logistics

1. Vehicle Routing Problem (VRP) Models:

The Vehicle Routing Problem (VRP) is a well-studied combinatorial optimization problem that involves determining the optimal set of routes for a fleet of vehicles to visit a given set of locations (customers) and return to a central depot while satisfying various constraints.

Details:

Objective:

Minimize the total time required for trucks to refill fuel from tanks in a depot area.

- **Constraints:**
  1. Each tank can be serviced by only one truck at a time.
  2. Each truck has a fixed capacity and cannot exceed it while refueling.
  3. Trucks must adhere to predefined time windows for refueling at each tank.

  4.The total time includes both travel time between tanks and refueling time at each tank.

SOLUTIONS:

1. Exact Algorithms:

Exact algorithms aim to find the optimal solution to the VRP problem by exhaustively searching through all possible solutions. While they guarantee optimality, they can be computationally intensive and may not be feasible for large problem instances.

a. Branch and Bound:

is a systematic enumeration algorithm that explores the solution space by recursively partitioning it into smaller subsets (branches) and bounding the search to only explore promising regions.

it start with initial solution and iteratively refines it by branching into subproblems , pruning branches that are guaranteed to lead suboptimal

solutions.

b. Integer Linear Programming (ILP) Formulations:

- **Description:** ILP formulations represent the VRP as a mathematical optimization problem with linear constraints and an objective function, which can be solved using optimization solvers.
- **Process:** Formulate the VRP as a set of binary decision variables representing the presence of edges between nodes (customers) in the routes. Define constraints to ensure that each customer is visited exactly once, vehicle capacity constraints are satisfied, and routes form connected paths.

2. Heuristic Algorithms:

they aim to find good-quality solution in a reasonable amount of time .

- **Algorithms:** There are many algorithms to solve VRP, including:
  - **Exact algorithms:** These guarantee the optimal solution but are computationally intensive, such as branch and bound or branch and cut.
  - **Heuristic algorithms:** These provide good-quality solutions in a reasonable amount of time but do not guarantee optimality. Examples include:
    - **Genetic algorithms:** Inspired by the process of natural selection, genetic algorithms evolve a population of candidate solutions over multiple generations.
    - **Simulated annealing:** Mimicking the annealing process in metallurgy, simulated annealing iteratively explores the solution space, accepting worse solutions with a decreasing probability.
    - **Ant colony optimization:** Inspired by the foraging behavior of ants, this metaheuristic algorithm iteratively constructs solutions using artificial ants that deposit pheromone trails.

Choice of Genetic Algorithm:

- Genetic algorithms are chosen for the vehicle refueling problem because they are well-suited for problems with a large solution space, such as optimizing the sequence of trucks visiting tanks.

- GA's ability to maintain diversity in the population, explore different regions of the solution space, and exploit promising solutions makes it effective for finding near-optimal solutions in complex optimization problems like vehicle routing.

Genetic Algorithm:

- **Operation:** Solutions are represented as chromosomes, which undergo genetic operations such as crossover and mutation to produce new offspring. The fitness of each solution determines its likelihood of being selected for reproduction and survival in subsequent generations.
- **Exploration vs. Exploitation:** GA emphasizes exploration by maintaining a diverse population and exploring different regions of the solution space. It also exploits promising regions by favoring solutions with higher fitness.
- **Advantages:** Suitable for problems with a large solution space, non-linear relationships between variables, and multiple optima.

In the context of the trucks refueling problem, GA can effectively explore different combinations of trucks visiting tanks, finding near-optimal solutions that minimize the total time.

Code :

```
import random

# Define the distance matrix (replace with actual data)
distance_matrix = {
    ('tank1', 'truck1'): {'time': 10},
    ('tank2', 'truck2'): {'time': 20},
    ('tank3', 'truck3'): {'time': 15},
    # Add more tank-truck pairs as needed
}

# Generate initial population
def generate_initial_population(tanks, trucks, population_siz
    population = []
    for _ in range(population_size):
```

```
        solution = []
        for tank in tanks:
            truck = random.choice(trucks)
            solution.append((tank, truck))
        population.append(solution)
    return population

# Fitness function
def fitness(solution):
    total_time = 0
    visited_tanks = set()
    for tank, truck in solution:
        if (tank, truck) in distance_matrix and tank not in v
            travel_time = distance_matrix[(tank, truck)]['tim
            total_time += travel_time  # Travel time
            total_time += 20  # Time for refueling (assuming
            visited_tanks.add(tank)  # Mark tank as visited
    return total_time

# Genetic Algorithm
def genetic_algorithm(tanks, trucks, population_size, generat
    population = generate_initial_population(tanks, trucks, p
    best_solution = min(population, key=lambda x: fitness(x))
    for _ in range(generations):
        new_population = []
        for _ in range(population_size):
            parent1, parent2 = random.choices(population, k=2
            crossover_point = random.randint(1, len(tanks) -
            child = parent1[:crossover_point] + parent2[cross
            new_population.append(child)
        population = new_population
        best_solution = min(population, key=lambda x: fitness
    return best_solution

# Example usage
tanks = ['tank1', 'tank2', 'tank3']  # Replace with actual ta
trucks = ['truck1', 'truck2', 'truck3']  # Replace with actua
POPULATION_SIZE = 10
```

```
GENERATIONS = 100

best_solution = genetic_algorithm(tanks, trucks, POPULATION_S

# Print the best solution and its fitness value
print("Best Solution:", best_solution)
print("Total Time Taken:", fitness(best_solution))
```

