

A Project Report On
Soft Computing and Optimization Algorithms
(Mini Project)

SUBMITTED BY

Nikhil Jain

Roll No:41425

Atharva Jagtap

Roll No:41423

Shivam Giri

Roll No:41418

CLASS: BE-4

GUIDED BY



DEPARTMENT OF COMPUTER ENGINEERING
PUNE INSTITUTE OF COMPUTER TECHNOLOGY
DHANKAWADI, PUNE-43

SAVITRIBAI PHULE PUNE UNIVERSITY
2020-21

Problem Statement

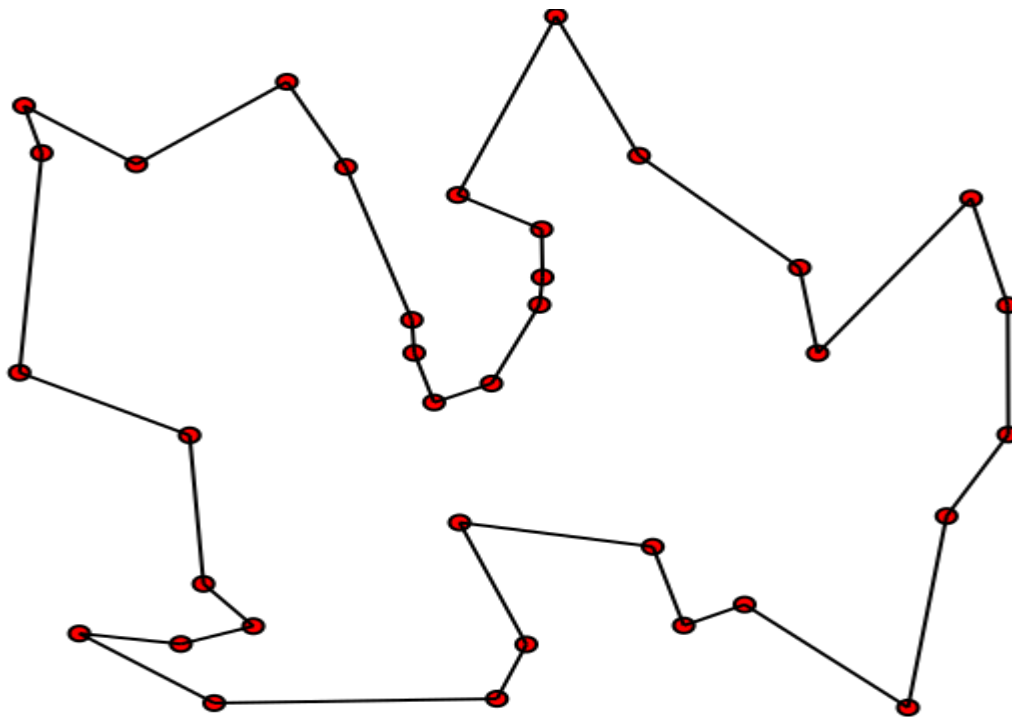
Mini-Project-1on Genetic Algorithm:

Apply the Genetic Algorithm for optimization on a dataset obtained from UCI ML repository. For Example: Travelling Salesman Problem.

Introduction

In this project, we'll be using a GA to find a solution to the traveling salesman problem (TSP). The TSP is described as follows:

“Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?”



Given this, there are two important rules to keep in mind:

1. Each city needs to be visited exactly one time
2. We must return to the starting city, so our total distance needs to be calculated accordingly

The approach

- Important Terminologies related to TSP
- **Gene:** a city (represented as (x, y) coordinates)
- **Individual (aka “chromosome”):** a single route satisfying the conditions above
- **Population:** a collection of possible routes (i.e., collection of individuals)
- **Parents:** two routes that are combined to create a new route
- **Mating pool:** a collection of parents that are used to create our next population (thus creating the next generation of routes)
- **Fitness:** a function that tells us how good each route is (in our case, how short the distance is)
- **Mutation:** a way to introduce variation in our population by randomly swapping two cities in a route
- **Elitism:** a way to carry the best individuals into the next generation

