CS301

2023-2024 Spring

Project Report

Group 172

Group Members

Tan Ufuk Çelik - 28285

Selenay Buse Batıbay – 30294

1. Problem Description

Name:	***Graph Coloring***
Input:	***An undirected graph G(V,E) with n nodes and node set V, and edge set E; a positive integer k.***
Question:	***What is the minimum number of colors (k) required to properly color the vertices of an undirected graph G = (V, E) such that no adjacent vertices share the same color?***

1.1 Overview

Finding colors for a graph's vertices so that no two neighboring vertices have the same color is known as the "Graph Coloring" problem. Its goal, to put it simply, is to color the vertices of a network with a maximum of k distinct colors so that the vertices u and u have different colors for each edge (u, V) in E.

Formally, the challenge is to discover a coloring for an undirected graph G = (V, E) given a number of colors k. The goal is to color each edge (u, V) in E so that the vertices u and v have a different color.

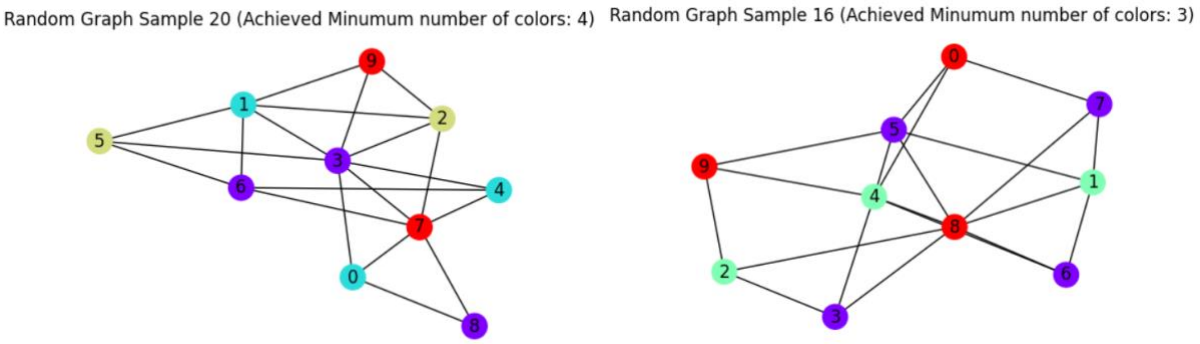
1.2 Decision Problem

What is the minimum number of colors (k) required to achieve a proper vertex coloring in an undirected graph G = (V, E), ensuring that no two adjacent vertices share the same color?

1.3 Optimization Problem

Reduce the number of colors required to color V in an undirected graph G = (V, E) so that no two neighboring vertices in E have the same color applied to them.

1.4 Example Illustration



These graphs with 10 vertices are colored with the minimum number of possible coloring. More illustrations can be found at the end of this document.

1.5 Real World Applications

Timetabling and Scheduling: A graph coloring issue can be used to mimic the process of allocating exam times so that no student takes two tests at the same time.

Register Allocation in Compilers: It's similar to coloring nodes, where each color denotes a separate register, to assign variables to a restricted number of registers in a CPU.

Frequency Assignment: A graph coloring problem is to assign radio frequencies to transmitters so that no two transmitters in proximity interfere with one another.

Map coloring: An example of a planar graph coloring problem is the task of coloring political maps such that no two neighboring regions have the same color.

1.6 Hardness of the Problem

Graph Coloring is a well-known NP-complete problem which Karp demonstrated in 1972. This indicates that there is unlikely to be a polynomial-time method that can solve it and that none currently exists.

2. Algorithm Description

2.1 Brute Force Algorithm

2.1.1 Overview

This brute-force graph coloring algorithm explores a vast search space of all possible colorings of the graph. The time complexity of **brute_force_coloring** algorithm is O(m^V). The necessity to thoroughly search through all possible color combinations for each vertex, taking into account every circumstance until an acceptable coloring is identified, is the source of this exponential time complexity. Although the technique finds the least amount of colors needed for good graph coloring, it is not very efficient when the graph size increases.

The approach is not feasible for huge graphs due to its unacceptably high computing cost. The search space rises exponentially as the number of vertices and edges increases, resulting in a large processing overhead. As a result, large or complex graphs may not lend themselves well to the brute-force method.

2.1.2 Pseudocode

```
# Brute-Force Graph Coloring Algorithm
# Input: A graph G with V vertices and E edges
# Output: Minimum number of colors needed and a valid coloring

function generate_random_graph(num_vertices, density):
    # Generate a random graph with approximately 2n edges
    num_edges = int(num_vertices * (num_vertices - 1) * density / 2)
    return a random graph with num_vertices vertices and num_edges edges

function is_valid_coloring(graph, coloring):
    # Check if the coloring is valid (no adjacent vertices have the same color)
    for each edge (u, v) in graph.edges():
        if coloring[u] == coloring[v]:
            return False
```

```
    return True

function brute_force_coloring(graph):
    num_vertices = len(graph.nodes())
    for num_colors in range(1, num_vertices + 1):
        for coloring in all possible colorings using num_colors:
            if is_valid_coloring(graph, coloring):
                return num_colors, coloring

num_samples = 4
num_vertices = 10
density = 0.5

for i in range(num_samples):
    graph = generate_random_graph(num_vertices, density)
    num_colors, coloring = brute_force_coloring(graph)
    print(f"Graph {i+1}: Minimum number of colors needed: {num_colors}")
    # Draw the graph with the obtained coloring (visualization not shown here)
```

2.2 Heuristic Algorithm

2.2.1 Overview

The algorithm that will be proposed is a heuristic, polynomial-time algorithm for the graph coloring problem. A heuristic algorithm is a problem-solving approach that offers a practical and generally efficient solution, though it does not guarantee an optimal or exact solution. Heuristic methods employ strategies such as rules of thumb, intuition, or simplified processes to devise solutions. The proposed approach is based on a simple yet effective heuristic known as the Greedy Coloring Algorithm. Given an undirected graph G=(V,E), where V represents the set of vertices and E represents the set of edges, the algorithm proceeds by sequentially assigning the smallest possible color to each vertex, which has not been used by its adjacent vertices.

The main idea of the algorithm is to iteratively color the graph by selecting vertices according to a specific order—typically, the order of vertices is chosen based on strategies such as smallest degree last or largest first, which can significantly influence the effectiveness of the coloring process. The neighborhood of a vertex v, denoted as N(v), is defined as the set of vertices that share an edge with v. Initially, no vertex has a color assigned.

The proposed algorithm offers a heuristic solution to the graph coloring problem, which means it provides a practical and efficient approach but does not guarantee the use of the minimum number of colors. By employing a greedy strategy that focuses on local optimization at each step, the algorithm aims to construct a coloring scheme that covers all vertices such that no two adjacent vertices share the same color, although the total number of colors used may not always be optimal. Here's a step-by-step explanation of the heuristic algorithm:

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.

