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 six algorithms need to use these variables to find the shortest path

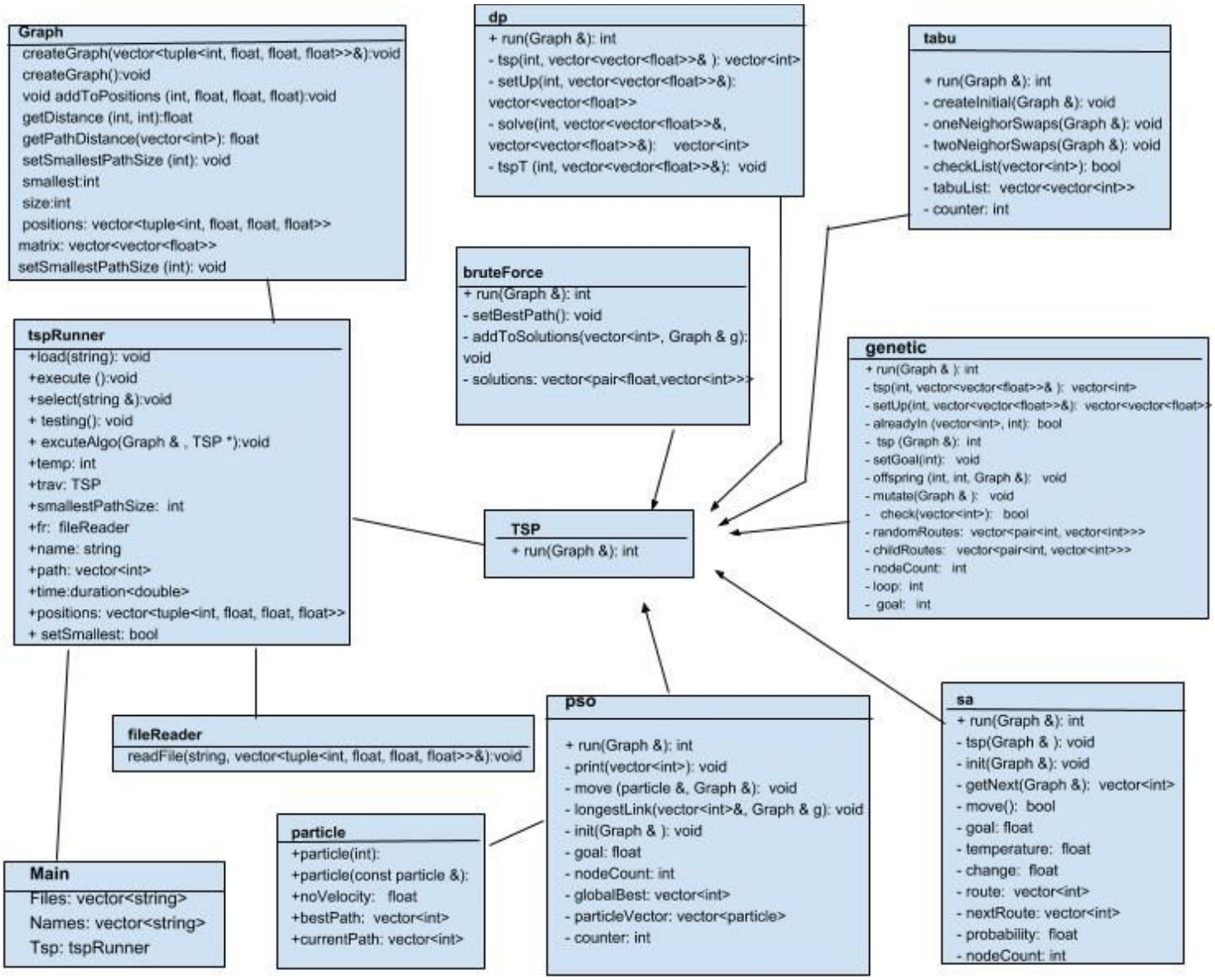
How my program is structured to achieve these tasks:

In my program I had 11 classes: tspRunner, fileReader, Graph, TSP, bruteForce, dp, tabu, genetic, sa, pso, and particle. I used the strategy design pattern by defining TSP as an abstraction of an algorithm that computes the Traveling Salesperson Problem, and then created six separate implementations of this algorithm, which are bruteForce, dp, tabu, genetic, sa and pso. 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 other algorithms, because some of the other algorithms need 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, the TSP pointer is set to that type of algorithm object. For example, if the name is "dp", then trav = new dp(), and the run method for this type of TSP is executed. The tspRunner class also has a function called 'testing' which creates randomly generated x,y,z values for the nodes and runs the algorithms based on these values. This made it easier to obtain results for input testing.

