might aim to achieve or maintain under typical conditions. The constancy of this line suggests an expectation that a well-performing heuristic should not deviate much from the ratio of one meaning that it should produce similar results to the brute force algorithm regardless of the graph size at the graphs with minimal connectivity. The input graph significantly impacts the algorithm's error rate, as demonstrated. We tested multiple input sizes and multiple placements of the one and only edge to reduce randomness and ensure representativeness the entire set. In the end, it's critical to take into account the size and shape of the input graph when utilizing heuristic algorithms for practical applications, and to provide additional details to guarantee the applicability of the heuristic method.

8. Experimental Analysis of the Correctness of the Implementation (Functional Testing)

Black Box Testing for the Heuristic Graph Coloring Algorithm

Black box testing is a software testing method that evaluates the functionality of an application without peering into its internal structures or workings. In this chapter, we apply black box testing to the heuristic graph coloring algorithm we proposed. The goal is to ensure that the algorithm assigns colors to the graph such that no two adjacent vertices share the same color. Here, we outline 9 test cases along with their expected outputs and explanations for each scenario.

Test Case 1: Empty Graph

Description: This test case assesses the algorithm's response to an empty graph.

Graph: {}

Expected Output: {}

Explanation: With no vertices to color, the algorithm should return an empty color mapping, indicating correct handling of edge cases.

Test Case 2: Single Vertex

Description: This test case evaluates the algorithm's behavior with a graph containing only one vertex and no edges.

Graph: {1: set()}

Expected Output: {1: 0}

Explanation: A single vertex with no edges should be assigned the first available color, typically 0, demonstrating the algorithm's basic functionality.

Test Case 3: Linear Graph

Description: This test case verifies the algorithm on a linear arrangement of vertices.

Graph: {1: {2}, 2: {1, 3}, 3: {2, 4}, 4: {3}}

Expected Output: Colors alternating between two values

Explanation: In a path graph, a two-color alternation is sufficient as no three consecutive vertices need more than two colors, validating the algorithm's efficiency in simple structures.

Test Case 4: Complete Graph

Description: This test evaluates how the algorithm handles a complete graph, where every vertex is connected to every other.

Graph: {i: set(range(1, 6)) - {i} for i in range(1, 6)}

Expected Output: Each vertex has a unique color

Explanation: A complete graph requires each vertex to have a distinct color. The test confirms that the algorithm correctly assigns a unique color to each vertex.

Test Case 5: Cycle Graph (Even)

Description: This case checks the algorithm's performance on a cycle graph with an even number of vertices.

Graph: {1: {2, 6}, 2: {1, 3}, 3: {2, 4}, 4: {3, 5}, 5: {4, 6}, 6: {5, 1}}

Expected Output: Two colors used

Explanation: An even cycle can be colored with two colors, which the test verifies by ensuring no adjacent vertices share a color.

Test Case 6: Cycle Graph (Odd)

Description: Tests the algorithm's effectiveness on a cycle graph with an odd number of vertices.

Graph: {1: {2, 5}, 2: {1, 3}, 3: {2, 4}, 4: {3, 5}, 5: {4, 1}}

Expected Output: Three colors used

Explanation: An odd cycle requires three colors to ensure adjacent vertices do not share the same color, testing the algorithm's adaptability to more complex requirements.

Test Case 7: Star Graph

Description: This case examines the coloring of a star graph.

Graph: {1: set(range(2, 7)), **{i: {1} for i in range(2, 7)}}

Expected Output: Two colors used

Explanation: The central vertex requires a unique color, while all outer vertices can share another, confirming the algorithm's strategic color assignment in star topologies.