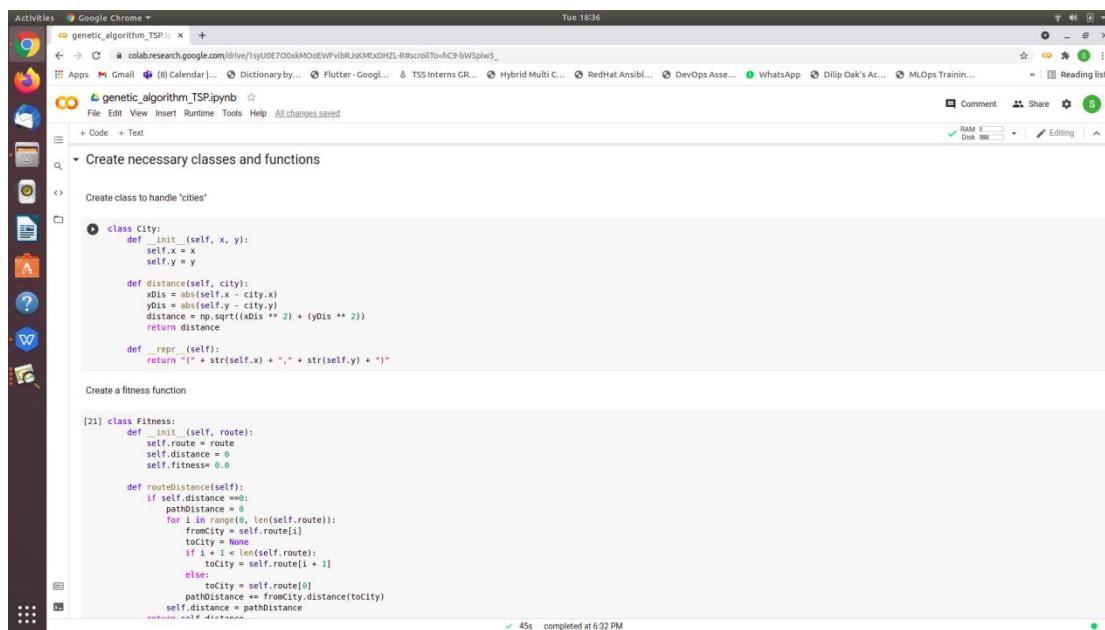
Our GA will proceed in the following steps:

1. Create the population

2. Determine fitness
3. Select the mating pool
4. Breed
5. Mutate
6. Repeat

Building our genetic algorithm

Create two classes: City and Fitness



The screenshot shows a Google Colab notebook titled 'genetic_algorithm_TSP.ipynb'. The notebook is open to a cell with the following code:

```
class City:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, city):
        xDis = abs(self.x - city.x)
        yDis = abs(self.y - city.y)
        distance = np.sqrt((xDis ** 2) + (yDis ** 2))
        return distance

    def __repr__(self):
        return "(" + str(self.x) + "," + str(self.y) + ")"

class Fitness:
    def __init__(self, route):
        self.route = route
        self.distance = 0
        self.fitness = 0.0

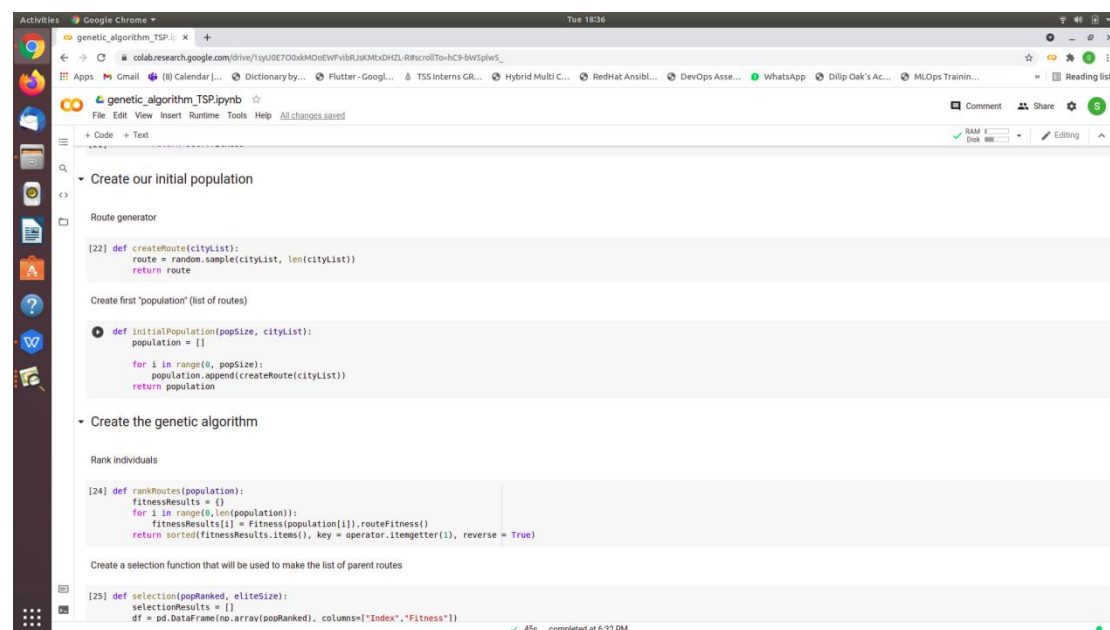
    def routeDistance(self):
        if self.distance == 0:
            pathDistance = 0
            for i in range(0, len(self.route)):
                fromCity = self.route[i]
                toCity = None
                if i + 1 < len(self.route):
                    toCity = self.route[i + 1]
                else:
                    toCity = self.route[0]
                pathDistance += fromCity.distance(toCity)
            self.distance = pathDistance
            return pathDistance
```