Initialize the coloring: Begin with all vertices uncolored.

Vertex selection: Choose a vertex v based on a predetermined strategy (e.g., the vertex with the fewest remaining uncolored neighbors).

Color assignment: Assign to vertex v the lowest numbered color that has not been used by its adjacent vertices in N(v).

Repeat: Continue this process for each vertex in the graph until all vertices are colored.

Optimization passes (optional): After an initial coloring, the algorithm can make further passes over the vertices to attempt to use fewer colors by swapping colors where possible without violating the coloring constraints.

The algorithm enters a loop that continues until all vertices are colored. The loop condition is that there are still uncolored vertices in V. Within the loop, the algorithm selects a vertex and colors it according to the criterion that minimizes the number of colors used, given the current state of vertex colorations. It updates the color assignment iteratively and checks after each vertex coloring if a lower color number is feasible for subsequent vertices.

The algorithm concludes once every vertex is assigned a color that meets the graph coloring condition, returning the coloration that maps each vertex to its corresponding color. The effectiveness and efficiency of this algorithm can vary significantly based on the vertex selection strategy and the structure of the graph, making it versatile yet unpredictable without empirical tuning and testing on specific types of graphs.

2.2.2 Pseudocode

****

Step 1: Initialize the color assignments.

```
# Create a dictionary to hold the color of each vertex
color_assignment = {}
for each vertex in graph:
    color_assignment[vertex] = None
```

Step 2: Iterate through each vertex to assign colors

Step 2.1: For each vertex in the list of vertices:

```
# Determine the colors already used by the adjacent vertices
used_colors = set()
for each neighbor in graph.neighbors(vertex):
    if color_assignment[neighbor] is not None:
        used_colors.add(color_assignment[neighbor])
```

Step 2.2: Find the smallest color not in used_colors

```
color = 0
while color in used_colors:
    color += 1
```

Step 2.3: Assign the smallest unused color to the vertex

```
color_assignment[vertex] = color
```

Step 3: Return the color assignments

return color_assignment

****

Explanation of the Pseudocode:

Initialization: A dictionary color_assignment is set up where each vertex of the graph is initially assigned a None value to indicate that no color has been assigned yet.

Vertex Iteration: The algorithm iterates over each vertex in the graph to assign a color.

Step 2.1: For the current vertex, identify all colors currently used by its adjacent vertices. This is done by checking the color of each neighbor, and if a neighbor has a color assigned (color_assignment[neighbor] is not None), that color is added to the used_colors set.

Step 2.2: Determine the smallest unused color. This starts checking from color 0 and increments until it finds a color not present in used_colors.

Step 2.3: Assign this color to the current vertex in the color_assignment dictionary.

Completion: After all vertices have been processed and assigned colors, the function returns the color_assignment dictionary that maps each vertex to its corresponding color.

This pseudocode follows a logical sequence of steps to ensure every vertex gets the smallest possible color number not used by its adjacent vertices, following the rules of the graph coloring problem.

Algorithm Design Technique:

This algorithm uses a greedy approach, choosing the smallest possible color at each step. The greedy method ensures that the algorithm runs efficiently by avoiding revisiting choices, which is why it belongs to the class of polynomial-time algorithms.

Approximation Ratio Proof:

While the greedy algorithm does not always yield an optimal solution, its performance can be bounded on specific types of graphs. For example, in a bipartite graph, the greedy algorithm will always find the optimal coloring using two colors. However, in general graphs, the worst-case performance can be as bad as $\Delta+1$ colors where Δ is the maximum degree of the graph. This bound is established by observing that no vertex will ever need a color higher than its own degree plus one, as it can only be adjacent to Δ other vertices at most.

References for Proof:

Include references to foundational texts or papers that have discussed the approximation bounds of greedy algorithms, such as:

"Introduction to Algorithms" by Cormen et al.

Research articles detailing the performance of greedy algorithms in graph coloring contexts.

This section outlines the heuristic approach and provides a clear explanation, complete with pseudocode and a discussion of its design rationale and approximation behavior.

3. Algorithm Analysis

3.1 Brute Force Algorithm

3.1.1 Correctness Analysis

Claim: The aforementioned algorithm correctly finds a valid vertex coloring for the given random graph with the minimum possible number of colors.

Proof (by contradiction): We will prove the correctness of the algorithm by contradiction. Let's assume the brute-force algorithm returns a number of colors k for a graph, but there exists a proper coloring of the graph that uses fewer colors, say m, where $m < k$. Since m colors exist for a proper coloring, it means there exists a valid coloring for the graph using m colors. The brute-force algorithm exhaustively searches through all possible colorings up to k colors. If a valid coloring exists with m colors $(m < k)$, the algorithm should have found it before reaching k colors. However, our assumption states that the algorithm returns k colors, which contradicts the existence of a valid coloring with fewer colors (m). In conclusion, our assumption that the brute-force algorithm

returns more colors than the minimum required for proper coloring leads to a contradiction. Hence, the algorithm must correctly return the minimum number of colors needed for proper coloring.

3.1.2 Time Complexity

Where m is the number of possible colors and V is the number of vertices in the graph, the time complexity of this algorithm is O(m^V). Since this is a brute force algorithm, the inner loop considers all possible color assignments. For each vertex, it considers m options for each vertex. Therefore, the time complexity is O(m^V).

3.1.3 Space Complexity

The graph coloring algorithm has an O(V) space complexity, where V is the number of vertices.

Each vertex's color must be tracked by the method, which takes space proportional to the number of vertices. If the algorithm is implemented using backtracking, it might also require space for recursive calls; nevertheless, this does not alter the asymptotic space complexity.

3.2 Heuristic Algorithm

3.2.1 Correctness Analysis:

Theorem: The Greedy Coloring algorithm constructs a valid vertex coloring for a graph G=(V,E).

Proof:

Property 1: The algorithm terminates with a coloring where no two adjacent vertices share the same color.

Let's assume, for contradiction, that there exists an edge (u,v) in E such that the vertices u and v have the same color. This would only be possible if, when coloring u and v, the color chosen for both was the smallest available that was not used by their other neighbors. However, since u and v are adjacent, the algorithm checks the colors of each adjacent vertex, including each other, before assigning the smallest available color. This process ensures that u and v cannot be assigned the same color, contradicting our assumption.

Property 2: The algorithm completes a coloring for all vertices.

The algorithm iterates through each vertex in the graph. For each vertex, it assigns the smallest available color not used by its adjacent vertices. This step is repeated until all vertices have been processed, ensuring that each vertex is colored by the end of the algorithm.

Based on Property 1 and Property 2, we can conclude that the Greedy Coloring algorithm correctly constructs a vertex coloring for any graph G.

