Group Members:

21ucs030 : Arpit Gupta

21ucs031 : Arpit Jain

21ucs073 : Divyansh Garg

21ucs077 : Elishben Manojbhai Baraiya

Fuzzy Logic Controller for Domestic Washing machine

- **Antecedent and Consequent Definition:**

  Dirtiness, grease, and time are linguistic variables representing antecedents and a consequent, respectively.

  The Antecedent class is used to define input linguistic variables (dirtiness and grease), and the Consequent class is used for the output linguistic variable (time).

- **Membership Functions:**

Automf is used to automatically generate membership functions for each linguistic variable. The parameters passed to automf specify the number of membership functions (e.g., 5 for dirtiness and time, 3 for grease) and the names of the linguistic terms.

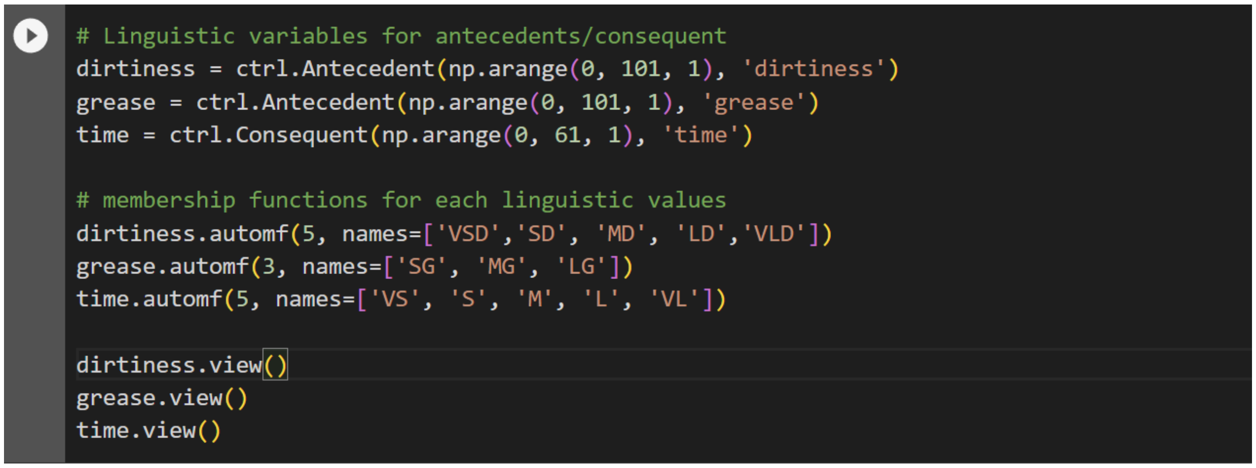
For dirtiness, the terms are 'VSD' (Very Slight Dirt), 'SD' (Slight Dirt), 'MD' (Moderate Dirt), 'LD' (Low Dirt), and 'VLD' (Very Low Dirt).

For grease, the terms are 'SG' (Small Grease), 'MG' (Medium Grease), and 'LG' (Large Grease).

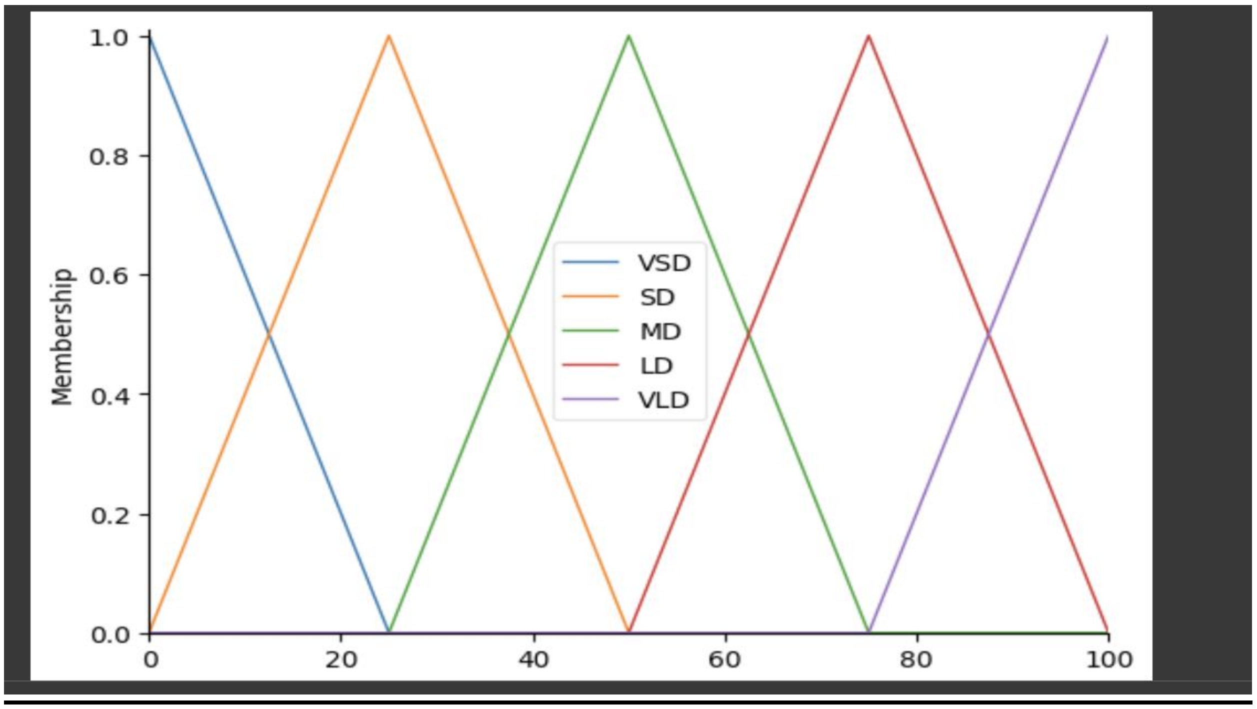
For time, the terms are 'VS' (Very Short), 'S' (Short), 'M' (Medium), 'L' (Long), and 'VL' (Very Long).

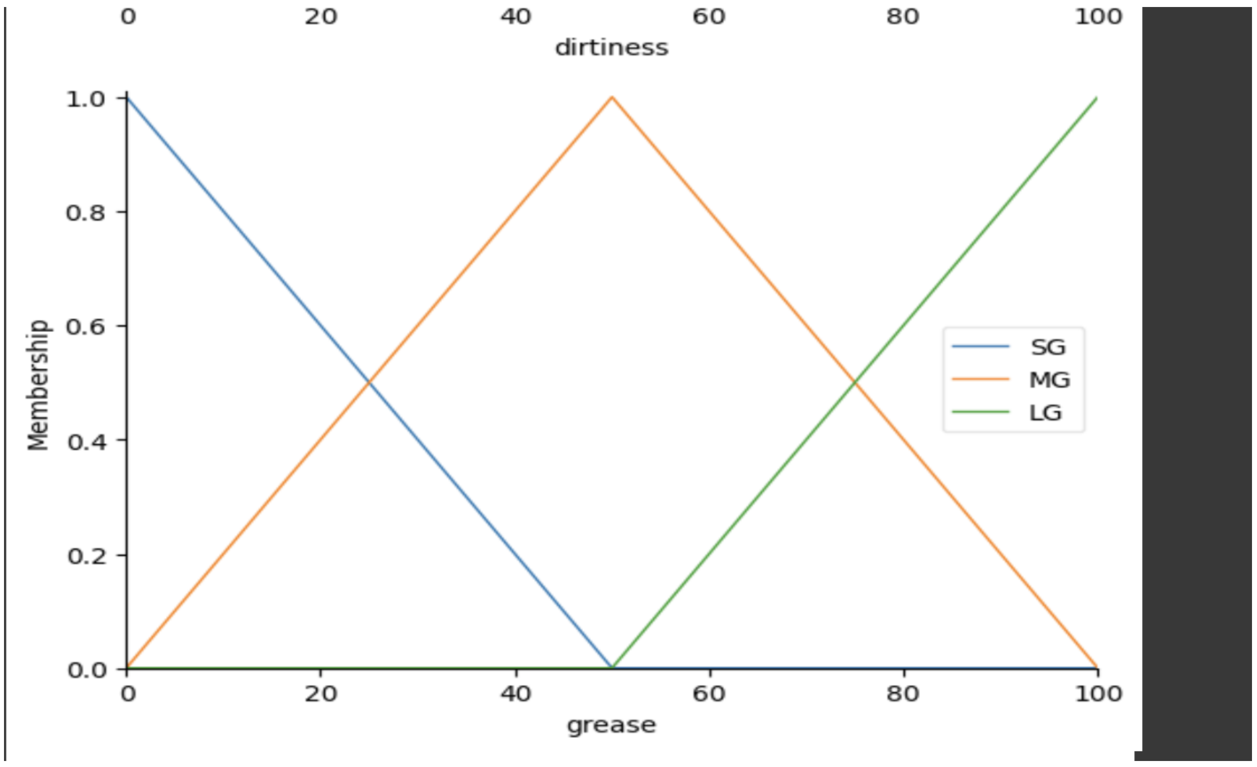
- **Visualization:**

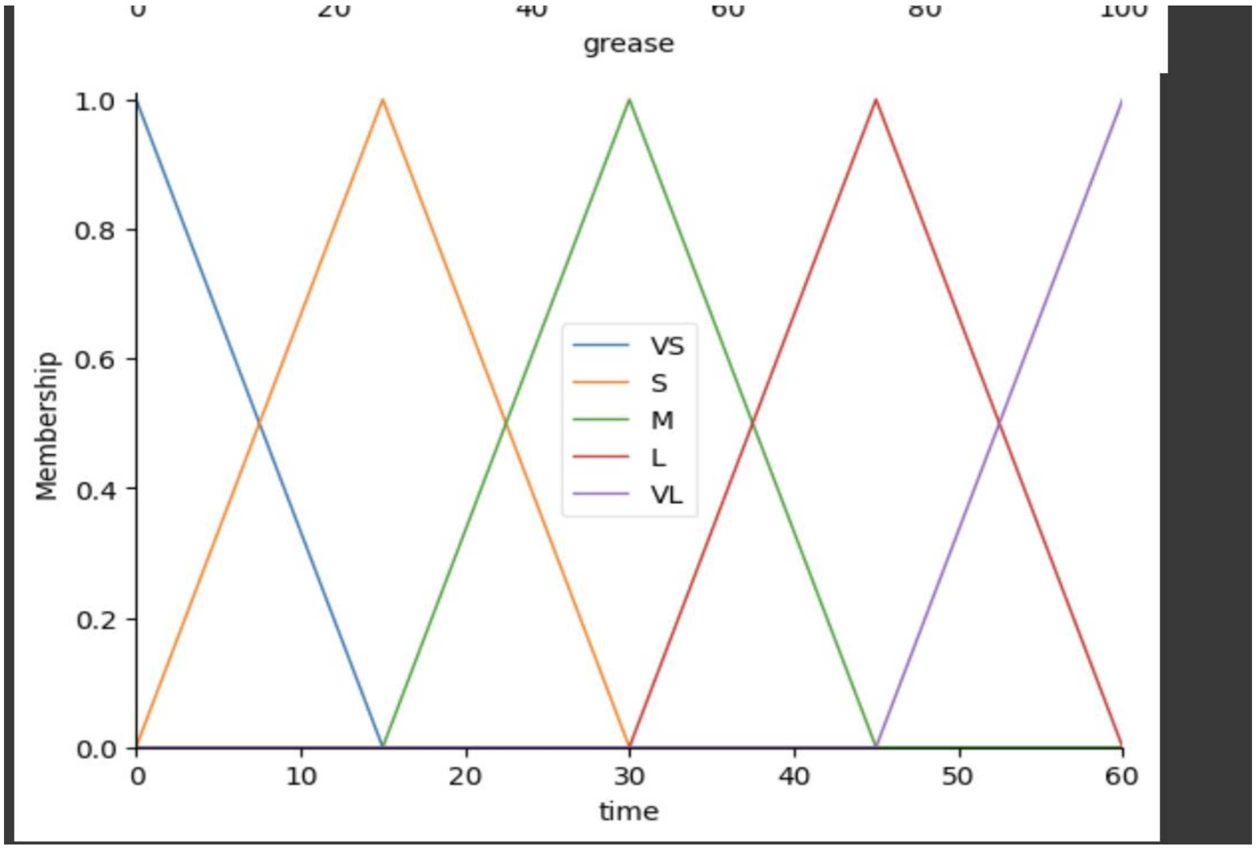
View is used to visualize the membership functions of each linguistic variable. Visualization helps in understanding how the input and output variables are fuzzified into different linguistic terms.



```python
# Linguistic variables for antecedents/consequent
dirtiness = ctrl.Antecedent(np.arange(0, 101, 1), 'dirtiness')
grease = ctrl.Antecedent(np.arange(0, 101, 1), 'grease')
time = ctrl.Consequent(np.arange(0, 61, 1), 'time')

# membership functions for each linguistic values
dirtiness.automf(5, names=['VSD','SD', 'MD', 'LD','VLD'])
grease.automf(3, names=['SG', 'MG', 'LG'])
time.automf(5, names=['VS', 'S', 'M', 'L', 'VL'])

dirtiness.view()
grease.view()
time.view()
```




0
20
40
60
80
100
dirtiness
Membership
1.0
0.8
0.6
0.4
0.2
0.0
SG
MG
LG
0
20
40
60
80
100
grease


grease
Membership
1.0
0.8
0.6
0.4
0.2
0.0
VS
S
M
L
VL
0
10
20
30
40
50
60
time

- **<u>Fuzzy Rules:</u>**

  You've defined 15 fuzzy rules (rule1 through rule15) that establish relationships between combinations of dirtiness, grease, and the resulting time.

