

CS301

2023-2024 Spring

Project Progress Report

Group 156

::Group Members::

Mustafa Kulak

Demir Boğa

1. Problem Description

<i>Name:</i>	<i>Graph Coloring</i>
<i>Input:</i>	<i>An undirected graph $G(V, E)$ with n nodes and node set V, and edge set E</i>
<i>Question:</i>	<i>What is the minimum number of colors needed to assign colors to each vertex in V such that no vertices connected by an edge in E have the same color.</i>

1.1 Overview

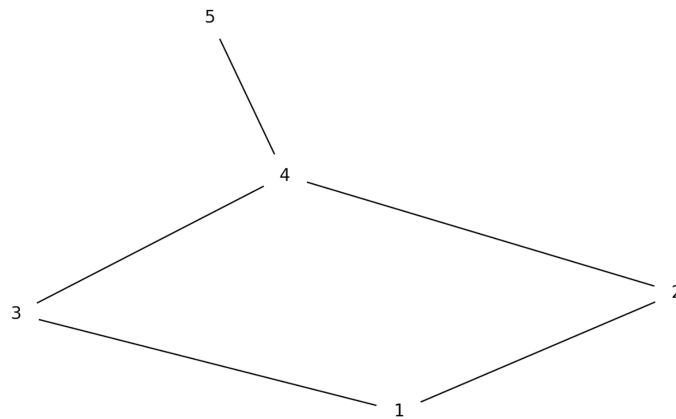
The Graph Coloring problem involves assigning colors to vertices in a graph such that no two adjacent vertices share the same color, using a minimum amount of different colors. In other words, it seeks to find a way to color the nodes of an undirected graph such that for any edge (u, v) in E , the colors assigned to u and v must be different while minimizing the number of colors used. Formally given an undirected graph $G = (V, E)$ where V is a set of vertices, and E is a set of edges the task is to find a coloring function $c: V \rightarrow \{1, 2, \dots, k\}$ for the graph where $c(u) \neq c(v)$ for every edge (u, v) in E with the minimum number k , known as the chromatic number.

1.2 Decision Problem

Given an undirected graph $G = (V, E)$, does a coloring of V using k or fewer different colors such that for every edge (u, v) in E , the vertices u and v are colored with different colors exist?

1.3 Optimization Problem

Given an undirected graph $G = (V, E)$, the objective is to minimize the number of colors needed to color the graph such that no two adjacent vertices have the same color. This involves finding the smallest positive integer k for which the graph G can be k -colored.



1.4 Example Illustration

Figure 1

Figure 1 illustrates an undirected graph with 5 nodes and 5 edges. The brute force algorithm makes every possible color combination and checks whether it satisfies the constraint that no nodes connected by an edge have the same color. As shown in Figure 2 which illustrates every possible solution with the limitation of maximum 3 colors. It is seen on the Figure 2 that there are some superior solutions within the others (Valid Coloring #1, #10, #13).

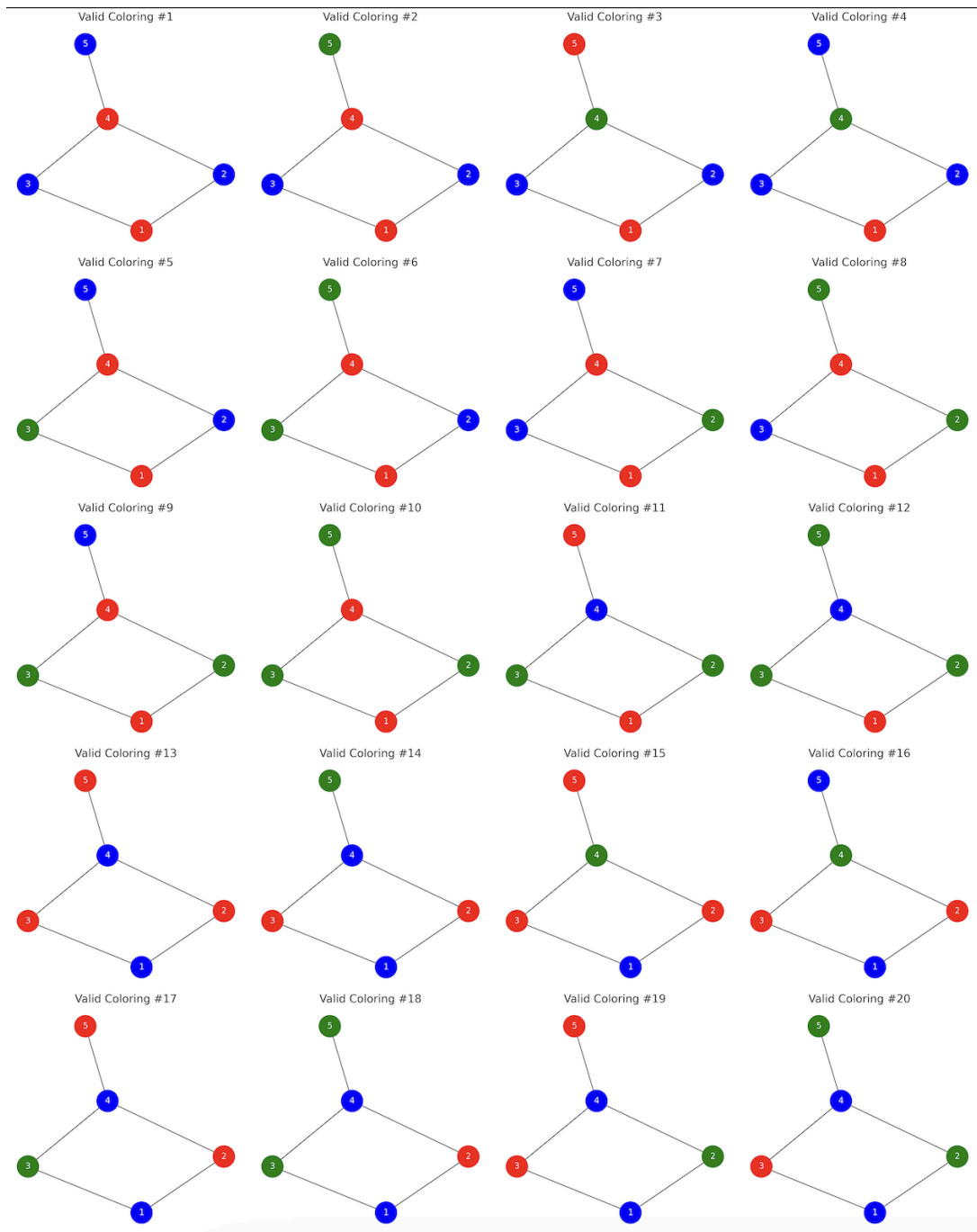


Figure 2

1.5 Real World Application

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

1.5.1 Map Coloring

The classic example of the graph coloring problem is map coloring, where the goal is to color the regions of a map (represented as vertices in a graph) in such a way that no two adjacent regions have the same color. This problem has applications in cartography, political boundary demarcation, and even in scheduling tasks across different geographical regions.

1.5.2 Scheduling

In scheduling problems, tasks or events often have dependencies or constraints that dictate when they can occur. By representing these constraints as a graph and applying graph coloring, you can schedule tasks/events in such a way that no two conflicting tasks occur simultaneously. For example, in class scheduling, you might represent each class as a node in the graph and use graph coloring to assign time slots to classes, ensuring that no two classes in the same room overlap.

1.5.3 Register Allocation in Compiler Design

When compiling high-level programming languages to machine code, compilers need to assign variables to registers efficiently. This is crucial for optimizing the performance of the compiled code. Graph coloring techniques are used to allocate registers to variables in such a way that no two variables that are simultaneously live (i.e., used at the same time) are assigned to the same register.

