EXPERIMENT - 1

Objective: IMPLEMENT BASIC SEARCH STRATEGIES 8-PUZZLE PROBLEM

Theory :

Given a 3×3 board with 8 tiles (each numbered from 1 to 8) and one empty space, the objective is to place the numbers to match the final configuration using the empty space. We can slide four adjacent tiles (left, right, above, and below) into the empty space.
In AI, the 8 Puzzle Problem is typically represented as a state space problem:

- State: A specific configuration of the tiles on the grid. Each unique arrangement of tiles is considered a distinct state.
- Action: A legal move that changes the position of the blank space and an adjacent tile.
- Goal Test: A condition that checks whether the current state matches the goal configuration.
- Cost: Each move has a uniform cost, typically 1 per move, making this an instance of a uniform cost search problem.

Step by step Algorithm using DFS :

1. Start from a node
2. Explore the leftmost child node recursively until you reach a leaf node or a goal state.
3. If a goal state is reached, return the solution.
4. If a leaf node is reached without finding a solution, backtrack to explore other branches.

Program :

```
from collections import deque

  # Function to display the puzzle state in a 3x3 grid
def print_state(state):
    for i in range(0, 9,
    3):
        print(state[i:i+3])
    print( )

  # Function to get possible moves from current state
def
    get_neighbors(state):
    neighbors = [ ]
    # Directions: up, down, left, right
```

```
    moves = {
        0: [1, 3],
        1: [0, 2, 4],
        2: [1, 5],
        3: [0, 4, 6],
        4: [1, 3, 5, 7],
        5: [2, 4, 8],
        6: [3, 7],
        7: [4, 6, 8],
        8: [5, 7]
    }
    blank = state.index("0") # find the blank space (represented as "0")
    for move in moves[blank]:
        new_state = list(state)
        # Swap blank with neighbor
        new_state[blank], new_state[move] = new_state[move], new_state[blank]
        neighbors.append("".join(new_state))
    return neighbors

  # Depth First Search
def dfs(start, goal):
    stack = [(start, [start])]
    visited = set()

    while stack:
        state, path = stack.pop()

        if state in visited:
            continue
        visited.add(state)

        if state == goal:
            return path

        for neighbor in get_neighbors(state):
            if neighbor not in visited:
                stack.append((neighbor, path + [neighbor]))

    return None

  # Driver code
if __name__ == "__main__":
    start_state = "125340678" # Example start state (0 is the
    blank) goal_state = "123456780"          # Goal state
```

```
print("Solving 8 Puzzle using DFS...")
solution = dfs(start_state, goal_state)

if solution:
    print("\nSteps to reach goal:")
    for step in solution:
        print_state(step)
else:
    print("No solution found.")
```

OUTPUT

Solving 8 Puzzle using DFS...

Steps to reach goal:

```
125
340
678

125
304
678

120
345
678

123
045
678

123
405
678

123
450
678

123
456
078
```

123
456
708

123
456
780

Algorithm for A* Approach :

1. Initialize a priority queue (min-heap) with the start state.
2. Keep a set of visited states to avoid repetition. While queue is not empty:

    1. Remove the state with the lowest f(n)f(n)f(n).
    2. If this state equals the goal → solution found.
    3. Otherwise, generate all possible moves (neighbors).
    4. For each neighbor:
        - Compute new g,h,fg, h, fg,h,f.
        - Push into priority queue if not visited.

Program :

```
import heapq
# Manhattan distance heuristic
def manhattan_distance(state, goal):
    distance = 0
    for i in range(1, 9): # tiles 1-8
        xi, yi = divmod(state.index(str(i)), 3)
        xg, yg = divmod(goal.index(str(i)), 3)
        distance += abs(xi - xg) + abs(yi - yg)
    return distance

  # Get neighbors of the current state
def
    get_neighbors(state):
    neighbors = []
    moves = {
        0: [1, 3],
        1: [0, 2, 4],
        2: [1, 5],
```

```
        3: [0, 4, 6],
        4: [1, 3, 5, 7],
        5: [2, 4, 8],
        6: [3, 7],
        7: [4, 6, 8],
        8: [5, 7]
    }
    blank = state.index("0")
    for move in
    moves[blank]:
        new_state = list(state)
        new_state[blank], new_state[move] = new_state[move], new_state[blank]
        neighbors.append("".join(new_state))
    return neighbors

  # A* Algorithm
def astar(start, goal):
    pq = []
    heapq.heappush(pq, (manhattan_distance(start, goal), 0, start, [start]))
    visited = set()

    while pq:
        f, g, state, path = heapq.heappop(pq)

        if state in visited:
            continue
        visited.add(state)

        if state == goal:
            return path

        for neighbor in get_neighbors(state):
            if neighbor not in visited:
                new_g = g + 1
                new_h = manhattan_distance(neighbor, goal)
                new_f = new_g + new_h
                heapq.heappush(pq, (new_f, new_g, neighbor, path + [neighbor]))
    return None

  # Print state nicely
def print_state(state):
    for i in range(0, 9,
    3):
        print(state[i:i+3])
    print()
```

```
  # Driver code
if __name__ == "__main__":
    start_state =
    "125340678"
    goal_state = "123456780"

    print("Solving 8 Puzzle using A* Search...\n")
    solution = astar(start_state, goal_state)

    if solution:
        print("Steps to reach goal (optimal path):")
        for step in solution:
            print_state(step)
        print("Total moves:", len(solution)-
    1) else:
        print("No solution found.")
```