The code defines two classes: `City` and `Fitness`. The `City` class has an `__init__` method to set `x` and `y` coordinates, a `distance` method to calculate the distance between two cities using the Pythagorean theorem, and a `__repr__` method to format the city as a coordinate pair. The `Fitness` class has an `__init__` method to set the `route`, `distance`, and `fitness` attributes, and a `routeDistance` method to calculate the total distance of the route.

We first create a `City` class that will allow us to create and handle our cities. These are simply our (x, y) coordinates. Within the `City` class, we add a `distance` calculation (making use of the Pythagorean theorem) in line 6 and a cleaner way to output the cities as coordinates with `__repr__` in line 12.

We'll also create a `Fitness` class. In our case, we'll treat the fitness as the inverse of the route distance. We want to minimize route distance, so a larger fitness score is better. Based on Rule #2, we need to start and end at the same place, so this extra calculation is accounted for in line 13 of the distance calculation.

Create the population



```
[22] def createRoute(cityList):
    route = random.sample(cityList, len(cityList))
    return route

Create first 'population' (list of routes)

[23] def initialPopulation(popSize, cityList):
    population = []
    for i in range(0, popSize):
        population.append(createRoute(cityList))
    return population

Create the genetic algorithm

Rank individuals

[24] def rankRoutes(population):
    fitnessResults = {}
    for i in range(0, len(population)):
        fitnessResults[i] = Fitness(population[i]).routeFitness()
    return sorted(fitnessResults.items(), key = operator.itemgetter(1), reverse = True)

Create a selection function that will be used to make the list of parent routes

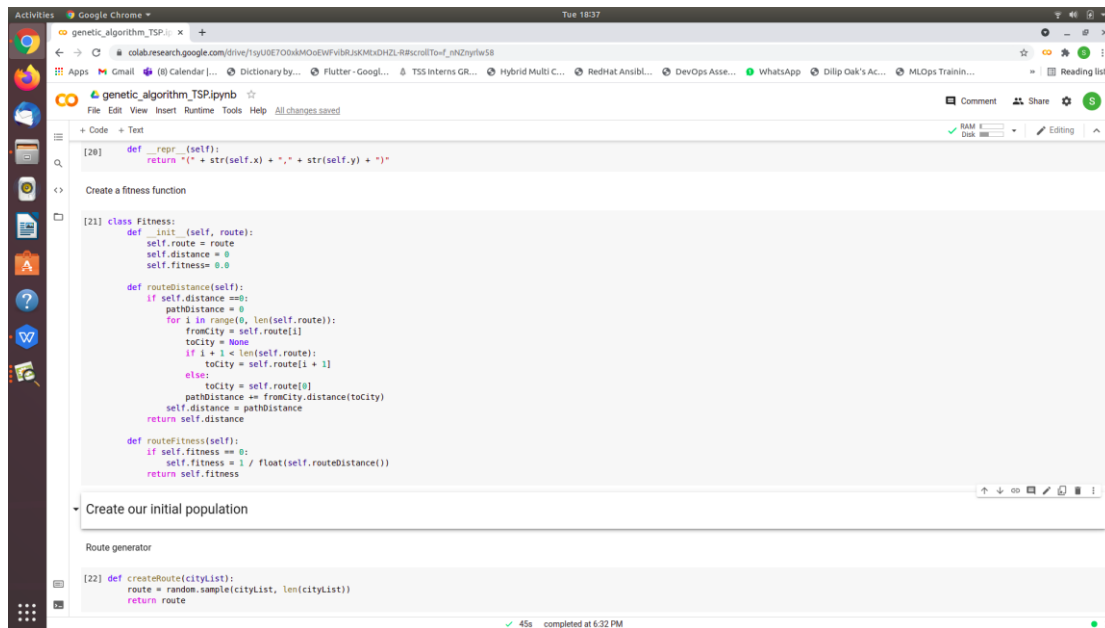
[25] def selection(popRanked, eliteSize):
    selectionResults = []
    df = pd.DataFrame(np.array(popRanked), columns=["Index", "Fitness"])
    df.sort_values("Fitness", ascending=False, inplace=True)
    popSize = len(popRanked)
    total = 0
    selected = {}
    for i in range(0, eliteSize):
        selected[i] = df.index[total]
        total += df["Fitness"].values[selected[i]]
    for i in range(0, popSize - eliteSize):
        r = random.randrange(0, df["Fitness"].values[selected[0]] - df["Fitness"].values[selected[-1]])
        selected[i + eliteSize] = df.index[(total - r) / (df["Fitness"].values[selected[0]] - df["Fitness"].values[selected[-1]])]
    return selected
```

We now can make our initial population (aka first generation). To do so, we need a way to create a function that produces routes that satisfy our conditions. To create an individual, we randomly select the order in which we visit each city:

This produces one individual, but we want a full population, so let's do that in our next function. This is as simple as looping through the `createRoute` function until we have as many routes as we want for our population.