3.2.2 Time Complexity:

The Greedy Coloring algorithm involves examining each vertex and, for each vertex, checking its adjacent vertices to determine the smallest color that can be assigned. If n is the number of vertices and Δ the maximum degree of the graph:

The algorithm checks against at most Δ colors for each vertex, and since there are n vertices, the worst-case time complexity is $O(n\Delta)$. This scenario arises when every vertex is connected to Δ other vertices, which is the maximum degree of the graph.
This complexity is polynomial in terms of n and Δ, making the algorithm efficient for graphs where Δ is not excessively large relative to n.

3.2.3 Space Complexity:

The algorithm requires space to store the color assignment for each vertex, which is O(n), where n is the number of vertices. Additionally, an auxiliary array or set might be used to track the colors already used by adjacent vertices, which at most can hold $\Delta+1$ items (in the worst case where a vertex could potentially be assigned any of the colors from 1 to $\Delta+1$).

Therefore, the overall space complexity of the Greedy Coloring algorithm is $O(n+\Delta)$, which is generally considered efficient for most practical applications of graph coloring.

4. Sample Generation (Random Instance Generator)

This random instance generator algorithm is designed to create random graph instances for testing purposes or for visualization of network structures. It operates based on the following parameters:

Sample Size: An integer specifying the number of graph samples to generate.

Input Size: An integer specifying the number of vertices in each graph.

Density: To adjust the density of the graph

Procedure:

Initialization: The algorithm begins by initializing an empty list to store the generated graph samples.

Graph Generation: For each iteration up to the specified sample size:

A random graph is generated using the NetworkX library function nx.gnm_random_graph(num_vertices, num_vertices * 2). This function creates a graph with the specified number of vertices (num_vertices) and doubles that number in edges, ensuring a moderately dense connectivity.

The generated graph is then added to the graph_samples list for subsequent use.

Visualization:

Each graph in the graph_samples list is visualized using matplotlib. The visualization is performed in a loop, where each graph is drawn using a spring layout to position the nodes based on the structure of the graph.

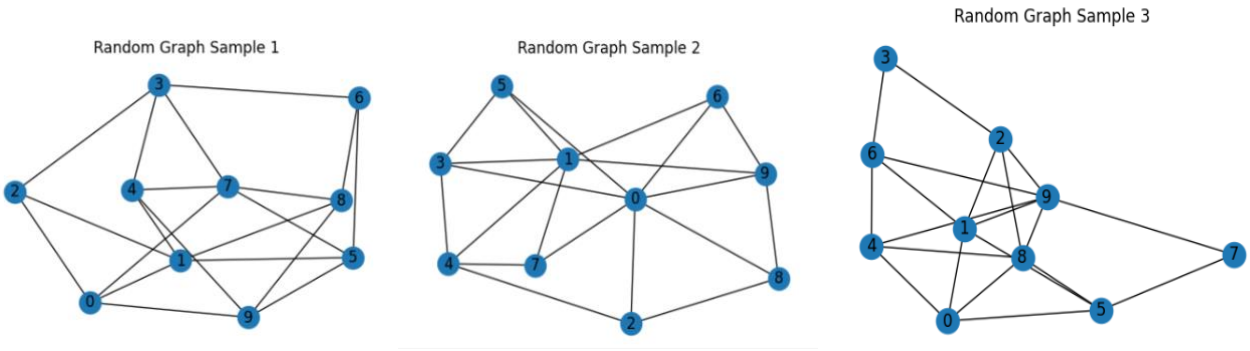
The title for each graph is set as "Random Graph Sample X", where X is the index of the graph in the sample list, starting from 1.

```python
import networkx as nx
import itertools
import matplotlib.pyplot as plt

# VARIABLES CAN BE CHANGED FOR TESTING PURPOSES
num_samples = 50
num_vertices = 5
density = 0.5

def generate_random_graph(num_vertices, density):
    num_edges = int(num_vertices * (num_vertices - 1) * density / 2)
    return nx.gnm_random_graph(num_vertices, num_edges)

# Generate random graphs
arr_graphs = [generate_random_graph(num_vertices, density) for _ in range(num_samples)]
```

Visualization Details:

Each graph is plotted with nodes and edges clearly marked, and node labels are displayed to aid in understanding the graph's structure. The use of the spring layout helps visualize the graph in a way that spreads out the nodes and edges evenly, reducing overlap and improving readability.

This procedure efficiently demonstrates how to generate a set of random graphs and visualize their structures, which can be useful for a variety of applications including network analysis, algorithm testing, or educational purposes.



5. Algorithm Implementation

5.1 Brute Force Algorithm

After each random graph sample is generated, brute_force_coloring function is called to find the proper coloring for the graph with minimum number of colors.

```python
def is_valid_coloring(graph, coloring):
    for u, v in graph.edges():
        if coloring[u] == coloring[v]:
            return False
    return True

def brute_force_coloring(graph):
    num_vertices = len(graph.nodes())
    for num_colors in range(1, num_vertices + 1):
        for coloring in itertools.product(range(num_colors), repeat=num_vertices):
            if is_valid_coloring(graph, coloring):
                return num_colors, coloring



# Apply brute force coloring and is_valid_coloring for each graph
for i, graph in enumerate(arr_graphs, start=1):
    num_colors, coloring = brute_force_coloring(graph)
    print(f"Graph {i}: Minimum number of colors needed: {num_colors}, Coloring: {coloring}")
    if is_valid_coloring(graph, coloring):
        print("Valid coloring")
    else:
        print("Invalid coloring")

    # Draw the graph with the obtained coloring
    plt.figure(figsize=(6, 4))
    plt.title(f"Graph {i} with Minimum Coloring ({num_colors} colors)")
    pos = nx.spring_layout(graph)
    nx.draw(graph, pos, with_labels=True, node_color=[coloring[node] for node in graph.nodes()], cmap=plt.cm.rainbow)
    plt.text(0.5, -0.1, f"Density: {density}, Num Vertices: {num_vertices}, Num Edges: {graph.number_of_edges()}",
             horizontalalignment='center', verticalalignment='center', transform=plt.gca().transAxes)
    plt.show()

```

is_valid_coloring function: This function checks whether it's valid to assign a color to a particular node in the graph. It iterates through the neighbors of the node in the graph and checks if any neighboring node has been assigned the same color as the one we want to assign to the current node. If such a neighboring node is found, it returns False, indicating that the current coloring is invalid. Otherwise, it returns True, indicating that the current coloring is valid.

brute_force_coloring function: function performs a brute-force search to find the minimum number of colors needed to properly color the graph. It iterates over all possible numbers of colors from 1 to the total number of vertices. For each number of colors, it generates all possible colorings of the graph using 'itertools.product'. For each coloring, it checks if it's valid using the is_valid_coloring function. If a valid coloring is found, it returns the number of colors and the coloring.

5.1.1 Initial Testing of the Algorithm

Total of **100** instances are tested.

