Travelling Santa 2018 - Prime Paths

Apurva Modi, Anoop Reddy Ananthula, Herambeshwar Pendyala

cs580 - Introduction to Artificial Intelligence | Course project | Old Dominion University

01122493, amodi002@odu.edu

01160083, aanan002@odu.edu

01130541, hpend001@odu.edu

***Abstract*— This document serves as a project report for course cs580 Introduction to Artificial intelligence. The problem is from kaggle, the solution is given using a genetic algorithm approach using various approaches.**

***Keywords*— Travelling Santa, Prime Paths, Kaggle, Genetic Algorithm, Ordered Crossover, Random Swapping.**

I. Introduction

Christmas is approaching and santa is getting ready with gifts and so is kaggle. Kaggle is an online platform for data science. It allows users to find and publish data sets, explore and build models. Every year at this time of christmas kaggle comes up with a santa problem that uses many algorithmic approaches proposed in Artificial Intelligence and Machine learning. As part of the course Introduction to Artificial intelligence, we were given a Kaggle problem to work on as a project. This proposed solution uses genetic algorithm to find the optimum path is discussed in the following sections.

II. Problem Statement

Rudolph the red-nosed reindeer has always believed in working smarter, not harder. And what better way to earn the respect of Comet and Blitzen than showing the initiative to improve Santa's annual route for delivering toys on Christmas Eve?

This year, Rudolph believes he can motivate the overworked Reindeer team by wisely choosing the order in which they visit the houses on Santa's list. The houses in prime cities always leave carrots for the Reindeers alongside the usual cookies and milk. These carrots are just the sustenance the Reindeers need to keep pace. In fact, Rudolph has found that if the Reindeer team doesn't originate from a prime city exactly every 10th step, it takes 10% longer than it normally would to make their next destination!

The problem is can we help Rudolph solve the Traveling Santa problem subject to his carrot constraint ? .

The given problem looks more like a travelling salesman problem where we need to find shortest path covering all nodes exactly once. Our approach to solve the problem uses genetic algorithm taught in the class which gives optimum solution for travelling santa problem.

III. Dataset

we are provided a list of cities and their coordinates in cities.csv. we must create the shortest possible path that visits all the cities. our submission file is simply the ordered list in which you visit each city. Below two figures give us a look of how dataset and submission file looks like.

	CityId	X	Y
0	0	316.836739	2202.340707
1	1	4377.405972	336.602082
2	2	3454.158198	2820.053011
3	3	4688.099298	2935.898056
4	4	1010.696952	3236.750989

Fig. 1 First 5 rows in the dataset

Path
0
1
2
3
4
5

Fig. 2 First 5 rows in the submission file

Paths have the following constraints:

i. Paths must start and end at the North Pole (CityId = 0)

ii. You must visit every city exactly once

iii. The distance between two paths is the 2D Euclidean distance, except...

iv. Every 10th step (stepNumber % 10 == 0) is 10% more lengthy unless coming from a prime CityId.

IV. Visualizations

As we are approaching a problem, it is always good we start with a proper visualization of the data. as this gives us more insights about the data. At the very beginning we plot the given data to see what it looks like, below is the image of the given data.

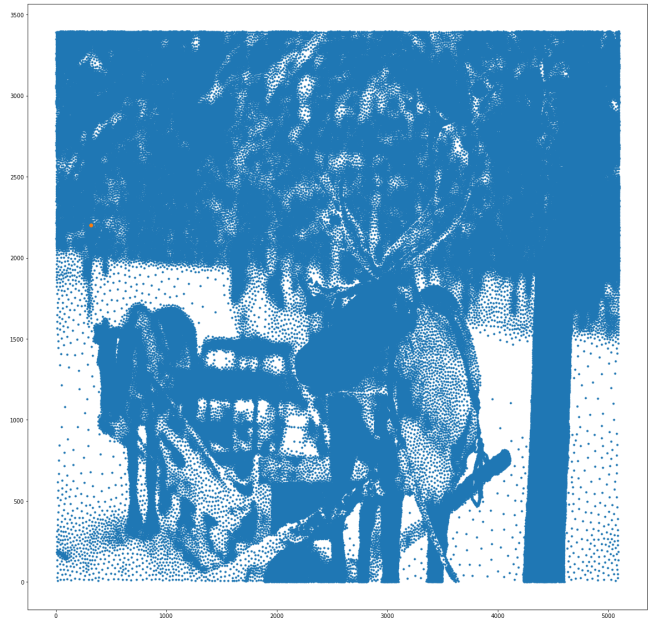

Fig. 3 visualization of the whole data

So it looks like the cities are arranged in a reindeer pattern. The red dot indicates the North Pole (CityId = 0). Our task is to find a path that goes from the red dot, touches all the other dots, and comes back to the red-dot, with minimum total distance travelled.

