**Project 5**

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 hybrid algorithm that combined genetic algorithm with a Wisdom of Crowds approach. This methodology refers to the finding that the aggregate of a set of proposed solutions from a group of individuals performs better than the majority of individual solutions. This approach takes several of the fittest individuals produced by the GA and combines their solutions to produce a better solution. This combination happens by going through each individual produced by the GA and seeing what similar edges they have and if an edge can be found in majority of the fittest solutions. It will be included in the final solution.

This algorithm can give you a diverse array of results because the genetic algorithm produces a unique solution every time. Sometimes you may have multiple similar edges helping to form an optimal solution. And sometimes you may get no similar edges at all. One run through you could get a close to if not optimal solution and others you may get something that is farther off from being optimal. And sometimes the series of individuals produced by the GA within them is a more optimal solution than the one that is produced by the WOC approach. When implementing this type of algorithm you’re not guaranteed to get the shortest route. It also depends greatly on the solutions your GA produces, if all those individuals are bad you’re more than likely not going to get an optimal solution. So there are a lot of variables at play here which is why this algorithm doesn’t guarantee the most optimal solution.

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 a nice challenge and I enjoyed putting it together. To build my 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. Also a list called ‘distances’ that would iterate through the list of coordinates and find the distance between each set of points. Then in conjunction with an asarray, all the values that are stored in the ‘distances’ list are imported into the array along with their indexes so that the asarray functions as a sort of look up table.

I also brought over my genetic algorithm that I used in the previous project, it used **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. And it used **swap mutation** which is when with a specified low probability, two cities will swap places in a route. The parameters were the same for every solution created: the number of elite solutions kept will be 1/5 of the number of cities in the dataset. So if you have 100 cities then the elite size will be 20, if it’s 200 then 40 and so on. The number of generations will be 500 and the swap mutation rate will be .001.

To create the wisdom of crowds part of the program I looped through the genetic algorithm 20 times and within the loops each edge from the resulting fittest solution was stored in a list. So when the loop was finished executing there would be a list that contained every edge from all the generated solutions. After this I created two sets of nested for loops that compared every edge from every solution to every other edge in all the other solutions. And I created a count variable that would keep track of if it found the same edge within another solution. Then once one edge has finished being compared to all others if the count variable was equal to half or more than half of the solutions it would be added to a list so it could be included in the final solution.

After this I created a function called ‘createFinalPath’ that took in the list of cities, the list of edges to be included, and the asarray that functions as a look up table. Within the function it creates two list called ‘fromNodes’ and ‘toNodes’ these list will hold the edges that must be included. One list holds the city being traveled from and the other holds the city that’s being traveled to. Then a random city is chosen to be the starting city and a list called ‘unvisited’ is created which holds the cities that have yet to be visited on the tour. Next I created a while loop that would continue to iterate till the unvisited list was empty. And within that loop it would check if the last visited city in the tour was within the ‘fromNodes’ or ‘toNodes’ list. And if it was it would find the index number of that city within the list and use that index to find the city within the opposite list. So that the edges that must be included in the final semester are chosen. But if the city is not included in an edge that must be in the final solution the program uses the greed algorithm nearest neighbor to find choose which city should be traveled to next. For this I brought in my nearest neighbor function that I used in a previous project. This function took in the last city added to the path, the list of unvisited cities, and the asarray. Within this function a for loop would iterate through the unvisited cities and using the asarray, look up all the distances from the current city to all other cities in the unvisited list of cities. And the one with shortest distance would be selected. This process continues until all cities are visited and a path is generated and returned by the function.

Next to calculate the distance of the path returned I used a for loop that would iterate for the length of the returned path and use the asarray to calculate the distance from one city to the next in the path and add them up inside a variable. Then the route and its distance were printed to the screen.