Note: we only have to use these functions to create the initial population. Subsequent generations will be produced through breeding and mutation.

Determine fitness



```
[20] def __repr__(self):
    return "(" + str(self.x) + "," + str(self.y) + ")"

Create a fitness function

[21] class Fitness:
    def __init__(self, route):
        self.route = route
        self.distance = 0
        self.fitness = 0.0

    def routeDistance(self):
        if self.distance == 0:
            pathDistance = 0
            for i in range(0, len(self.route)):
                fromCity = self.route[i]
                toCity = None
                if i + 1 < len(self.route):
                    toCity = self.route[i + 1]
                else:
                    toCity = self.route[0]
                pathDistance += fromCity.distance(toCity)
            self.distance = pathDistance
            return self.distance

    def routeFitness(self):
        if self.fitness == 0:
            self.fitness = 1 / float(self.routeDistance())
            return self.fitness

Create our initial population

Route generator

[22] def createRoute(cityList):
    route = random.sample(cityList, len(cityList))
    return route
```

Next, the evolutionary fun begins. To simulate our “survival of the fittest”, we can make use of `Fitness` to rank each individual in the population. Our output will be an ordered list with the route IDs and each associated fitness score.

Select the mating pool

```
[26] def matingPool(population, selectionResults):
    matingpool = []
    for i in range(0, len(selectionResults)):
        index = selectionResults[i]
        matingpool.append(population[index])
    return matingpool

Create a crossover function for two parents to create one child

def breed(parent1, parent2):
    child = []
    childP1 = []
    childP2 = []

    geneA = int(random.random() * len(parent1))
    geneB = int(random.random() * len(parent1))

    startGene = min(geneA, geneB)
    endGene = max(geneA, geneB)

    for i in range(startGene, endGene):
        childP1.append(parent1[i])

    childP2 = [item for item in parent2 if item not in childP1]
    child = childP1 + childP2
    return child

Create function to run crossover over full mating pool

[28] def breedPopulation(matingpool, eliteSize):
    children = []
    length = len(matingpool) - eliteSize
    pool = random.sample(matingpool, len(matingpool))

    for i in range(0, eliteSize):
        children.append(matingpool[i])
```

There are a few options for how to select the parents that will be used to create the next generation. The most common approaches are either **fitness proportionate selection** (aka “roulette wheel selection”) or **tournament selection**:

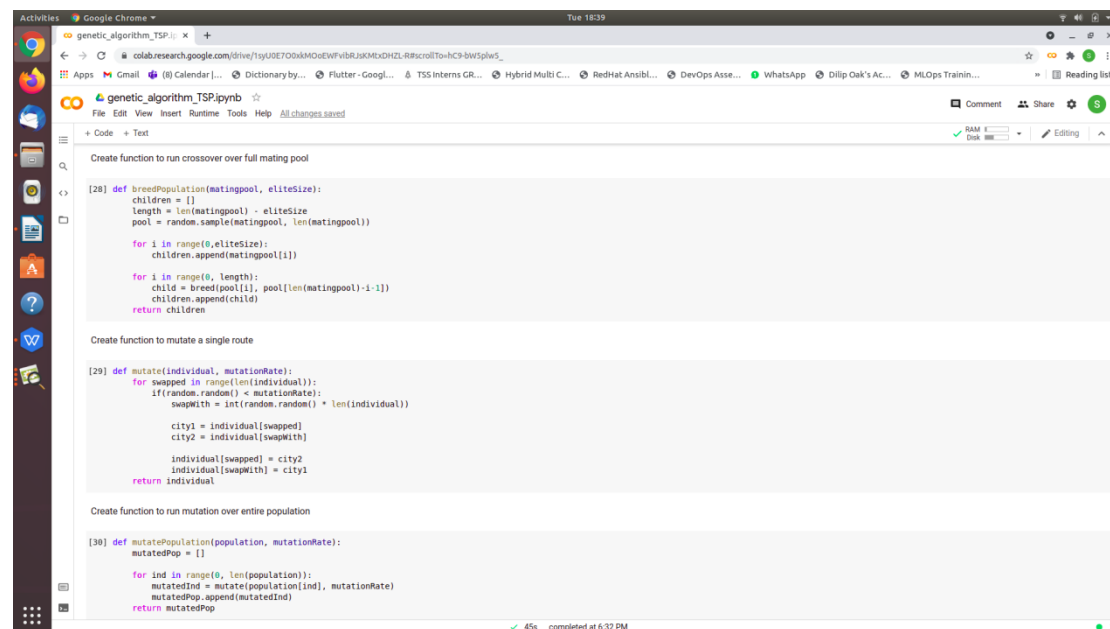
- ❖ **Fitness proportionate selection** (the version implemented below): The fitness of each individual relative to the population is used to assign a probability of selection. Think of this as the fitness-weighted probability of being selected.
- ❖ **Tournament selection**: A set number of individuals are randomly selected from the population and the one with the highest fitness in the group is chosen as the first parent. This is repeated to choose the second parent.

