LAB TERMINAL EXAM

Submitted by:	***SYEDA AIMA ABBAS***
REG # NO	***FA24-BSE-010***
INSTRUCTOR	***SIR IMRAN***
COURSE CODE	***CSC462***
COURSE TITLE	***ARTIFICIAL INTELLIGENCE***

DEPARTMENT OF COMPUTER SCIENCE
COMSATS UNIVERSITY ISLAMABAD,
ISLAMABAD CAMPUS

[CLO6]: Implement various searching technique, CSP and knowledge-based system to solve a problem.

Lab Project

Question 1: **[4 x 8 = 32 Marks]**

The Traveling Salesperson Problem (TSP) is a classic NP-hard optimization problem in which a salesperson must visit a set of cities exactly once, return to the starting point, and minimize the total travel distance. For large numbers of cities, brute-force or exhaustive search approaches become computationally infeasible. Genetic Algorithms (GAs) provide an effective heuristic method to obtain near-optimal solutions within reasonable time. To implement a GA-based solution for TSP, the following components are required:

1. **Population Initialization:** Generate an initial random population of tours.
2. **Chromosome Representation:** Represent a tour (sequence of cities) as a permutation of city names.
3. **Fitness Function:** Calculate the fitness of each tour (chromosome) based on its total distance.
4. **Selection:** Implement a Tournament selection mechanism to choose parents for reproduction. Preserve a certain number of the best individuals from one generation to the next.
5. **Crossover:** Implement a TSP-specific Partially Mapped Crossover (PMX) crossover operator to create offspring.
6. **Mutation:** Implement a Swap mutation operator to introduce diversity into the population.
7. **Termination Criteria:** Stop the GA after a fixed number of generations or if the fitness improvement stops.
8. **Output:** Display the best-found tour and its total distance.

SOLUTION

```python
import random
import math

# Step 1: City coordinates (x,y positions)
cities = [(0,0), (1,5), (5,2), (6,6), (8,3)]

# Step 2: Calculate distance between two cities
def distance(city1, city2):
    x1, y1 = city1
    x2, y2 = city2
    return math.sqrt((x1-x2)**2 + (y1-y2)**2)

# Step 3: One tour = list of city order [0,2,1,3,4]
class Tour:
    def __init__(self, order):
        self.order = order[:]  # copy the list
        self.distance = 0      # total distance will be calculated later

    def calc_distance(self):
        total = 0
        n = len(self.order)
```

```
        for i in range(n):
            # Distance between city i and next city
            total += distance(cities[self.order[i]],
cities[self.order[(i+1)%n]])
        self.distance = total
        return total

# Step 4: Create random population (50 random tours)
def create_population(size):
    population = []
    for _ in range(size):
        order = list(range(len(cities)))  # [0,1,2,3,4]
        random.shuffle(order)             # shuffle to random order
        population.append(Tour(order))
    return population

# Step 5: Tournament selection (pick best from 3 random tours)
def select_parent(population):
    tournament = random.sample(population, 3)  # pick 3 random tours
    return min(tournament, key=lambda t: t.distance)  # return shortest

# Step 6: Simple crossover (mix two parents)
def crossover(parent1, parent2):
    start = random.randint(0, len(cities)-3)
    end = start + 2

    # Child gets segment from parent1
    child_order = [-1] * len(cities)
    for i in range(start, end+1):
        child_order[i] = parent1.order[i]

    # Fill rest from parent2 (avoid duplicates)
    for i in range(len(cities)):
        if child_order[i] == -1 and parent2.order[i] not in child_order:
            child_order[i] = parent2.order[i]

    # Fill any remaining spots
    remaining = [city for city in parent2.order if city not in child_order]
    j = 0
    for i in range(len(cities)):
        if child_order[i] == -1:
            child_order[i] = remaining[j]
            j += 1

    return Tour(child_order)

# Step 7: Swap mutation (swap 2 cities with 10% chance)
def mutate(tour, rate=0.1):
```

```
    if random.random() < rate:
        i, j = random.sample(range(len(tour.order)), 2)
        tour.order[i], tour.order[j] = tour.order[j], tour.order[i]

# Step 8: Main Genetic Algorithm
def genetic_algorithm():
    population = create_population(50)  # 50 random tours
    for tour in population:
        tour.calc_distance()              # calculate distance for all

    best_tour = min(population, key=lambda t: t.distance)

    for generation in range(100):         # 100 generations
        new_population = []

        # Keep best 5 tours (elitism)
        new_population.extend(sorted(population, key=lambda t:
t.distance)[:5])

        # Create 45 new tours
        while len(new_population) < 50:
            parent1 = select_parent(population)
            parent2 = select_parent(population)
            child = crossover(parent1, parent2)
            mutate(child)
            child.calc_distance()
            new_population.append(child)

        population = new_population
        current_best = min(population, key=lambda t: t.distance)

        if current_best.distance < best_tour.distance:
            best_tour = current_best

        if generation % 20 == 0:
            print(f"Gen {generation}: Best distance =
{best_tour.distance:.2f}")

    return best_tour

# Run it!
if __name__ == "__main__":
    best = genetic_algorithm()
    print("\n*** BEST TOUR FOUND ***")
    print("Order:", best.order)
    print("Cities:", [cities[i] for i in best.order])
    print("Total distance:", best.distance)
```

