**Project 6**

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 tasked with implementing the hybrid Genetic Algorithm with Wisdom of Crowds that I designed within Project 5. To solve an NP complete problem which is a problem that cannot be solved in polynomial time in any known way. For this project the problem I chose was partitioning, which is the task of deciding whether a given set of positive integers can be partitioned into two subsets such that the sum of the numbers in the first subset equals the sum of the numbers in the second subset. Or if there’s no combination of two subsets that will be equivalent the difference between the two sums should be minimized.

The approach I took to solve this problem was using the hybrid algorithm that combined genetic algorithm with a Wisdom of Crowds approach. And repurposing the various functions in it in order to work for partitioning. 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 genetic algorithm part of this function can give you a diverse array of results but the Wisdom of Crowds part isn’t as effective this time. It’s difficult to find multiple identical subsets especially the larger the set of numbers used is. If similar subsets are found then they will automatically be the final solution. But if none are found then a greedy algorithm is used to generate a final result. When it comes to the genetic algorithm part of the program results vary greatly sometimes the initial partition already creates two subsets with a difference of 0 and then the genetic algorithm provides a worse solution other times it greatly improves off of the initial solutions.

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

The programming language I used for this project was Python. Solving a problem other than TSP was definetly a nice challenge and I enjoyed redoing my code to try to solve it. To build my algorithm I started by importing the math, collections, random, operator, pandas, and numpy libraries. Then I brought in my whole GA with WOC algorithm. And just deleted things I wouldn’t need for this problem like any distance formulas or tables that received data from files.

To get started I began by creating various variables that would hold the maximum amount of numbers in the initial set and a maximum and minimum number limit. I then randomly generated the numbers that would make up my set of numbers and they would all be distinct numbers. I also ran test with groups of numbers I knew had subset combinations that would be equivalent which I will talk about later. After this I began repurposing the functions that made up my genetic algorithm, by looking in the main genetic algorithm function and backtracking through the code. So I could change all the functions to work with partitions instead of cities with x and y coordinates.

I started by changing the initial population function to within it call the partition function which was my generate route function from my previous project repurposed to take in a list of numbers. Then within the partition function based on how many numbers is in a list a solution is represented by a binary string, parallel to an array of the input numbers. Such that each character in the binary string represents one number in the input array. Each number in the array is represented by the character in the binary string whose position in the string is the same as that number’s index in the array. For each binary character in a candidate solution, if the character’s value is 0, the corresponding number is a member of the first subset in the partition. Otherwise the binary character’s value is 1, and the number belongs to the second subset. This was how each generated individual would be partitioned, I will be referring to binary strings as chromosomes for the rest of the report. Initially a chromosome of all 0’s is generated then in a while loop numbers are randomly chosen from the number set and split into two subsets. Then I used a for loop to look at each number in the number set and if it was in the second subset. Using its index I’d change its value in the chromosome to a 1.

Next the genetic algorithm goes into a for loop that produces multiple generations based on my chosen value. In the loop a function called ‘nextGeneration’ is called. The first step in it was to rank the individuals by giving them a fitness score. To do this I took each chromosome in the population and iterated through each character seeing if it was a 0 or a 1 then using its index I appended the proper number from the set of numbers into its two subsets. Those two parts were then used in my changed fitness class which now accepts two list those being the partitions for an individual. And it calculates the fitness by computing the sum of the first subset and the computing the sum of the second subset. Then the fitness score is the absolute value of the difference between the two sums. Contrary to project 5 a lower fitness score was better this time because that means the difference between the two sums was closer to 0 which is what I want. This is done for every individual then I used a dictionary that would hold the index of a chromosome in the population and its associated fitness score and they were sorted by smallest to largest fitness score.

The next step was selection to create the mating pool, I didn’t have to change my function from project 5 which used 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. This also includes 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. This function uses the same table designed using the ranked partitions which held the partitions index and its 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 partitions which is always 1/5 of the number of numbers in the set. And then I use another for loop where a randomly generated 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.

Next it was time to generate children using the mating pool, in my ‘newPopulation’ function first the elite chromosomes are already added to the children. Then I switched to using **two-point crossover** instead of ordered crossover. Two-point crossover is when two indices of the binary string are chosen and each of the two parent chromosomes are ‘cut’ at those indices. For me, the first index was always 1/3 through the binary string and the other was 2/3 through the binary string. The offspring chromosome is created by taking the characters from the 0 index to the 1/3 index from the first parent. Appending the characters from the 1/3 index to the 2/3 index from the second parent, and appending the characters from the 2/3 index onward from the first parent. This process is done in a for loop to produce the amount children equivalent to the number of chromosome in the initial mating pool.

Next was to incorporate mutation into the mix in my ‘mutatePopulation’ function I used a **probabilistic bitwise mutation** instead of swap mutation. With probabilistic bitwise mutation for each character in the binary string, there is a 15% chance that the character will be swapped. For each swap, an existing 1 becomes a 0, and an existing 0 becomes a 1. This is implemented the same way as swap mutation from my previous project though, 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. This is done for every character in each chromosome thus creating a mutated population.

The whole process described above is ran for 500 generations and then the best chromosome is extracted and used to separate into two subsets. That show how the set of numbers were split up and the difference between their sums.

To create the wisdom of crowds part of the program I looped through the genetic algorithm a chosen amount of times and within the loops each subset from the resulting fittest solution is stored in a list. So, when the loop is finished executing there is a list that contained every subset from all the generated solutions. After this I created a set of nested for loops that compared every subset from every solution to every other subset in all the other solutions. And I created a count variable that would keep track of if it found the same subset within another solution. Then once one subset 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 ‘createFinalPartition’ that took in the set of numbers and the list of subsets to be included. Within the function it creates two list called ‘part1’ and ‘part2’ these list will hold the subsets that will make up the final solution. If there are subsets in the list of chosen subsets then those subsets will automatically become the final solution because if two common subsets make up majority of the solutions produced by the GA then they will be the only ones in the list. However if the list is empty than a greedy approach is used to create a final solution which goes back and forth from list ‘part1’ and ‘part2’ and the biggest number left in the set of numbers to each list until it’s empty. This wisdom of crowds part very solemnly works especially with large sets of numbers so usually it would be more beneficial to just use results produced by the genetic algorithm.

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. Within my genetic algorithm function I stored the best partition result each generation in a progress list then plotted the results in order to see how the results improved or worsened over time.

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

The algorithm performs well in an efficient amount of time for a lower amount of numbers but getting close to 100 numbers it starts to take a while. (Which is mainly because of my computer because my code runs faster on other peoples computers.)

* 1. **Comparisons**

Demonstrating different input sets including 10 random integers in the range [1,20], 100 random integers in the range [1, 10000], and 500 random integers in the range [1, 10000].

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Input Set | Algorithm | Best Fitness  (10 GA Solutions) | Worst Fitness  (10 GA Solutions) | Average | Runtime |
| {4,5,6,7,8} | GA | 0 | 10 | 2 | 4 secs |
| WOC | 0 |  |  |  |
| {1,2,3,4,5,7} | GA | 0 | 6 | 1.2 | 4 secs |
| WOC | 4 |  |  |  |
| 10  in  [1, 20] | GA | 1 | 1 | 1 | 5 secs |
| WOC | 9 |  |  |  |
| 50  in  [1, 100] | GA | 0 | 22 | 10 | 46 secs |
| WOC | 38 |  |  |  |
| 100  in  [1, 1000] | GA | 0 | 110 | 55.2 | 2 min 40 secs |
| WOC | 556 |  |  |  |

\*Runtime refers to the time to do 1 run through of GA

All Dataset Parameters:

* # of cities/5 = elite size
* Generations = 500
* Probabilistic bitwise mutation = .15

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Input Set | Algorithm | Best Fitness  (25 GA Solutions) | Worst Fitness  (25 GA Solutions) | Average | Runtime |
| {4,5,6,7,8} | GA | 0 | 4 | 2.6 | 7 secs |
| WOC | 2 |  |  |  |
| {1,2,3,4,5,7} | GA | 0 | 12 | 3.8 | 7 secs |
| WOC | 4 |  |  |  |
| 10  in  [1, 20] | GA | 0 | 8 | 3.2 | 9 secs |
| WOC | 12 |  |  |  |
| 50  in  [1, 100] | GA | 1 | 27 | 7.8 | 1 min 10 secs |
| WOC | 55 |  |  |  |
| 100  in  [1, 1000] | GA | 56 | 240 | 145 | 4 min 20 sec |
| WOC | 534 |  |  |  |

\*Runtime refers to the time to do 1 run through of GA

All Dataset Parameters:

* # of cities/5 = elite size
* Generations = 800
* Probabilistic bitwise mutation = .15
  1. **Data** (Describe the data you used.)

