EXPERIMENT – 03

AIM:

To Implement the Constraint Satisfaction problem using backtracking

DESCRIPTION:

Constraint Satisfaction Problems (CSPs) are a class of mathematical problems where a set of variables must satisfy a set of constraints. CSPs are widely used in artificial intelligence to model various real-world problems, such as scheduling, planning, and graph coloring.

Components of a CSP:

1. **Variables:** A set of variables, each of which can take on a value from a specified domain.
2. **Domains:** The set of possible values for each variable.
3. **Constraints:** A set of conditions that must be satisfied by the values assigned to the variables.

Types of Constraints:

1. **Unary constraints:** Involve only one variable.
2. **Binary constraints:** Involve two variables.
3. **Global constraints:** Involve more than two variables.

CRYPTARITHMETIC PUZZLE

A **Crypt-Arithmetic Puzzle**, also known as a cryptogram, is a type of mathematical puzzle in which we assign digits to alphabetical letters or symbols. The end goal is to find the unique digit assignment to each letter so that the given mathematical operation holds true. In this puzzle, the equation performing an addition operation is the most commonly used. However, it also involves other arithmetic operations, such as subtraction, multiplication etc.

The rules for a Crypt-arithmetic puzzle are as follows –

- We can use digits from 0 to 9 only to represent a unique alphabetical letter in the puzzle.
- The same digit cannot be assigned to different letters in the whole equation.
- The resulting equation formed by replacing letters with digits should be mathematically correct.

EXAMPLE:

```
5  4  3  2  1
   S  E  N  D
+  M  O  R  E
   c3 c2 c1
   -------------
 M O  N  E Y
```

where c1, c2, c3 are the carry forward numbers upon addition.

1. From Column 5, M=1, since it is only carry-over possible from sum of 2 single digit number in column 4.
2. To produce a carry from column 4 to column 5 'S + M' is at least 9 so 'S=8or9' so 'S+M=9or10' & so 'O=0or1' . But 'M=1', so '0=0'.
3. If there is carry from column 3 to 4 then 'E=9' & so 'N=0'. But 'O=0' so there is no carry & 'S=9' & 'c3=0'.
4. If there is no carry from column 2 to 3 then 'E=N' which is impossible, therefore there is carry & 'N=E+1' & 'c2=1'.
5. If there is carry from column 1 to 2 then 'N+R=E mod 10' & 'N=E+1' so 'E+1+R=E mod 10', so 'R=9' but 'S=9', so there must be carry from column 1 to 2. Therefore 'c1=1' & 'R=8'.
6. To produce carry 'c1=1' from column 1 to 2, we must have 'D+E=10+Y' as Y cannot be 0/1 so D+E is at least 12. As D is at most 7 & E is at least 5(D cannot be 8 or 9 as it is already assigned). N is atmost 7 & 'N=E+1' so 'E=5or6'.
7. If E were 6 & D+E atleast 12 then D would be 7, but 'N=E+1' & N would also be 7 which is impossible. Therefore 'E=5' & 'N=6'.
8. D+E is atleast 12 for that we get 'D=7' & 'Y=2'.

```
Solution:  9 5 6 7
         + 1 0 8 5
           ----------
         1 0 6 5 2
```

Values: S=9, E=5, N=6, D=7, M=1, O=0, R=8, Y=2

ALGORITHM:

1. Start by examining the rightmost digit of the topmost row, with a carry of 0
2. If we are beyond the leftmost digit of the puzzle, return true if no carry, false otherwise
3. If we are currently trying to assign a char in one of the addends If char already assigned, just recur on the row beneath this one, adding value into the sum If not assigned, then
    - for (every possible choice among the digits not in use) make that choice and then on row beneath this one, if successful, return true if !successful, unmake assignment and try another digit
    - return false if no assignment worked to trigger backtracking
4. Else if trying to assign a char in the sum
5. If char assigned & matches correct, recur on next column to the left with carry, if success return true,
6. If char assigned & doesn't match, return false
7. If char unassigned & correct digit already used, return false
8. If char unassigned & correct digit unused, assign it and recur on next column to left with carry, if success return true
9. return false to trigger backtracking