Another design feature to consider is the use of **elitism**. With elitism, the best performing individuals from the population will automatically carry over to the next generation, ensuring that the most successful individuals persist.

For the purpose of clarity, we'll create the mating pool in two steps. First, we'll use the output from `rankRoutes` to determine which routes to select in our `selection` function. In lines 3–5, we set up the roulette wheel by calculating a relative fitness weight for each individual. In line 9, we compare a randomly drawn number to these weights to select our mating pool. We'll also want to hold on to our best routes, so we introduce elitism in line 7. Ultimately, the `selection` function returns a list of route IDs, which we can use to create the mating pool in the `matingPool` function.

Now that we have the IDs of the routes that will make up our mating pool from the `selection` function, we can create the mating pool. We're simply extracting the selected individuals from our population.

Breed



```
[28] def breedPopulation(matingpool, eliteSize):
    children = []
    length = len(matingpool) - eliteSize
    pool = random.sample(matingpool, len(matingpool))

    for i in range(0, eliteSize):
        children.append(matingpool[i])

    for i in range(0, length):
        child = breed(pool[i], pool[len(matingpool)-i-1])
        children.append(child)
    return children

Create function to mutate a single route

[29] def mutate(individual, mutationRate):
    swapped = False
    for swapped, ind in zip(True, range(len(individual))):
        if random.random() < mutationRate:
            swapWith = int(random.random() * len(individual))

            city1 = individual[ind]
            city2 = individual[swapWith]

            individual[ind] = city2
            individual[swapWith] = city1
    return individual

Create function to run mutation over entire population

[30] def mutatePopulation(population, mutationRate):
    mutatedPop = []
    for ind in range(0, len(population)):
        mutatedInd = mutate(population[ind], mutationRate)
        mutatedPop.append(mutatedInd)
    return mutatedPop
```

With our mating pool created, we can create the next generation in a process called **crossover** (aka “breeding”). If our individuals were strings of 0s and 1s and our two rules didn’t apply (e.g., imagine we

were deciding whether or not to include a stock in a portfolio), we could simply pick a crossover point and splice the two strings together to produce an offspring.

However, the TSP is unique in that we need to include all locations exactly one time. To abide by this rule, we can use a special breeding function called **ordered crossover**. In ordered crossover, we randomly select a subset of the first parent string (see line 12 in `breed` function below) and then fill the remainder of the route with the genes from the second parent in the order in which they appear, without duplicating any genes in the selected subset from the first parent (see line 15 in `breed` function below).