  Each rule consists of antecedents (input conditions) and a consequent (output action). For example, rule1 states that if dirtiness is 'VSD' and grease is 'SG', then the time is 'VS' (Very Short).

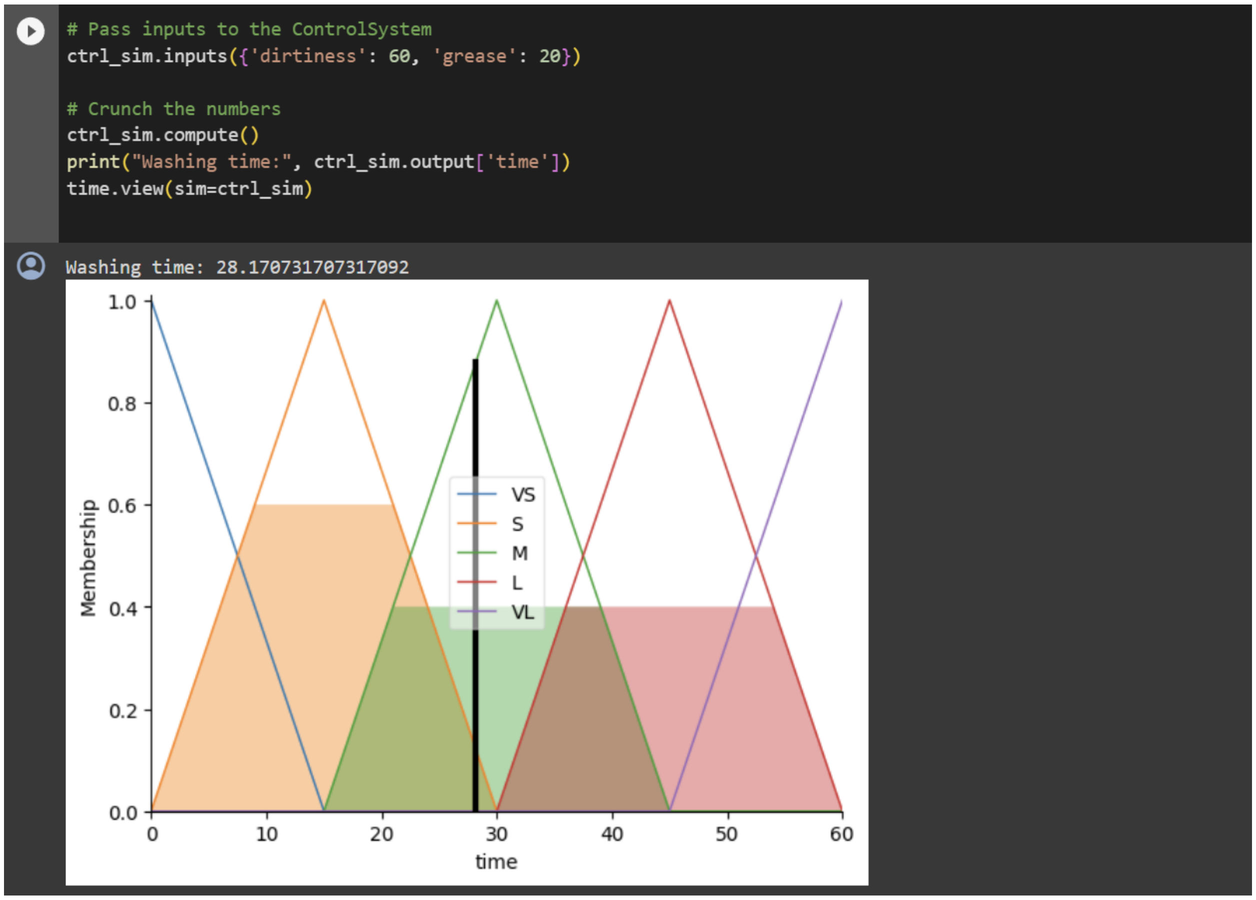
- **<u>Control System:</u>**

  ctrl.ControlSystem is used to create a fuzzy control system that incorporates the defined rules. You pass a list of rules to this constructor to build the rule base.

  The list contains all the rules you've defined ([rule1, rule2, ..., rule15]).

- **<u>Control System Simulation:</u>**

ctrl.ControlSystemSimulation is used to create a simulation object for the control system. This object allows you to input specific values, evaluate the rules, and obtain the output.

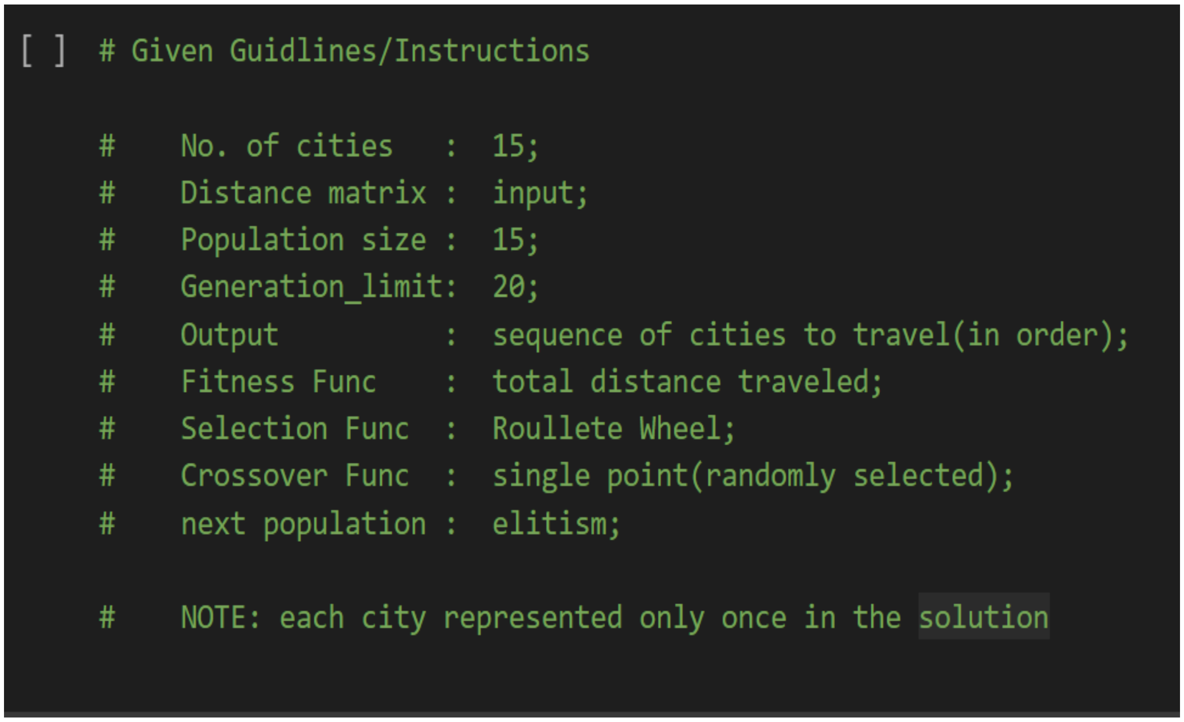


```
# Pass inputs to the ControlSystem
ctrl_sim.inputs({'dirtiness': 60, 'grease': 20})

# Crunch the numbers
ctrl_sim.compute()
print("Washing time:", ctrl_sim.output['time'])
time.view(sim=ctrl_sim)
```

```
Washing time: 28.170731707317092
```