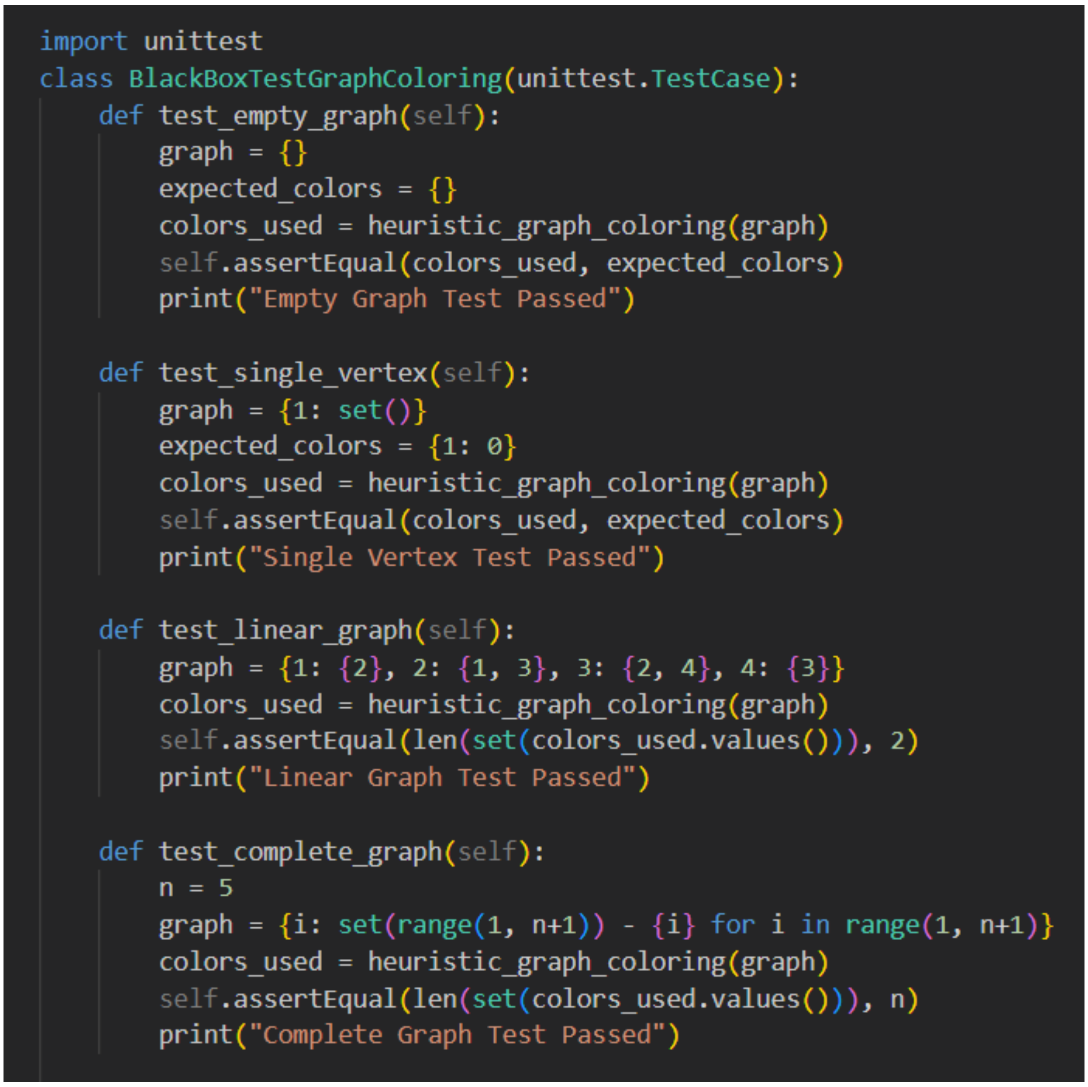
Test Case 8: Dense Graph

Description: Tests how the algorithm manages a graph with a high density of edges.

Graph: Graph generated with high edge density

Expected Output: High color usage

Explanation: Dense graphs might need many colors, approaching the number of vertices. This case ensures that the algorithm scales correctly with increasing complexity.