And lastly in order to visually represent the path taken I imported matplotlib.pyplot. Which is a plotting library for the python programming language which includes a general purpose GUI that will display the path I generated. First I created two list that would take the x coordinates and y coordinates out of my list that I used from previous projects that held the x and y coordinates for each city and stored them in separate list. I then created two more list that would take the path generated by my greedy function. To use 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 for a lower amount of cities but getting close to 100 cities it starts to take a while. (Which is mainly because of my computer because my code runs faster on other peoples computers.)

The genetic algorithm runs 20 times and I used those runs to develop the statistics depicted below.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Genetic Algorithm | | | | Wisdom of Crowds | |
| Min | Max | Average | Runtime\* | Distance | Runtime\*\* |
| 11 Cities | 351.0458 | 375.5623 | 357.0795 | 6 secs | 389.4708 | Instant |
| 22 Cities | 424.8851 | 559.6737 | 482.8296 | 12 secs | 416.0946 | Instant |
| 44 Cities | 684.5159 | 867.6777 | 792.513 | 35 secs | 699.0498 | .25 sec |
| 77 Cities | 1171.7857 | 1411.2420 | 1276.758 | 1 min 37 sec | 811.4841 | 1 sec |
| 97 Cities | 1570.5163 | 1984.1717 | 1701.725 | 2 min 33 sec | 957.2600 | 1 sec |
| 222 Cities | 4348.2139 | 5105.4921 | 4761.864 | 13 min 12 sec | 1291.0673 | 2 sec |

\*Runtime refers to the time to calculate the single final distance of one path

\*\* Runtime refers to the time it takes for the program to generate the final solution after all GA paths have been generated

Highlighted – had no common edges

All Dataset Parameters:

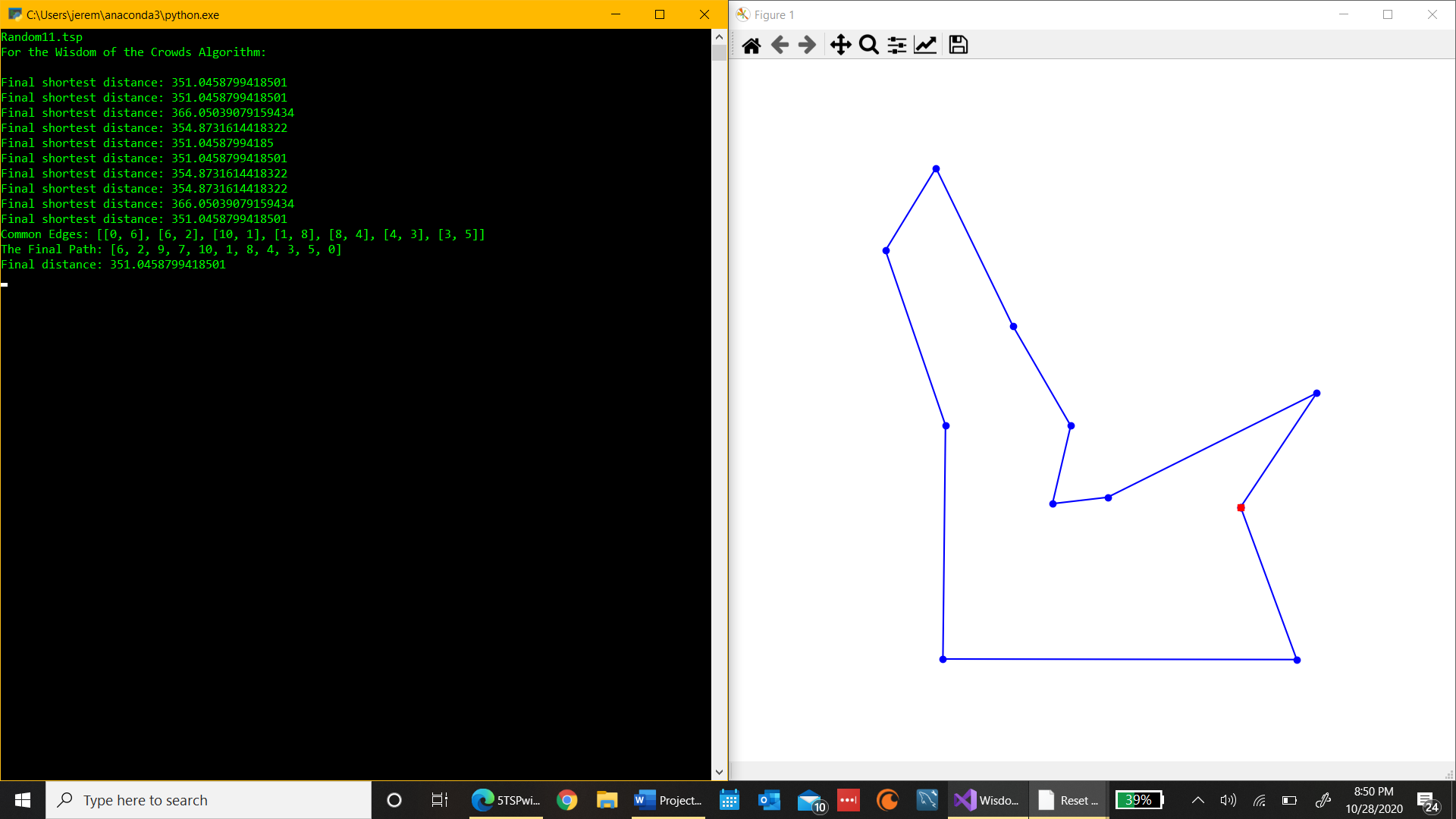
* # of cities/5 = elite size
* Generations = 500
* Swap Mutation rate = .001
  1. **Data** (Describe the data you used.)