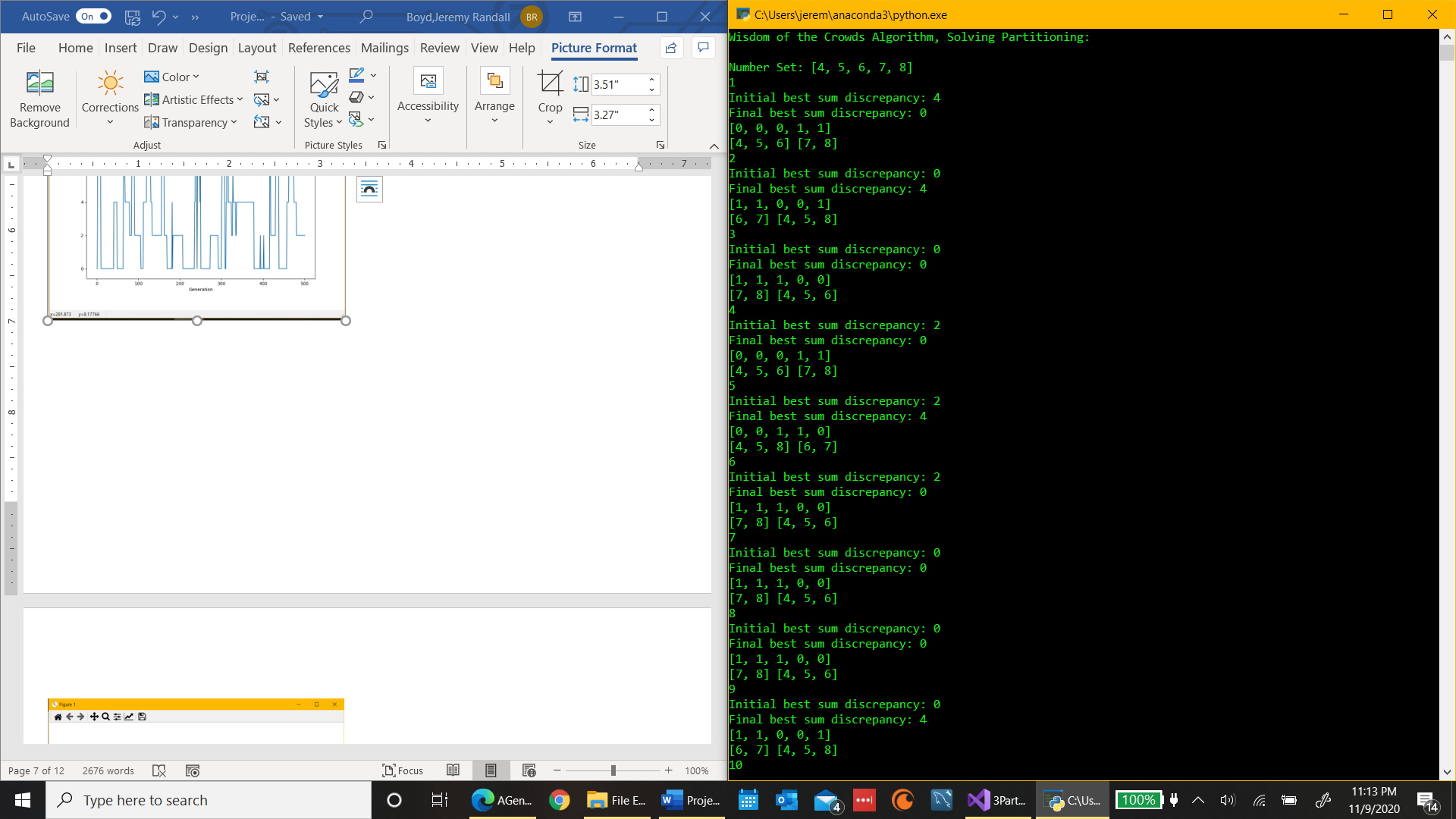
The first two data sets were numbers I came up with that I knew could create subsets with equivalent sums.

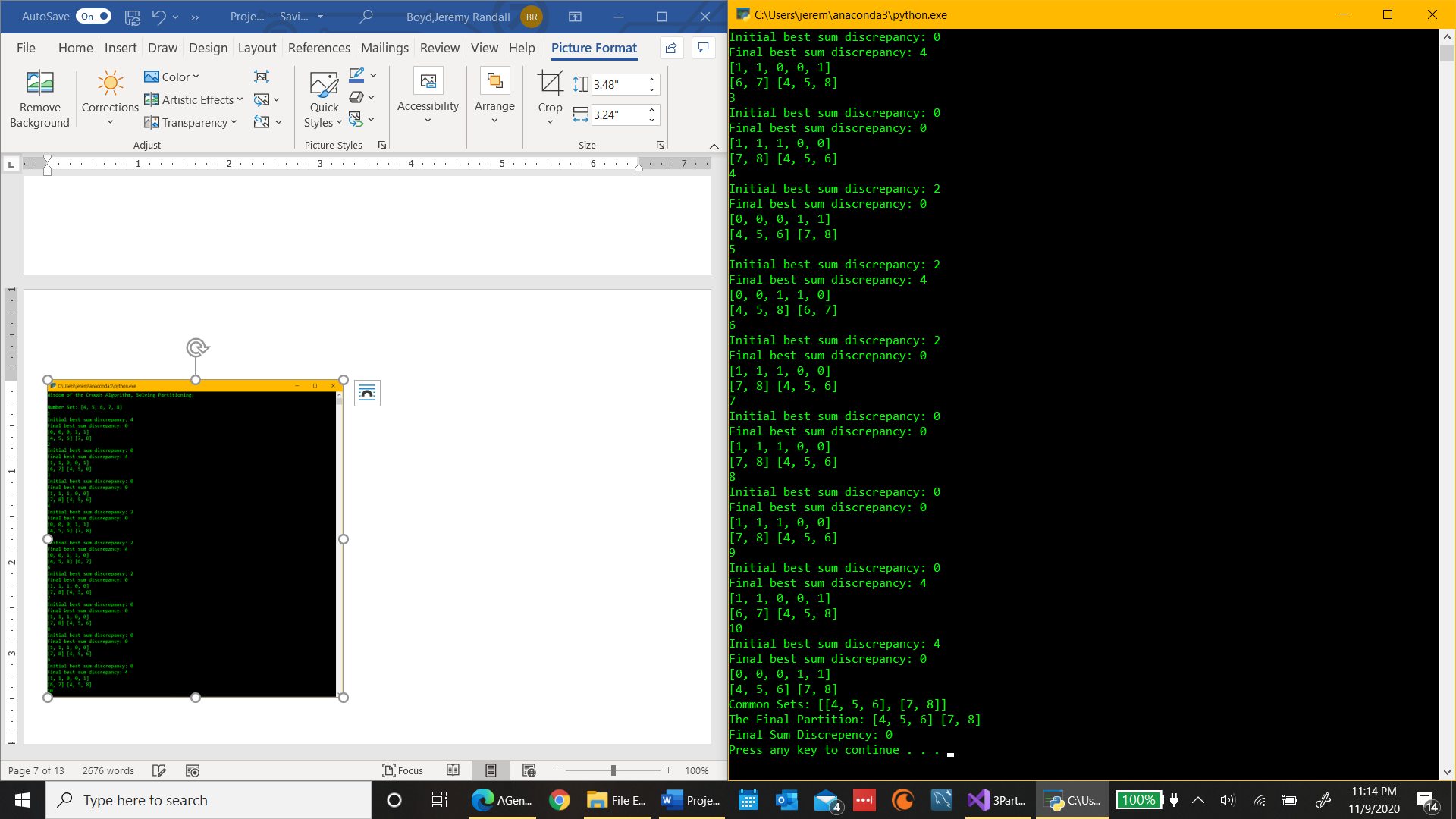
The last 3 data sets were numbers within a range that the program randomly generated on its own.

