In Lab 3 we were told to solve the Traveling Salesman Problem in two ways: Brute Force and Dynamic programming. The Brute Force solution is to generate every combination of nodes and determine the solution. This means that we are generating all permutations of the solution space giving us a time complexity of O(n!). The dynamic programming solution is O(n2\*2n) because we have O(n\*2n) subproblems and each subproblem(iteration through the function) takes n time to solve. The Dynamic Programming solution ends up being only slightly better than the Brute Force method because the Dynamic Programming method is still in exponential time, which is still horrible in terms of efficiency and slightly faster than factorial time according to *Figure 2*.

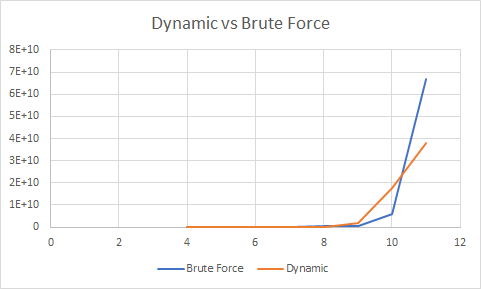
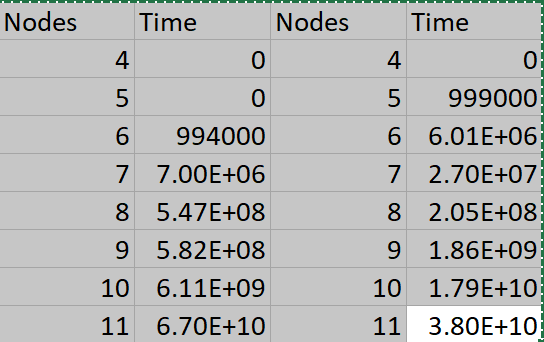


Figure 1

* 

In execution much of the same held true. In the beginning the Brute Force was faster. For example for 6 nodes Brute Force for was 994000 nanoseconds and Dynamic Programming was 6009000 nanoseconds. The math behind this checks out as 6! is 720 and 62\*26=2304. As seen in the graph, once you get past 8 nodes Brute Force quickly gets out of hand quickly eclipsing the Dynamic Programming. And the math once again backs this up as 9! is 362880 and 92\*29=41472. 

For the Brute Force method we are generating every permutation of possible paths. To do this I used a function from the STL library next\_permutation that takes in a range of elements and generates that next lexicographically greater permutation. As each path is generated they are placed in a priority queue that holds a pair of the path and the cost of the path and orders them by lowest cost. The cost function was sqrtf(((x2-x1)\*(x2-x1))+((y2-y1)\*(y2-y1)))+((z2-z1)\*(z2-z1)). Since next\_permutation generates the next lexicographically greater permutation, the source node is left off the brute force generation and is calculated in the cost separately because the source node has to always be the first and last element. For example, if the source node is 1 and we have 5 nodes, we would generate every permutation of 2,3,4 and then in the cost calculation function add the cost from the source to the first element and the last element to the source. This approach actually makes the time complexity O((n-1)!) because we are always leaving out the source node from permutation generation. This approach started to get too long after 10 nodes.

For the DP function I built my function based off of the Held Karp Algorithm discussed in my two references below. The algorithm is based on the base case that if there are only two elements left you return the distance from last element to the source node since it is making a Hamiltonian cycle. So to get to a base case you first put into the function the source and destination, which will be 1. For a four node graph of 4 it would work like this.

* Call it initially in Dynamic DP(1,1,{2,3,4},path);

First we calculate the minimum distances with these statements. Each of these are subproblems of the TSP problem calculating the distances with seperate paths.

* Then next call will be distance (1,2)+ DP(2,1,{3,4},path);
* Next is distance(1,3)+DP(3,1,{2,4},path);
* Finally distance(1,4) + DP(4,1,{2,3},path);

After we would just have to calculate the minimum between the remaining nodes using a sum vector we filled with all the values returned from the subproblems. Finally we would get to our base case and calculate the distance between the last node and the source node. This approach is O(n2\*2n) as discussed above. This approach got up to 11 nodes before it took too long.