CODE:

```
import itertools

def solve_cryptarithm(words, result):
    unique_letters = ''.join(set(''.join(words) + result))

    if len(unique_letters) > 10:
        print("Too many unique letters for a single-digit solution.")
        return

    digits = '0123456789'

    for perm in itertools.permutations(digits, len(unique_letters)):
        letter_to_digit = dict(zip(unique_letters, perm))

        def word_to_number(word):
            return int(''.join(letter_to_digit[letter] for letter in word))
```

```
    if any(letter_to_digit[word[0]] == '0' for word in words + [result]):
        continue

    sum_words = sum(word_to_number(word) for word in words)

    if sum_words == word_to_number(result):
        print("Solution found!")
        for word in words:
            print(f"{word} = {word_to_number(word)}")
        print(f"{result} = {word_to_number(result)}")
        print(f"Letter to Digit Mapping: {letter_to_digit}")
        return

  print("No solution found.")

input_words = input("Enter the words to sum (space-separated): ").upper().split()
input_result = input("Enter the result word: ").upper()

solve_cryptarithm(input_words, input_result)
```

OUTPUT:

```
Enter the words to sum (space-separated): BASE BALL
Enter the result word: GAMES
Solution found!
BASE = 7483
BALL = 7455
GAMES = 14938
Letter to Digit Mapping: {'B': '7', 'A': '4', 'M': '9', 'S': '8', 'E': '3', 'G': '1', 'L': '5'}
```

OUTPUT ANALYSIS:

Time Complexity:
O(10!/(10−N)!) (Factorial in N)

- Grows factorially with the number of unique letters (**N**).
- **Worst Case**: O(10!) when **N = 10**.

Space Complexity:
O(N) (Linear in N)

- Scales linearly with the number of unique letters.
- Efficient and manageable, especially since **N ≤ 10**.

SUDOKU PUZZLE

Sudoku is a number-placement puzzle where the objective is to fill a 9x9 grid with digits so that each row, each column, and each of the nine 3x3 subgrids contains all the digits from 1 to 9.

Formulating Sudoku as a CSP

1. **Variables:** Each cell in the grid is a variable.
2. **Domains:** The domain for each variable is the set of digits {1, 2, 3, 4, 5, 6, 7, 8, 9}.
3. **Constraints:**
    - **Row constraints:** Each digit can only appear once in a row.
    - **Column constraints:** Each digit can only appear once in a column.
    - **Subgrid constraints:** Each digit can only appear once in a 3x3 subgrid.

Solving Sudoku as a CSP

Various algorithms can be used to solve Sudoku as a CSP, including:

1. **Backtracking:** This is a simple recursive algorithm that tries to assign values to variables one by one. If a value assignment leads to a violation of constraints, it backtracks and tries a different value.
2. **Forward checking:** This algorithm checks after each variable assignment if any domain of a neighboring variable becomes empty. If this happens, the algorithm backtracks.
3. **Constraint propagation:** This involves identifying and enforcing constraints between variables to reduce the search space. Techniques like arc consistency can be used for this.
4. **Local search:** This involves starting with a random assignment and iteratively improving it by making local changes. Algorithms like min-conflict can be used for local search.

EXAMPLE:

	4	2	5				1	
1	5	3	2		8		4	7
	6	7	4		9			
		8	9	4	2	1		
		1				7		
		6	1	5	7	3		
			3		4	6	9	
2	1		6		5	4	7	3
	3				1	2	5	

To solve the Sudoku puzzle:

1. **Bottom-Right 3x3 Subgrid**:
   - Place 8 in the last cell of Row 9, Column 9.
   - Place 3 in Row 8, Column 8.
   - Place 7 in Row 7, Column 9.
2. **Middle Column in Right 3x3 Subgrid (Column 6)**:
   - Place 9 in the empty cell of Column 6.
3. **Middle-Left 3x3 Subgrid**:
   - Place 9 in Row 5, Column 2.
   - Place 4 in Row 5, Column 1.
4. **Middle Row (Row 5)**:
   - Place 7 in the last empty cell in this row.
5. **Complete the Remaining Cells**:
   - Fill in the remaining cells by checking the missing numbers in their respective rows, columns, and subgrids.