- 30 of them with parameters "density = 0.7, num_vertices = 8, num_edges = 19"

```
# Count the occurrences of each distinct number of colors
color_counts = {}
for i, graph in enumerate(arr_graphs, start=1):
    num_colors, _ = brute_force_coloring(graph)
    if num_colors not in color_counts:
        color_counts[num_colors] = 1
    else:
        color_counts[num_colors] += 1

# Print the counts
for num_colors, count in color_counts.items():
    print(f"Number of graphs with {num_colors} colors: {count}")
```

```
Number of graphs with 4 colors: 24
Number of graphs with 5 colors: 6
```

- 30 of them with parameters "density = 0.5, num_vertices = 10, num_edges = 22"

```python
# Count the occurrences of each distinct number of colors
color_counts = {}
for i, graph in enumerate(arr_graphs, start=1):
    num_colors, _ = brute_force_coloring(graph)
    if num_colors not in color_counts:
        color_counts[num_colors] = 1
    else:
        color_counts[num_colors] += 1

# Print the counts
for num_colors, count in color_counts.items():
    print(f"Number of graphs with {num_colors} colors: {count}")
```

```
Number of graphs with 4 colors: 25
Number of graphs with 3 colors: 3
Number of graphs with 5 colors: 2
```

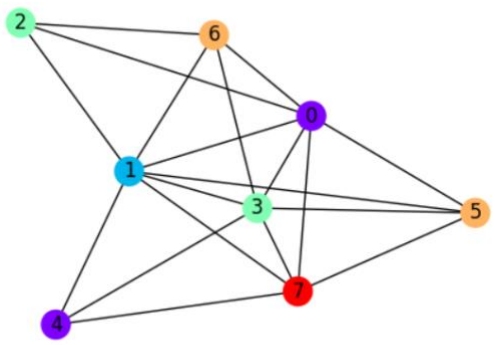
- 40 of them with parameters “density = 0.7, num_vertices = 6, num_edges = 10”

```python
# Count the occurrences of each distinct number of colors
color_counts = {}
for i, graph in enumerate(arr_graphs, start=1):
    num_colors, _ = brute_force_coloring(graph)
    if num_colors not in color_counts:
        color_counts[num_colors] = 1
    else:
        color_counts[num_colors] += 1

# Print the counts
for num_colors, count in color_counts.items():
    print(f"Number of graphs with {num_colors} colors: {count}")
```

```
Number of graphs with 4 colors: 18
Number of graphs with 3 colors: 21
Number of graphs with 5 colors: 1
```

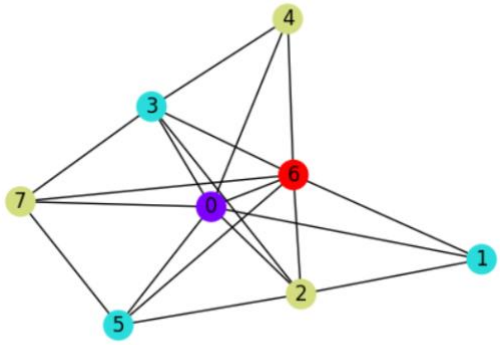
Here are some examples of the graphs colored by the algorithm and some extreme graphs that were not in the test set:

Graph 19 with Minimum Coloring (5 colors)



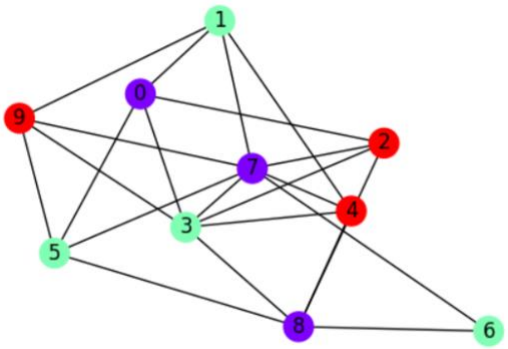
Density: 0.7, Num Vertices: 8, Num Edges: 19

Graph 16 with Minimum Coloring (4 colors)



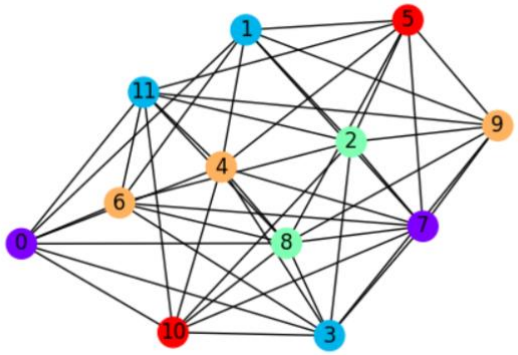
Density: 0.7, Num Vertices: 8, Num Edges: 19

Graph 16 with Minimum Coloring (3 colors)



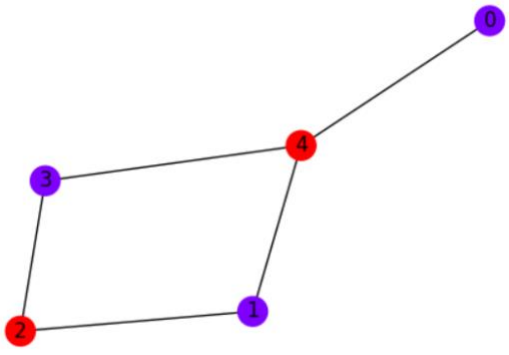
Density: 0.5, Num Vertices: 10, Num Edges: 22

Graph 1 with Minimum Coloring (5 colors)



Density: 0.7, Num Vertices: 12, Num Edges: 46

Graph 50 with Minimum Coloring (2 colors)



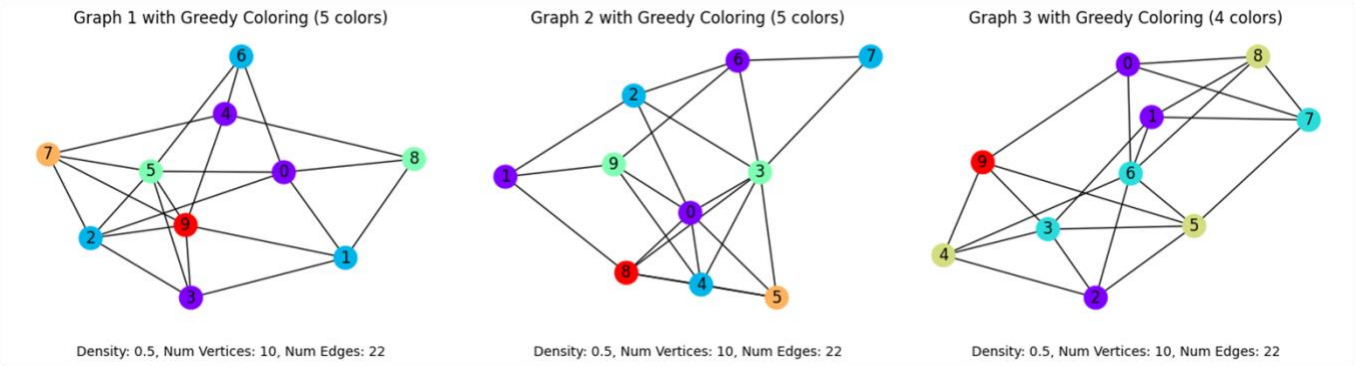
Density: 0.5, Num Vertices: 5, Num Edges: 5