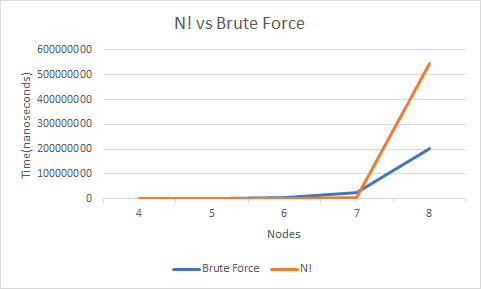
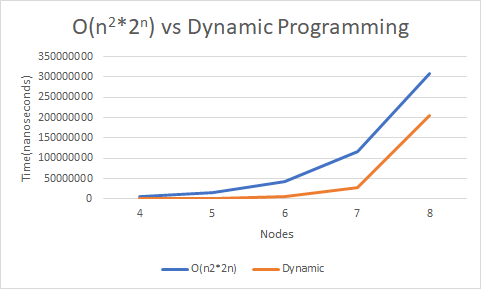
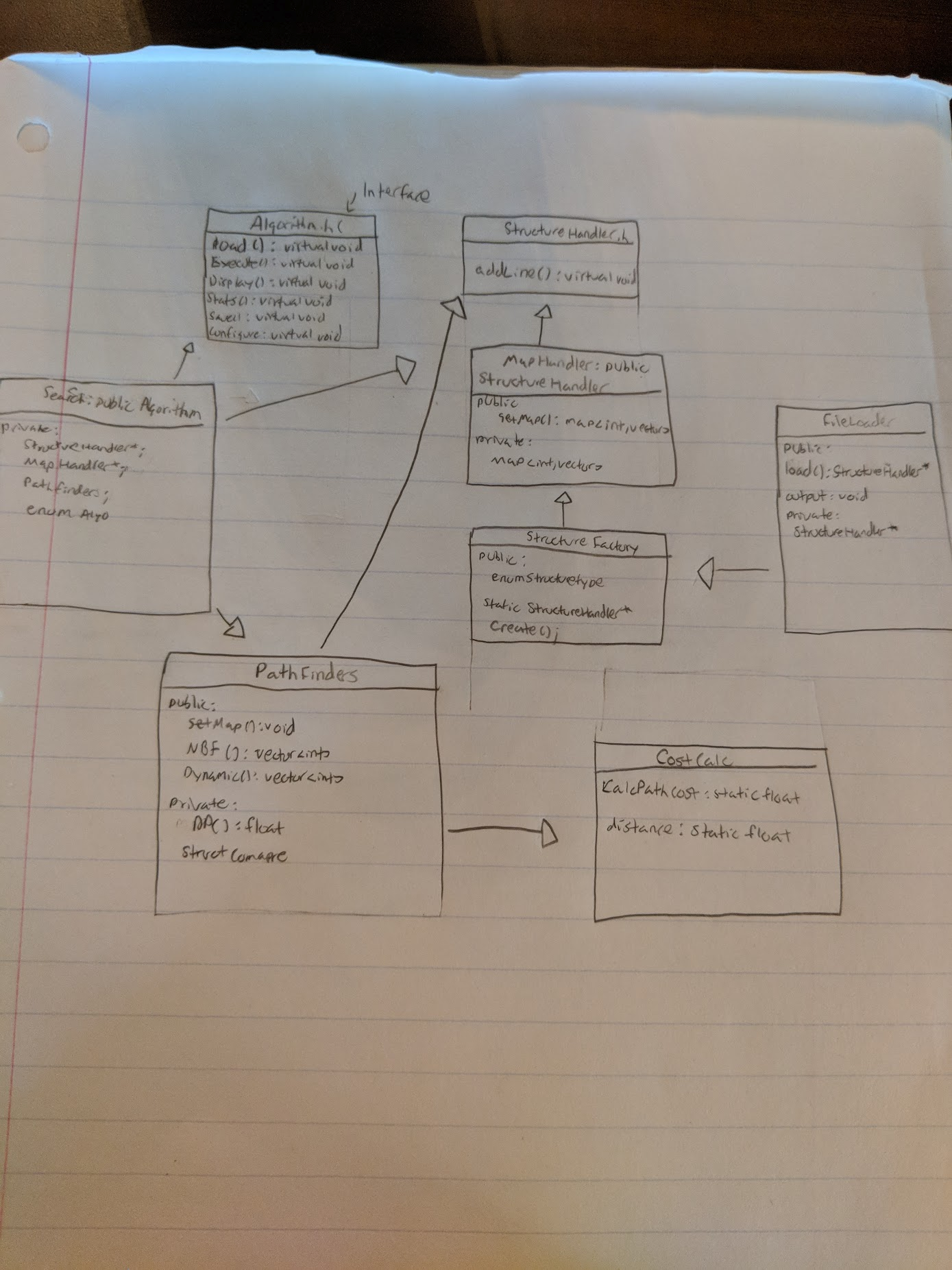
As for the actual performance of my algorithms. Comparing my brute force algorithm to N! It seems to perform better as N! gets a more significant spike at 7 than Brute Force. This is probably due to the fact that my Brute Force is (n-1)! and the uses of next\_permutation limits the amount of operations needed. As for my dynamic programming performance it seems to perform slightly better than O(n2\*2n), but still follows the same trend line. I’m honestly not sure what makes this slight change in performance and am compelled to say that this slight difference is in the realm of error for timing. Since these both follow the same trend, these perform essentially the same.

