Knowledge is Supreme

TOLANI COLLEGE OF COMMERCE (AUTONOMOUS)

150-151, Sher-E-Punjab Society Guru Gobind Singh Road,
Andheri East, Mumbai, Maharashtra 400 093

Department of B.Sc. (Information Technology)

CERTIFICATE

This is to certify that Mr. / Ms.____bearing Roll No_______ have completed the practicals in the Course of_________________________in accordance with the syllabus of B.Sc. (Information Technology) Programme of Semester_____ as prescribed by the Tolani College of Commerce (Autonomous) in the academic year 2024-2025.

Amit
25/09/2024

Internal Examiner **Programme Coordinator**

External Examiner

Date: 25 – SEP - 2024 **College Seal**

Index

Sr No.	Practicals	Date	Sign
1	Write a program to implement depth first search algorithm.	03-08-2024	
2	Write a program to implement breadth first search algorithm.	03-08-2024	
3	Write a program to simulate 4-Queen / N-Queen problem.	10-08-2024	
4	Write a program to solve tower of Hanoi problem.	10-08-2024	
5	Write a program to implement alpha beta search.	17-08-2024	
6	Write a program for Hill climbing problem.	17-08-2024	
7	Write a program to implement A* algorithm.	17-08-2024	
8	Write a program to implement AO* algorithm.	24-08-2024	
9	Write a program to solve water jug problem.	24-08-2024	
10	Design the simulation of tic – tac – toe game using min-max algorithm.	24-08-2024	
11	Write a program to solve Missionaries and Cannibals problem.	31-08-2024	
12	Design an application to simulate number puzzle problem.	31-08-2024	
13	Write a program to shuffle Deck of cards.	31-08-2024	
14	Solve traveling salesman problem using artificial intelligence technique.	14-09-2024	
15	Solve the block of World problem.	14-09-2024	
16	Solve constraint satisfaction problem	14-09-2024	
17	Derive the expressions based on Associative law	21-09-2024	
18	Derive the expressions based on Distributive law	21-09-2024	
19	Write a program to derive the predicate.	21-09-2024	

Q 1) Write a program to implement depth first search algorithm.

```
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, node, neighbor):
        if node not in self.graph:
            self.graph[node] = []
        if neighbor not in self.graph:
            self.graph[neighbor] = []
        self.graph[node].append(neighbor)
        self.graph[neighbor].append(node)  # For undirected graph

    def dfs(self, start):
        visited = set()
        self._dfs_util(start, visited)

    def _dfs_util(self, node, visited):
        visited.add(node)
        print(node)  # You can replace this with any action you want to perform on the node

        for neighbor in self.graph[node]:
            if neighbor not in visited:
                self._dfs_util(neighbor, visited)

g = Graph()
g.add_edge('A', 'B')
g.add_edge('A', 'C')
g.add_edge('B', 'D')
g.add_edge('B', 'E')
g.add_edge('C', 'F')
g.add_edge('E', 'F')

start_node = 'A'
g.dfs(start_node)
```

Output :-

```
A
B
D
E
F
C
```

Q 2) Write a program to implement breadth first search algorithm.

```
def breadth_first_search(graph, start):
    visited = set()
    queue = [start]
    visited.add(start)

    while queue:
        node = queue.pop(0)
        print(node)  # You can replace this with any action you want to perform on the node

        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

    return visited

graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

start_node = 'A'
visited_nodes = breadth_first_search(graph, start_node)
```

Output:-

```
A
B
C
D
E
F
```

Q 3) Write a program to simulate 4-Queen / N-Queen problem.

```
class NQueens:
    def __init__(self, n):
        self.n = n
        self.board = [[0] * n for _ in range(n)]

    def print_board(self):
        for row in self.board:
            print(" ".join("Q" if col else "." for col in row))
        print()

    def is_safe(self, row, col):
        # Check this row on left side
        for i in range(col):
            if self.board[row][i]:
                return False

        for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
            if self.board[i][j]:
                return False

        for i, j in zip(range(row, self.n, 1), range(col, -1, -1)):
            if self.board[i][j]:
                return False

        return True

    def solve_n_queens_util(self, col):
        if col >= self.n:
            return True

        for i in range(self.n):
            if self.is_safe(i, col):
                self.board[i][col] = 1
                if self.solve_n_queens_util(col + 1):
                    return True
                self.board[i][col] = 0  # Backtrack

        return False

    def solve(self):
        if not self.solve_n_queens_util(0):
            print("Solution does not exist")
            return False

        self.print_board()
        return True
```

```
n = 4
n_queens = NQueens(n)
n_queens.solve()
```

