Artificial Intelligence (CS2012)

Logical Map Coloring

Report 2

submitted by

Rajvardhan G	**CS22B2013**
Dhivya Dharshan V	**CS22B2053**

B. Tech. Computer Science and Engineering with major in AI

Department of Computer Science

INDIAN INSTITUTE OF INFORMATION TECHNOLOGY, DESIGN AND MANUFACTURING, KANCHEEPURAM

CHENNAI-600127

25 April 2024

Contents

1 Introduction

The **Map Coloring Problem** is a well-known challenge in computer science, involving the assignment of colors to map regions within certain constraints. In this report, we explore our AI-based solution to this problem. Through *logical reasoning* and *constraint satisfaction techniques*, we navigate the complexities of map coloring, showcasing the effectiveness of AI in computational problem-solving.

2 Constraints

To achieve a valid solution in the map coloring problem, three primary constraints must be satisfied:

i. **Each region must be assigned a color:** For every state x, there exists a color c such that x has that color.

   Symbolically, $\forall x(State(x)) \Rightarrow \exists c(Colored(x, c))$

ii. **Each region can have at most one color:** For every state x and every pair of colors $c1$ and $c2$, if x is assigned color $c1$ and color $c2$, then $c1$ must be equal to $c2$.

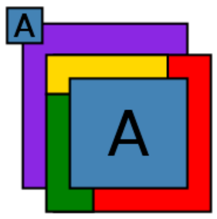
   Symbolically, $\forall x \forall c1 \forall c2(Colored(x, c1) \wedge Colored(x, c2) \Rightarrow ClrEq(c1, c2))$

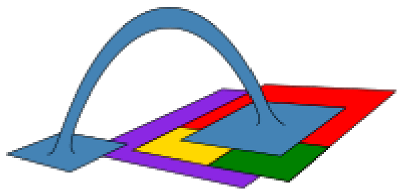
iii. **Adjacent regions must have different colors:** For every pair of adjacent states $x1$ and $x2$, there cannot exist a color c such that $x1$ and $x2$ are both assigned color c.

   Symbolically, $\forall x1 \forall x2(Adjacent(x1, x2) \Rightarrow \neg\exists c(Colored(x1, c) \wedge Colored(x2, c)))$

3 Four Color Theorem

The Four Color theorem states, “given any planar map, it is always possible to color the regions with *at most four colors* in such a way that no two adjacent regions have the same color.”





4 Algorithm

I. **Initialization:**

The problem (map) is represented using a dictionary named ‘adjSts’ (*adjacent states*), where each **key** represents a **state**, and the corresponding **value** is a list of **adjacent states.**

Initializations are done for assigning numerical values to states and colors using the dictionary ‘assign’.

II. **Knowledge Base setup:**

A **Pytholog** knowledge base ‘kb’ is initialized to store *facts* and *rules* using predicate logic.

III. **Representing States and Adjacency relations:**

Each state is represented as a predicate ‘State’ (**State(X)**) in the knowledge base, assigned a numerical value.

Adjacent states are represented using the ‘Adjacent’ predicate (**Adjacent(X, Y)**), asserting the adjacency relation between states.

IV. **Defining Colors:**

A set of colors {*red, green, blue, yellow*} is defined, and each color is assigned a numerical value.

Colors are represented using the ‘Color’ predicate (**Color(C)**) in the knowledge base.

V. **Handling Color Equality and Inequality:**

Predicates ‘ClrEq’ (**ClrEq(C1, C2)**) and ‘ClrNotEq’ (**ClrNotEq(C1, C2)**) are used to define *equality* and *inequality* constraints between colors.

These constraints help in ensuring the constraint: *no two adjacent states can have the same color.*

VI. **Formulating the Coloring Rule:**

A rule is formulated to represent the coloring constraint using the ‘Coloring’ (**Coloring(A, B, C, ...) :- ClrNotEq(A, B), ClrNotEq(A, C), ...**) predicate.

The rule ensures that no adjacent states have the same color.

VII. **Querying:**

A query ‘q’ is constructed to find valid colorings using the ‘Coloring’ predicate.

The query returns solutions that satisfy the coloring constraints.

VIII. **Checking for Minimal Colors:**

Solutions are evaluated to find the *minimum number of colors* required to color the map.

IX. **Output:**

The output displays the valid colorings for the regions/states along with the minimum number of colors used, while satisfying the constraints.

