Alex Yang

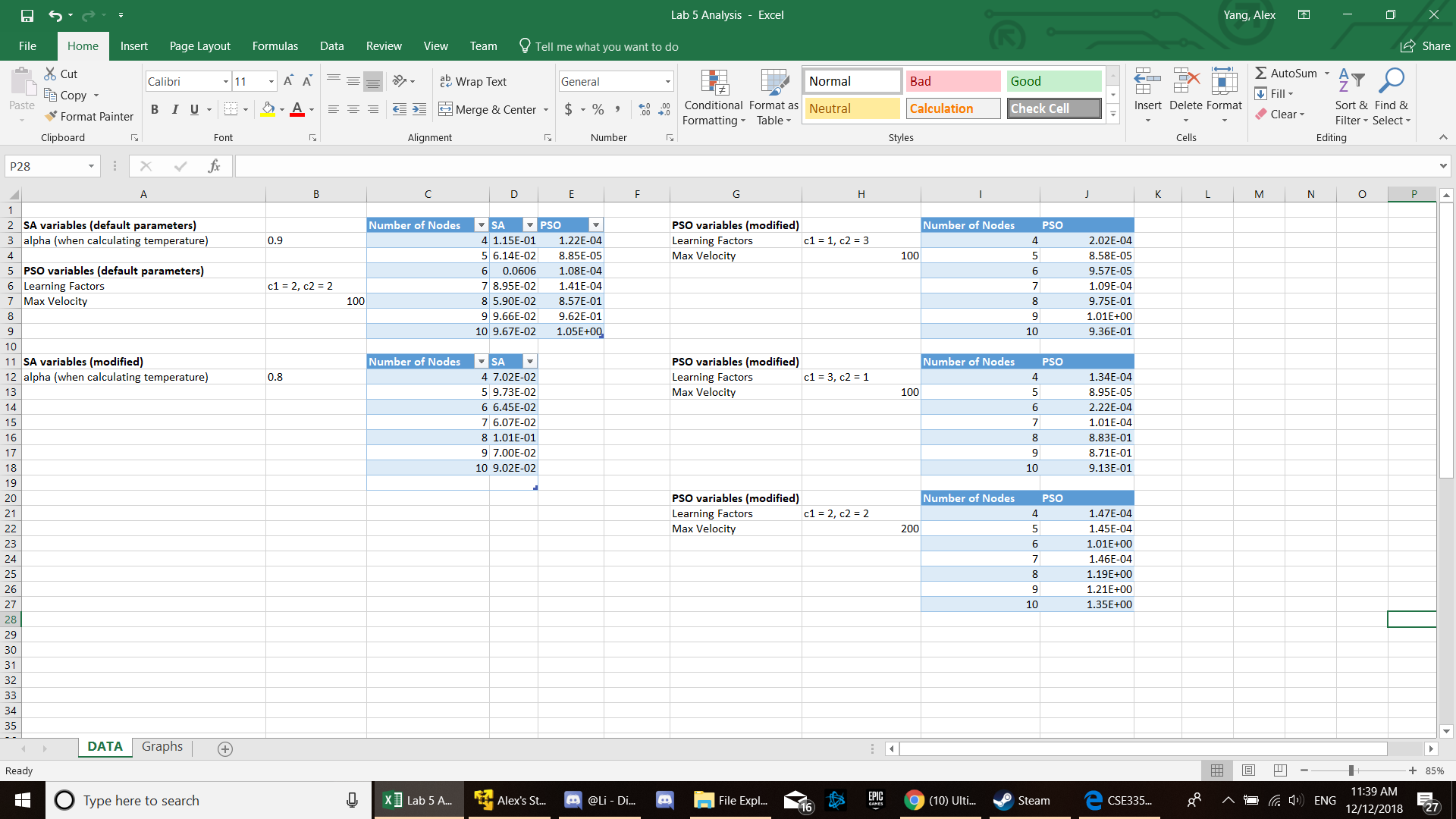
Professor Clark

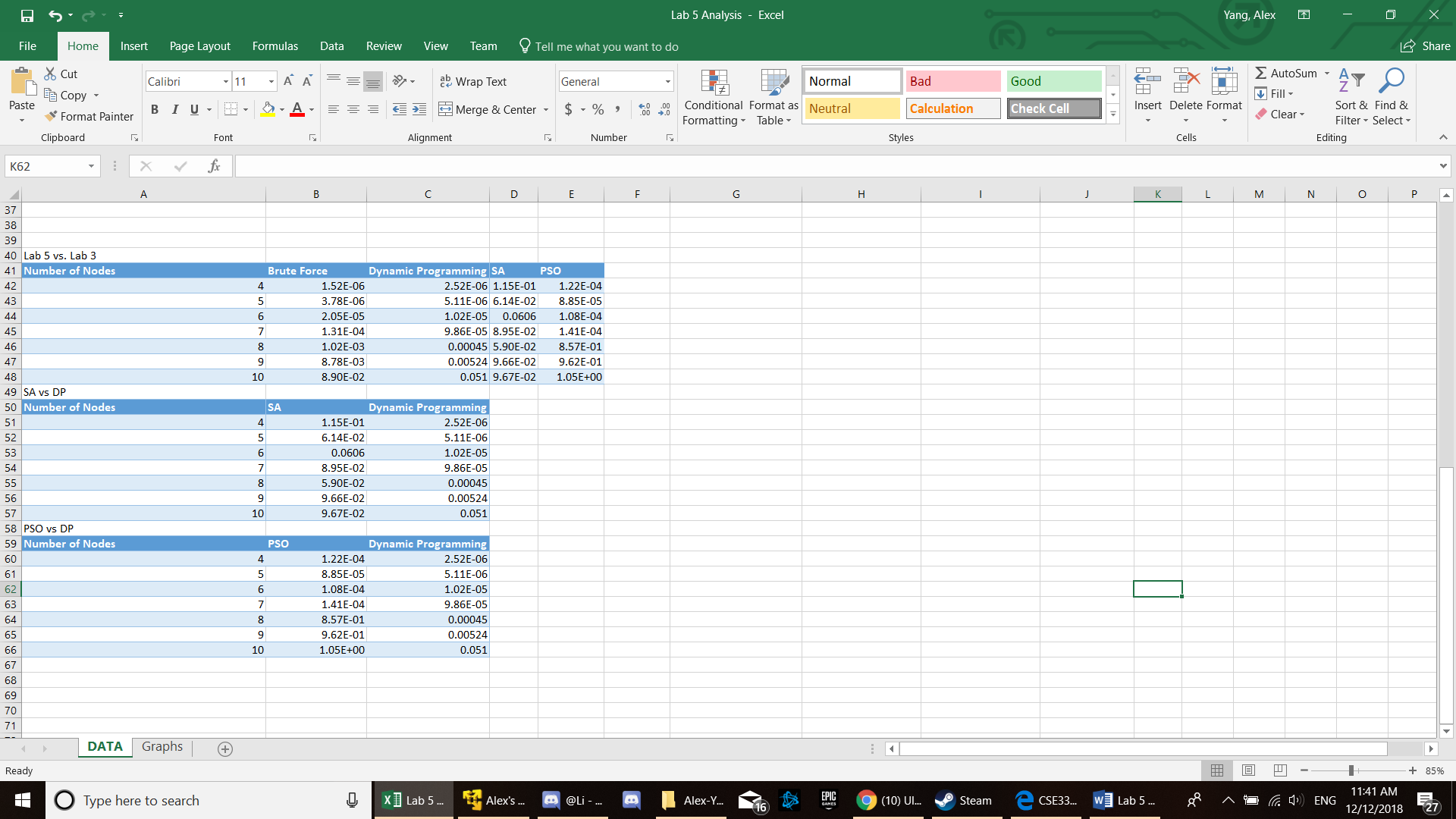
CSE 3353

11 December 2018

Lab 5 Report

Raw Data:





Graphs:

Analysis

Both my Simulated Annealing and Particle Swarm Optimization Algorithms ran until they reached the most efficient circuit, or they reached the maximum number of iterations (10,000). For my SA, As the number of nodes increased, there was not a huge difference in the total runtime (can’t match them with a big O notation). My PSO algorithm observed the same behavior before the node count reaches 7. It then makes an enormous leap at 8 nodes before maintaining an almost constant runtime. I think this is due to the fact that my PSO starts failing past 7 nodes and begins to return a local best solution rather than the global best (although the results are still very close to the best path).

A possible solution to the inaccuracy of my PSO past 7 nodes is starting the global best path equal to the greedy path. That way instead of using random shuffle, the global best is a pretty good path and it can improve from there on.

Compared to the Brute Force and Dynamic Programming algorithms from lab 3, the SA and PSO Algorithms were much more efficient as the number of nodes increased. This can be seen with how the Lab 5 Algorithms do not increase in time at nearly the same rate that brute force and dynamic programming do. They start off slower but as node count increases, they become more efficient. This is due to the fact that while Brute Force and Dynamic Programming algorithms try to find the absolute best path, the SA and PSO Algorithms settle for approximations once they reach the max amount of iterations.

