Optimizing Travelling Salesman Problem using Genetic Algorithm

Vimanyu Chopra
UW ID - 20889118
Email - v7chopra@uwaterloo.ca

Abstract

This study takes a look at Genetic Algorithms. The aim is to use self-evolving Genetic algorithm in order to optimize Travelling salesman problem (TSP). This is done using Python and its associated libraries. The results are then recorded and represented graphically. Performance of genetic algorithmic approach is compared to brute force method. The results are analyzed for performance and rate of improvement. Further scopeof improvement to this approach is also discussed.

I. INTRODUCTION

Travelling Salesman Problem is classified as an NP hard problem. It is one of the most renowned combinatorial problemsin computer science and mathematics. Several attempts have been made over the years to optimize it. The problem can be stated as – A traveller wants to visit n number of cities and he can only visit a city once. A further condition is that the traveller should return to the city he started from to mark the end of his journey. This seemingly simple problem can be represented in the form of a graph G with V vertices and E edges. The problemis to minimize the distance travelled by the salesperson. There are a number of variations of Travelling Salesman Problem which have been introduced and solved; this report focuses on the classic TSP.

Many optimization approaches have been used over the decades. Genetic algorithms take inspiration from Darwinian evolution which works on the principle of survival of the fittest. This ideacan be applied to solve TSP where the result constantly evolves, akin to the species in a population in order to reach an optimal result. In order to achieve the desired results, techniques like crossover, mutation and selection are employed.

TSP being a foundational problem in mathematics and computerscience has a wide range of real-world application potential in a variety of fields such as supply chain, electronics, and medical field. Optimizing TSP is beneficial to logistic companies mainly concerning last mile delivery which is the most expensive part of a supply chain. DNA sequencing can also be optimized and improved with better TS solutions, similar to how circuit board designs can become more efficient with shorter paths between circuits.

Genetic Algorithms are optimization algorithms and are a part of evolutionary computation**.** [1]. These algorithms draw inspiration from nature and the concept of Evolution presentedby Charles Darwin. Darwinian evolution theory states the survival of the fittest and Genetic Algorithms based on the principle of natural evolution. These algorithms try to imitate biological methods such as reproduction, selection and evolution in orderto optimize the results. The optimized solution is obtained after the solution goes through evolution over a number of iterations. Theresult keeps improving as the population evolves to find the "fittest" solution which is based on the fitness. In case of Travelling Salesman Problem, fitness function is the distance required to be travelled to meet the requirements.

Some terms associated with genetic algorithms are –

- Fitness function – A function defined in order to rank the individuals in a population
- Selection – Process of selecting best performing individuals
- Crossover – Process of recombining parents to produce offspring
- Mutation – Introduced to improve diversity among the population

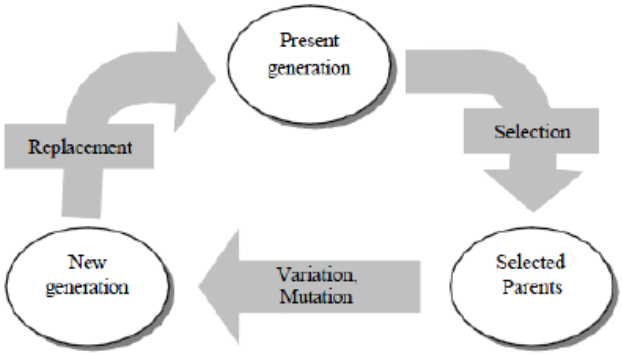


Figure 1 (Silva, Analide, 2021)

II. BACKGROUND

A. Introduction of TSP

Travelling Salesman Problem is an Np hard problem. It has (n-1) factorial possible solutions for n number of cities. This makes the problem exponentially more difficult computationally with increasing number of cities. For this project having 25 cities, the possible solutions would be 6.24*10^23. Thus, optimizing this solution is very important. Over the decades several attempts have been made to find a suitable solution.

B. History of TSP

The exact origin of Travelling Salesman Problem is not recorder. The idea of traveller came in 1932 from a book Der Handlungsreisende which explored the idea of optimizing paths travelled [3]. The earliest record of such a problem is 1932 when the mathematician Karl Menger mentioned the "Messenger Problem" in his work. The basic ideas of Messenger problem are similar to that to the classic TSP [4]. Karl sought the idea to solve this problem using brute force.

The problem became well known from 1940s with companies like RAND corporation taking interest and many attempts being made to solve it. In 1956 Flood made the connection between TSP and Hamiltonian cycles [5].

The NP hardness of this problem was proven by Richard M. Karp in 1972 thus providing explanation for its computational complexity [6]. By the 1980s TSP was solved upto 2392 cities was solved by Grötschel, Padberg, Rinaldi.

Up until recently that largest travelling salesman problem that was successfully solved was by William Cook with 85,900 cities taking microchip layout as the basis [7]. A new breakthrough was made in 2020 when Nathan Kelin offered a slightly improved performance of the metric Travelling Salesman Problem which has led to a newfound interest in the community with new advancement being made after many years [8]. This was also one of the inspirations for this project.

