Laboratory 2

Variant 5

Group 10

By Bora ILCI & Kaan Emre KARA

Introduction

The task for this laboratory is to solve the map coloring problem using a constraint satisfaction approach. In this task, each region (or node) of a map must be assigned a color such that no two adjacent regions share the same color. The algorithm implemented is a backtracking search enhanced with forward checking—a method widely used in constraint satisfaction problems (CSPs).

The backtracking algorithm systematically assigns colors to variables (regions) and checks for consistency with given constraints (neighbors must have different colors). The advantage of this approach is its simplicity and flexibility for a wide range of CSPs. However, without any heuristics, backtracking can explore many unnecessary paths. This is where forward checking comes into play, reducing the search space by pruning domains of future variables. A potential disadvantage is that the extra overhead may not be as beneficial in problems with very light constraints.

Implementation

Defining Variables, Domains, and Constraints

- **Variables:**

The regions to be colored are defined as the variables of the CSP. In our implementation, they are extracted from the keys of the input constraint dictionary. For example:

```
variables = list(map_constraints.keys())
```

- **Domains:**

Each variable has a domain, which is the list of possible colors that can be assigned. The domains are defined based on a fixed list of colors, and for each region the domain is initially the same:

```
domains = {var: colors[:] for var in variables}
```

- **Constraints:**

Constraints are provided as a dictionary where each key (a region) maps to a list of neighboring regions. The algorithm ensures that no two neighbors have the same color. The constraint is checked in the is_consistent method by verifying that if a neighbor is already assigned, it does not share the same color.

Forward Checking

The role of forward checking in our CSP solver is to reduce the search space early. When a variable is assigned a color, the forward_checking function iterates over its neighbors and removes the assigned color from their domains. If this removal results in any neighbor having an empty domain, the algorithm immediately backtracks. Here is the key snippet:

```
def forward_checking(self, var, value, assignment):
    removed = {}  # neighbor -> list of removed values
    for neighbor in self.constraints.get(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 not self.domains[neighbor]:
                    for n, vals in removed.items():
                        self.domains[n].extend(vals)
                    return None
    return removed
```

Forward checking improves the algorithm by:

- **Pruning the search space:** It quickly detects when an assignment will lead to failure, thus avoiding deeper unnecessary recursion.

- **Early conflict detection:** By checking neighbors immediately after an assignment, it prevents the algorithm from exploring branches that are doomed to fail.

In summary, yes—forward checking significantly improves the performance of the backtracking algorithm for many CSP instances, especially in cases with a higher density of constraints.

Visualization and Additional Code Details

The implementation also includes visualization code that uses the networkx and matplotlib libraries to display the colored maps. The function visualize_map generates a graph layout based on the problem structure and uses a predefined set of elegant colors to reflect the solution. This interactive visualization aids in verifying the solution and understanding the effect of variable assignments and constraints.

Discussion

Explanation of CSP Components

- **Variables:**

Each region of the map is treated as a variable. Their identity is derived directly from the input map constraint dictionary keys.

- **Domains:**

The domain for each variable consists of a set of available colors. These are the candidate values that can be assigned without violating the color differentiation rule.

- **Constraints:**

The constraints ensure that any two adjacent regions (as defined by the neighbor lists in the input dictionary) do not receive the same color. This is enforced in the is_consistent method.

Role of Forward Checking

Forward checking is implemented to enhance the backtracking algorithm by eliminating candidate colors from the domains of neighboring unassigned variables immediately after a variable is assigned a value. This proactive step:

- **Reduces unnecessary search:** It stops the algorithm from going further down a path that will lead to a dead end.
- **Improves efficiency:** In many cases, it detects conflicts early, thereby reducing the number of recursive calls and overall computational time.

Thus, forward checking is an effective optimization that improves the performance of our CSP solver, particularly in problems where constraints are tightly coupled.

Test Cases and Corner Cases

1. Canada Map:

- A realistic scenario using regions (like “ab”, “bc”, “mb”, etc.) representing parts of Canada.
- **Why:** Demonstrates performance on an irregular map with varying degrees of constraints.

2. **Complex Maps:**

• These include non-trivial structures where some regions have many neighbors.

• **Why:** To test the algorithm's robustness when faced with higher constraint densities.

3. **Complete Graph K8:**

• Every region is adjacent to every other region.

• **Why:** This is a corner case that stresses the algorithm as it requires as many colors as nodes, testing the worst-case performance.

4. **Cycle Map:**

• Regions are arranged in a cycle.

• **Why:** A cycle's parity (odd vs. even number of nodes) determines the minimum number of colors needed, which is a classic CSP corner case.

5. **Cross Map:**

• A non-trivial structure with overlapping constraints.

• **Why:** To evaluate the solver's efficiency on maps with intersecting constraints.

6. **Bipartite Graph:**

• A structured map where nodes can be divided into two sets, with all edges between sets.

• **Why:** Bipartite graphs are ideal for testing, as they can be colored using only two colors, representing an ideal scenario for forward checking.

Conclusion

In this laboratory, we implemented a CSP solver for the map coloring problem using a backtracking algorithm enhanced with forward checking.

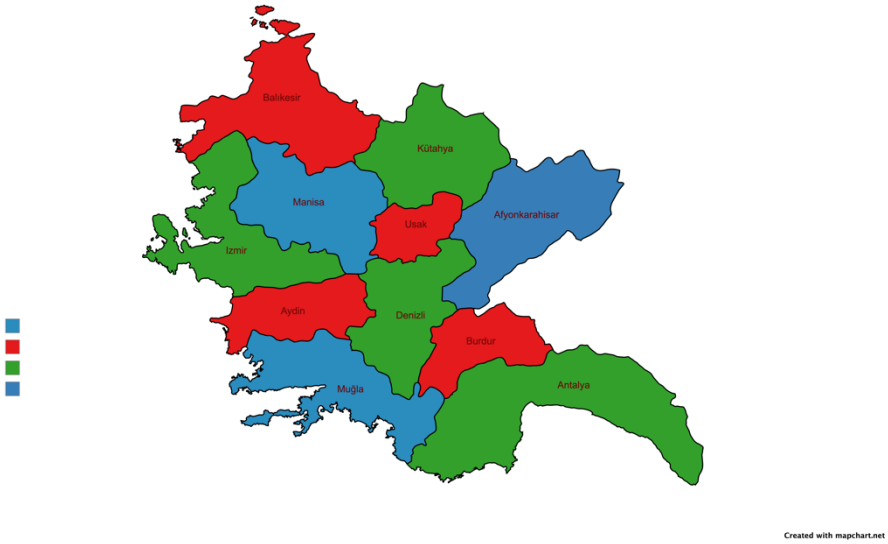
Forward checking plays a crucial role by pruning the search space and detecting dead ends early, thereby improving the overall efficiency of the algorithm. The variety of test cases—including realistic maps, complete graphs, cycles, and bipartite graphs—ensured that the solution is robust and performs well under different constraints and corner cases.

• The use of forward checking greatly reduces unnecessary computations.

• The importance of carefully selecting test cases to cover both typical and extreme scenarios.

Examples of Map Coloring

Aegean region of Turkey, contains 11 provinces



South-east of United States