Overview of my program structure

Before I started this program I identified a list of things that I needed to complete for this program to work, which is below.

Tasks needed:

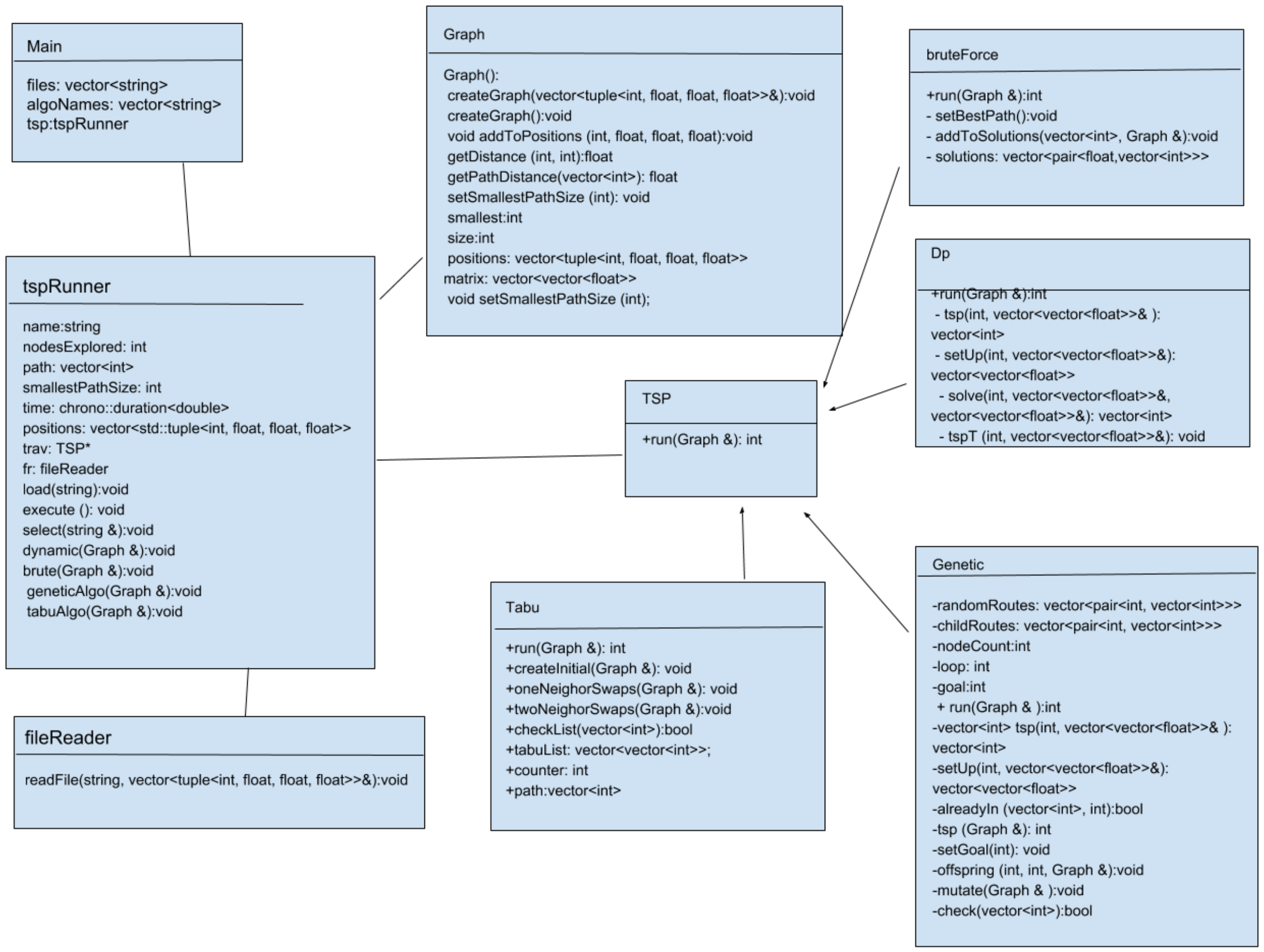
- The text file has to be read in
- The information read in needs to be organized into variables
- These variables need to be stored
- The Brute Force Algorithm needs to use these variables to find the shortest path
- The Dynamic Programming Algorithm needs to use these variables to find the shortest path
- The Tabu Search Algorithm needs to use these variables to find the shortest path
- The Genetic Algorithm needs to use these variables to find the shortest path

