كلية علوم الحاسب والمعلومات
قسم هندسة البرمجيات

King Saud University

College of Computer and Information Sciences

Department of Software Engineering

SWE 485 - Selected Topics in Software Engineering

MAP COLORING PROBLEM

Phase 2

#	Name	ID
1	Fay Alshubaili	442200415
2	Saadiya Abdulqader	442204801
3	Noura Alghamlas	442200410
4	Amani Aldossari	442200890
5	Sarah Alsaleh	442200150

Section #	60120
Group #	2
Supervisor(s)	Dr. Khoula hamdi

Submission Date: 31 Mar 2024

Table of Contents

كلية علوم الحاسب والمعلومات
قسم هندسة البرمجيات

Phase 1

Problem definition and formulation

Introduction

Map coloring problem

The Map Coloring Problem is a classic combinatorial problem in graph theory and computer science. It involves assigning colors to regions on a map in such a way that no two adjacent regions share the same color. The primary objective is to find a valid coloring for the map while adhering to this constraint.. Map coloring problem states that given a graph G {V, E} where V and E are the set of vertices and edges of the graph, all vertices in V need to be colored in such a way that no two adjacent vertices must have the same color. [1]
Real-World Applications of the coloring problem:

- Cartography: In creating maps for geographical regions, ensuring neighboring areas have distinct colors aids in visual clarity.
- Scheduling: Regions can represent tasks or events, and coloring helps optimize scheduling to prevent conflicts.
- Frequency Allocation: In wireless communication, the assignment of frequencies to adjacent regions without interference aligns with the map coloring concept.

Objective

The objective of this report is to employ Constraint Satisfaction Problem (CSP) formalization to solve the Map Coloring Problem. By defining the variables and domains, constraints, and objective function as a first step. this report aims to define the needed components to create a mathematical representation of the problem that is amenable to the application of search algorithms.

Problem definition

key elements of a Constraint Satisfaction Problem (CSP):

1. **Variables**: A set of regions that will be colored (X).

In the map coloring problem, each region on the map is treated as a variable. Let's assume we have a map with R_n regions, each of which needs to be colored. We can denote these regions (variables) as $R_1, R_2, R_3, \dots, R_n$

2. **Domains**:
   - Definition: Domains define the possible values that each variable can take. It represents the feasible options or potential assignments for the variables.

The domain for each variable is the set of colors that can be assigned to each region. Assuming we have a set of C_m colors available, we can represent the domain of colors as $C=\{c_1,c_2,c_3,\dots,c_m\}$.

3. **Constraints**:
   - Definition: Constraints define the relationships, limitations, or conditions that must be satisfied by the variable assignments to be considered valid solutions.
   - Constraints in the Map Coloring Problem refer to the rules that must be followed when assigning colors to regions on a map. In this context, constraints ensure that no two adjacent regions share the same color. So If R_i and R_j share a common border then the constraints would be $C_i \neq C_j$

4. **Objective function:**
   - Definition: In the context of the search algorithms, the objective function is the function we aim to maximize or minimize, it could represent the cost of the solution, the measure of fitness, or the overall quality of a candidate solution within the search space.[2]
   - Map coloring problem objective function: The objective of the Map Coloring Problem is to find a valid assignment of colors to regions that satisfies all constraints. There might not be a numerical objective function in the traditional sense, but the goal is to find a solution that adheres to the constraints.

Example

Since the map coloring problem is a general problem and it doesn't have a single scenario, we considered providing an example to better illustrate each element in the CSP :
Let's consider a simple example with four regions: R1,R2,R3,R4, and three colors: C1,C2,C3.