X. **Validity:**

[*Please refer Glossary for function abbreviations*]

i. **checkOnlyOneClr:**
This function ensures that each region/state is precisely assigned one color. It relies on the predicate ‘StClrd’ (*State is Colored*), defined as “**StClrd(X) :- State(X), Color(C), Colored(X, C)**”, asserting that every state ‘X’ is indeed colored with a single color.

ii. **checkAdjSameClr:**

This function ensures that adjacent regions/states do not share the same color. It relies on the predicate 'AdjSameClr' (*Adjacent states have Same Color*), defined as "**AdjSameClr(X, Y) :- State(X), State(Y), Adjacent(X, Y), Color(C1), Color(C2), Colored(X, C1), Colored(Y, C2), ClrEq(C1, C2)**", which asserts that adjacent states 'X' and 'Y' have different colors.

Thus, by the above two functions the validity of the solution is verified.

XI. **Application of Constraints and Logic:**

The algorithm effectively applies constraints using predicate logic and ensures *logical consistency* throughout the solution process.

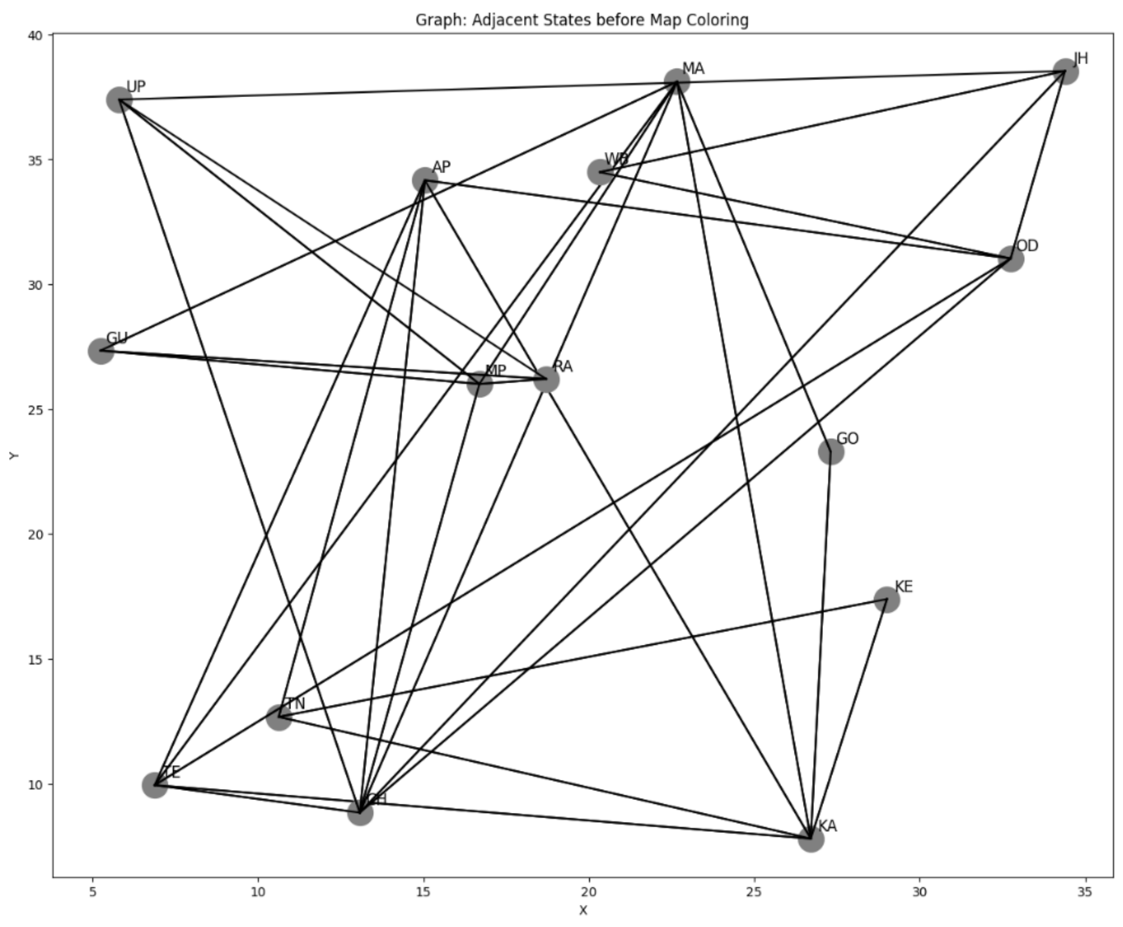
In summary, the algorithm employs a combination of *constraint satisfaction*, *predicate logic* and *query-based reasoning* to efficiently solve the Map Coloring Problem. It formulates the problem as a CSP, applies logical constraints and systematically searches for valid colorings, ultimately providing solutions that adhere to the problem requirements.