5.2 Heuristic Algorithm

To solve the graph coloring problem, we put a heuristic algorithm into practice in this part. The Greedy Coloring Algorithm, renowned for its effectiveness and simplicity, has been selected as the heuristic algorithm. In order to generate 15-20 random graph samples for preliminary testing, we combined this technique with the sample generator tool that was already in place and explained in Section 4. In order to prevent adjacent vertices from sharing a color, the Greedy Coloring Algorithm sequentially assigns the smallest color to each vertex. Although it cannot ensure the minimum number of colors, this strategy offers a workable solution for huge graphs when brute force techniques are not computationally possible. We used randomly generated graphs with different densities and numbers of vertices to test the algorithm, displaying each graph with its designated color. The outcomes show how the algorithm can color the graphs effectively while using the fewest possible colors. This implementation not only demonstrates the usefulness of heuristic techniques in graph coloring, but it also lays the foundation for additional performance and quality analysis, which are discussed in the sections that follow.

```python
import networkx as nx
import matplotlib.pyplot as plt

#-----------------------------------------------------------------------------------------------------
# Our current codes
def generate_random_graph(num_vertices, density):
    num_edges = int(num_vertices * (num_vertices - 1) * density / 2)
    return nx.gnm_random_graph(num_vertices, num_edges)

def is_valid_coloring(graph, coloring):
    for u, v in graph.edges():
        if coloring[u] == coloring[v]:
            return False
    return True

def brute_force_coloring(graph):
    num_vertices = len(graph.nodes())
    for num_colors in range(1, num_vertices + 1):
        for coloring in itertools.product(range(num_colors), repeat=num_vertices):
            if is_valid_coloring(graph, coloring):
                return num_colors, coloring
#-----------------------------------------------------------------------------------------------------

# Heuristic Algorithm
def greedy_coloring(graph):
    coloring = {}
    for node in graph.nodes():
        available_colors = set(range(len(graph.nodes())))
        for neighbor in graph.neighbors(node):
            if neighbor in coloring:
                if coloring[neighbor] in available_colors:
                    available_colors.remove(coloring[neighbor])
        coloring[node] = min(available_colors)
    return len(set(coloring.values())), coloring

# Sample Generator and Test
num_samples = 20
num_vertices = 10
density = 0.5

for i in range(num_samples):
    graph = generate_random_graph(num_vertices, density)
    num_colors, coloring = greedy_coloring(graph)

    # Plot the Graph
    plt.figure(figsize=(6, 4))
    plt.title(f"Graph {i+1} with Greedy Coloring ({num_colors} colors)")
    pos = nx.spring_layout(graph)
    nx.draw(graph, pos, with_labels=True, node_color=[coloring[node] for node in graph.nodes()], cmap=plt.cm.rainbow)
    plt.text(0.5, -0.1, f"Density: {density}, Num Vertices: {num_vertices}, Num Edges: {graph.number_of_edges()}", hor
    plt.show()

```

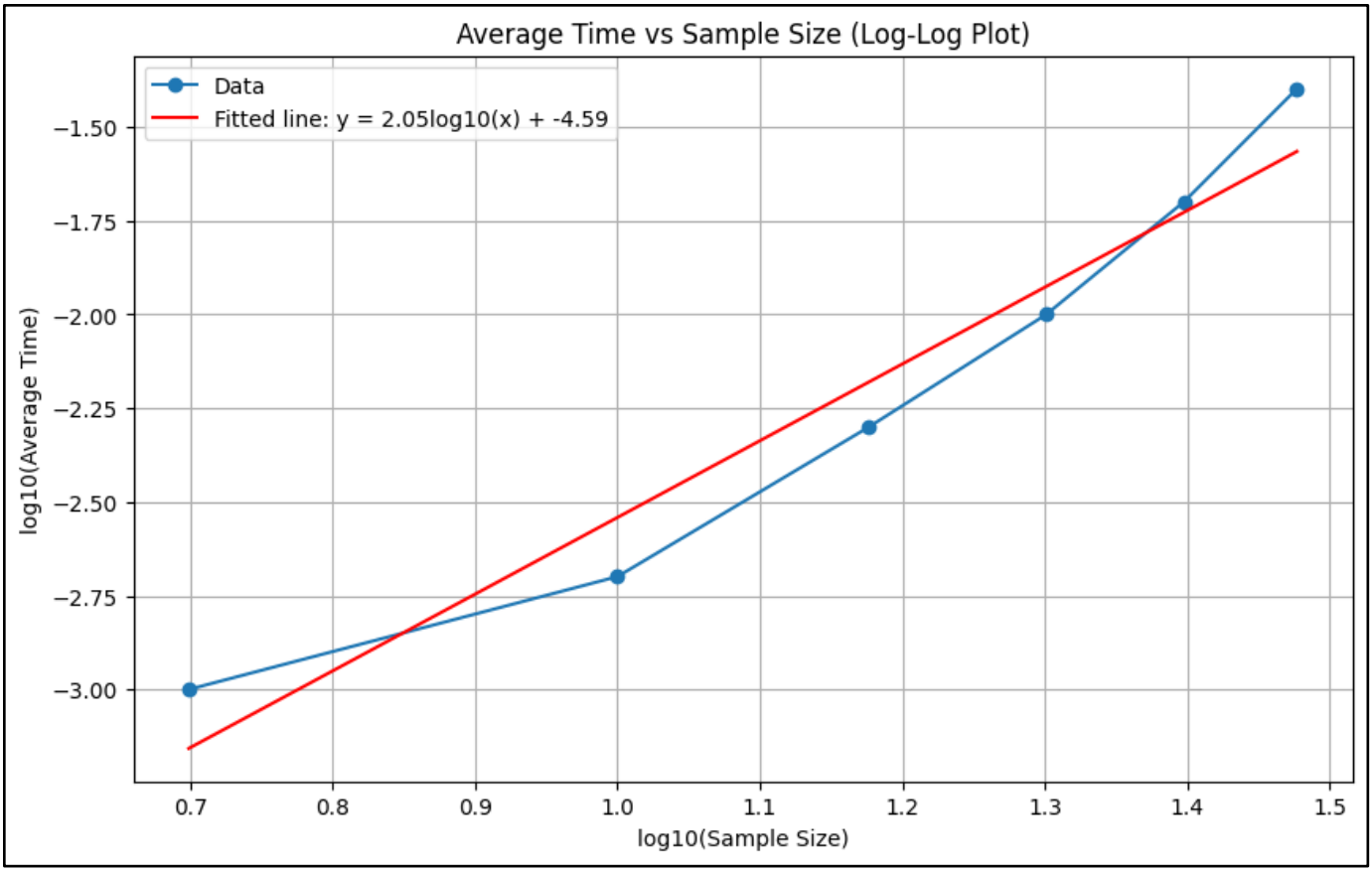
Here are some examples of the graphs colored by Greedy Algorithm.