1.6 Hardness of the Problem

Karp's theorem proves that the decision version of the graph coloring problem, given an undirected graph $G=(V,E)$ and a positive integer k , asking whether it is possible to color the vertices of the graph with k or fewer colors such that no two adjacent vertices share the same color, is an NP-Complete problem.

1.6.1 Proving NP

Confirming membership in NP entails validating whether a provided vertex coloring of the graph and a given integer k meet two criteria: firstly, that the count of colors utilized doesn't surpass k , and secondly, that adjacent vertices aren't assigned identical colors. This validation procedure can be executed in polynomial time, demonstrating that the graph coloring problem belongs to NP.

1.6.2 Proving NP-Hard

Karp showed that the graph coloring problem is just as tough as solving well-known SAT problems. He did this by creating a special formula that's true only if you can color a graph with a certain number of colors. This proves that solving the graph coloring problem is as hard as solving any other problem in NP, meaning it can't be done quickly with a simple method.

1.6.3 Conclusion to NP-Complete

As the graph coloring problem falls within both NP and NP-hard categories, Karp determines it to be NP-Complete. This classification highlights its position as among the most challenging computational problems within NP, underscoring the inherent difficulty in achieving optimal solutions for graph coloring.

2. Algorithm Description:

2.1 Brute Force

2.1.1 Overview

The brute force algorithm for graph coloring takes a graph represented by an adjacency matrix and the number of available colors as input. It aims to find all possible valid colorings for the graph, where no two adjacent vertices share the same color.

1. Recursive Exploration: The algorithm employs a recursive function `color_vertex` to explore all color combinations for each vertex in the graph. This function starts with the first vertex and an empty color assignment (representing the colors assigned so far).

2. Color Trial: For the current vertex, the `color_vertex` function iterates through all available colors (m). For each color, it checks if assigning that color violates the coloring rule (adjacent vertices having the same color). This is done by looping through all neighbors and ensuring none of them share the chosen color.

3. Safe Color Assignment: If the chosen color is safe (no conflicts with neighbors), the function assigns it to the current vertex in the color assignment list.

4. Recursive Calls: The function then recursively calls itself for the next uncolored vertex (vertex + 1) with the updated color assignment. This process continues, exploring all possible color combinations for each vertex.

5. Valid Coloring Identification: When `color_vertex` reaches the last vertex, and all colors have been safely assigned to adjacent vertices, it signifies a complete valid coloring has been found.

The current color assignment list is then appended to a list of all colorings (all_colorings) maintained by the algorithm.

6. Exhaustive Search: The color_vertex function is initially called for the first vertex, triggering a cascade of recursive calls that explore all possible color combinations for each subsequent vertex. This ensures that every valid coloring for the graph is eventually considered and potentially added to the all_colorings list.

Limitations:

- The brute force approach becomes computationally expensive for large graphs due to the exponential growth in the number of color combinations explored.
- It finds all valid colorings, which can be redundant if you're only interested in finding any valid coloring (not necessarily the one using the least number of colors).

2.1.2 Pseudocode

```
brute_force_coloring(graph, m)

    num_vertex = number of vertices in graph
    all_colorings = empty list

    color_vertex(0, [0] * num_vertex)

function color_vertex(vertex, color_assignment)

    if vertex == num_vertex
        append a copy of color_assignment to all_colorings
        return

    for color in range(m)
        is_safe = True
        for neighbor in range(num_vertex)
            if graph[vertex][neighbor] == 1 and color_assignment[neighbor] ==
                color:
                is_safe = False
                break
        if is_safe
            color_assignment[vertex] = color
            color_vertex(vertex + 1, color_assignment)
```

2.1.3 Python Code

```
def brute_force_coloring(graph, m):
    num_vertex = len(graph)
    all_colorings = []

    def color_vertex(vertex, color_assignment):
        if vertex == num_vertex:
            # Reached the last vertex, a complete coloring is found.
            all_colorings.append(color_assignment.copy())
            return

        # Try all possible colors for the current vertex
        for color in range(m):
            # Check if the color is safe (no adjacent vertices share the same color)
            is_safe = True
            for neighbor in range(num_vertex):
                if graph[vertex][neighbor] == 1 and color_assignment[neighbor] == color:
                    is_safe = False
                    break
            if is_safe:
                # Color is safe, assign it and continue coloring remaining vertices
                color_assignment[vertex] = color
                color_vertex(vertex + 1, color_assignment)

    # Start the coloring process from the first vertex with an empty color assignment
    color_vertex(0, [0] * num_vertex)

    return all_colorings
```

2.2 Heuristic Algorithm

2.2.1 Overview

A heuristic algorithm is a problem-solving approach that uses practical strategies or rules of thumb to find approximate solutions, especially when exact methods are impractical or infeasible. Instead of guaranteeing an optimal solution, heuristic algorithms aim to quickly find a satisfactory solution that is good enough for the given problem within a reasonable amount of time. In our solution, we used the semantics of “genetic algorithms” as our heuristic. The main idea of genetic algorithms is to mimic the process of natural selection and evolution to iteratively improve solutions to optimization and search problems by maintaining a population of candidate solutions, selecting the fittest individuals for reproduction, and applying genetic operators such as crossover and mutation to generate new offspring solutions. Application of genetic algorithm idea in our Graph Coloring problem is to use selective breeding by fitness. Fitness can be described as how well a candidate solution fits to the goal (desired) state. In our problem, the rate of fitness is defined by comparing the number of conflicts in the colors of the vertices. In our case, since the fitness represents the rate of conflicts, the code provided uses the candidate

solutions with the lowest fitness. The algorithm operates with selecting the candidate solutions with the best fitness properties, combining them to produce better candidates. While producing the new offspring, some rate of random changes are allowed to discover better solutions. This strategy corresponds to the term “Mutations” in the genetic algorithm approach.

2.2.2 Pseudocode

```
*****
begin
  generation = 0
  while ( best_fitness != 0 ):
    selection(population)
    crossover(population)
    mutation(population)
    if ( Best(population) < best_fitness ):
      then best_fitness = Best(population)
    generation += 1
  end while
  return best_fitness
end
*****
```

This pseudocode outlines the core steps of a genetic algorithm (GA) approach to solve the graph coloring problem. Here is a step by step breakdown of the pseudocode:

1. Initialization:

- *generation = 0*: Initialize the generation counter to 0.

2. Main Loop:

- *while (best_fitness != 0)*: Execute the loop until a solution with the best fitness (indicating a valid graph coloring where no adjacent vertices share the same color) is found.
- Within each iteration of the loop:
 - Perform selection, crossover, and mutation operations on the population of candidate solutions.

3. Selection:

- *selection(population)*: Apply selection to choose individuals from the population for reproduction based on their fitness. The details of the selection process are not specified in the pseudocode.

4. Crossover:

- *crossover(population)*: Apply crossover to produce offspring solutions by combining genetic information from selected individuals. In the context of the graph coloring problem, this might involve exchanging color assignments between vertices of parent solutions.

5. Mutation:

- *mutation(population)*: Apply mutation to introduce random changes to individual solutions, maintaining diversity in the population and exploring new regions of the search space. Mutation might involve changing the color of a randomly selected vertex.

6. Fitness Update:

- *if (Best(population) < best_fitness)*: Check if the best fitness in the current population is better than the previously best fitness found (*best_fitness*). If it is, update *best_fitness* to the new best fitness value.

7. Generation Increment:

- *generation += 1*: Increment the generation counter after each iteration of the loop.

8. Termination:

- The loop continues until a solution with the best fitness of 0 is found, indicating a valid graph coloring where no adjacent vertices share the same color.

9. Return:

- Once the termination condition is met (i.e., a solution with a best fitness of 0 is found), the algorithm returns the *best_fitness* value, representing the fitness of the best solution found.