How my program is structured to achieve these tasks:

In my program I had 8 classes: tspRunner, fileReader, Graph, TSP, bruteForce, dp, tabu, and genetic. I used the strategy design pattern by defining TSP as an abstraction of an algorithm that computes the Traveling Salesperson Problem, and then created four separate implementations of this algorithm, which are bruteForce, dp, tabu and genetic. When the program first starts, in main.cpp two vectors are created: one which holds the file paths, and one which holds the algorithms that will be used. These vectors can be changed accordingly, but it is important to note that the Dynamic Programming or Brute Force approach needs to occur before the Genetic Algorithm approach, because the Genetic Algorithm needs to know when it has found the most optimal path.

I created a tspRunner class because I thought it made the most logical sense to have a separate class that sets up all the information needed for the algorithm to be computed. I created the Graph object because I wanted to store all the information about the nodes in an object that was separate from the algorithm itself. The tspRunner class has a pointer to a TSP object called trav, a string attribute called name, an int attribute called smallestPathSize, and a vector of tuples called positions. The tspRunner has a function called load which takes in a string as a parameter. In this function, a fileReader object is used to read in the text file input. The fileReader object uses a function called readFile, which takes in a vector of tuples (positions) by reference as a parameter. When it reads the file it adds the node, x position, y position, and z position to this vector. The tspRunner also has a function called select, which takes in a string to set the string attribute name equal to the algorithm name being used. The tspRunner also has a function called execute. In this function, it uses the vector of tuples to create a Graph object. Depending on what the name attribute is set to, it calls one of four methods. The four different methods are called dynamic, brute, geneticAlgo, tabuAlgo, which each take in a Graph object as a parameter. In the dynamic function, the TSP pointer is defined as a dp algorithm and the run method for this is called, sending it the Graph. Similarly, In the brute function, the TSP pointer is defined as a bruteForce algorithm and the run method for this is called, sending it the Graph. The same process is repeated inside the geneticAlg function and the tabuAlgo function.

UML Diagram


Main
files: vector<string>
algoNames: vector<string>
tsp:tspRunner
Graph
Graph():
createGraph(vector<tuple<int, float, float, float>>&):void
createGraph():void
void addToPositions (int, float, float, float):void
getDistance (int, int):float
getPathDistance(vector<int>): float
setSmallestPathSize (int): void
smallest:int
size:int
positions: vector<tuple<int, float, float, float>>
matrix: vector<vector<float>>
void setSmallestPathSize (int);
bruteForce
+run(Graph &):int
- setBestPath():void
- addToSolutions(vector<int>, Graph &):void
- solutions: vector<pair<float,vector<int>>>
Dp
+run(Graph &):int
- tsp(int, vector<vector<float>>&): vector<int>
- setUp(int, vector<vector<float>>&): vector<vector<float>>
- solve(int, vector<vector<float>>&, vector<vector<float>>&): vector<int>
- tspT (int, vector<vector<float>>&): void
tspRunner
name:string
nodesExplored: int
path: vector<int>
smallestPathSize: int
time: chrono::duration<double>
positions: vector<std::tuple<int, float, float, float>>
trav: TSP*
fr: fileReader
load(string):void
execute (): void
select(string &):void
dynamic(Graph &):void
brute(Graph &):void
geneticAlgo(Graph &):void
tabuAlgo(Graph &):void
TSP
+run(Graph &): int
fileReader
readFile(string, vector<tuple<int, float, float, float>>&):void
Tabu
+run(Graph &): int
+createInitial(Graph &): void
+oneNeighorSwaps(Graph &): void
+twoNeighorSwaps(Graph &):void
+checkList(vector<int>):bool
+tabuList: vector<vector<int>>;
+counter: int
+path:vector<int>
Genetic
-randomRoutes: vector<pair<int, vector<int>>>
-childRoutes: vector<pair<int, vector<int>>>
-nodeCount:int
-loop: int
-goal:int
+ run(Graph &):int
-vector<int> tsp(int, vector<vector<float>>&): vector<int>
-setUp(int, vector<vector<float>>&): vector<vector<float>>
-alreadyIn (vector<int>, int):bool
-tsp (Graph &): int
-setGoal(int): void
-offspring (int, int, Graph &):void
-mutate(Graph &):void
-check(vector<int>):bool