5 Outputs

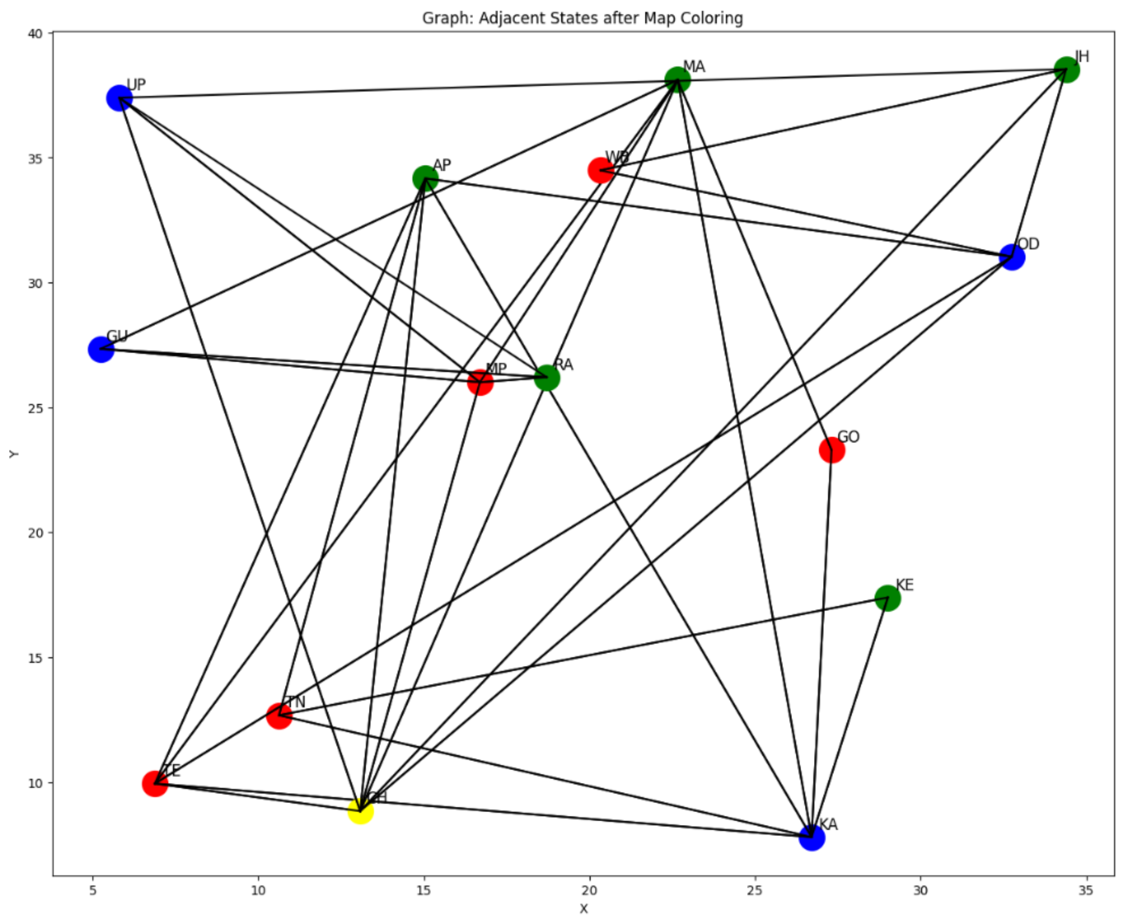
Our success with **predicate logic** is evident in our achievement of coloring the map *without any adjacent states sharing the same color.* This is demonstrated through two visual representations:

The initial state of the map, and the map after applying our coloring solution. In the "*before*" graph, state adjacency is depicted with gray circles and black lines, while the "*after*" graph shows each state appropriately colored, ensuring no neighboring states share the same color. This graphical evidence highlights the effectiveness of our approach in meeting the requirement of distinct colors for adjacent states.

before map coloring:-



after map coloring:-

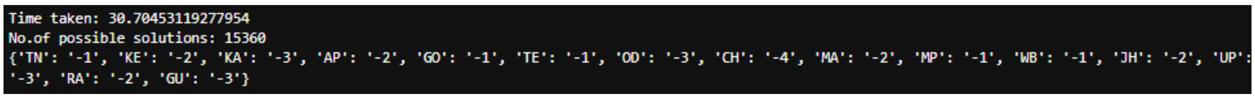


6 Observations

Predicate logic offers a high level of *expressiveness*, allowing for the representation of complex constraints in a natural and intuitive manner.