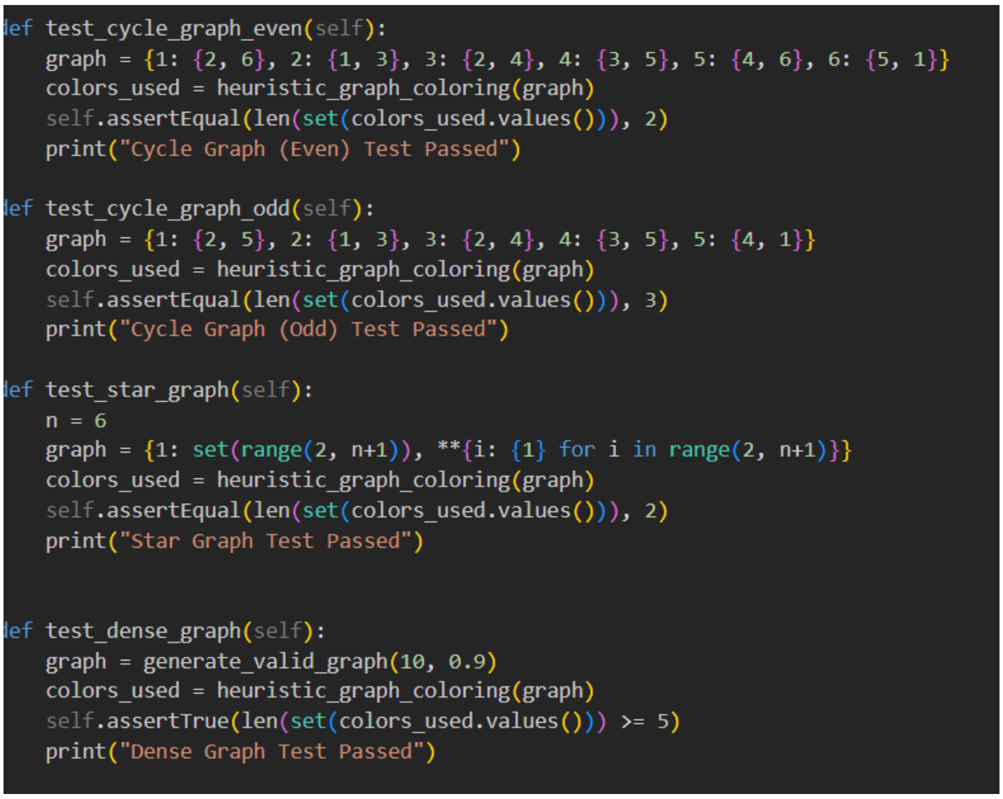
```python
import unittest
class BlackBoxTestGraphColoring(unittest.TestCase):
    def test_empty_graph(self):
        graph = {}
        expected_colors = {}
        colors_used = heuristic_graph_coloring(graph)
        self.assertEqual(colors_used, expected_colors)
        print("Empty Graph Test Passed")

    def test_single_vertex(self):
        graph = {1: set()}
        expected_colors = {1: 0}
        colors_used = heuristic_graph_coloring(graph)
        self.assertEqual(colors_used, expected_colors)
        print("Single Vertex Test Passed")

    def test_linear_graph(self):
        graph = {1: {2}, 2: {1, 3}, 3: {2, 4}, 4: {3}}
        colors_used = heuristic_graph_coloring(graph)
        self.assertEqual(len(set(colors_used.values())), 2)
        print("Linear Graph Test Passed")

    def test_complete_graph(self):
        n = 5
        graph = {i: set(range(1, n+1)) - {i} for i in range(1, n+1)}
        colors_used = heuristic_graph_coloring(graph)
        self.assertEqual(len(set(colors_used.values())), n)
        print("Complete Graph Test Passed")
```