UML Diagram


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>>
setSmallestPathSize (int): void
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
tabu
+ run(Graph &): int
- createInitial(Graph &): void
- oneNeighorSwaps(Graph &): void
- twoNeighorSwaps(Graph &): void
- checkList(vector<int>): bool
- tabuList: vector<vector<int>>
- counter: int
bruteForce
+ run(Graph &): int
- setBestPath(): void
- addToSolutions(vector<int>, Graph & g): void
- solutions: vector<pair<float,vector<int>>>
tspRunner
+load(string): void
+execute ():void
+select(string &):void
+ testing(): void
+ excuteAlgo(Graph & , TSP *):void
+temp: int
+trav: TSP
+smallestPathSize: int
+fr: fileReader
+name: string
+path: vector<int>
+time:duration<double>
+positions: vector<tuple<int, float, float, float>>
+ setSmallest: bool
TSP
+ run(Graph &): int
genetic
+ run(Graph &): 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
- randomRoutes: vector<pair<int, vector<int>>>
- childRoutes: vector<pair<int, vector<int>>>
- nodeCount: int
- loop: int
- goal: int
fileReader
readFile(string, vector<tuple<int, float, float, float>>&):void
pso
+ run(Graph &): int
- print(vector<int>): void
- move (particle &, Graph &): void
- longestLink(vector<int>&, Graph & g): void
- init(Graph &): void
- goal: float
- nodeCount: int
- globalBest: vector<int>
- particleVector: vector<particle>
- counter: int
sa
+ run(Graph &): int
- tsp(Graph &): void
- init(Graph &): void
- getNext(Graph &): vector<int>
- move(): bool
- goal: float
- temperature: float
- change: float
- route: vector<int>
- nextRoute: vector<int>
- probability: float
- nodeCount: int
particle
+particle(int):
+particle(const particle &):
+noVelocity: float
+bestPath: vector<int>
+currentPath: vector<int>
Main
Files: vector<string>
Names: vector<string>
Tsp: tspRunner

SA Approach:

In my simulated annealing class, I have two vector attributes called route and nextRoute. I also have float attributes called goal, temperature, probability, and change. My class functions include a public function called run, which accepts a Graph object by reference as a parameter and returns an int, which will eventually be the path size. I also have three private class functions that will be discussed further on.

In the run function, I first set the algorithm goal (optimal path size), which I get from the Graph. Next I call a function called init. In the init function I initialize the route vector to a random path of nodes and initialize the temperature to the value of 1.0. After calling the init function, I set a variable called pathSize equal to the current route distance. Next, there is a while loop, which stops when the temperature of 0.00001 has been reached or the optimal path size has been found. In each iteration of the loop, I call a function called getNext.

In the getNext function, I create a temporary vector called temp and set this equal to the route vector (which holds the current node path). Next, I swap two random spots of temporary vector, and return this temporary vector. I set the nextRoute vector equal to the return of this function. Next, I compare the total distances of the two vectors. The variable called change is set to the distance of the next route minus the distance of the new route. If the new vector is smaller than the current vector I set the current vector equal to the new vector, decrement the the temperature by 0.00001 and update the pathSize to the distance of the current path. If the new vector is not smaller than the current vector, I call the move function. In the move function, I set the probability equal to the change divided by the temperature. I then then set an int variable called random equal to a random number between 0 and 100, using the rand( ) function. If this random value is larger than the probability, I return true; Otherwise, I return false. For example, if the change in distance is 5 and the temperature is .95, then probability would be roughly 5.26. If the random variable generated is between 6 and 100, this move

will be accepted. If the change in distance is 5 and the temperature is 0.055, then the probability would be 90.9, so the random variable would need to be between 91 and 100 for this move to be accepted. The less detrimental the change is and the temperature higher the temperature is, the high the odds are of the move being accepted. The more detrimental the change is and the lower the temperature, the lower the odds of the move being accepted is. The idea behind this is that when the algorithm first starts, the path is probably not very close to the goal, so it is less damaging earlier on to accept bad moves. As the algorithm gets closer to the goal, bad moves can still be accepted but the odds of the move being accepted decrease greatly. “Bad” moves are still needed, because although they might temporarily add distance to the path, they also add variability to the path which can opens the doors for a very good move. If the return value of the move function is true, I set the current vector equal to the next vector, decrement the temperature by 0.00001 and update the pathSize attribute to the distance of the current path.

When I first started working on this algorithm, I had the temperature starting at 10 and decrementing after every while loop iteration by 0.00001. In my move function, I was computing the probability by multiplying the route’s distance change by the temperature. If the probability was less than a random number generated between 0 and 100, the move was accepted. This made sense to me because if the temperature was at 9 and the distance change was 10, then there was only a 10% chance on the move being accepted. If the temperate was at 9 and the distance change was 2, there was an 82% chance of it being accepted. When I ran my algorithm, it was taking a very long time and the results were no where close to optimal. Upon further inspection, I realized that my formula for calculating probability was not accurately encouraging the behavior that I wanted. For example, at temperature 1 if the distance change is 2, then there would be a 98 % chance that the move is accepted, opposed to the 82% chance if the temperature was at 9. For a given distance change, it should be less likely for the move to be accepted at a lower temperature than a higher temperature. When I changed the

temperature to start at 1.0 and decrement it by 0.00001, I also changed the probability formula to be change divided by temperature, which gave me much better results and made the algorithm faster. Additionally, I decided to only decrement the temperature if a move was made. In my algorithm, a move is made if the next path is more optimal then the current path, or if the move function returns true. If the next path is worse than the current path, and the move is not accepted, I realized it was pointless to decrement the counter because the algorithm wasn't getting actively closer to the goal.

PSO Algorithm Approach

For the particle search optimization, I created a class called particle. The particle object has a vector of ints called bestPath, and vector of ints called currentPath, a float called noVelocity, and a float called starting distance. In my particle class, there are also two constructors: one that takes an int (largest node number), and a copy constructor. In the constructor that takes in an int, it creates a random path of all the nodes, and sets currentPath and bestPath equal to this path, and the noVelocity to 0. The noVelocity attribute is increased every time the path fails to improve the distance of the current path, so it keeps track of how many iterations the particle has gone through where there has been no improvement. Once the particle improves it's path, it is set back to 0.

In my PSO algorithm class there is a function called run, which takes a graph object by references as a parameter. In this function, a vector of 20 particles is created.After creating the vector of particles, I find out which particle has the most optimal current route, and set a vector called globalBest to this path. Next, there is a while loop, which continues going until the optimal path is found. In each iteration I call a function called 'move' on each of the particles in the particle vector.

