AIR UNIVERSITY

Department of Electrical and Computer Engineering

Lab # 04: Constraint Satisfaction Problems

Lab Instructor: Muhammad Awais

Constraint Satisfaction Problems (CSP) in Artificial Intelligence

Finding a solution that meets a set of constraints is the goal of constraint satisfaction problems (CSPs), a type of AI issue. Finding values for a group of variables that fulfill a set of restrictions or rules is the aim of constraint satisfaction problems. For tasks including resource allocation, planning, scheduling, and decision-making, CSPs are frequently employed in AI.

There are mainly three basic components in the constraint satisfaction problem:

1. **Variables:**The things that need to be determined are variables. Variables in a CSP are the objects that must have values assigned to them in order to satisfy a particular set of constraints.
2. **Domains:** The range of potential values that a variable can have is represented by domains.
3. **Constraints:** The guidelines that control how variables relate to one another are known as constraints. Constraints in a CSP define the ranges of possible values for variables.

N Queen Problem

```
def is_safe(board, row, col, N):
    for i in range(N):
        if board[i][col] == 1:
            return False

    x = row
    y = col

    while x >= 0 and y >= 0:
        if board[x][y] == 1:
          return False
        x = x - 1
        y = y - 1

    x = row
    y = col

    while x >= 0 and y < N:
        if board[x][y] == 1:
          return False
        x = x - 1
        y = y + 1
    return True
```

```
def print_board(board, n):
  print("Solution:")

  for i in range(n):
    for j in range(n):
      if board[i][j] == 1:
        print("Q", end=" ")
      else:
        print("--", end=" ")
    print()

N = 4
board = [[0]*N for i in range(N)]
solution_count = 0


def n_Queens(board, row, N):
    global solution_count

    if row >= N:
        solution_count += 1
        print("Solution ", solution_count, ":\n")
        print_board(board, N)
        print("\n")
        return

    for col in range(N):
        if is_safe(board, row, col, N):
            board[row][col] = 1
            n_Queens(board, row + 1, N)
            board[row][col] = 0

n_Queens(board, 0, N)
print("Total number of solutions:", solution_count)
```

```
Solution  1 :

Solution:
-- Q -- --
-- -- -- Q
Q -- -- --
-- -- Q --


Solution  2 :

Solution:
-- -- Q --
Q -- -- --
-- -- -- Q
-- Q -- --


Total number of solutions: 2
```

Graph Coloring

```
#Graph Declaration
graph = {
    'Gilgit': ['AJK', 'KPK'],
    'KPK': ['AJK', 'Gilgit', 'Punjab', 'Balochistan'],
    'AJK': ['Gilgit', 'KPK', 'Punjab'],
    'Punjab': ['AJK', 'KPK', 'Balochistan', 'Sindh'],
    'Sindh': ['Punjab', 'Balochistan'],
    'Balochistan': ['KPK', 'Punjab', 'Sindh']
}


def is_valid(graph, node, color, color_map):
    for neighbor in graph[node]:
        if neighbor in color_map and color_map[neighbor] == color:
            return False
    return True
```

```
def solve_map_coloring(graph, colors):
    solutions = []
    color_map = {}
    stack = [(list(graph.keys())[0], iter(colors))]

    while stack:
        node, color_iter = stack[-1]
        try:
            color = next(color_iter)
            if is_valid(graph, node, color, color_map):
                color_map[node] = color
                if len(color_map) == len(graph):
                    solutions.append(color_map.copy())
                for neighbor in graph[node]:
                    if neighbor not in color_map:
                        stack.append((neighbor, iter(colors)))
                        break
            else:
                continue
        except StopIteration:
            color_map.pop(node, None)
            stack.pop()

    return solutions


colors = ['red', 'green', 'blue']

# Solve the map coloring problem
solutions = solve_map_coloring(graph, colors)

# Print the solutions
if solutions:
    for i, solution in enumerate(solutions, 1):
        print(f"Solution {i}:")
        for node, color in solution.items():
            print(f"{node}: {color}")
        print()
else:
    print("No solutions found.")
```

```
Solution 1:
Gilgit: red
AJK: green
KPK: blue
Punjab: red
Balochistan: green
Sindh: blue

Solution 2:
Gilgit: red
AJK: blue
KPK: green
Punjab: red
Balochistan: blue
Sindh: green

Solution 3:
Gilgit: green
AJK: red
KPK: blue
Punjab: green
Balochistan: red
Sindh: blue

Solution 4:
Gilgit: green
AJK: blue
KPK: red
Punjab: green
Balochistan: blue
Sindh: red

Solution 5:
Gilgit: blue
AJK: red
KPK: green
Punjab: blue
Balochistan: red
```

```
Sindh: green

Solution 6:
Gilgit: blue
AJK: green
KPK: red
Punjab: blue
Balochistan: green
Sindh: red
```

Conclusion

In these tasks, we delved into two classic problems in constraint satisfaction programming (CSP): the Map Coloring Problem and the N-Queens Problem. Both exemplify the application of CSP techniques in solving combinatorial optimization challenges efficiently. In addressing the Map Coloring Problem, we aimed to assign colors to different regions of a map such that no neighboring regions shared the same color. Utilizing a backtracking algorithm, we traversed the search space, ensuring each color assignment adhered to the constraint of neighboring regions having distinct colors. We extended the solution to enumerate and present all feasible colorings of the map. Similarly, when tackling the N-Queens Problem, our goal was to place N queens on an N×N chessboard without any two queens threatening each other. Employing a backtracking approach with optimizations, we explored the solution space, ensuring each queen placement met the constraint of non-attackability. By modifying the algorithm to track multiple solutions, we could enumerate all valid queen configurations on the chessboard. Through these tasks, we showcased the versatility and effectiveness of CSP techniques in solving a diverse range of constraint-based puzzles and combinatorial optimization problems.