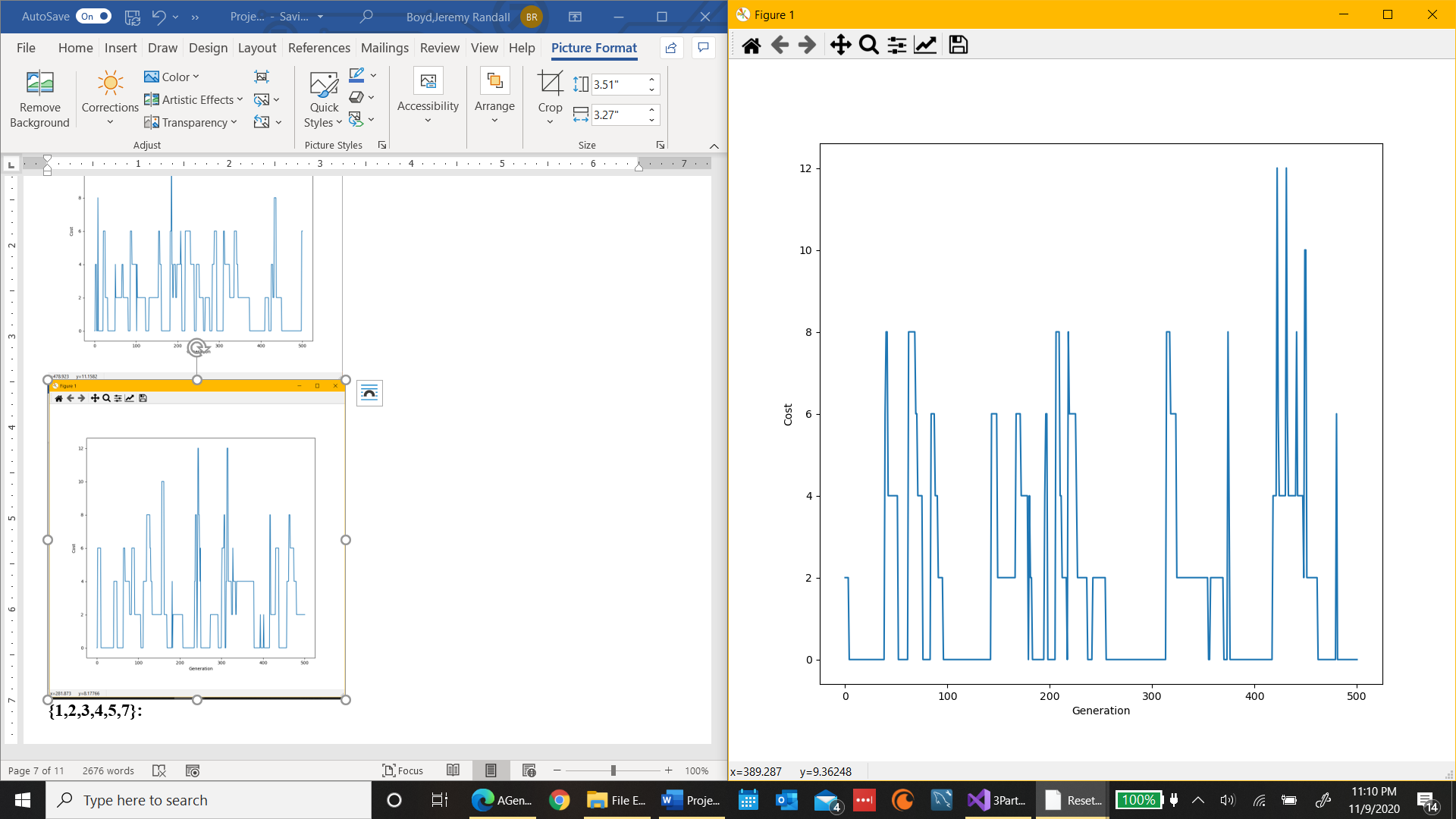
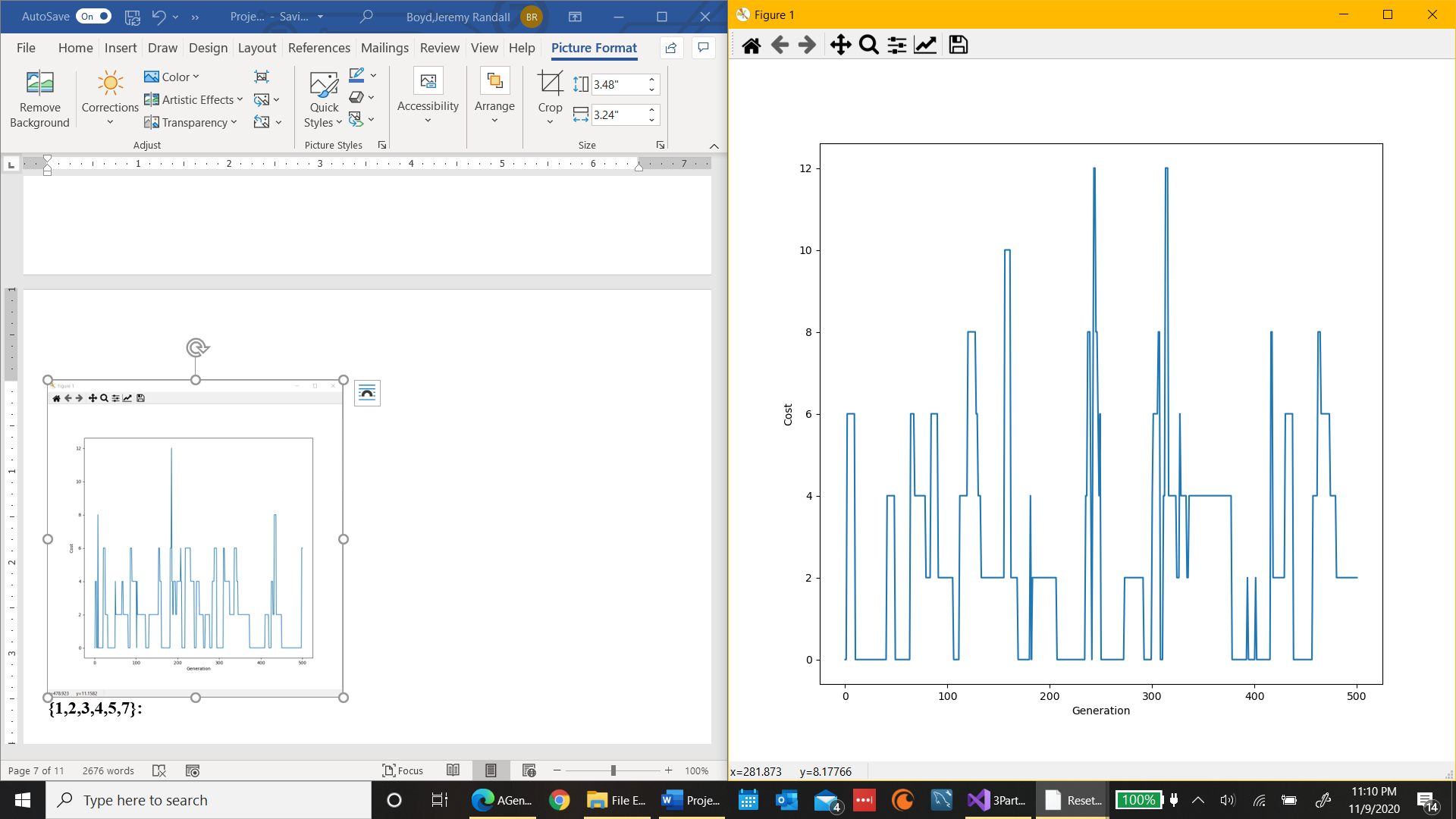
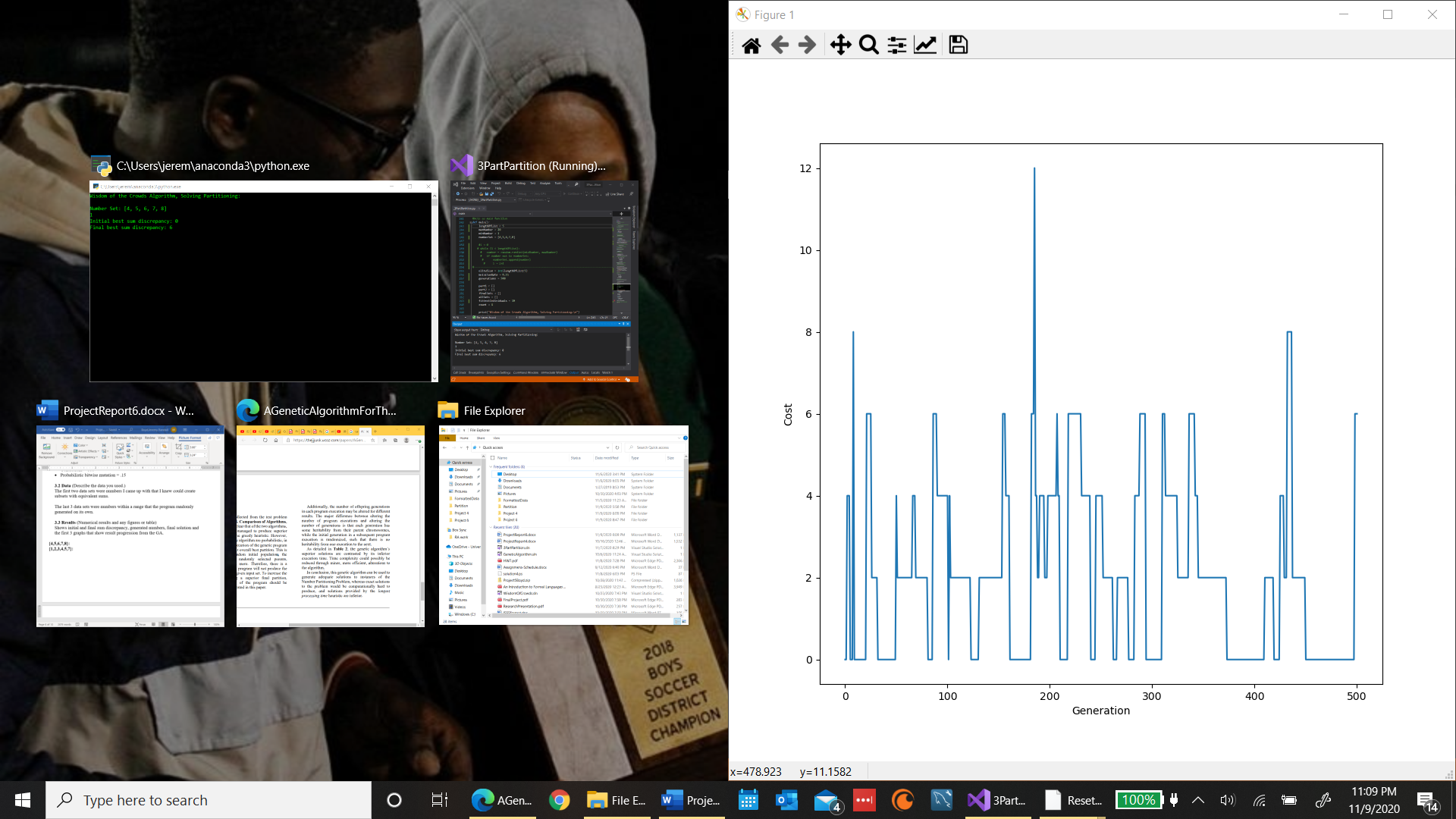
* 1. **Results** (Numerical results and any figures or table)

