**Project 4**

Jeremy Boyd

[Computer Science and Engineering]

Speed School of Engineering

University of Louisville, USA

[jrboyd03@louisville.edu](mailto:jrboyd03@louisville.edu)

1. **Introduction** (What did you do in this project and why?)

For this project I was responsible for solving the Travelling Salesperson Problem which is a problem programmers have been working on since forever. It involves a salesman being given a list of cities to travel to and their coordinates the problem is to find the most efficient route to take.

The approach I took to solve this problem was to create a genetic algorithm, this is an approach that reflects the process of natural selection. Where the fittest individuals are selected for reproduction in order to produce offspring of the next generation. Fitness meaning a scale that tells us how good the option is for solving our problem. This process starts with the selection of the fittest individuals from a population, they then produce an offspring which inherits their characteristics and will be added to the next generation. This process iterates and at the end, a generation with the fittest individuals will be found.

This algorithm can give you a diverse array of results depending on how you build your algorithm. You could get a really good solution or you could get a really bad one depending on the type of crossover and mutation you use. When implementing this type of algorithm you’re not guaranteed to get the shortest route. For the crossover and mutation methods I used for this project there is quite a bit of randomness to it since I do not control the initially generated paths. So it’s unknown how fit those initial paths may be and also although I set the mutation rate I do not control when a mutation actually happens and whether that mutation is actually beneficial to the offspring or not. So there are a lot of variables at play here which is why this algorithm doesn’t guarantee the most optimal solution or even a good solution depending on how you create it.

1. **Approach** (Describe algorithm you are using for this project)

The programming language I used for this project was Python. The algorithm for this project was definetly the most difficult to implement so far in my opinion. To build my genetic algorithm I started by importing the math, collections, random, operator, pandas, and numpy libraries. Then I brought in everything I’d be using from my previous project including my ‘distance’ function to calculate the distance between two points. Also the table function that would read in the data from the tsp file and store it in a table, a list that held all the x and y coordinates stored in the table, and 3 separate list that individually held the number of cities, the x coordinates, and the y coordinates.

Next, I created a ‘City’ class that would allow me to create all the cities using their x and y coordinates. Within the City class I put my distance function, after this I created a ‘Fitness’ class, you can think of the fitness as acting as the inverse of the route distance. Meaning I’m trying to get as small of a route distance as possible but a larger fitness score is better. The Fitness class holds a route, its distance, and its fitness score. It also includes a ‘routeDistance’ function that calculates the distance of the route and that route distance is used in the ‘routeFitness’ function that I included in the class. Which gives a route a fitness score based off its calculated distance.

With the City and Fitness classes created, I then needed to create my initial population or first generation. Population meaning a collection of possible routes. To create an individual route I created a function called ‘generateRoute’ that takes in a list of cities and then randomly selects the order in which each city is visited. This function produces one individual route so in order to generate a full population I created a function called ‘initialPopulation’ which takes in the number of cities and the list of cities. It then loops through the ‘generateRoute’ function and adds each generated route to a list. The population size is determined by the number of cities in the list. So if there’s 100 cities then 100 routes will be generated. This way we always start with a good size population in correspondence with how many cities need to be visited.

Next in order to simulate the “survival of the fittest” aspect of natural selection I made use of each routes Fitness and created a function to rank each individual in the population. Which would return an ordered list with the route IDs and each ones associated fitness score. After this the next step was to select the mating pool or the parents that will be used to create the next generation. The approach I took to do this was fitness proportionate selection. Using this method, the fitness of each individual relative to the population is used to assign a probability of it being selected. Giving each individual a fitness-weighted probability of being selected. I also included a feature called elitism which ensures that the best performing individuals from the population will automatically carry over to the next generation, ensuring that the most successful individuals persist.

So to implement these approaches I created a function called ‘selection’ which takes in the list from the ‘rankRoutes’ function and a chosen size for the number of elite individuals that will be kept. I then created a table out of the ranked routes which held the routes index and fitness scores. I then used cumsum to include a row in the table that would compute the cumulative sum of all the fitness scores up to their index in the table. Then turned that value into a percentage which would be the fitness weight for each individual. I then used a for loop to retrieve the elite paths which is a number I choose. And then I use another for loop where a randomly drawn number is compared to the weights of all the remaining paths to determine if they will be included in the mating pool or not. After this I created a function to extract the individuals from the population that were selected to be in the mating pool.

Now that I have my mating pool has been created it was time to create the next generation by using a breeding function called **ordered crossover**. In ordered crossover, a random subset of the first parent route is selected and then the remainder of the route is filled with cities from the second parent in the order in which they appear without duplicating any cities selected from the first parent. To do this I created a function called ‘breed’ which takes in two parent routes and then selects a random city from each of them. Between these two cities the smaller numbered city becomes the start of the gene and the larger numbered city becomes the end of the gene. All the cities between the start and end point make up what parts of the path from parent one will be included in the offspring and then the rest is filled in by parent two. Next, I generalized this process to create an offspring population

Next I needed to introduce mutations and I specifically used **swap mutation** which is when with a specified low probability, two cities will swap places in a route. I used random to generate a random float in between 0.0-1.0 and if that number is less than my chosen mutation rate a mutation will occur. Then I created a function that ran the whole population through my ‘mutation’ function to create a mutated population.

