Name: Nikhil Premachandra rao **Student ID: 02105149**

CIS 561 - Artificial Intelligence Project 1

Project 1. Solving the Traveling Salesman Problem Using Genetic Algorithm

The traveling salesman problem (TSP) is a popular AI problem that asks for the most efficient trajectory possible given a set of points and distances that must all be visited. The problem can be applied to the most efficient calculation in computer science.

Task 1: Study the paper attached to Project 1. The paper is “Traveling Salesman Problem: An Overview of Applications, Formulations, and Solution Approaches”.

1. What is the traveling salesman problem?

The TSP is a combinatorial optimization problem where the objective is to find the shortest possible route that visits a set of cities exactly once and returns to the starting point. Mathematically, given a set of cities and distances between each pair, the problem seeks the minimal-length Hamiltonian cycle. The complexity arises from the exponential number of possible routes as the number of cities increases, making the problem NP-hard. This means that there is no known polynomial-time solution, and solving the problem becomes computationally expensive as the number of cities grows. Due to its real-world relevance, it has been studied extensively in fields like operations research, logistics, and computer science.

2. What are the applications of the traveling salesman problem?

Drilling of Printed Circuit Boards (PCBs): Optimizes the sequence of drilling holes, minimizing the movement of the drilling head.
Overhauling Gas Turbine Engines: Arranges nozzle-guide vanes to optimize gas flow and reduce vibration in turbines.
X-Ray Crystallography: Minimizes positioning time in experiments by optimizing the sequence in which crystal positions are measured.
Computer Wiring: Finds the shortest Hamiltonian path to optimize connections between components on a circuit board.
Order Picking in Warehouses: Optimizes the collection route for items in warehouses, minimizing travel time.
Vehicle Routing: Minimizes the distance for delivering goods or collecting mail from various locations using vehicles.
Mission Planning for Military or UAVs: Determines optimal paths for military missions or unmanned aerial vehicles (UAVs) to complete multiple tasks.

3. According to Approximate approaches (section 5) in the paper, make a table to briefly describe different approaches. An example looks like below. Only two approaches are listed here. You need to list all approximate approaches in this paper. Each approach's description is no more than 3 sentences.

Approach	Description
Closest Neighbor	Always visits the nearest unvisited city. Typically keeps the tour within 25% of optimality.
Greedy Heuristic	Constructs a tour by selecting the shortest available edge while avoiding cycles. Typically within 15-20% of the optimal solution.
Insertion Heuristic	Starts with a small subset of cities and incrementally adds the nearest unvisited city, ensuring minimal insertion cost.
Christofides Heuristic	Builds a minimum spanning tree, matches odd-degree nodes, and creates an Eulerian circuit. Provides a solution within 10% of optimality.
2-opt and 3-opt	Improves an existing tour by swapping two or three edges, reducing the path length. Often reduces the tour to 3-5% above optimal.
Lin-Kernighan Heuristic	A variable k-way exchange heuristic that adapts k at each iteration. Known to get within 2% of the optimal solution.
Tabu Search	Neighborhood search heuristic that avoids local optima by maintaining a tabu list to prevent revisiting bad solutions.
Simulated Annealing	A probabilistic approach that accepts worse solutions early on to escape local minima and gradually converges to an optimal or near-optimal solution.
Genetic Algorithm	Uses evolutionary principles such as crossover and mutation to evolve a population of solutions, improving their fitness over iterations.
Ant Colony Optimization	Mimics ant behavior by using pheromone trails to guide searches toward shorter paths based on previous iterations.

Task 2: Give 10 cities located within 1,000 miles (left to right) by 1,000 miles (bottom to top) region and calculate the shortest traveling path from the traveling salesman problem. The 10 cities are A, B, C, D, E, F, G, H, I, and J. Locations of the 10 cities are **The traveling path between any two cities is considered a straight line.**