Variables and Domains: In this example, each of the four regions is a variable that we need to assign a color to. Therefore, the variables are**:** R1,R2,R3,R4**.**
The domain for each variable is the set of colors that can be assigned to it. Since we have three colors available (*C*1, *C*2,*C*3), the domain for each of the regions is the same, consisting of these three colors.
Hence, the domains for the variables are as follows:
Domain for *R*1: D (R1)={ *C*1,*C*2,*C*3**}**
Domain for *R*2: D (R2) ={ *C*1,*C*2,*C*3**}**
Domain for *R*3: D (R3) ={ *C*1,*C*2,*C*3**}**
Domain for *R*4: D (R4) ={ *C*1,*C*2,*C*3**}**

Constraints:

In the context of the Map Coloring Problem, the main constraint is that for every pair of adjacent regions, the colors assigned to them must be different. This constraint can be expressed as follows:
If there is an edge between two regions Ri and Rj (represented as (Ri, Rj) in the graph), then the colors assigned to Ri and Rj, denoted as Ci and Cj, must be different. This constraint can be formally stated as Ci ≠ Cj.
For the example with four regions (R1, R2, R3, R4) and three colors (C1, C2, C3), the constraints can be specified as:
- R1 and R2 are adjacent: C(R1) ≠ C(R2)
- R1 and R3 are adjacent: C(R1) ≠ C(R3)
- R1 and R4 are adjacent: C(R1) ≠ C(R4)
- R2 and R3 are adjacent: C(R2) ≠ C(R3)
- R2 and R4 are adjacent: C(R2) ≠ C(R4)
- R3 and R4 are adjacent: C(R3) ≠ C(R4)

Objective Function:

- Find a valid assignment of colors to R1,R2,R3,R4 that satisfies all the constraints.

جامعة الملك سعود
King Saud University

كلية علوم الحاسب والمعلومات
قسم هندسة البرمجيات

Phase 2

Incremental formulation

Used heuristic

In the context of the map coloring problem, estimating the cost to reach a goal state where the goal is to find the minimum number of colors needed to color a map such that no two adjacent regions share the same color—requires a tailored heuristic approach. One commonly used heuristic for this problem is the concept of "chromatic number," which estimates the least number of colors necessary to achieve a legal coloring of the map. This heuristic can be informed by analyzing the graph's structure derived from the map, focusing on properties like the degree of nodes (regions) and the density of edges (boundaries between regions). For example, a heuristic might approximate the cost to the goal state by considering the maximum degree of any vertex in the graph plus one, based on the observation that in a planar graph, four colors are sufficient and sometimes necessary. While this does not directly calculate the exact number of colors needed, it provides a useful estimate for guiding search algorithms by prioritizing states that are closer to achieving a valid coloring with fewer colors. This heuristic approach facilitates a more efficient exploration of possible colorings, especially in complex maps where brute-force methods would be computationally infeasible.

Used data structure for variables and states presentation

1. Map Representation

The map is represented using an adjacency list. Each region on the map is considered a node in a graph, and edges between nodes represent adjacency (i.e., two regions sharing a border). **Adjacency List** is a list where each element corresponds to a region and contains a list of its adjacent regions. It's efficient in terms of space, especially for sparse maps where most regions don't border each other.

2. State Representation

The state of the map colouring process is represented as a **List** where each element corresponds to a region, and the value of each element represents the colour assigned to that region. integers are used to represent different colours. For example, 0 for red, 1 for blue, and so on. An uncoloured region is represented by “-1”.