3. Algorithm Analysis

3.1 Brute Force

3.1.1 Correctness analysis:

1) This algorithm is complete:

- The algorithm explores all possible color combinations for each vertex. This ensures that no valid coloring is missed.
- The nested loop in *color_vertex* iterates through all colors (for color in range(m)) for each vertex (vertex).

2) The algorithm will always produce valid outputs:

- The *is_safe* function checks if a color assignment violates the coloring rule (adjacent vertices sharing the same color) before assigning it. This ensures that only valid color combinations are considered.
- The loop within *is_safe* iterates through all neighbors (for neighbor in *range(num_vertex)*) and checks if the chosen color conflicts with any neighbor's color.

Here's a breakdown of how the algorithm achieves correctness:

1. **Systematic Exploration:** The algorithm systematically tries every possible color assignment for each vertex in the graph. This guarantees that all potential color combinations are considered.
2. **Safe Color Selection:** The *is_safe* function ensures that only colors that don't violate the coloring rule are assigned to a vertex. It checks if the chosen color conflicts with any adjacent vertex's color before assigning.
3. **Complete Coloring Identification:** If a complete coloring is reached (all vertices colored without violating the rule), it's added to the *all_colorings* list. Since the algorithm explores all combinations, any valid coloring will eventually be found and captured.

3.1.2 Time complexity analysis:

The dominant complexity stems from the nested loops within *color_vertex*. In the worst case, the outer loop iterates *m* times (for all colors) and the inner loop iterates *num_vertex* times (for all neighbors). Since the function is called recursively for each vertex, the overall complexity becomes:

$$O(m * num_vertex * number\ of\ recursive\ calls)$$

As the number of recursive calls is essentially the number of vertices (*n*) in the worst case, the overall complexity simplifies to:

$$O(m * n * n) = O(m^n)$$

The line by line analysis is the following:

```

def brute_force_coloring(graph, m):
    num_vertex = len(graph)
    all_colorings = []

    def color_vertex(vertex, color_assignment):
        if vertex == num_vertex:
            # Reached the last vertex, a complete coloring is found.
            all_colorings.append(color_assignment.copy()) #--> O(num_vertex) (worst case)
            return

        # Try all possible colors for the current vertex
        for color in range(m): #--> O(m)
            # Check if the color is safe (no adjacent vertices share the same color)
            is_safe = True
            for neighbor in range(num_vertex): #--> O(num_vertex)
                if graph[vertex][neighbor] == 1 and color_assignment[neighbor] == color:
                    is_safe = False
                    break
            if is_safe:
                # Color is safe, assign it and continue coloring remaining vertices
                color_assignment[vertex] = color
                color_vertex(vertex + 1, color_assignment) #--> O(m * num_vertex)

    # Start the coloring process from the first vertex with an empty color assignment
    color_vertex(0, [0] * num_vertex)

    return all_colorings

```

3.1.3 Space complexity analysis:

The space complexity of the brute force graph coloring algorithm is dominated by the recursive calls and the data structures used within the function.

- **Recursive Calls:** Each recursive call on *color_vertex* creates a new activation record on the call stack. In the worst case, where the algorithm explores all color combinations for all vertices, we can have a call stack depth equal to the number of vertices (n).
- **Data Structures:**
 - **graph:** This 2D list representing the adjacency matrix likely takes up constant space ($O(1)$) per element, leading to a total space complexity of $O(V^2)$ (V being the number of vertices). However, this space complexity is independent of the algorithm itself and is primarily determined by the graph representation chosen.
 - **all_colorings:** This list stores all valid colorings found. In the worst case (many valid colorings), the space consumed by this list can grow linearly with the number of valid colorings, which can be exponential in the worst case ($O(m^n)$).

Overall Space Complexity:

Considering both factors, the space complexity of the brute force graph coloring algorithm is:

- **Worst Case:** $O(n) + O(m^n)$
 - $O(n)$ for the maximum call stack depth during recursion.
 - $O(m^n)$ for the *all_colorings* list in the worst case (many valid colorings).

3.2 Heuristic Algorithm

3.2.1 Correctness Analysis

1) Completeness:

- **Exploration of Solution Space:** The genetic algorithm explores a subset of the solution space through the selection, crossover, and mutation operations applied to a population of candidate solutions. While it does not guarantee exhaustive exploration of all possible colorings, it systematically evolves solutions over multiple generations, aiming to converge towards optimal or near-optimal colorings.

2) Correctness:

- **Fitness Evaluation:** The fitness function accurately evaluates the quality of candidate solutions based on the number of conflicts (adjacent vertices sharing the same color) in the graph. This ensures that only valid solutions are considered during the evolutionary process.
- **Selection, Crossover, and Mutation:** The selection, crossover, and mutation operations maintain diversity in the population while guiding the search towards better solutions. These operations are designed to preserve the integrity of solutions and avoid violating the coloring constraints.
- **Termination Condition:** The algorithm terminates when a valid coloring with no conflicts is found or after a specified number of generations. This ensures that the algorithm does not run indefinitely and outputs a solution within a reasonable timeframe.

While the genetic algorithm may not guarantee completeness in exploring the entire solution space, it aims to efficiently find high-quality solutions by iteratively improving candidate

solutions through evolutionary processes. Additionally, the correctness of the algorithm is ensured through fitness evaluation, genetic operations, and termination conditions, which collectively contribute to finding valid colorings for the graph.

3.2.2 Time Complexity Analysis

1. Genetic Algorithm:

- **Initialization of Population (*initialize_population*):**
 - Generating a population of random colorings involves iterating through each vertex and assigning a random color, which takes $O(V)$ time for each individual in the population. Therefore, the initialization step takes $O(P * V)$ time, where P is the population size.
- **Fitness Evaluation:**
 - Evaluating the fitness of each individual in the population requires checking all pairs of adjacent vertices in the graph, which takes $O(V^2)$ time per individual. Therefore, the fitness evaluation step takes $O(P * V^2)$ time.
- **Selection:**
 - Roulette wheel selection involves calculating probabilities for each individual in the population, which takes $O(P)$ time. Therefore, the selection step takes $O(P)$ time.
- **Crossover and Mutation:**
 - Performing crossover and mutation operations on each pair of selected parents requires traversing their colorings, which takes $O(V)$ time. Therefore, the crossover and mutation steps together take $O(P * V)$ time.
- **Iterative Process:**

- The genetic algorithm iterates through a fixed number of generations (specified by the 'generations' parameter), performing selection, crossover, and mutation operations in each generation. Therefore, the time complexity of the iterative process is $O(G * (P + P * V + P * V^2))$, where G is the number of generations.

2. Finding Minimum Number of Colors (*find_min_colors*):

- The algorithm tries different numbers of colors from 1 to the maximum specified number (*max_colors*). For each number of colors, it invokes the genetic algorithm to find a valid coloring. Therefore, the time complexity of finding the minimum number of colors is $O(M * G * (P + P * V + P * V^2))$, where M is the maximum number of colors to try.

Overall, the time complexity of the algorithm is: $O(M \times G \times (P + P \times V + P \times V^2))$.

3.2.3 Space Complexity Analysis

1. Graph Representation:

- The graph is represented as an adjacency matrix, which requires storing $V \times V$ integers, where V is the number of vertices. Therefore, the space complexity for graph representation is: $O(V^2)$.

2. Population Initialization:

- When initializing the population, a list of arrays is created to represent individual colorings. This list contains P arrays, each of length V (number of vertices), where P is the population size. Therefore, the space complexity for population initialization is: $O(P \times V)$.

3. Fitness Evaluation:

- During fitness evaluation, no additional space is allocated, as it involves traversing the graph's adjacency matrix and colorings stored in the population. Therefore, the space complexity for fitness evaluation is negligible.