C. Genetic Algorithms

Genetic Algorithms being a part of evolutionary computation are inspired by biological processes and are based on Darwinian Evolution. This makes them highly adaptable as with every iteration, these algorithms improve the performance leading to optimized results.

The first genetic algorithm software was written by Nils Barricelli in 1954 while trying to produce artificial life. The version of genetic algorithm widely known with genetic operators was introduced by John Holland in 1970s and this was a huge landmark in the field [9].

Over the years, many new approaches to crossovers, mutations and fitness functions have been developed. In recent times Genetic Algorithms have been used with Neural Networks for improved efficiency and accuracy [10]

D. Real world Applications of GA

Due to being powerful, yet simplistic in nature, Genetic Algorithms have become one of the most widely used strategies in artificial intelligence problems and this has led to them being used in various real world applications**.** Some of the applications of Genetic algorithms are –

- Robotics – Genetic algorithms can be used to design therobots and also to help in their navigation. By using genetic programming approach with navigation problems such as object detection and routing, the performance of the agent can be increased.
- Data Encryption – Genetic algorithms can be used to encrypt or decrypt the data. As the solutions are self-evolving, it is possible to create new methods by increasing complexity making genetic approach an exciting prospect for the world of cryptography [2].
- Design Engineering – GA can be used in conjunction with computer design applications in order to create and improve engineering models. These can be used to select materials and shape of the final product while also pointing out the defects present in a model for further improvement [11].

Apart from these GA is currently being used or can be employed in telecom, finance, gaming and automotive industries among other technological fields.

This inspired the formulation of this project as this part of artificial life (evolutionary computation and Genetic Algorithms) is being utilized in many different fields and TSP has real world significance.

III. METHODOLOGY

This project is in Python which is a high-level interpreted programming Language. Libraries used are NumPy, pandas for managing the data, random for generating random paths, math for calculations and Matplotlib for plotting the results and representing them graphically. As the genetic algorithm takes inspiration from biological processes, some of the terminology used in genetic algorithms is inspired by biology-

1. A gene represents each which in case of TSP is a city.
2. A chromosome or individual is a possible solution which in case of TSP is a possible route.
3. Population is the set of all individuals.
4. Parents are the chromosomes chosen to produce solutions.
5. Fitness is the parameter which determines the performance of each possible [12].

Below are the steps followed for the project -

A. Initializing population

The population used for calculating distance is initialized. This includes the number of cities and their coordinates. The numbers of cities taken for this project is 25 and are numbered 0 to 24. The x and y coordinates for every city in the list is randomly initialized in a custom range which in this project is the range of 0 to 2400. The city number along with the values of x and y coordinates are placed as a tuple in a list of tuples named towns and it is the population used for Travelling Salesman Problem.

B. Brute Force Method

Brute force method as the name suggests take a random pathin order to solve TSP. An arbitrary path is chosen and the distance travelled is calculated. For this project, the list of tuples named towns is sent to the function generatePath which randomly rearranges the list and form a route called as random_path. The list random path is then used to calculate the distance using the function getDistance which calculates the Euclidian distancebetween two cities is used. This function first calculates the distance between the 25 cities and then adds the distance to return to the first city.

The result of the path travelled is then plotted using plotTowns and plotPaths functions. These plotting functions plot the cities and the corresponding routes travelled between them respectively. The path between 25 cities is plotted and thenthe path from last to the first city is plotted to complete the cycle. This graph contains the position of all cities and the route taken by the traveller according to the randomized list random_path. This serves as a reference and the basis of comparison when the genetic algorithmis applied.

C. Applying Genetic Algorithm

For applying Genetic Algorithm to Travelling Salesman Problem the following algorithm is followed –

Algorithm 1: Steps to apply genetic algorithm

1 Initializing Population

2 Initializing Hyperparameters

3 Determine the fitness function

4 Selection

5 Crossover

6 Mutation

7 Replace the population with evolved population

8 Repeat steps 4-7 till condition is satisfied

D. Initializing Hyperparameters

The hyperparameters which defined for this project are –

1. Population_number – These are the number of possible solutions calculated
2. Iterations – The number of iterations performed on the population
3. Elite – It is the elitism factor used to ensure best performing results are chosen
4. mutation_factor – It is the rate of mutation
5. crossover_factor – It is the rate at which crossover is performed

These parameters control the outcome of the genetic approach and can be varied in order to find the optimal path.

The path taken and the total distance travelled by the brute force method is used for genetic algorithm. The objective is to improve on its performance by decreasing the total distance travelled. A DataFrame species is initialized with the route being one column and fitness of each candidate solution occupying another column. Population_number which in case of this project is 100 is used as a limit for number of solutions. The three genetic operators – Selection, Crossover and mutation are repeatedly executed over number of iterations specified in hyperparameters (which in the project is 500) in order to evolve the population of solutions in order to obtain the optimized result [1].

E. Selection

Selection is the process of choosing solutions which will beused for generating offspring. Fitness function is used to determine the likelihood of each "chromosome" or route solution in case of TSP being selected. A fitness function defines how good a candidate in the population is. Fitness function is chosen on the basis of what needs to be optimized.