As we have seen how visualization helps us to see what is there in data, We will be doing more visualizations in the next sections of the paper as well.

V. Methods

To start with solving the problem we would like to see what is a travelling salesman problem and approaches to find the optimum path, followed by how do we calculate the cost while keeping up with the constraints, getting all the prime cities and then apply Genetic Algorithm to find the optimum path of the problem.

A. Travelling Salesman problem

The travelling salesman problem is an optimization problem in computer science, which tries to find the shortest path given a number of cities on a map. As there are many possible combinations to generate a shortest path, this problem comes under NP-hard problem. The worst-case running time for any algorithm for the TSP increases exponentially with the number of cities.

In this project we would be solving the travelling santa with the use of genetic algorithm which tends to be an optimal approach taught in the course.

B. Calculating the Cost

The cost is given as distance between any two points(cities) is given by Euclidean distance is the cost between any two cities, below is the equation to calculate the distance.

$$\text{Distance}(x,y) = [(x_1 - x_2)^2 + (y_1 - y_2)^2]^{1/2}$$

we should also see that at every 10th city the cost is increased by 10% unless it is coming from a prime city. In such cases given below pseudo code calculates the distance given a path of cities.

```
def calculateDistacne(traversePath):
    step_num = 0
    distance = 0
    currentCity = firstCity
    for city in traversePath:
        nextCity = city
        distance = distance +
        euclidean(currentCity, nextCity)*(1+
        0.1*((step_num % 10 ==
        0)*int(not(prime_cities[prev_city]))))
        currentCity = nextCity
        step_num + = step_num
```

C. Getting all the prime cities

In order to satisfy the constraint to reduce the cost we need to get the prime cities first. so we applied a function to generate another column isPrime that has true if the cityId is prime and false if not. we use sieve of eratosthenes to find the prime numbers in a fast and efficient way. Below are the first five rows of data after the column is appended.

	CityId	X	Y	isPrime
0	0	316.836739	2202.340707	False
1	1	4377.405972	336.602082	False
2	2	3454.158198	2820.053011	True
3	3	4688.099298	2935.898056	True
4	4	1010.696952	3236.750989	False

Fig. 4 Data after appending prime

From the above we can see it is convenient for us to check if the city is prime or not. There are total 17802 prime cities out of total 197769 cities. now we can apply genetic algorithm to generate optimum paths.

D. Genetic Algorithm

Genetic algorithm is a type of Evolutionary Algorithm that is taught as a foundation concept in Artificial Intelligence course. Genetic Algorithm is inspired by the process of natural selection and are commonly used to generate high-quality solutions to optimization and search problems by relying on bio-inspired operators such as mutation, crossover and selection.

Working of Genetic Algorithm : It basically has three stages – selection, crossover (also known as recombination), and mutation.

During the selection phase, a part of the existing population (the subjects or the features are called population) is chosen for breeding. Ideally, the percent of the population that satisfy the fitness function are chosen for the selection phase.

The fitness function in this case will be the cost of the path, we will only choose the best fit population to be passed to the next generation i.e the one with lowest path threshold are only passed to the next generation.

Then comes the crossover phase where we mate the strings we have chosen in the previous phase. Crossover, in general, involves swapping the strings that we have chosen during this phase.There are many types of crossover types but for this problem, we have chosen the ordered crossover as it generates more fit population. Below is a figure showing how ordered crossover is done.

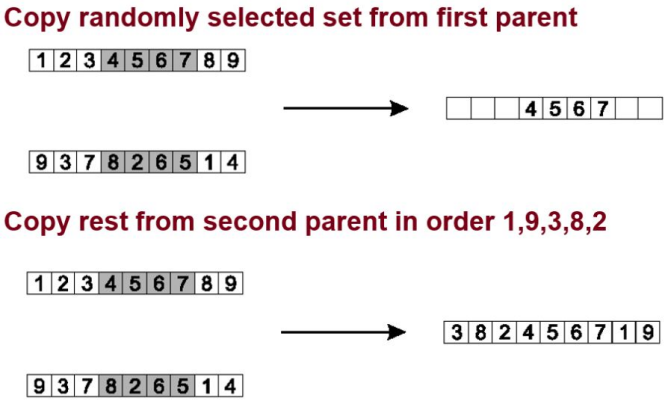

Fig. 5 doing ordered crossover

After this phase comes the next phase.The third phase is the mutation phase. Here, we apply random mutation. We copy the bits from the crossover to the next population. For each bit we are to copy to

the next population, we take in a small probability of error.