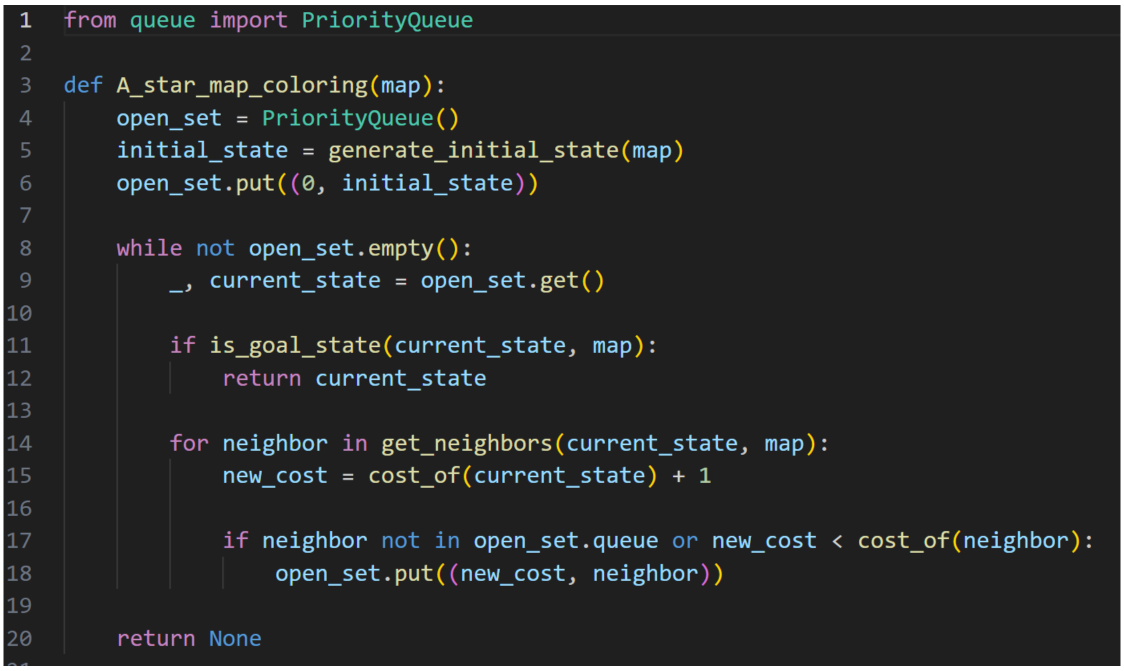
pseudocode of the algorithm

```
function A_star_map_coloring(map):
    open_set = PriorityQueue()  # Priority queue of states, prioritized by heuristic + cost
    open_set.add((initial_state, 0))  # Add initial state with cost 0

    while not open_set.isEmpty():
        current_state, cost = open_set.pop()

        if is_goal_state(current_state):
            return current_state  # Goal state reached

        for neighbor in get_neighbors(current_state, map):
            new_cost = cost + 1  # Increment cost for coloring a region

            if neighbor not in open_set or new_cost < cost_of(neighbor):
                open_set.add((neighbor, new_cost))  # Add neighbor to open set with new cost

    return None  # No solution found
```

Github repository link

https://github.com/Saadiya222/SWE485--project--Group2

Screenshots of relevant parts of the code

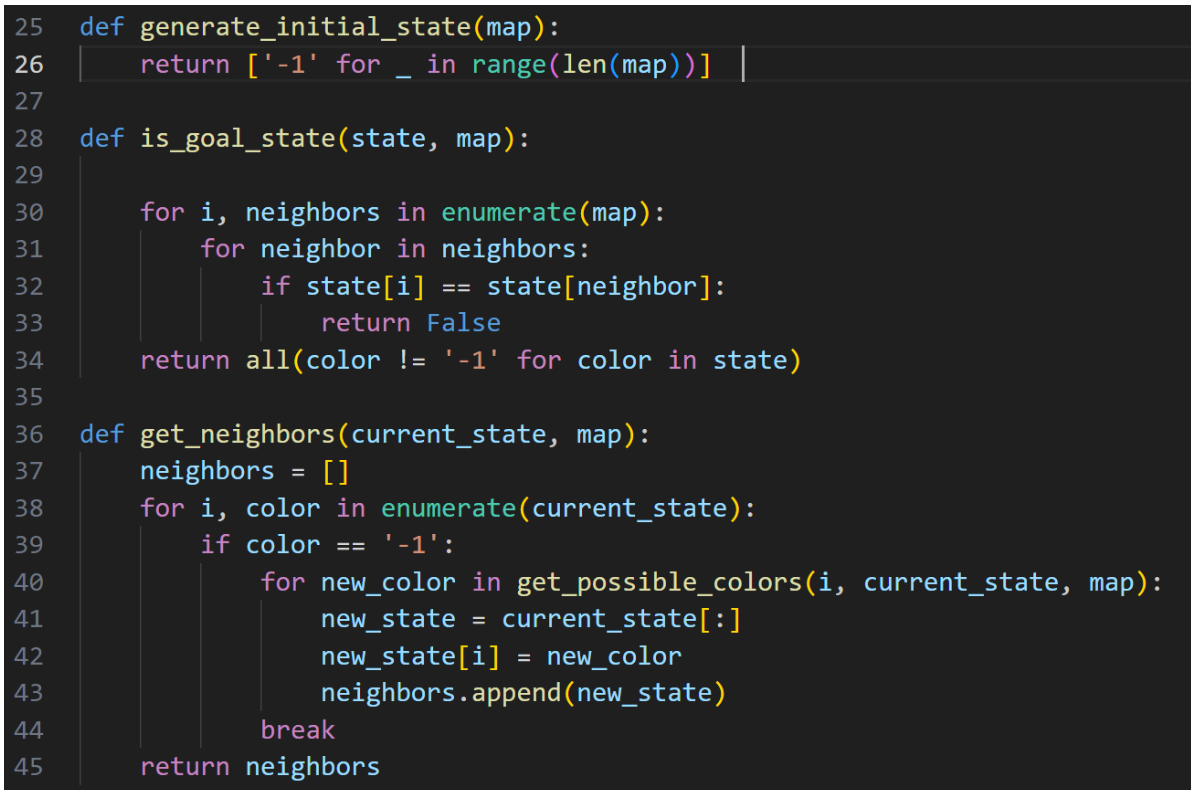


```python
from queue import PriorityQueue

def A_star_map_coloring(map):
    open_set = PriorityQueue()
    initial_state = generate_initial_state(map)
    open_set.put((0, initial_state))

    while not open_set.empty():
        _, current_state = open_set.get()

        if is_goal_state(current_state, map):
            return current_state

        for neighbor in get_neighbors(current_state, map):
            new_cost = cost_of(current_state) + 1

            if neighbor not in open_set.queue or new_cost < cost_of(neighbor):
                open_set.put((new_cost, neighbor))

    return None
```