Output:-

```
. . Q .
Q . . .
. . . Q
. Q . .
```

Q 4) Write a program to solve tower of Hanoi problem.

```
class TowerOfHanoi:
    def __init__(self, n):
        self.n = n

    def solve(self):
        self._move_disks(self.n, 'A', 'B', 'C')

    def _move_disks(self, n, source, auxiliary, destination):
        if n == 1:
            self._print_move(1, source, destination)
            return

        self._move_disks(n - 1, source, destination, auxiliary)

        self._print_move(n, source, destination)

        self._move_disks(n - 1, auxiliary, source, destination)

    def _print_move(self, disk, source, destination):
        print(f"Move disk {disk} from {source} to {destination}")

if __name__ == "__main__":
    n = 3
    hanoi = TowerOfHanoi(n)
    hanoi.solve()
```

Output:-

Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C

Q 5) Write a program to implement alpha beta search.

```
MAX = float('inf')
MIN = float('-inf')

def minmax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):
    if depth == 3:
        return values[nodeIndex]

    if maximizingPlayer:
        best = MIN
        for i in range(2):  # Assuming binary tree with two children
            val = minmax(depth + 1, nodeIndex * 2 + i, False, values, alpha, beta)
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha:
                break
        return best
    else:
        best = MAX
        for i in range(2):  # Assuming binary tree with two children
            val = minmax(depth + 1, nodeIndex * 2 + i, True, values, alpha, beta)
            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha:
                break
        return best

if __name__ == "__main__":
    values = [3, 5, 6, 9, 1, 2, 0, -1]
    print("The optimal value is:", minmax(0, 0, True, values, MIN, MAX))
```

Output:-

The optimal value is: 5

Q 6) Write a program for Hill climbing problem.

```
def hill_climbing_min_cost(cost):
    n = len(cost)
    if n == 0:
        return 0
    if n== 1:
        return cost[0]

    cost_from_start = [float('inf')] * n
    cost_from_start[0] = cost[0]
    cost_from_start[1] = cost[1]

    i = 0
    while i < n:
        if i + 1 < n:
            cost_from_start[i+1] = min(cost_from_start[i+1],cost_from_start[i] + cost[i+1])
        if i + 2 < n:
            cost_from_start[i+2] = min(cost_from_start[i+2],cost_from_start[i] + cost[i+2])
        i += 1

    return min(cost_from_start[n-1],cost_from_start[n-2])

cost = [10,15,20]
print(hill_climbing_min_cost(cost))
```

Output:-

```
15
```

Q 7) Write a program to implement A* algorithm.

```
import heapq

class PuzzleNode:
    def __init__(self, state, parent=None, move=None, depth=0):
        self.state = state
        self.parent = parent
        self.move = move
        self.depth = depth
        self.cost = 0  # g(n)
        self.heuristic = 0  # h(n)
        self.total = 0  # f(n) = g(n) + h(n)

    def __lt__(self, other):
        return self.total < other.total

def goal_state(size):
    return tuple(tuple((i * size + j + 1) % (size * size) for j in range(size)) for i in range(size))

goal = goal_state(3)

start_state = ((7, 2, 4),
               (5, 0, 6),
               (8, 3, 1))

def manhattan_heuristic(state, goal):
    size = len(state)
    distance = 0
    for i in range(size):
        for j in range(size):
            if state[i][j] != 0:
                x, y = divmod(state[i][j] - 1, size)
                distance += abs(i - x) + abs(j - y)
    return distance

def get_neighbors(node):
    neighbors = []
    size = len(node.state)
    x, y = [(ix, iy) for ix, row in enumerate(node.state) for iy, i in enumerate(row) if i == 0][0]
    directions = {'Up': (x - 1, y), 'Down': (x + 1, y), 'Left': (x, y - 1), 'Right': (x, y + 1)}

    for move, (new_x, new_y) in directions.items():
        if 0 <= new_x < size and 0 <= new_y < size:
            new_state = [list(row) for row in node.state]
            new_state[x][y], new_state[new_x][new_y] = new_state[new_x][new_y], new_state[x][y]
            neighbors.append(PuzzleNode(tuple(tuple(row) for row in new_state), node, move,
node.depth + 1))
    return neighbors

def reconstruct_puzzle_path(node):
```

```
    path = []
    while node:
        path.append((node.move, node.state))
        node = node.parent
    return path[::-1]

def astar_puzzle(start, goal):
    open_list = []
    closed_set = set()
    start_node = PuzzleNode(start)
    goal_node = goal
    start_node.heuristic = manhattan_heuristic(start, goal_node)
    start_node.total = start_node.heuristic
    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)
        if current_node.state == goal_node:
            return reconstruct_puzzle_path(current_node)
        closed_set.add(current_node.state)

        for neighbor in get_neighbors(current_node):
            if neighbor.state in closed_set:
                continue
            neighbor.cost = current_node.depth + 1
            neighbor.heuristic = manhattan_heuristic(neighbor.state, goal_node)
            neighbor.total = neighbor.cost + neighbor.heuristic
            heapq.heappush(open_list, neighbor)

    return None  # No solution found

solution = astar_puzzle(start_state, goal)

for move, state in solution:
    print(f"Move: {move}")
    for row in state:
        print(row)
    print()
```