Based on these three operators majorly, we run our algorithm. we execute this algorithm for a certain amount of generations to get the best fit with shortest path cost. Below is a flow chart for working of a genetic algorithm.

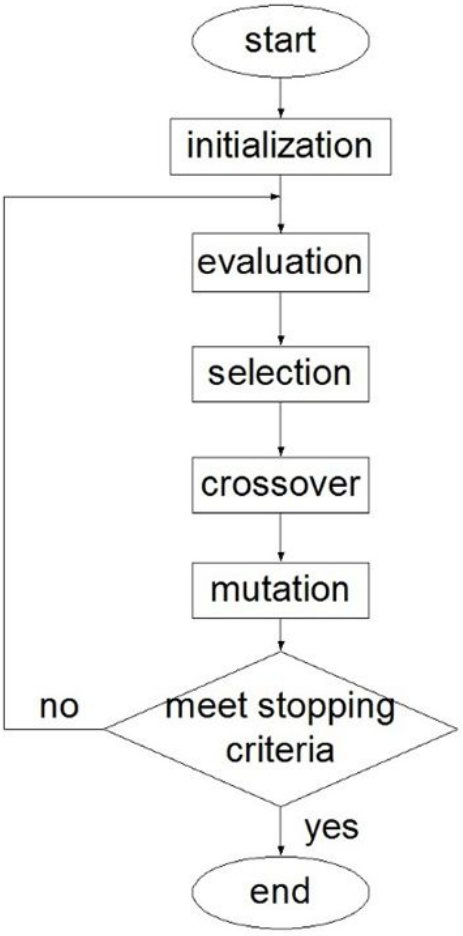

Fig. 6 Working of a genetic Algorithm

VI. Languages and Packages

We used Python as programming language to solve this problem as it is easy to implement. We used Numpy and Random for operations like calculating Euclidean distance and picking random points or random shuffling. We used Pandas to read the dataset and work on them and used Matplotlib to plot the path.

VII. Experiments and Results

To start with the shortest path we took the point which was visit every CityId order(given order) and come back to the first city when we reach the end. This path can be termed as brute force or dumbest path that can be taken.

For a Genetic Algorithm we give this dumb path as the starting point and generate random population of size N. Then we generate next generation using ordered crossover, swap mutation for Nc generations and then see the shortest path. below is a small flow chart describing how this process works.

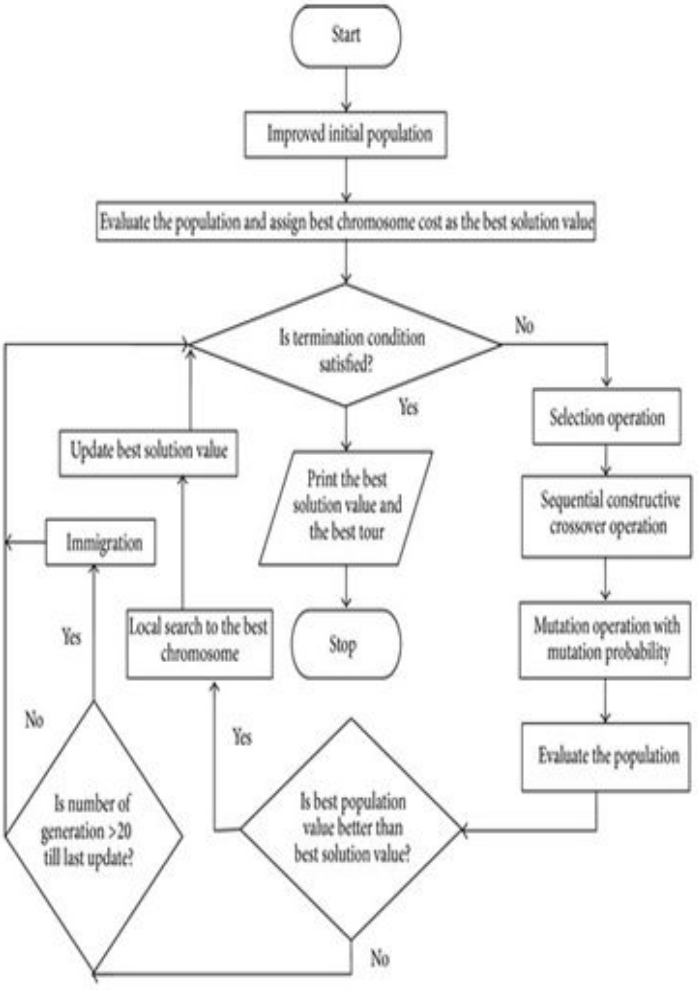

Fig. 7 Travelling Santa Genetic Algorithm Approach

Below are the figures showing dumb path and optimum path generated from the genetic

algorithm approach we used for a small subset of data.