Figure 12: Block Box Tests



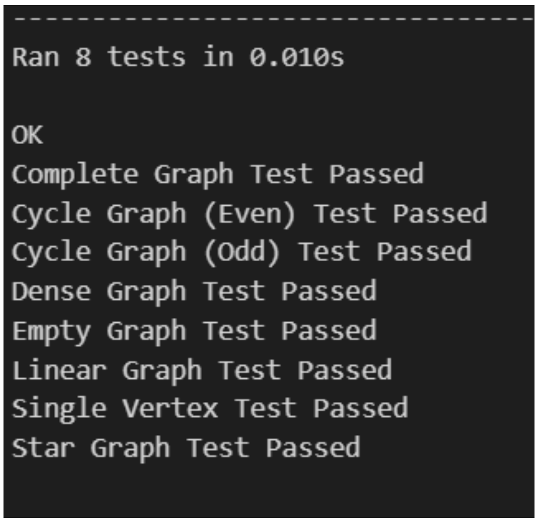
```
lef test_cycle_graph_even(self):
    graph = {1: {2, 6}, 2: {1, 3}, 3: {2, 4}, 4: {3, 5}, 5: {4, 6}, 6: {5, 1}}
    colors_used = heuristic_graph_coloring(graph)
    self.assertEqual(len(set(colors_used.values())), 2)
    print("Cycle Graph (Even) Test Passed")

lef test_cycle_graph_odd(self):
    graph = {1: {2, 5}, 2: {1, 3}, 3: {2, 4}, 4: {3, 5}, 5: {4, 1}}
    colors_used = heuristic_graph_coloring(graph)
    self.assertEqual(len(set(colors_used.values())), 3)
    print("Cycle Graph (Odd) Test Passed")

lef test_star_graph(self):
    n = 6
    graph = {1: set(range(2, n+1)), **{i: {1} for i in range(2, n+1)}}
    colors_used = heuristic_graph_coloring(graph)
    self.assertEqual(len(set(colors_used.values())), 2)
    print("Star Graph Test Passed")


lef test_dense_graph(self):
    graph = generate_valid_graph(10, 0.9)
    colors_used = heuristic_graph_coloring(graph)
    self.assertTrue(len(set(colors_used.values())) >= 5)
    print("Dense Graph Test Passed")
```

Figure 13: Block Box Tests



```
----------------------------------
Ran 8 tests in 0.010s

OK
Complete Graph Test Passed
Cycle Graph (Even) Test Passed
Cycle Graph (Odd) Test Passed
Dense Graph Test Passed
Empty Graph Test Passed
Linear Graph Test Passed
Single Vertex Test Passed
Star Graph Test Passed
```

Figure 14 :Block Box Test Results

Conclusion for the Black-box Testing

As depicted in the implementation results, all test cases are successfully passed by the algorithm, underscoring its robustness and effectiveness. The tests span a range of graph types, from simple to complex, ensuring comprehensive coverage of potential real-world scenarios. Black box testing for the heuristic graph coloring algorithm confirms its functionality and adherence to theoretical expectations. By addressing diverse cases, including empty graphs, and those with varying vertex and edge configurations, we have validated the algorithm's capability to perform graph coloring efficiently and accurately. The positive outcomes across all test cases provide strong evidence of the algorithm's reliability and operational readiness.

White Box Testing for the Heuristic Graph Coloring Algorithm

White box testing, or clear box testing, involves examining the internal structure and logic of an algorithm. In this section, we delve into the white box testing of our heuristic graph coloring algorithm, focusing on statement coverage, decision coverage, and path coverage.

Statement Coverage:

Our testing ensures that every line of code in the graph coloring function is executed at least once. The tests cover various graph configurations:

1. ***Test for Maximum Degree Node Selection:*** This ensures the function correctly identifies the node with the highest connectivity, thus testing statements within the select_node function.
2. ***Test for Coloring Isolated Nodes:*** By inputting graphs with isolated nodes, we verify the initialization and execution of the algorithm's loops and condition checks, ensuring that even nodes without edges are assigned colors if applicable.
3. ***Test for Complete Graphs:*** These tests cover the algorithm's ability to adapt the coloring based on the graph's density, ensuring that different branches and loops in the code are exercised.