Genetic Algorithm Approach:

I first created a bunch of random paths out of the nodes, and got the total path distance. I stored these paths in a vector of pairs called randomRoutes, with the pair being an int (path distance) and a vector (path). Next I sorted this vector of pairs according to the path distance, then I called the TSP method. I set a variable called ‘pSize’ to the current shortest path length. I made a while loop that keeps looping until the current shortest path is equal to the optimal path (which I got from the DP algorithm). I then start the selection process. I did the selection by pairing the most optimal paths with a random paths. I chose the crossover site randomly and the nodes at this crossover site were exchanged to create a new path. The total path distance was calculated for this new path. The path vector and total distance were added as a pair to the randomRoutes vector. After the "mating" process, I made 100 mutations occur. I simulated mutations by choosing a random path, and randomly switching two of the elements in the path. This process continued until the optimal route was found.

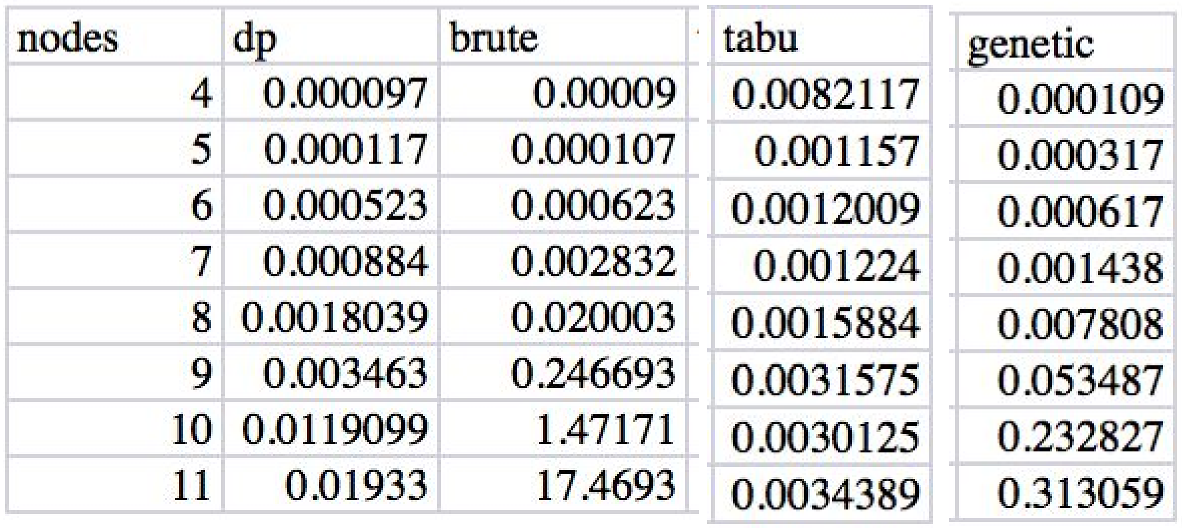
For the selection process I tried a few different ways of doing it. First, I tried doing it in pairs, with the first two paths mating, the third and fourth paths mating, etc. This wasn't very success because there didn't seem to be enough variation. Next I tried looping through all of the paths, and choosing a random path for each of these to mate with. Both of these approaches took too long, so I just decided to use the most optimal routes and have those randomly mate with another path.
For the cross over I initially put the crossover point as the halfway mark, which didn't create enough randomness. I then decided to have the crossover point be random. My method for creating mutations basically stayed the same through the time I was working on the program. One thing that I did find interesting though was that the more times I called the mutation function, the quicker the algorithm went. That is why I decided to do 100 mutations after each mating process.

