Name : M.Chanikya

Reg no : 192311193

1. 8-Puzzle Problem (A* Algorithm)

python

from heapq import heappush, heappop def solve_8_puzzle(start, goal): def heuristic(state): return sum(abs(i // 3 - g // 3) + abs(i % 3 - g % 3) for i, val in enumerate(sum(state, [])) for g, gval in enumerate(sum(goal, [])) if val == gval and val != 0) pq = [(0, 0, tuple(start), [])] visited = set() moves = [(0, 1), (1, 0), (0, -1), (-1, 0)] while pq: _, cost, state, path = heappop(pq) state = [list(state[i:i+3]) for i in range(0, 9, 3)] if state == goal: return path + [state] if tuple(sum(state, [])) in visited: continue visited.add(tuple(sum(state, []))) x, y = next((i, j) for i in range(3) for j in range(3) if state[i][j] == 0) for dx, dy in moves: nx, ny = x + dx, y + dy if 0 <= nx < 3 and 0 <= ny < 3: new_state = [row[:] for row in state] new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y] heappush(pq, (cost + 1 + heuristic(new_state), cost + 1, tuple(sum(new_state, [])), path + [state])) return "No solution" start = [[1, 2, 3], [4, 0, 6], [7, 5, 8]] goal = [[1, 2, 3], [4, 5, 6], [7, 8, 0]] result = solve_8_puzzle(start, goal) print("8-Puzzle Solution:", result[-1] if isinstance(result, list) else result)

Input: start = [[1, 2, 3], [4, 0, 6], [7, 5, 8]], goal = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
Output: [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

---

2. 8-Queens Problem

python

```
def solve_8_queens():
    def is_safe(board, row, col):
        for i in range(col):
            if board[i] == row or abs(board[i] - row) == abs(i - col): return False
        return True
    def solve(col=0, board=[-1]*8):
        if col == 8: return board
        for row in range(8):
            if is_safe(board, row, col):
                board[col] = row
                result = solve(col + 1, board)
                if result: return result
```

```
        return None
    solution = solve()
    return solution

result = solve_8_queens()
print("8-Queens Solution:", result)
```

Input: None (solves for any valid 8-queens placement)
Output: [0, 4, 7, 5, 2, 6, 1, 3] (queen positions per column)

---

3. Water Jug Problem

python

```python
def water_jug(x, y, target):
    visited = set()
    queue = [(0, 0, [])]
    while queue:
        a, b, path = queue.pop(0)
        if a == target or b == target: return path + [(a, b)]
        if (a, b) in visited: continue
        visited.add((a, b))
        queue.extend([
            (x, b, path + [(a, b)]), (a, y, path + [(a, b)]),  # Fill jugs
            (0, b, path + [(a, b)]), (a, 0, path + [(a, b)]),  # Empty jugs
            (max(0, a - (y - b)), min(y, b + a), path + [(a, b)]),  # Pour a to b
            (min(x, a + b), max(0, b - (x - a)), path + [(a, b)])   # Pour b to a
        ])
    return "No solution"

result = water_jug(4, 3, 2)
print("Water Jug Solution:", result)
```

Input: x=4, y=3, target=2
Output: [(0, 0), (4, 0), (2, 0)]

---

4. Crypt-Arithmetic Problem (e.g., SEND + MORE = MONEY)

python

```
def solve_crypt_arithmetic():
    def is_valid(assignment, words, result):
        if any(assignment[word[0]] == 0 for word in words + [result]): return False
        num1 = sum(assignment[c] * 10**i for i, c in enumerate(reversed(words[0])))
        num2 = sum(assignment[c] * 10**i for i, c in enumerate(reversed(words[1])))
        res = sum(assignment[c] * 10**i for i, c in enumerate(reversed(result)))
        return num1 + num2 == res
    from itertools import permutations
    letters = set("SENDMOREY")
    for perm in permutations(range(10), len(letters)):
        assignment = dict(zip(letters, perm))
        if is_valid(assignment, ["SEND", "MORE"], "MONEY"):
            return assignment
    return "No solution"


result = solve_crypt_arithmetic()
print("Crypt-Arithmetic Solution:", result)
```