After this I put all the steps together in one function that will produce a new generation. First it ranks the routes in the current generation, then determines potential parents by running the selection function, which creates the mating pool using the mating pool function. And finally the new generation is created using the ‘breedPopulation’ function. Then lastly in a function called ‘geneticAlgorithm’ that takes in a list of cities, the number of cities, a number for how many elite paths will be chosen, a mutation rate, and how many generations to be produced. I create the initial population in the function and then loop through these steps for as many generations as I desire. The function also displays the initial best route and the final best route so we are able to see the improvement.

Lastly, in order to visually represent my results I imported matplotlib.pyplot. Which is a plotting library for the python programming language which includes a general purpose GUI. I created a function that would store the shortest distance from each generation in a progress list then plot the results in order to see how the distance improved over time. I then created two list that would take the final path generated by my genetic algorithm. I used the indexes of the cities in the path in order to append the x and y coordinates in the proper order that corresponds to where the city is located in the generated path. So they can be plotted in the proper order, I then created a function that would take in a list of x coordinates and y coordinates it would then plot each of the points in the order indicated by the list and display them using the GUI.

1. **Results** (How well did the algorithm perform?)

The algorithm performs well in an efficient amount of time but gets mixed results depending on the crossover and mutation combination you use.

|  |  |  |
| --- | --- | --- |
|  | Population Size: 100 | Population Size: 200 |
| Swap Mutation rate: .01 | Dataset 1A | Dataset 1B |
| Swap Mutation rate: .001 | Dataset 2A | Dataset 2B |

Dataset A Parameters: elite size = 20, generations = 500

Dataset B Parameters: elite size = 40, generations = 500

I ran each dataset 10 times to develop performance statistics which are depicted in the table below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Min | Max | Average | Standard Deviation | Runtime\* |
| Dataset 1A | 3177.724 | 3655.607 | 3424.427 | 138.122 | 2 min 45 sec |
| Dataset 1B | 8362.953 | 9056.528 | 8797.984 | 188.635 | 10 min 45 sec |
| Dataset 2A | 1576.616 | 1825.032 | 1673.286 | 78.811 | 2 min 37 sec |
| Dataset 1B | 4342.183 | 5034.953 | 4777.689 | 213.832 | 9 min 55 sec |

\*Runtime refers to the time to calculate the final distance path not runtime of the whole program including the graphs

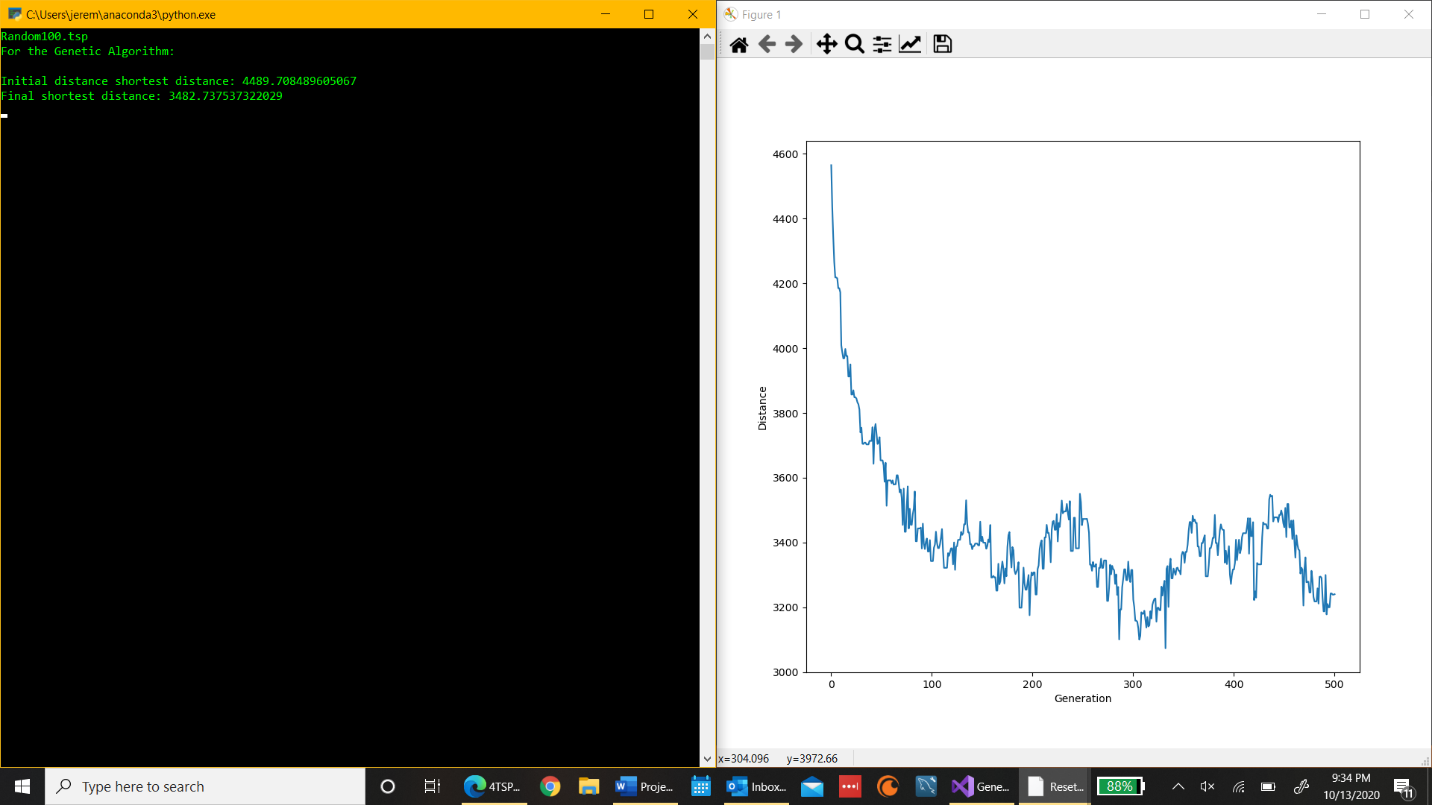
* 1. **Data** (Describe the data you used.)