Shows initial and final sum discrepancy, generated numbers, final solution and the first 3 graphs that show result progression from the GA.

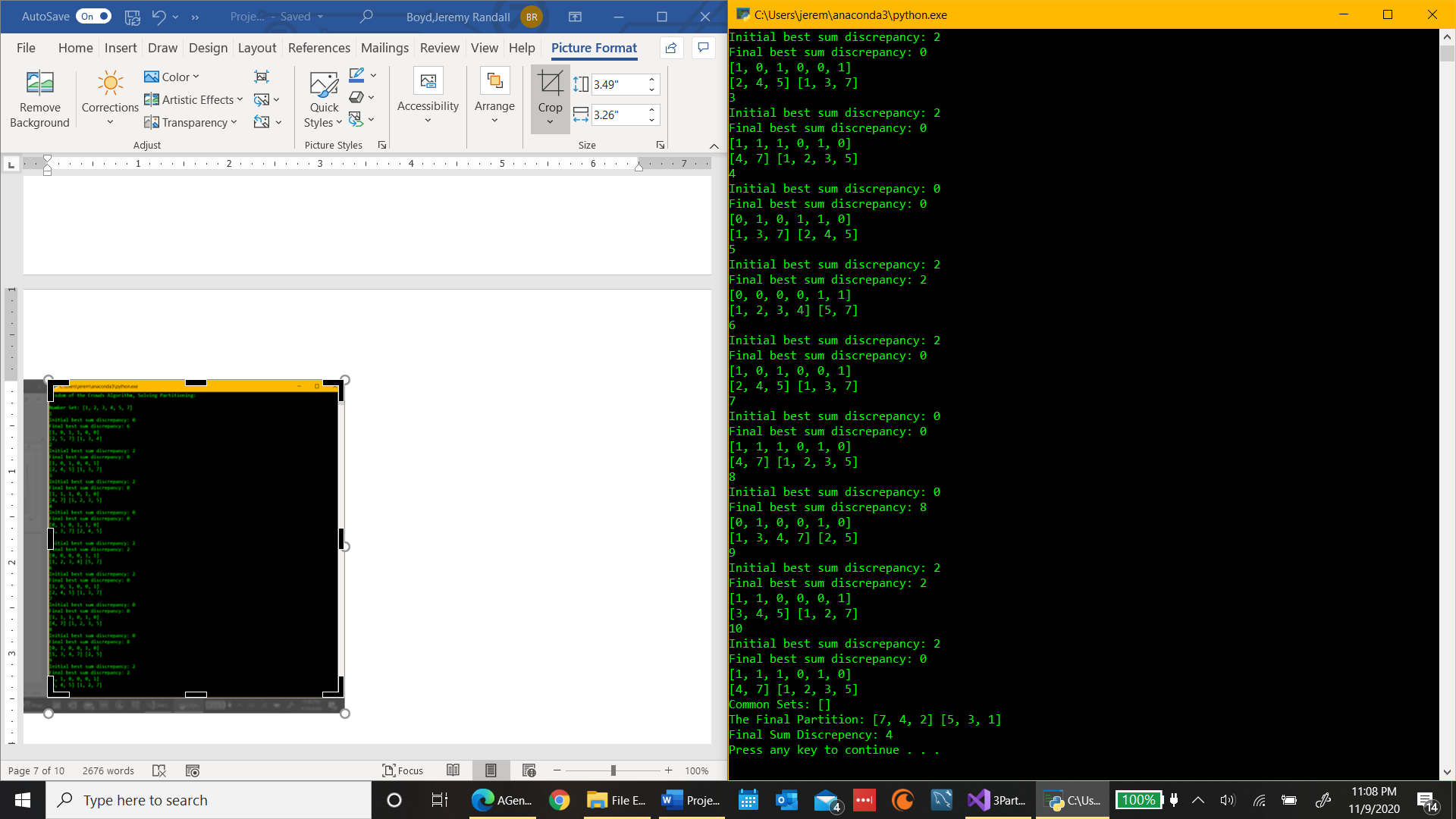
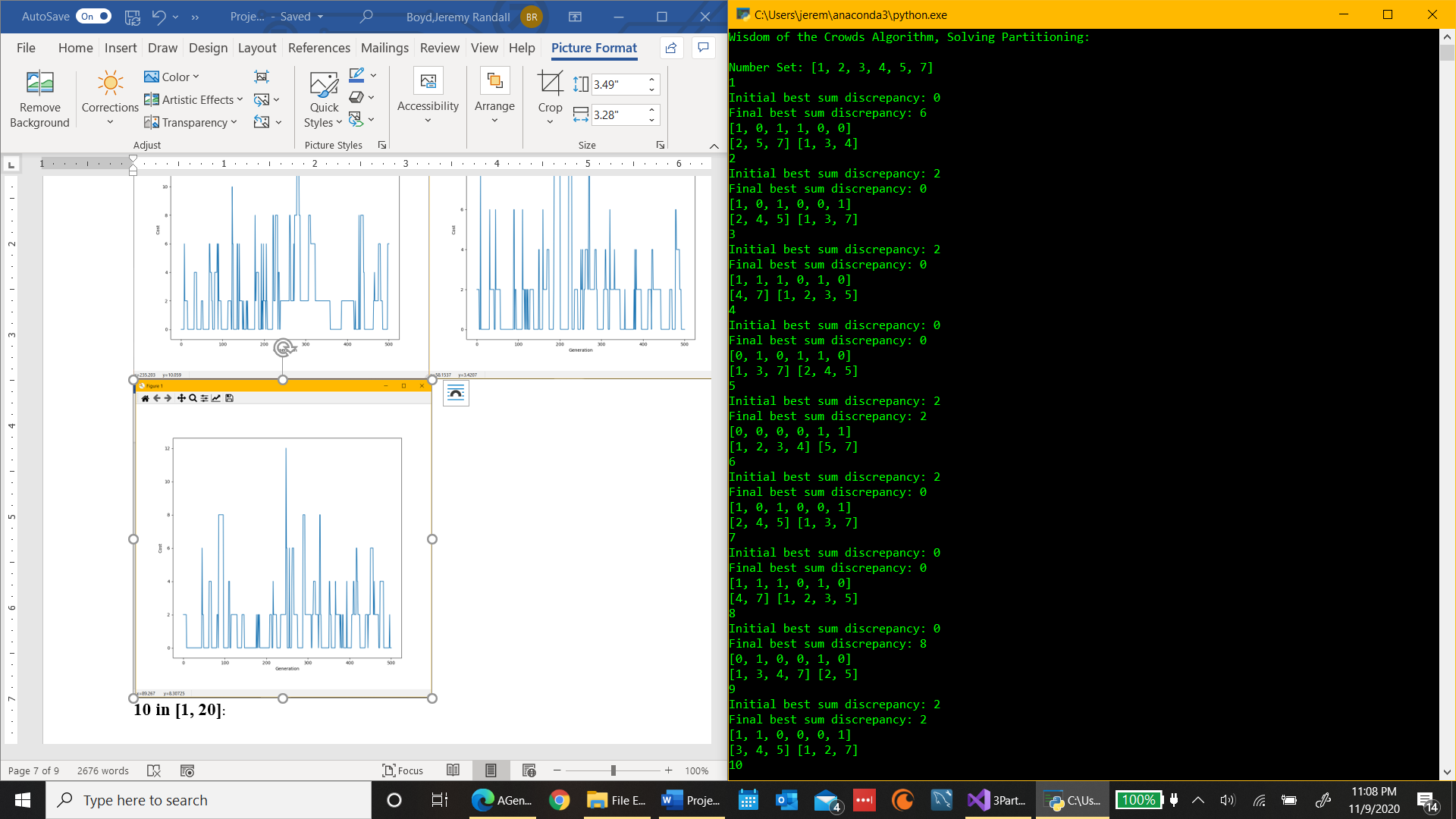
**{4,5,6,7,8}:**