I used the given data in the tsp files we were given, which included 11, 22, 44, 77, 97, and 222 numbered cities and their respective x and y coordinates.

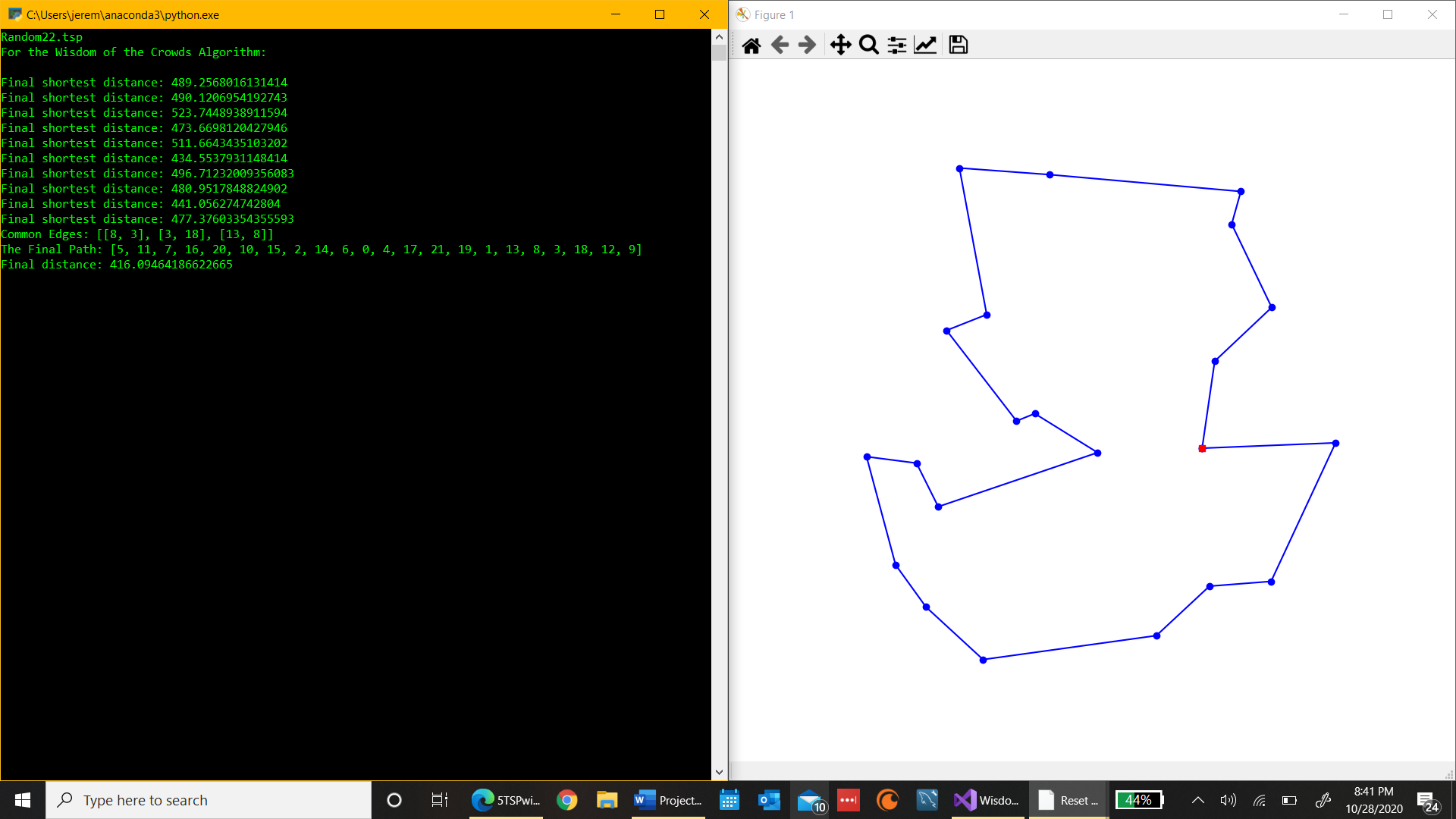
I also used the tsp files from previous projects with less cities to see how close the distances I was getting were to the actually shortest distance I got from other algorithms.

