جامعة الملك سعود
King Saud University
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: 4 Mar 2024

Table of Contents

جامعة الملك سعود
King Saud University
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):

- **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 $R1, R2, R3, \ldots, Rn$

- **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 $C = \{c1, c2, c3, \ldots, cm\}$.

- **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 Rj share a common border then the constraints would be $C_i \neq Cj$
- **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
```

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
```

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
```

- *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)
```

- *get_possible_colors:* Determines possible colors for a region without violating constraints.
- *cost_of(state): R*eturns 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")

```

```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

}
```

```
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']
```

Computational time testing results

```
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
```

جامعة الملك سعود
King Saud University
كلية علوم الحاسب والمعلومات
قسم هندسة البرمجيات

Phase 3

Complete formulation

A description of the move/operator to transit from one state to another.

Move/Operator Description :

1. **Select a Region**: Randomly choose a region from the map.
2. **Select a New Color**: Randomly select a new color that is different from the current color of the chosen region.
3. **Apply the Change**: Change the color of the selected region to the new color.
4. **Check Validity**: Ensure that the new coloring is valid, meaning no two adjacent regions share the same color.
    - If the new coloring violates the adjacency constraint, revert the change and select a different region or color combination.
    - If the new coloring is valid, proceed with the updated state.

This move/operator allows the algorithm to explore neighboring states by locally modifying the current coloring of the map while aiming to improve the quality of the solution. By iteratively applying such moves and evaluating the resulting states, the algorithm attempts to converge towards a valid and optimal solution to the map coloring problem.

The pseudocode of the algorithm.

This pseudocode outlines the basic structure of a local search algorithm for the problem, as it iteratively explores the solution space by generating neighbouring solutions and selecting the best one that improves the objective function value then terminates when no better neighbour can be found.

```
function LocalSearch(Map, Colours):
    current_solution = RandomInitialization(Map, Colours)  # Initialize a random valid solution
    while True:
        neighbour = GenerateNeighbor(current_solution) # Generate a neighbouring solution using the swap operator
        if neighbor is not None:
            current_solution = neighbor  # Update the current solution with the neighbour if valid
        else:
            return current_solution  # If no valid neighbour is found, return the current solution.
```

Screenshots of relevant parts of the code with comments

```python
def local_search(map, max_iterations=1000):
    current_state = generate_initial_state(map)
    best_state = current_state[:]
    best_cost = cost_of(current_state)

    for _ in range(max_iterations):
        neighbor = get_random_neighbor(current_state, map)
        if neighbor is None:
            continue

        neighbor_cost = cost_of(neighbor)

        if neighbor_cost < best_cost:
            best_state = neighbor[:]
            best_cost = neighbor_cost

        current_state = neighbor[:]  # Move to the nei#ghbor regardless of whether it's better

    return best_state
```

The "*local_search*" function:

- - The "*local_search*" function takes a map and an optional maximum number of iterations as input. It performs a local search algorithm to find a solution to the map coloring problem.
- - It initializes the "*current_state*" to a randomly generated initial state and keeps track of the "*best_state*" and its cost encountered during the search.
- - The function iterates for a specified number of "*max_iterations*", generating a random neighbouring state at each iteration.
- - If the neighbour is a valid state (not None), it calculates the cost of the neighbour and updates the "*best_state*" and "*best_cost*" if the neighbor has a lower cost.
- - The function returns the "*best_state*" found during the search.

```python
def get_random_neighbor(current_state, map):
    neighbors = []
    for i, color in enumerate(current_state):
        if color == '-1':
            possible_colors = get_possible_colors(i, current_state, map)
            if possible_colors:
                new_color = random.choice(possible_colors)
                neighbor = current_state[:]
                neighbor[i] = new_color
                neighbors.append(neighbor)
    if not neighbors:
        return None
    return random.choice(neighbors)
```

The "*get_random_neighbor*" function:

- The "*get_random_neighbor*" function takes the "*current_state*" and the map as input. It generates a random neighboring state by selecting a random uncolored region and assigning it a random color from the available colors that do not violate the constraints.
- It returns the generated neighbor if a valid neighbor is found, otherwise, it returns None.

```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)
```

The "*get_possible_colors*" function:

- The "*get_possible_colors*" function takes the "*region_index*", the current state, and the map as input. It determines the possible colors for a region without violating the constraints.
- It first creates a list of all available colors. Then, it removes any colors that are already used by neighboring regions from the list of available colors.
- It returns a list of the remaining available colors for the region.

The "*cost_of*" function:

- The "*cost_of*" function takes a state as input and calculates the cost associated with the state. In the context of map coloring, the cost represents the number of colors used so far, excluding the initial uncolored state represented by '-1'.
- It returns the calculated cost.

```
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)
```

The "*generate_initial_state*" function:

- The "*generate_initial_state*" function takes the map as input and generates the initial state of the map with no regions colored (or all regions marked with an initial color that represents uncolored).
- It returns a list of '-1' values representing uncolored regions.

The "*is_goal_state*" function:

- - The is_goal_state function takes a state and the map as input. It checks if the current state is a goal state, meaning all regions are colored following the constraints.
- - It iterates through each region and its neighbors in the map, checking if any neighboring regions have the same color. If it finds any, it returns False.
- - If no neighboring regions have the same color, it returns True, indicating that the current state is a goal state.