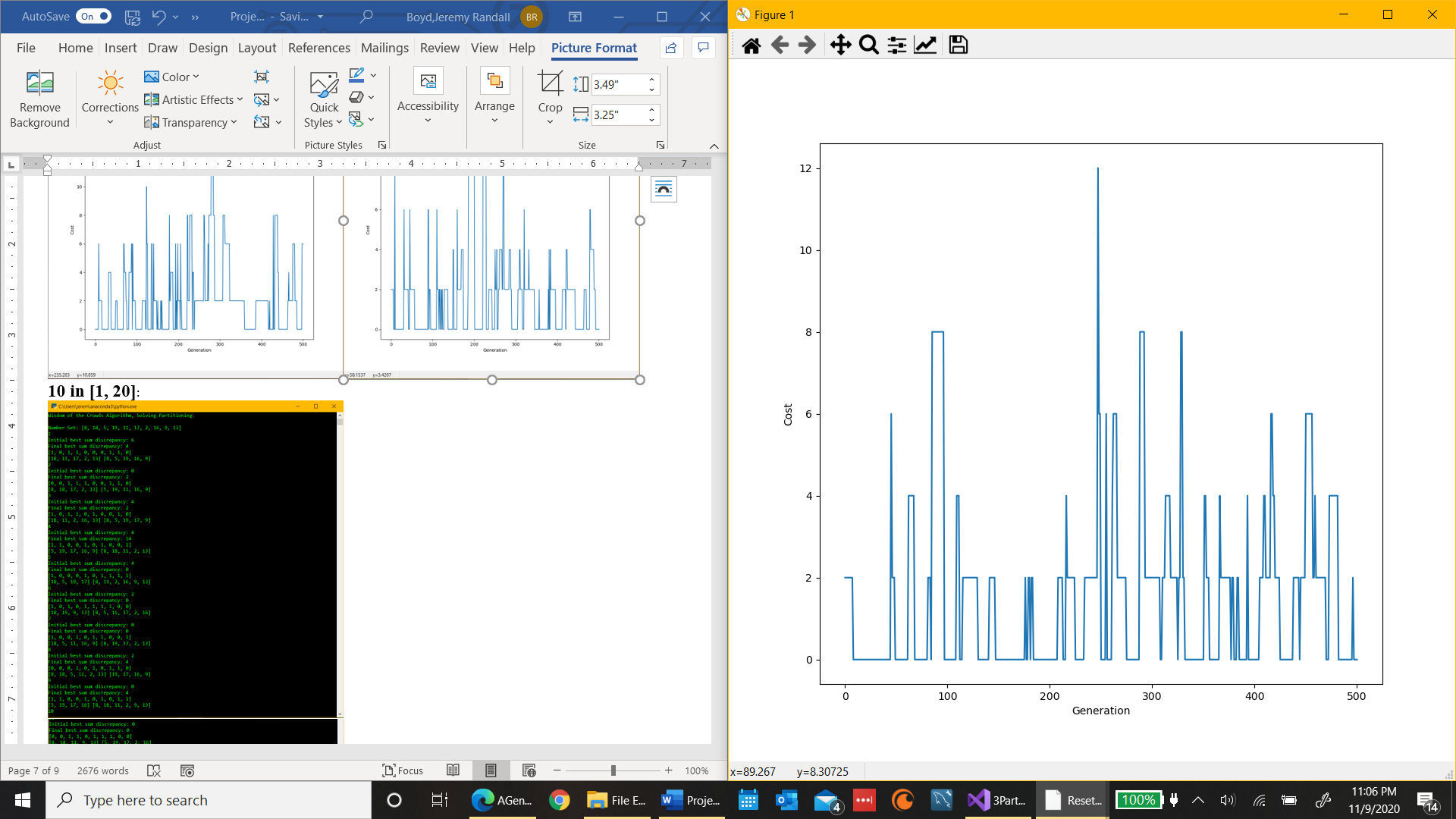
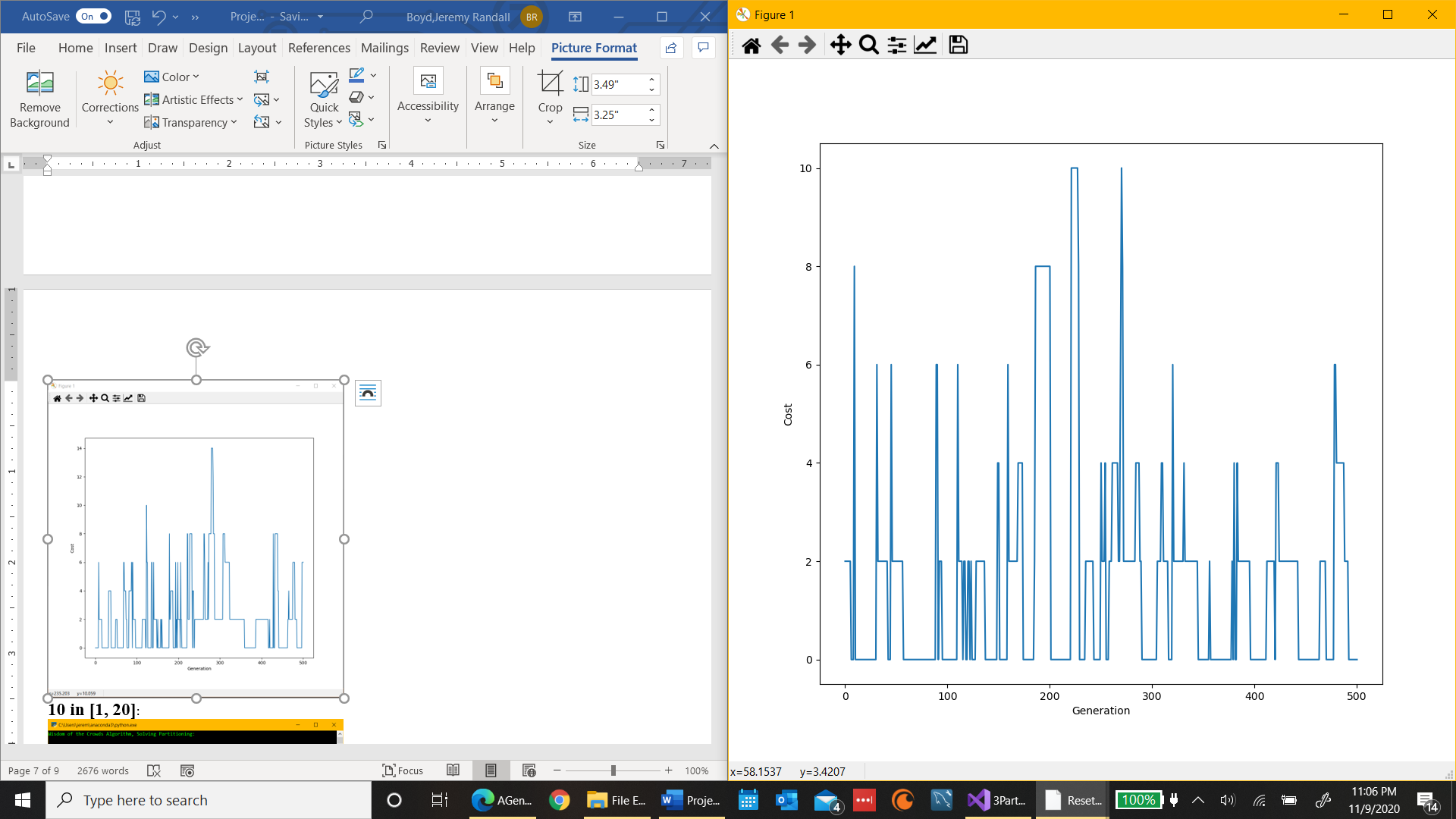
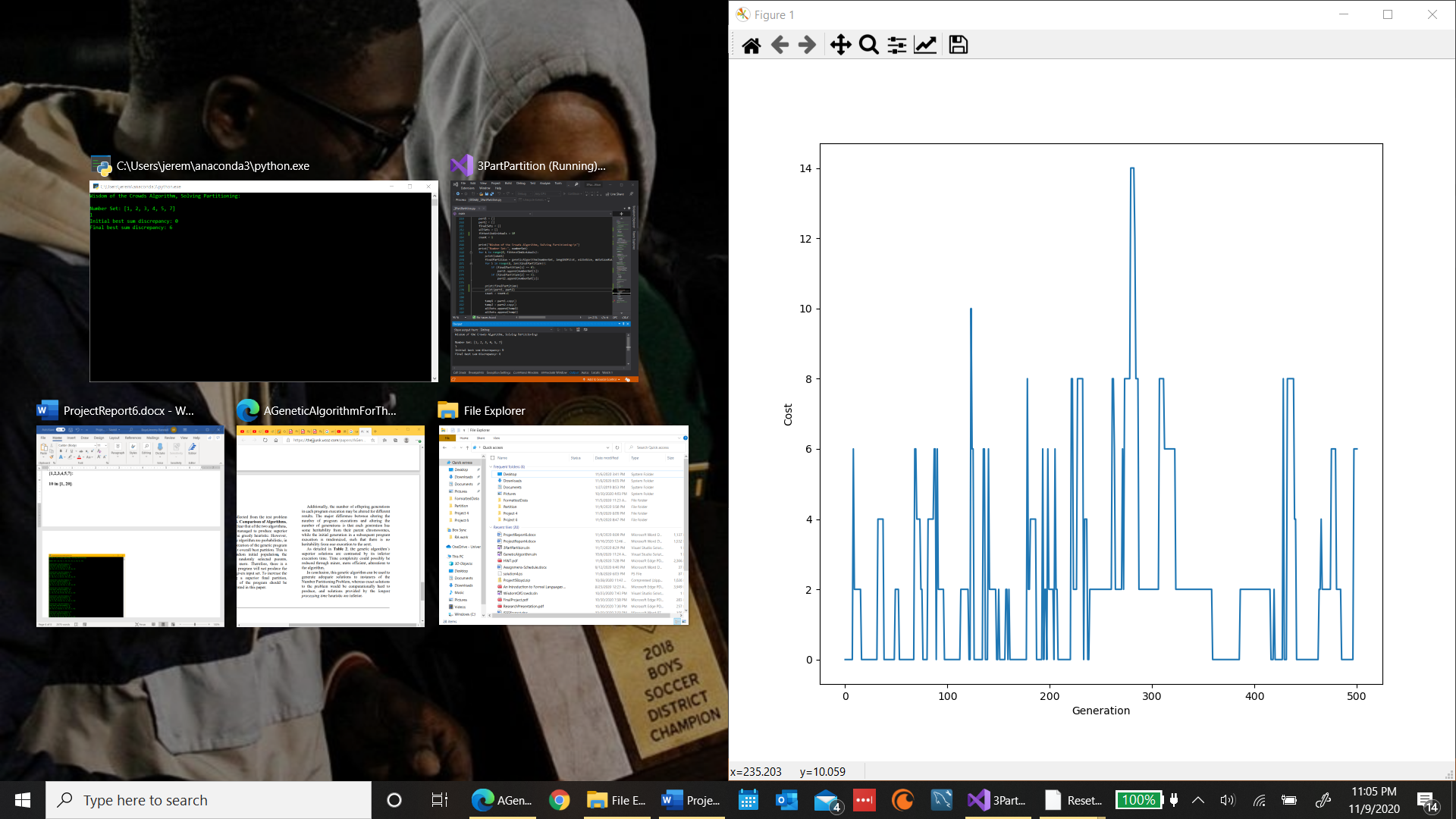




