Laboratory 2
Variant 5
Group 5

Paweł Uhma Maurizio Carrasquero

March 31, 2025

1 Introduction

Our task was to implement and test a Constraint Satisfaction Problem (CSP) solver for a map-coloring scenario using backtracking with forward checking.

- **Task Description:**
  - Develop a CSP solver that uses backtracking combined with forward checking to efficiently assign colors to regions on a map.
  - Test the implementation with various maps
- **Algorithm Overview:**
  - *Backtracking:* This technique incrementally builds candidates to the solution and removes a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a solution. For our problem, it assigns a color to each region one at a time, and if an assignment leads to a conflict, it backtracks and tries a different color.
  - *Forward Checking:* Forward checking works in in pair with backtracking by looking ahead. When a region is assigned a color, forward checking removes that color from the domains of all its unassigned neighbors.

- **Advantages:**
  - **Early Conflict Detection:** Forward checking allows the algorithm to detect conflicts at an early stage, avoiding unnecessary computation.
  - **Efficiency:** The combination of backtracking with forward checking is significantly more efficient than just backtracking.
- **Disadvantages:**
  - **Exponential Complexity:** Despite the improvements, the worst-case time complexity remains exponential.

2 Implementation

2.1 Data Structures

We represent map as a dictionary, where each state or region is a key, and its value is a list of neighbors. For example:

Listing 1: US map dictionary

```
cmap = {
    "CA": ["OR", "NV", "AZ"],
    "OR": ["WA", "ID", "NV", "CA"],
    "NV": ["OR", "ID", "UT", "AZ", "CA"],
    ...
}
```

2.2 CSP Class and Functions

Here is an outline of the CSP solver class:

Listing 2: CSP class outline

```
class CSP:
    def __init__(self, variables, domains, constraints):
        self.variables = variables
        self.domains = domains
        self.constraints = constraints
        self.solution = None
```

```
    def forward_checking(self, var, value, assignment):
        # Removes 'value' from domains of neighbors, etc.

    def backtrack(self, assignment):
        # Standard backtracking approach with forward
           checking

    def solve(self):
        return self.backtrack({})
```

2.2.1 Forward Checking

Forward checking is a technique used during the search process in constraint satisfaction problems. When a region `var` is assigned a color `value`, forward checking immediately removes `value` from the domains of all its unassigned neighboring regions. If, after removal, any neighbor's domain becomes empty, it ensures that no valid color can be assigned to that neighbor.

```
def forward_checking(self, var, value, assignment):
    removed = {}
    for neighbor in list(self.constraints[var]):
        if neighbor not in assignment:
            if value in self.domains[neighbor]:
                if neighbor not in removed:
                    removed[neighbor] = []
                self.domains[neighbor].remove(value)
                removed[neighbor].append(value)
                if len(self.domains[neighbor]) == 0:
                    # Restore removed values before
                       backtracking
                    for n in removed:
                        for v in removed[n]:
                            self.domains[n].append(v)
                    return None
    return removed
```

2.2.2 Backtracking

Backtracking is a recursive method that systematically explores the possible color assignments. It operates as follows:

1. **Completion Check:** If all regions have been assigned a color, the complete assignment is returned.

2. **Variable Selection:** If there are unassigned regions, an unassigned region is chosen.

3. **Value Assignment:** For the chosen region, each color in its domain is tried one by one.

4. **Conflict Check:** For each color, the algorithm checks if any assigned neighbor already has that color. If a conflict is detected, the color is skipped.

5. **Forward Checking:** If no conflict exists, the color is tentatively assigned, and forward checking is applied to remove that color from the domains of neighboring regions.

6. **Recursive Search:** The algorithm recursively attempts to complete the assignment with the updated domains.

7. **Backtracking:** If the recursive call fails to produce a complete assignment, the tentative assignment is undone, and any changes made during forward checking are restored. The algorithm then tries the next available color.

```
def backtrack(self, assignment):
    if len(assignment) == len(self.variables):
        return assignment
    var = None
    for v in self.variables:
        if v not in assignment:
            var = v
            break
    for value in list(self.domains[var]):
        conflict = False
        for neighbor in list(self.constraints[var]):
            if neighbor in assignment and assignment[neighbor
                ] == value:
                conflict = True
                break
        if conflict:
            continue
        assignment[var] = value
        removed = self.forward_checking(var, value,
            assignment)
        if removed is not None:
```

```
            result = self.backtrack(assignment)
            if result is not None:
                return result
        # Backtracking: undo the assignment and restore the
            domains.
        del assignment[var]
        if removed is not None:
            for neighbor, vals in removed.items():
                for v in vals:
                    self.domains[neighbor].append(v)
    return None
```

2.3 Example Run and Visualizations

Applying the algorithm on the map of states of the USA.