In the move function, the particle's velocity is checked. If the noVelocity attribute is checked. If this is between 0 and 3, a function called 'longestLink' is called, and the particles current path and the graph object are sent as parameters. If the particle's noVelocity is between 4 and 6, that means it hasn't been making any progress for a while. Due to this, the particle's current path is set to the particle's best path, and then the longest link function is is called, and the particles current path and the graph object are sent as parameters. If the particle noVelocity is over 6, then the particle's current path is set to the global best path, and then the longest link function is is called, and the particles current path and the graph object are sent as parameters.

When the algorithm is looking for a path of a relatively smaller size of nodes, the particle's noVelocity is rarely over 6, so the optimization is more dependant on each of the particles following and updating their local best path. When the path sizes get larger, the particles become more dependent on the global best.

In the longest link function, I use a for loop to figure out which two nodes that are next to each other in the path have the longest distance. I then swap one of these nodes with another node. My thought was that this would be an attempt to improve the particle's current path distance but getting rid of the longest link and trying a different yet similar route.

After this function is done, the program returns back to the move function. I check if the newly generated path's distance is less the particle's best path distance, and if it is I set the local best path equal to the new path. Similarly I check if the newly generated path's distance is less than or equal to global best path distance, and if it is I set the global best path equal to the new path. So essentially, if the particle's path is not improving. I instead decide to reproduce a variation of either the particles local best or the global best.

I'm not really sure how to feel about my algorithm. On the surface, it kind of resembles particle swarm optimization, because of the fact that the next path is determined by a certain velocity (or lack of, in my algorithm), as well as the local

best and global best paths but I don't feel like this way the best way to do it. I tried many different approaches to get my algorithm to work appropriately. Initially, the way my move function worked was that if first checks if the particle's current path is equal to the global best. If it is, then all three vectors (bestPath, currentPath, globalBest) are equal and I would just do a random swap, since the global best and local best are already incorporated in current path. If the current path was equal to the local best but not global best, I would check which elements of currentPath are positioned in the same spot in globalBest, and called these 'saved' indexes. I did one swap between two of the indexes that are not saved.

If the current path is not the particle's best path or the global best, I repeat the same process, except this time the index was 'saved' if the element at that index is equal to the element in the particle's best path or the global best path at the same position. If the current path distance was 3 greater than the starting path distance, then the velocity was poor. If this is the case and there are 4 unsaved spots in the current path, I did 2 swaps between indexes that are not saved. Otherwise, if there are two free spots, I did one swap between these indexes that are not saved. If there were no free spots, I would do another random swap. I also tried incorporating a crossover function, similar to the one used in GA.

I changed this process to the way mine is now because I was realizing that the particle paths were not really becoming more similar to the global/local bests, or "swarming" toward them; there was too much randomness. The way I'm doing it now forces the particles path's to be very similar to the local/global best, but with a slight variation. The variation is also somewhat intelligently made because it tried to eliminate the most detrimental link in the path for hopes of improval. One thing I also changes a few times was the size of the particle vector. Initially my vector only had a 5 particles. It was moving rather slowly, so I decided to increase the vector size. When I increased the particle vector to 20 there was more variation in the initial paths which was helpful.

Results of Input Testing

Time Testing on all 6 Algorithms - Node Count 4 to 11

Nodes	DP	BF	Tabu	GA	SA	PSO
4	0.000097	0.0009	0.0082117	0.000109	0.151184	0.000115
5	0.000117	0.000107	0.001157	0.000317	0.167258	0.000202
6	0.000523	0.000623	0.0012009	0.000617	0.204249	0.005444
7	0.000884	0.002832	0.001224	0.001438	0.229235	0.007374
8	0.0018039	0.020003	0.0015884	0.007808	0.242283	0.042445
9	0.0003463	0.246693	0.0031575	0.053487	0.244646	0.0287843
10	0.0119099	1.47171	0.0031575	0.232827	0.248179	0.066506
11	0.0192	17.4693	0.0034389	0.313059	0.257616	0.230745

*Time in seconds

Time Testing on DP, Tabu, SA, PSO - Node Count 4 to 20

Nodes	DP	Tabu	SA	PSO
4	0.000097	0.00082117	0.151184	0.000115
5	0.000117	0.001157	0.167258	0.000202
6	0.000523	0.0012009	0.204249	0.005444
7	0.000884	0.001224	0.229235	0.007374
8	0.0018039	0.0015884	0.242283	0.042445
9	0.0003463	0.0031575	0.244646	0.0287843
10	0.0119099	0.0031575	0.248179	0.066506
11	0.0192	0.0034389	0.257616	0.230745
12	0.00292132	0.00303	0.24634	0.372097
13	0.0691082	0.003645	0.268728	0.876083
14	0.160066	0.004307	0.275573	0.567859
15	0.34339	0.0042464	0.293005	4.33506
16	0.76663	0.00523	0.302684	5.62563
17	1.64206	0.005809	0.319295	9.92706
18	3.58197	0.006263	0.330865	35.4292
19	9.15537	0.00786	0.342652	173.387
20	17.6184	0.007838	0.374906	n/a

*time in seconds

Average Distance from Optimal Path for SA and Tabu

Node Count	SA	Tabu
4	0	0
5	0	0
6	0	1.2
7	0	1.6
8	0	2.6
9	0	3.2
10	0	4.7
11	0	6.32
12	0	6.8
13	0.4	7.96
14	1.08	7.76
15	0.92	8.52
16	0.76	11.2
17	2.48	10
18	1.84	10.52
19	3.9	14.8
20	3.7	10.4
21	3.5	17.83