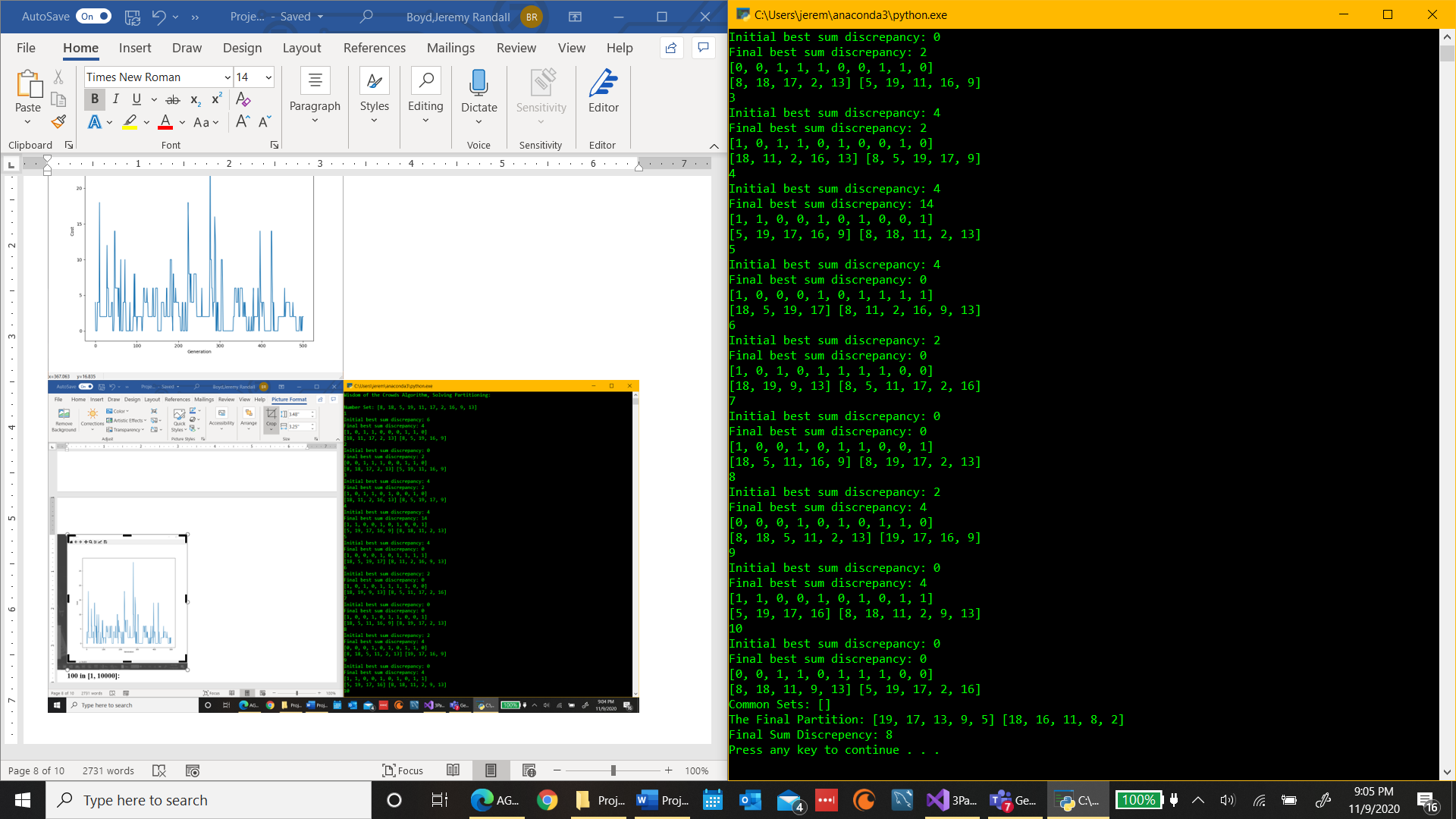
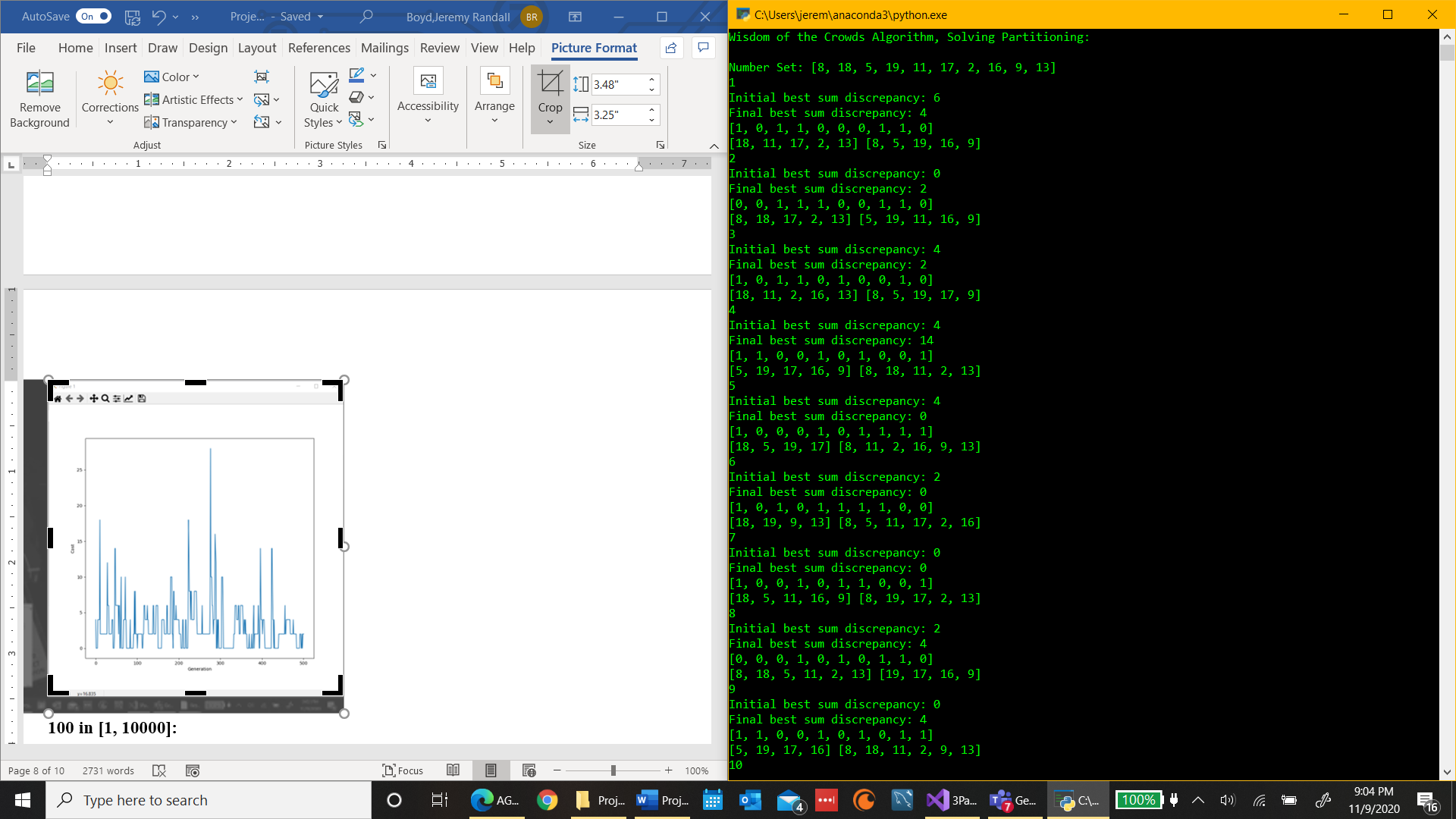


