UNIVERSITI TUNKU ABDUL RAHMAN FACULTY OF ENGINEERING AND SCIENCE

UECS2053 UECS2153 UEMH3073 UEMH3163 Artificial Intelligence May 2022 Trimester
Lab 2: Genetic Algorithm

Practical Group: P3

Group Members:

Name	Student ID	Year/Trimester	Programme
Lau Yong Yang	2003578	Y3/T1	MH
Ong Kar Ming	1803674	Y4/T1	AM
Siow Wen Hao	2003860	Y3/T1	MH
Teh Shao Yi	1904381	Y3/T1	BI

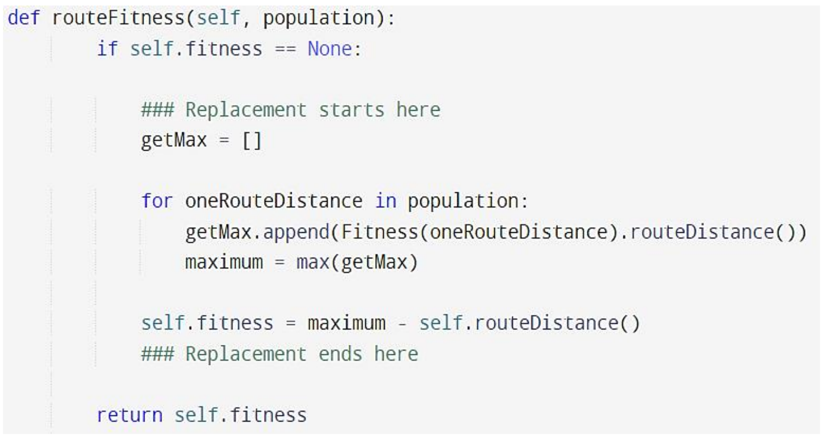
Lecturer: Prof. Dr Yau Kok Lim

Date of Submission: 09/08/2022, 5 p.m.

REPLACEMENT CODE & EXPLANATION

Fitness

#TODO1 – Fitness Function 2



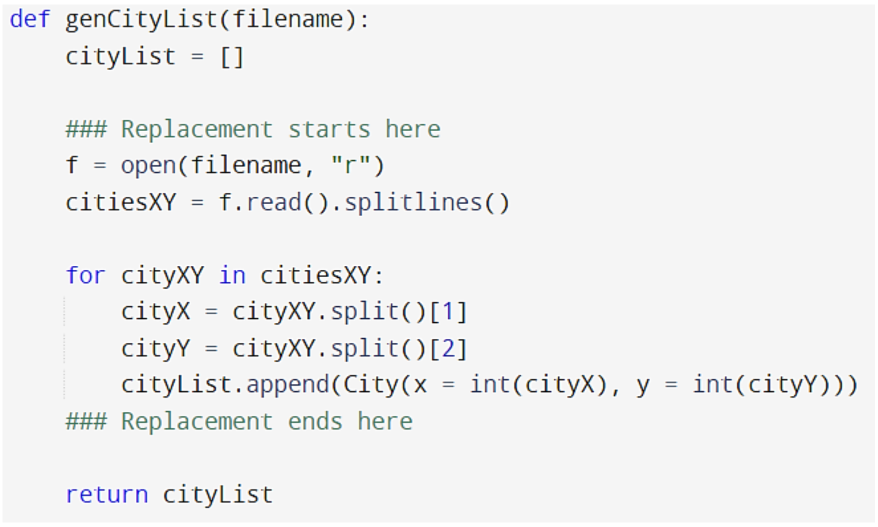
```
def routeFitness(self, population):
    if self.fitness == None:

        ### Replacement starts here
        getMax = []

        for oneRouteDistance in population:
            getMax.append(Fitness(oneRouteDistance).routeDistance())
            maximum = max(getMax)

        self.fitness = maximum - self.routeDistance()
        ### Replacement ends here

    return self.fitness
```

Figure 1: New code for #TODO1

Firstly, create a list to store all the route distances calculated for the current generation (population). Then, loop through the generation (population) and calculate the route distance of each route in the current generation (population). After that, store them in getMax. This variable represents the best-known route with the highest fitness value in the data. Then, choose a route to calculate its fitness value and subtract it from 'max.' The result is then assigned to self. fitness in which the objective is to determine the difference between the current route and the best-known route with the maximum fitness value.

Population Initialization

#TODO2 – Read a set of cities from text file



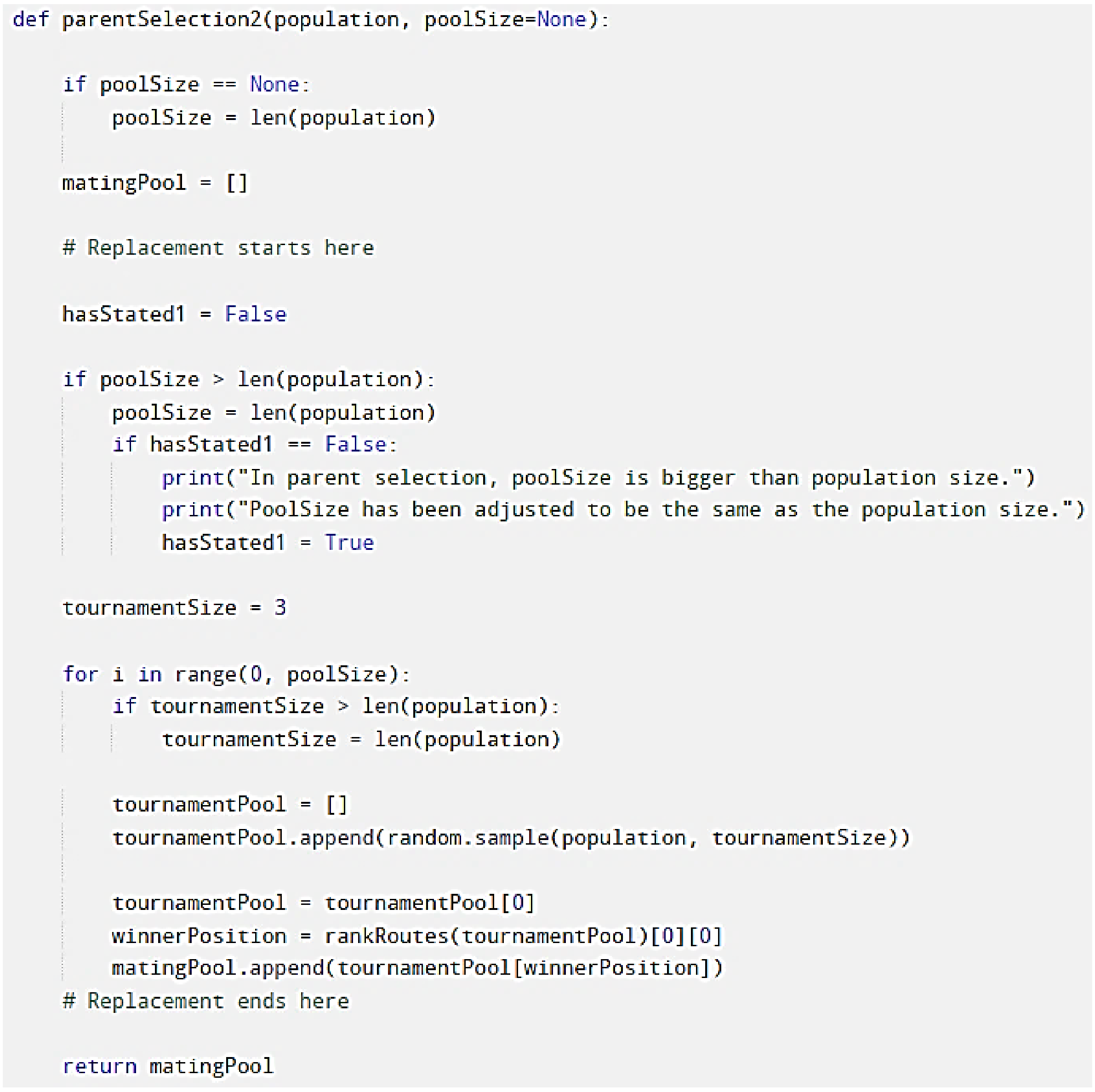
```
def genCityList(filename):
    cityList = []

    ### Replacement starts here
    f = open(filename, "r")
    citiesXY = f.read().splitlines()

    for cityXY in citiesXY:
        cityX = cityXY.split()[1]
        cityY = cityXY.split()[2]
        cityList.append(City(x = int(cityX), y = int(cityY)))
    ### Replacement ends here

    return cityList
```

Figure 2: New code for #TODO2

Firstly, read the "cities8.txt" or "cities500.txt" file. Then, the line is split into words by using space as a divider and it is stored in the variable ‘city’. The first variable in the variable 'city' is the city number, the second variable is the city's x coordinate, and the third variable is the city's y coordinate. Then, using the second and third variables, a city object is created and added to the city list. After all data has been recorded, the function will return the cityList to the main program. The file will be closed when the block within the with statement is exited (finishing the instructions after opening the file) without explicitly calling the close() method.

<u>Parent Selection</u>

#TODO3 – Tournament Selection

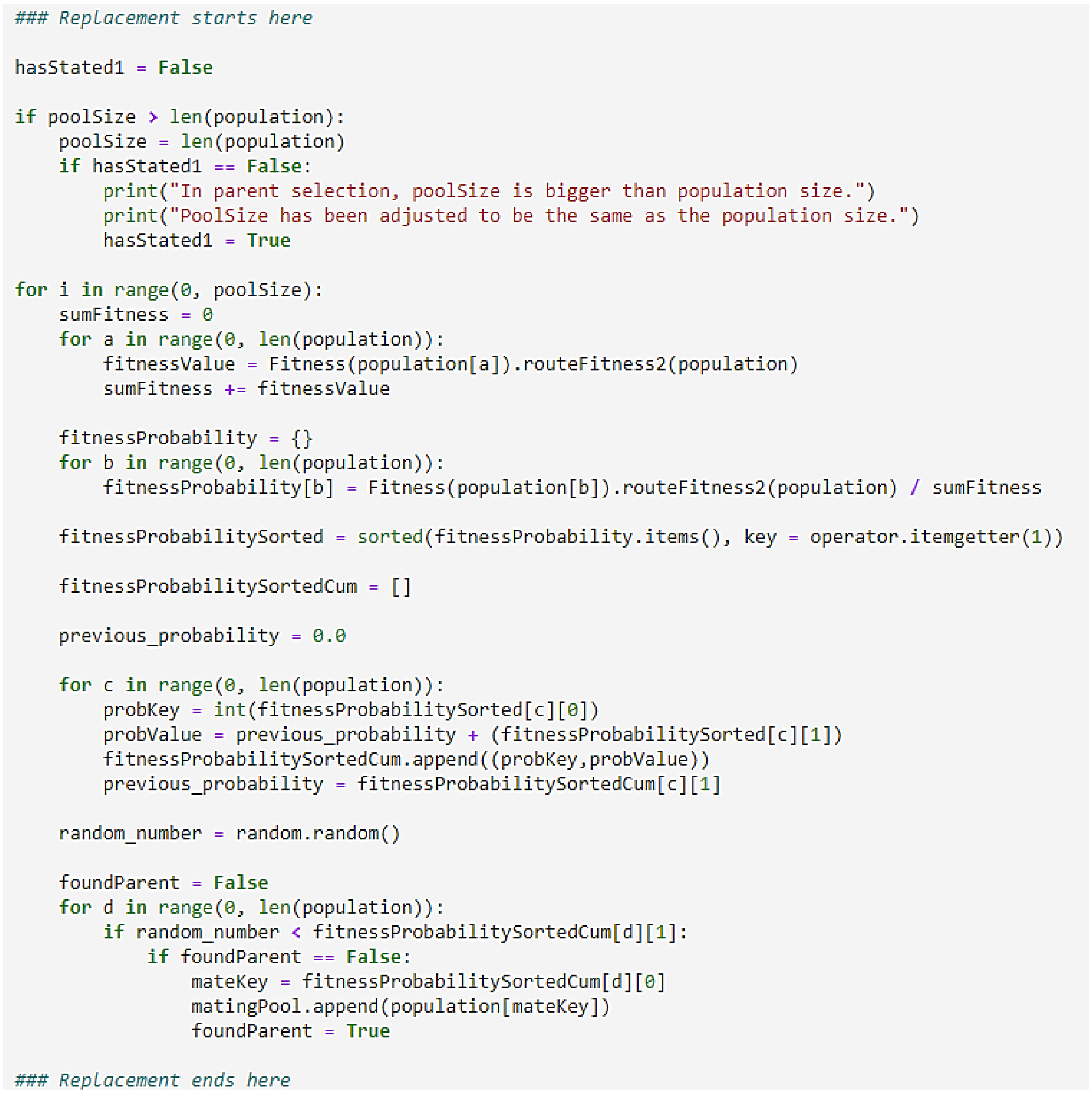


```
def parentSelection2(population, poolSize=None):

    if poolSize == None:
        poolSize = len(population)

    matingPool = []

    # Replacement starts here

    hasStated1 = False

    if poolSize > len(population):
        poolSize = len(population)
        if hasStated1 == False:
            print("In parent selection, poolSize is bigger than population size.")
            print("PoolSize has been adjusted to be the same as the population size.")
            hasStated1 = True

    tournamentSize = 3

    for i in range(0, poolSize):
        if tournamentSize > len(population):
            tournamentSize = len(population)

        tournamentPool = []
        tournamentPool.append(random.sample(population, tournamentSize))

        tournamentPool = tournamentPool[0]
        winnerPosition = rankRoutes(tournamentPool)[0][0]
        matingPool.append(tournamentPool[winnerPosition])
    # Replacement ends here

    return matingPool
```

Figure 3: New code for #TODO3

In this tournament selection, the idea is to select 3 parents randomly and choose the winner (higher fitness value). Then the winner is not allowed to join the next tournament to prevent the same winner occurs and the loop repeats until the number of the poolSize is met. If the poolSize is bigger than population, set the poolSize to the population size. After that, we set the tournament size as 3. Moving on, we need to make sure that there are enough competitors in the population to fit in the tournament size. Else, we just take everyone in the population as the competitors. Next, we create the list of tournament pools and randomly choose the competitors based on the tournament size and fix the tournament pool. Then, append the mating pool with the winner. Highest fitness value, which is the first one after sorting, and get the key of the dictionary (position of the winner). After the selection operations are executed, the outcomes will be appended into the matingPool.

#TODO4 – Proportional Selection



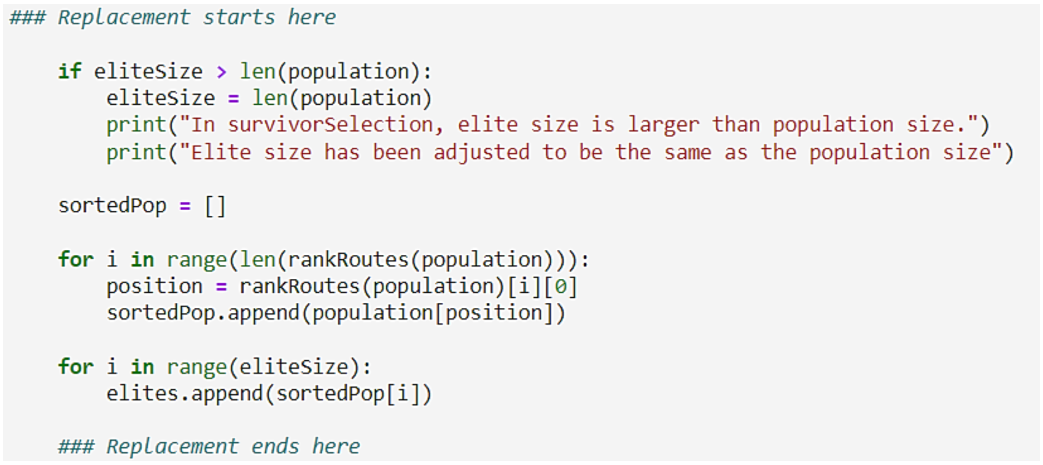
```python
### Replacement starts here

hasStated1 = False

if poolSize > len(population):
    poolSize = len(population)
    if hasStated1 == False:
        print("In parent selection, poolSize is bigger than population size.")
        print("PoolSize has been adjusted to be the same as the population size.")
        hasStated1 = True

for i in range(0, poolSize):
    sumFitness = 0
    for a in range(0, len(population)):
        fitnessValue = Fitness(population[a]).routeFitness2(population)
        sumFitness += fitnessValue

    fitnessProbability = {}
    for b in range(0, len(population)):
        fitnessProbability[b] = Fitness(population[b]).routeFitness2(population) / sumFitness

    fitnessProbabilitySorted = sorted(fitnessProbability.items(), key = operator.itemgetter(1))

    fitnessProbabilitySortedCum = []

    previous_probability = 0.0

    for c in range(0, len(population)):
        probKey = int(fitnessProbabilitySorted[c][0])
        probValue = previous_probability + (fitnessProbabilitySorted[c][1])
        fitnessProbabilitySortedCum.append((probKey,probValue))
        previous_probability = fitnessProbabilitySortedCum[c][1]

    random_number = random.random()

    foundParent = False
    for d in range(0, len(population)):
        if random_number < fitnessProbabilitySortedCum[d][1]:
            if foundParent == False:
                mateKey = fitnessProbabilitySortedCum[d][0]
                matingPool.append(population[mateKey])
                foundParent = True

### Replacement ends here
```

Figure 4: New code for #TODO4

Proportional selection is another parent selection technique, every chromosome can become a parent with a probability proportional to its fitness. Hence, the fitter individuals have a better chance of mating and passing on their features to the next generation. If the poolSize is bigger than population, set the poolSize to the population size. Next, the number of poolSize is loop through to obtain the fitness value for each chromosome (route) and to find the sum of fitness value of the chromosome (route) in the population for fitnessProbability calculation. Then, fitnessProbabilitySortedCum is used to store sorted cumulative probability after the calculation of cumulative probability with the cumulative probability is set to 0.0 for initialization. After that, the key (position of the winner) is obtained from the sorted probability and will be append with the value added to the cumulative probability by the previous probability. Lastly, a number between 0.0 and 0.1 is randomly selected to choose the parent for mating.

Survival Selection

#TODO5 – Merge, Sort & Truncate



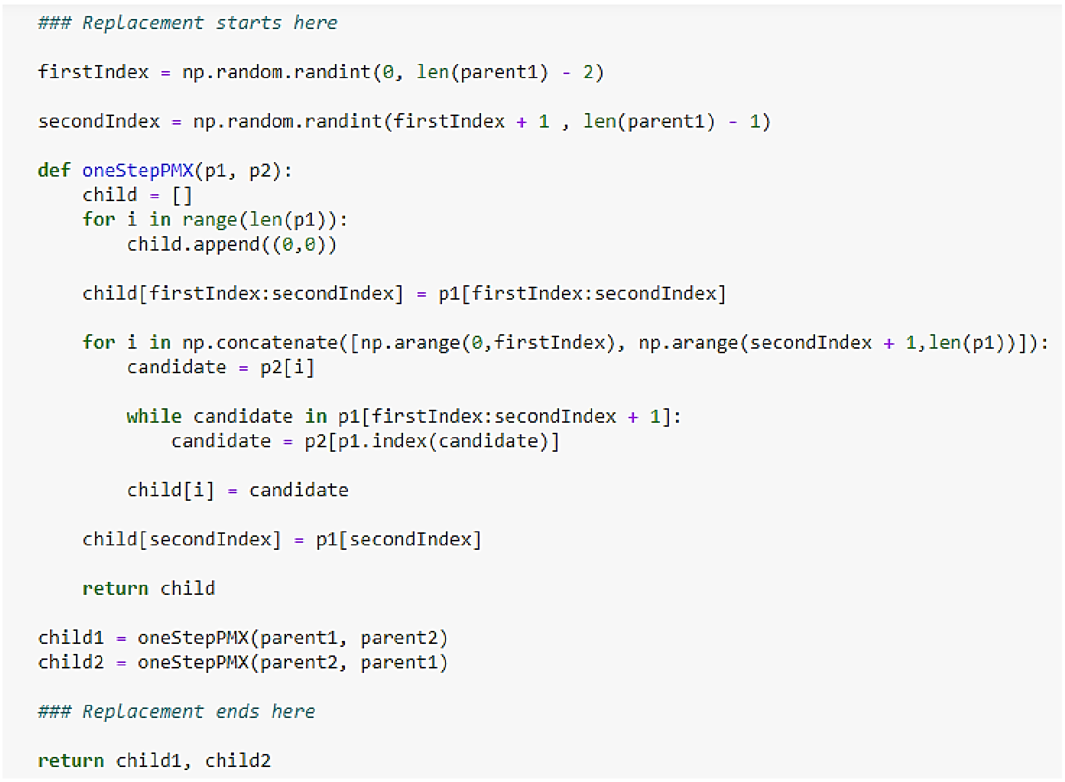
```
### Replacement starts here

    if eliteSize > len(population):
        eliteSize = len(population)
        print("In survivorSelection, elite size is larger than population size.")
        print("Elite size has been adjusted to be the same as the population size")

    sortedPop = []

    for i in range(len(rankRoutes(population))):
        position = rankRoutes(population)[i][0]
        sortedPop.append(population[position])

    for i in range(eliteSize):
        elites.append(sortedPop[i])

    ### Replacement ends here
```

Figure 5: New code for #TODO5

The survivor selection determines who will be kicked out and who will be kept in the next generation. The current solutions (routes) and children produced form a new population. The merged solutions are then sorted based on their fitness values, with better solutions retained and worst solutions truncated. To implement this survivor selection, the elite size is to be make sure smaller or equal to the population size. If not, make the eliteSize be the same as population size. Next, a temporary list is created to store the sorted population using sortedPop. Then, the population is sorted to obtain the ranking and the respective position while append to the temporary elites list. Lastly, append the tops of the sorted population into the elites list based on the eliteSize.

Crossover

#TODO6 - Partially Mapped Crossover (PMX)



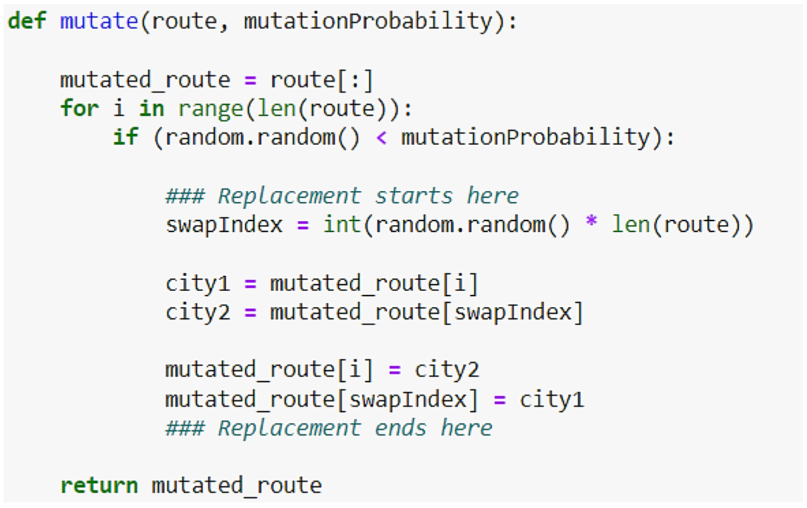
```
### Replacement starts here

firstIndex = np.random.randint(0, len(parent1) - 2)

secondIndex = np.random.randint(firstIndex + 1 , len(parent1) - 1)

def oneStepPMX(p1, p2):
    child = []
    for i in range(len(p1)):
        child.append((0,0))

    child[firstIndex:secondIndex] = p1[firstIndex:secondIndex]

    for i in np.concatenate([np.arange(0,firstIndex), np.arange(secondIndex + 1,len(p1))]):
        candidate = p2[i]

        while candidate in p1[firstIndex:secondIndex + 1]:
            candidate = p2[p1.index(candidate)]

        child[i] = candidate

    child[secondIndex] = p1[secondIndex]

    return child

child1 = oneStepPMX(parent1, parent2)
child2 = oneStepPMX(parent2, parent1)

### Replacement ends here

return child1, child2
```

Figure 6: New code for #TODO6

The genetic operator crossover is used to combine the genetic information of two parents to produce new offspring. Exchanging parts of two solutions usually results in an invalid solution, which is undesirable for the travelling salesman problem, which required a valid travelling path, i.e., no repetition of the chromosome genes. Partially Mapped Crossover (PMX) is used to eliminate the duplication. To implement this partially mapped crossover, firstIndex is used to denote the first index of the mapping selections (under the assumption that at least 3 cities are within a route) while secondIndex point denotes the last index of mapping sections. An inner function oneStepPMX(p1, p2) is declared for ease in following codes. Next, a list of (0,0) with the length of p1 is created to copy the mapping section from p1 to child. Then, copy the remaining section (not in the mapping section) while make sure no duplication in a chromosome and np.concatenate() joins the front and back part of the mapping section (not the mapping section itself). Followingly, a candidate is assigned particular position of element in p2 with while-loop used to make sure that at the end, the candidate does not have the same values with the genes in the mapping section. If the candidate is the same as the gene in the mapping section, the corresponding gene in the opposite chromosome with the same position is picked with the help of index(candidate) to find the corresponding index and p2[p1.index(candidate)] simply means take the corresponding gene in p2 that has the same index(position) of gene. Subsequently, the corresponding gene in the child chromosome with the final value of candidate is updated and updated back child[secondIndex] = p1[secondIndex] as the codes above considers the exclusion of the gene of the position second index. Lastly, oneStepPMX inner function is called with different input of parents to obtain 2 children.

<u>**Mutation**</u>

#TODO7 – Swap Mutation



```python
def mutate(route, mutationProbability):

    mutated_route = route[:]
    for i in range(len(route)):
        if (random.random() < mutationProbability):

            ### Replacement starts here
            swapIndex = int(random.random() * len(route))

            city1 = mutated_route[i]
            city2 = mutated_route[swapIndex]

            mutated_route[i] = city2
            mutated_route[swapIndex] = city1
            ### Replacement ends here

    return mutated_route
```

Figure 7: New code for #TODO7

Mutation of a Genetic Algorithms can be defined as a random tweak in the chromosome to obtain a new chromosome. The genetic algorithms will reduce to a random search when the mutation operation is applied with high probability, hence it is applied with low probability to maintain and introduce diversity in the genetic population. First, index of a city that needed to be swapped is determined randomly and temporary assign the current position's gene values to city1 and city2. The position of the genes with values of city1 and city2 were swapped randomly. The sequence of the route will be changed as a result by selecting 2 genes on the chromosomes at random and interchange the value.

<u>RESULTS & ANALYSIS</u>

#TODO8 – Performance Evaluation

Performance will evaluate based on different fitness function with fix parent selection (tournament selection).

<u>Fitness Function 1 (Simple Division)</u>

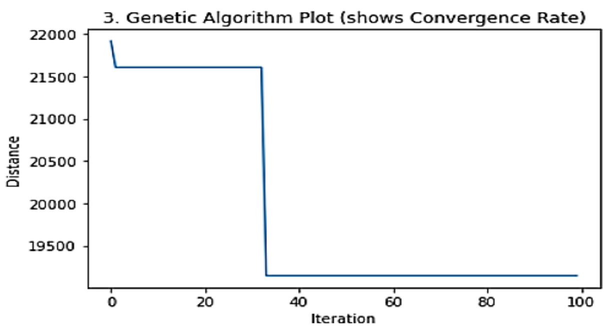
- **Simulation with 8 cities**

  Best distance for initial population : 21964.57355095115

  Best distance for population in last iteration : 19145.65899266102

  Time taken : 0.7399992942810059s

  Optimal path : [(84,4787), (2360,4809), (4116,4225), (4672,1791), (6178,2991), (6248,5294), (6233,5420), (1300,7375)]



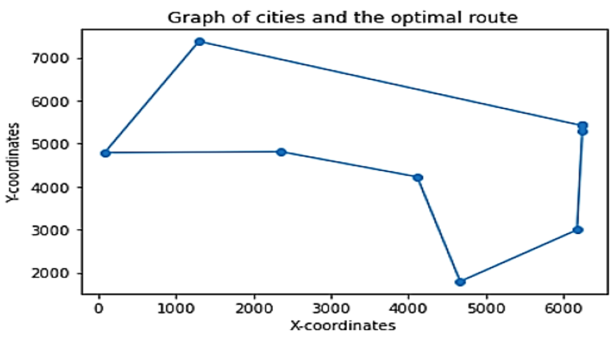


Figure 8.1: Graph of Distance against Iteration *Figure 8.2: Graph of Cities and Optimal Route*

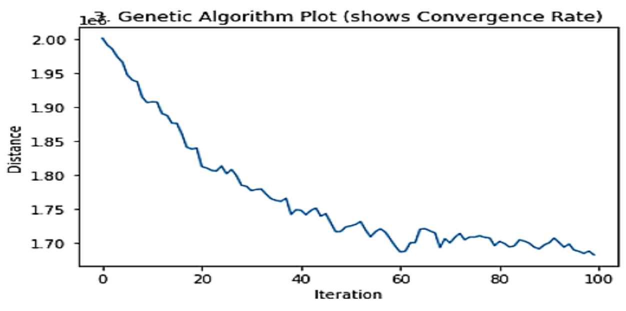
- **Simulation with 500 cities**

  Best distance for initial population : 2005936.4585671655

  Best distance for population in last iteration : 1682396.993714968

  Time taken : 37.65384316444397s

  Optimal path : refer to ipynb file (too many cities)



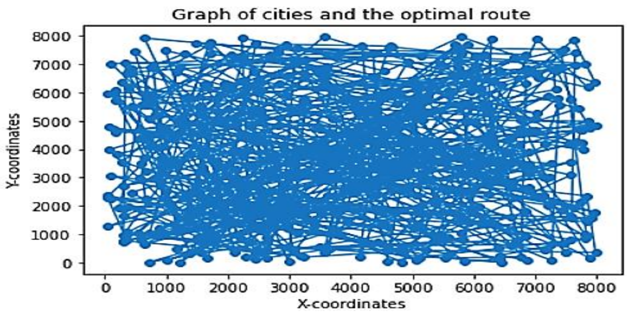


Figure 8.3: Graph of Distance against Iteration *Figure 8.4: Graph of Cities and Optimal Route*

Fitness Function 2 (Maximum Difference)

- **Simulation with 8 cities**

  Best distance for initial population : 21964.57355095115

  Best distance for population in last iteration : 19145.65899266102

  Time taken : 11.144721508026123s

  Optimal path : [(6233,5420), (6248,5294), (6178,2991), (4672,1791), (4116,4225), (2360,4809), (84,4787), (1300,7375)]

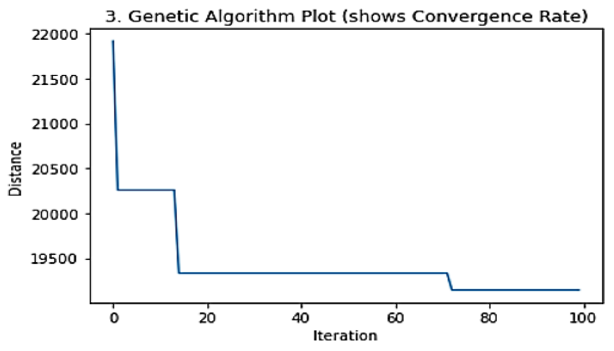


Figure 8.5: Graph of Distance against Iteration

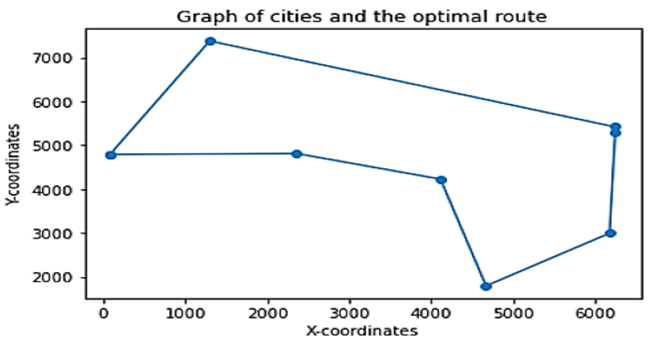


Figure 8.6: Graph of Cities and Optimal Route

- **Simulation with 500 cities**

  Best distance for initial population : 2005936.4585671655

  Best distance for population in last iteration : 1708546.4953848112

  Time taken : 1304.7523748874664s

  Optimal path : refer to ipynb file (too many cities)

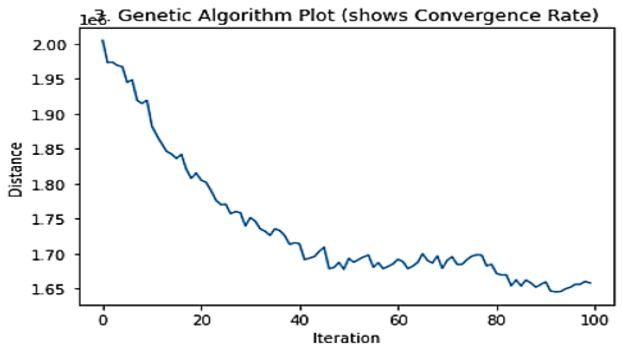


Figure 8.7: Graph of Distance against Iteration

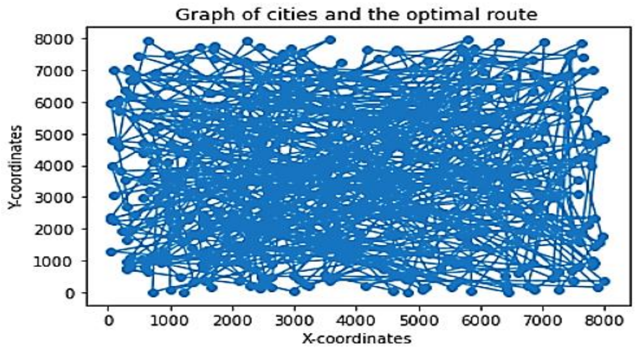


Figure 8.8: Graph of Cities and Optimal Route

Comparison of Performance with Different Fitness Function for Cities8.txt

Fitness Function	Time(seconds)	Shortest Distance	Convergence Rate
Function 1 - Simple Division	0.74	19145.66	Fast
Function 2 - Maximum Difference	11.14	19145.66	Very Fast

- **Time**:
  Function 1 (simple division) is much faster in performing the simulation than Function 2 (maximum difference). This is due to Function 1 only needs to perform inverse of the distance to obtain fitness value. Meanwhile, Function 2 has to store the distances and find maximum value, then only perform subtraction to obtain the fitness value.
- **Shortest Distance**:
  Function 1 can achieve the same optimal route as Function 2.
- **Convergence Rate**:
  Function 2 converges faster than Function 1, but this may cause the reduction of the diversity of the individual in a generation. The results can be observed by comparing figure 8.1 and figure 8.5

Comparison of Performance with Different Fitness Function for Cities500.txt

Fitness Function	Time(seconds)	Shortest Distance	Convergence Rate
Function 1 - Simple Division	37.65	1682396.99	Moderate
Function 2 - Maximum Difference	645.76	1657683.59	Moderately Fast

- **Time**:
  Function 1 (simple division) is much faster in performing the simulation than Function 2 (maximum difference). As the number of cities increases, the amount of distance values to be stored in getMax increase, and the amount of the time needed to loop through the list to obtain maximum value increases.
- **Shortest Distance**:
  Function 2 manages to obtain better optimal route than Function 1, as it converges slightly faster than Function 1.
- **Convergence Rate**:
  From the observation from figure 8.3 and figure 8.7, both functions have almost the same convergence rate.

CONCLUSION

At lower number of cities (population), Function 1 seems to be favoured in term of simulation time and slightly only lower convergence rate than Function 2. At higher number of cities (population), Function 2 can be chosen. Although it seems to take more time for the simulation to run, the optimal route obtained is much better than the one obtained through Function 1. Overall, Function 2 is favoured as it is optimized and works better as the number of populations increases. The best distance for 500 cities population with function 2 (Maximum Difference) is 1657683.59 while for function 1 (Simple Division) is 1682396.99. The best distance for 8 cities population with function 2 (Maximum Difference) and function 1 (Simple Division) is 19145.66.

REFERENCES

De Luca, G. (2020, October 19). *Roulette Selection in Genetic Algorithms*. https://www.baeldung.com/cs/genetic-algorithms-roulette-selection , accessed on 6 August 2022.

Peng Chen, "An Improved Genetic Algorithm for Solving the Traveling Salesman Problem",

In Proceedings of 9th International Conference on Natural Computation, Shenyang, China, 2013.

Stoltz, E. (2018, July 18). *Evolution of a salesman: A complete genetic algorithm tutorial for Python.* Medium. https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35 , accessed on 8 August 2022.