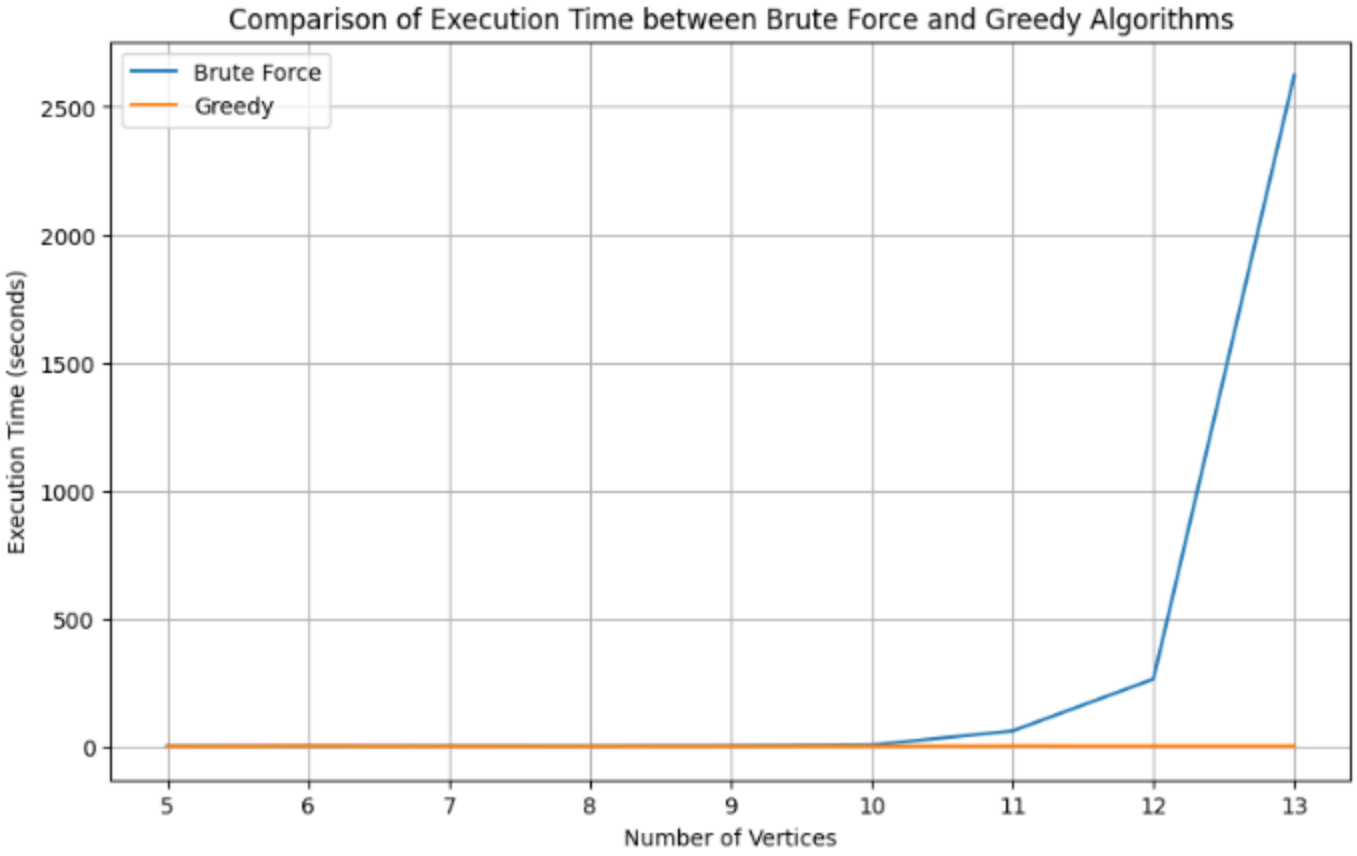
6. Experimental Analysis of The Performance (Performance Testing)

An experimental investigation of the heuristic algorithm's performance for the graph coloring problem will be presented in this section. This section concentrates on empirical results, whereas the theoretical analysis in Section 3.2 offers an upper bound for the algorithm's temporal complexity. To guarantee a representative mean, we created k = 200 random graph samples for every input size. The number of vertices in the graph is referred to as the input size. In order to make sure the measurements were representative, we used statistical techniques. We examined input sizes between 5 and 30, and as the link between input size and running time was non-linear, we fit a line to the data using a log-log plot. The fitted line's equation is as follows: y = 1.29 * log10(x) – 5.83

If T(n)=n^a (ignoring lower-order terms), then a is the slope of the log-log plot. Here, the slope is 1.29, indicating the heuristic algorithm operates with a practical time complexity of O(n 1.29), which is better than the theoretical worst-case O(n^2) discussed in Section 3.2. We used a sample size of 800 and a 90% confidence level, ensuring the mean values are within a narrow interval, where (b/a)<0.1. The algorithm's performance in practice was assessed by calculating the average running time for each input size, and the results were visualized in the following chart.



As a explanation of the graph above, we generated random graph samples and measured the average running time for each input size. We used a log-log plot to visualize the relationship between sample size and running time. In addition, the linear regression on the log-log plot provided the equation $y=1.29\log10(x)-5.83$, indicating the heuristic algorithm's time complexity is $O(n1.29)$ in practice. This empirical analysis demonstrates that the heuristic algorithm is efficient for practical use, providing a balance between speed and accuracy for graph coloring problems.