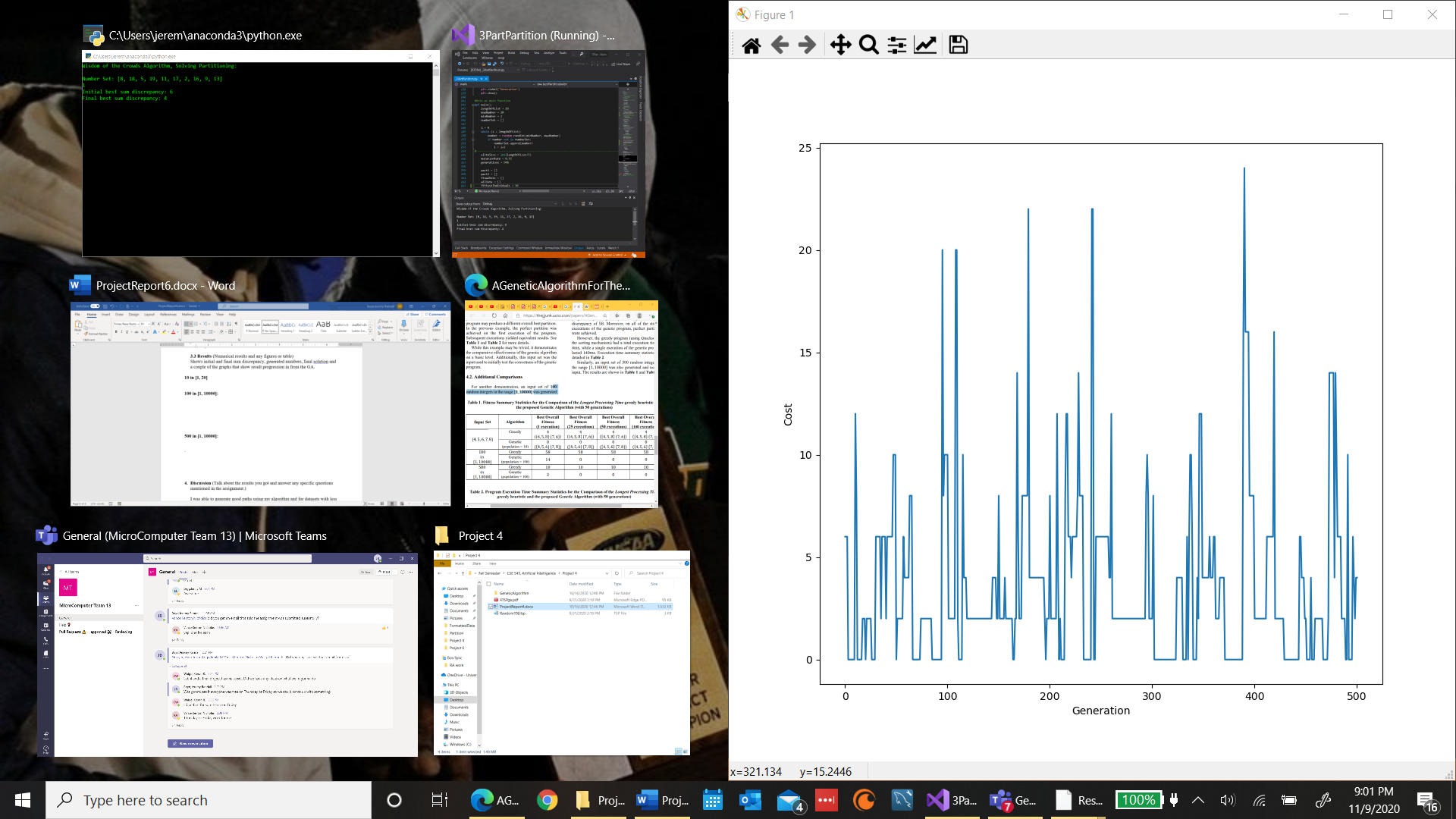
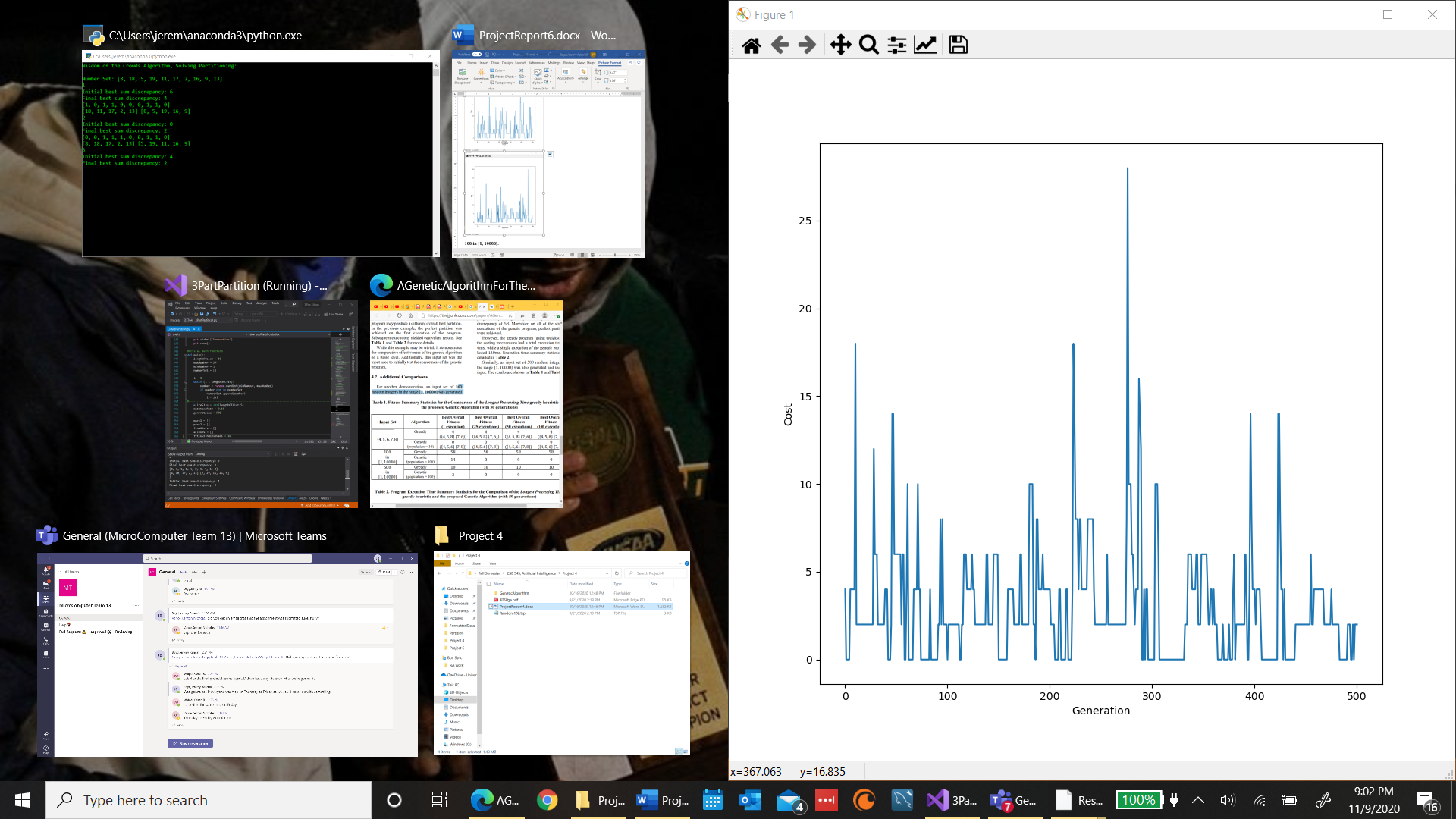
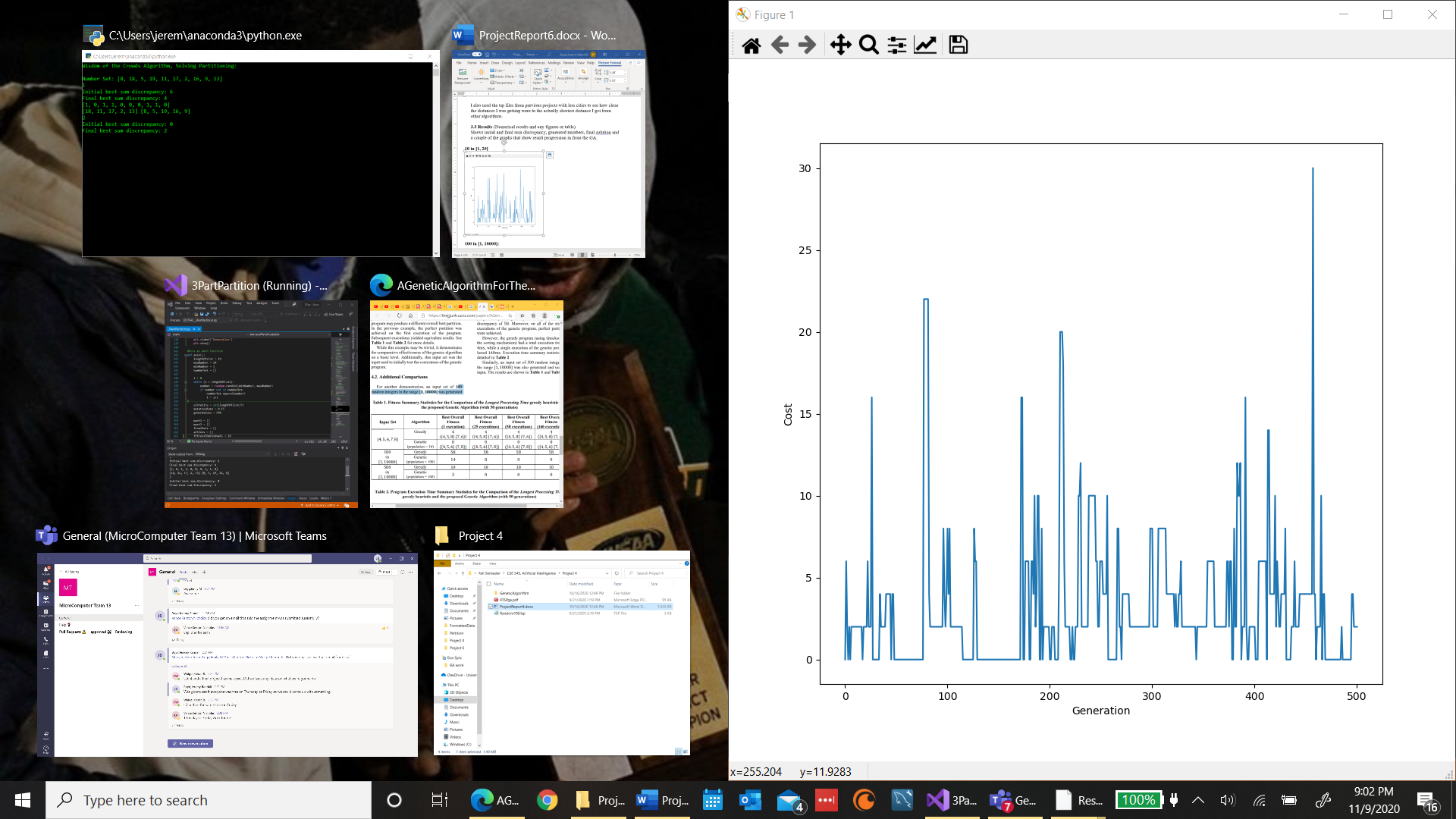
**{1,2,3,4,5,7}:**