Figure 1: A_star_map_coloring function (the main function)

The “*A_star_map_coloring*” function:

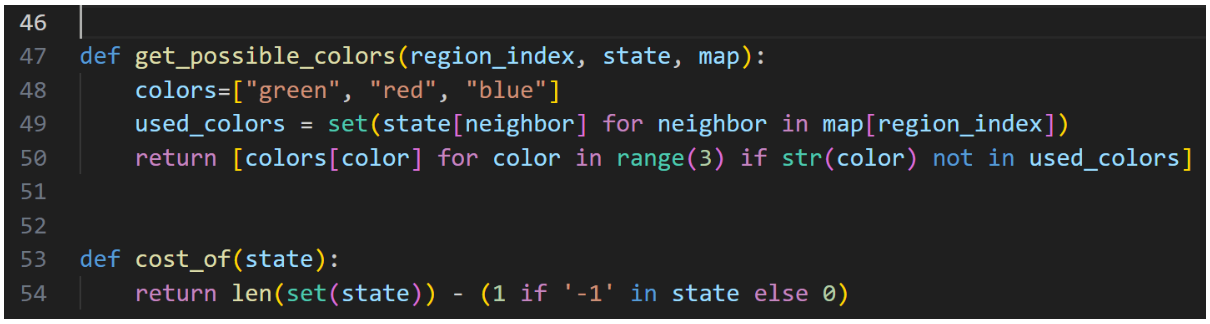
- It takes a map as input and uses the A* algorithm to solve the map coloring problem
- It initializes a priority queue *open_set* to store states prioritized by their heuristic + cost.
- It generates the initial state of the map using the *generate_initial_state* function.
- It puts the initial state with a cost of 0 into the priority queue.
- The loop continues until the priority queue is empty. It gets the state with the lowest priority (based on the cost + heuristic) from the priority queue.
- The (if *is_goal_state*) statement checks if the current state is a goal state (all regions colored without adjacent regions having the same color). If it is, it returns the current state.
- The for loop expands the search space for each neighbour of the current state.
- It calculates the new cost for coloring a region, and if the neighbour is not in the priority queue or has a lower cost, it updates the neighbour’s cost and adds it to the priority queue.
- If no solution is found (priority queue becomes empty), it returns None.



```python
def generate_initial_state(map):
    return ['-1' for _ in range(len(map))]

def is_goal_state(state, map):

    for i, neighbors in enumerate(map):
        for neighbor in neighbors:
            if state[i] == state[neighbor]:
                return False
    return all(color != '-1' for color in state)

def get_neighbors(current_state, map):
    neighbors = []
    for i, color in enumerate(current_state):
        if color == '-1':
            for new_color in get_possible_colors(i, current_state, map):
                new_state = current_state[:]
                new_state[i] = new_color
                neighbors.append(new_state)
            break
    return neighbors
```