In this project, the fitness function is taken as the inverse of total distance travelled by the salesman.The candidate solutions are ranked according to fitness

$$f(x) = \frac{1}{Distance\ Trvelled} \qquad (1)$$

There are a few ways in which the selection is performed-

1. Roulette Wheel Selection – Also known as fitness proportionate selection, roulette wheel selection uses the fitnessof each chromosome to determine the probability of the candidate being selected for the mating pool. This means that the probability of an individual being selected is has linear relationship to its fitness.

$$p(a) = \frac{f(a)}{\sum_{i=0}^{n} f(i)} \qquad (2)$$

Here probability of each fitness function value is calculatedby dividing it with sum of all fitness values and is then used to select mates for selection.

This technique is employed in the project.

2. Tournament Selection – This method n number of candidates are chosen and compete against each other and the individual with the higher fitness value moves to the next round.A series of rounds take place in order to select individuals.

3. Rank Method – Rank selection is a method in which the chromosomes are ranked on the basis of their fitness values andthis rank then determines the selection process. It is beneficial touse rank selection for population having close margins or negative values [10].

In order to ensure that a few fittest possible routes are alwayschosen elitism is applied in the project which ensures that certain individuals (in the project 20) having the highest fitness functionare always chosen for the evolution process. In the project of the hundred population size, the chosen number of individuals ranked the highest are always chosen to be a part of the genetic algorithm process in order to get the most optimized results.

For the project, the population is ranked according to fitness values in descending order so that the best performing candidates occupy the initial positions in the species dataframe.SelectAncestor function is called which is used to apply the fitness proportionate method by calculating probability of eachindividual being selected which is calculated by dividing each individuals' fitness with the sum of all fitness values. These probability values are then used as a parameter to select parents for the mating pool by applying random choice function over this probability distribution. This ensures that the higher probability values havea higher chance of being chosen while also giving the lower fitness values an opportunity to be selected. The parents selected according to this probability are then over to crossover genetic operator is then applied.

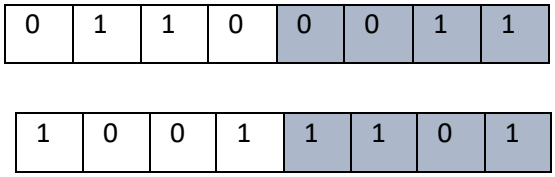
F. Crossover

Crossover is the process by which the parent individuals selected are recombined to from an offspring for the evolved population. In crossover genetic operator, the parent chromosomes are recombined by choosing a random position in both the strings and then changing the sequence before and after that chosen position with the other parent chosen. This leads tothe creation of two offspring called "children".
The aim of crossover is to combine the genetic information of the two chosen individuals in order to form children much like biological processes. The children have parts of genes from both the parents while being a completely different entity.

There are many ways in which crossover can be implemented –

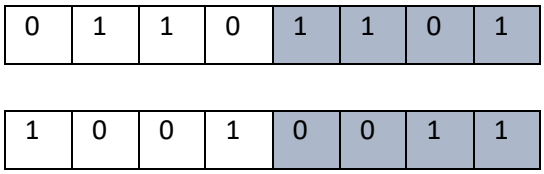
1. One Point Crossover – In this method, a crossover point is chosen and the sequence from that point is swapped to produce offspring.

For example -The parents with crossover point at cell 4



(a)

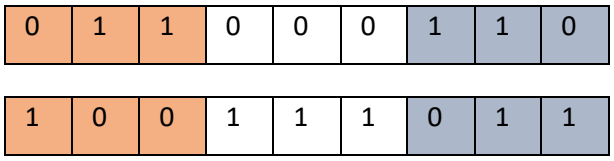
Upon crossover produce the following result



(b)

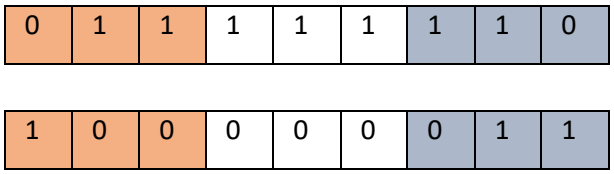
Figure 2 (a) Parent gene (b) Results after crossover

2. Two point Crossover - In two-point crossover two crossoverpoints are selected instead of one and the genetic material which in this case are the bits are exchanged. For example – the parent strings having crossover points onlocation 3 and 6.



(a)

Produces the following offspring –



(b)

Figure 3 (a) Parent gene (b) Results after crossover

N-point crossovers can also be implemented in a similar way.

3. Ordered Crossover – In this method, a subset from gene1 is chosen randomly for the resulting offspring and then the remaining positions are filled by parts of gene2 in the order which they appear. Checking of duplicates is also performed [13]. The crossover in the project is inspired from this approach. For example -

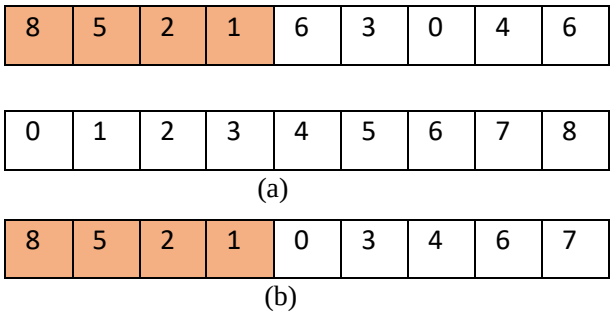


Figure: 4 (a) Parent gene (b) Result after ordered crossover

As Travelling Salesman Problem requires uniqueness in cities travelled, therefore in this project ordered crossover is employed. In order to form new child chromosomes, random selection from first parent is taken and then is added to first part of chromosome while the rest of the chromosome is formed withthe cities from second parent chromosome while taking into account that the city is not already present. Both these parts are added to form child chromosome.

