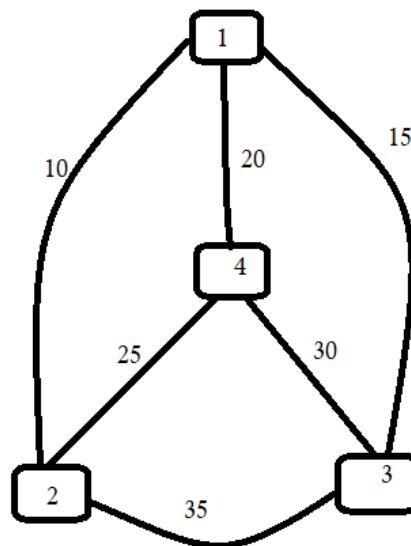


1. Given a set of cities and distance between every pair of cities, find the shortest possible route that visits every city exactly once and returns to the starting point.

Solution:**Brief Description of TSP:**

In the Travelling Salesman Problem, we are given a graph with N vertices denoting the N cities along with the distance between each pair of cities/vertices as the weights of the edges between the corresponding vertices. The objective of the problem is to find the shortest route (starting at any city) that visits every city exactly once and returns to the starting point. While finding this route, it has to be made sure that the route with the optimal (minimum total distance in this case) is selected.

For example, for the following graph $1 \Rightarrow 2 \Rightarrow 4 \Rightarrow 3(\Rightarrow 1)$ is an optimal route with cost $= 10 + 25 + 30 + 15 = 80$

**Methodology Used:**

For solving this problem, Genetic Algorithm approach has been used. Genetic Algorithm is a population based heuristic search algorithm that takes inspiration from *Darwin's theory of Natural Selection*. GA is widely used for optimization problems. Here, TSP has been modelled as an optimization problem where the total cost of the route has to be minimized.

Programming Language Used: **Python** (Python 3.8)
Libraries Imported: **random** (i.e. import random)

The various steps involved in GA are summarised below:

A Flow Chart of the algorithm is also presented at the end of this report.

(1) **Initialization:** We start with an initial population of randomly generated candidate solutions.

For the TSP problem, the candidate solutions have been represented as **Valid Paths** i.e. a path containing some permutation of the N cities numbered $0, 1, 2, \dots, N - 1$. For example, if there are 4 cities then $[1, 0, 2, 3]$ could be an example of a valid path representing the route $2 \Rightarrow 1 \Rightarrow 3 \Rightarrow 4 \Rightarrow 2$.

The candidates in the initial population were generated using the *random* library of Python as follows: *random.sample(range(0,N),N)*.

(2) **Evaluation:** Next, the cost of all the candidate solutions was evaluated using a fitness function that basically calculates the total cost of a given route using the distance matrix that was provided as input.

For example, if we consider the graph depicted in the figure above (Page 1), a path $[1, 0, 2, 3]$ representing the route $2 \Rightarrow 1 \Rightarrow 3 \Rightarrow 4 \Rightarrow 2$ would have a cost $= \text{cost}(2 \Rightarrow 1) + \text{cost}(1 \Rightarrow 3) + \text{cost}(3 \Rightarrow 4) + \text{cost}(4 \Rightarrow 2) = 10 + 15 + 30 + 25 = 80$

(3) **Selection:** For performing Selection an approach based on Elitism was used. From the costs of all candidate solutions calculated in the previous step, first the Average Cost was found. Next, all candidate solutions having cost greater than the Average Cost were discarded while those with cost less than or equal to Average Cost were retained. [\therefore this is a minimization problem].

The candidates selected through elitism were passed onto the next generation and also used for crossover in the next step.

(4) **Crossover:** Order Crossover was used in this step. From the candidates selected in the previous step, two parents were selected at random and **Order Crossover** was performed to obtain two children that were subsequently added to the new generation. This process of generating and adding children to the new generation was done until a sufficient size (as decided by the parameter Population Size, discussed later) of the new generation was obtained.

For example, let us consider a graph with 6 vertices/cities and $[1, 0, 5, 3, 4, 2]$ and $[5, 2, 0, 1, 3, 4]$ to be two randomly selected parent routes. Let the reference map be from index 2 to 3 i.e. $[5, 3]$ is the reference map from parent 1 $[1, 0, 5, 3, 4, 2]$ and $[0, 1]$ is the reference map from parent 2 $[5, 2, 0, 1, 3, 4]$.