Decision Coverage:

This aspect of white box testing requires that every decision in the code (e.g., if-else statements) is executed in both directions:

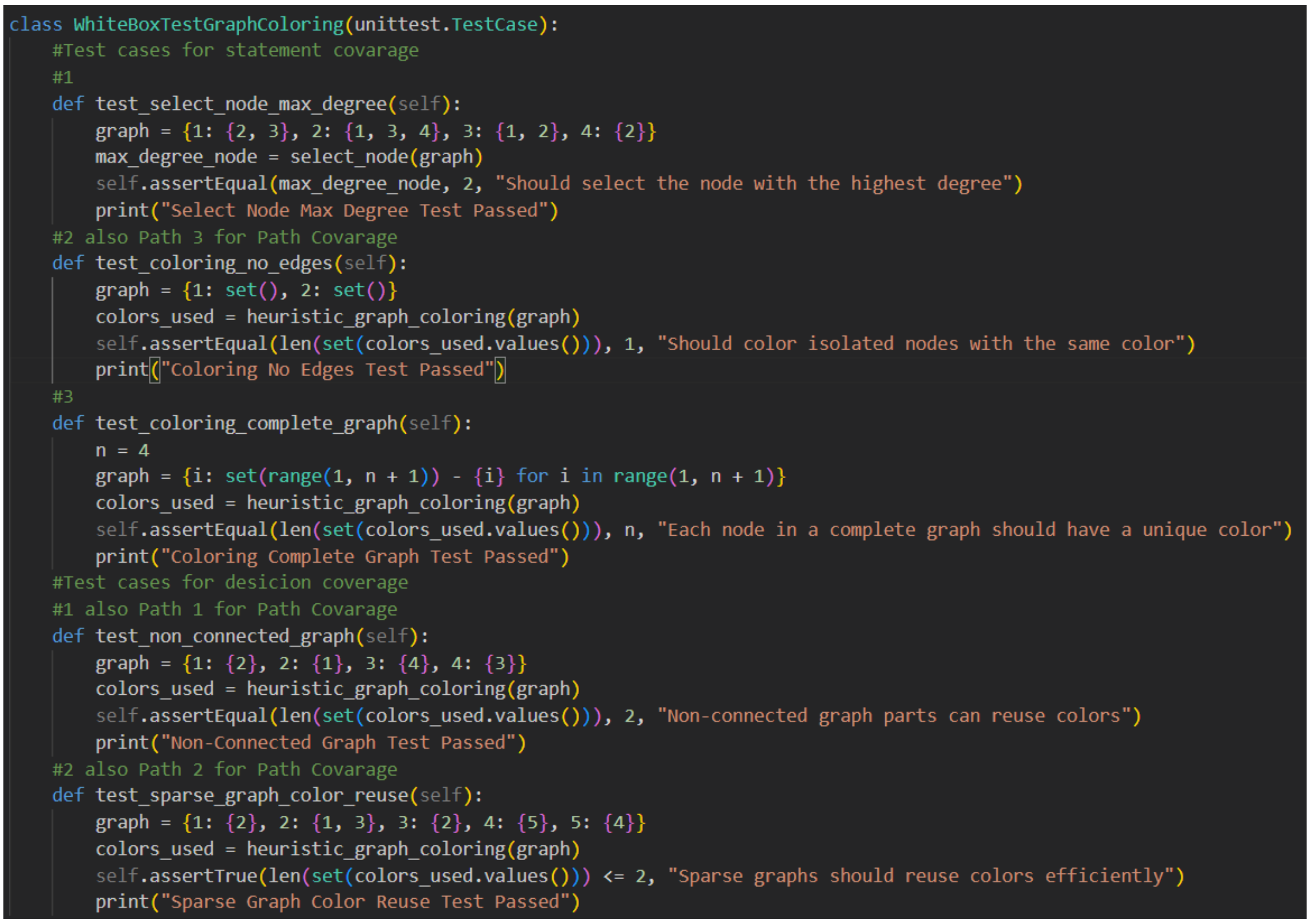
1. ***Non-Connected Graph Test***: It checks whether the algorithm can efficiently reuse colors in different components of the graph, testing decision points related to color assignment and reuse.