Figure 3

To write this program I used two design patterns for different parts of the program. I used a factory that generates different data structures based on an enum for the File Loading and used the strategy pattern we have been using for the past two labs for the pathfinding algorithms.

The first part of the program is the StructureHandler, StructureHandler is a pure virtual interface that has one public method to add a line. The purpose of this interface is so you can derive child classes such as MapHandler or something that would manage a vector or linked list and add a line from FileLoader to it. In this program I was only using a Map, so in MapHandler, it inherits from StructureHandler (*Figure 3*) and overrides add line to add the node and position information to a map of an int node and three float position values. It also has another public function called getMap that allows the Search class to retrieve the node map information when the algorithms are being run. The FileLoader inherits from StructureFactory (*Figure 3*) and calls its static Create function to choose a structure to load into. StructureFactory points a StructureHandler pointer to a new child class based upon the enum. In the scope of this program it will point to a new MapHandler. The FileLoader then parses out each line and calls addLine on the structure specified in the StructureFactory. This allows the FileLoader to be able to load a file into any structure of the users choosing as long as there is a handler class for their specific structure. I designed it this way so this file loading code could be reused for other programs that require other data structures.

For the actual searching part of my Traveling Salesman Program, I opted to reuse the strategy pattern from Labs 1 and 2 as the Pathfinding algorithms fit into the design fairly easily. For this pattern everything stems from the Algorithm pure virtual interfaces. For this program I have a Search class that inherits from Algorithm to act as a hub where all the functionality of the program is run through. It also allows each algorithm from PathFinders to be dynamically switched to using function pointers. Search’s load function calls theload function from FileLoader and sets it’s private data member map to what FileLoader returns. It also has to check if it has returned a map as FileLoader could return different types of structures. Execute() just calls the function the function pointer is pointing to, and display() prints out the path. Stats prints out relevant statistics to the project like timing, and the path. Finally save() calls FileLoaders save function. Search in the scope of this project uses the functions from Pathfinders to run. The strategy pattern was great for this program because it allowed me to have the same complex functionality of past programs (function pointers, statistics, and such) while only having to change the searching/sorting algos file to Pathfinders. Additionally, it keeps everything organized very well, and as is the biggest benefit with this pattern it lets me dynamically swap between algorithms on the fly without having to hard code it in main.

References:

<https://stanford.edu/~rezab/classes/cme323/S16/projects_reports/burton.pdf>

<https://people.eecs.berkeley.edu/~vazirani/algorithms/chap6.pdf>