Alex Yang

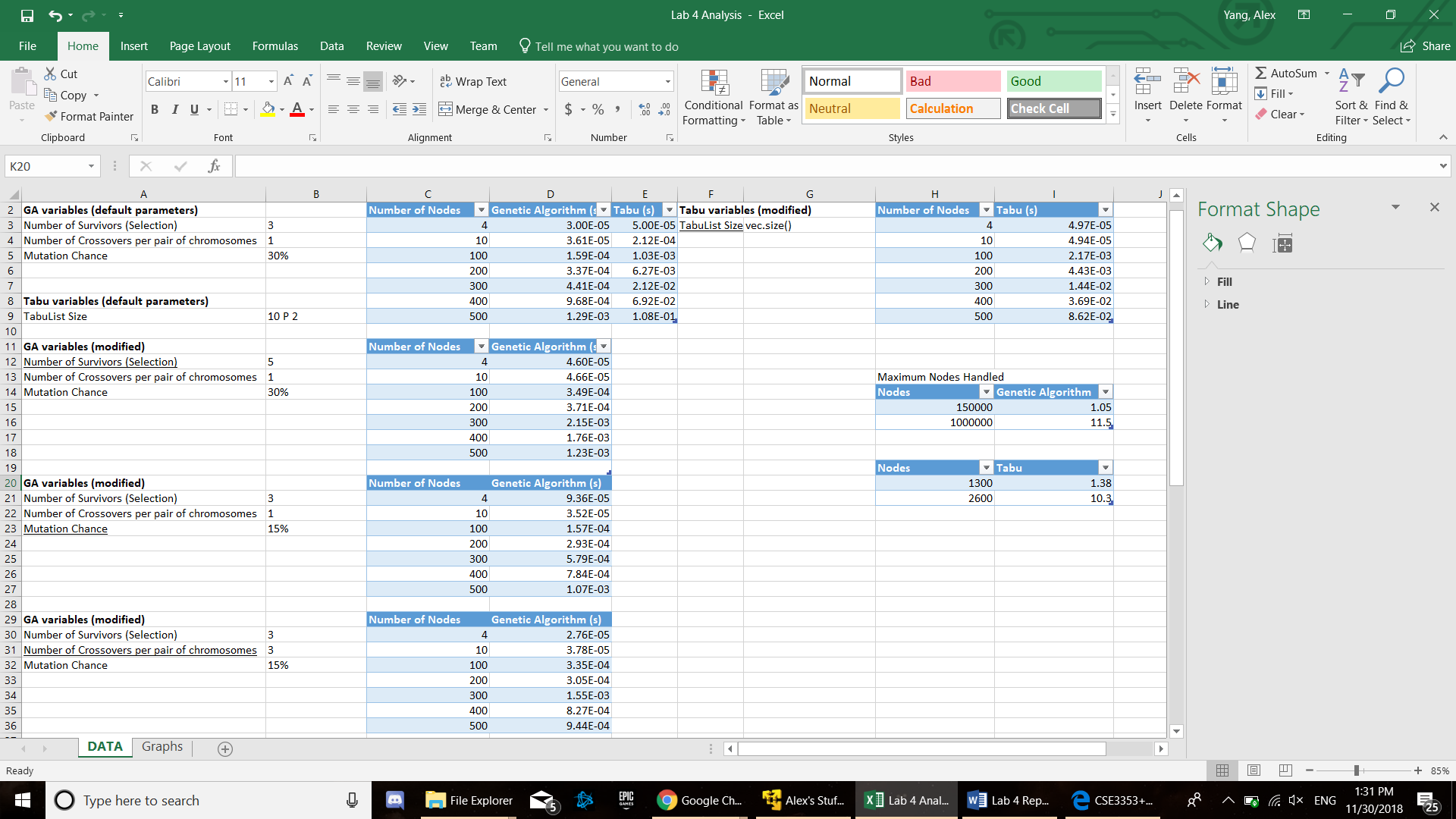
Professor Clark

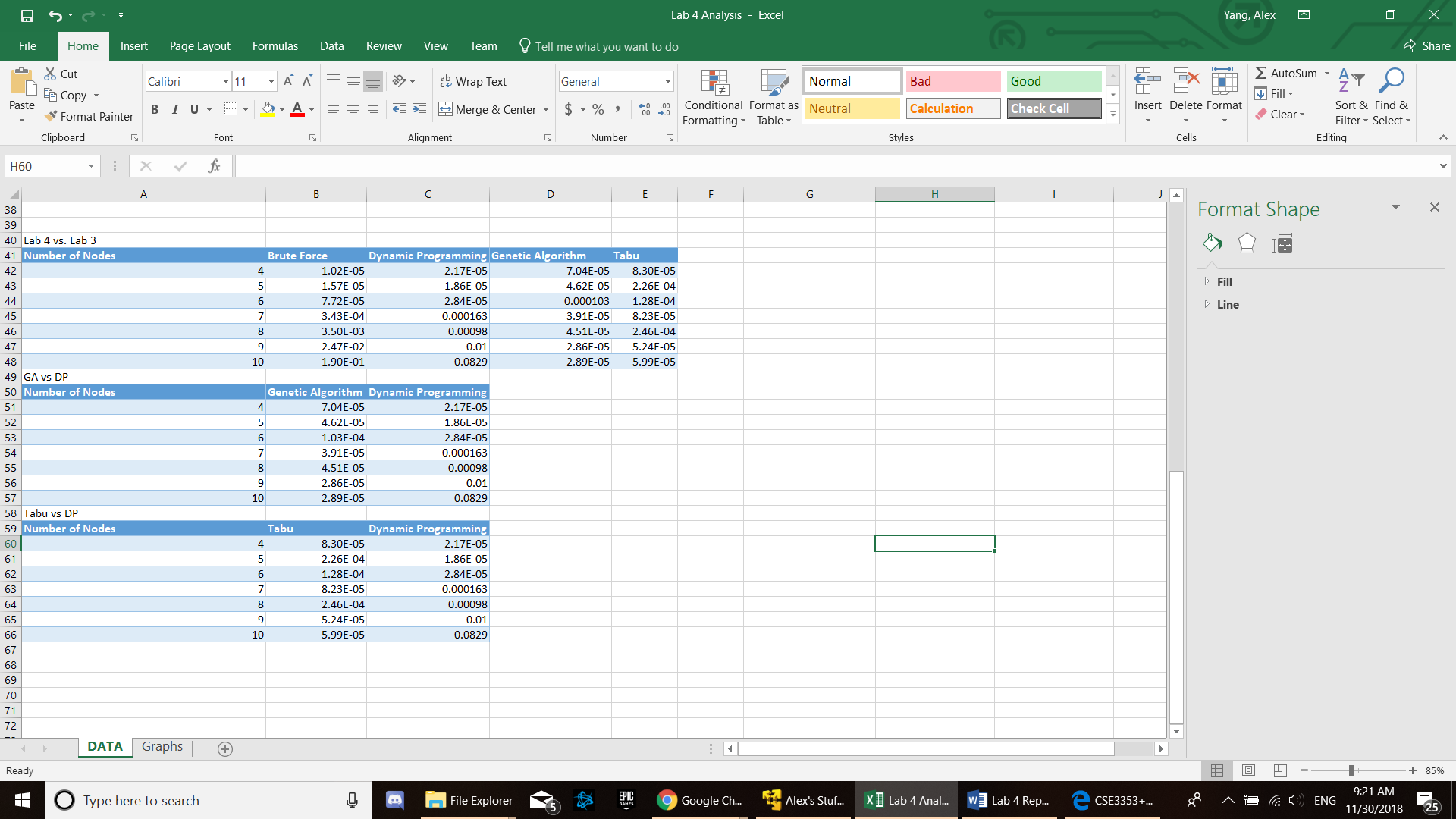
CSE 3353

29 November 2018

Lab 4 Report

Raw Data:





Graphs:

Analysis

Both my Genetic Algorithm and Tabu Search only ran a set number of times so that when the number of nodes was increased, there was not a huge difference in the total runtime (can’t match them with a big O notation. As you can see in my graphs, I could find a Hamiltonian circuit for 500 nodes in under a second. Of course, the downside is that neither of my algorithms would find the most efficient circuit, however they came close. Eventually my Tabu Search gets less and less efficient in comparison to my genetic algorithm. I think this is due to the fact that my Tabu Search uses has many more memory-intensive operations such as clear() and was overall less optimized than my genetic algorithm. On the other hand, my Tabu Search seems to more often find the more efficient path because it starts it’s initial state as the greedy path which is not a bad estimation. On the other hand, my genetic algorithm’s initial state is completely random which really messed up its accuracy as more nodes were added. I believe my GA was relatively accurate until the node count reaches past 10. Part of the reason why this algorithm becomes so inaccurate is because my genetic algorithm only simulates 5 generations no matter the node count. When I change my code to run through (population size ^ 2) generations, it got more accurate results up till 20 nodes and sometimes even beat the tabu search for shortest path. However, it also became many, many times more inefficient when I run it through that many generations.