Travelling Salesman Problem solved by Genetic Algorithm

- **Define the Problem:**



```
[ ]  # Given Guidlines/Instructions

     #    No. of cities    :  15;
     #    Distance matrix :  input;
     #    Population size :  15;
     #    Generation_limit:  20;
     #    Output           :  sequence of cities to travel(in order);
     #    Fitness Func     :  total distance traveled;
     #    Selection Func   :  Roullete Wheel;
     #    Crossover Func   :  single point(randomly selected);
     #    next population :  elitism;

     #    NOTE: each city represented only once in the solution
```

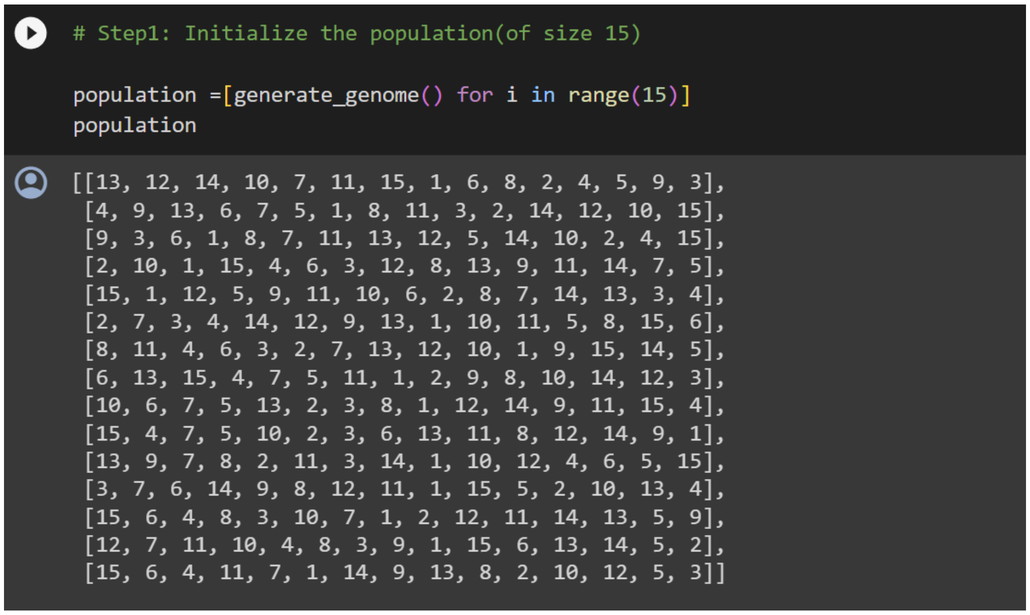
- **Representation:**

1) **Chromosome Representation:**
   Here we have represented the sequence of cities the salesman travels in the form of array of size 15 in which each cell will have a value in range 1-15 (NOTE: no city will be repeated).

2) **Distance Matrix:**
   We also have to represent the distance matrix so that we can later use it to calculate total distance the salesman would have to travel in the whole trip.We have used a 2D matrix for the same.

- **<u>Initialization:</u>**

The population of size 15 is now generated using generate_genome() function which automatically populates the gen0 with 15 random genomes**.**

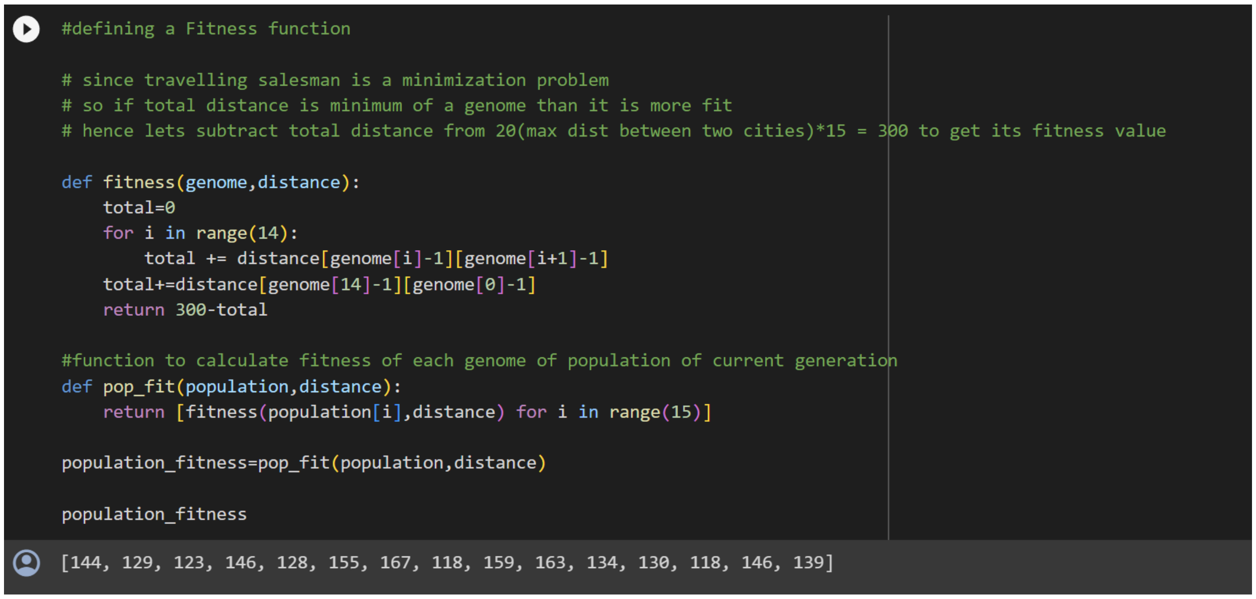


```
# Step1: Initialize the population(of size 15)

population =[generate_genome() for i in range(15)]
population
```

```
[[13, 12, 14, 10, 7, 11, 15, 1, 6, 8, 2, 4, 5, 9, 3],
 [4, 9, 13, 6, 7, 5, 1, 8, 11, 3, 2, 14, 12, 10, 15],
 [9, 3, 6, 1, 8, 7, 11, 13, 12, 5, 14, 10, 2, 4, 15],
 [2, 10, 1, 15, 4, 6, 3, 12, 8, 13, 9, 11, 14, 7, 5],
 [15, 1, 12, 5, 9, 11, 10, 6, 2, 8, 7, 14, 13, 3, 4],
 [2, 7, 3, 4, 14, 12, 9, 13, 1, 10, 11, 5, 8, 15, 6],
 [8, 11, 4, 6, 3, 2, 7, 13, 12, 10, 1, 9, 15, 14, 5],
 [6, 13, 15, 4, 7, 5, 11, 1, 2, 9, 8, 10, 14, 12, 3],
 [10, 6, 7, 5, 13, 2, 3, 8, 1, 12, 14, 9, 11, 15, 4],
 [15, 4, 7, 5, 10, 2, 3, 6, 13, 11, 8, 12, 14, 9, 1],
 [13, 9, 7, 8, 2, 11, 3, 14, 1, 10, 12, 4, 6, 5, 15],
 [3, 7, 6, 14, 9, 8, 12, 11, 1, 15, 5, 2, 10, 13, 4],
 [15, 6, 4, 8, 3, 10, 7, 1, 2, 12, 11, 14, 13, 5, 9],
 [12, 7, 11, 10, 4, 8, 3, 9, 1, 15, 6, 13, 14, 5, 2],
 [15, 6, 4, 11, 7, 1, 14, 9, 13, 8, 2, 10, 12, 5, 3]]
```