**10 in [1, 20]**:



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

I was able to generate good solutions using my algorithm for datasets with less numbers, I was often able to find equivalent subsets using the GA. As I thought would happen though the Wisdom of Crowds part of the program wasn’t very effective. With very small datasets that I came up with it works and you always get the subsets with equivalent sums. But with larger datasets no subsets matched in majority of the GA solutions so it’s pretty ineffective. The higher the amount of numbers used in partitioning the more sporadic the final results were. They often greatly depended on the initially generated partitions if they were good the genetic algorithm would produce good subsets. However, if they were bad I often got numbers no where close to 0 especially in the dataset with 100 numbers.

I thought that increasing the number of generations would result in better solutions but I was mistaken. The results from the runs with 500 generations performed significantly better. Although randomness plays a huge part in the algorithm, I thought that the more generations that were produced the solutions would progressively get better but that didn’t happen.

The genetic algorithm was always able to generate equivalent subsets at least once for almost all the run-throughs though and the ones it didn’t I’m assuming there was no combination of numbers that would’ve resulted in equivalent subsets. The algorithm works well but it needs work to get better results more consistently something I could have figured out how to do but time wouldn’t allow. But on average larger datasets didn’t seem to perform that well with GA

The size of the problem definetly makes a huge difference for this algorithm because with smaller amounts of numbers there’s less options to choose from. And also the range of numbers was smaller as well which made it more likely to find numbers that could give you subsets closer to 0. 100 numbers is the larger dataset I tested and it rendered the WOC ineffective and gave very inconsistent results using the GA. There were more test I wanted to run to learn some other things about the algorithm but I ran out of time.

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.)