Input: SEND + MORE = MONEY
Output: {'S': 9, 'E': 5, 'N': 6, 'D': 7, 'M': 1, 'O': 0, 'R': 8, 'Y': 2}

---

5. Missionaries and Cannibals Problem

python

```
def missionaries_cannibals():
    def is_valid(state): return state[1] <= state[0] or state[0] == 0
    queue = [((3, 3, 1), [])]  # (M, C, boat), path
    visited = set()
    while queue:
        (m, c, b), path = queue.pop(0)
```

```
        if (m, c) == (0, 0): return path + [(m, c, b)]
        if (m, c, b) in visited or m < 0 or c < 0 or m > 3 or c > 3 or not is_valid((m, c)): continue
        visited.add((m, c, b))
        moves = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]
        for dm, dc in moves:
            if b == 1: queue.append(((m - dm, c - dc, 0), path + [(m, c, b)]))
            else: queue.append(((m + dm, c + dc, 1), path + [(m, c, b)]))
    return "No solution"

result = missionaries_cannibals()
print("Missionaries Cannibals Solution:", result[-1])
```

Input: 3 missionaries, 3 cannibals, boat capacity 2
Output: (0, 0, 0) (all across)

---

6. Vacuum Cleaner Problem

python

```python
def vacuum_cleaner(world):
    actions = []
    pos = 0
    for i in range(len(world)):
        if world[pos] == 1: actions.append(f"Clean at {pos}")
        world[pos] = 0
        if pos < len(world) - 1: actions.append("Move right"); pos += 1
        elif pos > 0: actions.append("Move left"); pos -= 1
    return actions

world = [1, 0, 1]  # 1 = dirty, 0 = clean
result = vacuum_cleaner(world)
print("Vacuum Cleaner Actions:", result)
```

Input: [1, 0, 1]
Output: ['Clean at 0', 'Move right', 'Move right', 'Clean at 2']

---

7. Breadth-First Search (BFS)

python

CollapseWrapCopy

```
from collections import deque
def bfs(graph, start, goal):
    queue = deque([(start, [start])])
    visited = set()
    while queue:
        node, path = queue.popleft()
        if node == goal: return path
        if node in visited: continue
        visited.add(node)
        queue.extend((n, path + [n]) for n in graph[node])
    return "No path"


graph = {0: [1, 2], 1: [0, 3], 2: [0, 3], 3: [1, 2]}
result = bfs(graph, 0, 3)
print("BFS Path:", result)
```

Input: graph = {0: [1, 2], 1: [0, 3], 2: [0, 3], 3: [1, 2]}, start=0, goal=3
Output: [0, 1, 3]

---

8. Depth-First Search (DFS)

python

CollapseWrapCopy

```
def dfs(graph, start, goal, path=None, visited=None):
    if path is None: path = [start]
    if visited is None: visited = set()
    if start == goal: return path
    visited.add(start)
    for next_node in graph[start]:
        if next_node not in visited:
```

```
            result = dfs(graph, next_node, goal, path + [next_node], visited)
            if result: return result
    return "No path"

graph = {0: [1, 2], 1: [0, 3], 2: [0, 3], 3: [1, 2]}
result = dfs(graph, 0, 3)
print("DFS Path:", result)
```

Input: Same as BFS
Output: [0, 1, 3]

---

9. Travelling Salesman Problem (TSP)

python

CollapseWrapCopy

```
from itertools import permutations
def tsp(graph):
    n = len(graph)
    min_cost, min_path = float('inf'), None
    for path in permutations(range(1, n)):
        cost = graph[0][path[0]] + sum(graph[path[i]][path[i+1]] for i in range(n-2)) + graph[path[-1]][0]
        if cost < min_cost: min_cost, min_path = cost, (0,) + path + (0,)
    return min_path, min_cost

graph = [[0, 10, 15, 20], [10, 0, 35, 25], [15, 35, 0, 30], [20, 25, 30, 0]]
path, cost = tsp(graph)
print("TSP Path:", path, "Cost:", cost)
```