Parents

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

9	8	7	6	5	4	3	2	1
---	---	---	---	---	---	---	---	---

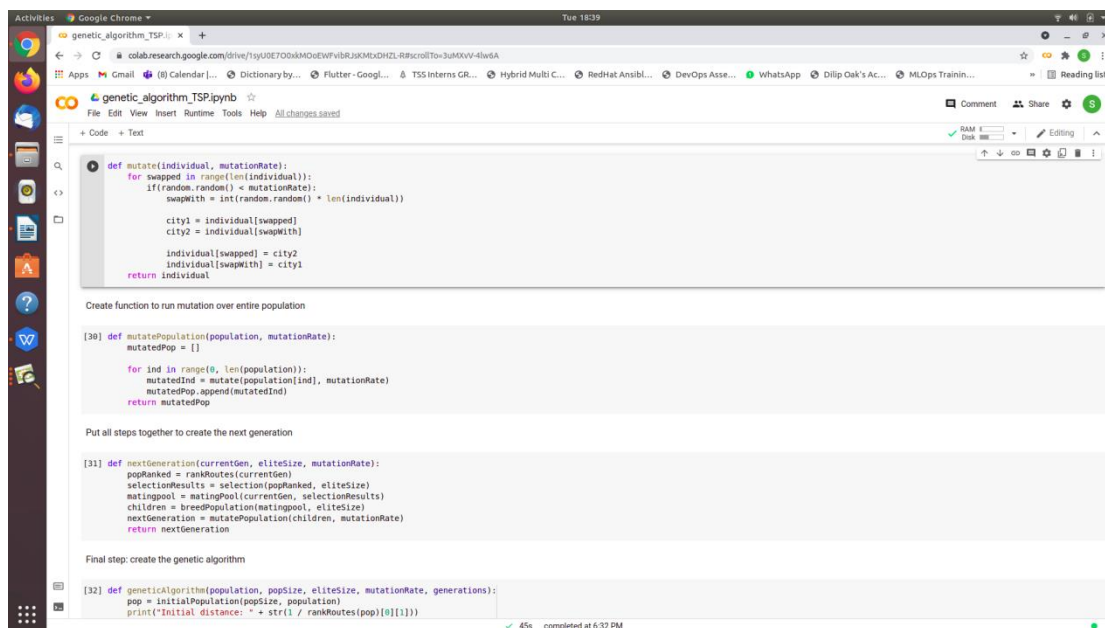
Offspring

					6	7	8	
--	--	--	--	--	---	---	---	--

9	5	4	3	2	6	7	8	1
---	---	---	---	---	---	---	---	---

Next, we'll generalize this to create our offspring population. In line 5, we use elitism to retain the best routes from the current population. Then, in line 8, we use the `breed` function to fill out the rest of the next generation.

Mutate



```
def mutate(individual, mutationRate):
    swapped = False
    for swapped in range(len(individual)):
        if(random.random() < mutationRate):
            swapWith = int(random.random() * len(individual))

            city1 = individual[swapped]
            city2 = individual[swapWith]

            individual[swapped] = city2
            individual[swapWith] = city1
    return individual

# Create function to run mutation over entire population
[30] def mutatePopulation(population, mutationRate):
    mutatedPop = []
    for ind in range(0, len(population)):
        mutatedInd = mutate(population[ind], mutationRate)
        mutatedPop.append(mutatedInd)
    return mutatedPop

# Put all steps together to create the next generation
[31] def nextGeneration(currentGen, eliteSize, mutationRate):
    popRanked = rankRoutes(currentGen)
    selectionResults = selection(popRanked, eliteSize)
    matingpool = matingPool(currentGen, selectionResults)
    children = breedPopulation(matingpool, eliteSize)
    nextGeneration = mutatePopulation(children, mutationRate)
    return nextGeneration

# Final step: create the genetic algorithm
[32] def geneticAlgorithm(population, popSize, eliteSize, mutationRate, generations):
    pop = InitialPopulation(popSize, population)
    print("Initial distance: " + str(1 / rankRoutes(pop)[0][1]))
```

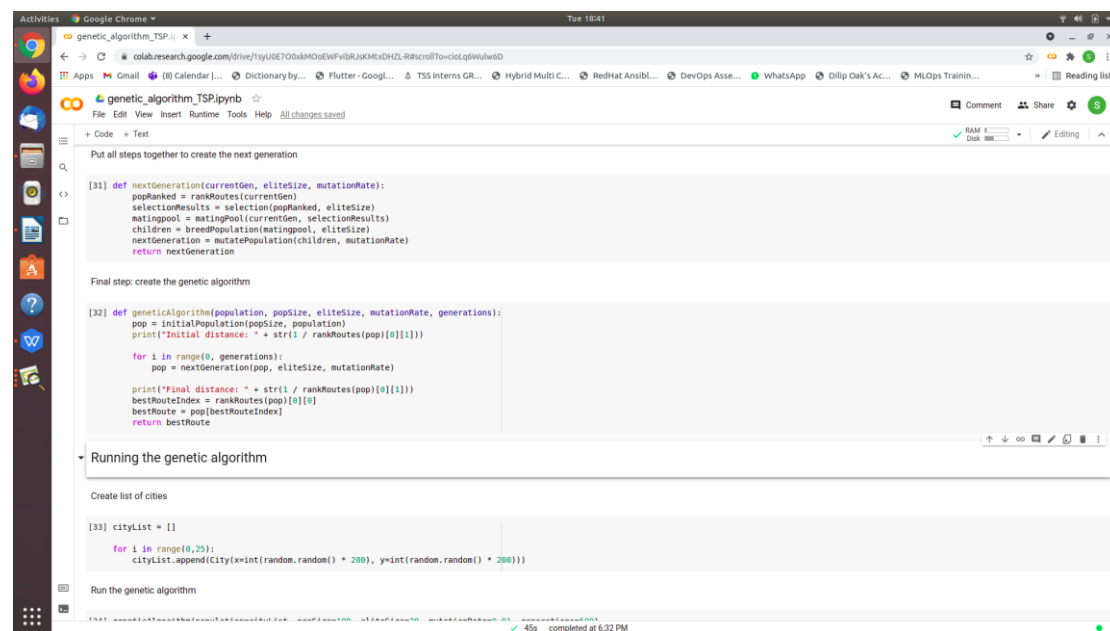
Mutation serves an important function in GA, as it helps to avoid local convergence by introducing novel routes that will allow us to explore other parts of the solution space. Similar to crossover, the TSP has a special consideration when it comes to mutation. Again, if we had a chromosome of 0s and 1s, mutation would simply mean assigning a low probability of a gene changing from 0 to 1, or vice versa (to continue the example from before, a stock that was included in the offspring portfolio is now excluded).