Figure 2: Helper functions implementation

- *generate_initial_state(map)*: Generates the initial state of the map with no regions colored (or all regions marked with an initial color that represents uncolored).
- *is_goal_state(state, map)*: Checks if the current state is a goal state, meaning all regions are colored following the constraints.
- *get_neighbors(current_state, map)*: Finds all valid neighbouring states by trying to color a single uncolored region in all possible ways that don't violate the constraints.

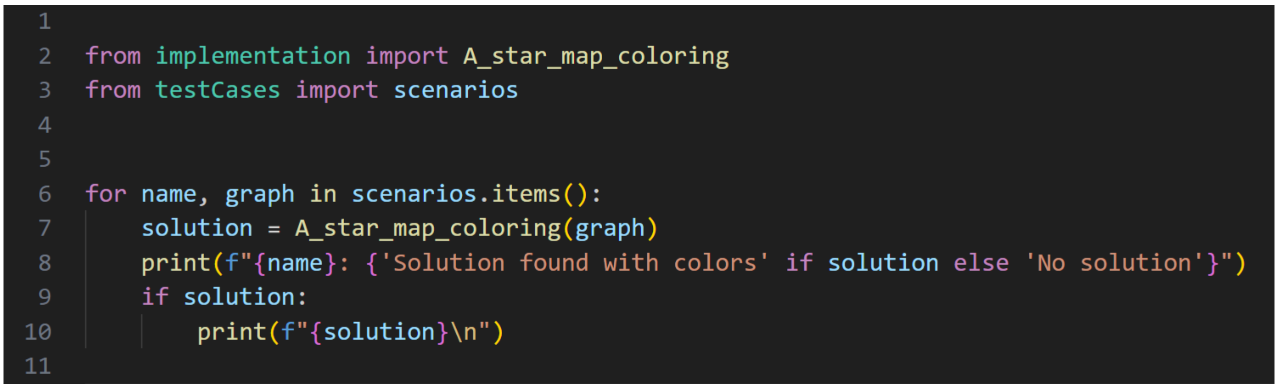


```python

def get_possible_colors(region_index, state, map):
    colors=["green", "red", "blue"]
    used_colors = set(state[neighbor] for neighbor in map[region_index])
    return [colors[color] for color in range(3) if str(color) not in used_colors]


def cost_of(state):
    return len(set(state)) - (1 if '-1' in state else 0)
```