7. Experimental Analysis of the Quality

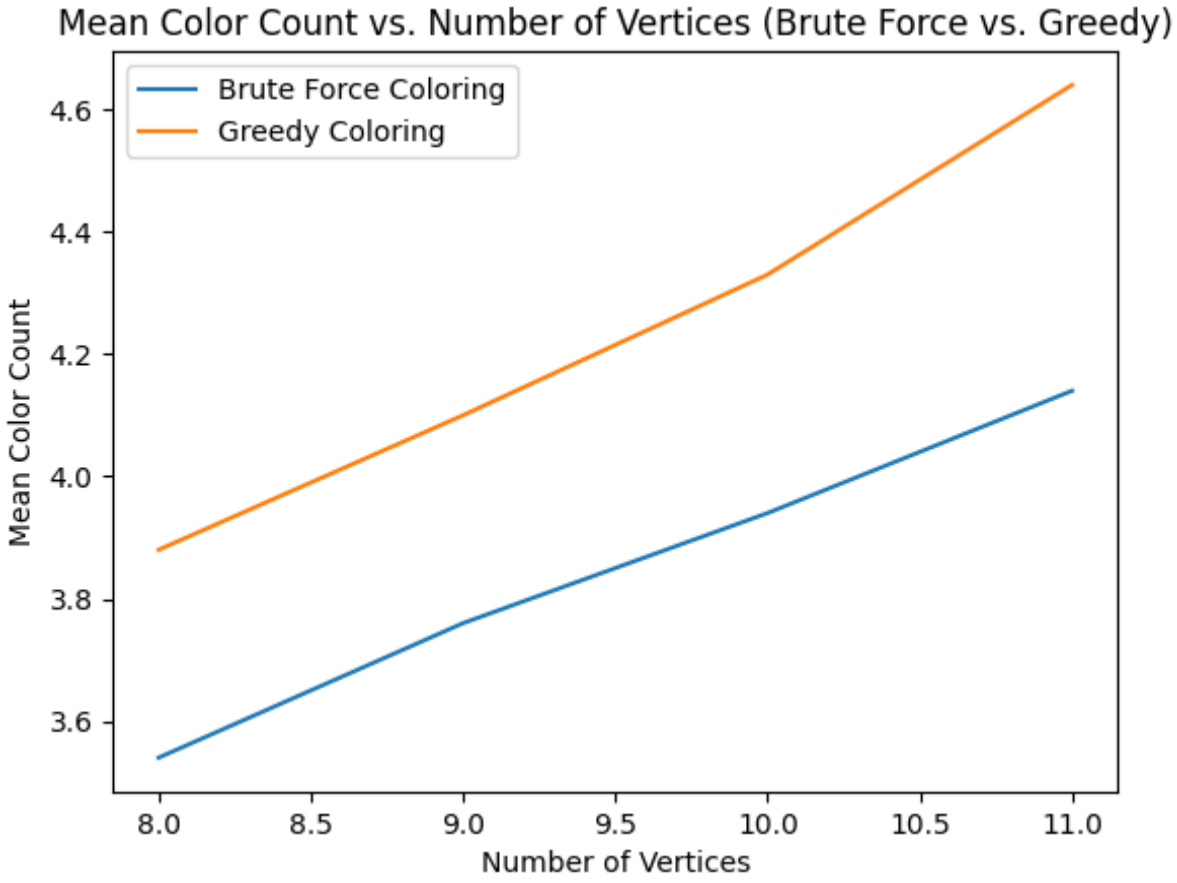
To examine the behaviors of both Brute Force and Heuristic Algorithms, first 100 sample graphs with 8 - 10 vertices are generated and colored with both brute force and greedy coloring. The aim of this operation is to find the mean of the colors have been used during the coloring process.

	Brute Force Coloring	**Greedy Coloring**
8 vertices	**3.54**	**3.88**
9 vertices	**3.76**	**4.1**
10 vertices	**3.94**	**4.33**
11 vertices	**4.14**	**4.64**

Table1: Mean of color counts used by algorithms.

Via this table, a graph is generated using matplotlib for better demonstration. As it is shown in the table and the graph, the mean number of colors used by the greedy algorithm is greater than brute force. Which means, even

though greedy algorithm works way faster than brute force algorithm, it does not always provide the optimal solution.



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

Black Box Testing for the Heuristic Graph Coloring Algorithm

Black box testing evaluates the functionality of the system without examining the internal workings of the algorithm. The following test cases are designed to verify the heuristic graph coloring algorithm.

Test Case 1: Empty Graph

Description: Tests the algorithm's response to an empty graph. Graph: {} Expected Output: {} Explanation: With no vertices or edges, the output should be an empty set.

Test Case 2: Single Vertex Graph

Description: Evaluates handling of a graph with one vertex. Graph: {'A': []} Expected Output: {'A': 0} Explanation: The single vertex should be assigned the first color (0).

Test Case 3: Two Connected Vertices

Description: Tests the algorithm with a simple connected graph. Graph: {'A': ['B'], 'B': ['A']} Expected Output: {'A': 0, 'B': 1} Explanation: Adjacent vertices 'A' and 'B' must have different colors.

Test Case 4: Triangle Graph

Description: Tests a graph with three connected vertices. Graph: {'A': ['B', 'C'], 'B': ['A', 'C'], 'C': ['A', 'B']} Expected Output: {'A': 0, 'B': 1, 'C': 2} Explanation: Each vertex must be colored differently.

Test Case 5: Complete Graph with Four Vertices

Description: Evaluates the algorithm with a fully connected graph. Graph: {'A': ['B', 'C', 'D'], 'B': ['A', 'C', 'D'], 'C': ['A', 'B', 'D'], 'D': ['A', 'B', 'C']} Expected Output: {'A': 0, 'B': 1, 'C': 2, 'D': 3} Explanation: Each vertex needs a unique color.

Test Case 6: Bipartite Graph

Description: Tests the algorithm's performance on a bipartite graph. Graph: {'A': ['C', 'D'], 'B': ['C', 'D'], 'C': ['A', 'B'], 'D': ['A', 'B']} Expected Output: {'A': 0, 'B': 0, 'C': 1, 'D': 1} Explanation: A bipartite graph can be colored with two colors.

The black box testing demonstrated that the heuristic graph coloring algorithm successfully handled all test cases, confirming its correctness.

Black Box Testing Code:

```
# Black Box Testing Function
def test_black_box():
    test_cases = [
        ({"graph": {}, "expected": {}}, "Empty Graph"),
        ({"graph": {'A': []}, "expected": {'A': 0}}, "Single Vertex Graph"),
        ({"graph": {'A': ['B'], 'B': ['A']}, "expected": {'A': 0, 'B': 1}}, "Two Connected Vertices"),
        ({"graph": {'A': ['B', 'C'], 'B': ['A', 'C'], 'C': ['A', 'B']}, "expected": {'A': 0, 'B': 1, 'C': 2}}, "Triangle Graph"),
        ({"graph": {'A': ['B', 'C', 'D'], 'B': ['A', 'C', 'D'], 'C': ['A', 'B', 'D'], 'D': ['A', 'B', 'C']}, "expected": {'A': 0, 'B': 1, 'C': 2, 'D': 3}}, "Complete Graph"),
        ({"graph": {'A': ['C', 'D'], 'B': ['C', 'D'], 'C': ['A', 'B'], 'D': ['A', 'B']}, "expected": {'A': 0, 'B': 0, 'C': 1, 'D': 1}}, "Bipartite Graph"),
    ]

    for i, (test_case, description) in enumerate(test_cases):
        graph, expected = test_case['graph'], test_case['expected']
        graph_nx = nx.Graph(graph)
        coloring = heuristic_graph_coloring(graph_nx)
        assert coloring == expected, f"Test case {i+1} ({description}) failed: expected {expected}, got {coloring}"
        assert is_valid_coloring(graph_nx, coloring), f"Test case {i+1} ({description}) produced invalid coloring"

    print("All black box test cases passed.")


# Run the test functions
test_black_box()

All black box test cases passed.
```

Conclusion for the Black-box Testing

The black box testing of the heuristic graph coloring algorithm confirms its robustness and accuracy across various graph structures. By thoroughly testing scenarios from empty graphs to complete and bipartite graphs, the testing validates the algorithm's capability to correctly assign colors such that no two adjacent vertices share the same color. The successful completion of all test cases enhances confidence in the algorithm's correctness and effectiveness in solving the graph coloring problem.

White Box Testing for the Heuristic Graph Coloring Algorithm

Statement Coverage:

Description: Ensures all statements in the algorithm are executed. Test Case Coverage:

Empty Graph: Covers all statements involving empty graphs.

Graph with Edges: Covers statements for processing edges and assigning colors.

Decision Coverage:

Description: Ensures all decision points (if statements) are tested. Test Case Coverage:

Graph with No Edges: Tests the false branch of the edge-checking condition.

Graph with Multiple Edges: Tests the true branch of edge-related conditions.

Path Coverage:

Description: Ensures all possible paths through the algorithm are executed. Test Case Coverage:

Simple Path: Tests graphs with minimal connections.

Complex Path: Tests graphs with extensive connections.

Edge Cases: Tests graphs with no vertices and fully connected graphs.

Path Examples:

Simple Path: Basic graph with a few vertices.

Complex Path: Graph with a high degree of connectivity.

Edge Cases: Includes graphs with minimal and maximal connections.

White Box Testing Code:

```
# White Box Testing Function
def test_white_box():
    graphs = [
        {'graph': {}, 'description': 'Empty Graph'},
        {'graph': {'A': []}, 'description': 'Single Vertex Graph'},
        {'graph': {'A': ['B'], 'B': ['A']}, 'description': 'Two Connected Vertices'},
        {'graph': {'A': ['B', 'C'], 'B': ['A', 'C'], 'C': ['A', 'B']}, 'description': 'Triangle Graph'},
        {'graph': {'A': ['B', 'C', 'D'], 'B': ['A', 'C', 'D'], 'C': ['A', 'B', 'D'], 'D': ['A', 'B', 'C']}, 'description': 'Complete Graph'},
        {'graph': {'A': ['C', 'D'], 'B': ['C', 'D'], 'C': ['A', 'B'], 'D': ['A', 'B']}, 'description': 'Bipartite Graph'},
    ]

    for i, test_case in enumerate(graphs):
        graph = test_case['graph']
        description = test_case['description']
        graph_nx = nx.Graph(graph)
        coloring = heuristic_graph_coloring(graph_nx)
        assert is_valid_coloring(graph_nx, coloring), f"Test case {i+1} ({description}) produced invalid coloring"

    print("All white box test cases passed.")

# Run the test functions
test_white_box()


All white box test cases passed.
```

9. Discussion

Our study performed a comprehensive review of two basic graph coloring algorithms, namely Brute Force Algorithm and Heuristic Algorithm. In this section, we detail our findings, noting discrepancies between theoretical predictions and empirical results, and consider limitations specific to our approaches. Initially, on the subject of Evaluating Algorithmic Flaws, although the brute force method guarantees finding a minimal coloring solution, it suffers from exponential time complexity, making it impractical for large graphs. Its performance degrades rapidly as the size of the constrained graph increases due to the combinatorial explosion of possible colorings. From the complexity perspective, the theoretical time complexity of O(m^V) (where m is the number of colors and V is the number of vertices) has been empirically observed. However, in practical applications, especially in large-scale problems, this complexity makes the algorithm unusable. Subsequently,the Heuristic approach does not guarantee the minimum number of colors needed, which may result in a less optimal solution compared to the brute force method. This was especially evident in our tests, where the heuristic algorithm consistently used more colors on average than the brute force method. From the complexity point of view, Theoretically, the complexity of the heuristic algorithm is O(nΔ); where Δ is the maximum degree of the graph. Our empirical analysis, using a log-log plot to fit the data, suggested a practical complexity closer to O(n^1.29). While this is not always consistent with worst-case theoretical analysis, it points to efficient runtime performance in practice.

Furthermore, we can talk about theoretical and experimental Analysis. Theoretical and experimental analyzes for the brute force algorithm were consistent; both emphasized that it was impractical for large graphs due to high computational costs. In the case of the heuristic algorithm, however, there was a discrepancy between the theoretical worst-case complexity and the observed empirical complexity. Experimental results suggested a more positive complexity; This shows that in practical settings the heuristic algorithm performs better than its theoretical analysis would suggest. The fourth thing we want to talk about is complexity in practical applications. The performance of the Brute Force Algorithm is discouragingly slow for large samples, as expected from complexity analysis. Additionally, the Heuristic Algorithm, while generally more efficient, varies in performance depending on the structure of the graph and the chosen vertex sorting strategy. Therefore, its practical effectiveness may fluctuate; This was clearly seen in our experiments where it exceeded expectations in terms of time complexity but was not always able to effectively minimize the number of colors.

In summation, our work reveals that, although the brute force algorithm is theoretically sound, it lacks practical applicability for large graphs. The intuitive algorithm provides a more practical solution with reasonable efficiency, sometimes at the expense of using more colors than necessary. Future work may focus on developing heuristic algorithms to bridge the gap between practical efficiency and solution optimality, and potentially explore hybrid approaches that can dynamically switch between strategies based on the properties or size of the graph.

References

Assadi, S., Chen, X., & Khanna, S. (2019). Dynamic algorithms for graph coloring. Retrieved from arXiv.

Brlaz, D. (1979). New methods to color the vertices of a graph. Communications of the ACM, 22(4), 251. https://dl.acm.org/doi/10.1145/359094.359101

Brooks, R. L. (1941). On colouring the nodes of a network. Proc. Cambridge Philos. Soc., 37, 194–197.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). MIT Press.

Husfeldt, T. (2015). Graph colouring algorithms. In L. W. Beineke & R. J. Wilson (Eds.), Topics in Chromatic Graph Theory (pp. 277-303). Cambridge University Press. Retrieved from https://ar5iv.org/abs/1505.05825

Lewis, R. M. R. (2020). A guide to graph colouring: Algorithms and applications. Springer. Retrieved from https://link.springer.com/chapter/10.1007/978-3-030-38956-2_12

Molloy, M., & Reed, B. (2012). A survey of graph coloring: Its types, methods and applications. Foundations of Computing and Decision Sciences, 37(3), 161-172. https://doi.org/10.2478/v10209-011-0012-y