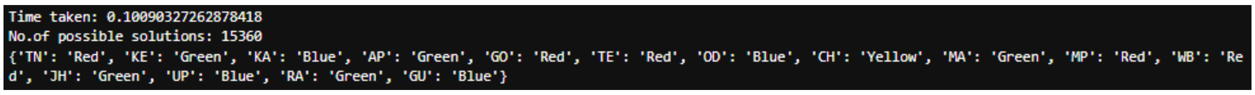
With the completion of our code, we aimed to benchmark it against established solutions for map coloring, such as **backtracking**. Upon comparison, it became apparent that backtracking offered a more efficient solution in terms of *runtime*.

Predicate Logic:-



```
Time taken: 30.70453119277954
No.of possible solutions: 15360
{'TN': '-1', 'KE': '-2', 'KA': '-3', 'AP': '-2', 'GO': '-1', 'TE': '-1', 'OD': '-3', 'CH': '-4', 'MA': '-2', 'MP': '-1', 'WB': '-1', 'JH': '-2', 'UP':
'-3', 'RA': '-2', 'GU': '-3'}
```

Backtracking:-



```
Time taken: 0.10090327262878418
No.of possible solutions: 15360
{'TN': 'Red', 'KE': 'Green', 'KA': 'Blue', 'AP': 'Green', 'GO': 'Red', 'TE': 'Red', 'OD': 'Blue', 'CH': 'Yellow', 'MA': 'Green', 'MP': 'Red', 'WB': 'Re
d', 'JH': 'Green', 'UP': 'Blue', 'RA': 'Green', 'GU': 'Blue'}
```

Our approach incurred slower execution time due to the *inherent computational overhead* associated with manipulating *logical expressions* and conducting *query-based reasoning*. This resulted in comparatively slower performance when benchmarked against specialized algorithms such as **backtracking**.

7 Conclusion

Despite the observed inefficiencies, there is potential for optimization in our predicate logic-based approach. Techniques such as **constraint propagation** and **query optimization** could improve the *efficiency* and *scalability* of the algorithm.

Moving forward, exploring *hybrid approaches that combine the strengths of predicate logic with more efficient search and optimization techniques* could lead to more effective solutions for challenging problem domains like map coloring.

8 Contributions

CS22B2013 [Rajvardhan G]:

- Major contributions in *report 1.*
- Major contributions in *simulation and visualization of results.*

CS22B2053 [Dhivya Dharshan V]:

- Major contributions in *report 2.*
- Major contributions in *constructing the main working code.*

Overall:

- Both developed algorithm for the code.
- Both contributed to all aspects of the project.

9 Glossary

9.1 Predicates

• **State(X):**	Represents a state on the map.
• **Adjacent(X, Y):**	Indicates that state 'X' is adjacent to 'Y'.
• **Color(C):**	Represents a color.
• **ClrEq(C1, C2):**	Indicates that color 'C1' is equal to color 'C2'.
• **ClrNotEq(C1, C2):**	Indicates that color 'C1' is not equal to color 'C2'.
• **Coloring(...):**	Predicate used in the rule to represent a *valid coloring* of the map.
• **Colored(X, C):**	Indicates that state 'X' is colored with color 'C'.
• **StClrd(X):**	Predicate used to represent a state being colored.
• **AdjSameClr(X, Y):**	Indicates that adjacent states 'X' and 'Y' have the same color.

9.2 Rules

- **Coloring(A, B, C, ...) :- ClrNotEq(A, B), ClrNotEq(A, C), ...:** The main rule used to define a valid coloring of the map. It ensures that adjacent states have different colors.
- **StClrd(X) :- State(X), Color(C), Colored(X, C):** Rule used to check if each state is colored with exactly one color.
- **AdjSameClr(X, Y) :- State(X), State(Y), Adjacent(X, Y), Color(C1), Color(C2), Colored(X, C1), Colored(Y, C2), ClrEq(C1, C2):** Rule used to check if adjacent states have the same color.

9.3 Function Abbreviations

- **checkOnlyOneClr**: Checks if each region has only one color assigned to it.
- **checkAdjSameClr**: Checks if adjacent regions are assigned with the same color.

9.4 Abbreviations

- **TN**: Tamil Nadu
- **KE**: Kerala
- **KA**: Karnataka
- **AP**: Andhra Pradesh
- **GO**: Goa
- **TE**: Telangana
- **OD**: Odisha
- **CH**: Chhattisgarh
- **MA**: Maharashtra
- **MP**: Madhya Pradesh
- **WB**: West Bengal
- **JH**: Jharkhand
- **UP**: Uttar Pradesh
- **RA**: Rajasthan
- **GU**: Gujarat

10 References

https://en.wikipedia.org/wiki/Four_color_theorem

https://pypi.org/project/pytholog/