Figure 3: Helper functions implementation

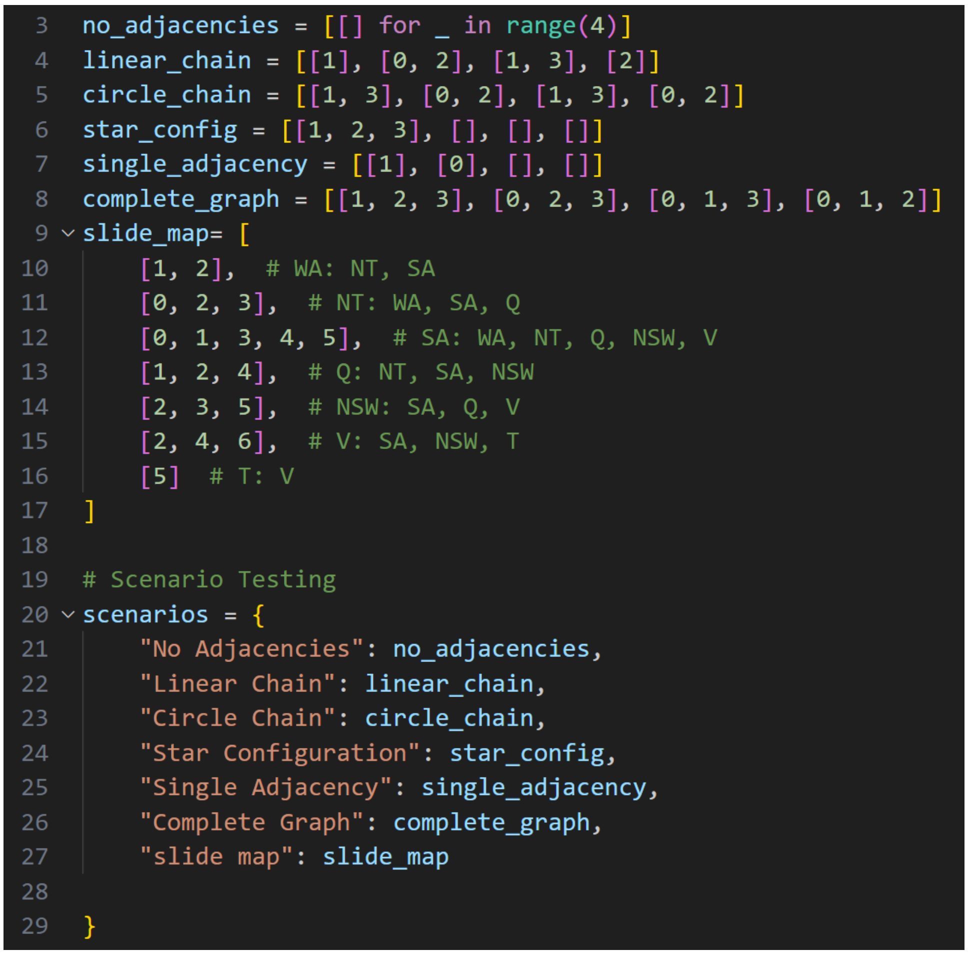
- *get_possible_colors*: Determines possible colors for a region without violating constraints.
- *cost_of(state)*: Returns the cost associated with a state. In the context of map coloring, this might simply be the number of colors used so far, if minimizing the number of colors is a goal

Testing



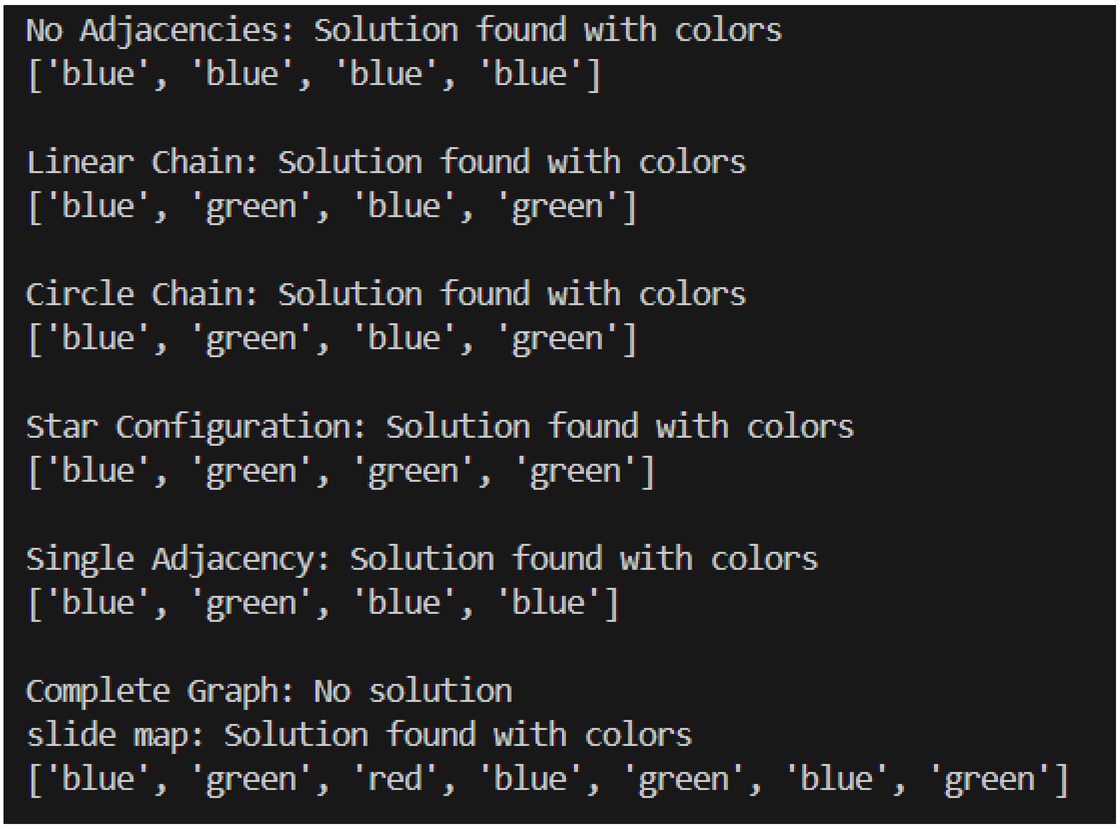
```python

from implementation import A_star_map_coloring
from testCases import scenarios


for name, graph in scenarios.items():
    solution = A_star_map_coloring(graph)
    print(f"{name}: {'Solution found with colors' if solution else 'No solution'}")
    if solution:
        print(f"{solution}\n")