Output:-

```
Move: None
(7, 2, 4)
(5, 0, 6)
(8, 3, 1)

Move: Down
(7, 2, 4)
(5, 3, 6)
(8, 0, 1)

Move: Right
(7, 2, 4)
```

```
(5, 3, 6)
(8, 1, 0)

Move: Up
(7, 2, 4)
(5, 3, 0)
(8, 1, 6)

Move: Left
(7, 2, 4)
(5, 0, 3)
(8, 1, 6)

Move: Left
(7, 2, 4)
(0, 5, 3)
(8, 1, 6)

Move: Up
(0, 2, 4)
(7, 5, 3)
(8, 1, 6)

Move: Right
(2, 0, 4)
(7, 5, 3)
(8, 1, 6)

Move: Right
(2, 4, 0)
(7, 5, 3)
(8, 1, 6)

Move: Down
(2, 4, 3)
(7, 5, 0)
(8, 1, 6)

Move: Left
(2, 4, 3)
(7, 0, 5)
(8, 1, 6)

Move: Down
(2, 4, 3)
(7, 1, 5)
(8, 0, 6)

Move: Left
(2, 4, 3)
(7, 1, 5)
(0, 8, 6)

Move: Up
(2, 4, 3)
(0, 1, 5)
```

```
(7, 8, 6)

Move: Right
(2, 4, 3)
(1, 0, 5)
(7, 8, 6)

Move: Up
(2, 0, 3)
(1, 4, 5)
(7, 8, 6)

Move: Left
(0, 2, 3)
(1, 4, 5)
(7, 8, 6)

Move: Down
(1, 2, 3)
(0, 4, 5)
(7, 8, 6)

Move: Right
(1, 2, 3)
(4, 0, 5)
(7, 8, 6)

Move: Right
(1, 2, 3)
(4, 5, 0)
(7, 8, 6)

Move: Down
(1, 2, 3)
(4, 5, 6)
(7, 8, 0)
```

Q 8) Write a program to implement AO* algorithm.

```
class Node:
    def __init__(self, name, is_goal=False):
        self.name = name
        self.is_goal = is_goal
        self.successors = []
        self.cost = float('inf')
        self.best_successor = None
        self.visited = False

    def add_successor(self, successor, relation='OR', cost=1):
        self.successors.append((successor, relation, cost))

class AOStarAlgorithm:
    def __init__(self):
        self.nodes = {}

    def add_node(self, name, is_goal=False):
        node = Node(name, is_goal)
        self.nodes[name] = node

    def add_edge(self, from_node, to_node, relation='OR', cost=1):
        self.nodes[from_node].add_successor(self.nodes[to_node], relation, cost)

    def search(self, start_node)
        node = self.nodes[start_node]
        self.ao_star_util(node)

    def ao_star_util(self, node):
        if node.is_goal:
            node.cost = 0
            return node.cost

        if node.visited:
            return node.cost

        node.visited = True
        min_cost = float('inf')
        best_successor = None

        for successor, relation, cost in node.successors:
            if relation == 'OR':
                current_cost = self.ao_star_util(successor) + cost
                if current_cost < min_cost:
                    min_cost = current_cost
                    best_successor = (successor, relation)

            elif relation == 'AND':
                current_cost = cost
                for and_successor, _, and_cost in successor.successors:
```

```
                current_cost += self.ao_star_util(and_successor)

            if current_cost < min_cost:
                min_cost = current_cost
                best_successor = (successor, relation)

        node.cost = min_cost
        node.best_successor = best_successor

        return node.cost

    def print_solution(self, node_name):
        node = self.nodes[node_name]
        if node.is_goal:
            print(f"Goal node: {node.name}")
            return


        if node.best_successor:
            print(f"Node: {node.name} -> Successor: {node.best_successor[0].name} via
{node.best_successor[1]} relation")
            self.print_solution(node.best_successor[0].name)

if __name__ == "__main__":
    ao_star = AOStarAlgorithm()
   ao_star.add_node('A')
    ao_star.add_node('B', is_goal=True)
    ao_star.add_node('C')
    ao_star.add_node('D', is_goal=True)
   ao_star.add_edge('A', 'B', 'OR', 4)
    ao_star.add_edge('A', 'C', 'OR', 2)
    ao_star.add_edge('C', 'D', 'OR', 1)
   ao_star.search('A')


    ao_star.print_solution(‘A')
```

Output:-

```
Node: A -> Successor: C via OR relation
Node: C -> Successor: D via OR relation
Goal node: D
```