However, since we need to abide by our rules, we can't drop cities. Instead, we'll use **swap mutation**. This means that, with specified

low probability, two cities will swap places in our route. We'll do this for one individual in our `mutate` function:

Next, we can extend the `mutate` function to run through the new population.

Repeat



```
[31] def nextGeneration(currentGen, eliteSize, mutationRate):
    popRanked = rankRoutes(currentGen)
    selectionResults = selection(popRanked, eliteSize)
    matingpool = matingPool(currentGen, selectionResults)
    children = breedPopulation(matingpool, eliteSize)
    nextGeneration = mutatePopulation(children, mutationRate)
    return nextGeneration

Final step: create the genetic algorithm

[32] def geneticAlgorithm(population, popSize, eliteSize, mutationRate, generations):
    pop = InitialPopulation(popSize, population)
    print("Initial distance: " + str(1 / rankRoutes(pop)[0][1]))

    for i in range(0, generations):
        pop = nextGeneration(pop, eliteSize, mutationRate)

        print("Final distance: " + str(1 / rankRoutes(pop)[0][1]))
        bestRouteIndex = rankRoutes(pop)[0][0]
        bestRoute = pop[bestRouteIndex]
        return bestRoute

Running the genetic algorithm

Create list of cities

[33] cityList = []

    for i in range(0,25):
        cityList.append(City(x=int(random.random() * 200), y=int(random.random() * 200)))

Run the genetic algorithm

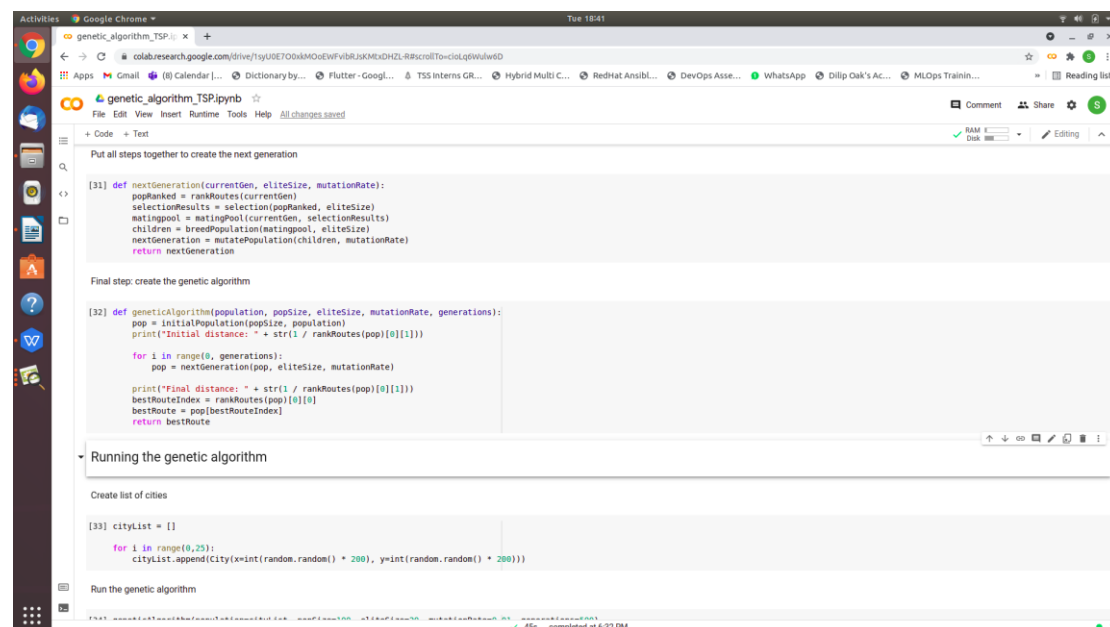
45s completed at 6:32 PM
```

We're almost there. Let's pull these pieces together to create a function that produces a new generation. First, we rank the routes in the current generation using `rankRoutes`. We then determine our potential parents by running the `selection` function, which allows us to create the mating pool using the `matingPool` function. Finally, we then create our new generation using the `breedPopulation` function and then applying mutation using the `mutatePopulation` function.