OUTPUT:

```
eRunnerFile.py"
Gen 0: Best distance = 22.35
Gen 20: Best distance = 22.35
Gen 40: Best distance = 22.35
Gen 60: Best distance = 22.35
Gen 80: Best distance = 22.35

*** BEST TOUR FOUND ***
Order: [4, 2, 0, 1, 3]
Cities: [(8, 3), (5, 2), (0, 0), (1, 5), (6, 6)]
Total distance: 22.35103276995244
PS C:\Users\AIMAF\Desktop\Comsats\3rd Sem\AI\project>
```

Question2: **[2 x 6 = 12 Marks]**

Consider an agent A located in a Wumpus World environment as shown in Figure 1. The environment consists of a grid where some cells may contain pits or a Wumpus. The agent perceives the environment through its sensors and must reason about the locations of pits and the Wumpus.

Use Prolog to build a knowledge base (KB) for a Wumpus World environment and implement logical inference to determine the presence or absence of pits and the Wumpus in specific cells. The KB should be sufficient to prove the following statements:

1. There is no pit in cells (2,2) and (1,3), i.e., KB $\models \neg Pit_{2,2} \wedge \neg Pit_{1,3}$ **[6]**
2. There is a Wumpus in cell (1,3), i.e., KB $\models W_{1,3}$ **[6]**

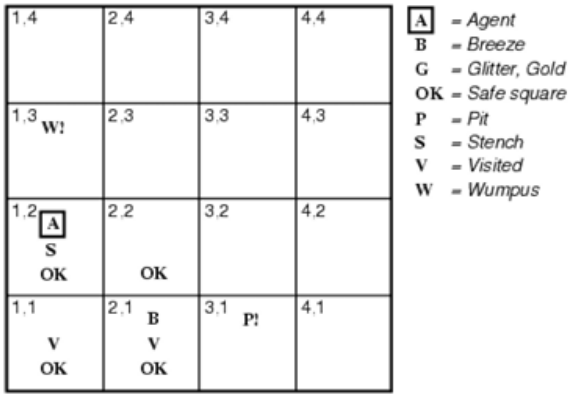

Figure 1. Wumpus World Environment

SOLUTION

```
% ========================================
% WUMPUS WORLD KNOWLEDGE BASE (KB)
% ========================================
:- discontiguous pit/2.
% ----------------------------------------
% WORLD SIZE (4x4 GRID)
% ----------------------------------------
cell(X,Y) :-
    between(1,4,X),
    between(1,4,Y).

% ----------------------------------------
% OBSERVATIONS (FROM ENVIRONMENT)
% ----------------------------------------
breeze(2,1).
stench(1,2).

% ----------------------------------------
% KNOWN FACTS
% ----------------------------------------
pit(3,1).         % Given pit
safe(1,1).
safe(1,2).
safe(2,1).
safe(2,2).

% ----------------------------------------
% ADJACENCY (WITH BOUNDARY CHECK)
% ----------------------------------------
adjacent(X,Y,X1,Y) :-
    X1 is X+1,
    cell(X1,Y).

adjacent(X,Y,X1,Y) :-
    X1 is X-1,
    cell(X1,Y).

adjacent(X,Y,X,Y1) :-
    Y1 is Y+1,
    cell(X,Y1).

adjacent(X,Y,X,Y1) :-
    Y1 is Y-1,
    cell(X,Y1).

% ----------------------------------------
% PIT INFERENCE RULES
```

```
% ----------------------------------------

% A pit may exist in a cell if there is a breeze nearby
possible_pit(X,Y) :-
    breeze(A,B),
    adjacent(A,B,X,Y),
    cell(X,Y).

% No pit in safe cells
no_pit(X,Y) :-
    safe(X,Y).

% A pit exists only if it is possible and not proven absent
pit(X,Y) :-
    possible_pit(X,Y),
    \+ no_pit(X,Y).

% ----------------------------------------
% WUMPUS INFERENCE RULES
% ----------------------------------------

% A Wumpus may exist in a cell if there is a stench nearby
possible_wumpus(X,Y) :-
    stench(A,B),
    adjacent(A,B,X,Y),
    cell(X,Y).

% No Wumpus in safe cells
no_wumpus(X,Y) :-
    safe(X,Y).

% Wumpus exists in a cell if it is possible and not safe
wumpus(X,Y) :-
    possible_wumpus(X,Y),
    \+ no_wumpus(X,Y).

% ----------------------------------------
% QUERY HELPERS (FOR REPORT)
% ----------------------------------------

% Prove no pit in specific cells
no_pit_22 :-
    \+ pit(2,2).

no_pit_13 :-
    \+ pit(1,3).

% Prove Wumpus location
```

```
prove_wumpus :-
    wumpus(1,3).
```

OUTPUT: