Lizzy Hanna

CS 3353 – Algorithms

11/12/19

Lab 3 Report

In this lab I explored how to solve the Traveling Salesman Problem using both brute force and dynamic programming approaches. The highest amount of nodes my Brute Force algorithm could solve in a reasonable time (about 45 minutes) is 13 nodes…if you do not want to wait 45 minutes, my BF algorithm can solve 12 nodes in around 3 minutes. The highest amount of nodes my DP algorithm could solve in a reasonable time (about 8 minutes) is 27 nodes.

The time complexity of the Brute Force algorithm is O(n!). This can be proven through the graph of the time complexity of the timings I took for Brute Force in Figure 1 (which follow the slope of a factorial graph) and also by the fact that my algorithm looks at n!/n, or (n-1)!, permutations every time it runs, since it tries every permutation of the given set of nodes where the starting and ending node are equal to 1. The reason that my brute force algorithm looks at (n-1)! permutations and not n! permutations is because we only look at a portion of the possible permutations and don't consider any permutations that start and end with anything other than 1. For example, if there are 3 nodes, there are 6 possible permutations. However, my algorithm will only consider 2 permutations (1-2-3-1 and 1-3-2-1), and (3-1)! = 2. Additionally, if there is only 1 node, I had to account for that edge case (in the dynamic programming approach as well) and just return the city itself with a distance of 0. Figure 1 still demonstrates how the slope of each line is n!, and because the y axis is in microseconds and I only graphed values of n! without regard for time, the lines do not exactly overlay but still get the general idea across. The time complexity is still O(n!) instead of O((n-1)!) because in timing complexity, constants do not make a huge difference when it comes to massive orders of calculations and are usually thrown out of the equation.

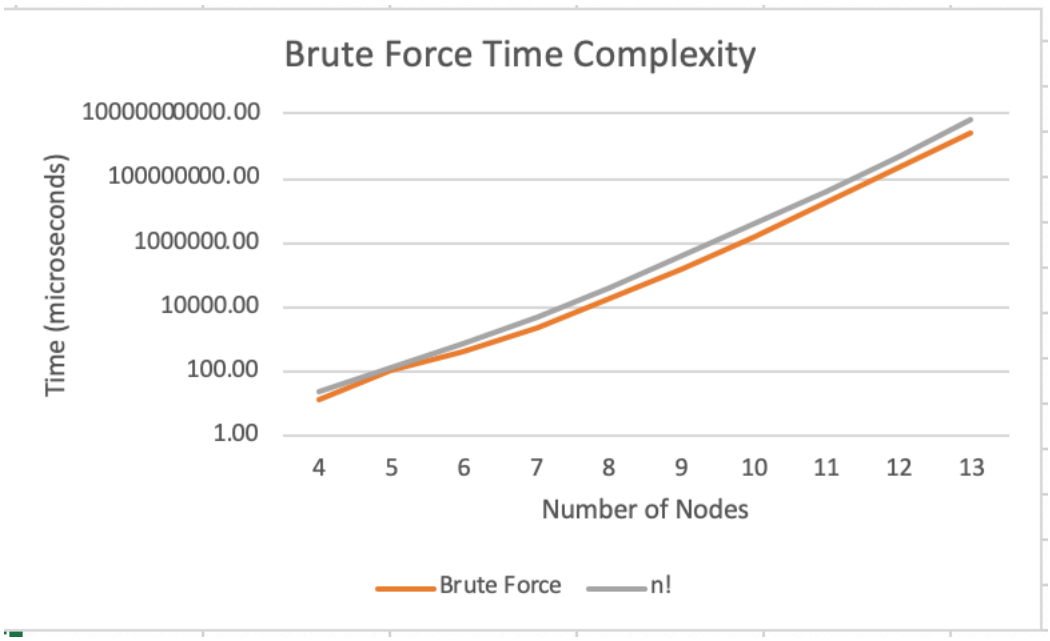

Figure 1. Brute Force Time Complexity and asymptotic timing

The time complexity of my Dynamic Programming algorithm is $O(2^n n^2)$. This is because there are 2^n subsets, and for each subsets we will be looping through every node n in the graph while also looking at the next possible node n, creating 2^n x n x n timing, or $O(2^n n^2)$ timing. My dynamic programming approach uses the top-down memorization technique. The subproblems are the already known partially completed tours found while executing this algorithm that we can reuse in other sub-paths. Here are the steps:

1) Find the optimized distance from the starting node to every other node that is not the start itself. This will yield a number of optimized partially completed tours of length 2.
2) Store the set of visited nodes in the partially completed tour (represented as a bit field), the optimized distance of the partially completed tour, and the last visited node. We will store the distance of this subset in the memo table, which is of size n x 2^n because there exist n possible nodes that could be our last visited node and there are 2^n subsets of visited nodes.

3) Repeat the last two steps while slowly incrementing the size of the partially completed tour. So, instead of finding the distance of the first node to every other node, solve for the distance of the last visited node to every other node not in the subset already and add that distance to the current subset's distance.
4) Once all "completed" tours are found, look at all "completed" tours in the memo table (where the bit state is made up of entirely of 1's, AKA the end state…"completed" is in quotation marks because the tours are not yet a circuit but all nodes have been visited once), add the distance of the last node back to the starting node for each tour, and find the minimum distance while you loop through the memo table.
5) Recursively backtrack to find the tour's actual path.

Figure 2 demonstrates the graph of the dynamic programming timing versus the asymptotic timing. The two lines do not share the same values because the dynamic programming approach is measured in microseconds and the asymptotic timing is measured in number of calculations but are shown to have the same slope, meaning that the dynamic programming approach is of exponential timing and not factorial like brute force.

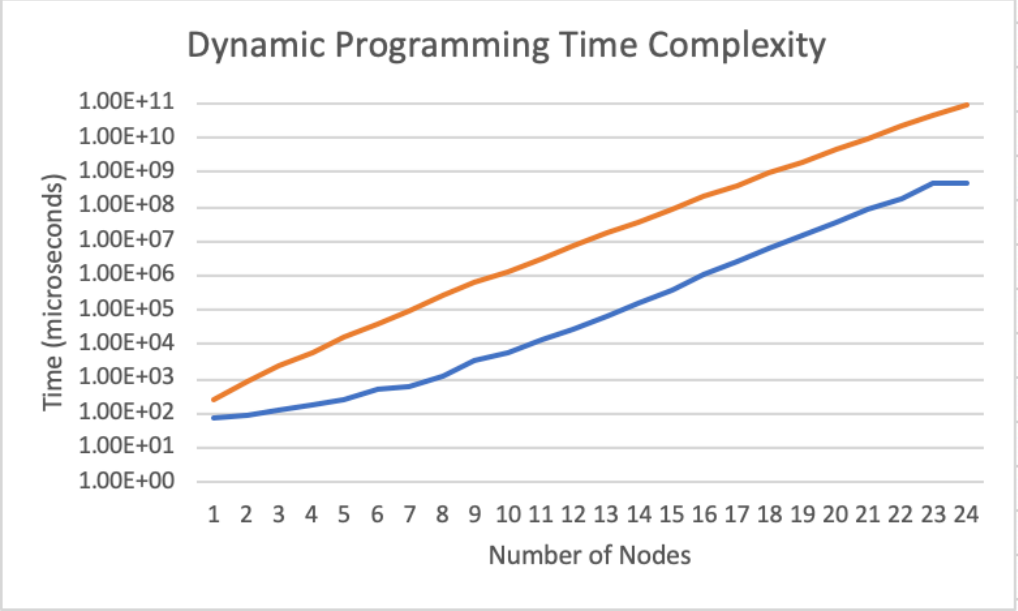

Figure 2. Dynamic Programming Approach and Asymptotic Timing

Figure 3 is a graph comparing the time it took to complete the algorithm for the brute force approach versus the dynamic programming approach. At five nodes and fewer, it appears the brute force algorithm is more efficient. However, after five nodes, it is apparent that the dynamic programming approach is much faster than the brute force algorithm.

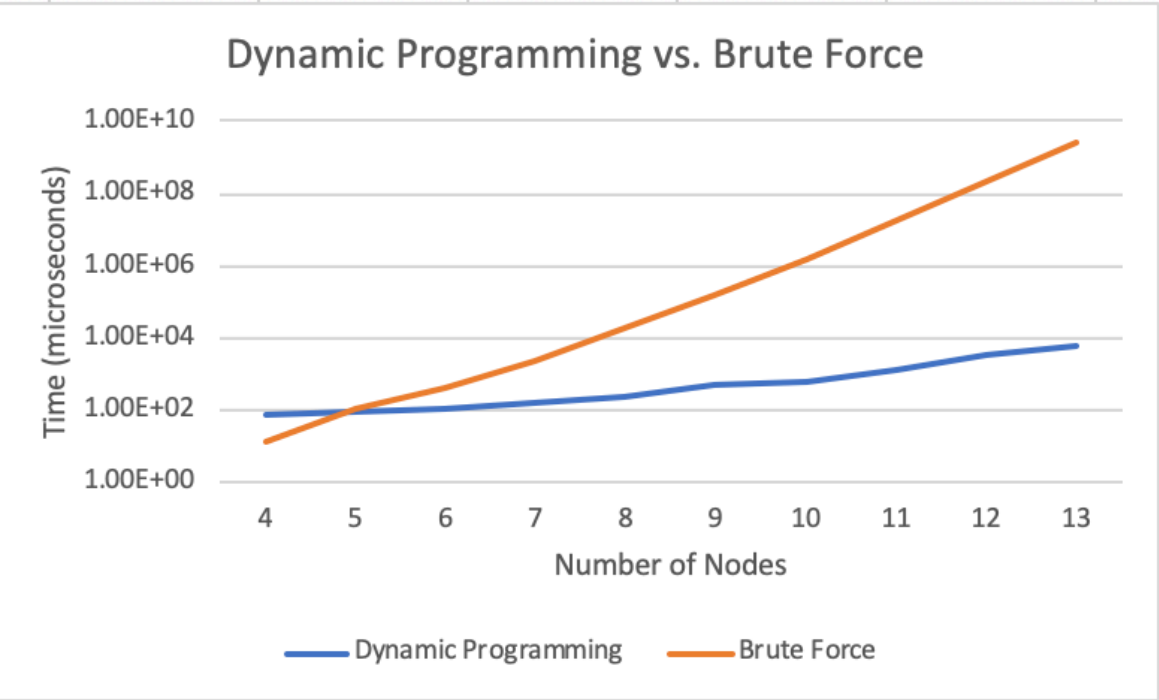

Figure 3. Dynamic Programming vs. Brute Force Timings for 4 to 13 nodes

. If we compare the time complexities, O(n!) versus $O(2^n n^2)$, we can see that O(n!) begins to overtake $O(2^n n^2)$ at about 8 nodes, which isn't too far off from the timing graph above.

n	**O(n!)**	**$O(2^n n^2)$**
1	1	2
2	2	16
3	6	72
4	24	256
5	120	800

6	720	2304
7	5040	6272
8	40320	16384

Figure 4. (above) chart of asymptotic timing values

Number of Nodes	Brute Force	Dynamic
4	13.66	6.80E+01
5	101.04	9.20E+01
6	378.38	1.17E+02
7	2308.99	1.72E+02
8	18319.70	2.44E+02
9	156448.00	4.78E+02
10	1540000.00	5.83E+02
11	16820300.00	1.30E+03
12	199244000.00	3.28E+03
13	2.69E+09	5.73E+03
14		1.33E+04
15		2.88E+04
16		6.14E+04
17		1.49E+05
18		3.79E+05
19		1.03E+06
20		2.64E+06
21		6.43E+06
22		1.46E+07
23		3.32E+07
24		8.05E+07
25		1.84E+08
26		5.01E+08
27		5.01E+08

Figure 5. (left) Table of Results for both algorithms

It is clear to see that Dynamic Programming is the winner when it comes to problems with around 6 or more nodes, and that we can solve far beyond what Brute Force is capable of in a given time. For additional reference, here are the two algorithms plotted on a graph without a logarithmic scale on the y axis:

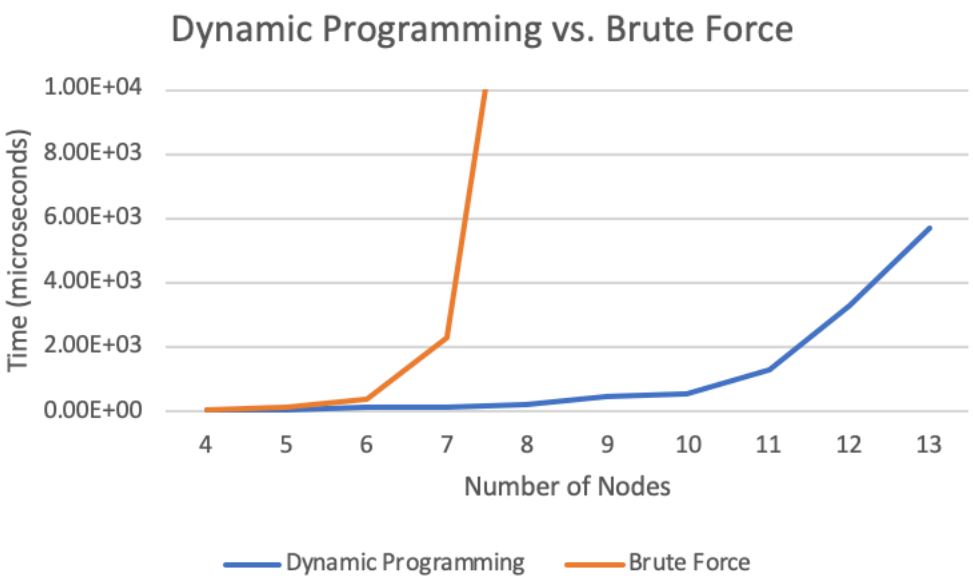

Figure 6. Non-logarithmic graphing of the two algorithms.

I used the Strategy design pattern to create my program. I chose the strategy pattern because I was familiar with it from the last two labs and also because it allows my program to cycle through different algorithms dynamically. If I wanted to, I could add a variable that accepts user input on which algorithm to perform (for example, “Press 0 for Brute Force, press 1 for Dynamic Programming”) and then I could perform that algorithm using function pointers instead of hard-coding a call to the function itself or creating any objects of type TSP. The strategy pattern allows us to hide the concrete implementations of the brute force and DP algorithms from the user so that main is kept simple and clean by decoupling the interface from the implementation. (See next page for class diagram)

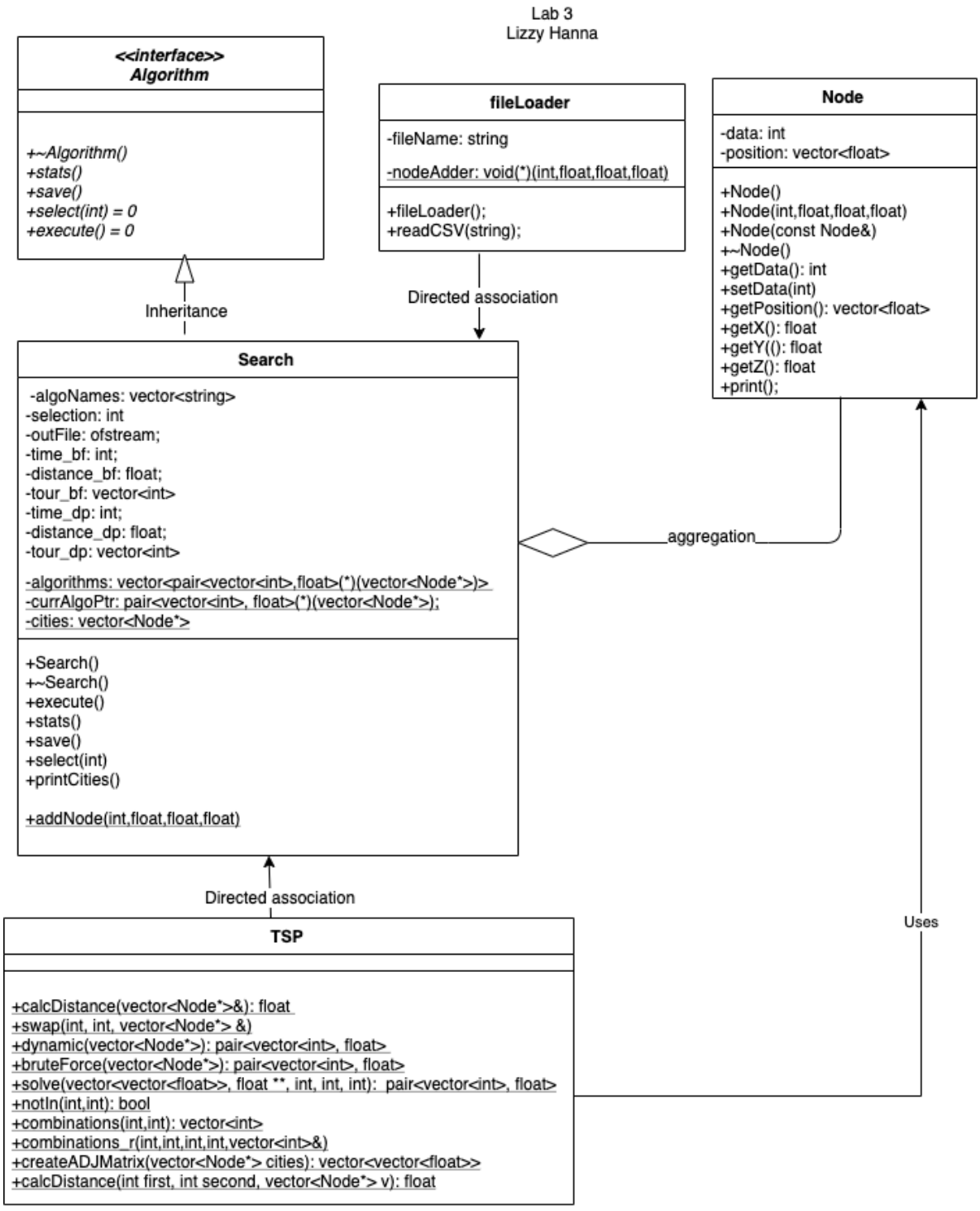

Figure 7. UML class diagram

Algorithm is an abstract class, AKA an interface, so it is notated by italics. Search class inherits from Algorithm, denoted by an unfilled arrow pointing towards the parent class. The

relationship between Search class and fileLoader class is a directed association (notated with an arrow pointing in the direction of the flow of information) because fileLoader supplies data to the Search class via a function pointer to functions in the Search class and the flow of information is in a single direction only, however fileLoader does not contain any objects of type Search or vice versa so this is not an aggregation nor composition relationship. The Search class contains instances of the Node class but the Node class can live independently of the Search class, so this is an aggregation relationship. The relationship between Node and TSP, however, is an association (using) relationship because TSP is passed instances of Node but doesn't create any instances of Node itself. The relationship between TSP (where my algorithms are) and Search is an association relationship because TSP is never instantiated as an object and its functions are only called via function pointers in Search.

The use of static functions and variables (underlined) was to facilitate the use of function pointers between classes to allow for classes to easily provide information to each other without needing to own instances of each other. For example, my fileLoader class has a function pointer to the addNode(int,float,float,float) function in Search so that it can provide data to the Search class before the Search class is even instantiated as an object. Through this, fileLoader never has to know what a "Node" is; it just passes the information to Search and Search does all the creation of Node objects in its static function. Additionally, the Search class never needs to create an object of type TSP because it can just use function pointers to static functions in TSP. The downside of this though is that since TSP is never instantiated, all of its functions needed to be static in order to be used.