Exercise # 8

1. **Consider succeeding travelling salesman problem**

A salesman would like to begin at city 1, visit each of the other cities once and only once, and then return to city 1. In what order should he visit the cities so that the total distance he travels is minimized?

CITY	A	B	C	D	E	F
A	-	19	14	11	23	24
B	24	-	12	30	30	19
C	40	42	-	20	36	15
D	20	35	37	-	45	33
E	15	26	18	25	-	30
F	22	17	14	30	28	-

2. **Write the program code in python that implements GENETIC ALGORITHM to solve the problem.**

The Python code below implements the genetic algorithm used in the travelling salesman problem.

```python
import random

# Define the distance matrix between cities
distance_matrix = [
    [0, 19, 14, 11, 23, 24],
    [24, 0, 12, 30, 30, 19],
    [40, 42, 0, 20, 36, 15],
    [20, 35, 37, 0, 45, 33],
    [15, 26, 18, 25, 0, 30],
    [22, 17, 14, 30, 28, 0],
]

# Number of cities
num_cities = len(distance_matrix)
```

```
# Population size
population_size = 50

# Number of generations
num_generations = 100

# Crossover rate
crossover_rate = 0.8

# Mutation rate
mutation_rate = 0.02

def calculate_fitness(route):
    """Calculate the fitness of a route, which is inversely proportional
to the total distance."""
    total_distance = 0
    for i in range(num_cities - 1):
        total_distance += distance_matrix[route[i]][route[i + 1]]
    total_distance += distance_matrix[route[-1]][route[0]]  # Return to
the starting city
    return 1 / total_distance  # Fitness is inversely proportional to
total distance

def initialize_population():
    """Initialize the population with random routes."""
    population = []
    for _ in range(population_size):
        route = list(range(num_cities))
        random.shuffle(route)
        population.append(route)
    return population

def selection(population):
    """Perform selection based on fitness, keeping the top half of the
population."""
    return sorted(population, key=calculate_fitness,
reverse=True)[:int(population_size * 0.5)]

def crossover(parent1, parent2):
    """Perform crossover between two parents to create a child."""
    crossover_point = random.randint(0, num_cities - 1)
     child = parent1[:crossover_point] + [city for city in parent2 if city not in parent1[:crossover_point]]
    return child

def mutate(route):
    """Perform mutation on a route with a certain probability."""
    if random.random() < mutation_rate:
        index1, index2 = random.sample(range(num_cities), 2)
        route[index1], route[index2] = route[index2], route[index1]
    return route

def genetic_algorithm():
    """Run the genetic algorithm to find the best route."""
    population = initialize_population()

    for generation in range(num_generations):
        population = selection(population)

        for _ in range(int(population_size * (1 - crossover_rate))):
            parent1, parent2 = random.sample(population, 2)
            child = crossover(parent1, parent2)
            child = mutate(child)
            population.append(child)

    best_route = max(population, key=calculate_fitness)
    return best_route

# Run the genetic algorithm
best_route = genetic_algorithm()

# Print the results
print("Best route:", best_route)
print("Total distance:", 1 / calculate_fitness(best_route))
```

3. **Give a detailed documentation of your python program. Explain how you do each step in the algorithm.**

The detailed documentation will be discussed below

```
import random
```

This line imports the random module for generating random numbers.

```
# Define the distance matrix between cities
distance_matrix = [
    [0, 19, 14, 11, 23, 24],
    [24, 0, 12, 30, 30, 19],
    [40, 42, 0, 20, 36, 15],
    [20, 35, 37, 0, 45, 33],
    [15, 26, 18, 25, 0, 30],
    [22, 17, 14, 30, 28, 0],
]
```

This matrix represents the distance between the cities.

```
# Number of cities
num_cities = len(distance_matrix)

# Population size
population_size = 50

# Number of generations
num_generations = 100

# Crossover rate
crossover_rate = 0.8

# Mutation rate
mutation_rate = 0.02
```