Q 9) Write a program to solve water jug problem.

```
from collections import deque

def is_measurable(m, n, d):
    if d % gcd(m, n) != 0:
        return False
    return d <= max(m, n)

def gcd(a, b):
    while b:
        a, b = b, a % b
    return a

def water_jug_solver(m, n, d):
    if not is_measurable(m, n, d):
        print("It is not possible to measure the desired amount.")
        return

    visited = set()
    queue = deque([(0, 0)])
    parent = {}

    while queue:
        jug1, jug2 = queue.popleft()

        if jug1 == d or jug2 == d:
            print_solution(jug1, jug2, parent)
            return

        if (jug1, jug2) in visited:
            continue

        visited.add((jug1, jug2))

        if (m, jug2) not in visited:
            queue.append((m, jug2))
            parent[(m, jug2)] = (jug1, jug2)

        if (jug1, n) not in visited:
            queue.append((jug1, n))
            parent[(jug1, n)] = (jug1, jug2)

        if (0, jug2) not in visited:
            queue.append((0, jug2))
            parent[(0, jug2)] = (jug1, jug2)

        if (jug1, 0) not in visited:
            queue.append((jug1, 0))
            parent[(jug1, 0)] = (jug1, jug2)
```

```
        pour_to_jug2 = min(jug1, n - jug2)
        new_jug1 = jug1 - pour_to_jug2
        new_jug2 = jug2 + pour_to_jug2
        if (new_jug1, new_jug2) not in visited:
            queue.append((new_jug1, new_jug2))
            parent[(new_jug1, new_jug2)] = (jug1, jug2)

        pour_to_jug1 = min(jug2, m - jug1)
        new_jug1 = jug1 + pour_to_jug1
        new_jug2 = jug2 - pour_to_jug1
        if (new_jug1, new_jug2) not in visited:
            queue.append((new_jug1, new_jug2))
            parent[(new_jug1, new_jug2)] = (jug1, jug2)

    print("It is not possible to measure the desired amount.")

def print_solution(jug1, jug2, parent):
    path = []
    state = (jug1, jug2)
    while state in parent:
        path.append(state)
        state = parent[state]
    path.append((0, 0))
    path.reverse()

    for i, step in enumerate(path):
        print(f"Step {i}: Jug1 = {step[0]} liters, Jug2 = {step[1]} liters")

if __name__ == "__main__":
    m = 5
    n = 3
    d = 4
    water_jug_solver(m, n, d)
```

Output:-

Step 0: Jug1 = 0 liters, Jug2 = 0 liters
Step 1: Jug1 = 5 liters, Jug2 = 0 liters
Step 2: Jug1 = 2 liters, Jug2 = 3 liters
Step 3: Jug1 = 2 liters, Jug2 = 0 liters
Step 4: Jug1 = 0 liters, Jug2 = 2 liters
Step 5: Jug1 = 5 liters, Jug2 = 2 liters
Step 6: Jug1 = 4 liters, Jug2 = 3 liters

Q 10) Design the simulation of tic – tac – toe game using min-max algorithm.

```
import math

HUMAN = 'O'
AI = 'X'

def print_board(board):
    for row in board:
        print("|".join(row))
        print("-" * 5)

def is_moves_left(board):
    for row in board:
        if '_' in row:
            return True
    return False

def evaluate(board):
    for row in board:
        if row[0] == row[1] == row[2]:
            if row[0] == AI:
                return 10
            elif row[0] == HUMAN:
                return -10
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col]:
            if board[0][col] == AI:
                return 10
            elif board[0][col] == HUMAN:
                return -10
    if board[0][0] == board[1][1] == board[2][2]:
        if board[0][0] == AI:
            return 10
        elif board[0][0] == HUMAN:
            return -10
    if board[0][2] == board[1][1] == board[2][0]:
        if board[0][2] == AI:
            return 10
        elif board[0][2] == HUMAN:
            return -10
    return 0

def minimax(board, depth, is_max):
    score = evaluate(board)
    if score == 10:
        return score - depth
    if score == -10:
        return score + depth
    if not is_moves_left(board):
        return 0
```

```
    if is_max:
        best = -math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == '_':
                    board[i][j] = AI
                    best = max(best, minimax(board, depth + 1, not is_max))
                    board[i][j] = '_'
        return best
    else:
        best = math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == '_':
                    board[i][j] = HUMAN
                    best = min(best, minimax(board, depth + 1, not is_max))
                    board[i][j] = '_'
        return best

def find_best_move(board):
    best_val = -math.inf
    best_move = (-1, -1)
    for i in range(3):
        for j in range(3):
            if board[i][j] == '_':
                board[i][j] = AI
                move_val = minimax(board, 0, False)
                board[i][j] = '_'
                if move_val > best_val:
                    best_move = (i, j)
                    best_val = move_val
    return best_move

def play_game():
    board = [
        ['_', '_', '_'],
        ['_', '_', '_'],
        ['_', '_', '_']
    ]
    print("Initial board:")
    print_board(board)
    while is_moves_left(board) and evaluate(board) == 0:
        num = int(input("Enter position (1-9) to place your 'O': ")) - 1
        row = num // 3
        col = num % 3
        if board[row][col] == '_':
            board[row][col] = HUMAN
            print("\nBoard after Human's move:")
            print_board(board)
        else:
            print("Invalid move! Try again.")
```

```
            continue
        if evaluate(board) != 0 or not is_moves_left(board):
            break

print("============================================================================================")
        print("\nAI is making a move...")
        best_move = find_best_move(board)
        board[best_move[0]][best_move[1]] = AI
        print("\nBoard after AI's move:")
        print_board(board)

print("============================================================================================")
    score = evaluate(board)
    if score == 10:
        print("AI wins!")
    elif score == -10:
        print("Human wins!")
    else:
        print("It's a draw!")

if __name__ == "__main__":
    play_game()
```