Compared to the Brute Force and Dynamic Programming algorithms from lab 3, the GA and Tabu Algorithms were much more efficient as the number of nodes increased. This can easily be seen with how the Lab 4 Algorithms can easily handle 500 nodes while Brute Force and Dynamic stopped before 20. This is due to the fact that while Brute Force and Dynamic Programming algorithms try to find the absolute best path, the GA and Tabu Algorithms settle for approximations and “good enough” paths which allow them to run much faster (local bests) because they are not testing every single path.

In my testing I found that my Genetic Algorithm can handle up to 150,000 nodes before it takes over 1 second. It can run through a list of 1,000,000 nodes in just over 10 seconds. However it becomes inaccurate after 10 nodes.

My Tabu search on the other hand can only handle 1300 nodes before it takes over 1 second and can run through a list of 2600 before it takes over 10 seconds. However, it is much more accurate than the Genetic Algorithm.

Design Decisions

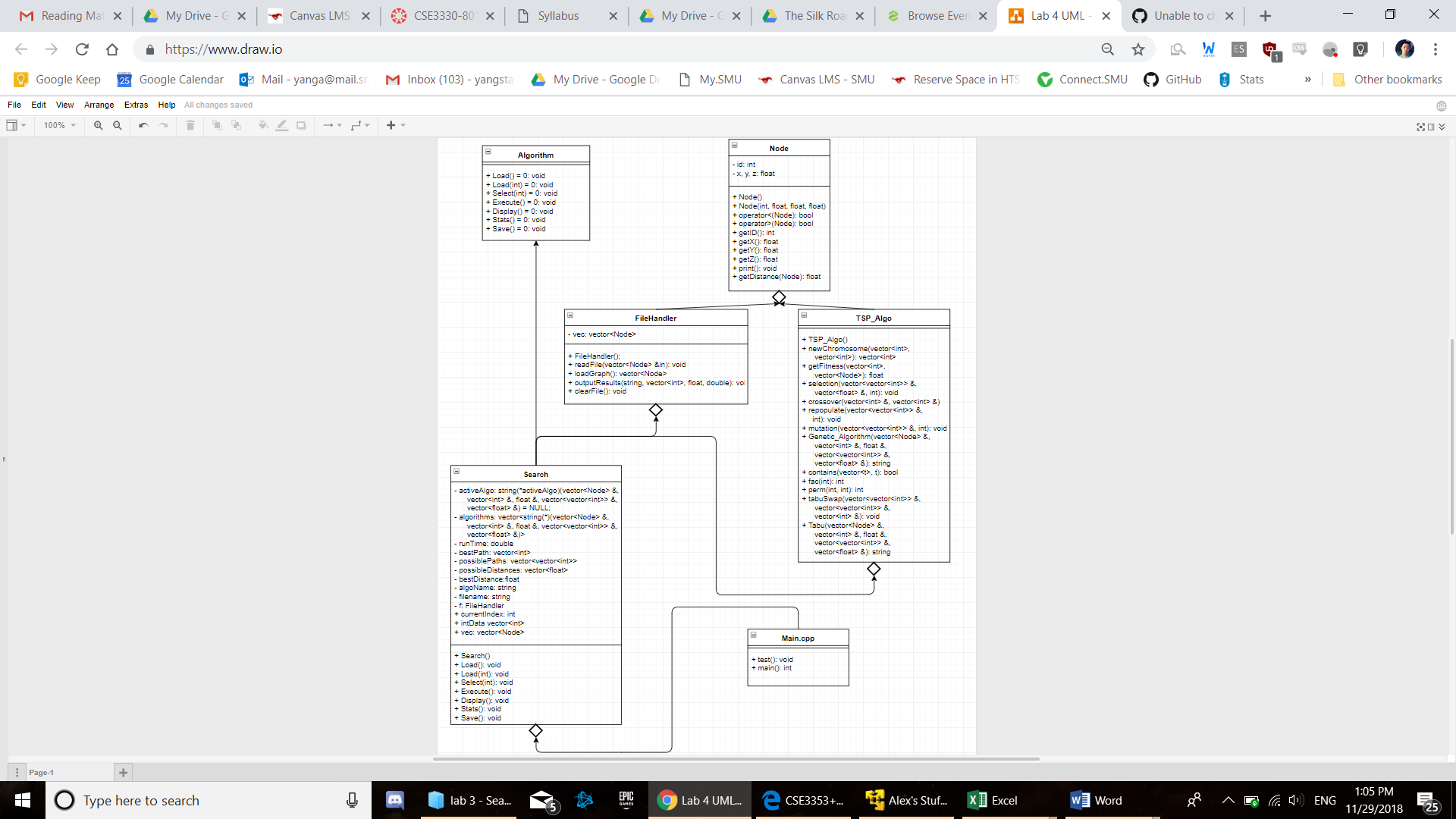
I used the strategy design pattern as my design pattern. Strategy seemed the most appealing to me because I can just write new algorithms in my SearchAlgo class and be able to easily add them to my Search class’s vector of algorithms. Because I’m using the strategy design pattern, I can also run every algorithm the same way as seen in my test function in my main.cpp. I can test multiple different sized graphs and see the results. All I need to do is call the load, select, and execute functions no matter which algorithm and it will run them. If I need to add a new algorithm, I can just write a new one in my TSP\_Algo class as a function.