The child gene from the parents in the program is formed by first taking a random subset of ancestor1 called offspring1 and then filling the rest of the chromosome with genes from ancestor2 in the given order while checking for duplicity of cities with offspring1, producing offspringP2. Offspring1 and offspringP2 are then appended to produce the recombined result offspring. In order to model this genetic approach according to biological evolution, the rate of crossover is chosen as a much higher value (in this project 95 percent) as compared to mutation rate in order to replicate the percentage in natural evolution.

G. Mutation

Mutation is the process of introducing a small change in thepopulation. It is performed to increase diversity in the candidate population. Just like in natural environment mutation helps the species evolve by introducing new characteristic in the species, similarly mutations help improve the optimized solution in GA. There are a few ways of introducing mutation. In binary strings, mutation can be introduced by flipping the bit value or swapping two random chromosomes.

1. Bit flip mutation –

A random bit is chosen and its value is flipped or inversed to give produce a new mutant gene.

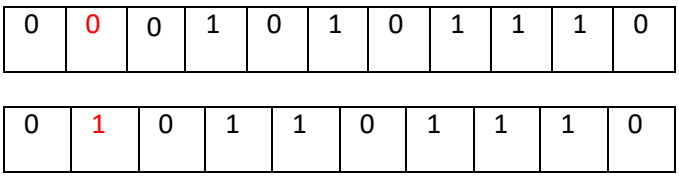


Figure:5 (a) Parent gene (b) Result after bitflip mutation

2. Swap mutation –

Two random chromosomes are swapped.

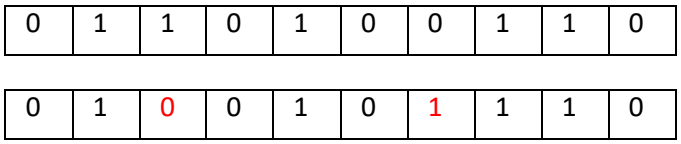


Figure:6 Gene before and after swap mutation

A mutation factor is chosen to determine the rate of mutation in the population. For this project, a mutation rate of 10 percent is chosen [1]. This factor determines the percentage of population which will be mutated throughout each iteration. As bit-flip mutation like binary mutation cannot be employed for travelling salesman problem, two individuals are randomly chosen and swapped in order to introduce mutants in the evolved population. This is done with the help of a mutate function which used NumPy random functionality in order to choose the random positions to be swapped. Without mutation, there is a risk of the evolution of the population becoming stagnant. Therefore,mutation in GA is important as helps avoid premature convergence to local minima and also aids the exploration of routes apart from the explored ones, aiding the discovery of solutions which may not be discovered using traditional methods.

The evolved population obtained after applying genetic operators- selection, crossover and mutation is then iterated over all these processes till the termination condition which is the required number of iterations specified is reached.

H. Displaying results

After the number of iterations has been reached, a dataframe result is formed and is passed to the calculateDistance function which calculates Euclidian distance between cities and returns the sum of distances between the cities and provides the total distance travelled by the salesman by adding the last city to the route.

Optimal path is displayed with the help of list optimal which contains the path travelled from the first city while traversing all cities and then back to the initial one. The route taken by the traveller in accordance to the Genetic algorithm is also plotted by using the plotTowns and plotPath functions. These functions output a graph which represents the route followed from the initial city and back to it after visiting all the towns once.

The distance travelled by the traveller over the iterations are also stored in a list named lis and it is then used to plot a graph which represents the progress of the algorithm with increasing number of iterations.

In order to analyze the results, the simulations are run a number of times. The parameters such as mutation rate and number of cities are varied to study how they affect the behaviour of the genetic algorithm and these results are recorded.

IV. RESULTS

The questions that need to be answered by the simulations run are –

- Does the genetic algorithm approach improve the performance of travelling salesman problem?
- If the performance is improved, to what degree?
- What affect does varying the parameters have?
- Does this approach work for different scenarios?

In order to answer these questions, results are analyzed. Graphs indicating cities according to their coordinates and the path taken to travel between them are used. For testing the following values are chosen –

Number of cities - 25 and Range of coordinates – 2400

Initializing the population results in the generation of a list with city numbers and x, y coordinates. When brute force approach is applied to these towns, it results in the formation of a path. For these parameters over multiple runs, the total distance travelled generally ranges between 24000 and 38000. It was observed that the random path never chose the optimized route during the testing simulations. The table 1 gives the series of recordings over multiple runs.

S.no	No. of towns	Initial Distance	Optimized Distance	Improvement Percentage
1	**25**	**32754.02**	**11217.11**	**65.75**
2	**25**	**34570.13**	**10615.41**	**69.29**
3	**30**	**37314.06**	**12762.50**	**65.79**
4	**30**	**34209.73**	**12667.53**	**62.97**

Table 1 Recording of some sample runs of the project indicating the ditance travelled in each run.

```
Plaease enter number of towns 25
Please enter the range for x and y coordinates 2400

Initial path distance is: 33143.773569378136
Random path chosen is
7
6
9
10
4
2
0
17
24
14
8
19
5
16
18
20
12
11
22
23
21
15
1
3
13
7
```

(a)

```
Initial route distance is: 33143.773569378136
Optimized path distance is: 11060.267385899417
Optimal Path visited is
8
1
23
21
14
11
20
7
24
9
16
0
12
5
19
6
10
17
22
3
18
15
4
2
13
8
```

(b)

Figure:7 (a) Output of initial distance travelled

(b) Results after applying GA

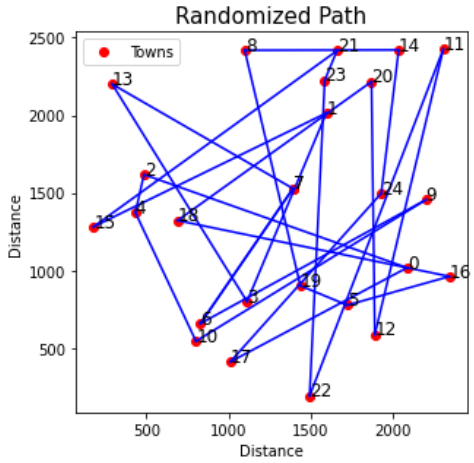


Figure 8– Graph showing the cities and random path chosen by brute force method

When genetic algorithm is applied to this initial total distance and path by using genetic operators – crossover, mutation and selection it results in a new path being chosen. From the results recorded and Fig 7 (a) and (b), it is observed that the total distance travelled was significantly reduced as compared to the previous approach.

From the multiple runs in Table 1, the improvement recorded is in the range of 65 percent meaning that the genetic algorithm was able to reduce the distance to one third the original value.

Increasing the number of cities to 100 or the range beyond 4000 did not affect the performance of the project apart from the increased computation time. The graph obtained after applying optimizations looks much cleaner than the random path travelled indicating improvement through visual medium.

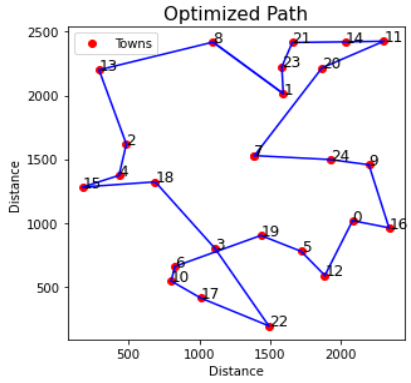


Figure 9– Graph showing the cities and optimized path chosen by applying the Genetic Algorithm

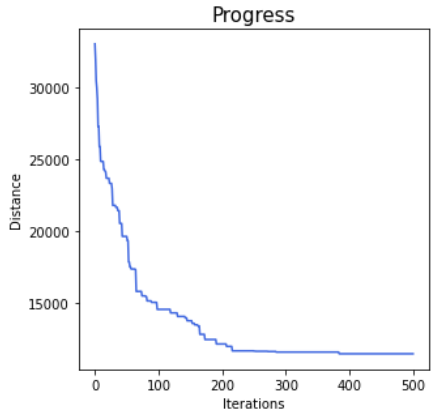


Figure 10– Graph showing the gradual reduction in distance travelled with the increasing number of iterations ranging from 0 to 500

S.no	Mutation Factor	Random Distance	Optimized Distance
1	0.1	31410.027	10676.414
2	0.3	31410.027	11587.656
3	0.5	31410.027	12004.788
4	0.7	31410.027	13386.24

Table 2 Studying Mutation Factor

The mutation probability thus is optimal for values around 10 and 20 percent for size of population 100. Increasing mutation factor too much impacts the result obtained as seen in Table 2. If mutation is not applied it results in the premature convergence to local minima. When mutation is applied, it can be seen in the progress graph (figure 10) that there is some change in the distance travelled around the 370 iterations mark due to mutation continuing to explore new paths and hence improving performance. Eliminating mutation results would result in premature convergence to a minima and these improvements in the later number of runs being lost. Total number of iterations were chosen to be 500 for this configuration as the improvement in performance after these are minimal.

Thus, the results obtained are quite favorable and the genetic algorithm accomplished its goal of optimizing Travelling Salesman Problem.

V. FUTURE IMPROVEMENTS

The results obtained from this project are quite promising with noticeably improvement over random approach. Still, there are a few limitations in this approach. The number of cities chosen during testing ranged from 25 to 100. If the number of cities is increased further to thousands or the number of iterations is increased considerably then the processing time taken increases drastically.

Also, the hyperparameters are chosen based on literature reviews and common knowledge. For different sizes and conditions, the parameters should be adjusted accordingly for the best possible results.

Some suggestions for future improvements include the optimization of running time and improved crossover or selection approaches. Multiple selection and crossover approaches can be run simultaneously to choose the best performing one in that scenario. Neural networks can be used to improve performance. Different machine learning approaches like clustering can also be implemented alongside genetic algorithms to form cluster of best cities to optimize Travelling Salesman Problem even further.

Additionally, a feature can be introduced with the help of which hyperparameters can be dynamically chosen based on number of cities and range of coordinates. Genetic Algorithms can be applied to choose these parameters as well to select the best possible ones. As Travelling salesman is an NP hard problem, a large number of improvements can be made in order to optimize the results further and further to improve its performance to get it as close to optimal path as possible.

VI. CONCLUSION

This project is aimed at providing a simple Genetic Algorithmic approach to find optimized solution for travelling Salesman Problem. The test simulations were based on scenarios having different cities and ranges. The results recorded showed a considerable improvement over the brute force approach. The path obtained while not the best possible quite closely resembled the optimal path solution. The hyperparameters such as crossover and mutation factors were chosen to follow the patterns of biological processes in evolution. The progress made during its runs showed that it improves with every iteration by using historical data and keeps improving without converging to local minima. Travelling salesman problem has a plethora of real-world scenarios in various fields like telecommunication, logistics, electronics among others. Genetic approach is one of the candidates

for this solution and it performs very well to solve combinatorial optimization problem like TSP without being exceedingly computationally expensive which makes it an ideal prospect and this project can be considered a small step towards completely optimizing TSP.

VII. REFERENCES

[1] Tabassum, Mujahid & Mathew, Kuruvilla. (2014). A Genetic Algorithm Analysis towards Optimization solutions. International Journal of Digital Information and Wireless Communications (. 4. 124-142. 10.17781/P001091

[2] Real World Uses of Genetic Algorithms", brainz learn something, http://brainz.org/15-real-world-applications-genetic- algorithms/

[3] Cook, William J. In Pursuit of the Traveling Salesman: Mathematics at the Limit of Computation. Princeton, NJ: Princeton UP, 2012. Print

[4] https://www.theorsociety.com/about-or/or-methods/heuristics/a-brief-history-of-the-travelling-salesman-problem/

[5] Alexander Schrijver, On the History of Combionatorial Optimization (till 1960), Handbooks in Operations Research and Management, Volume 12, 2005, Pp 1-68.

[6] Brucato, Corinne (2013) The Traveling Salesman Problem. Master's Thesis, University of Pittsburgh.

[7] Cook, W. "In pursuit of the traveling salesman: mathematics at the limits of computation. 2012."

[8] Anna R. Karlin and Nathan Klein and Shayan Oveis Gharan, "A (Slightly) Improved Approximation Algorithm for Metric TSP", 2021. arXiv:2007.01409

[9] Carr, J.. "An Introduction to Genetic Algorithms." (2014).

[10] L. Haldurai, T. Madhubala, R. Rajalakshmi "A Study on Genetic Algorithm and its Applications." *International Journal of Computer Sciences and Engineering* 4.10 (2016): 139-143.

[11] Fang, Xiaopeng, "Engineering design using genetic algorithms" (2007). *Retrospective Theses and Dissertations*. 15943.

[12] L. Haldurai, T. Madhubala, R. Rajalakshmi, "A Study on Genetic Algorithm and its Applications," *International Journal of Computer Sciences and Engineering*, Vol.4, Issue.10, pp.139-143, 2016.

[13] https://www.rubicite.com/Tutorials/GeneticAlgorithms/CrossoverOperators/Order1CrossoverOperator.aspx

APPENDIX 1 – Code for the project

```
# Optimizing TSP with GA


# Initializing Libraries
import warnings
warnings.filterwarnings('ignore')
import numpy as np
import random as r
import matplotlib.pyplot as plt
import math
import pandas as pd
kgf=0
def main():
    #Initializing Hyperparameters
    plt.figure(figsize=(12,12))
    plt.subplots_adjust(wspace=0.5, hspace=0.5)
    Population_number = 100
    Iterations = 500
    Elite = 20
    mutation_factor = 0.1
    crossover_factor = 0.95


    # Mutation function
    def mutation(individual):


        position_1 = r.randint(0, len(individual)-1)
        position_2 = r.randint(0, len(individual)-1)


        mutant = individual.copy()
```

```
    mutant[position_1] = individual[position_2]
    mutant[position_2] = individual[position_1]

    return mutant

# Town plot function to plot cities
def plotTowns():
    global kgf
    l=0
    while(l<len(towns)):
        if kgf==0:

            plt.subplot(2, 2, 1)
            plt.title("Randomized Path",fontsize=15)
            plt.xlabel("Distance")
            plt.ylabel("Distance")
            plt.legend(["Towns"])
            plt.plot(towns[l][1], towns[l][2], 'bo',color="red")
            plt.annotate(towns[l][0], (towns[l][1], towns[l][2]),fontsize=12)
        else:
            plt.subplot(2, 2, 2)
            plt.title("Optimized Path",fontsize=15)
            plt.xlabel("Distance")
            plt.ylabel("Distance")
            plt.legend(["Towns"])
            plt.plot(towns[l][1], towns[l][2], 'bo',color="red")
            plt.annotate(towns[l][0], (towns[l][1], towns[l][2]),fontsize=12)

        l=l+1
    kgf=kgf+1
```

```
# Plotting path function to plot routes
def plotPath(path_taken):
    global kgf
    for i in range(len(path_taken)):
        if i+1==len(path_taken):
            x_values = [path_taken[i][1], path_taken[1][1]]
            y_values = [path_taken[i][2], path_taken[1][2]]
            if kgf==1:
                plt.subplot(2, 2, 1)
                plt.plot(x_values, y_values,color="blue")
            else:
                plt.subplot(2, 2, 2)
                plt.plot(x_values, y_values,color="blue")

        else:
            x_values = [path_taken[i][1], path_taken[i+1][1]]
            y_values = [path_taken[i][2], path_taken[i+1][2]]
            if kgf==1:
                plt.subplot(2, 2, 1)
                plt.plot(x_values, y_values,color="blue")
            else:
                plt.subplot(2, 2, 2)
                plt.plot(x_values, y_values,color="blue")
            #plt.plot(x_values, y_values,color="blue")

# Path generator (random)
def generatePath():
    path_taken = towns
    r.shuffle(path_taken)
    return path_taken
```

```
    # Euclidian distance calculator function using formula sqrt((x1-x2)^2+ (y1-y2)^2)

    def calculateDistance(path_taken):


        distance = 0

        for i in range(0,len(path_taken)):

            if i+1==len(path_taken):

                distance += math.sqrt(((path_taken[i][1]-path_taken[0][1])**2)+((path_taken[i][2]-
path_taken[0][2])**2))

            else:

                distance += math.sqrt(((path_taken[i][1]-path_taken[i+1][1])**2)+((path_taken[i][2]-
path_taken[i+1][2])**2))



        return distance


    #fitness function

    def computeFitness(path_taken):

        return 1/float(calculateDistance(path_taken))


    #crossover function

    def crossover(ancestor1, ancestor2):

        offspring = []

        offspringP1 = []

        offspringP2 = []


        geneticA = int(r.random() * len(ancestor1))

        geneticB = int(r.random() * len(ancestor1))


        initialGene = min(geneticA, geneticB)

        finalGene = max(geneticA, geneticB)

        while(initialGene<finalGene):
```

```
        offspringP1.append(ancestor1[initialGene])
        initialGene=initialGene+1

    offspringP2 = [item for item in ancestor2 if item not in offspringP1]

    offspring = offspringP1 + offspringP2
    offspring
    return offspring



#selection function
def selectAncestors(species):
    prob_dist = []
    fit_sum = sum(species['fitness'])
    for i in species['fitness']:
        prob_dist.append(i/fit_sum)

    done = False
    p1_idx = 0
    p2_idx = 0
    while not done:
        if p1_idx==p2_idx:
            p1_idx = np.random.choice(np.arange(0, 100), p=prob_dist)
            p2_idx = np.random.choice(np.arange(0, 100), p=prob_dist)
        else:
            done = True

    ancestor1 = species.iloc[p1_idx]['results']
    ancestor2 = species.iloc[p2_idx]['results']
```

```
    return ancestor1, ancestor2


# initializing program


n_towns=input("Plaease enter number of towns ")

xy_range = input("Please enter the range for x and y coordinates ")

n_towns=int(n_towns)

xy_range= int(xy_range)

#n_towns = 25

#xy_range = 2400

towns = []


for t in range(n_towns):

    x = r.randint(0, xy_range)

    y = r.randint(0, xy_range)

    towns.append((t,x, y))



random_path = generatePath()

#constant route to check hyperparameters


#towns= [(0, 529, 1071), (1, 2396, 1224), (2, 2189, 705), (3, 779, 840), (4, 1207, 211), (5, 182, 1118), (6, 1622, 1689), (7, 2050, 869), (8, 1851, 646), (9, 2175, 1409), (10, 420, 1443), (11, 453, 1518), (12, 466, 2436), (13, 1867, 174), (14, 963, 928), (15, 2290, 2145), (16, 2144, 1755), (17, 2388, 2179), (18, 3, 1137), (19, 647, 1886), (20, 1317, 2197), (21, 315, 335), (22, 1696, 188), (23, 1292, 1643), (24, 71, 710)]   #baseline for our model


#random_path=[(17, 2388, 2179), (22, 1696, 188), (4, 1207, 211), (20, 1317, 2197), (18, 3, 1137), (0, 529, 1071), (3, 779, 840), (2, 2189, 705), (10, 420, 1443), (11, 453, 1518), (6, 1622, 1689), (19, 647, 1886), (16, 2144, 1755), (23, 1292, 1643), (15, 2290, 2145), (24, 71, 710), (8, 1851, 646), (7, 2050, 869), (14, 963, 928), (21, 315, 335), (5, 182, 1118), (13, 1867, 174), (9, 2175, 1409), (1, 2396, 1224), (12, 466, 2436)]
```

```
print('Initial path distance is: '+str(calculateDistance(random_path)))


chosen_path=[]


for i in random_path:
    chosen_path.append(i[0])
chosen_path.append(chosen_path[0])


print("Random path chosen is ")
for i in chosen_path:
    print(i)


# printing result and random path
addfirst=[]
addfirst=random_path
addfirst.append(addfirst[0])


plotTowns()
plotPath(addfirst)




# GA Approach
# Initializing population


species = pd.DataFrame({'results':[], 'fitness': []})
for _ in range(Population_number):


    species = species.append({'results': random_path, 'fitness': computeFitness(random_path)},
ignore_index=True)
```