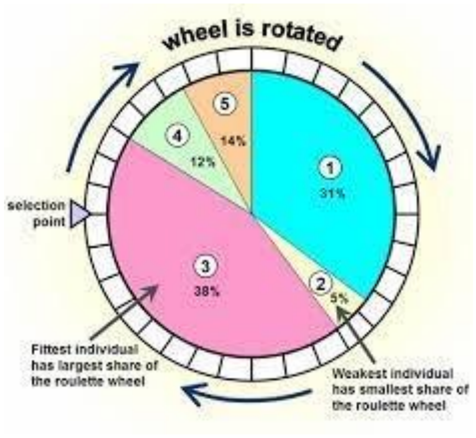
- **<u>Fitness Function :</u>**

Traveling salesman is a minimization problem(of total distance) but inorder to use Roullete wheel in selection process we have made it a maximization problem by subtracting the total distance from max total distance.



```
#defining a Fitness function

# since travelling salesman is a minimization problem
# so if total distance is minimum of a genome than it is more fit
# hence lets subtract total distance from 20(max dist between two cities)*15 = 300 to get its fitness value

def fitness(genome,distance):
    total=0
    for i in range(14):
        total += distance[genome[i]-1][genome[i+1]-1]
    total+=distance[genome[14]-1][genome[0]-1]
    return 300-total

#function to calculate fitness of each genome of population of current generation
def pop_fit(population,distance):
    return [fitness(population[i],distance) for i in range(15)]

population_fitness=pop_fit(population,distance)

population_fitness
```

```
[144, 129, 123, 146, 128, 155, 167, 118, 159, 163, 134, 130, 118, 146, 139]
```

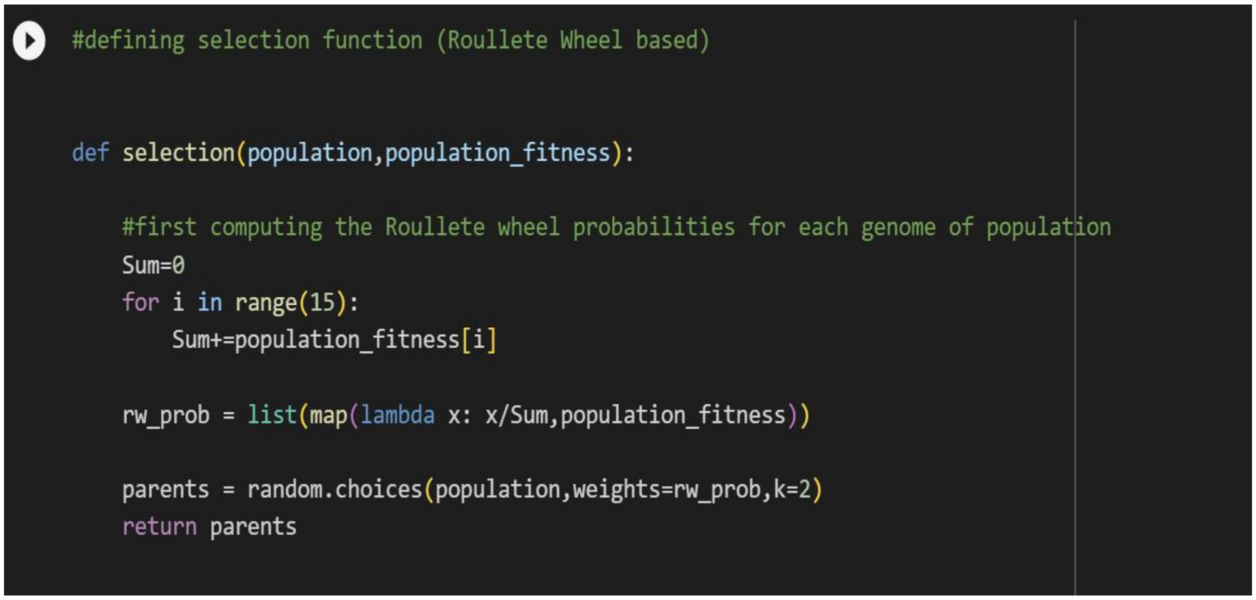

wheel is rotated
5
14%
4
12%
1
31%
selection point
3
38%
2
5%
Fittest individual has largest share of the roulette wheel
Weakest individual has smallest share of the roulette wheel

- **<u>Selection:</u>**

Select individuals from the current population to be parents for the next generation. Individuals are usually selected with a probability proportional to their fitness. Common selection methods include roulette wheel selection, tournament selection, and rank-based selection.

Since we are using the roullete wheel selection mechanism we first need to calculate total fitness and then divide individual fitness with this total to calculate probability of selection of each genom.

After this random.choices() functions is selecting 2 parents based on their probability we have calculated for each genome.