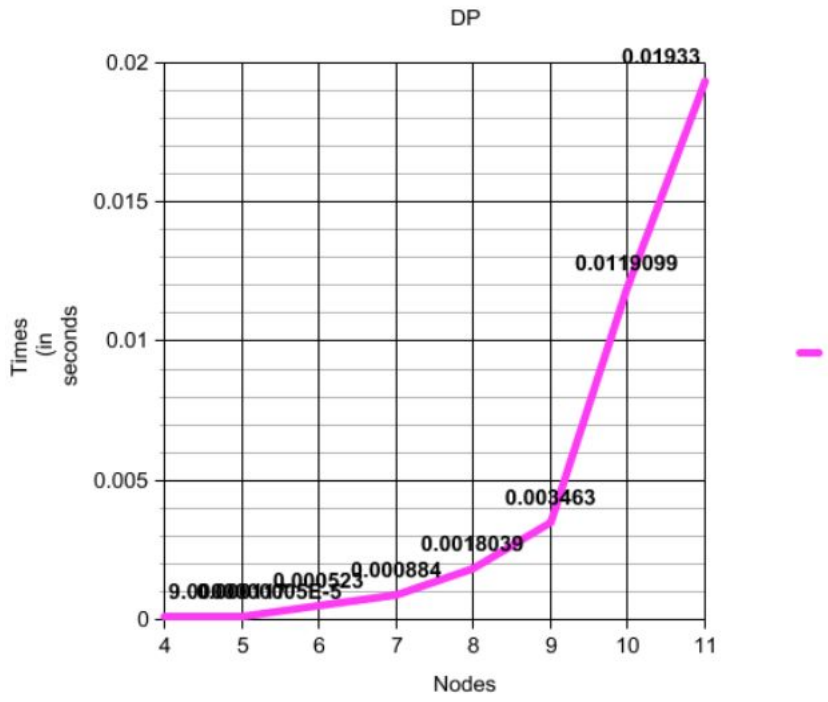
The first table shows the average time (in seconds) for each of the Algorithms used. The first table only shows the timing for paths up to 10 nodes, because after 10 nodes it is difficult to test my version of Brute Force and GA. The second table shows the average time in seconds for DP, Tabu, SA and PSO up to a path size of 20 nodes. I obtained these results using the function called ‘testing’ which is in my tspRunner class, running the algorithms 20 times on each of the node amounts. In this function, first a starting node count is chosen. Next, there is a

while loop, which ends when the node code reaches the desired ending node count. 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.

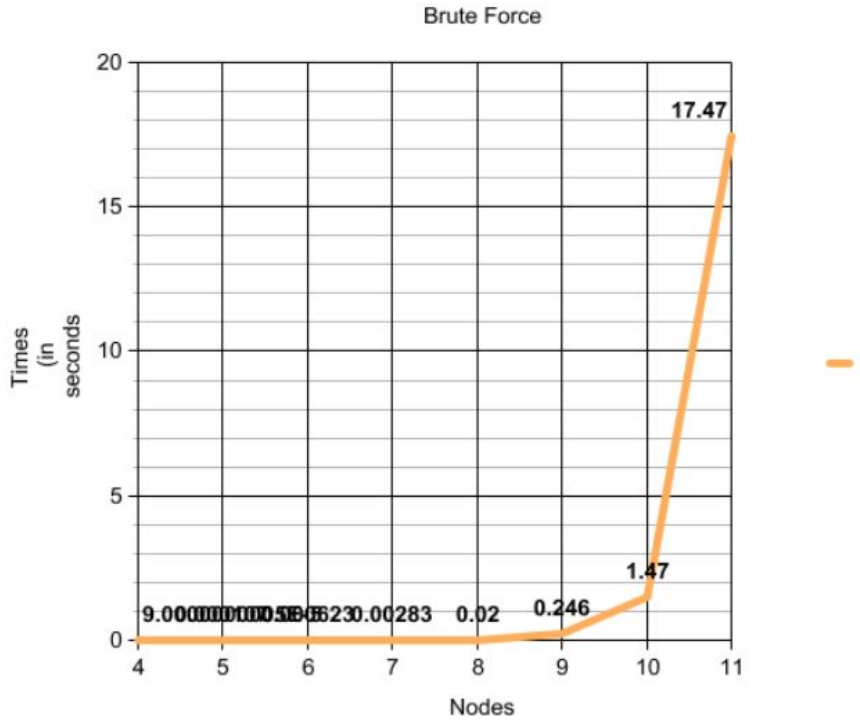
The third table compares how close the average paths found by Tabu and SA are to the most optimal path, from path sizes 4 to 20. Theses results were also obtained in the testing function of the tspRunner class, and computed using a two dimensional vector in a similar way to how I calculated the average times. I compared these two algorithms because my goal for both of them was to find a good path, but not necessarily the most optimal path. I thought it would be interesting to see which one is usually closer to an optimal solution.