The parameters that are set here are:
num_cities: Number of cities in the TSP.
population_size: Number of individuals (routes) in each generation.
num_generations: Number of generations in the genetic algorithm.
crossover_rate: Probability of crossover occurring between two routes.
mutation_rate: Probability of mutation occurring in an individual route.

```
def calculate_fitness(route):
     """Calculate the fitness of a route, which is inversely proportional
to the total distance."""
    total_distance = 0
    for i in range(num_cities - 1):
        total_distance += distance_matrix[route[i]][route[i + 1]]
     total_distance += distance_matrix[route[-1]][route[0]]   # Return to
the starting city
     return 1 / total_distance   # Fitness is inversely proportional to
total distance
```

This function computes the fitness of a route, where fitness is defined as the inverse of the total distance traveled. The shorter the distance, the higher the fitness.

```
def initialize_population():
    """Initialize the population with random routes."""
    population = []
    for _ in range(population_size):
        route = list(range(num_cities))
        random.shuffle(route)
        population.append(route)
    return population
```

This function generates a population of random routes. Each route is a permutation of cities, representing a possible solution to the TSP.

```
def selection(population):
     """Perform selection based on fitness, keeping the top half of the
population."""
                   return     sorted(population,     key=calculate_fitness,
reverse=True)[:int(population_size * 0.5)]
```

Selection is performed by sorting the population based on fitness in descending order and keeping the top half of individuals. This way, individuals with higher fitness have a higher chance of being selected.

```
def crossover(parent1, parent2):
    """Perform crossover between two parents to create a child."""
    crossover_point = random.randint(0, num_cities - 1)
     child = parent1[:crossover_point] + [city for city in parent2 if city
not in parent1[:crossover_point]]
    return child
```

Crossover combines genetic material from two parents to create a child. In this case, a random crossover point is chosen, and the child inherits a portion of one parent and the remaining cities from the other parent.

```
def mutate(route):
    """Perform mutation on a route with a certain probability."""
    if random.random() < mutation_rate:
        index1, index2 = random.sample(range(num_cities), 2)
        route[index1], route[index2] = route[index2], route[index1]
    return route
```

Mutation introduces small changes in an individual route. In this case, with a certain probability (mutation_rate), two random cities in the route are swapped.

```
def genetic_algorithm():
    """Run the genetic algorithm to find the best route."""
    population = initialize_population()

    for generation in range(num_generations):
        population = selection(population)

        for _ in range(int(population_size * (1 - crossover_rate))):
            parent1, parent2 = random.sample(population, 2)
            child = crossover(parent1, parent2)
            child = mutate(child)
            population.append(child)

    best_route = max(population, key=calculate_fitness)
    return best_route
```

The main genetic algorithm function initializes a population, performs selection, crossover, and mutation for a certain number of generations. It returns the best route found during the optimization process.

```
# Run the genetic algorithm
best_route = genetic_algorithm()

# Print the results
print("Best route:", best_route)
print("Total distance:", 1 / calculate_fitness(best_route))
```

Finally, the program runs the genetic algorithm and prints the best route and the total distance for that route.

4. **Give your final answer. Indicate the neighborhood definition and the stopping criterion you used and tell whether you arrived at an optimal or near optimal solution.**

The output that we got using the code above is,

```
Best route: [3, 1, 2, 5, 4, 0]
Total distance: 116.0
```

The neighborhood in this genetic algorithm is defined implicitly through the crossover and mutation operations. Crossover combines genetic material from two parents, and mutation introduces small changes to an individual route. These operations allow the algorithm to explore the solution space by creating new routes based on existing ones, forming a dynamic and evolving neighborhood. The stopping criterion in this implementation is a fixed number of generations (num_generations). The algorithm runs for a specified number of iterations, and after reaching the predefined number of generations, it stops. Another common stopping criterion is to monitor the convergence of the algorithm, such as when there is no significant improvement in the best route over several generations or when a specific target fitness value is reached.

Given the large number of iterations and consistent results in multiple runs, we can confidently conclude that this solution is optimal for the given problem.