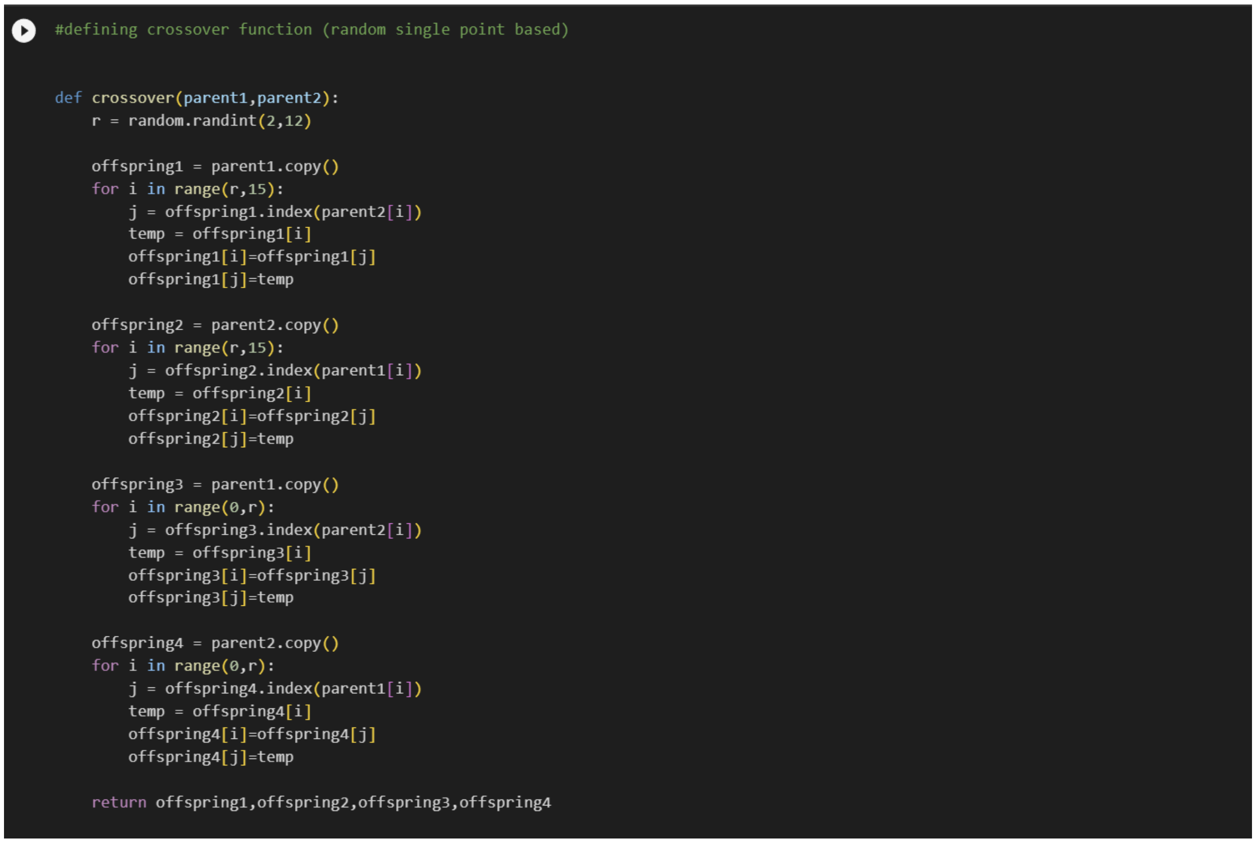
```
#defining selection function (Roullete Wheel based)


def selection(population,population_fitness):

    #first computing the Roullete wheel probabilities for each genome of population
    Sum=0
    for i in range(15):
        Sum+=population_fitness[i]

    rw_prob = list(map(lambda x: x/Sum,population_fitness))

    parents = random.choices(population,weights=rw_prob,k=2)
    return parents
```

- **Crossover (Recombination):**

Combine the genetic material of selected parents to create new individuals (offspring). Crossover is typically performed at randomly chosen points in the genomes of the parents.

Now for the crossover function we have used 2 parents to produce 4 offsprings based on single point crossover.

But since we have to note that each city occurs only once in a genome hence we are going use a swaping mechanism to replace either later/prior part of a genome of parent1/parent2.

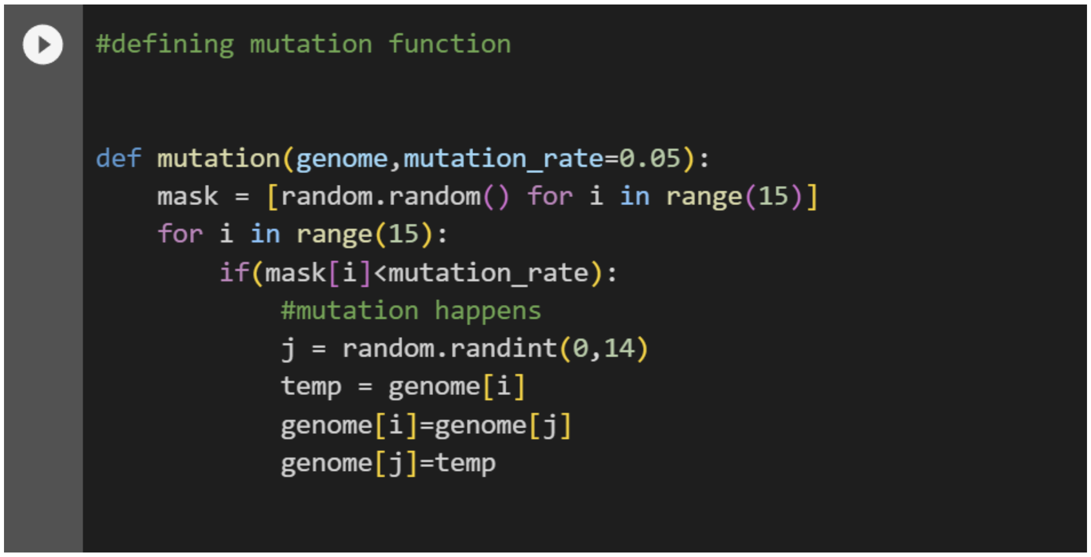


```
#defining crossover function (random single point based)


def crossover(parent1,parent2):
    r = random.randint(2,12)

    offspring1 = parent1.copy()
    for i in range(r,15):
        j = offspring1.index(parent2[i])
        temp = offspring1[i]
        offspring1[i]=offspring1[j]
        offspring1[j]=temp

    offspring2 = parent2.copy()
    for i in range(r,15):
        j = offspring2.index(parent1[i])
        temp = offspring2[i]
        offspring2[i]=offspring2[j]
        offspring2[j]=temp

    offspring3 = parent1.copy()
    for i in range(0,r):
        j = offspring3.index(parent2[i])
        temp = offspring3[i]
        offspring3[i]=offspring3[j]
        offspring3[j]=temp

    offspring4 = parent2.copy()
    for i in range(0,r):
        j = offspring4.index(parent1[i])
        temp = offspring4[i]
        offspring4[i]=offspring4[j]
        offspring4[j]=temp

    return offspring1,offspring2,offspring3,offspring4
```

- **<u>Mutation:</u>**

Introduce small random changes (mutations) to some of the offspring's genes. This adds diversity to the population and helps explore new regions of the solution space.