Design Decisions

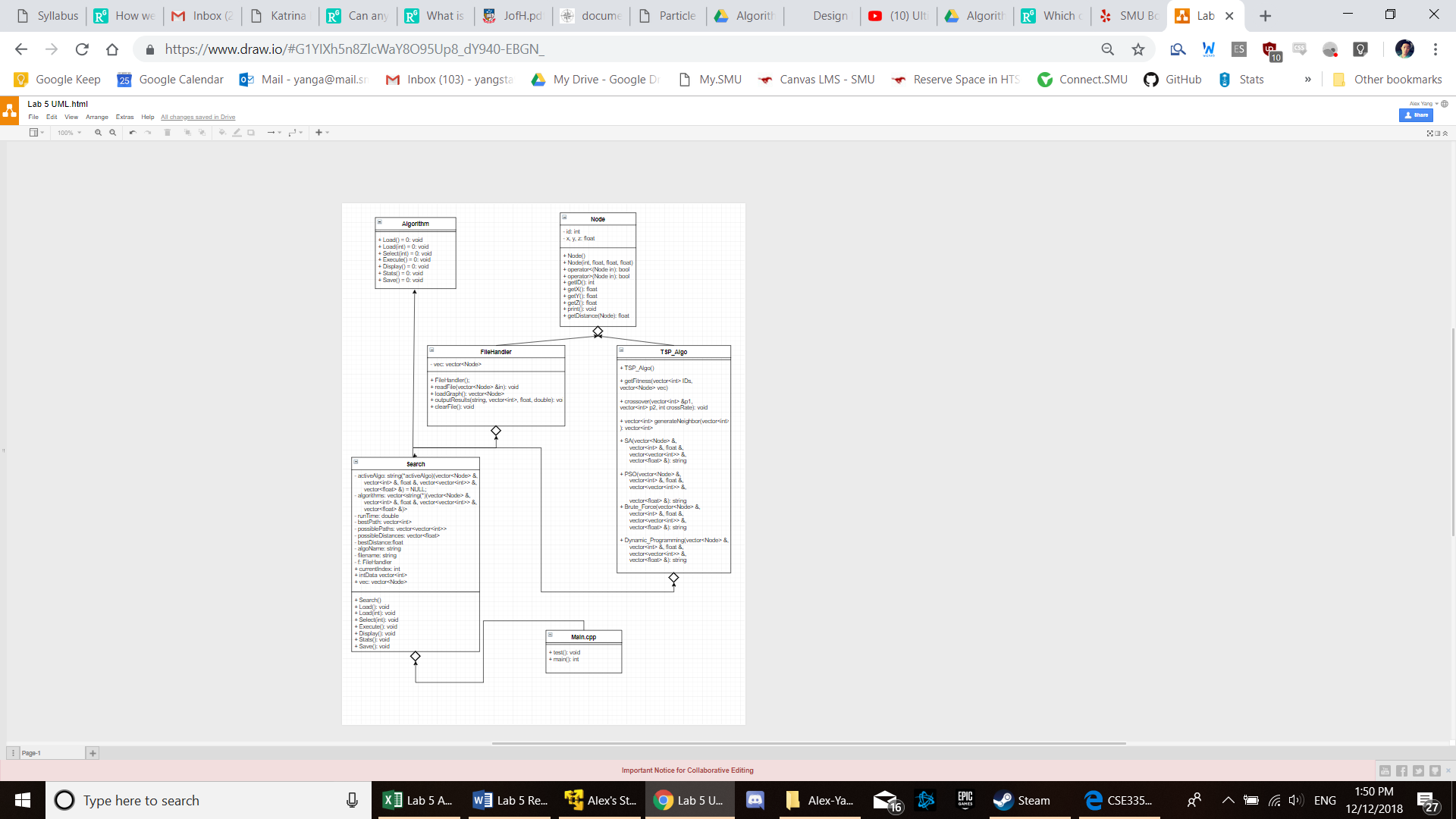
I used the strategy design pattern as my design pattern. Strategy seemed the most appealing to me because I can just write new algorithms in my SearchAlgo class and be able to easily add them to my Search class’s vector of algorithms. Because I’m using the strategy design pattern, I can also run every 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 alpha (temperature decay factor) after each iteration (a float from 0 to .99) of my SA, my two learning factors (c1 and c2) in my PSO which would skew the velocity to be more dependent on either the personal best path of each particle and the global best path of each particle, and the maximum velocity of my PSO which crosses the current path with the best path velocity/10 times every iteration.

UML Diagram



Variations of GA and Tabu

The temperature in my SA algorithm starts at 1. After every iteration, it is multiplied by alpha which is a variable between 0 and 1. This means that as the number of iterations increases, the temperature decreases, and the less likely my program will be to switch to a new hill (in hill climbing). I found that when I decreased alpha, my program ran faster. This is because as my alpha decreases, the rate that my temperature decreases is increased, which means I won’t have to switch hills as often and will converge faster. This also means, however, that my SA algorithm will fail faster since it won’t find the optimal path as nodes increase.

The main variables I altered in my PSO Algorithm were the learning factors and my maximum velocity. By altering my learning factors (c1 and c2) I could tilt the velocity function to be more heavily influenced by a particle’s personal best (c1) or its global best (c2). As you can see from my RAW data, when I decreased c1 and increased c2, my algorithm slowed down. I believe this is due to the fact that if my velocity function favors the global best, there is (usually) a larger distance between a particle’s current fitness and the global fitness than there would be between a particle’s current fitness and its best fitness. Because the higher a particle’s velocity, the more crossovers occur, this would cause the program to slow down, but also make it return the optimal path more often. When I increased c1 and decreased c2, my algorithm sped up. This is because fewer crossovers occur, but it may not return the optimal path as often as if c2 was greater.

My maximum velocity determined the maximum number of crossovers that could occur in each iteration of my PSO algorithm. The number of crossovers was set to velocity/100. When I increased the max velocity of my particles, more crossovers could occur, resulting in a slower run time (although it would return the optimal path slightly more often than at a lower max velocity).