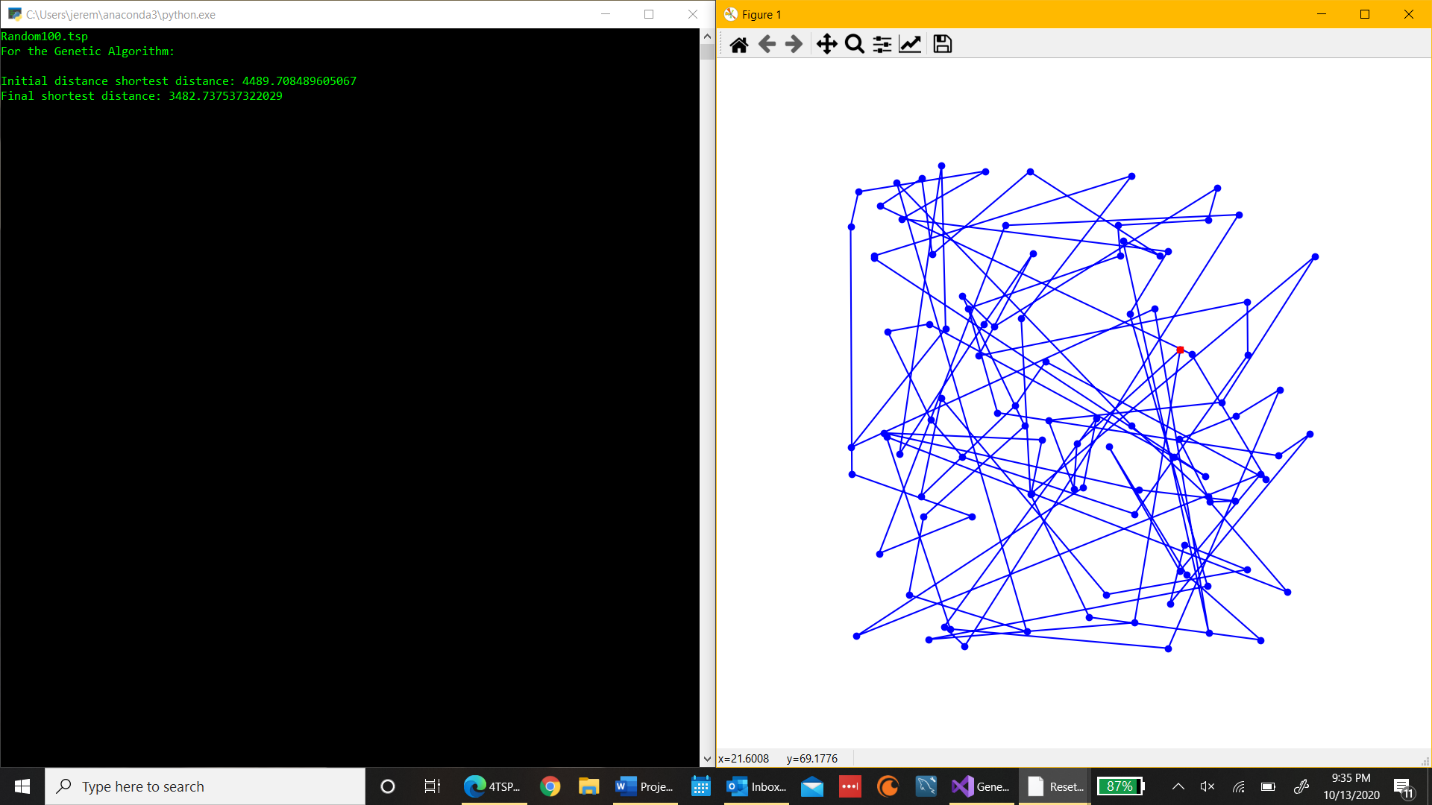
I used the given data in the tsp files we were given, which included 100 numbered cities and their respective x and y coordinates. And then I randomly generated 100 more cities to run a dataset with 200 cities.

I also used the tsp files from previous projects with less cities to see how close the distances I was getting were to the actually shortest distance I got from other algorithms.