```

Figure 4: map coloring problem testing code



```python
no_adjacencies = [[] for _ in range(4)]
linear_chain = [[1], [0, 2], [1, 3], [2]]
circle_chain = [[1, 3], [0, 2], [1, 3], [0, 2]]
star_config = [[1, 2, 3], [], [], []]
single_adjacency = [[1], [0], [], []]
complete_graph = [[1, 2, 3], [0, 2, 3], [0, 1, 3], [0, 1, 2]]
slide_map= [
    [1, 2],  # WA: NT, SA
    [0, 2, 3],  # NT: WA, SA, Q
    [0, 1, 3, 4, 5],  # SA: WA, NT, Q, NSW, V
    [1, 2, 4],  # Q: NT, SA, NSW
    [2, 3, 5],  # NSW: SA, Q, V
    [2, 4, 6],  # V: SA, NSW, T
    [5]  # T: V
]

# Scenario Testing
scenarios = {
    "No Adjacencies": no_adjacencies,
    "Linear Chain": linear_chain,
    "Circle Chain": circle_chain,
    "Star Configuration": star_config,
    "Single Adjacency": single_adjacency,
    "Complete Graph": complete_graph,
    "slide map": slide_map

}
```

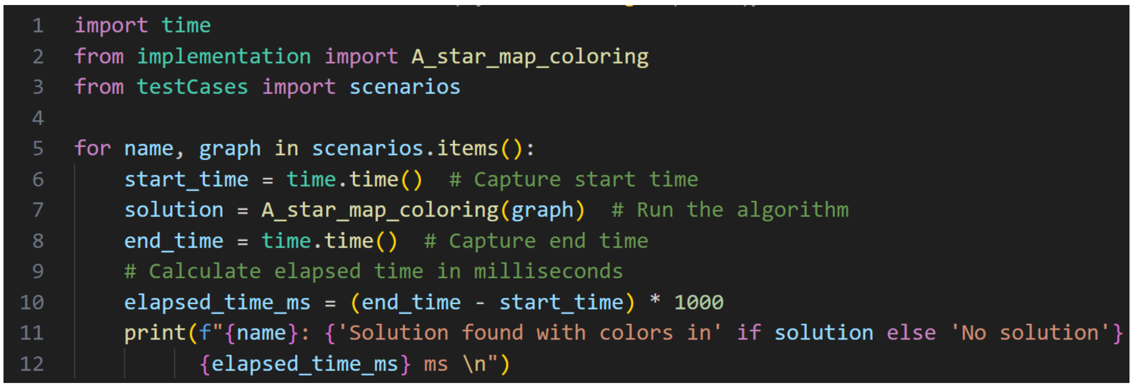
Figure 5: map coloring problem test cases



```
No Adjacencies: Solution found with colors
['blue', 'blue', 'blue', 'blue']

Linear Chain: Solution found with colors
['blue', 'green', 'blue', 'green']

Circle Chain: Solution found with colors
['blue', 'green', 'blue', 'green']

Star Configuration: Solution found with colors
['blue', 'green', 'green', 'green']

Single Adjacency: Solution found with colors
['blue', 'green', 'blue', 'blue']

Complete Graph: No solution
slide map: Solution found with colors
['blue', 'green', 'red', 'blue', 'green', 'blue', 'green']
```