Output :

Solving 8 Puzzle using A* Search...

Steps to reach goal (optimal path):

```
125     123       Total Moves : 8
340     450
678     678

125     123
345     456
678     078

120     123
345     456
678     708

123     123
045     456
678     780

123
405
678
```

EXPERIMENT - 2

Objective: IMPLEMENT BASIC SEARCH STRATEGIES 8-QUEENS PROBLEM

Theory :

Problem Statement

Given an 8x8 chess board, you must place 8 queens on the board so that no two queens attack each other. Print all possible matrices satisfying the conditions with positions with queens marked with '1' and empty spaces with '0'. You must solve the 8 queens problem using backtracking.

- A queen can move vertically, horizontally and diagonally in any number of steps.

Example - In the first row, the queen is at E8 square, so we have to make sure no queen is in column E and row 8 and also along its diagonals. Similarly, for the second row, the queen is on the B7 square, thus, we have to secure its horizontal, vertical, and diagonal squares. The same pattern is followed for the rest of the queens.

Output

```
0 0 0 0 1 0 0 0
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0
0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0
1 0 0 0 0 0 0 0
```

Algorithm : Backtracking Approach

Step 1: Traverse all the rows in one column at a time and try to place the queen in that position.
Step 2: After coming to a new square in the left column, traverse to its left horizontal direction to see if any queen is already placed in that row or not. If a queen is found, then move to other rows to search for a possible position for the queen.
Step 3: Like step 2, check the upper and lower left diagonals. We do not check the right side because it's impossible to find a queen on that side of the board yet.
Step 4: If the process succeeds, i.e. a queen is not found, mark the position as '1' and move ahead.
Step 5: Recursively use the above-listed steps to reach the last column. Print the solution matrix if a queen is successfully placed in the last column.
Step 6: Backtrack to find other solutions after printing one possible solution.

Code :

```
def print_solution(board, N):
    for row in board:
        print(" ".join(str(x) for x in row))
    print("\n")

def is_safe(board, row, col, N):
    # Check left side of current row
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check upper diagonal on left side
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check lower diagonal on left side
    for i, j in zip(range(row, N), range(col, -1, -1)):
        if board[i][j] == 1:
            return

    False return

    True

def solve_nqueens_util(board, col, N, solutions):
    # Base case: all queens placed
    if col >= N:
```

```
            print_solution(board, N)
            solutions[0] += 1
            return True

        res = False
        # Try placing queen in all rows of this column
        for i in range(N):
            if is_safe(board, i, col, N):
                board[i][col] = 1 # Place queen
                res = solve_nqueens_util(board, col + 1, N, solutions) or res
                board[i][col] = 0 # Backtrack

        return res

    def solve_nqueens(N):
        board = [[0] * N for _ in range(N)]
        solutions = [0] # use list to make it mutable inside recursion
        if not solve_nqueens_util(board, 0, N, solutions):
            print("No solution exists")
        else:
            print(f"Total Solutions = {solutions[0]}")

    # Solve for 8 Queens
    solve_nqueens(8)
```

Output

```
0 0 0 0 1 0 0 0
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0
0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0
1 0 0 0 0 0 0 0
```

EXPERIMENT - 3

Objective: IMPLEMENT BASIC SEARCH STRATEGIES-CRYPT ARITHMETIC

Theory :

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:

1. We can use digits from 0 to 9 only to represent a unique alphabetical letter in the puzzle.
2. 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.