Timing Graphs of DP and BF

(1)

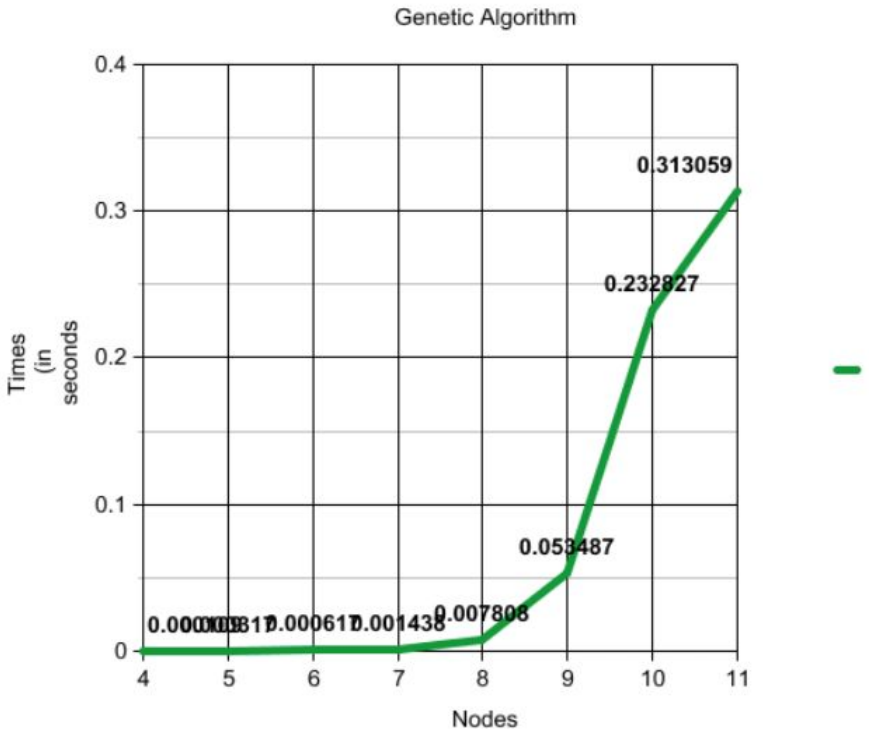


(2)

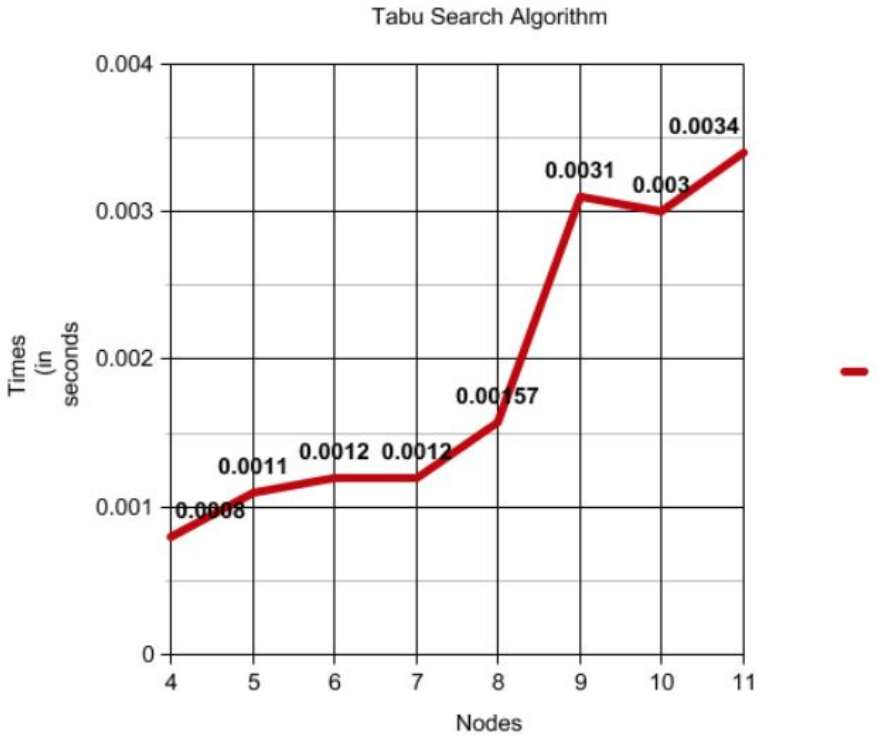


Timing Graphs of GA and Tabu

(3)

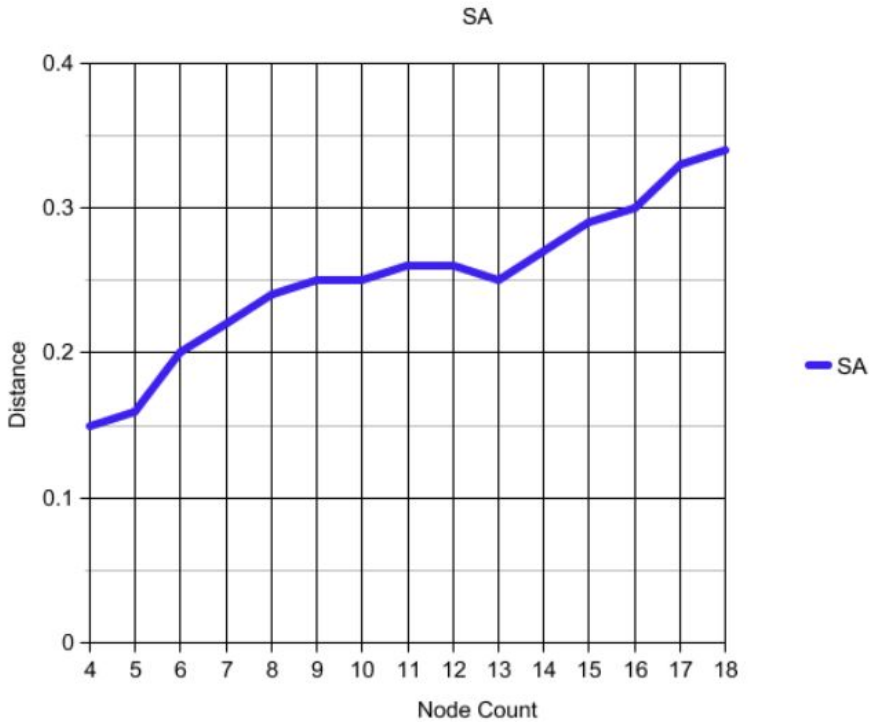


(4)

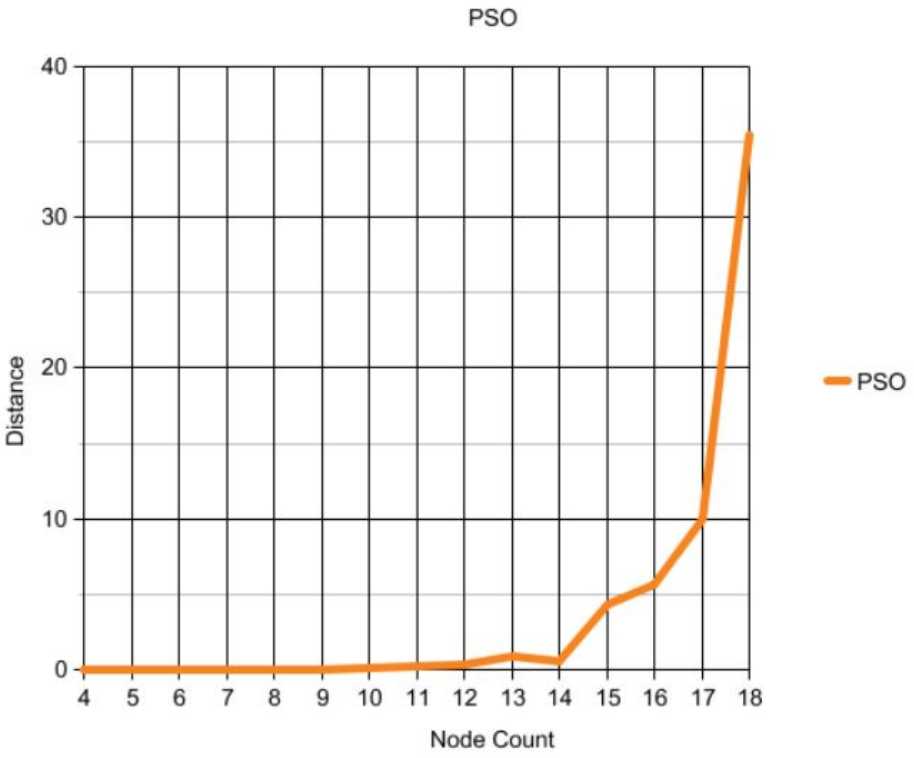


Time Graphs of SA and PSO

(5)