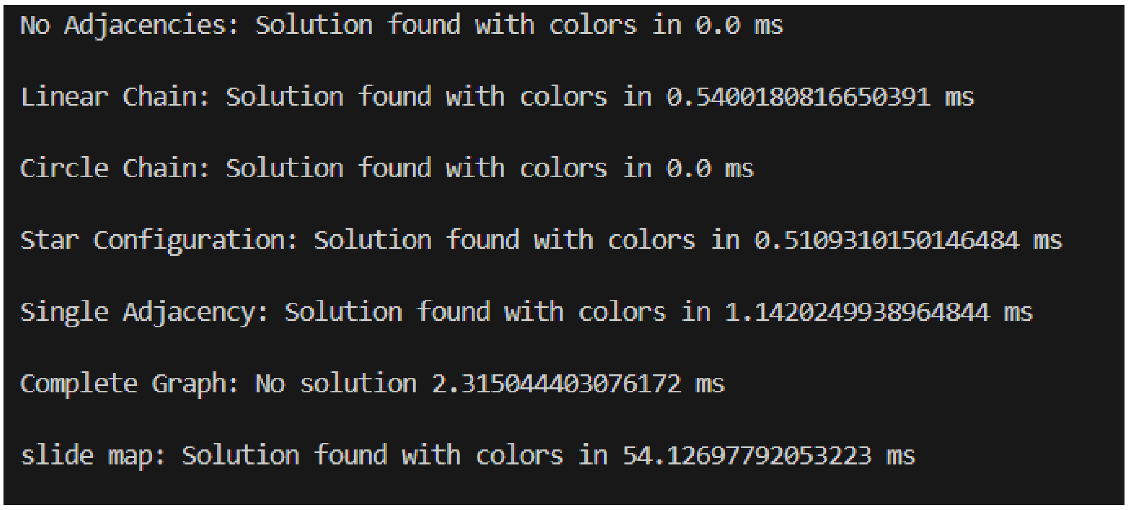
Figure 6: the result from running the test code

Computational time testing results



```python
import time
from implementation import A_star_map_coloring
from testCases import scenarios

for name, graph in scenarios.items():
    start_time = time.time()  # Capture start time
    solution = A_star_map_coloring(graph)  # Run the algorithm
    end_time = time.time()  # Capture end time
    # Calculate elapsed time in milliseconds
    elapsed_time_ms = (end_time - start_time) * 1000
    print(f"{name}: {'Solution found with colors in' if solution else 'No solution'}
          {elapsed_time_ms} ms \n")
```

Figure 7: computational time estimation code



```
No Adjacencies: Solution found with colors in 0.0 ms

Linear Chain: Solution found with colors in 0.5400180816650391 ms

Circle Chain: Solution found with colors in 0.0 ms

Star Configuration: Solution found with colors in 0.5109310150146484 ms

Single Adjacency: Solution found with colors in 1.1420249938964844 ms

Complete Graph: No solution 2.315044403076172 ms

slide map: Solution found with colors in 54.12697792053223 ms
```

Figure 8: the result from running computational time estimation code

References

[1] Map colouring algorithm (no date) Tutorialspoint. Available at: https://www.tutorialspoint.com/data_structures_algorithms/map_colouring_algorithm.htm (Accessed: 03 March 2024).

[2] Abbas AliAbbas Ali 56633 gold badges1010 silver badges1717 bronze badges and nbronbro 40.2k1212 gold badges103103 silver badges185185 bronze badges (1964) What is an objective function?, Artificial Intelligence Stack Exchange. Available at: https://ai.stackexchange.com/questions/9005/what-is-an-objective-function (Accessed: 03 March 2024).