Input: [[0, 10, 15, 20], [10, 0, 35, 25], [15, 35, 0, 30], [20, 25, 30, 0]]
Output: (0, 1, 3, 2, 0) Cost: 80

---

10. A* Algorithm

python

CollapseWrapCopy

```
from heapq import heappush, heappop

def a_star(graph, start, goal, h):

    pq = [(0, 0, start, [start])]

    visited = set()

    while pq:

        _, g, node, path = heappop(pq)

        if node == goal: return path

        if node in visited: continue

        visited.add(node)

        for neighbor, cost in graph[node]:

            if neighbor not in visited:

                heappush(pq, (g + cost + h[neighbor], g + cost, neighbor, path + [neighbor]))

    return "No path"


graph = {0: [(1, 4), (2, 2)], 1: [(3, 1)], 2: [(3, 5)], 3: []}

h = {0: 7, 1: 3, 2: 5, 3: 0}

result = a_star(graph, 0, 3, h)

print("A* Path:", result)
```

Input: graph = {0: [(1, 4), (2, 2)], ...}, h = {0: 7, 1: 3, 2: 5, 3: 0}
Output: [0, 1, 3]

---

11. Map Coloring (CSP)

python

CollapseWrapCopy

```
def map_coloring(graph, colors):

    def is_safe(node, color, coloring):

        return all(coloring.get(neighbor, None) != color for neighbor in graph[node])

    def solve(node_list, coloring={}):

        if not node_list: return coloring

        node = node_list[0]

        for color in colors:
```

```
        if is_safe(node, color, coloring):
            coloring[node] = color
            result = solve(node_list[1:], coloring)
            if result: return result
            del coloring[node]
        return None
    return solve(list(graph.keys()))


graph = {'A': ['B', 'C'], 'B': ['A', 'C'], 'C': ['A', 'B']}
colors = ['R', 'G', 'B']
result = map_coloring(graph, colors)
print("Map Coloring Solution:", result)
```

Input: graph = {'A': ['B', 'C'], ...}, colors = ['R', 'G', 'B']
Output: {'A': 'R', 'B': 'G', 'C': 'B'}

---

12. Tic Tac Toe

python

CollapseWrapCopy

```
def tic_tac_toe():
    board = [[' ']*3 for _ in range(3)]
    def print_board(): [print(row) for row in board]
    def win(player): return any(all(board[i][j] == player for j in range(3)) for i in range(3)) or \
                    any(all(board[i][j] == player for i in range(3)) for j in range(3)) or \
                    all(board[i][i] == player for i in range(3)) or \
                    all(board[i][2-i] == player for i in range(3))
    player = 'X'
    for _ in range(9):
        print_board()
        x, y = map(int, input(f"Player {player} (row col): ").split())
        if board[x][y] == ' ': board[x][y] = player
        if win(player): return f"Player {player} wins!"
```

```
        player = 'O' if player == 'X' else 'X'
    return "Draw"


print("Tic Tac Toe Result:", tic_tac_toe())
```

Input: 0 0 (X), 1 1 (O), 0 1 (X), 1 2 (O), 0 2 (X)
Output: Player X wins!