To describe the structure of my program, my main class includes from Search which inherits from the virtual base class Algorithm. Search runs all the Strategy pattern commands (load, select, execute, etc.). Search includes TSP\_Algo which contains the brute force and DP algorithms in the form of static functions. Search has a vector of function pointers (algorithms) which it adds to from TSP\_Algo in my Search classes constructor. My graph is represented as a vector of nodes called vec contained in my Search class. Each node contains the X, Y, and Z coordinates of each point in the graph.

I also implemented a fileHandler object that incorporates both the file loader and output system into a single interface. The Load() and Save() functions in my Search class now call the fileHandler object and it takes care of reading and writing to files. In addition, if I were to re-use this program for a different type of algorithm with different text files, I could just change how the object reads in the input without touching any other parts of my code (so I know problem will be isolated to that class alone).

In addition, I had variables that control the number of neighbors that survive each generation in my genetic algorithm, variables that control mutation chance, and variables that control the max tabuList size so that I could see how changing those variables would affect the total runtime of each algorithm.

UML Diagram



Variations of GA and Tabu

The main variables I altered in my GA algorithm were the number of neighbors that survive each generation, the number of crossovers performed on each chromosome per generation, and the mutation chance of each neighbor every generation. As you can see from the tables in my raw data, when I increase the number of neighbors that survive every generation, the algorithm becomes less efficient and has a slightly higher total runtime when I used 5 survivors instead of 3. This means that the amount of survivors left after each selection is directly related to the total runtime of the algorithm.

When I increase the number of crossovers that occur per chromosome from 1 to 3 each generation, my algorithm’s run time barely changes, but it must increase ever so slightly. This is because when I perform more crossovers per generation, those extra crossovers add a tiny bit of time. The reason it is barely discernible is because my crossovers aren’t very resource intensive and it only adds two more per generation. Compared to other parts of my Genetic Algorithm such as Selection, crossover does not have a huge impact on the runtime.

In addition, when I decrease my mutation chance from 30% to 15%, my algorithm became slightly more efficient. This is due to the fact that a lower mutation chance makes it the algorithm call the mutation function fewer times, resulting in a faster runtime, at the expense of a potentially less accurate algorithm. This means that the mutation chance is inversely related to the runtime of the algorithm (Higher mutation chance = slower algorithm)

The variable I altered in my tabuList was the size of my tabuList. I originally have my tabu list’s max size set to the permutation of the number of nodes and 2:

(number of nodes)! / (number of nodes – 2)!

This made it so that the tabuList’s max size is equal to the amount of unique positions that can be swapped in the tabuList. When I decrease my tabuList’s max size to just the number of nodes itself (n), my tabu search becomes more efficient. This is because my tabuList fills up faster which means my tabu algorithm performs less swaps. This would result in a faster runtime at the expense of a potentially less accurate algorithm. This means that the max size of the tabuList is inversely related to the runtime of the Tabu Search.