



|  |  |
| --- | --- |
| **Student Name** | **Arun Adhikari** |
| **SRN No** | 202100406 |
| **Roll No** | 31 |
| **Program** | **Department of Computer Engineering** |
| **Year** | 2024-2025 |
| **Division** | G |
| **Subject** | Artificial Intelligence |
| **Assignment No** | 6 |



**Experiment Number - 06**

**Title /Problem Statement**: Implement Constraint Satisfaction Algorithm for the following problems: a. Cryptarithmetic b. Crossword puzzle c. Map coloring problem

**Cryptarithmetic Problem:**

Cryptarithmetic is a type of constraint satisfaction problem where the objective is to assign digits to letters in such a way that a given arithmetic equation holds true. Each letter in the problem represents a unique digit from 0 to 9, and the same letter must always represent the same digit. A classic example of this problem is "SEND + MORE = MONEY". Here, the variables are the unique letters: S, E, N, D, M, O, R, and Y. The domain of each variable is the set of digits {0, 1, ..., 9}. Several constraints apply: First, all letters must map to unique digits—this is known as the all-different constraint. Second, the leading letters of multi-digit numbers (e.g., S in SEND and M in MORE) cannot be zero. The core constraint is the arithmetic one, where the numerical values formed by SEND, MORE, and MONEY must satisfy the equation SEND + MORE = MONEY. This requires converting the letter-based words into their numerical equivalents and ensuring the equation is mathematically correct. The solution is typically found using backtracking combined with constraint propagation**.**

**Crossword Puzzle:**

The crossword puzzle problem in the context of constraint satisfaction involves placing words on a grid in such a way that certain spatial constraints are satisfied. A simplified version of the problem involves placing one word horizontally (across) and another word vertically (down) such that they intersect at a particular position—for instance, the middle character. In this setup, the variables are the two word positions: one across and one down. The domain for each variable is a predefined list of valid words of the required length (e.g., 3-letter words). The main constraint is that the words must share the same character at the intersection point. For example, if both words intersect at their second character, the condition across[1] == down[1] must be satisfied. Additional constraints may include ensuring that the two selected words are different or that they fit within a certain structure. The solution is found by evaluating all valid combinations of words and selecting those that satisfy the overlap condition**.**

**Map Coloring Problem:**

The map coloring problem is a well-known constraint satisfaction problem where the goal is to assign colors to regions on a map such that no two adjacent regions share the same color. This problem is often used to illustrate concepts of graph coloring in artificial intelligence and constraint programming. In this problem, the variables are the regions or territories on the map—for example, the regions of Australia such as Western Australia (WA), Northern Territory (NT), South Australia (SA), and so on. The domain for each region is a set of available colors, typically three or four (e.g., red, green, and blue). The constraints ensure that any two adjacent regions—those sharing a border—do not have the same color. These constraints are binary, meaning they apply between pairs of variables. The CSP solver assigns colors to each region while checking that all adjacency constraints are satisfied. The map coloring problem is often solved using backtracking with constraint propagation techniques and is a classical example of how CSPs can model real-world geographical and scheduling problems

**Code:**

**Cryptarithmetic :**

from constraint import \*

def solve\_cryptarithmetic():

    problem = Problem()

    letters = "SENDMORY"

    problem.addVariables(letters, range(10))

    problem.addConstraint(AllDifferentConstraint())

    # Constraints for the arithmetic: SEND + MORE = MONEY

    def equation(S, E, N, D, M, O, R, Y):

        send = 1000\*S + 100\*E + 10\*N + D

        more = 1000\*M + 100\*O + 10\*R + E

        money = 10000\*M + 1000\*O + 100\*N + 10\*E + Y

        return send + more == money

    problem.addConstraint(equation, letters)

    problem.addConstraint(lambda S, M: S != 0 and M != 0, ("S", "M"))

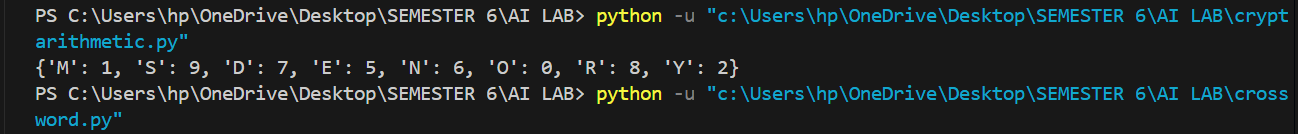
    solutions = problem.getSolutions()

    for sol in solutions:

        print(sol)

solve\_cryptarithmetic()

**Output:**



2.crossword puzzle:

from constraint import \*

def solve\_crossword():

    words = ["cat", "dog", "rat", "hat", "bat", "mat"]

    problem = Problem()

    # Define variables for across and down words

    problem.addVariable("across", words)

    problem.addVariable("down", words)

    # Constraint: overlapping letter (middle character) must match

    def overlap(a, d):

        return a[1] == d[1]

    problem.addConstraint(overlap, ("across", "down"))

    # Get all valid solutions

    solutions = problem.getSolutions()

    for i, sol in enumerate(solutions, start=1):

        across = sol["across"]

        down = sol["down"]

        print(f"\n🔹 Solution {i}: Across = {across}, Down = {down}")

        print("🧩 Crossword Grid:")