Evolution in motion

We finally have all the pieces in place to create our GA! All we need to do is create the initial population, and then we can loop through as many generations as we desire. Of course we also want to see the best route and how much we've improved, so we capture the initial distance in line 3 (remember, distance is the inverse of the fitness), the final distance in line 8, and the best route in line 9.

Running the genetic algorithm



```
[31] def nextGeneration(currentGen, eliteSize, mutationRate):
    popRanked = rankRoutes(currentGen)
    selectionResults = selection(popRanked, eliteSize)
    matingpool = matingPool(currentGen, selectionResults)
    children = breedPopulation(matingpool, eliteSize)
    nextGeneration = mutatePopulation(children, mutationRate)
    return nextGeneration

Final step: create the genetic algorithm

[32] def geneticAlgorithm(population, popSize, eliteSize, mutationRate, generations):
    pop = initialPopulation(popSize, population)
    print("Initial distance: " + str(1 / rankRoutes(pop)[0][1]))

    for i in range(0, generations):
        pop = nextGeneration(pop, eliteSize, mutationRate)

        print("Final distance: " + str(1 / rankRoutes(pop)[0][1]))
        bestRouteIndex = rankRoutes(pop)[0][0]
        bestRoute = pop[bestRouteIndex]
    return bestRoute

Running the genetic algorithm

Create list of cities

[33] cityList = []

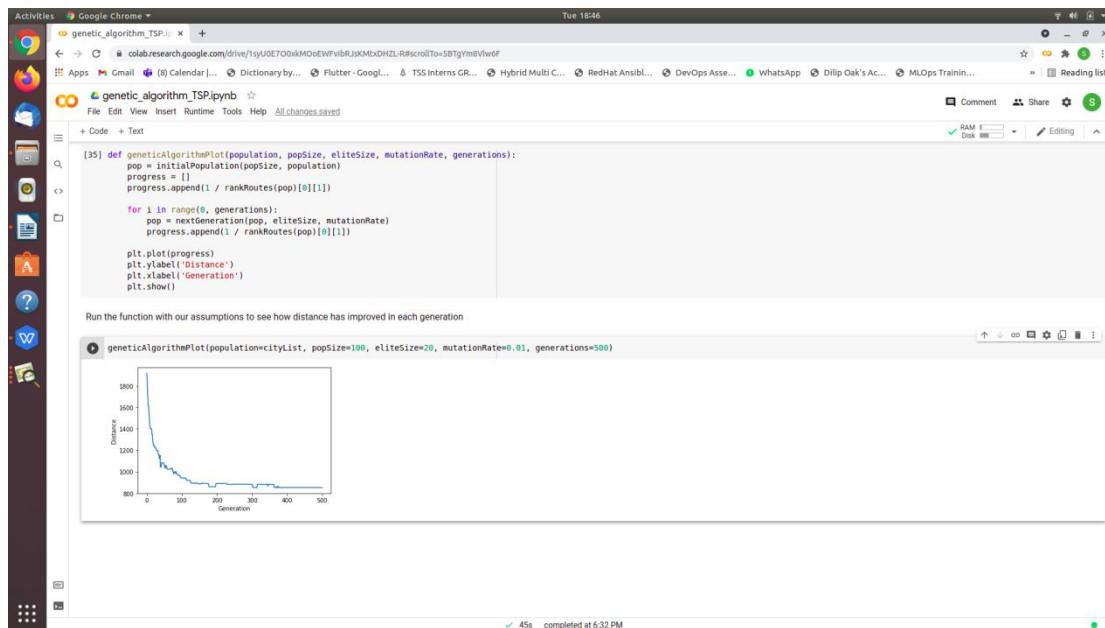
for i in range(0,25):
    cityList.append(City(x=int(random.random() * 200), y=int(random.random() * 200)))

Run the genetic algorithm
```

With everything in place, solving the TSP is as easy as two steps:

First, we need a list of cities to travel between. For this demonstration, we'll create a list of 25 random cities. Then, running the genetic algorithm is one simple line of code. This is where art meets science; you should see which assumptions work best for you. In this example, we have 100 individuals in each generation, keep 20 elite individuals, use a 1% mutation rate for a given gene, and run through 500 generations:

Plot the improvement



It's great to know our starting and ending distance and the proposed route, but we would be remiss not to see how our distance improved over time. With a simple tweak to our `geneticAlgorithm` function, we can store the shortest distance from each generation in a `progress` list and then plot the results.

Run the GA in the same way as before, but now using the newly created `geneticAlgorithmPlot` function

Conclusion

We were successful in building our own Genetic algorithm and applying it to Travelling Salesman Problem for optimization.