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

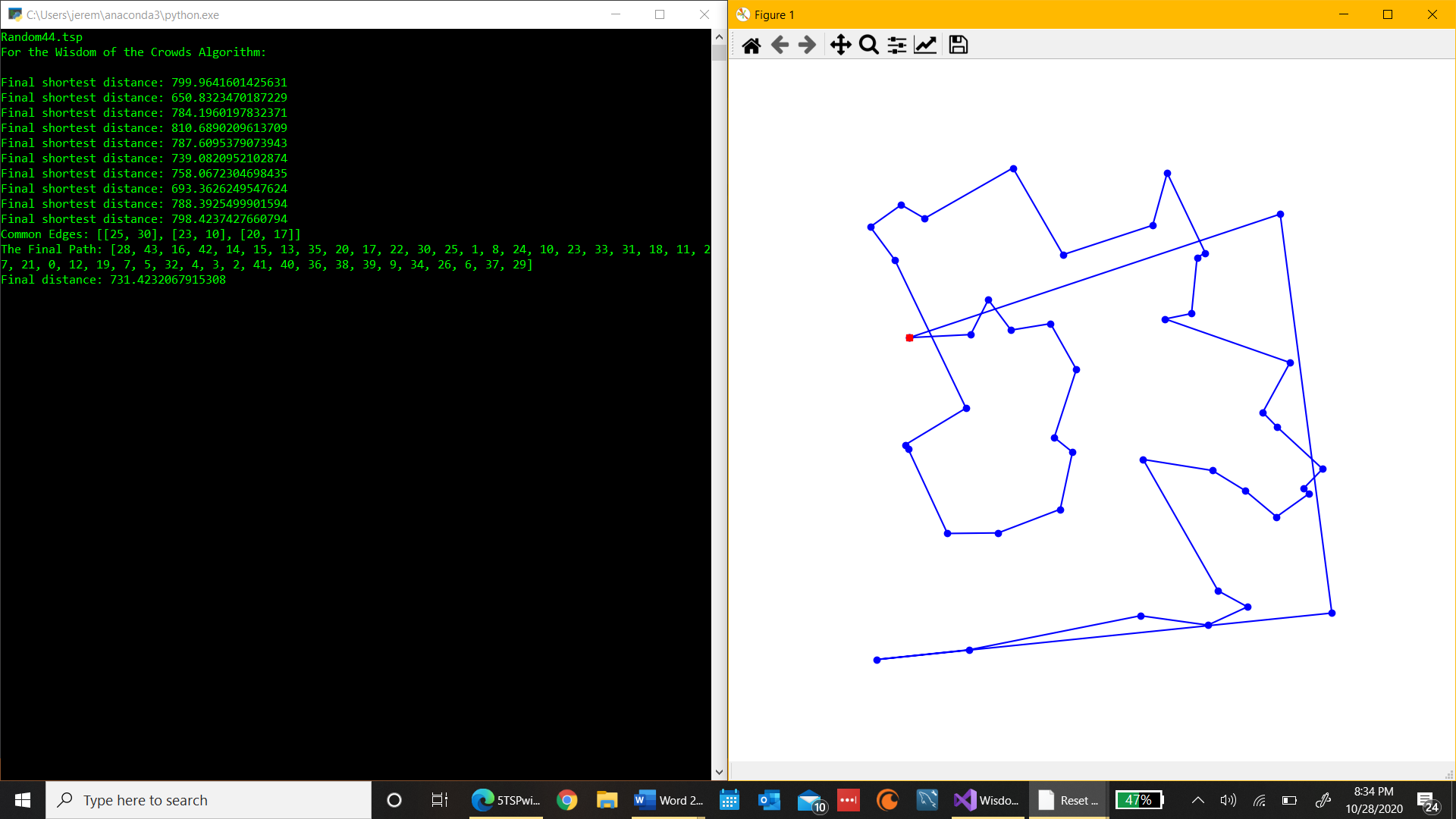
**Random11.tsp Example:**



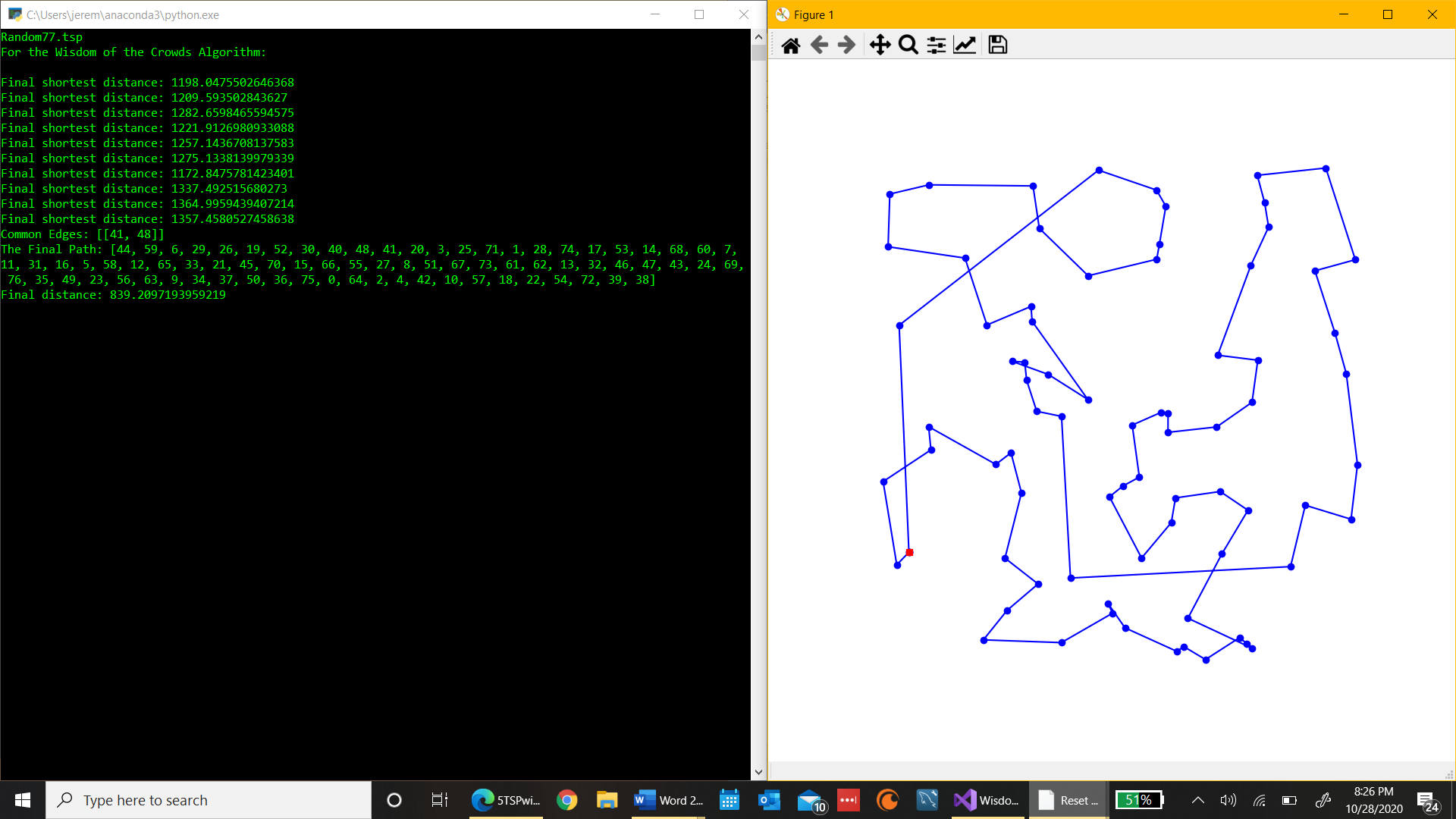
**Random22.tsp Example:**



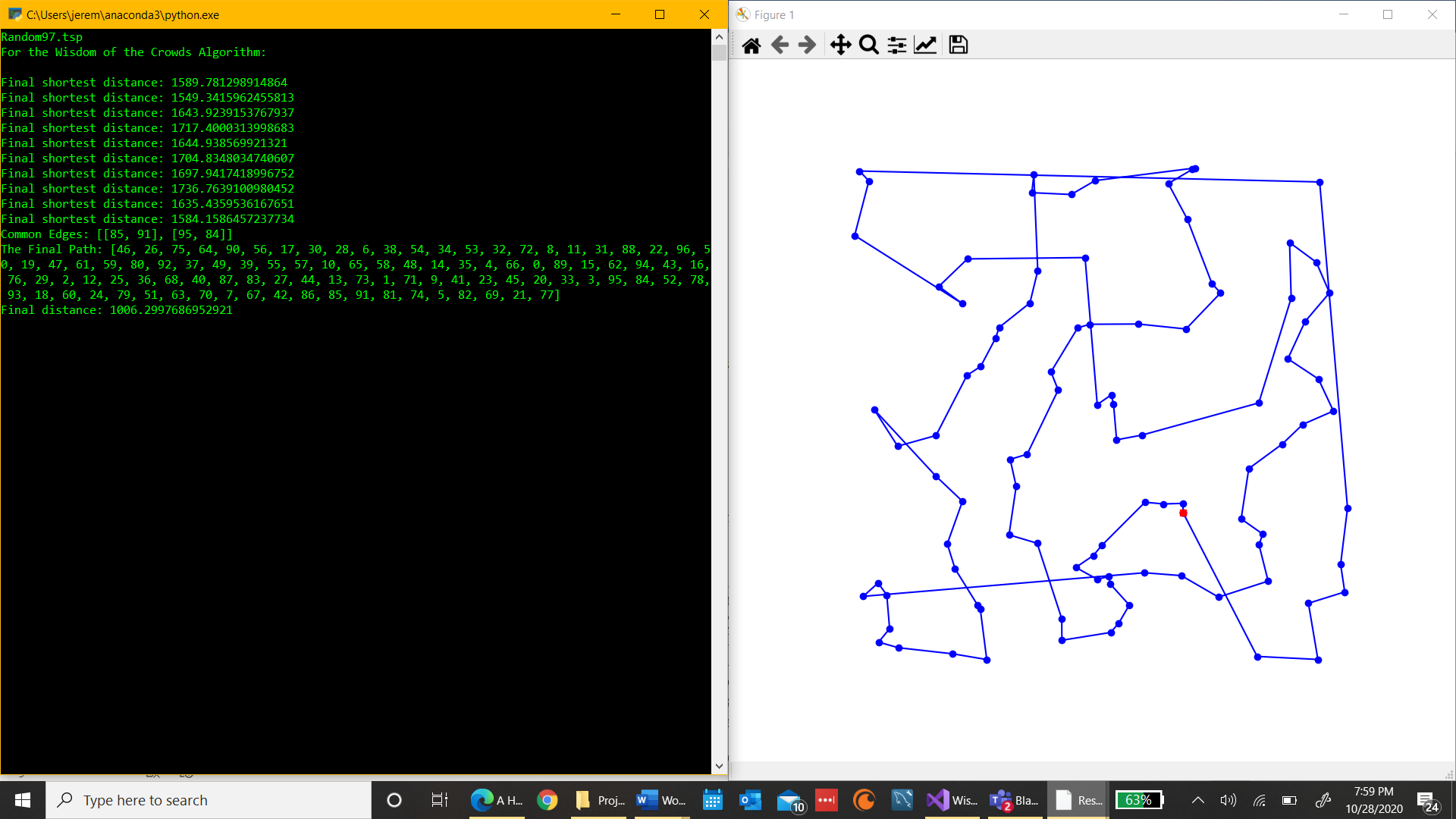
**Random44.tsp Example:**



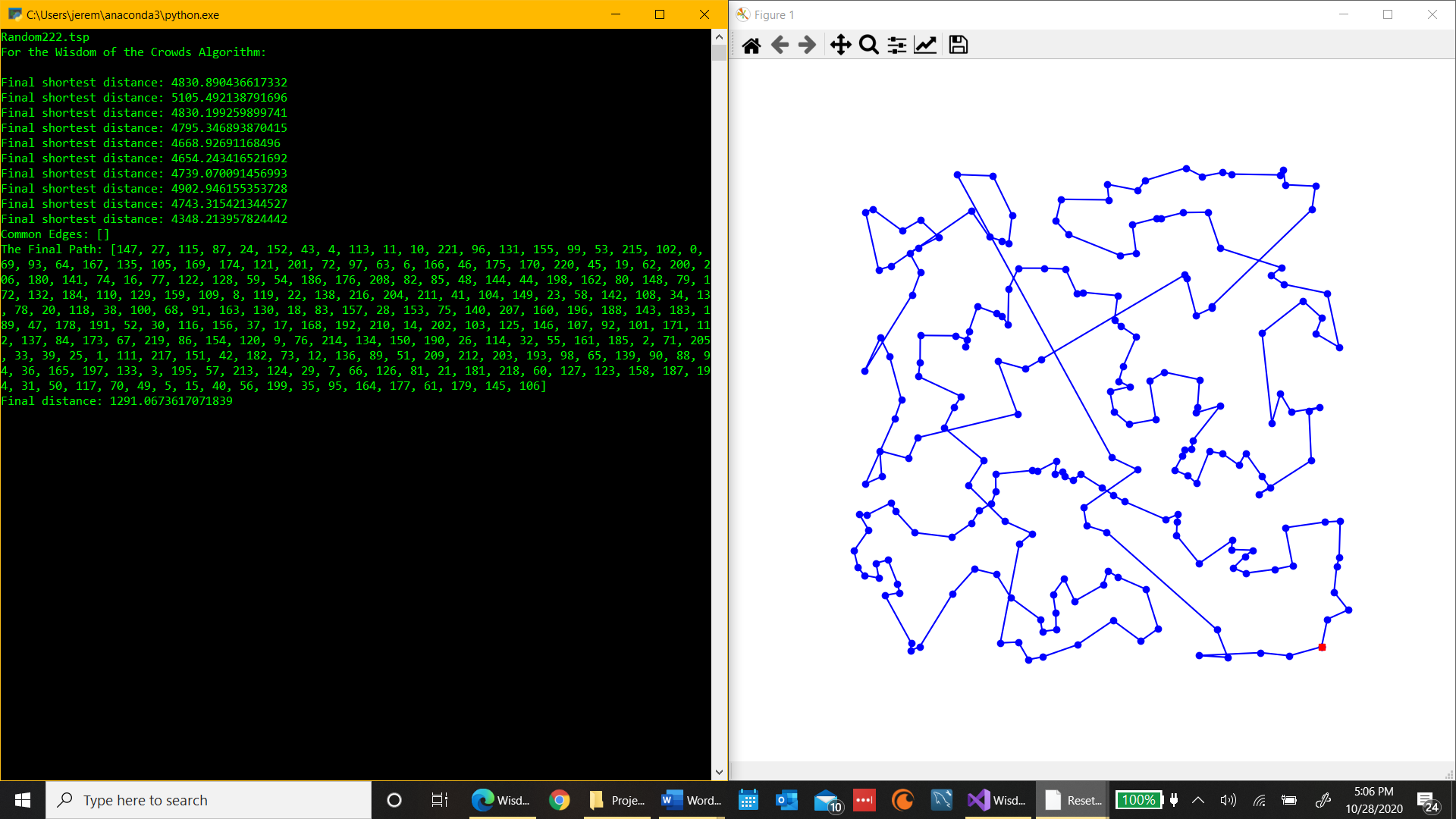
**Random77.tsp Example:**



**Random97.tsp Example:**



**Random222.tsp Example:**

-

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

I was able to generate good paths using my algorithm and for datasets with less cities I was even able to