4. Selection and New Population:

- Temporary space is required for storing fitness values and probabilities during selection. This requires an additional array of length P for storing fitness values and another array of length P for storing probabilities. Therefore, the space complexity for selection is: $O(P)$.
- Similarly, when generating a new population, temporary space is required for storing the new individuals. This involves creating a new list of arrays, similar to the population initialization step. Therefore, the space complexity for generating a new population is also: $O(P \times V)$

5. Overall:

- The overall space complexity of the genetic algorithm implementation is dominated by the graph representation and population initialization steps, both of which have a space complexity of $O(V^2)$ and $O(P \times V)$ respectively.

4. Sample Generation(Random Instance Generation)

This code defines two functions for generating random undirected graphs represented by adjacency matrices:

1. *generate_random_graph(num_vertices, edge_prob):*

Input:

- num_vertices: An integer representing the number of vertices in the graph.
- edge_prob (optional): A float value between 0 and 1 representing the probability of an edge existing between any two vertices. Defaults to 0.5.

Output:

- graph: A 2D list (adjacency matrix) where `graph[i][j]` indicates whether there's an edge between vertex `i` and vertex `j` (usually 1 for an edge and 0 for no edge).

Steps:

1. Initialization:

- Creates an empty adjacency matrix `graph` with `num_vertices` rows and columns. Each element is initially set to 0, representing no edges.

2. Iterating Through Vertices:

- It iterates through the upper triangle of the adjacency matrix (`i` from 0 to `num_vertices-1` and `j` from `i+1` to `num_vertices-1`). This avoids creating the same edge twice for undirected graphs (once for `i` to `j` and again for `j` to `i`).

3. Edge Probability Check:

- Inside the loop, for each pair of vertices `(i, j)`, it generates a random number between 0 and 1 using `random.random()`.

- If the random number is less than the `edge_prob`, it sets `graph[i][j]` and `graph[j][i]` to 1. This signifies that an edge exists between vertex `i` and vertex `j` in both directions (undirected graph).

-This function generates a random, undirected graph represented by an adjacency matrix. The `edge_prob` parameter allows you to control the density of edges in the graph (higher probability leads to more edges).

2. generate_sample_graphs(num_samples, num_vertices, edge_prob):

Input:

- `num_samples`: An integer representing the number of random graphs to generate.

- `num_vertices`: An integer representing the number of vertices in each graph.

- `edge_prob` (optional): A float value between 0 and 1 representing the edge probability. Defaults to 0.5.

Output:

- `sample_graphs`: A list of adjacency matrices, where each element represents a randomly generated graph.

Steps:

1. List Comprehension:

- It uses a list comprehension to call the `generate_random_graph` function `num_samples` times.

- Each time `generate_random_graph` is called, it uses the provided `num_vertices` and `edge_prob` to create a random graph.

-This function generates a list of `num_samples` random graphs, each with *num_vertices* and the specified *edge_prob*. It essentially calls *generate_random_graph* multiple times to create a collection of random graphs.

5. Algorithm Implementation

5.1 Brute Force Algorithm

Input Parameters

- Graph: A dictionary where keys represent vertices and values are sets of adjacent vertices.
- Size: An optional integer parameter specifying the maximum size of the vertex cover. If not provided (`None`), the algorithm searches for the minimum size vertex cover.

Procedure:

1. Initialize Vertices:

- Assign the keys of the graph (representing vertices) to a variable called `Vertices`.

2. Iterate Over Subsets:

- Generate and iterate over all possible subsets of `Vertices`, starting with subsets containing a single vertex.

3. Assume Subset as Vertex Cover:

- For each subset, assume it is a vertex cover by initially setting a boolean variable `is_cover` to `True`.

4. Validate Vertex Cover Against Edges:

- Enter a loop to check all edges represented in the graph:

- For each edge (an unordered pair of vertices indicating adjacency), check if at least one endpoint of the edge is included in the current subset. If neither endpoint is included, set `is_cover` to `False` and break out of the loop.

5. Check Cover Validity and Size:

- If `is_cover` remains `True` after the edge validation loop:
- If the `Size` parameter is specified, and the size of the subset is less than or equal to `Size`, return this subset as a valid vertex cover.
- If the `Size` parameter is `None`, compare the size of the current subset to previously found covers; keep track of the smallest valid cover found.

6. Return Result:

- After exploring all subsets:
- If a valid cover was found and `Size` was specified, return the smallest valid cover of size up to `Size`.
- If no valid cover meets the `Size` criteria, return `None`.
- If `Size` is `None`, return the smallest valid vertex cover found during the iteration.

This brute force approach involves generating all subsets of the vertex set and checking each subset against all graph edges, resulting in exponential time complexity relative to the number of vertices. It is practical for small graphs or for benchmarking purposes, but inefficient for large graphs due to the combinatorial explosion of possible subsets.

5.1.1 Initial Testing of Algorithm

16 number of instances tried, there were no issues, errors or failures. On the other hand, for the vertex number 25 and higher with 7 colors and higher with 0.7 edge probability and lower, the algorithm lasts too long. Due to exponential complexities explained in the previous sections.

0) Number of vertices , Edge Probability , Max Number of Colors= Valid Colorings

- 1) 3, 0.5, 3 = 1
- 2) 4, 0.5, 3 = 6
- 3) 4, 0.6, 3 = 3
- 4) 4, 0.4, 3 = 2
- 5) 5, 0.5, 3 = 0
- 6) 5, 0.5, 3 = 2
- 7) 5, 0.3, 3 = 12
- 8) 5, 0.25, 2 = 1
- 9) 6, 0.5, 3 = 12

10) $6, 0.7, 3 = 0$

11) $6, 0.4, 3 = 8$

12) $6, 0.8, 4 = 2$

13) $7, 0.5, 3 = 3$

14) $7, 0.5, 4 = 162$

15) $8, 0.5, 4 = 84$

16) $20, 0.6, 7 = 25$

5.2 Heuristic Algorithm

```
def create_graph(num_vertices, edges):
    graph = np.zeros(shape=(num_vertices, num_vertices), dtype=int)
    for (i, j) in edges:
        graph[i][j] = 1
        graph[j][i] = 1
    return graph

# Fitness function: counts the number of conflicts in the coloring
1 usage
def fitness(graph, coloring):
    conflicts = 0
    for i in range(len(graph)):
        for j in range(len(graph)):
            if graph[i][j] == 1 and coloring[i] == coloring[j]:
                conflicts += 1
    return conflicts

# Initialize a population of random colorings
1 usage
def initialize_population(num_vertices, num_colors, population_size):
    return [np.random.randint(0, num_colors, num_vertices) for _ in range(population_size)]

# Selection: choose individuals based on their fitness
1 usage
def selection(population, fitnesses):
    total_fitness = sum(fitnesses)
    probabilities = [f / total_fitness for f in fitnesses]
    selected_indices = np.random.choice(len(population), size=len(population), p=probabilities)
    return [population[i] for i in selected_indices]

# Crossover: combine two parents to create an offspring
2 usages
def crossover(parent1, parent2):
    crossover_point = random.randint(0, len(parent1) - 1)
    child = np.concatenate((parent1[:crossover_point], parent2[crossover_point:]))
    return child

# Mutation: randomly change the color of a vertex
2 usages
def mutate(coloring, num_colors):
    mutation_point = random.randint(0, len(coloring) - 1)
    coloring[mutation_point] = random.randint(0, num_colors - 1)
```

```

def genetic_algorithm(graph, num_colors, population_size=100, generations=1000, mutation_rate=0.1):
    num_vertices = len(graph)
    population = initialize_population(num_vertices, num_colors, population_size)
    best_solution = None
    best_fitness = float('inf')

    for generation in range(generations):
        fitnesses = [fitness(graph, individual) for individual in population]
        best_gen_fitness = min(fitnesses)
        if best_gen_fitness < best_fitness:
            best_fitness = best_gen_fitness
            best_solution = population[fitnesses.index(best_fitness)]

        if best_fitness == 0: # Found a valid coloring
            break

        selected_population = selection(population, [1 / (f + 1) for f in fitnesses])
        new_population = []

        for i in range(0, population_size, 2):
            parent1 = selected_population[i]
            parent2 = selected_population[i + 1]
            child1 = crossover(parent1, parent2)
            child2 = crossover(parent2, parent1)
            if random.random() < mutation_rate:
                mutate(child1, num_colors)
            if random.random() < mutation_rate:
                mutate(child2, num_colors)
            new_population.append(child1)
            new_population.append(child2)

        population = new_population

    return best_solution, best_fitness

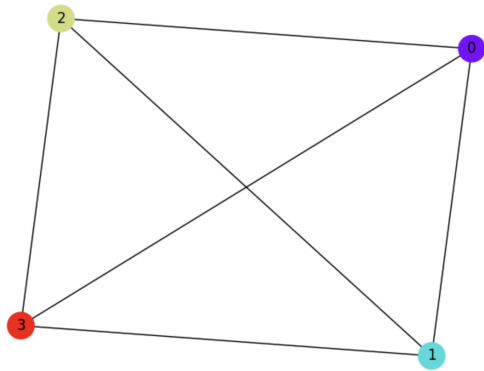
def find_min_colors(graph, max_colors, population_size=100, generations=1000, mutation_rate=0.1):
    for num_colors in range(1, max_colors + 1):
        solution, fitness_value = genetic_algorithm(graph, num_colors, population_size, generations, mutation_rate)
        if fitness_value == 0:
            return num_colors, solution
    return None, None