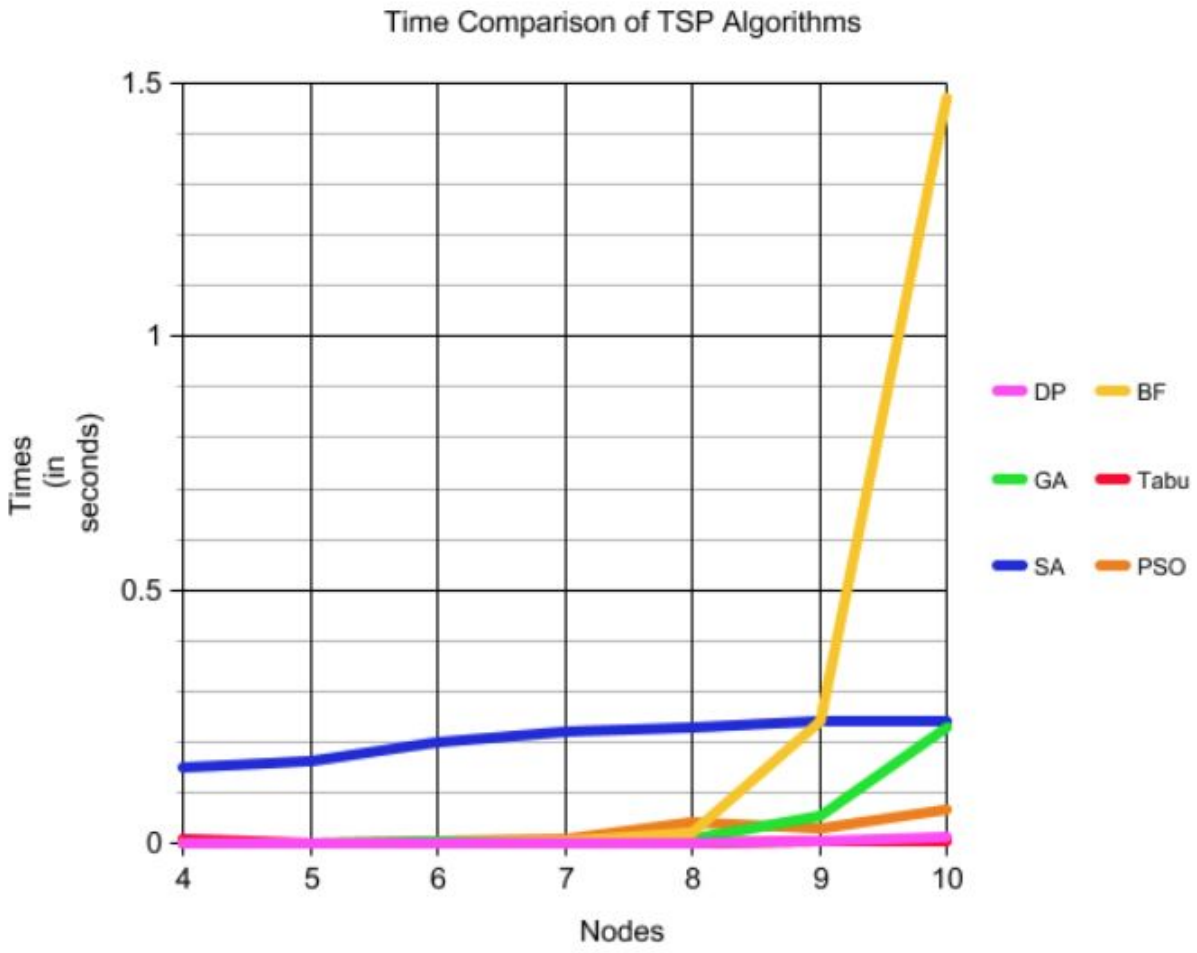


(6)



Comparing All 6 Algorithms

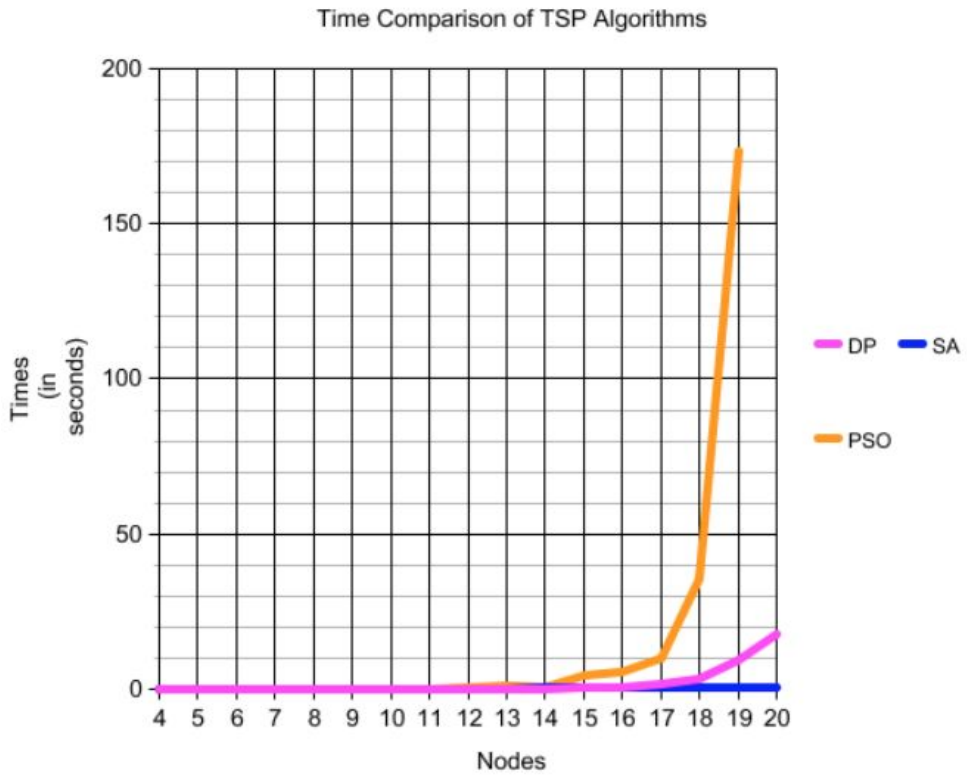
(7)



*note: Tabu line is hard to see because DP is overlapping it

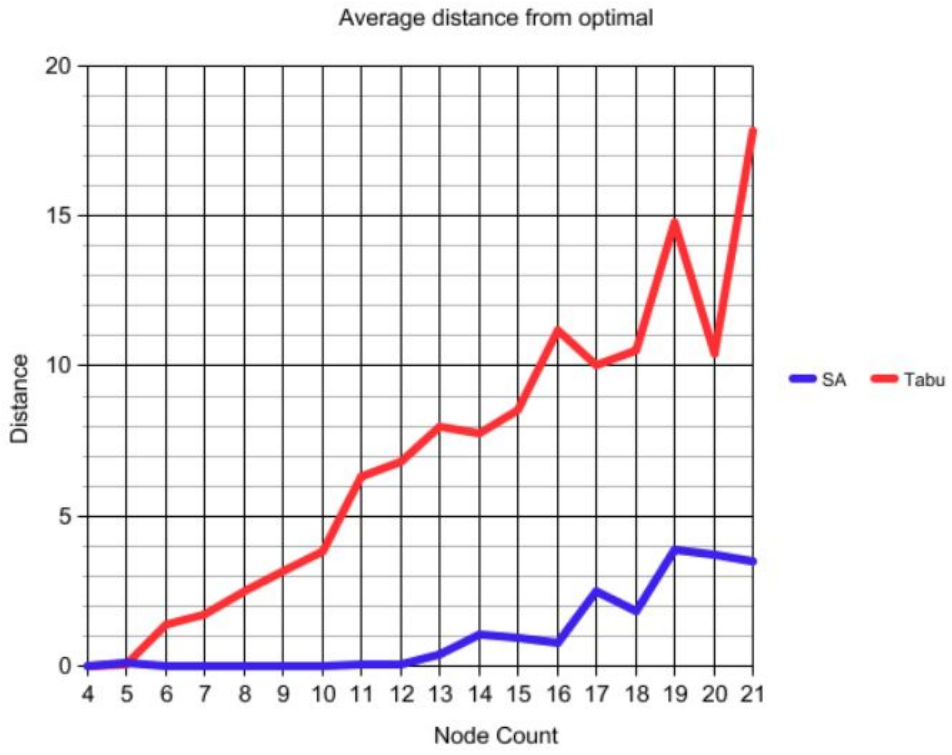
Comparing DP, PSO and SA from Nodes 4 to 20