2. ***Sparse Graph Color Reuse Test:*** Specifically targets the algorithm's logic in efficiently using the minimal number of colors, ensuring that the decision to assign new colors or reuse existing ones is correctly managed.

Path Coverage:

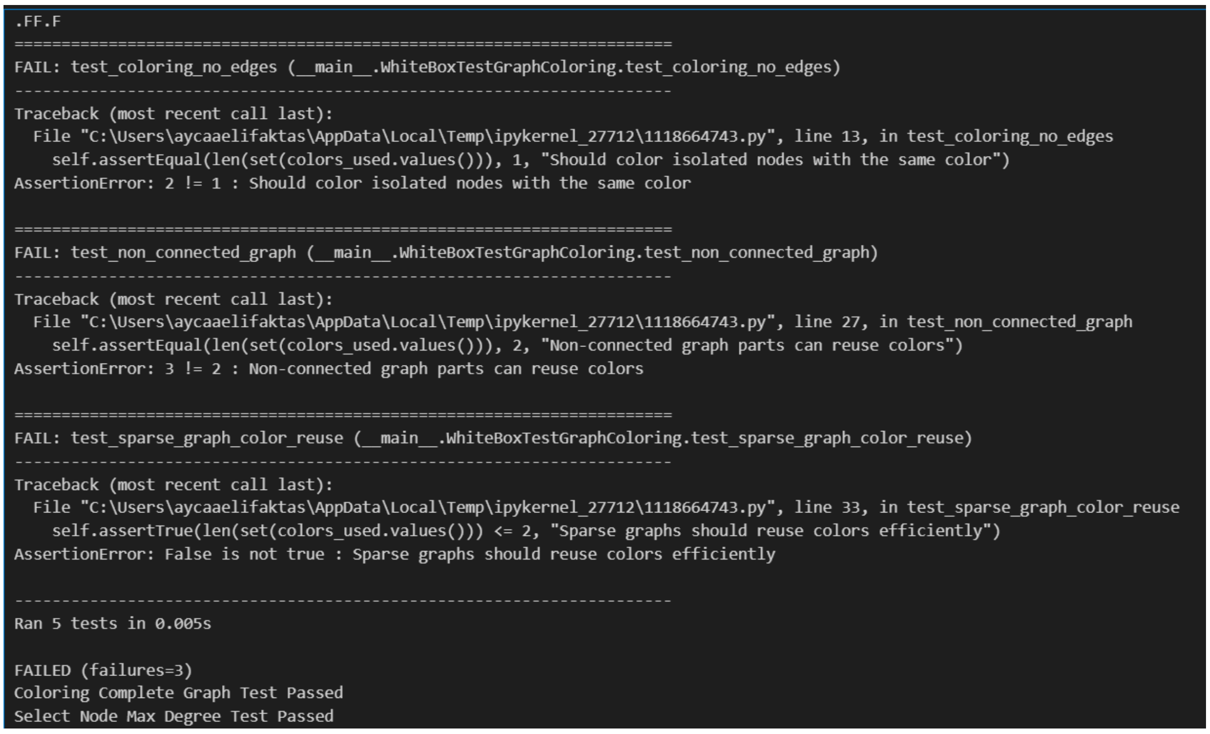
Path coverage ensures that all possible routes through the code are tested, which is crucial for a heuristic algorithm where different graph configurations can lead to different execution paths:

1. ***Path Involving Non-connected Components***: Validates that the algorithm handles separate graph components correctly.
2. ***Path for Sparse Graphs***: Ensures the logic to minimize color usage is triggered.
3. ***Path with No Edges:*** Checks the algorithm's response to graphs where nodes are entirely isolated.



```python
class WhiteBoxTestGraphColoring(unittest.TestCase):
    #Test cases for statement covarage
    #1
    def test_select_node_max_degree(self):
        graph = {1: {2, 3}, 2: {1, 3, 4}, 3: {1, 2}, 4: {2}}
        max_degree_node = select_node(graph)
        self.assertEqual(max_degree_node, 2, "Should select the node with the highest degree")
        print("Select Node Max Degree Test Passed")
    #2 also Path 3 for Path Covarage
    def test_coloring_no_edges(self):
        graph = {1: set(), 2: set()}
        colors_used = heuristic_graph_coloring(graph)
        self.assertEqual(len(set(colors_used.values())), 1, "Should color isolated nodes with the same color")
        print("Coloring No Edges Test Passed")
    #3
    def test_coloring_complete_graph(self):
        n = 4
        graph = {i: set(range(1, n + 1)) - {i} for i in range(1, n + 1)}
        colors_used = heuristic_graph_coloring(graph)
        self.assertEqual(len(set(colors_used.values())), n, "Each node in a complete graph should have a unique color")
        print("Coloring Complete Graph Test Passed")
    #Test cases for desicion coverage
    #1 also Path 1 for Path Covarage
    def test_non_connected_graph(self):
        graph = {1: {2}, 2: {1}, 3: {4}, 4: {3}}
        colors_used = heuristic_graph_coloring(graph)
        self.assertEqual(len(set(colors_used.values())), 2, "Non-connected graph parts can reuse colors")
        print("Non-Connected Graph Test Passed")
    #2 also Path 2 for Path Covarage
    def test_sparse_graph_color_reuse(self):
        graph = {1: {2}, 2: {1, 3}, 3: {2}, 4: {5}, 5: {4}}
        colors_used = heuristic_graph_coloring(graph)
        self.assertTrue(len(set(colors_used.values())) <= 2, "Sparse graphs should reuse colors efficiently")
        print("Sparse Graph Color Reuse Test Passed")
```

Figure 15: White Box Tests



```
.FF.F
======================================================================
FAIL: test_coloring_no_edges (__main__.WhiteBoxTestGraphColoring.test_coloring_no_edges)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\aycaaelifaktas\AppData\Local\Temp\ipykernel_27712\1118664743.py", line 13, in test_coloring_no_edges
    self.assertEqual(len(set(colors_used.values())), 1, "Should color isolated nodes with the same color")
AssertionError: 2 != 1 : Should color isolated nodes with the same color

======================================================================
FAIL: test_non_connected_graph (__main__.WhiteBoxTestGraphColoring.test_non_connected_graph)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\aycaaelifaktas\AppData\Local\Temp\ipykernel_27712\1118664743.py", line 27, in test_non_connected_graph
    self.assertEqual(len(set(colors_used.values())), 2, "Non-connected graph parts can reuse colors")
AssertionError: 3 != 2 : Non-connected graph parts can reuse colors

======================================================================
FAIL: test_sparse_graph_color_reuse (__main__.WhiteBoxTestGraphColoring.test_sparse_graph_color_reuse)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\aycaaelifaktas\AppData\Local\Temp\ipykernel_27712\1118664743.py", line 33, in test_sparse_graph_color_reuse
    self.assertTrue(len(set(colors_used.values())) <= 2, "Sparse graphs should reuse colors efficiently")
AssertionError: False is not true : Sparse graphs should reuse colors efficiently

----------------------------------------------------------------------
Ran 5 tests in 0.005s

FAILED (failures=3)
Coloring Complete Graph Test Passed
Select Node Max Degree Test Passed
```

Figure 16: : White Box Test Results

Conclusion for the White-box Testing

The white box testing of the heuristic graph coloring algorithm provided crucial insights into its internal logic and functionality. The tests covered various aspects of the algorithm, including the selection of nodes with the highest degree, color assignment for different graph structures, and the handling of edge cases. The results of the tests highlighted both strengths and areas needing improvement. While the heuristic graph coloring algorithm demonstrated strong performance in certain scenarios, such as selecting the maximum degree node and coloring complete graphs, the white box testing revealed significant areas for improvement. Specifically, the algorithm needs refinement in handling isolated nodes, non-connected graphs, and sparse graphs to optimize color usage and efficiency. Addressing these issues will enhance the overall robustness and applicability of the algorithm across a broader range of graph structures.