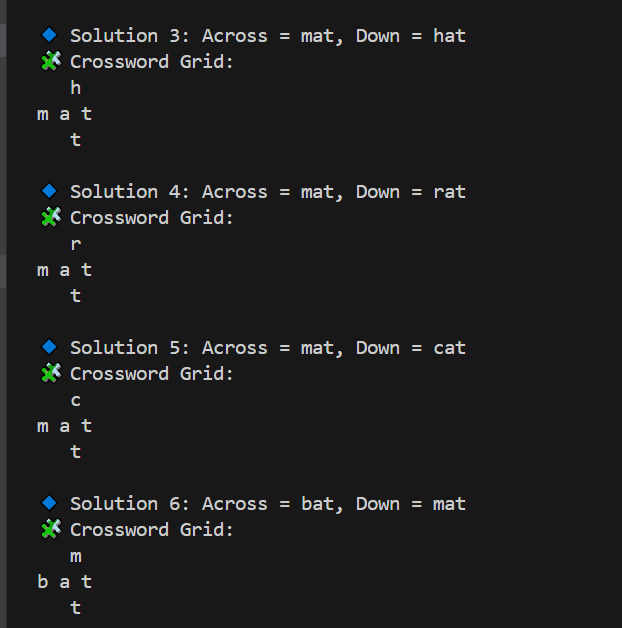
        print(" " \* 3 + down[0])

        print(f"{across[0]} {across[1]} {across[2]}")

        print(" " \* 3 + down[2])

solve\_crossword()

**Output:**



**3.mapcoloring:**

from constraint import \*

import networkx as nx

import matplotlib.pyplot as plt

def solve\_map\_coloring():

    colors = ["red", "green", "blue"]

    regions = ["WA", "NT", "SA", "Q", "NSW", "V", "T"]

    neighbors = {

        "WA": ["NT", "SA"],

        "NT": ["WA", "SA", "Q"],

        "SA": ["WA", "NT", "Q", "NSW", "V"],

        "Q": ["NT", "SA", "NSW"],

        "NSW": ["Q", "SA", "V"],

        "V": ["SA", "NSW"],

        "T": []

    }

    # Step 1: Define the problem

    problem = Problem()

    for region in regions:

        problem.addVariable(region, colors)

    # Step 2: Add constraints - neighboring regions ≠ same color

    for region in neighbors:

        for neighbor in neighbors[region]:

            problem.addConstraint(lambda r, n: r != n, (region, neighbor))

    # Step 3: Solve the CSP

    solution = problem.getSolution()

    print("Region Color Assignments:")

    for region in solution:

        print(f"{region}: {solution[region]}")

    # Step 4: Visualize using networkx

    G = nx.Graph()

    G.add\_nodes\_from(regions)

    for region in neighbors:

        for neighbor in neighbors[region]:

            G.add\_edge(region, neighbor)

    pos = nx.spring\_layout(G, seed=42)  # for consistent layout

    node\_colors = [solution[region] for region in G.nodes]

    plt.figure(figsize=(8, 6))

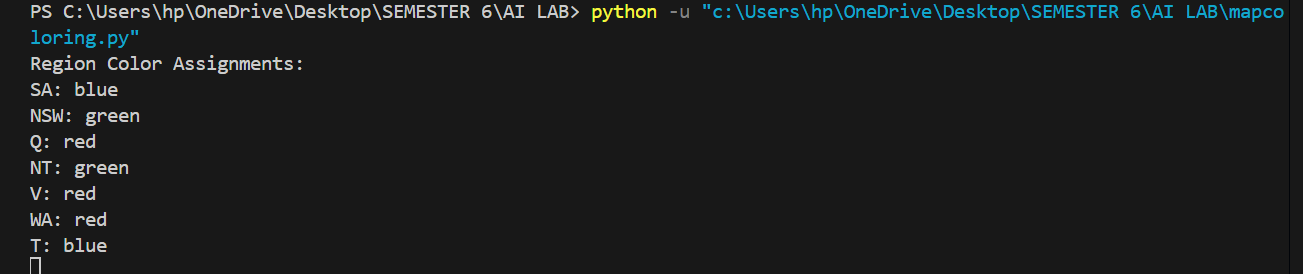
    nx.draw(G, pos, with\_labels=True, node\_color=node\_colors, node\_size=1500, font\_weight='bold', font\_color='white', edge\_color='gray')

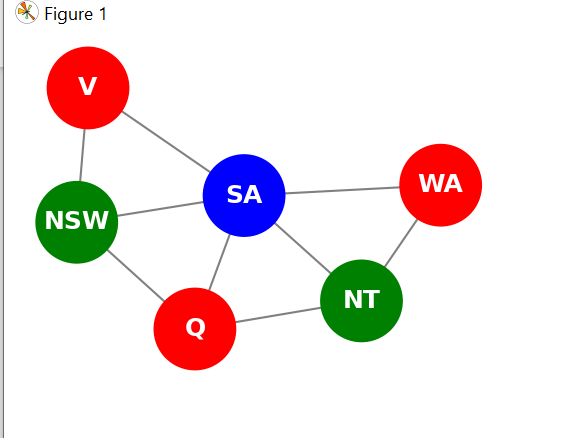
    plt.title("Australia Map Coloring using CSP")

    plt.show()

solve\_map\_coloring()

**Output**:





**Conclusion:**

Constraint Satisfaction Problems (CSPs) provide a powerful and flexible framework for solving a wide range of combinatorial problems by defining variables, assigning domains, and applying constraints to limit the solution space. Through the implementation of three classic problems—Cryptarithmetic, Crossword Puzzle, and Map Coloring—we see how CSPs can be used to model and solve real-world challenges involving logic, spatial arrangements, and resource allocation. Each problem, though different in nature, follows the same CSP structure, demonstrating the versatility of the approach. Cryptarithmetic showcases numerical constraint solving, Crossword Puzzles emphasize positional character alignment, and Map Coloring illustrates region-based conflict avoidance. Solving these problems not only deepens understanding of backtracking and constraint propagation but also highlights the practical applications of artificial intelligence in solving structured problems efficiently.