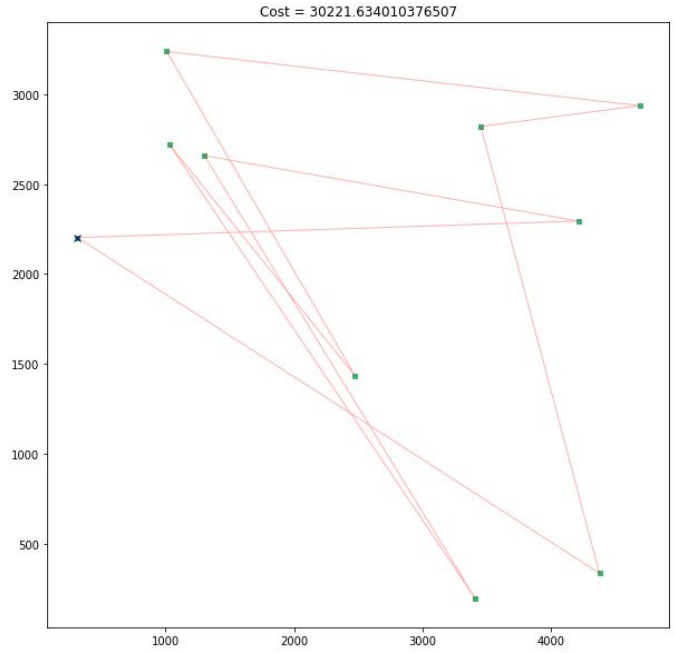

Fig. 8 initial Dumb Path for first 10 cities.

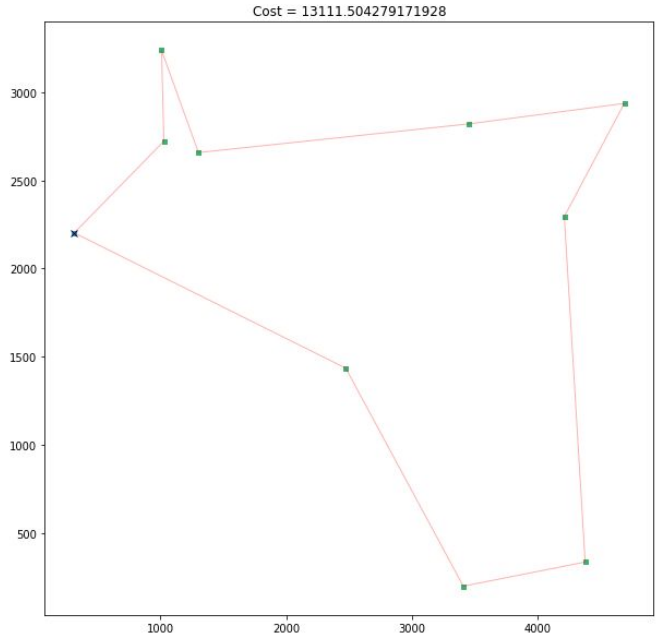

Fig. 9 final shortest Path for first 10 cities.

From the above 2 figures we can see that there are no intersections between any city in the path generated by our genetic algorithm, This can be clearly seen in the cost of the entire path as well, the dumb path generates a cost of '30221.634' where as the optimum path cost generated by the algorithm is '13111.50'. we can see the efficiency of the algorithm in finding the shortest path.

for the above result we have used a population size of '250' and executed the genetic algorithm from '500' generations, with a runtime of 9.6 ms.

VIII. Analysis

As we have seen in the previous section we generate an optimum path starting from a dumb path, to improve the efficiency of the algorithm we tried to sort the data based on the co-ordinates of the city and they apply genetic algorightm to it everytime we initiate a population in a generation. In the beginning we were able to see the difference as the sorted method tend to produce more optimum result compared to the previous unsorted method. below are the paths generated by the sorted methods.