Code:

```
import numpy as np
import random

# Coordinates of the cities
cities = {
   'A': (100, 300), 'B': (200, 130), 'C': (300, 500), 'D': (500, 390),
   'E': (700, 300), 'F': (900, 600), 'G': (800, 950), 'H': (600, 560),
   'I': (350, 550), 'J': (270, 350)
}

# Calculate the Euclidean distance between two cities
def euclidean_distance(city1, city2):
   x1, y1 = cities[city1]
   x2, y2 = cities[city2]
   return np.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)

# Fitness function: total distance of the route
def fitness(route):
   distance = 0
   for i in range(len(route) - 1):
       distance += euclidean_distance(route[i], route[i + 1])
   # Return to starting city
   distance += euclidean_distance(route[-1], route[0])
   return distance

# Initialize a random population
def create_population(pop_size, cities):
   population = []
   for _ in range(pop_size):
       route = list(cities.keys())[1:]  # Exclude city A
       random.shuffle(route)
       route = ['A'] + route + ['A']  # Start and end at A
       population.append(route)
   return population

# Selection: Tournament selection
def selection(population, fitnesses, k=3):
   selected = random.sample(list(zip(population, fitnesses)), k)
   selected.sort(key=lambda x: x[1])  # Sort by fitness (distance)
   return selected[0][0]  # Return the best route
```

```python
# Crossover: Order Crossover (OX)
def crossover(parent1, parent2):
   size = len(parent1)

   # Ensure size is sufficient
   if size <= 2:
       raise ValueError("Parent size must be greater than 2 for crossover.")

   # Sample start and end indices
   start, end = sorted(random.sample(range(1, size - 1), 2))  # Exclude 'A'

   # Create child with 'None' placeholders
   child = [None] * size
   child[0], child[-1] = 'A', 'A'  # Keep 'A' at both ends
   child[start:end+1] = parent1[start:end+1]  # Inherit a slice from parent1

   # Fill the remaining cities from parent2 in the order they appear
   parent2_cities = [city for city in parent2 if city not in child]

   # Replace 'None' values in the child with cities from parent2
   j = (end + 1) % size  # Start filling from the next index after 'end'
   for city in parent2_cities:
       while child[j] is not None:
           j = (j + 1) % size  # Move to the next index if already filled
       child[j] = city  # Fill in missing cities
       j = (j + 1) % size  # Move to the next index

   return child

# Mutation: Swap Mutation
def mutate(route, mutation_rate=0.1):
   if random.random() < mutation_rate:
       i, j = random.sample(range(1, len(route) - 1), 2)  # Exclude 'A'
       route[i], route[j] = route[j], route[i]
   return route

# Genetic Algorithm
def genetic_algorithm(cities, pop_size=100, generations=500, mutation_rate=0.1):
   population = create_population(pop_size, cities)

   # Print the initial population
   print("\nInitial population:")
   for i, route in enumerate(population):
       print(f"Route {i+1}: {route}")

   for gen in range(generations):
       fitnesses = [fitness(route) for route in population]
```

```
        # Print fitness values for the generation
        print(f"\nGeneration {gen+1} fitness values:")
        for i, f in enumerate(fitnesses):
            print(f"Route {i+1}: Fitness = {f:.2f}")

        new_population = []
        for _ in range(pop_size):
            parent1 = selection(population, fitnesses)
            parent2 = selection(population, fitnesses)

            # Print selected parents
            print(f"Selected Parent 1: {parent1}")
            print(f"Selected Parent 2: {parent2}")

            child = crossover(parent1, parent2)
            print(f"Child after crossover: {child}")

            child = mutate(child, mutation_rate)
            print(f"Child after mutation: {child}")

            # Debugging: Ensure all cities are present in the child
            if None in child:
                print(f"Error in child: {child}")
            else:
                new_population.append(child)

        population = new_population

    # Print final population
    print("\nFinal population:")
    for i, route in enumerate(population):
        print(f"Route {i+1}: {route}")

    best_route = min(population, key=fitness)

    # Print best route and best fitness (distance)
    print(f"\nBest route found: {best_route}")
    print(f"Best distance: {fitness(best_route):.2f}")

    return best_route, fitness(best_route)

# Run the Genetic Algorithm
best_route, best_distance = genetic_algorithm(cities)
print("Best route:", best_route)
print("Best distance:", best_distance)
```

Output

```
Best route found: ['A', 'B', 'D', 'E', 'F', 'G', 'H', 'I', 'C', 'J', 'A']
Best distance: 2627.47
Best route: ['A', 'B', 'D', 'E', 'F', 'G', 'H', 'I', 'C', 'J', 'A']
Best distance: 2627.471013137072
```

Shortest distance value: 2627.47 miles
Sequence order of 10 cities: A -> B -> D -> E -> F -> G -> H -> I -> C -> J -> A