Output:-

```
Initial board:
 | |_
-----
 | |_
-----
 | |_
-----
Enter position (1-9) to place your 'O':  5

Board after Human's move:
 | |_
-----
 |O|_
-----
 | |_
-----
============================================================================================

AI is making a move...

Board after AI's move:
X| |_
-----
 |O|_
-----
 | |_
-----
```

```
================================================================================
================================
Enter position (1-9) to place your 'O':  3

Board after Human's move:
X| |O
-----
 |O|_
-----
 | |_
-----
================================================================================
================================

AI is making a move...

Board after AI's move:
X| |O
-----
 |O|_
-----
X| |_
-----
================================================================================
================================
Enter position (1-9) to place your 'O':  4

Board after Human's move:
X| |O
-----
O|O|_
-----
X| |_
-----
================================================================================
================================

AI is making a move...

Board after AI's move:
X| |O
-----
O|O|X
-----
X| |_
-----
================================================================================
================================
Enter position (1-9) to place your 'O':  2

Board after Human's move:
X|O|O
-----
O|O|X
-----
X| |_
-----
================================================================================
================================
```

```
AI is making a move...

Board after AI's move:
X|O|O
-----
O|O|X
-----
X|X|_
-----
==========================================================================================
Enter position (1-9) to place your 'O':  9

Board after Human's move:
X|O|O
-----
O|O|X
-----
X|X|O
-----
It's a draw!
```

Q 11) Write a program to solve Missionaries and Cannibals problem.

```
from collections import deque

def is_valid_state(missionaries, cannibals):
    if missionaries < 0 or cannibals < 0 or missionaries > 3 or cannibals > 3:
        return False
    if missionaries > 0 and missionaries < cannibals:
        return False
    if (3 - missionaries) > 0 and (3 - missionaries) < (3 - cannibals):
        return False
    return True

def generate_successors(state):
    successors = []
    m, c, boat = state
    moves = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]

    if boat == 1:
        for move in moves:
            new_m, new_c = m - move[0], c - move[1]
            if is_valid_state(new_m, new_c):
                successors.append((new_m, new_c, 0))
    else:
        for move in moves:
            new_m, new_c = m + move[0], c + move[1]
            if is_valid_state(new_m, new_c):
                successors.append((new_m, new_c, 1))

    return successors

def missionaries_and_cannibals():
    start_state = (3, 3, 1)
    goal_state = (0, 0, 0)

    queue = deque([(start_state, [])])
    visited = set()

    while queue:
        state, path = queue.popleft()

        if state in visited:
            continue

        visited.add(state)

        if state == goal_state:
            return path + [state]

        for successor in generate_successors(state):
            queue.append((successor, path + [state]))
```

```
    return None

def print_solution(solution):
    if not solution:
        print("No solution found.")
        return
    print("Solution path:")
    for step in solution:
        m, c, b = step
        boat_position = "Left" if b == 1 else "Right"
        print(f"Missionaries: {m}, Cannibals: {c}, Boat: {boat_position}")

if __name__ == "__main__":
    solution = missionaries_and_cannibals()
    print_solution(solution)
```

Output:-

```
Solution path:
Missionaries: 3, Cannibals: 3, Boat: Left
Missionaries: 3, Cannibals: 1, Boat: Right
Missionaries: 3, Cannibals: 2, Boat: Left
Missionaries: 3, Cannibals: 0, Boat: Right
Missionaries: 3, Cannibals: 1, Boat: Left
Missionaries: 1, Cannibals: 1, Boat: Right
Missionaries: 2, Cannibals: 2, Boat: Left
Missionaries: 0, Cannibals: 2, Boat: Right
Missionaries: 0, Cannibals: 3, Boat: Left
Missionaries: 0, Cannibals: 1, Boat: Right
Missionaries: 1, Cannibals: 1, Boat: Left
Missionaries: 0, Cannibals: 0, Boat: Right
```

Q 12) Write a program to solve Missionaries and Cannibals problem.

```
import heapq

def print_board(board):
    for row in board:
        print(" ".join(str(cell) for cell in row))
    print()

def manhattan_distance(board, goal):
    distance = 0
    for i in range(3):
        for j in range(3):
            if board[i][j] == 0:
                continue
            x, y = divmod(goal.index(board[i][j]), 3)
            distance += abs(x - i) + abs(y - j)
    return distance

def is_solved(board, goal):
    return board == goal

def generate_successors(board):
    successors = []
    for i in range(3):
        for j in range(3):
            if board[i][j] == 0:
                x, y = i, j
                break

    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for move in moves:
        new_x, new_y = x + move[0], y + move[1]
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_board = [row[:] for row in board]
            new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
            successors.append(new_board)
    return successors

def a_star(start, goal):
    priority_queue = []
    heapq.heappush(priority_queue, (0, start, []))
    visited = set()

    while priority_queue:
        _, current_board, path = heapq.heappop(priority_queue)

        if is_solved(current_board, goal):
            return path + [current_board]
```

```
        board_tuple = tuple(tuple(row) for row in current_board)
        if board_tuple in visited:
            continue
        visited.add(board_tuple)

        for successor in generate_successors(current_board):
            new_path = path + [current_board]
            priority = len(new_path) + manhattan_distance(successor, sum(goal, []))
            heapq.heappush(priority_queue, (priority, successor, new_path))

    return None

def solve_puzzle(start_board, goal_board):
    print("Initial board:")
    print_board(start_board)

    print("Goal board:")
    print_board(goal_board)

    solution = a_star(start_board, goal_board)

    if solution:
        print("Solution found in {} moves!".format(len(solution) - 1))
        for step, board in enumerate(solution):
            print(f"Step {step}:")
            print_board(board)
    else:
        print("No solution found.")

if __name__ == "__main__":
    start_board = [
        [4, 5, 7],
        [8, 1, 2],
        [3, 6, 0]    ]

    goal_board = [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 0]
    ]

    solve_puzzle(start_board, goal_board)
```

Output:-

```
Initial board:
4 5 7
8 1 2
3 6 0

Goal board:
1 2 3
4 5 6
7 8 0

Solution found in 22 moves!
Step 0:
4 5 7
8 1 2
3 6 0

Step 1:
4 5 7
8 1 0
3 6 2

Step 2:
4 5 0
8 1 7
3 6 2

Step 3:
4 0 5
8 1 7
3 6 2

Step 4:
4 1 5
8 0 7
3 6 2

Step 5:
4 1 5
8 7 0
3 6 2

Step 6:
4 1 5
8 7 2
3 6 0

Step 7:
4 1 5
8 7 2
3 0 6

Step 8:
4 1 5
8 0 2
3 7 6

Step 9:
4 1 5
0 8 2
3 7 6

Step 10:
4 1 5
3 8 2
0 7 6

Step 11:
4 1 5
3 8 2
7 0 6

Step 12:
4 1 5
3 0 2
7 8 6

Step 13:
4 1 5
0 3 2
```

```
7 8 6

Step 14:
0 1 5
4 3 2
7 8 6

Step 15:
1 0 5
4 3 2
7 8 6

Step 16:
1 3 5
4 0 2
7 8 6

Step 17:
1 3 5
4 2 0
7 8 6

Step 18:
1 3 0
4 2 5
7 8 6

Step 19:
1 0 3
4 2 5
7 8 6

Step 20:
1 2 3
4 0 5
7 8 6

Step 21:
1 2 3
4 5 0
7 8 6

Step 22:
1 2 3
4 5 6
7 8 0
```

Q 13) Write a program to shuffle Deck of cards.

```
import random

suits = ['Hearts', 'Diamonds', 'Clubs', 'Spades']
ranks = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'Jack', 'Queen', 'King', 'Ace']

deck = [f'{rank} of {suit}' for suit in suits for rank in ranks]

def shuffle_deck(deck):
    random.shuffle(deck)
    return deck

def deal_cards(deck, num_cards):
    if num_cards <= len(deck):
        dealt_cards = deck[:num_cards]
        deck = deck[num_cards:]
        return dealt_cards, deck
    else:
        print("Not enough cards left in the deck!")
        return [], deck

def main():
    print("Original deck:")
    print(deck)
    print("\nShuffling deck...\n")

    shuffled_deck = shuffle_deck(deck[:])

    print("Shuffled deck:")
    print(shuffled_deck)

    num_cards_to_deal = 5
    dealt_cards, remaining_deck = deal_cards(shuffled_deck, num_cards_to_deal)

    print(f"\nDealt {num_cards_to_deal} cards:")
    print(dealt_cards)

    print(f"\nRemaining cards in the deck ({len(remaining_deck)} cards left):")
    print(remaining_deck)

if __name__ == "__main__":
    main()
```

Output:-

Original deck:

```
['2 of Hearts', '3 of Hearts', '4 of Hearts', '5 of Hearts', '6 of Hearts', '7 of Hearts', '8 of Hearts', '9 of Hearts', '10 of Hearts', 'Jack of Hearts', 'Queen of Hearts', 'King of Hearts', 'Ace of Hearts', '2 of Diamonds', '3 of Diamonds', '4 of Diamonds', '5 of Diamonds', '6 of Diamonds', '7 of Diamonds', '8 of Diamonds', '9 of Diamonds', '10 of Diamonds', 'Jack of Diamonds', 'Queen of Diamonds', 'King of Diamonds', 'Ace of Diamonds', '2 of Clubs', '3 of Clubs', '4 of Clubs', '5 of Clubs', '6 of Clubs', '7 of Clubs', '8 of Clubs', '9 of Clubs', '10 of Clubs', 'Jack of Clubs', 'Queen of Clubs', 'King of Clubs', 'Ace of Clubs', '2 of Spades', '3 of Spades', '4 of Spades', '5 of Spades', '6 of Spades', '7 of Spades', '8 of Spades', '9 of Spades', '10 of Spades', 'Jack of Spades', 'Queen of Spades', 'King of Spades', 'Ace of Spades']

Shuffling deck...

Shuffled deck:
['Ace of Clubs', '8 of Spades', '6 of Diamonds', 'Queen of Hearts', '5 of Spades', 'King of Hearts', '7 of Clubs', '6 of Clubs', 'Queen of Spades', 'Jack of Hearts', '7 of Diamonds', '8 of Diamonds', '9 of Clubs', 'Ace of Hearts', 'Queen of Clubs', '6 of Hearts', '5 of Hearts', '6 of Spades', '2 of Spades', 'Jack of Diamonds', '9 of Spades', '2 of Clubs', '10 of Hearts', '7 of Spades', '3 of Diamonds', '2 of Hearts', '10 of Diamonds', '3 of Spades', '5 of Diamonds', '5 of Clubs', '4 of Spades', '2 of Diamonds', 'King of Clubs', '8 of Hearts', '3 of Hearts', '9 of Hearts', '10 of Clubs', 'Ace of Diamonds', 'Ace of Spades', '4 of Hearts', 'King of Spades', '7 of Hearts', '4 of Clubs', '3 of Clubs', 'King of Diamonds', '9 of Diamonds', '10 of Spades', '4 of Diamonds', 'Jack of Spades', 'Queen of Diamonds', '8 of Clubs', 'Jack of Clubs']

Dealt 5 cards:
['Ace of Clubs', '8 of Spades', '6 of Diamonds', 'Queen of Hearts', '5 of Spades']

Remaining cards in the deck (47 cards left):
['King of Hearts', '7 of Clubs', '6 of Clubs', 'Queen of Spades', 'Jack of Hearts', '7 of Diamonds', '8 of Diamonds', '9 of Clubs', 'Ace of Hearts', 'Queen of Clubs', '6 of Hearts', '5 of Hearts', '6 of Spades', '2 of Spades', 'Jack of Diamonds', '9 of Spades', '2 of Clubs', '10 of Hearts', '7 of Spades', '3 of Diamonds', '2 of Hearts', '10 of Diamonds', '3 of Spades', '5 of Diamonds', '5 of Clubs', '4 of Spades', '2 of Diamonds', 'King of Clubs', '8 of Hearts', '3 of Hearts', '9 of Hearts', '10 of Clubs', 'Ace of Diamonds', 'Ace of Spades', '4 of Hearts', 'King of Spades', '7 of Hearts', '4 of Clubs', '3 of Clubs', 'King of Diamonds', '9 of Diamonds', '10 of Spades', '4 of Diamonds', 'Jack of Spades', 'Queen of Diamonds', '8 of Clubs', 'Jack of Clubs']
```

Q 14) Solve traveling salesman problem using artificial intelligence technique.

```
import random
import numpy as np

distance_matrix = np.array([[0, 20, 42, 35],
                            [20, 0, 30, 34],
                            [42, 30, 0, 12],
                            [35, 34, 12, 0]])

def calculate_distance(tour):
    total_distance = sum(distance_matrix[tour[i], tour[i + 1]] for i in range(len(tour) - 1))
    total_distance += distance_matrix[tour[-1], tour[0]]
    return total_distance

def create_random_tour(num_cities):
    tour = list(range(num_cities))
    random.shuffle(tour)
    return tour

def mutate(tour):
    a, b = random.sample(range(len(tour)), 2)
    tour[a], tour[b] = tour[b], tour[a]

def crossover(parent1, parent2):
    start, end = sorted(random.sample(range(len(parent1)), 2))
    child = [None] * len(parent1)
    child[start:end] = parent1[start:end]

    current_pos = end % len(parent1)
    for city in parent2:
        if city not in child:
            child[current_pos] = city
            current_pos = (current_pos + 1) % len(parent1)

    return child

def genetic_algorithm(num_cities, population_size, generations):
    population = [create_random_tour(num_cities) for _ in range(population_size)]

    for _ in range(generations):
        population = sorted(population, key=calculate_distance)
        next_generation = population[:2]
        while len(next_generation) < population_size:
            parent1, parent2 = random.choices(population[:population_size // 2], k=2)
            child = crossover(parent1, parent2)
            if random.random() < 0.1:
                mutate(child)
            next_generation.append(child)
        population = next_generation
```

```
    best_tour = min(population, key=calculate_distance)
    return best_tour, calculate_distance(best_tour)

best_tour, best_distance = genetic_algorithm(num_cities=4, population_size=100,
generations=1000)
print(f"Best Tour: {best_tour}, Distance: {best_distance}")
```

Output:-

```
Best Tour: [0, 1, 2, 3], Distance: 97
```

Q 15) Solve the block of World problem.

```
from collections import deque

class State:
    def __init__(self, stacks):
        self.stacks = stacks

    def __str__(self):
        return str(self.stacks)

    def is_goal(self, goal):
        return self.stacks == goal.stacks

    def get_possible_moves(self):
        moves = []
        for i, stack in enumerate(self.stacks):
            if stack:
                top_block = stack[-1]
                for j, other_stack in enumerate(self.stacks):
                    if i != j:
                        new_stacks = [s[:] for s in self.stacks]
                        new_stacks[i] = new_stacks[i][:-1]
                        new_stacks[j].append(top_block)
                        moves.append((State(new_stacks), f'Move {top_block} from stack {i} to stack {j}'))
        return moves

def bfs(initial_state, goal_state):
    queue = deque([(initial_state, [])])
    visited
```

Output:-

```
Solution found: [['C', 'A'], [], ['B', 'D']]
Actions taken:
Move A from stack 0 to stack 1
Move D from stack 2 to stack 1
Move B from stack 0 to stack 2
Move D from stack 1 to stack 2
Move A from stack 1 to stack 2
Move C from stack 1 to stack 0
Move A from stack 2 to stack 0
```

Q 16) Solve constraint satisfaction problem.

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

    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    for i, j in zip(range(row, n), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True

def solve_n_queens(board, col, n):
    if col >= n:
        return True

    for i in range(n):
        if is_safe(board, i, col, n):
            board[i][col] = 1

            if solve_n_queens(board, col + 1, n):
                return True

            board[i][col] = 0

    return False

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

def solve_n_queens_problem(n):
    board = [[0] * n for _ in range(n)]

    if solve_n_queens(board, 0, n):
        print("Solution:")
        print_board(board, n)
    else:
        print("No solution exists.")

solve_n_queens_problem(6)
```

Output:-

```
. . . Q . .
Q . . . . .
. . . . Q .
. Q . . . .
. . . . . Q
. . Q . . .
```

Q 17) Derive the expressions based on Associative law.

```
def associative_addition(a, b, c):
    left_associative = (a + b) + c
    right_associative = a + (b + c)
    print(f"Left associative (a + b) + c: {left_associative}")
    print(f"Right associative a + (b + c): {right_associative}")
    return left_associative == right_associative

def associative_multiplication(a, b, c):
    left_associative = (a * b) * c
    right_associative = a * (b * c)
    print(f"Left associative (a * b) * c: {left_associative}")
    print(f"Right associative a * (b * c): {right_associative}")
    return left_associative == right_associative

a, b, c = 5, 6, 7

print("Addition is associative:", associative_addition(a, b, c))
print("Multiplication is associative:", associative_multiplication(a, b, c))
```

Output:-

```
Left associative (a + b) + c: 18
Right associative a + (b + c): 18
Addition is associative: True
Left associative (a * b) * c: 210
Right associative a * (b * c): 210
Multiplication is associative: True
```

Q 18) Derive the expressions based on Distributive law.

```
def distributive_multiplication_addition(a, b, c):
    left_side = a * (b + c)
    right_side = (a * b) + (a * c)

    print(f"Left side a * (b + c): {left_side}")
    print(f"Right side (a * b) + (a * c): {right_side}")
    return left_side == right_side

def distributive_multiplication_subtraction(a, b, c):
    left_side = a * (b - c)
    right_side = (a * b) - (a * c)

    print(f"Left side a * (b - c): {left_side}")
    print(f"Right side (a * b) - (a * c): {right_side}")
    return left_side == right_side

a, b, c = 8, 6, 3

print("Multiplication over Addition is distributive:", distributive_multiplication_addition(a, b, c))
print("Multiplication over Subtraction is distributive:", distributive_multiplication_subtraction(a, b,
c))
```

Output:-

```
Left side a * (b + c): 72
Right side (a * b) + (a * c): 72
Multiplication over Addition is distributive: True
Left side a * (b - c): 24
Right side (a * b) - (a * c): 24
Multiplication over Subtraction is distributive: True
```

Q 19) Write a program to derive the predicate.

```
class Predicate:
    def __init__(self, subject, predicate):
        self.subject = subject
        self.predicate = predicate

    def __str__(self):
        return f"{self.subject} is {self.predicate}"

def derive_predicate(p1, p2):
    if p1.predicate == p2.subject:
        return Predicate(p1.subject, p2.predicate)
    return None

p1 = Predicate("Sachin", "batsman")
p2 = Predicate("batsman", "cricketer")

print(f"Premise 1: {p1}")
print(f"Premise 2: {p2}")

derived_predicate = derive_predicate(p1, p2)

if derived_predicate:
    print(f"Conclusion: {derived_predicate}")
else:
    print("No valid conclusion can be derived.")
```

Output:-

```
Premise 1: Sachin is batsman
Premise 2: batsman is cricketer
Conclusion: Sachin is cricketer
```