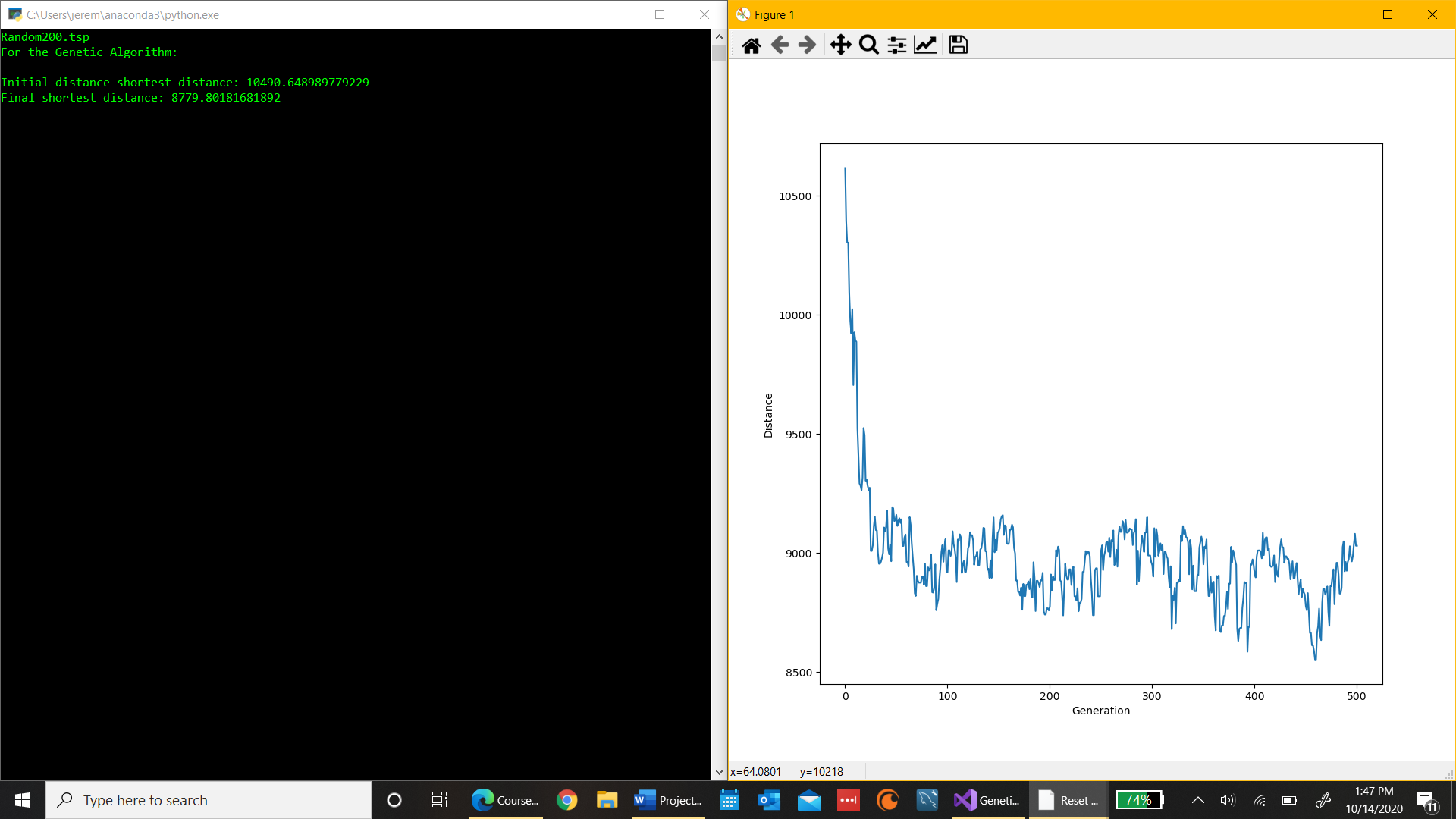
* 1. **Results** (Numerical results and any figures or table)