code explanation :

1. `distance_matrix = {...}` :

Defines a distance matrix that represents the time required for trucks to travel between tanks in the depot area. It is a dictionary where each key is a tuple `(tank, truck)` representing a tank-truck pair, and the value is another dictionary containing the time required for travel (`'time'`).

2. `def generate_initial_population(tanks, trucks, population_size):` :

Defines a function `generate_initial_population` that takes three arguments: `tanks` (list of tank names), `trucks` (list of truck names), and `population_size` (desired size of the initial population).

3. `population = []` :

Initializes an empty list to store the population of solutions, `for _ in range(population_size):` : Iterates `population_size` times to generate multiple initial solutions for the population.

`solution = []` : Initializes an empty list to represent a solution (sequence of tank-truck pairs).

`for tank in tanks:` : Iterates over each tank in the list of tanks.

`truck = random.choice(trucks)` : Randomly selects a truck from the list of trucks using the `random.choice()` function.

`solution.append((tank, truck))`: Appends a tuple `(tank, truck)` representing the assignment of the current tank to the randomly selected truck to the solution list.

`population.append(solution)`: Appends the generated solution to the population list.

`return population`: Returns the generated initial population of solutions.

4. `def fitness(solution):`:

Defines a function `fitness` that calculates the fitness (total time) of a given solution.

`total_time = 0`: Initializes a variable `total_time` to store the total time required for the solution.

`visited_tanks = set()`: Initializes an empty set `visited_tanks` to keep track of visited tanks to ensure each tank is visited only once.

`for tank, truck in solution:`: Iterates over each tank-truck pair in the given solution.

`if (tank, truck) in distance_matrix and tank not in visited_tanks:`: Checks if the tank-truck pair exists in the distance matrix and if the tank has not been visited before.

`travel_time = distance_matrix[(tank, truck)]['time']`: Retrieves the travel time from the distance matrix for the current tank-truck pair.

`total_time += travel_time`: Adds the travel time to the total time.

`total_time += 20`: Adds a fixed time of 20 minutes for refueling (assuming 20 minutes per truck).

`visited_tanks.add(tank)`: Adds the current tank to the set of visited tanks to mark it as visited.

`return total_time`: Returns the total time required for the solution.

5. `def genetic_algorithm(tanks, trucks, population_size, generations):`:

Defines a function `genetic_algorithm` that takes four arguments: `tanks`, `trucks`, `population_size`, and `generations`.

`population = generate_initial_population(tanks, trucks, population_size)`: Generates the initial population of solutions using the `generate_initial_population` function.

`best_solution = min(population, key=lambda x: fitness(x))`: Finds the solution with the minimum fitness (total time) from the initial population using the `min()` function and the `fitness` function as the key for comparison.

`for _ in range(generations):`: Iterates over the specified number of generations.

`new_population = []`: Initializes an empty list to store the new population of solutions for the next generation.

`for _ in range(population_size):`: Iterates over the population size to generate new solutions for the next generation.

`parent1, parent2 = random.choices(population, k=2)`: Randomly selects two parent solutions (with replacement) from the current population using the `random.choices()` function.

`crossover_point = random.randint(1, len(tanks) - 1)`: Randomly selects a crossover point between 1 and the length of the tanks list - 1.

`child = parent1[:crossover_point] + parent2[crossover_point:]`: Creates a child solution by combining parts of the two parent solutions using crossover.

`new_population.append(child)`: Appends the newly created child solution to the new population list.

`population = new_population`: Updates the current population with the new population for the next generation.

`best_solution = min(population, key=lambda x: fitness(x))`: Finds the solution with the minimum fitness (total time) from the new population for the current generation.

`return best_solution`: Returns the best solution found after the specified number of generations.