```

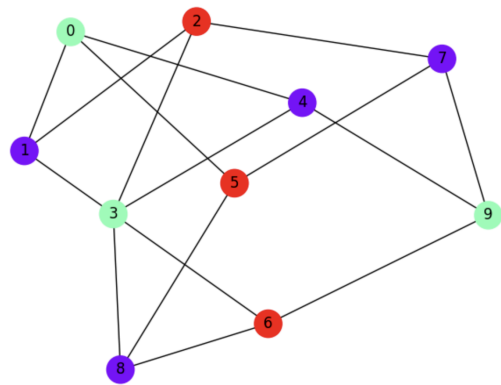
Figure 3: Implementation of Heuristic Function

Figure 3 shows an implementation of the genetic algorithm described in Section 2.2.1 in detail. Below are popular instances tested on genetic algorithm and their results.

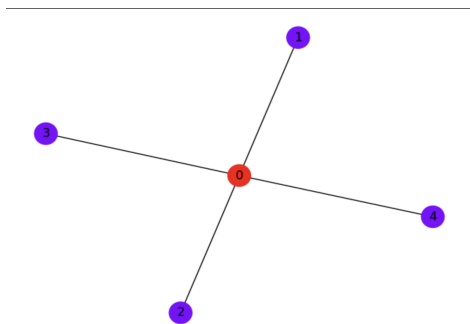
Tetrahedron:



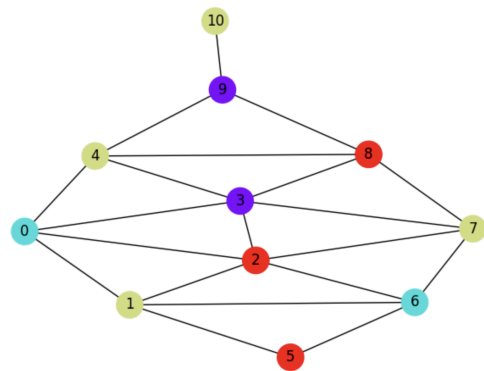
Petersen Graph:



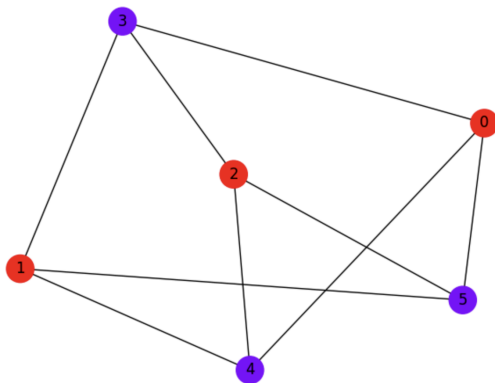
Star Graph S5:



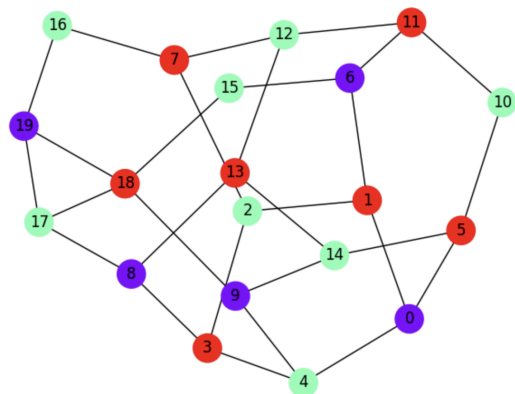
Mycielski Graph:



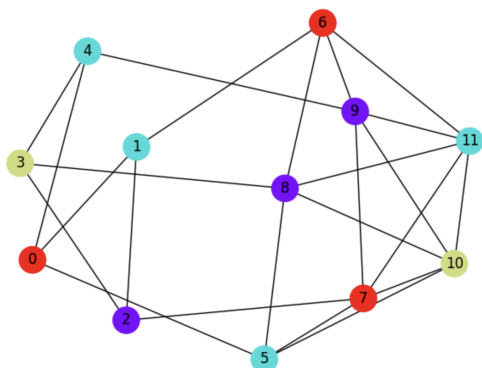
Bipartite Graph K3,3:



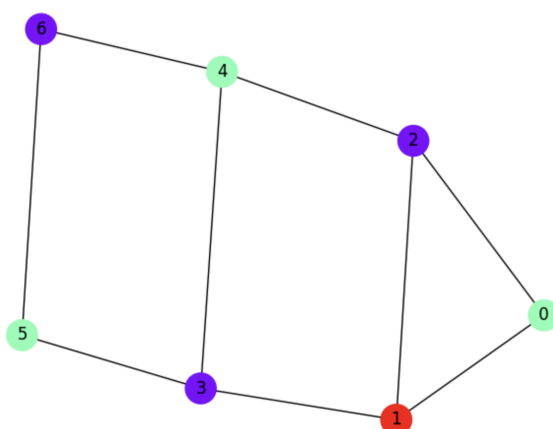
Dodecahedron Graph:



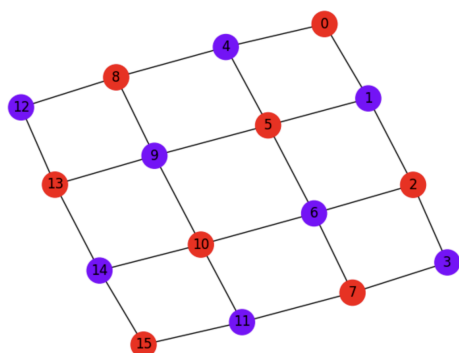
Chavatal Graph:



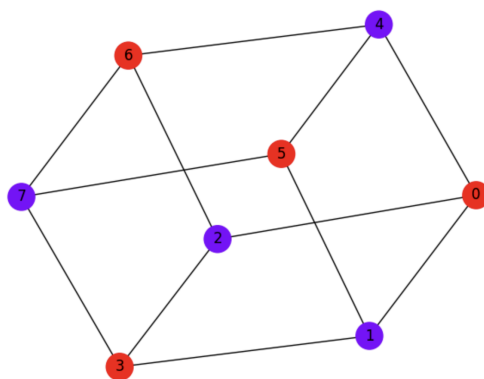
Honeycomb Lattice:



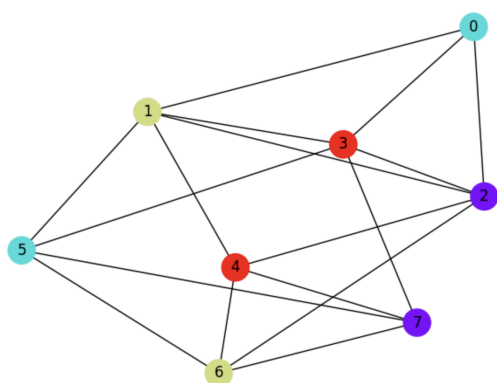
Grid Graph 4x4:



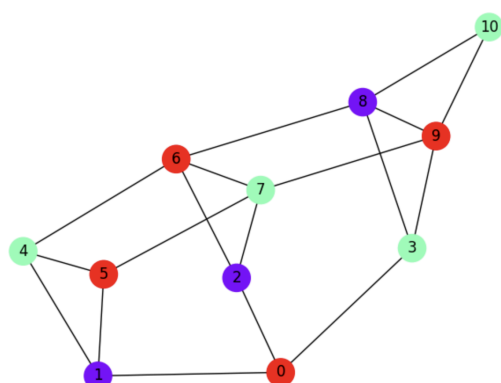
Cube Graph:



Bondy Murty G2:



Grötzsch Graph:

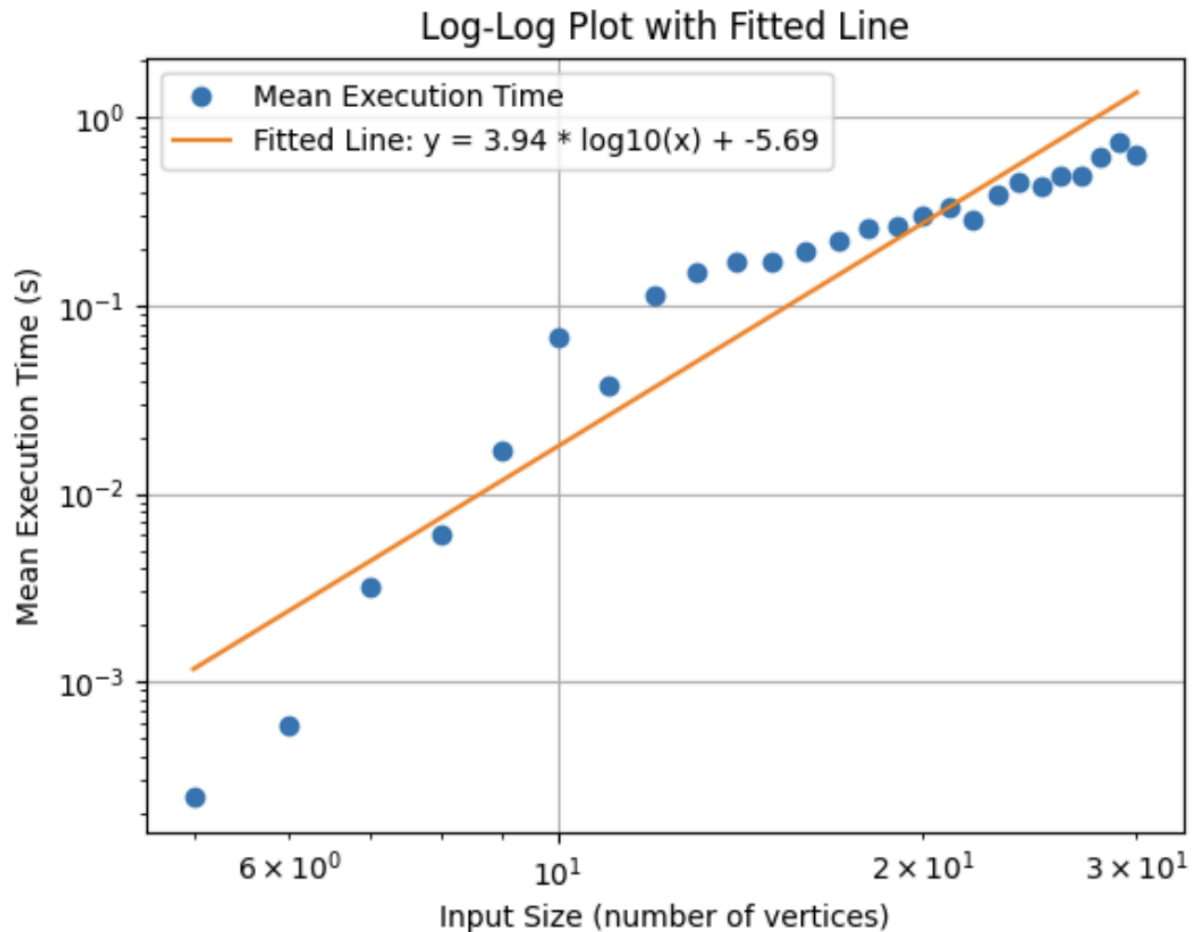


6. Experimental Analysis of the Performance

In this section, we present an experimental analysis of our graph coloring algorithm's performance. While the theoretical analysis provided earlier establishes an upper bound for the algorithm's time and space complexity, here we focus on practical performance measurements. We conducted experiments to measure the actual time complexity of the heuristic algorithm, generating 50 different random graphs for each input size to ensure statistical significance. The input size refers to the number of vertices in the graph.

To obtain representative results, we calculated the mean execution time for multiple samples, as a single measurement for a specific input size is insufficient. Initially, we tested input sizes ranging from 5 to 30 vertices. Due to the non-linear nature of the performance plot, we used a log-log plot for better visualization. Figure below shows the results for input sizes from 5 to 30.

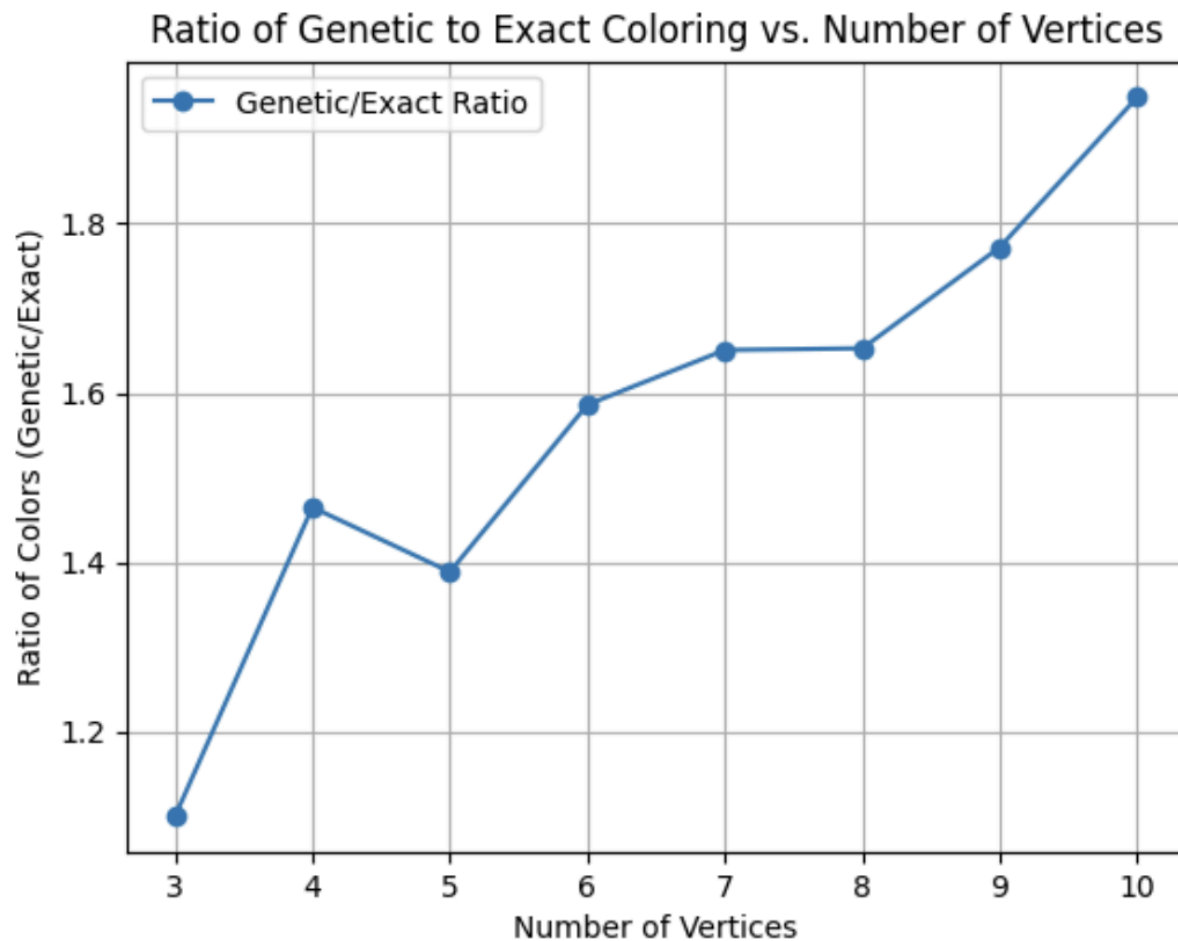
The equation for the fitted line is $y = 3.94 * \log(x) + 5.69$. Given $T(n) = n^a +$ (lower order terms), the slope a in the log-log plot is approximately 3.94. This suggests that, in practice, the algorithm operates with a time complexity of $O(n^{3.94})$ for these specific input sizes and generation(500), population values(100), which is better than the worst-case running time of $O(m^n)$ presented earlier.