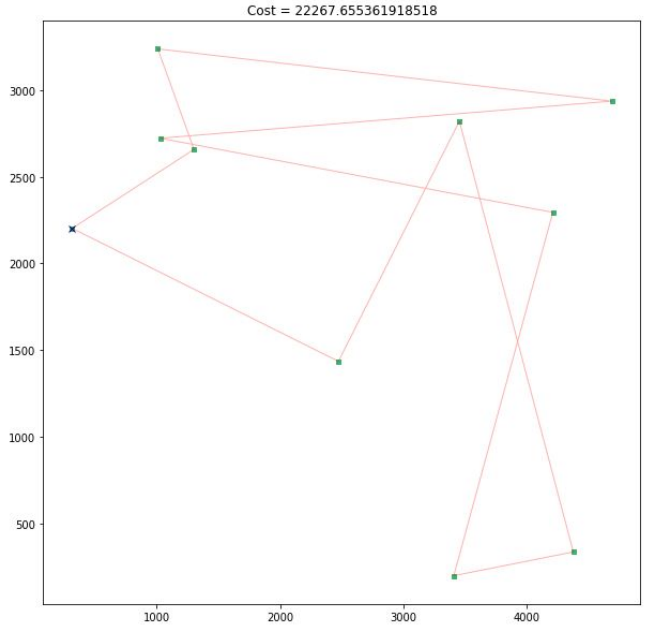

Fig. 10 initial Dumb Path for first 10 sorted cities.

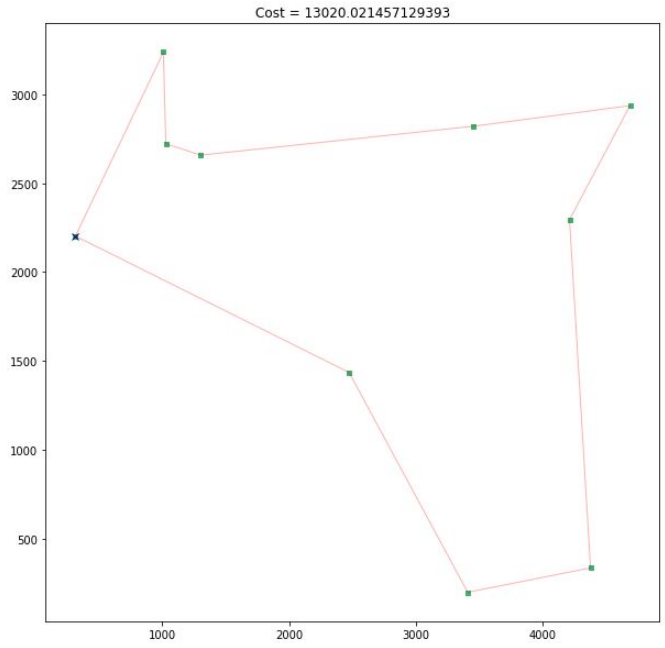

Fig. 11 Final Path for first 10 sorted cities.

However while executing for a larger subset of data we found that it is not so efficient as the order in which we generate tend to be in a random way. below table gives an overview of the optimum path cost and dumb path cost for various subsets of data for a population size of 250 and 500 generations.

TABLE I
PATH COSTS FOR RUNS ON VARIOUS SUBSETS OF DATA

Population size	Genetic Algorithm path cost		
	Dumb Path	Unsorted path	Sorted Path
10	30221.634	13111.50	13020.02
25	60651.781	18709.97	22796.07
100	214757.18	60519.64	56615.83
1000	2192177.34	1766486.29	1800182.51

IX. CONCLUSIONS

We tried different methods to get an optimum path leading to shortest path to travel every city exactly once while satisfying the prime city constraint. we generated a path cost of '96,478,811.25' with the dumb path cost being '446,884,407.52' this took us around 15 hours to run on our machines with a population size of 250 and for 500 generations.

This implementation can be further improved by various methods available. The first spot on kaggle was able to achieve a score of '1,513,747.36'. we can tweak the population size and number of generations to get a more optimized solution when executed on a cluster.

ACKNOWLEDGMENT AND FUTURE SCOPE

As we have implemented only basic genetic algorithm taught in the class, this can be improved to give a better performance and find optimum path cost. This can be achieved by tweaking the hyper parameters population size and number of generations.This also requires a much more computing power and due to limitations we were unable to find the most optimum solution for the problem.

The other approach we wanted to implement was to find the optimum path for all the prime cities first and then insert the remaining cities in an order that gives us less cost but this approach requires a lot more computing power, so we have kept this as a future scope.

REFERENCES

[1] https://www.kaggle.com/c/traveling-santa-2018-prime-paths/overview
[2] https://opensourc.es/blog/kaggle-santa-2018
[3] https://www.rookieslab.com/posts/fastest-way-to-check-if-a-number-is-prime-or-not
[4] https://github.com/jessiejc23/travelling-santa-2018/blob/master/v3.0%20GA.ipynb.
[5] https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35
[6] https://figshare.com/articles/_Flow_Chart_of_Genetic_Algorithm_with_all_steps_involved_from_beginning_until_termination_conditions_met_6_/1418786