9	4	2	5	7	3	8	1	6
1	5	3	2	6	8	9	4	7
8	6	7	4	1	9	5	3	2
3	7	8	9	4	2	1	6	5
5	9	1	8	3	6	7	2	4
4	2	6	1	5	7	3	8	9
7	8	5	3	2	4	6	9	1
2	1	9	6	8	5	4	7	3
6	3	4	7	9	1	2	5	8

ALGORITHM:

1. Create a function that checks after assigning the current index the grid becomes unsafe or not. Keep Hashmap for a row, column and boxes. If any number has a frequency greater than 1 in the hashMap return false else return true; hashMap can be avoided by using loops.
2. Create a recursive function that takes a grid.
3. Check for any unassigned location.
    - If present then assigns a number from 1 to 9.
    - Check if assigning the number to current index makes the grid unsafe or not.
    - If safe then recursively call the function for all safe cases from 0 to 9.
    - If any recursive call returns true, end the loop and return true. If no recursive call returns true then return false.
4. If there is no unassigned location then return true.

CODE:

```
def is_valid(board, row, col, num):
    for x in range(9):
        if board[row][x] == num or board[x][col] == num:
            return False
    start_row, start_col = 3 * (row // 3), 3 * (col // 3)
    for i in range(3):
        for j in range(3):
            if board[start_row + i][start_col + j] == num:
                return False
    return True
def solve_sudoku(board):
    empty = find_empty_location(board)
    if not empty:
        return True
    row, col = empty
    for num in range(1, 10):
        if is_valid(board, row, col, num):
            board[row][col] = num
            if solve_sudoku(board):
                return True
            board[row][col] = 0
```

```
    return False

def find_empty_location(board):
    for row in range(9):
        for col in range(9):
            if board[row][col] == 0:
                return (row, col)
    return None

def print_board(board):
    for row in board:
        print(" ".join(str(num) if num != 0 else '.' for num in row))

def input_board():
    print("Enter the Sudoku puzzle (9x9 grid). Use 0 for empty cells:")
    board = []
    for i in range(9):
        while True:
            try:
                row = list(map(int, input(f"Row {i+1}: ").strip().split()))
                if len(row) != 9:
                    raise ValueError("Each row must have exactly 9 numbers.")
                if any(num < 0 or num > 9 for num in row):
                    raise ValueError("Numbers must be between 0 and 9.")
                board.append(row)
                break
            except ValueError as e:
                print(f"Invalid input: {e}. Please enter the row again.")
    return board

if __name__ == "__main__":
    board = input_board()
    if solve_sudoku(board):
        print("Solved Sudoku:")
        print_board(board)
    else:
        print("No solution exists")
```

OUTPUT:

```
Enter the Sudoku puzzle (9x9 grid). Use 0 for empty cells:
Row 1: 5 3 0 0 7 0 0 0 0
Row 2: 6 0 0 1 9 5 0 0 0
Row 3: 0 9 8 0 0 0 0 6 0
Row 4: 8 0 0 0 6 0 0 0 3
Row 5: 4 0 0 8 0 3 0 0 1
Row 6: 7 0 0 0 2 0 0 0 6
Row 7: 0 6 0 0 0 0 2 8 0
Row 8: 0 0 0 4 1 9 0 0 5
Row 9: 0 0 0 0 8 0 0 7 9

Solved Sudoku:
5 3 4 6 7 8 9 1 2
6 7 2 1 9 5 3 4 8
1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 2 3
4 2 6 8 5 3 7 9 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 4 5 2 8 6 1 7 9
```

OUTPUT ANALYSIS:

Time complexity:
O(9^(N*N)), For every unassigned index, there are 9 possible options so the time complexity is O(9^(n*n)). The time complexity remains the same but there will be some early pruning so the time taken will be much less than the naive algorithm but the upper bound time complexity remains the same.
Space Complexity:
O(N*N), To store the output array a matrix is needed.