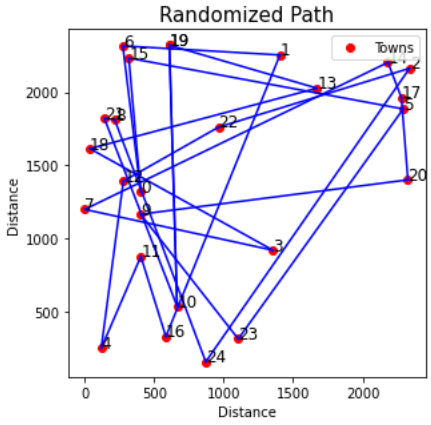
```
species = species.sort_values('fitness', ascending=False)

lis=[]

check = species.iloc[0]['results']
print('Initial route distance is: '+str(calculateDistance(check)))
lis.append(calculateDistance(check))



# loop for iterations
for _ in range(Iterations):
    evolution_species = []

    #perform elitist method, selection, crossover and mutation
    for i in range(Elite):

        evolution_species.append(species.iloc[i]['results'])

    while len(evolution_species)<Population_number:

        ancestor1, ancestor2 = selectAncestors(species)

        #crossover and mutation

        if r.random() <= crossover_factor:
            evolution_species.append(crossover(ancestor1, ancestor2))

        if r.random() <= mutation_factor:
```

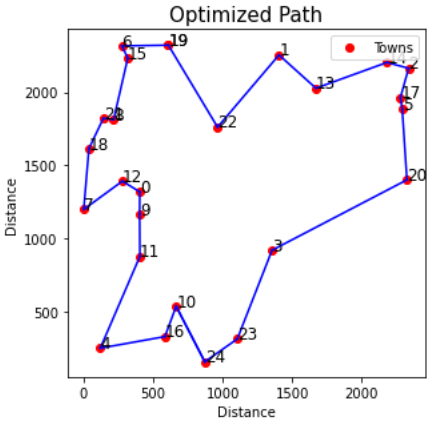
```
            evolution_species= mutation(evolution_species)


    #redifiining species
    species['results']=evolution_species
    for index, row in species.iterrows():
        species.at[index, 'fitness'] = computeFitness(row['results'])

    species = species.sort_values('fitness', ascending=False)
    newsol = species.iloc[0]['results']
    hff=calculateDistance(newsol)
    lis.append(hff)


species = species.sort_values('fitness', ascending=False)
result = species.iloc[0]['results']


print('Optimized path distance is: '+str(calculateDistance(result)))

# adding first city to the list and printing optimized route
#print(result)
optimal=[]
#print("i is")
for i in result:
    optimal.append(i[0])
optimal.append(optimal[0])

print("Optimal Path visited is")
```

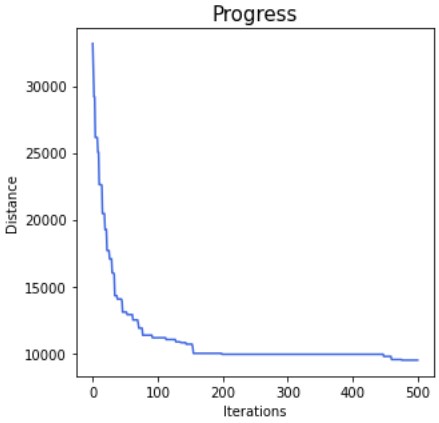
```
    for i in optimal:

        print(i)

    new_list=[]
    new_list=result
    new_list.append(new_list[0])

    #plotting the optimized graph and progress
    plotTowns()
    plotPath(result)
    plt.subplot(2, 2, 3)
    plt.title("Progress",fontsize=15)
    plt.xlabel("Iterations")
    plt.ylabel("Distance")
    plt.plot(lis, color= "royalblue")
    plt.show()
if __name__ == "__main__":
    main()
```

Appendix 2 - Sample Output

```
Plaease enter number of towns 25
Please enter the range for x and y coordinates 2400
Initial path distance is: 33189.19570728931
Random path chosen is
19
10
21
8
24
2
22
12
4
11
16
1
6
0
15
5
23
9
20
17
14
7
3
18
13
19
Initial route distance is: 33189.19570728931
Optimized path distance is: 9490.701800506362
Optimal Path visited is
10
24
23
3
20
5
17
2
14
13
1
22
19
6
15
8
21
18
7
12
0
9
11
4
16
10
```


Randomized Path
Towns
Distance
Distance
0
500
1000
1500
2000


Optimized Path
Towns
Distance
Distance
0
500
1000
1500
2000


Progress
Distance
Iterations
10000
15000
20000
25000
30000
0
100
200
300
400
500