generate an optimal path. The above picture depict run throughs where the genetic algorithm only went through 10 times but that was just so I could have some examples to show. The table with statistics included was generated using run throughs with 20 GA solutions.

You can also see with datasets like the one with 222 cities no common edges were found in majority of the solutions so the program relied on Nearest Neighbor to create a final solution. Which performed much better than the genetic algorithm itself and same with 97 cities only a couple common edges were found so it relied heavily on Nearest Neighbor to create a valid TSP solution. But the 11 and 22 cities datasets both generated optimal paths and had a lot more matching edges. Their generated solutions by the GA also seemed to be a lot closer to each other. Keep in mind these results differ greatly between run throughs as sometimes even in the 11 and 22 cities I only had one or two matching edges. Results can be very inconsistent and quite sporadic.

On average compared the Wisdom of Crowds approach performed better compared to the standard unenhanced GA. Because most of the time the distance of the WOC was less than the average distance of the GA. Now the GA was always able to generate a path that was less than the final WOC result (unless there were no matching edges) but there’s no guarantee you’ll get a result that good every time. There’s also no comparison between the times each part takes because the GA takes a lot more computation than the WOC does.

The size of the problem definetly makes a huge difference for this algorithm because with smaller amounts of cities there’s less routes to take. But with 222 cities which is the largest dataset I tested, there’s so many different paths that can be generated it’s much harder to find common edges regardless of how many times the GA is ran. That’s why the lower the number of cities the higher chance of their being a lot more common edges to be used in the Wisdom of Crowds approach. But WOC is like GA in the fact that you could run each dataset 100 times and you’ll get 100 different solutions which could differ greatly.

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