For mutation we have generated a mask of random numbers between 0 and 1 for the genome and if mask is less than mutation rate(by default 0.05) we are going to swap this city with another random city.

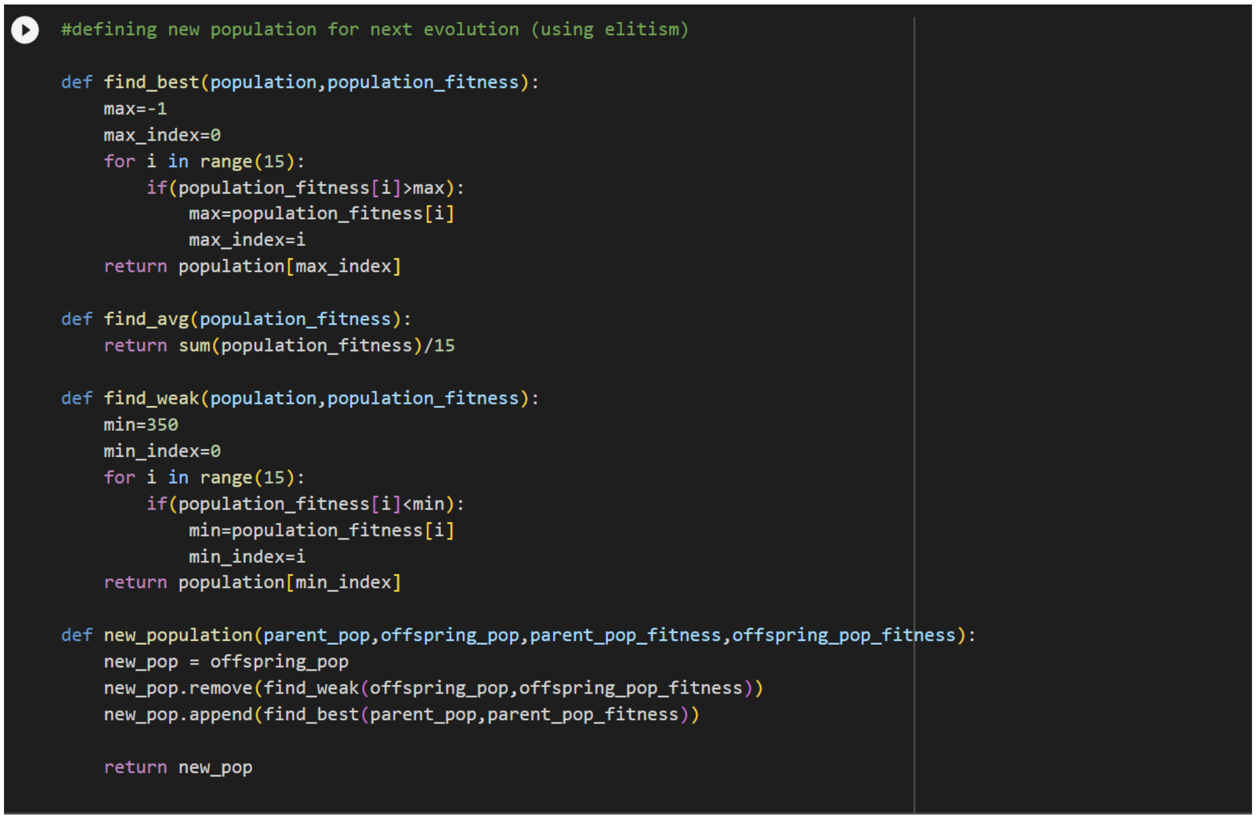


```python
#defining mutation function


def mutation(genome,mutation_rate=0.05):
    mask = [random.random() for i in range(15)]
    for i in range(15):
        if(mask[i]<mutation_rate):
            #mutation happens
            j = random.randint(0,14)
            temp = genome[i]
            genome[i]=genome[j]
            genome[j]=temp
```

- **<u>Replacement:</u>**

Replace the old population with the new population, which includes parents and offspring. This step ensures that the population evolves over time.

For defining the new generation we have used Elitism principal by removing the weakest genome among offsprings and adding best genome of parent population.