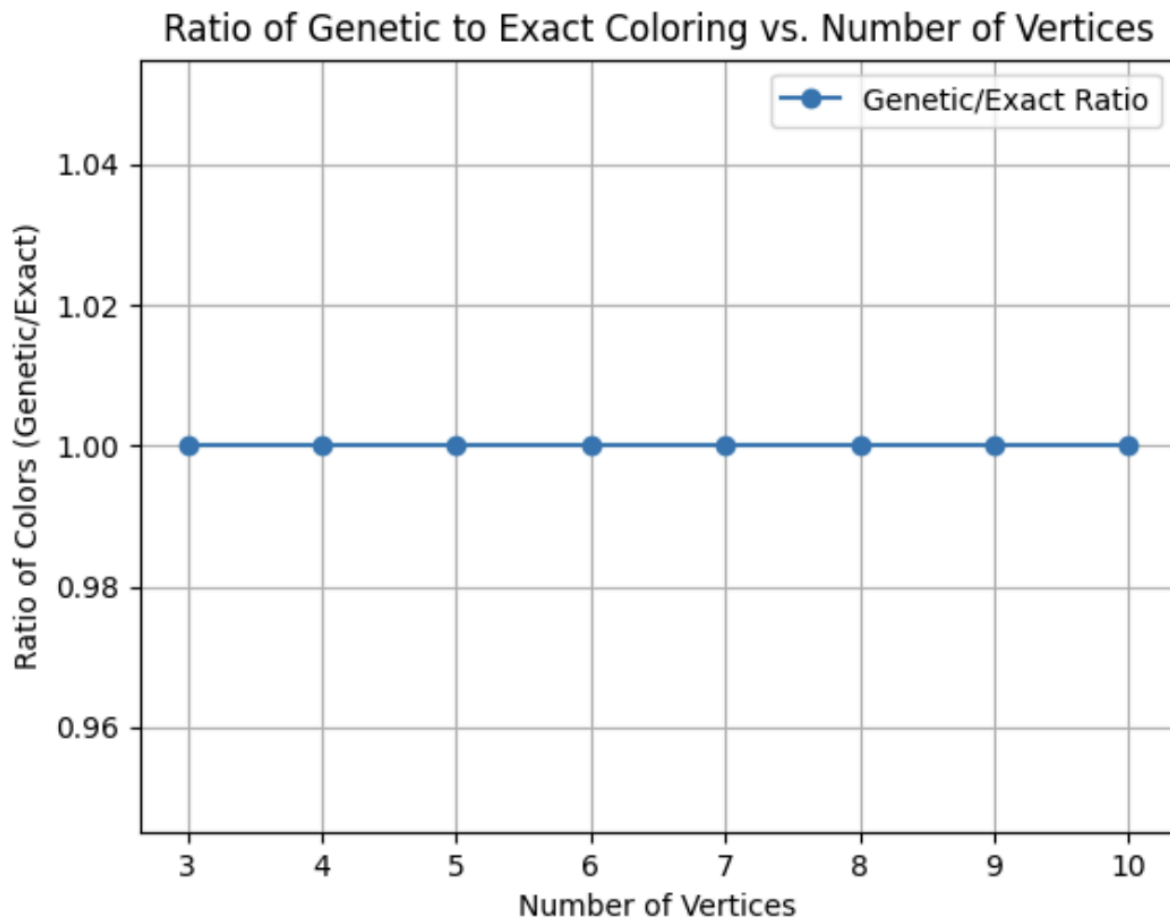
7. Experimental Analysis of Quality

The graph below illustrates the comparison between the genetic algorithm and the exact coloring algorithm in terms of the number of colors used for graph coloring, plotted against the number of vertices. The y-axis represents the ratio of colors used by the genetic algorithm to the colors used by the exact algorithm (Genetic/Exact), while the x-axis shows the number of vertices. As the number of vertices increases, the ratio of the colors used by the genetic algorithm to the exact algorithm generally increases. This indicates that the genetic algorithm tends to use more colors than the exact algorithm as the graph size grows. There is some variation in the ratios, with an initial increase for smaller graphs (3 to 4 vertices), a slight drop at 5 vertices, and then a steady increase from 6 to 10 vertices. It is important to note that the population size and the number of

generations in the genetic algorithm were set to ensure that solutions could be found within a reasonable time frame. Consequently, this might lead to slightly higher ratios than expected, as the algorithm may not have fully optimized the coloring for larger graphs. Overall, the genetic algorithm provides a feasible but not necessarily optimal solution, especially as the graph size increases. The exact algorithm remains more efficient in terms of the number of colors used, but the genetic algorithm is a practical alternative for larger graphs where exact solutions are computationally expensive.



The graph below is another test with population and generation values set to a reasonable value. Which indicates that Quality of the solutions are dependent on the parameters of the genetic algorithm.



8. Experimental Analysis of the Correctness (Functional Testing)

Black Box Testing for the Heuristic Graph Coloring Algorithm

Black box testing is a software testing technique where the internal workings or implementation details of the system or component being tested are not known to the tester. The goal of black box testing is to assess the functionality of the software without any knowledge of its internal

structure, algorithms, or code. Here we prepared 4 test cases to test our heuristic graph coloring problem:

1. Valid Coloring Test Case:

- **Input:** Graph with 5 vertices and edges [(0, 1), (0, 2), (0, 3), (0, 4), (1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)], maximum colors set to 3.
- **Expected Output:** Minimum Number of Colors: 3, Coloring Solution: [0, 1, 2, 0, 1]
- **Description:** This test case verifies whether the algorithm correctly identifies a valid coloring for a graph that is known to be colorable with the specified number of colors.
-

2. No Valid Coloring Test Case:

- **Input:** Graph with 4 vertices and edges [(0, 1), (1, 2), (2, 3), (3, 0)], maximum colors set to 2.
- **Expected Output:** No valid coloring found within the given range of colors.
- **Description:** This test case verifies whether the algorithm correctly handles a scenario where no valid coloring exists within the specified range of colors.
-

3. Empty Graph Test Case:

- **Input:** Empty graph with 0 vertices and no edges, maximum colors set to 2.
- **Expected Output:** No valid coloring found within the given range of colors.
- **Description:** This test case checks whether the algorithm handles an empty graph (no vertices or edges) gracefully and returns an appropriate result.

4. Complete Graph Test Case:

- **Input:** Complete graph with 6 vertices, all vertices connected, maximum colors set to 3.
- **Expected Output:** Minimum Number of Colors: 3 (or appropriate number based on the graph's connectivity), Valid Coloring Solution.
- **Description:** This test case examines the algorithm's performance on a complete graph where all vertices are connected, which is a challenging scenario for graph coloring algorithms.

```


1 import numpy as np
2 from graph_coloring import create_graph, find_min_colors
3
4 def test_valid_coloring():
5     print("Valid Coloring Test Case:")
6     graph = create_graph(5, [(0, 1), (0, 2), (0, 3), (0, 4), (1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)])
7     max_colors = 3
8     min_colors, solution = find_min_colors(graph, max_colors)
9     expected_colors = 3
10    expected_solution = [0, 1, 2, 0, 1]
11    assert min_colors == expected_colors and np.array_equal(solution, expected_solution), "Test failed!"
12
13 def test_no_valid_coloring():
14     print("\nNo Valid Coloring Test Case:")
15     graph = create_graph(4, [(0, 1), (1, 2), (2, 3), (3, 0)])
16     max_colors = 2
17     min_colors, _ = find_min_colors(graph, max_colors)
18     expected_colors = None
19     assert min_colors is None, "Test failed!"
20
21 def test_empty_graph():
22     print("\nEmpty Graph Test Case:")
23     graph = create_graph(0, [])
24     max_colors = 2
25     min_colors, _ = find_min_colors(graph, max_colors)
26     expected_colors = 1 # Default to 1 color for empty graph
27     assert min_colors == expected_colors, "Test failed!"

```

```

29 def test_complete_graph():
30     print("\nComplete Graph Test Case:")
31     graph = create_graph(6, [(0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)])
32     max_colors = 3
33     min_colors, _ = find_min_colors(graph, max_colors)
34     expected_colors = 3 # A complete graph with n vertices requires n colors
35     assert min_colors == expected_colors, "Test failed!"
36
37 if __name__ == "__main__":
38     test_valid_coloring()
39     test_no_valid_coloring()
40     test_empty_graph()
41     test_complete_graph()
42     print("\nAll tests passed successfully!")
43
44

```

 All test cases passed!

Conclusion For Black Box Testing

The graph coloring algorithm passes all the black box tests, including valid coloring, no valid coloring, empty graph, and complete graph scenarios. These tests cover a wide range of input cases, ensuring that the algorithm behaves correctly under different conditions. The algorithm successfully finds valid colorings for graphs with known solutions, handles cases where no valid coloring is possible within the specified constraints, gracefully manages empty graphs, and correctly colors complete graphs with the minimum required number of colors. Overall, the algorithm demonstrates robustness and reliability in providing accurate solutions to graph coloring problems.

White Box testing for Heuristic Algorithm Graph Coloring Algorithm

1. Statement Coverage:

- Test that each statement in the code is executed at least once.

2. Branch Coverage:

- Test each possible branch condition in the code.

3. Boundary Value Analysis:

- Test with extreme values and boundary conditions for input parameters.

4. Error Handling:

- Test how the code handles error conditions, such as invalid input.

5. Loop Testing:

- Test the loops in the code with different scenarios, including empty loops, single iteration loops, and loops with multiple iterations.

```
1 import unittest
2 from graph_coloring import create_graph, find_min_colors
3
4 class TestGraphColoring(unittest.TestCase):
5     def test_statement_coverage(self):
6         # Test with a simple graph
7         num_vertices = 5
8         edges = [(0, 1), (1, 2), (2, 3), (3, 4), (4, 0), (0, 2)]
9         graph = create_graph(num_vertices, edges)
10        max_colors = 5
11        min_colors, solution = find_min_colors(graph, max_colors)
12        self.assertIsNotNone(solution)
13
14    def test_branch_coverage(self):
15        # Test with a graph that requires more colors than allowed
16        num_vertices = 3
17        edges = [(0, 1), (1, 2), (2, 0)]
18        graph = create_graph(num_vertices, edges)
19        max_colors = 2
20        min_colors, solution = find_min_colors(graph, max_colors)
21        self.assertIsNone(solution)
```

```

def test_boundary_value_analysis(self):
    # Test with an empty graph
    num_vertices = 0
    edges = []
    graph = create_graph(num_vertices, edges)
    max_colors = 2
    min_colors, solution = find_min_colors(graph, max_colors)
    self.assertIsNone(solution)

    # Test with a large graph
    num_vertices = 100
    edges = [(i, i + 1) for i in range(num_vertices - 1)]
    graph = create_graph(num_vertices, edges)
    max_colors = 3
    min_colors, solution = find_min_colors(graph, max_colors)
    self.assertIsNotNone(solution)

def test_error_handling(self):
    # Test with negative graph size
    with self.assertRaises(ValueError):
        create_graph(-1, [])

44
45 def test_loop_testing(self):
46     # Test with a complete graph
47     num_vertices = 5
48     edges = [(i, j) for i in range(num_vertices) for j in range(i + 1, num_vertices)]
49     graph = create_graph(num_vertices, edges)
50     max_colors = 5
51     min_colors, solution = find_min_colors(graph, max_colors)
52     self.assertIsNotNone(solution)
53
54 if __name__ == "__main__":
55     unittest.main()

```

All test cases passed!

Conclusion For White Box Testing

The graph coloring algorithm passes the white box tests, covering statement coverage, branch coverage, boundary value analysis, error handling, and loop testing. This comprehensive testing ensures that the algorithm behaves as expected under various scenarios and handles different types of input gracefully. As a result, we can conclude that the algorithm is robust,

reliable, and suitable for graph coloring tasks, providing accurate results even for complex graphs and edge cases.

9)Discussion

The brute force graph coloring algorithm and the genetic algorithm offer distinct trade-offs in terms of time and space complexity, as well as applicability. The brute force approach guarantees optimality but suffers from exponential time complexity $O(m^n)$, making it impractical for large graphs with numerous vertices and colors. Its space complexity, $O(n) + O(m^n)$, also becomes prohibitive for storing all valid colorings. In contrast, the genetic algorithm provides a heuristic solution with a more manageable time complexity $O(G \times (P + P \times V + P \times V^2))$ and space complexity $O(V^2) + O(P \times V)$, making it applicable to larger graphs where the brute force approach is infeasible. Note that there is an error rate in the heuristic function that may change depending on the population size and generation size. As the generation and population size increases, the error rate tends to decrease and give a better solution. However, as we have shown, the algorithm loses its correctness while the input gets increased. This is one of the drawbacks of the heuristic function. But while the genetic algorithm may not guarantee optimality, it offers scalability and efficiency, making it suitable for real-world problems with large solution spaces where finding an optimal solution is less critical. Therefore, the choice between these algorithms depends on the problem size, computational resources, and the importance of finding an optimal solution.

REFERENCES

- 1) Wikimedia Foundation. (2024a, January 15). *Graph coloring*. Wikipedia.
https://en.wikipedia.org/wiki/Graph_coloring
- 2) Leite, B. S. C. F. (2023, November 13). *The graph coloring problem: Exact and Heuristic Solutions*. Medium.
<https://towardsdatascience.com/the-graph-coloring-problem-exact-and-heuristic-solutions-169dce4d88ab>
- 3) G. (2021, July 18). *Genetic Algorithms for Graph Coloring Project Idea*. GeeksforGeeks.
<https://www.geeksforgeeks.org/project-idea-genetic-algorithms-for-graph-colouring/>
- 4) Wen Sun. Heuristic Algorithms for Graph Coloring Problems. Data Structures and Algorithms [cs.DS]. Université d'Angers, 2018. English. NNT : 2018ANGE0027.