Listing 3: Results

```
Solution:
 {'AL': 'Red', 'AK': 'Red', 'AZ': 'Red', 'AR': 'Red', 'CA': '
    Green', 'CO': 'Green', 'CT': 'Red', 'DE': 'Red', 'FL': '
    Green', 'GA': 'Blue', 'HI': 'Red', 'ID': 'Red', 'IL': '
    Red', 'IN': 'Green', 'IA': 'Green', 'KS': 'Red', 'KY': '
    Blue', 'LA': 'Blue', 'ME': 'Red', 'MD': 'Green', 'MA': '
    Green', 'MI': 'Red', 'MN': 'Blue', 'MS': 'Yellow', 'MO':
    'Yellow', 'MT': 'Green', 'NE': 'Blue', 'NV': 'Yellow', '
    NH': 'Blue', 'NJ': 'Green', 'NM': 'Yellow', 'NY': 'Yellow
    ', 'NC': 'Red', 'ND': 'Yellow', 'OH': 'Yellow', 'OK': '
    Blue', 'OR': 'Blue', 'PA': 'Blue', 'RI': 'Blue', 'SC': '
    Green', 'SD': 'Red', 'TN': 'Green', 'TX': 'Green', 'UT':
    'Blue', 'VT': 'Red', 'VA': 'Yellow', 'WA': 'Green', 'WV':
     'Red', 'WI': 'Yellow', 'WY': 'Yellow'}
}
```

Four colors is the minimal number where the algorithm was able to find the solution.

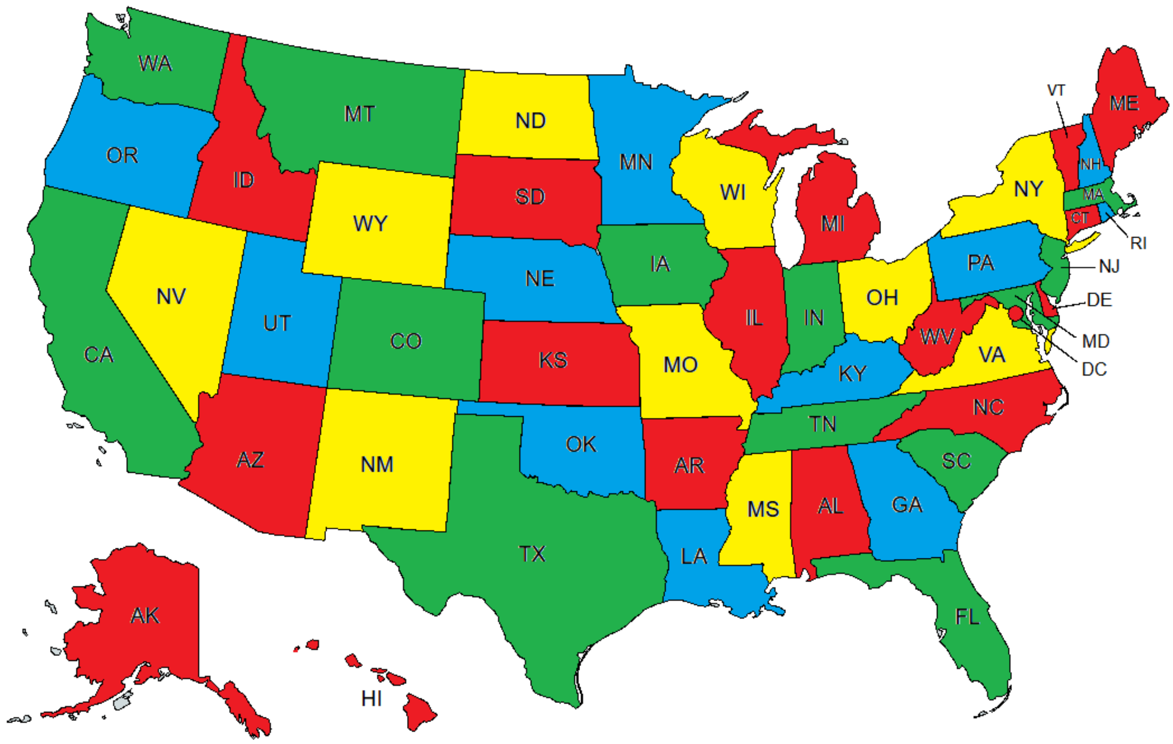

Figure 1: Colored map with a solution
We see how each region has been assigned a distinct color from its neighbors.

3 Discussion

3.1 Test Cases and Corner Cases

We validated our solution using the following four test cases:

- **Test Case 1 (Complex Map):** A 13-region map with multiple constraints, which tests the solver on a realistic scenario.
- **Test Case 2 (United States Map):** A map of U.S. states and their neighbors, representing a large and complex real-world problem.
- **Test Case 3 (Triangle):** A minimal map with three regions in a triangle configuration.

- **Test Case 4 (Linear Chain):** A simple linear chain of regions, demonstrating that alternating colors are correctly assigned in a straightforward scenario.

Each test case confirmed that our algorithm consistently produces a valid color assignment.

4 Conclusion

- **What we learned:** We deepened our understanding of Constraint Satisfaction Problems, backtracking, and the benefits of forward checking in pruning the search space.

- **Difficulties encountered:** We had some problems with understanding how back tracking and forward checking should be implemented in this example.