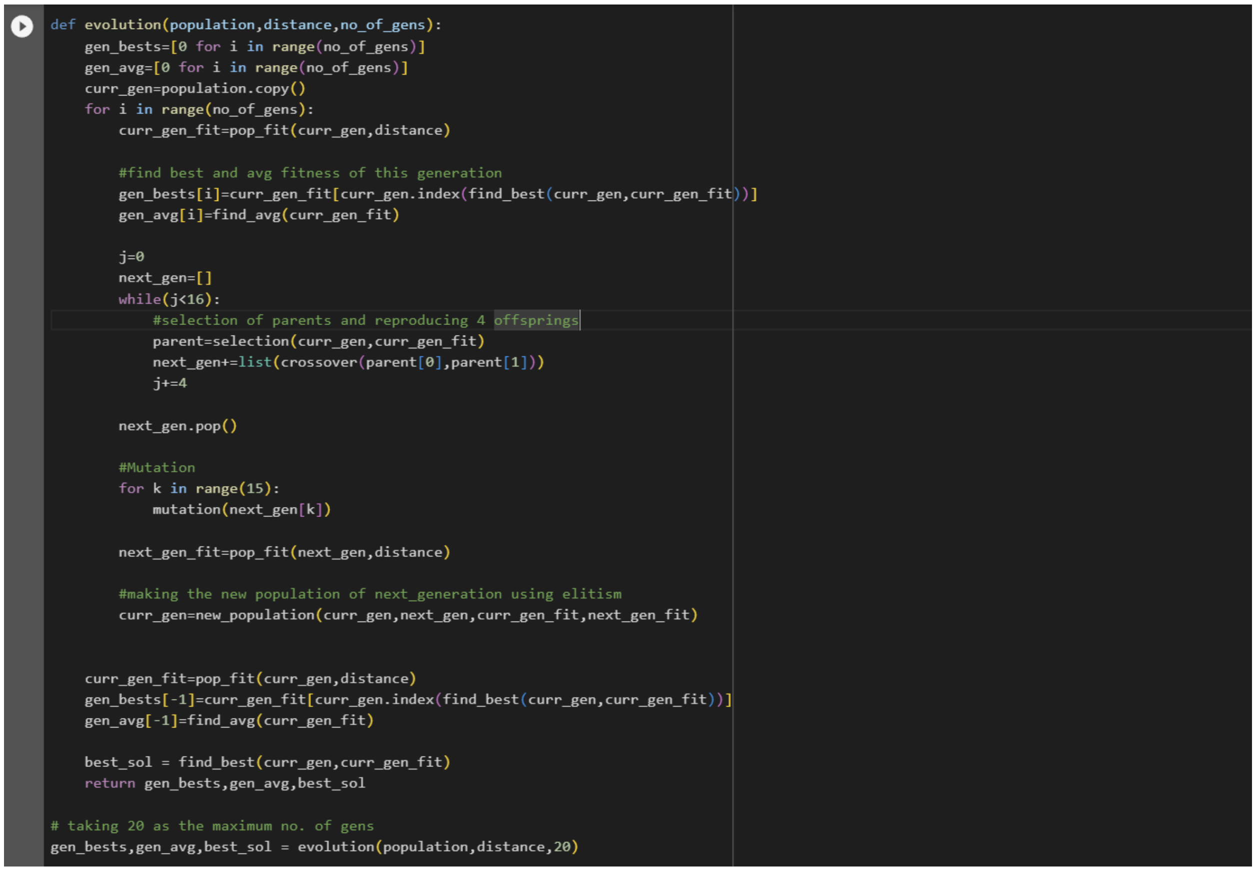


```python
#defining new population for next evolution (using elitism)

def find_best(population,population_fitness):
    max=-1
    max_index=0
    for i in range(15):
        if(population_fitness[i]>max):
            max=population_fitness[i]
            max_index=i
    return population[max_index]

def find_avg(population_fitness):
    return sum(population_fitness)/15

def find_weak(population,population_fitness):
    min=350
    min_index=0
    for i in range(15):
        if(population_fitness[i]<min):
            min=population_fitness[i]
            min_index=i
    return population[min_index]

def new_population(parent_pop,offspring_pop,parent_pop_fitness,offspring_pop_fitness):
    new_pop = offspring_pop
    new_pop.remove(find_weak(offspring_pop,offspring_pop_fitness))
    new_pop.append(find_best(parent_pop,parent_pop_fitness))

    return new_pop
```

- **Final Evolutionary Algorithm:**

Determine the conditions under which the algorithm should stop. This could be a fixed number of generations, reaching a satisfactory solution, or other convergence criteria.

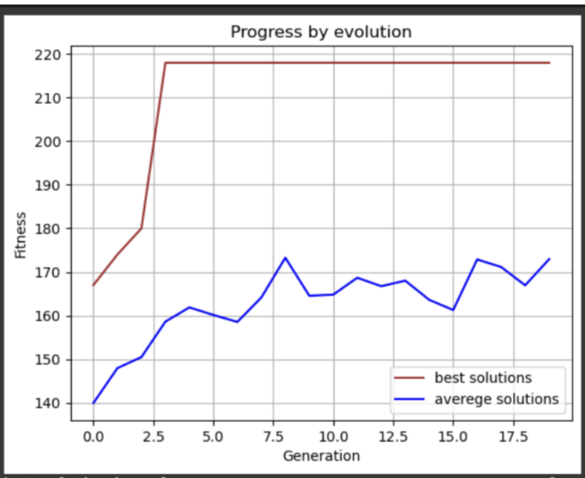
Finally putting together all the functions in final evolution algorithm in proper order by passing population, distance and no. of generations(stopping criteria) we will get best solutions of all times in the final generation.



```
def evolution(population,distance,no_of_gens):
    gen_bests=[0 for i in range(no_of_gens)]
    gen_avg=[0 for i in range(no_of_gens)]
    curr_gen=population.copy()
    for i in range(no_of_gens):
        curr_gen_fit=pop_fit(curr_gen,distance)

        #find best and avg fitness of this generation
        gen_bests[i]=curr_gen_fit[curr_gen.index(find_best(curr_gen,curr_gen_fit))]
        gen_avg[i]=find_avg(curr_gen_fit)

        j=0
        next_gen=[]
        while(j<16):
            #selection of parents and reproducing 4 offsprings
            parent=selection(curr_gen,curr_gen_fit)
            next_gen+=list(crossover(parent[0],parent[1]))
            j+=4

        next_gen.pop()

        #Mutation
        for k in range(15):
            mutation(next_gen[k])

        next_gen_fit=pop_fit(next_gen,distance)

        #making the new population of next_generation using elitism
        curr_gen=new_population(curr_gen,next_gen,curr_gen_fit,next_gen_fit)


    curr_gen_fit=pop_fit(curr_gen,distance)
    gen_bests[-1]=curr_gen_fit[curr_gen.index(find_best(curr_gen,curr_gen_fit))]
    gen_avg[-1]=find_avg(curr_gen_fit)

    best_sol = find_best(curr_gen,curr_gen_fit)
    return gen_bests,gen_avg,best_sol

# taking 20 as the maximum no. of gens
gen_bests,gen_avg,best_sol = evolution(population,distance,20)
```

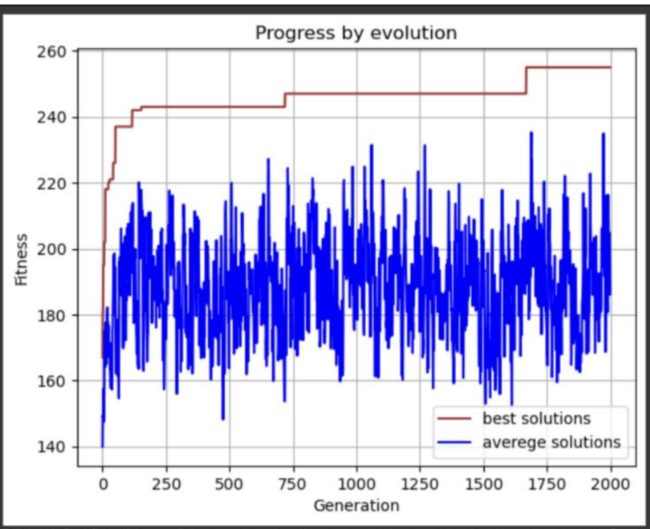
<u>Output:</u>

- <u>No of Generation = 20</u>



```
best solution is :  [6, 3, 9, 1, 12, 7, 14, 4, 10, 2, 11, 5, 13, 15, 8]  : salesman will have to travel  82 km in total
```

- <u>No of Generation = 2000</u>



```
best solution is :  [8, 13, 5, 9, 7, 6, 2, 14, 4, 10, 3, 1, 12, 15, 11]  : salesman will have to travel  45 km in total
```