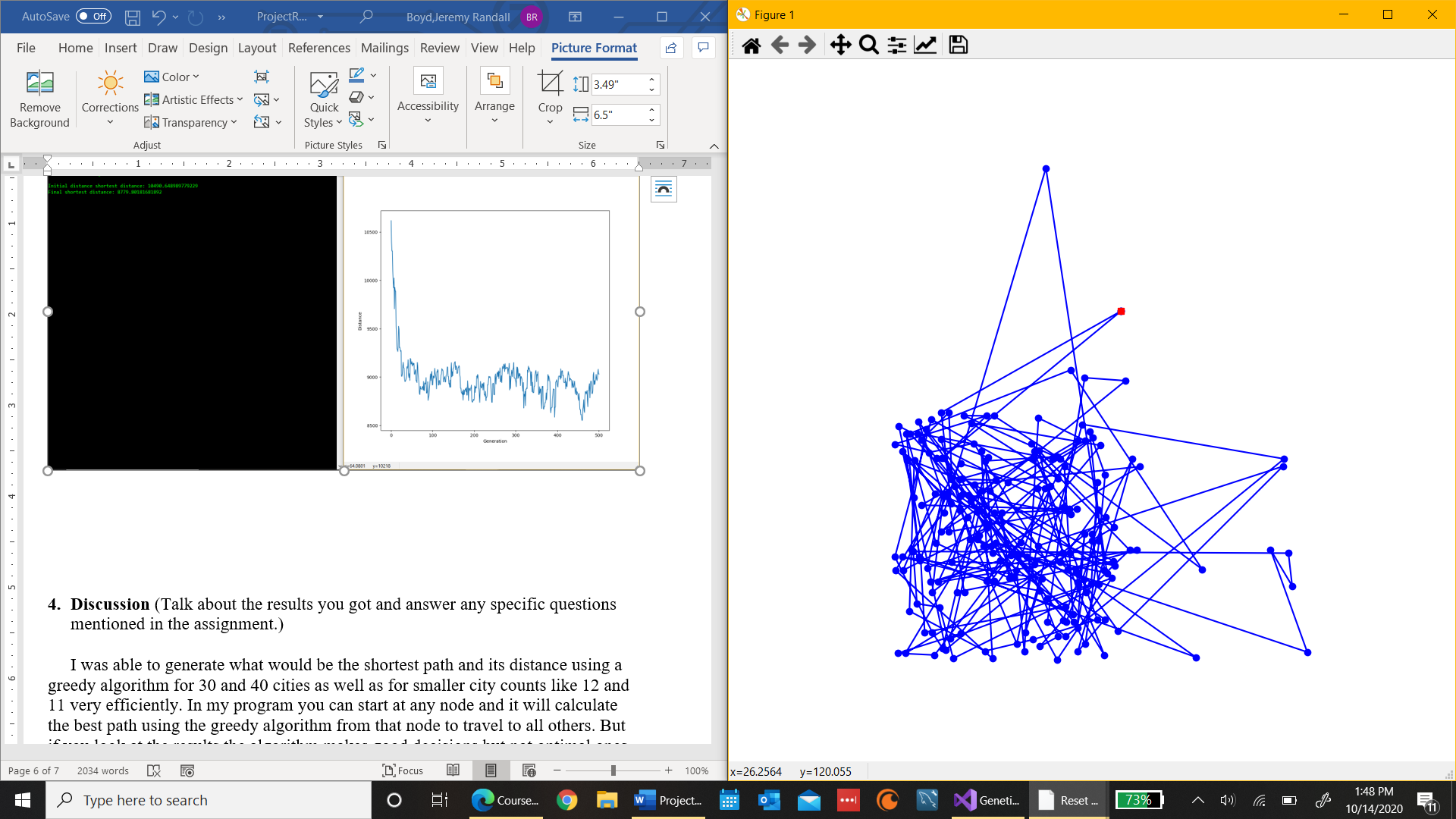
**Dataset 1A Example:**

-

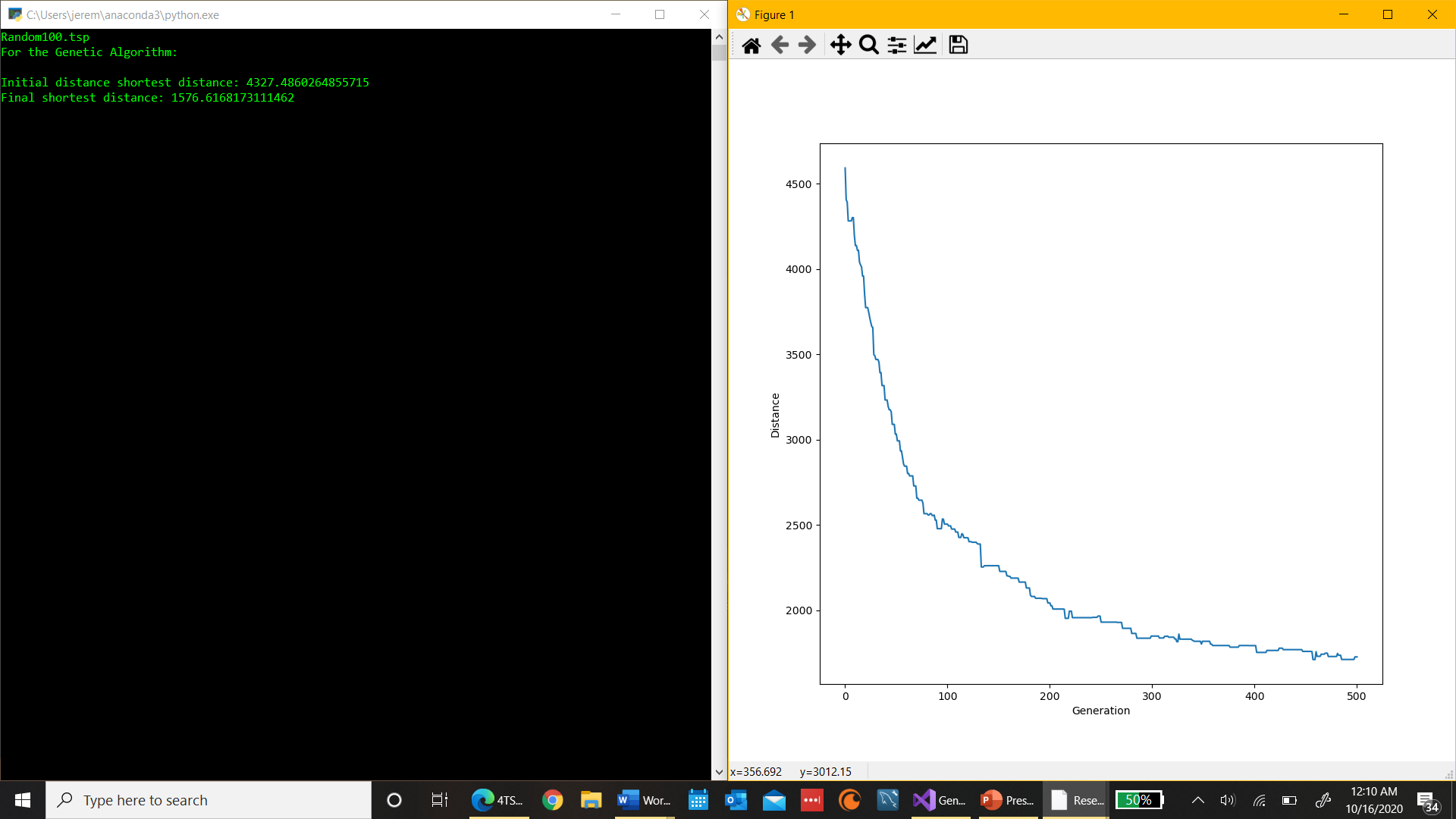


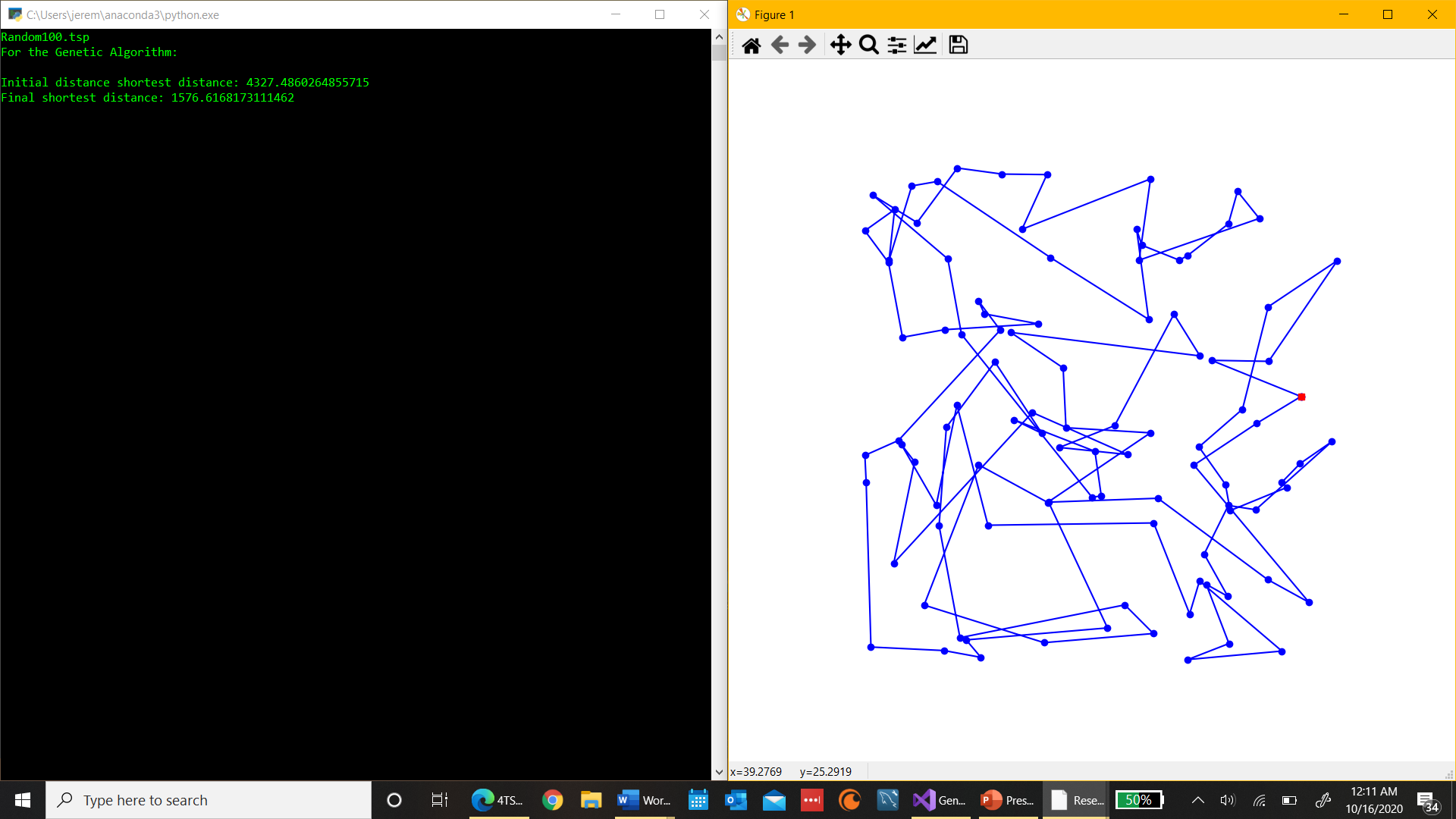
**Dataset 1B Example:**



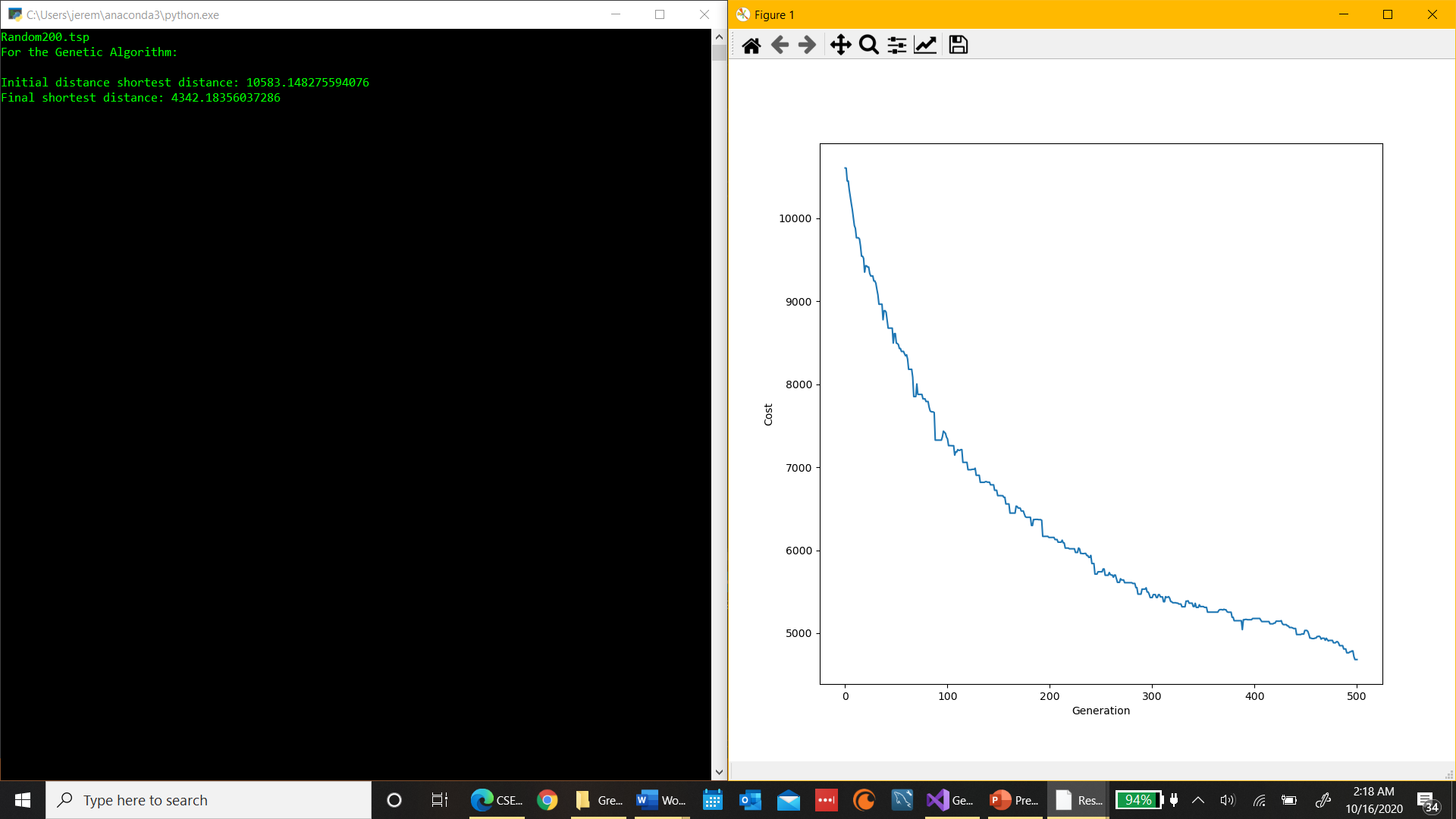


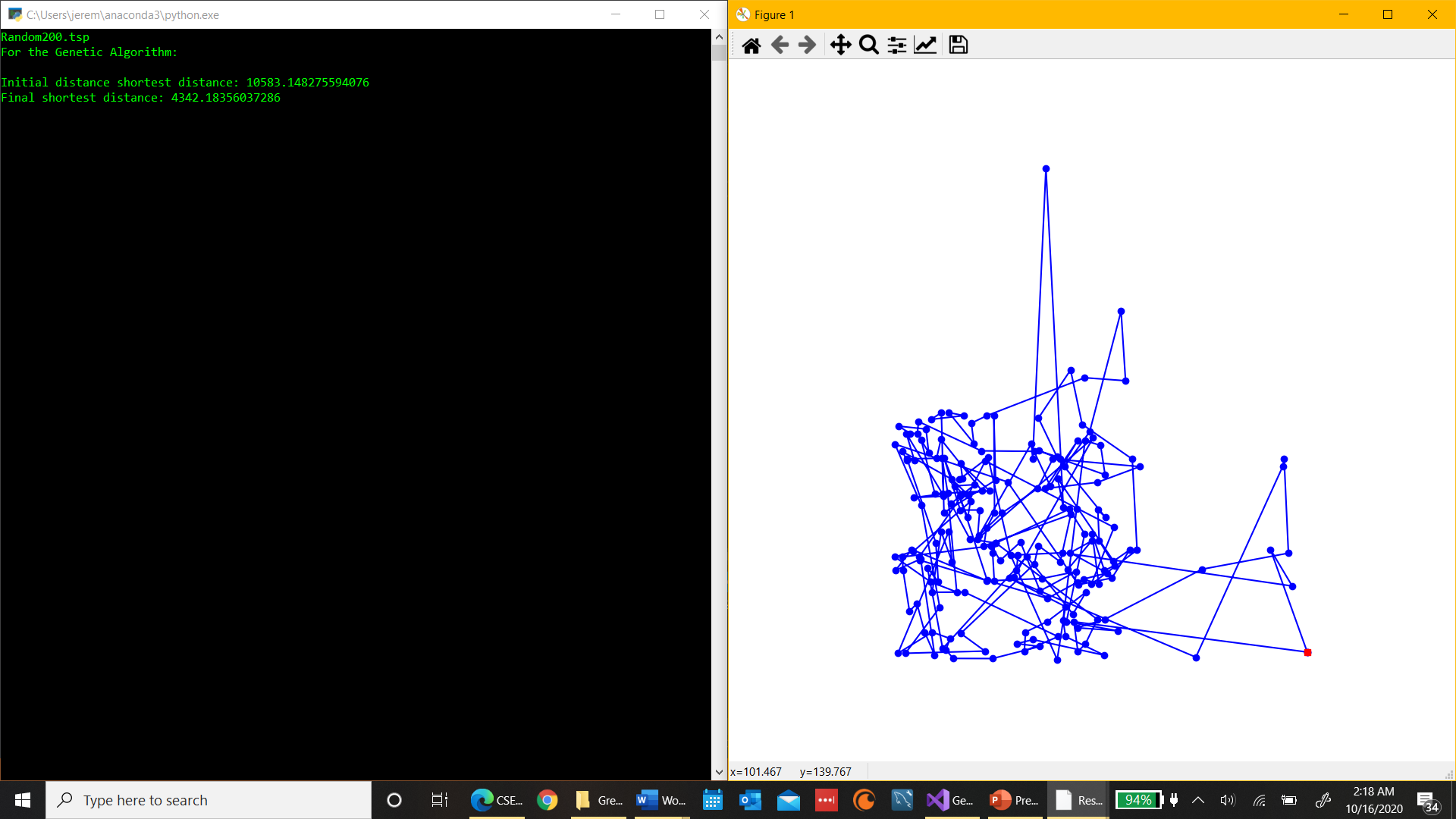
**Dataset 2A Example:**





**Dataset 2B Example:**





1. **Discussion** (Talk about the results you got and answer any specific questions mentioned in the assignment.)

I was able to generate a paths using my algorithm as you can see the mutation rate played a huge role in how optimal of a path would be generated. I did some more testing with higher mutation rates and the higher the rate the worse the algorithm performed. But, there is significance in the fact that compared to the initially found shortest distance the final distance is significantly shorter even if it doesn’t produce the shortest route possible. Using the ordered crossover combined with the swap mutation these things helped lead to a much shorter path than the one that is initially generated by the program. I’ll leave the reference to the website where I got the idea from at the end of the report. There’s definetly more optimal combinations of mutations and crossovers but this one works very well when using a smaller mutation rate. They rely quite a bit these were just the first ones I came across and decided I’d implement them for this project.