(8)



Average Distance from the of Optimal Path for Tabu and SA

(9)



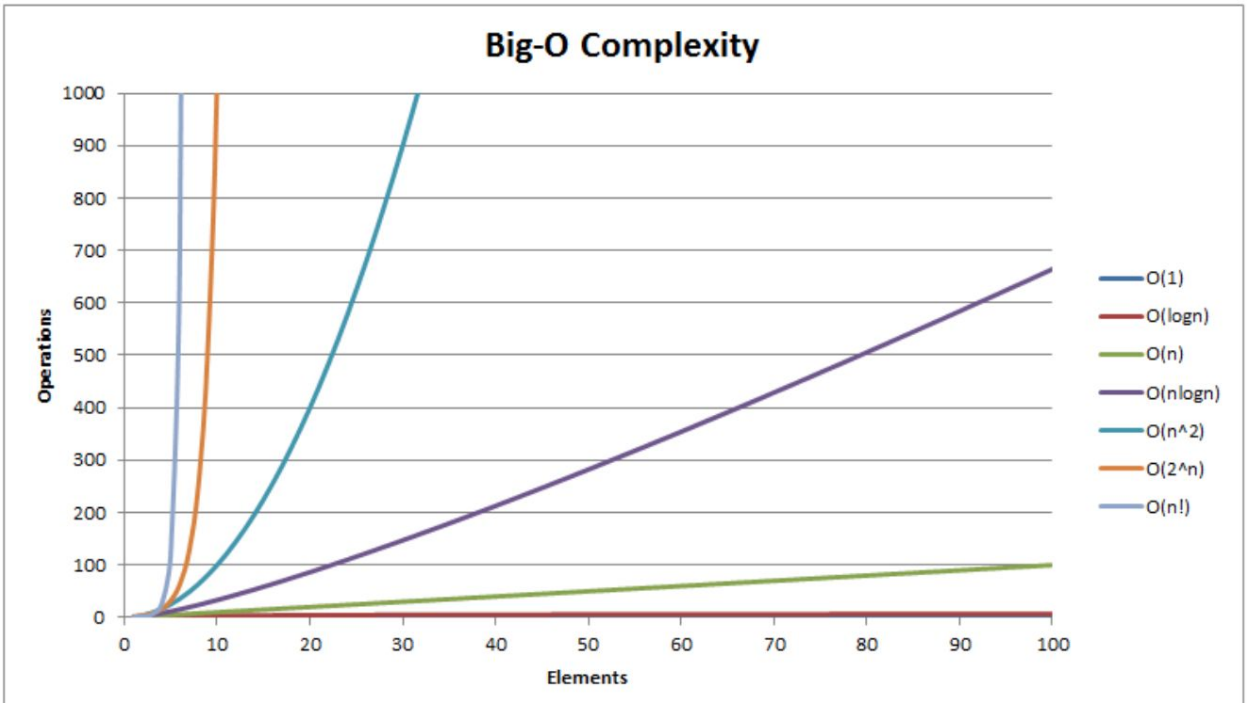
Analysis of Results

DP, GA, and the Tabu Search approach to TSP all were pretty quick when trying to figure out the most optimal for lower node counts. Although DP is quick at first, the slope begins to increase exponentially. By 20 nodes, it takes DP over 17 seconds to find the shortest path. The Brute Force Algorithm is quick until about 8 nodes, but after that it increases at a factorial pace, which made testing for time difficult on larger path sizes. The Simulated Annealing algorithm starts off a lot slower than the other algorithms, but the slope (timing vs node count) only increases slightly as the node count increases. To put it in perspective, from 4 nodes to 20 nodes SA's average time only increases by roughly 1/5th of a second, where as DP's timing increases by 17 seconds. Although the simulated annealing algorithm doesn't continue going until the optimal path is found, the accuracy of the routes are pretty impressive. For 20 nodes it took SA under half a second to find a close to optimal path, and the path it found was on average was 4 units larger than the most optimal route (avg. optimal path of 20 nodes ~ 120 distance units). My Tabu Algorithm also doesn't go until the most optimal path is found. Although the Tabu Algorithm is quick, the accuracy of SA is way better. For a path of 20 nodes, the path found by Tabu is usually 17 units away from the most optimal path, which can be seen in my graph comparison of the two Algorithms. My PSO algorithm is relatively quick until it hits 14 nodes. From that point on, the time

increases drastically as the node count increases. 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.

In summary of my algorithms, if you're testing a path that is between 4 and 9 nodes and you want a quick way to find the most optimal path, DP, Brute Force, PSO, and GA are all suitable options. If you're testing a path that is between 10 and 14 nodes and you want a quick way to find the most optimal path, the best algorithms to use are DP and PSO. If you want the quickest solution to a decent path but not optimal path, Tabu is a good choice. If you want to find a close to optimal solution for a path with a large amount of nodes, SA is the best Algorithm to use.

Time Complexity



The time complexity for the PSO algorithm is relative to the size of the path, the the amount of particles, and the method used to find the next vector. The timing for my PSO algorithm started off well, and then seemed to exponentially increase. The timing for this algorithm was not as bad as factorial timing, but it was worse than the dp timing, which has a known time complexity of $O(2^n)$.

The SA algorithm's timing was the most stable of the six algorithms; it had a slow, evenly increasing slope. At first, it started off slower than the other algorithms, but because of its slow increase, it's timing is much better than almost all of the other algorithms at higher node counts.