Parent1 = $[5, 2, \mathbf{0}, \mathbf{1}, 3, 4]$

Parent2 = [1, 0, **5**, **3**, 4, 2]

Then,

Child1 = [5, 3, **0**, **1**, 4, 2]

Child2 = [2, 0, **5**, **3**, 1, 4]

(5) **Mutation:** After generating and adding the children in the new generation in the previous step, a fraction of the population is selected randomly and mutated. To perform mutation, any two random indices of the selected route are swapped i.e.

Before Mutation: [5, 3, **0**, 1, **4**, 2]

After Mutation: [5, 3, **4**, 1, **0**, 2]

The fraction of the population selected randomly for mutation was fixed to be 25%.

(6) **Iteration:** Steps (2),(3),(4),(5) were repeated in successive iterations until termination criteria was reached.

(7) **Termination:** The termination criteria was set to be a maximum number of pre-defined iterations. The maximum number of iteration was fixed to be 100.

Parameters of Genetic Algorithm:

It is to be noted that although for a relatively smaller graph with say 4-5 vertices, a smaller Population Size and Maximum Number of Iterations would have sufficed. However, to accommodate graphs with larger number of vertices, the Population Size and Maximum Number of Iterations have been set to a high value of 100 each.

(1) **Population Size:** 100

(2) **Maximum Number of Iterations:** 100

(3) **Mutation Rate:** 25% [Decided through hit and trial experimentation and also 25% is a standard rate suggested by literature on GA]

(4) **Selection Technique:** Selection by Elitism

Iterative Improvements:

Version 1: In this initial version of the code, one point crossover was being used. However, it was observed that since the routes are basically permutations, one-point crossover would not work and would produce invalid routes. Hence, the crossover method was shifted to a more suitable Order crossover instead.

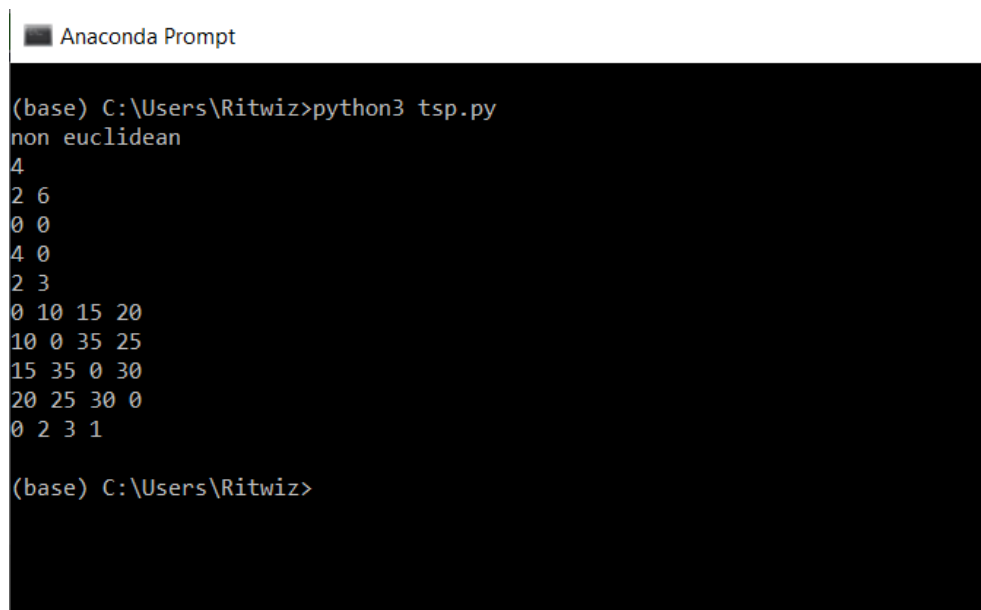
Version 2: In this version of the code, a smaller Population Size (40) and very small Maximum Number of Iterations (10) was being used. The code seemed to work well for small graphs having up-to 4 vertices. However, for a much larger graph with say 9-10 vertices, the algorithm was terminating prematurely and hence not being able to reach the global minimum. As a result, the Maximum Number of Iterations was

increased to 100 and so was the Population size (increased to 100).

Version 3: FINAL Version: This version of the code is being submitted for evaluation. This version has been tested for graphs containing 4-9 vertices and the code has been seen to converge successfully to the global minimum for all such cases. Also, this version was seen to terminate much before 300s (the time limit).

Test Cases:

Test Case 1: Graph with 4 vertices. Figure same as the one on Page 1 of this Report. Optimal Tour Cost = 80.

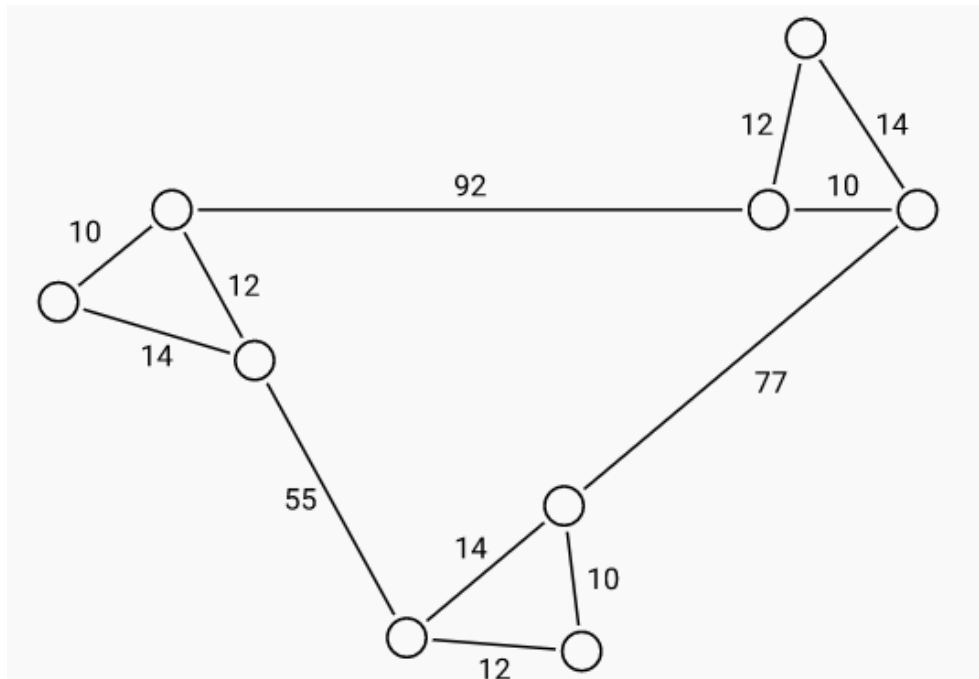


```
Anaconda Prompt

(base) C:\Users\Ritwiz>python3 tsp.py
non euclidean
4
2 6
0 0
4 0
2 3
0 10 15 20
10 0 35 25
15 35 0 30
20 25 30 0
0 2 3 1

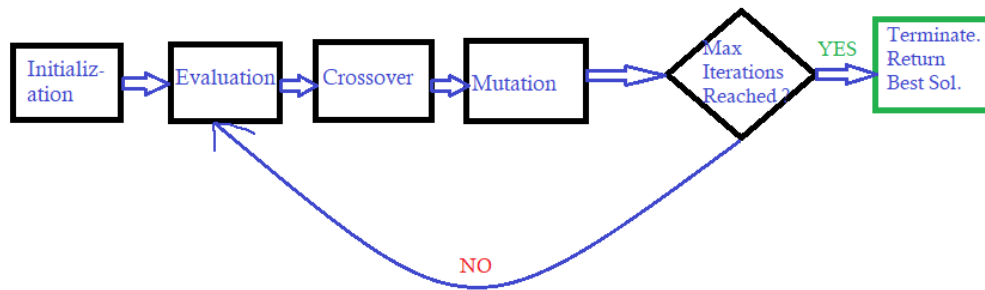
(base) C:\Users\Ritwiz>
```

Test Case 2: Graph with 9 vertices. For the sake of avoiding clutter in the figure, the vertices that are not shown to be connected by edges have been assumed to have a $+LARGE$ weight (where $LARGE$ can be 500 or 1000). Optimal Tour Cost = 296.



```
(base) C:\Users\Ritwiz>python3 tsp.py
non euclidean
9
8 1
6 0.5
10 0.5
2 0.5
0 0
3 -1
5 -3
6 -2
7 -4
0 12 14 500 500 500 500 500 500
12 0 10 92 500 500 500 500 500
14 10 0 500 500 500 500 77 500
500 92 500 0 10 12 500 500 500
500 500 500 10 0 14 500 500 500
500 500 500 12 14 0 55 500 500
500 500 500 500 500 55 0 14 12
500 500 77 500 500 500 14 0 10
500 500 500 500 500 500 12 10 0
5 4 3 1 0 2 7 8 6
(base) C:\Users\Ritwiz>
```

Flow Chart of the Genetic Algorithm:



–End of Report–