While using the ordered crossover and the swap mutation the elements I investigated were a change in population size and a change in the mutation rate. The change in population size caused a huge increase in the runtime of my program and the change in mutation rate cause a huge change in whether I got an optimal path or not. The difference in path distances from the two mutation rates is quite astonishing but a welcomed surprise for me. Runtimes differed greatly from the 100 to 200 cities as depicted in my table in the results section. The times went from a bit over two minutes to over ten minutes and the results I was getting with the higher mutation rate were quite discouraging. But after lowering it that wait time was more worth it because the results I was getting were much better.

My stopping criteria was when a specified number of generations had been produced in the test I ran, the number was 500 but you’re able to choose how many generations the program will go through before it ends. The best solution for my solved problem was definetly using the smaller mutation rate with the smaller set of cities it was able to produce a decent solution which was much better than the ones with the higher mutation rate. And it took the least amount of time compared to all the other datasets.

The computer I’m using is a LENOVO YOGA 720-12IKB with a Intel Core i7 CPU with a clock rate of 2.80 GHz, it has a 64-bit operating system, and I’m using Visual Studios as my IDE to run my code.

1. **Critical Thinking**

The biggest problem I had was getting my code to run all the way through I had to create several functions that were interrelated and getting everything to work properly took quite a bit of time. The testing stages took me hours because there was always some type of small issue I had to find and fix which is normal for most programs. But this was a rather large program so finding those smaller errors was like finding a needle in a haystack.

The one thing I would change if I could do this project again is, I’d use a different combination of mutation and crossover that does a better job of producing an optimal path. I came across the crossover and mutation I used while doing research on the project and decided to implement them but if I could go back I’d try to create an algorithm that’s faster and produces more optimal solutions. Although it was still a good learning experience creating the algorithm and doing all the testing. Being able to create an algorithm that can use natural selection to create the most optimal path would’ve been interesting.

Using GA is my first experience with an algorithm that naturally improves itself. With all the other algorithms we’ve created they’re implemented in a way that you’re supposed to get the most optimal time every time you run it by using some type of tree method (besides brute force). And those algorithms are really only good for path finding. But this GA algorithm could be useful in a multitude of ways and can be used for a variety of problems. This opened my mind up to the greater possibilities that there are of creating algorithms that produce solutions and then improve on them over time.

Overall, the genetic algorithm as a problem solving technique I believe is an extremely effective one if implemented properly. The ability for a program to improve its solution over time could be really beneficial for different real world problems. For the TSP problem it could be the best algorithm if done correctly because it’s able to handle a high number of cities and produce a good solution. But if the algorithm is not created using an effective crossover and mutation combination it could be better to use another algorithm that may be better for guaranteeing that you actually get the shortest path.

1. **References** (If you used any sources in addition to lectures please include them here.)

Stoltz, Eric. “Evolution of a Salesman: A Complete Genetic Algorithm Tutorial for Python.” *Medium*, Towards Data Science, 17 July 2018, towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35.