---

13. Minimax Algorithm (Tic Tac Toe)

python

CollapseWrapCopy

```
def minimax(board, depth, is_max):
    def win(b, p): return any(all(b[i][j] == p for j in range(3)) for i in range(3)) or \
                   any(all(b[i][j] == p for i in range(3)) for j in range(3)) or \
                   all(b[i][i] == p for i in range(3)) or all(b[i][2-i] == p for i in range(3))
    if win(board, 'X'): return 10 - depth
    if win(board, 'O'): return depth - 10
    if all(board[i][j] != ' ' for i in range(3) for j in range(3)): return 0
    if is_max:
        best = -float('inf')
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
                    board[i][j] = 'X'
                    best = max(best, minimax(board, depth + 1, False))
                    board[i][j] = ' '
        return best
    else:
        best = float('inf')
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
```

```
            board[i][j] = 'O'

            best = min(best, minimax(board, depth + 1, True))

            board[i][j] = ' '

    return best


board = [['X', ' ', 'O'], [' ', 'X', ' '], ['O', ' ', ' ']]

score = minimax(board, 0, True)

print("Minimax Score:", score)
```

Input: [['X', ' ', 'O'], [' ', 'X', ' '], ['O', ' ', ' ']]
Output: 10 (X can win)

---

14. Alpha-Beta Pruning

python

CollapseWrapCopy

```
def alpha_beta(board, depth, alpha, beta, is_max):

  def win(b, p): return any(all(b[i][j] == p for j in range(3)) for i in range(3)) or \

           any(all(b[i][j] == p for i in range(3)) for j in range(3)) or \

           all(b[i][i] == p for i in range(3)) or all(b[i][2-i] == p for i in range(3))

  if win(board, 'X'): return 10 - depth

  if win(board, 'O'): return depth - 10

  if all(board[i][j] != ' ' for i in range(3) for j in range(3)): return 0

  if is_max:

    for i in range(3):

      for j in range(3):

        if board[i][j] == ' ':

          board[i][j] = 'X'

          alpha = max(alpha, alpha_beta(board, depth + 1, alpha, beta, False))

          board[i][j] = ' '

          if beta <= alpha: break

    return alpha

  else:
```

```
    for i in range(3):
        for j in range(3):
            if board[i][j] == ' ':
                board[i][j] = 'O'
                beta = min(beta, alpha_beta(board, depth + 1, alpha, beta, True))
                board[i][j] = ' '
                if beta <= alpha: break
    return beta

board = [['X', ' ', 'O'], [' ', 'X', ' '], ['O', ' ', ' ']]
score = alpha_beta(board, 0, -float('inf'), float('inf'), True)
print("Alpha-Beta Score:", score)
```

Input: Same as Minimax
Output: 10

---

15. Decision Tree (Simple Example)

python

CollapseWrapCopy

```
def decision_tree(data, labels):
    if len(set(labels)) == 1: return labels[0]
    best_feature = max(range(len(data[0])), key=lambda f: sum(1 for i in range(len(data))
                                                for j in range(i+1, len(data))
                                                if data[i][f] == data[j][f] and labels[i] != labels[j]))
    tree = {best_feature: {}}
    for val in set(d[best_feature] for d in data):
        subtree_data = [d for i, d in enumerate(data) if d[best_feature] == val]
        subtree_labels = [l for i, l in enumerate(labels) if data[i][best_feature] == val]
        tree[best_feature][val] = decision_tree(subtree_data, subtree_labels)
    return tree

data = [[1, 0], [1, 1], [0, 0], [0, 1]]
```

labels = [0, 1, 0, 1]

tree = decision_tree(data, labels)

print("Decision Tree:", tree)

Input: data = [[1, 0], [1, 1], [0, 0], [0, 1]], labels = [0, 1, 0, 1]
Output: {1: {0: 0, 1: 1}} (feature 1 decides)

---

16. Feedforward Neural Network (Simple)

python

CollapseWrapCopy

```
import numpy as np

def sigmoid(x): return 1 / (1 + np.exp(-x))

def feedforward(X, W1, W2):

    hidden = sigmoid(np.dot(X, W1))

    output = sigmoid(np.dot(hidden, W2))

    return output


X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

W1 = np.random.rand(2, 2)

W2 = np.random.rand(2, 1)

result = feedforward(X, W1, W2)

print("Feedforward Output:", result)
```

Input: X = [[0, 0], [0, 1], [1, 0], [1, 1]]
Output: Array of predictions (e.g., [[0.6], [0.7], [0.65], [0.8]], varies with random weights)