Tabu Search Algorithm Approach

I created the initial route by simply starting with node 1 and choosing the next closest node, and then the next closest node to that, and so on. I had two functions for neighbor swaps. The first swap switched nodes that directly neighbored each other in the path. If this swap made the path shorter and the path was not one of the most 20 recent paths added to the TabuList, the swap stayed. The second swap function went through and basically did the same thing, but instead of swapping with the neighbor directly next to it, it swapped with all the nodes further along in the path. Similarly, if the swap made the path shorter and the path was not one of the most 20 recent paths added to the TabuList, the swap stayed. I call each of these swapping methods 100 times. I don't think I did this exactly correct because the algorithm only sometimes finds the shortest path. If the path chosen by this algorithm is not the optimal path, it is usually pretty close, but some of the time it's off.

For this algorithm I initially only used a random route as the starting route. When I tried the method of starting with node 1 and choosing the next closest node, the results were closer to optimal. Also at first I only used the first swapping method. I added the second swapping method out of curiosity and it seemed to improve the results, so I kept it. I decided to call the swapping methods 100 times because when I only called them a few times the results were not very good. I decided not to call the swapping methods more than 100 times because the algorithm doesn’t seem to improve after that point. Lastly, I initially was only checking the 10 most recent vectors in the TabuList, but changed this to 20 to ensure cycling would not occur.