```python
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
```

The "*get_neighbors*" function:

- The "*get_neighbors*" function takes the "*current_state*" and the map as input. It finds all valid neighboring states by trying to color a single uncolored region in all possible ways that don't violate the constraints.
- It iterates through each uncolored region in the "*current_state*" and generates a new state by assigning it a color from the available colors that do not violate the constraints.
- It returns a list of all valid neighboring states.

Results of comparison

This section illustrates the difference between incremental formulation and the complete formulation implemented in phase 3, the main comparison sides are the computational time and the objective function.

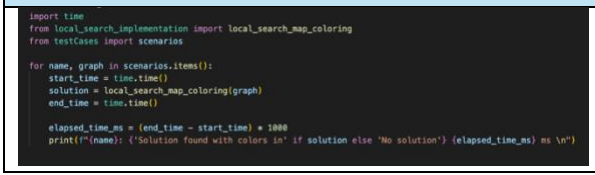
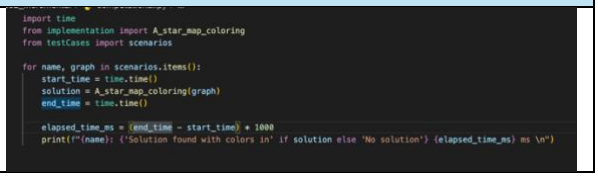
We considered various ways to compare the computational time of the algorithms we conducted brief comparison as shown in *table 1* to help us make more reasonable decision.

approach	pros	cons	Evaluation
Test scenarios	Allows for empirical comparison of algorithms on specific problem instances.	Results can be influenced by the choice of test cases and may not generalize to all scenarios.	Useful for understanding how algorithms perform on specific instances but may not provide a comprehensive view of their overall efficiency.
Big O Notation	Provides a theoretical upper bound on the growth rate of an algorithm's time complexity.	Does not account for constants, lower-order terms, or specific details of the problem instance.	Useful for comparing algorithms in terms of their scalability and efficiency in handling large input sizes but may not reflect real-world performance accurately.
Statistical Analysis	Provides a robust way to compare algorithms by analyzing their performance across multiple runs or datasets.	Requires careful design of experiments and may be computationally intensive.	Useful for gaining insights into the variability of algorithm performance and identifying trends that may not be apparent from individual measurements.

Table 1

after a careful evaluation of the approaches we narrowed our decision between ‘test scenarios’ and ‘Big O Notation’ and we decided to choose test scenarios used in phase 2, this is because using Big O notation to compare the performance of two algorithms can be challenging for certain problems, especially when the algorithms have different characteristics or complexities. In the case of the Map Coloring Problem, both the incremental formulation and the local search algorithm are heuristic search algorithms, making it difficult to express their worst-case time complexities in terms of Big O notation. So we decided to conduct the comparison using the same test cases and scenario used in the previous phase.

Table 2 below show the test used for both incremental formulation and the local search algorithm.

local search algorithm	incremental formulation
	

```
import time
from local_search_implementation import local_search_map_coloring
from testCases import scenarios

for name, graph in scenarios.items():
    start_time = time.time()
    solution = local_search_map_coloring(graph)
    end_time = time.time()

    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")
```

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

for name, graph in scenarios.items():
    start_time = time.time()
    solution = A_star_map_coloring(graph)
    end_time = time.time()

    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")
```

Table 2

Results of comparison:

incremental formulation results

```
(eval):1: unmatched "
FAY@MacBook-Pro-3 swe485 % /opt/homebrew/bin/python3 /Users/FAY/Documents/
No Adjacencies: Solution found with colors in 0.05698204040527344 ms

Linear Chain: Solution found with colors in 0.1347064971923828 ms

Circle Chain: Solution found with colors in 0.12993812561035156 ms

Star Configuration: Solution found with colors in 0.1380443572998047 ms

Single Adjacency: Solution found with colors in 0.12803077697753906 ms

Complete Graph: No solution 0.29206275939941406 ms

slide map: Solution found with colors in 8.210182189941406 ms

FAY@MacBook-Pro-3 swe485 %
```

Complete formulation results

```
No Adjacencies: Solution found with colors in 0.1971721649169922 ms

Linear Chain: Solution found with colors in 0.19097328186035156 ms

Circle Chain: Solution found with colors in 0.18286705017089844 ms

Star Configuration: Solution found with colors in 0.1809597015380859

Single Adjacency: Solution found with colors in 0.18215179443359375

Complete Graph: Solution found with colors in 0.18191337585449222 ms

slide map: Solution found with colors in 0.2472400665283203 ms
```

- **Objective Function:**
  - Both the Incremental Formulation and the Complete Formulation aim to minimize the number of colors used to color the map. This ensures efficient coloring of the map while adhering to the adjacency constraints.

- **Computational time:**
  - The complete formulationgenerally shows faster computational times compared to the incremental formulation for most scenarios. This improvement in computational time can be attributed to the local search algorithm's ability to efficiently explore neighbouring solutions and avoid exhaustive search, especially for larger and more complex maps.

As a conclusion, the complete formulation demonstrates improved efficiency in terms of computational time compared to the incremental formulation, while maintaining the same objective of minimizing the number of colors used to color the map.

Github repository link

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

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).