Algorithm :

Identify Unique Characters: Find all unique letters in the puzzle and ensure there are no more than 10.
Assign Digits Recursively: Use a recursive function to assign a unique digit (0-9) to each unique letter one by one.
Check for Validity: After assigning a digit to every letter, check if the resulting numerical equation is correct.
Backtrack: If the equation is incorrect or a digit has been used, undo the last assignment and try another digit.
Find Solution: Repeat this process until a valid solution is found or all possibilities are exhausted.

```
Input
:
              B A S E
              B A L L
            --------
            G A M E
            S
```

In the above equation, the words namely "BASE" and "BALL" are added to produce "GAMES". The algorithm will associate each letter of the given words with a unique number from 0 to 9. For the above input, the output should be −

Letter	A	B	E	G	L	M	S
Values	4	2	1	0	5	9	6

Code :

```
class Main:


    #Set 1 when one character is assigned previously
    use = [0] * 10
    class Node:
        def init (self):
            self.letter = ''
            self.value = 0

    def isValid(self, nodeList, count, s1, s2, s3):
        val1 = 0
        val2 = 0
        val3 = 0
        m = 1
        j = 0
        i = 0
        for i in range(len(s1) - 1, -1, -1):
            ch = s1[i]
            for j in range(count):
                #when ch is present, break the loop
                if nodeList[j].letter == ch:
                    break
            val1 += m * nodeList[j].value
            m *= 10
m = 1                                       #
                                            f
                                            i
                                            n
                                            d
                                            n
                                            u
                                            m
                                            b
m = 1                                       e
                                            r
                                            f
                                            o
                                            r
                                            t
                                            h
                                            e
                                            s
                                            e
                                            c
```

```
ondstringforiinrange(len(s2)-1,-1,-1):
    ch = s2[i]
    for j in range(count):
        if node
            List[j].lettter==ch:
                break
    val2+=m*nodeList[j].val
```

```
    l
    u
    e
    m
    *
    =
    1
    0

f
    o
    r
    i
    i
    n
    r
    a
    n
    g
    e
    (
    l
    e
    n
    (
    s
    3
    )
    -
    1
    ,
    -
    1
    ,
    -
    1
    )
    :
    c
    h
    =
    s
    3
    [
    i
    ]
for j in range(count):
    i
        f
        n
        o
        d
        e
        L
        i
        s
        t
        [
        j
        ]
        .
        l
        e
        t
        t
        e
        r
        =
        =
        c
        h
        :
        b
        r
        e
        a
        k
    v
    a
    l
    3
    +
    =
    m
    *
    n
    o
    d
    e
    L
    i
    s
```

```
t
[
j
]
.
v
a
l
u
e
m
*
=
1
0
#check whether the sum is the same as the 3rd
string or not
```

```
        if val3 == (val1 + val2):
            return 1
        return 0
def permutation(self, count, nodeList, n, s1, s2, s3): if n
== count - 1: for i in range(10):
                if self.use[i] == 0:
                    nodeList[n].value = i
                    if self.isValid(nodeList, count, s1, s2, s3) == 1:
                        print("Solution found:", end='')
                         #print code, which is assigned
                        for j in range(count):
                            print(f" {nodeList[j].letter} = {nodeList[j].value}", end='')
                        return 1
            return 0

        for i in range(10):
            #for those numbers, which are not used
            if self.use[i] == 0:
                nodeList[n].value =
                i self.use[i] = 1
                if self.permutation(count, nodeList, n + 1, s1, s2, s3) == 1:
                    return 1
                 self.use[i] = 0
        return 0

    def solvePuzzle(self, s1, s2, s3):
        characters uniqueChar = 0
        len1 = len(s1)
        len2 =
        len(s2) len3
        = len(s3)
        #There are 26 different characters
        freq = [0] * 26

        for i in range(len1):
            freq[ord(s1[i]) - ord('A')] += 1
```

```
        for i in range(len2):
            freq[ord(s2[i]) - ord('A')] += 1

        for i in range(len3):
            freq[ord(s3[i]) - ord('A')] += 1

        for i in range(26):
            #whose frequency is > 0, they are present
            if freq[i] > 0:
                uniqueChar += 1
        #as there are 10 digits in the decimal system
        if uniqueChar > 10:
            print("Invalid
            strings") return 0

        nodeList = [self.Node() for _ in
        range(uniqueChar)] j = 0

        for i in range(26):
            #assign all characters found in three strings
            if freq[i] > 0:
                nodeList[j].letter = chr(i +
                ord('A')) j += 1
        return self.permutation(uniqueChar, nodeList, 0, s1, s2,

s3) if name == "main":

    main = Main()
    s1 = "BASE"
    s2 = "BALL"
    s3 =
    "GAMES"
    if main.solvePuzzle(s1, s2, s3) == 0:
        print("No solution")
```

Output -

A=4 **B**=2 **E**=1 **G**=0 **L**=5 **M**=9 **S**=6