9. Discussion

We'll talk about the findings from each part at the conclusion of the report. First and foremost, even though the brute force method always yields the right answer, it is extremely slow and unsuitable for solving real-world issues since it grows exponentially with time. As previously mentioned, the brute force algorithm's running time is $O(n^n \times (m + n))$, where m and n are the numbers of edges and vertices, respectively. Furthermore, we have demonstrated that this is an NP-complete issue. Additionally, we have presented practical applications of the graph coloring problem, which further underscore the topic's significance. The heuristic algorithm has a temporal complexity of $O(n \cdot \Delta)$ for n vertices and Δ represents the highest degree, significantly less than the brute force algorithm. As a result, the algorithm's applicability to real-world problems improves. However, there is also a difficulty with the heuristic algorithm. As we have shown, the method loses correctness as the input increases. The minimum number of colors required to color a graph is computed with some error, which may increase with larger input sizes. The algorithm also considers the graph's shape and edges at each iteration. When applying these findings to real-world problems, it's important to keep these considerations in mind. Additionally, when dealing with huge inputs, error rates should be factored in. Despite multiple attempts, the heuristic algorithm's predicted values do not significantly depart from the real values, with a ratio of no more than 1.6 in the tested instance sizes (see to Chapter 7 Tabel 1). The mistake rate is acceptable for graph coloring applications, as a specific number of colors can be used to get a close-to-minimum color scheme while saving time. Furthermore, we have obtained results from the performance of the heuristic algorithm. First of all, we should also note that the algorithm did encounter some error or defect in the implementation testing part. All of the tests that are applied black-box testings, passed successfully by the algorithm but some of the white box tests failed. As for the performance measures, experimental results (see to Chapter 6) showed that the average running of the algorithm's time complexity is $O(n^{1.755682})$. As a result, in practice, algorithm performs better than the worst-case running time of $O(n \cdot \Delta)$ that is stated in section 3.2. In practice, the algorithm's running time is less than its worst-case scenario. This was to be expected, given that the algorithm's worst-case running time occurs so rarely. The heuristic technique clearly outperforms the brute force algorithm in terms of time performance. However, when input sizes increase, we lose some accuracy while saving time.

We demonstrated this loss in Chapter 7 by comparing the precise accurate findings. Overall, the heuristic graph coloring algorithm demonstrated strong performance in many scenarios, confirming its potential applicability in real-world graph coloring problems. However, the tests revealed specific areas needing improvement, particularly in handling isolated nodes, non-connected graphs, and sparse graphs. These findings suggest that while the algorithm is effective, further refinement and optimization are necessary to enhance its robustness and efficiency across all graph types.

References

ALMARA’BEH, Hilal, and Amjad SULEIMAN. *Heuristic Algorithm for Graph Coloring Based On Maximum Independent Set*. Stefan cel Mare University of Suceava.

Aslan, Murat, and Nurdan Akhan Baykan. “A Performance Comparison of Graph Coloring Algorithms”. International Journal of Intelligent Systems and Applications in Engineering 4, no. Special Issue-1 (December 2016): 1-7. https://doi.org/10.18201/ijisae.273053.

FormanowiczPiotr and TanaśKrzysztof. "A survey of graph coloring - its types, methods and applications" *Foundations of Computing and Decision Sciences* 37, no.3 (2012): 223-238. https://doi.org/10.2478/v10209-011-0012-y

Garey, M. R. and Johnson, D. S.. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. First Edition : W. H. Freeman, 1979.