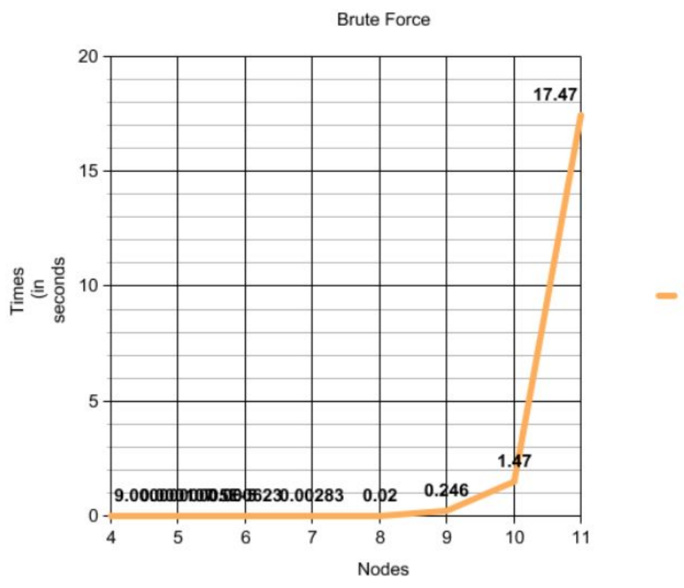
Results of Input Testing

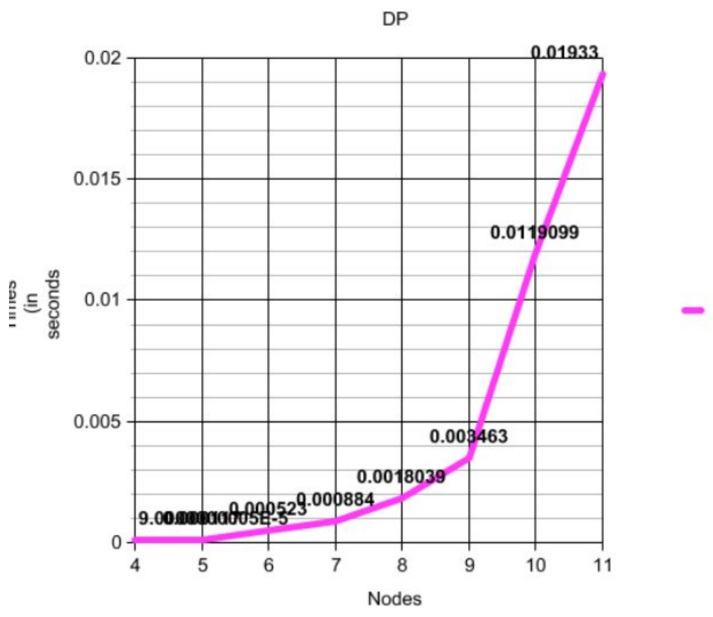


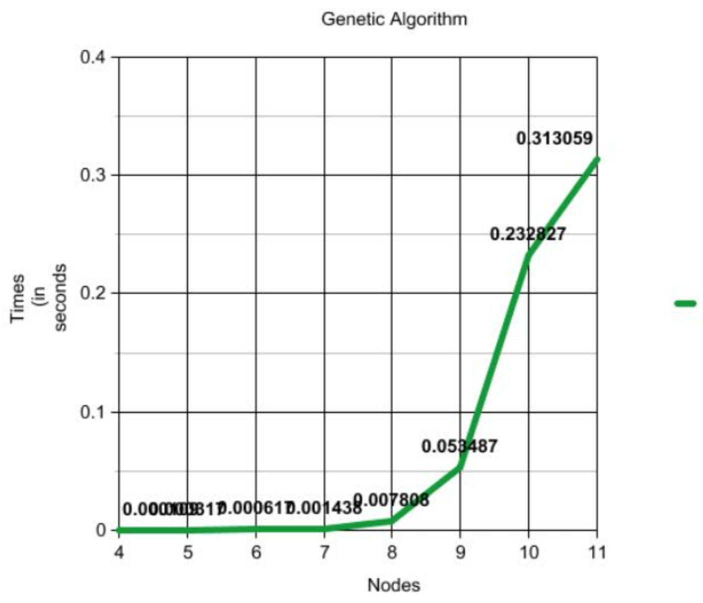
nodes	dp	brute	tabu	genetic
4	0.000097	0.00009	0.0082117	0.000109
5	0.000117	0.000107	0.001157	0.000317
6	0.000523	0.000623	0.0012009	0.000617
7	0.000884	0.002832	0.001224	0.001438
8	0.0018039	0.020003	0.0015884	0.007808
9	0.003463	0.246693	0.0031575	0.053487
10	0.0119099	1.47171	0.0030125	0.232827
11	0.01933	17.4693	0.0034389	0.313059

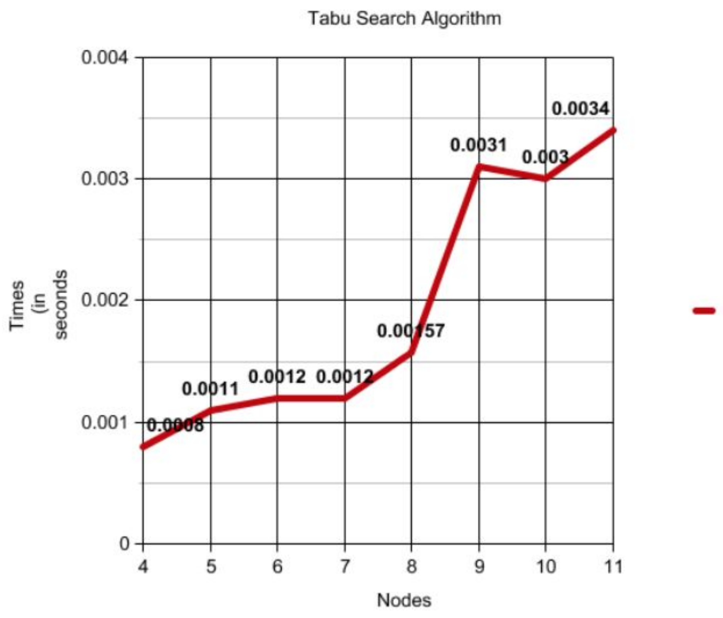
This chart show the average time (in seconds) for each of the Algorithms used. I obtained these results using a function called ‘testing’ which is in my AlgoRunner class, running the algorithms 20 times on each of the node amounts.. In this function, first a node count is chosen. Next, inside a for loop each of the algorithms performs their search on a random list of nodes/positions. The timing results are put into a 2 dimensional vector, and after the for loop ends the average time is computed.

Graphs

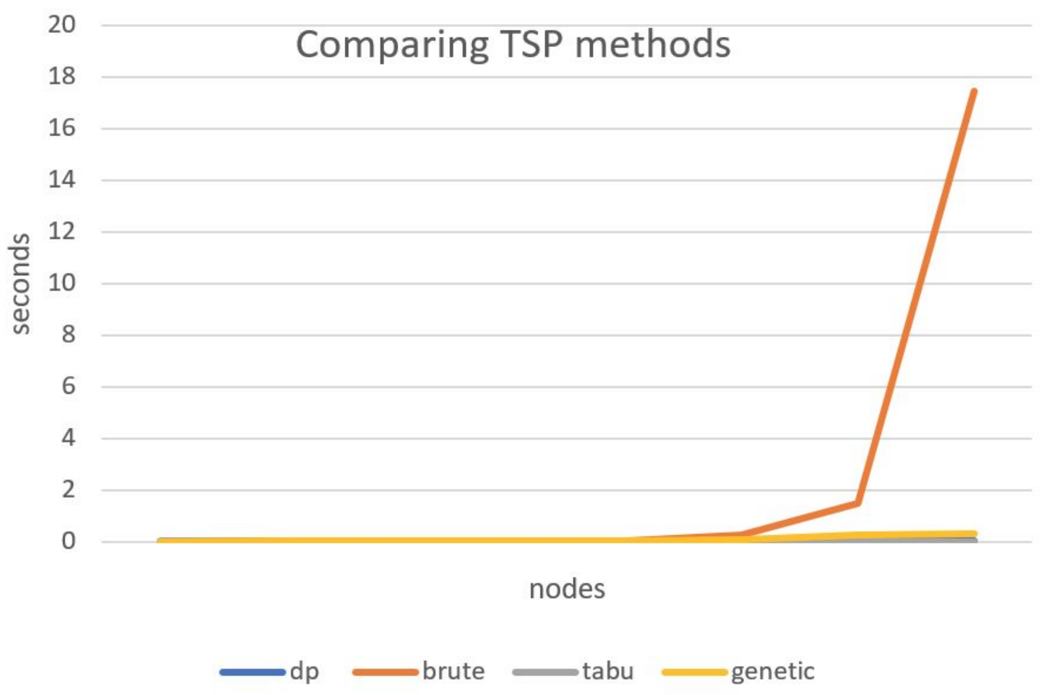

Brute Force
Times (in seconds)
Nodes
17.47
1.47
0.246
0.02
0.00283


DP
Times (in seconds)
Nodes
0.01933
0.0119099
0.003463
0.0018039
0.000884
0.000523


Genetic Algorithm
Times (in seconds)
Nodes
0.313059
0.232827
0.053487
0.007808
0.001438
0.000617


Tabu Search Algorithm
Times (in seconds)
Nodes
0.0034
0.0031
0.003
0.00157
0.0012
0.0012
0.0011
0.0008

Graphs Continued

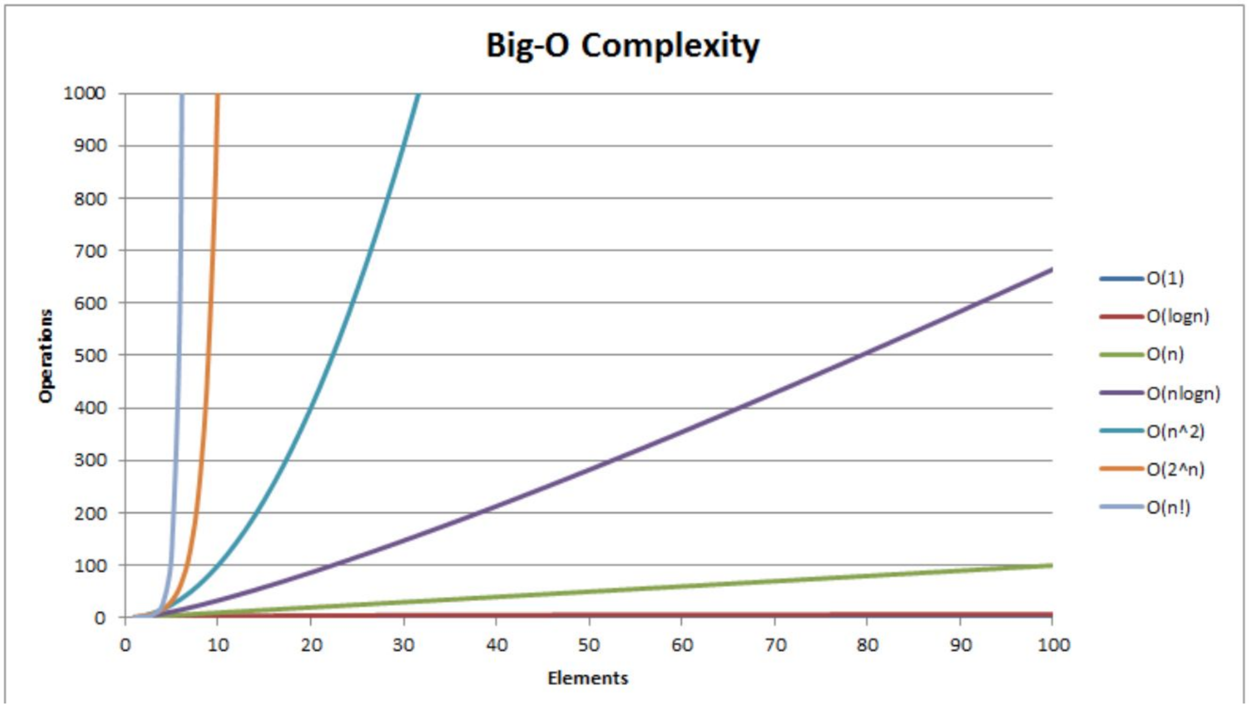


*note: dp line is hard to see because tabu and genetic are overlapping it

Analysis of Results

The dp, genetic algorithm, and tabu search approach to TSP all were pretty quick when trying to figure out the most optimal path. The tabu search algorithm didn't have as good as a chance of finding the most optimal path as the genetic algorithm or the dynamic programming approach. The genetic algorithm seems to always find the most optimal path, but one issue I saw many times was that the algorithm can get stuck in a state where it is 1 distance away from the most optimal path, and then stays like that for a long time. When the node count increases, this issue is more likely to be seen. Most of the time the genetic algorithm very quickly finds the most optimal path, but the average time is higher than the dp approach and the tabu search approach because of this tendency to get stuck.

Time Complexity



The time complexity for Genetic Algorithm simplifies to O(O(Fitness) * (O(mutation) + O(crossover))), so the time complexity is relative to the number of nodes, the number of generations and the computation time per generation. As the number of nodes gets larger, the number of generations usually increases from the added complexity, and because of this the computation time gets larger. The time Genetic Algorithm graph is the most similar to the time complexity of O(n^2), which isn't quite as bad as factorial or logarithmic.

Tabu Search time complexity is dependent on the methods used for neighbor swapping, and the amount of iterations. In mine I do 100 iterations of two different types of neighbor swaps. The amount of neighbor swaps is dependent on the amount of nodes, so the slope of the graph should be increasing steadily but not extremely rapidly, which can be seen on my tabu search graph output. The tabu search algorithm that I implemented has the most similar time complexity to O(n log n).