Andrew Melland

Prof. Clark

CS3353

Nov 9, 2020

Lab 3

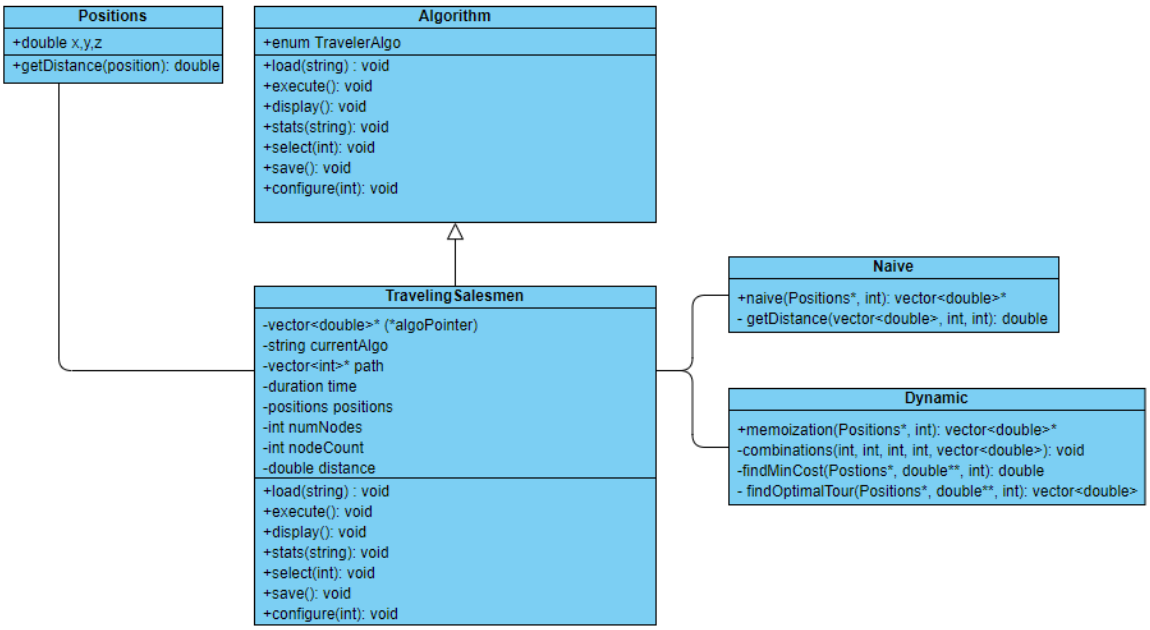
For this report we will be designing, testing, and analyzing algorithms that solve the traveling salesmen problem. There will be one Naïve salesmen algorithm and one memoization salesmen algorithm that uses top down dynamic programing to reduce redundant processes and reduce execution time. Each algorithm will be tested on a graph with number of nodes starting from 4 until each algorithm has reached the hardware limits of my system.

Design

For this lab I used a Strategy Design Pattern to create a TravelingSalesmen class that inherits from Algorithms and contains two different “strategies” or functions that can be selected and used to solve the traveling salesmen problem given a list of nodes and positions. The Traveling Salesmen class also loads in a file path as a string and loads the files data into memory. I also used an Adapter Design Pattern to create a File Reader and File Writer class. The File Reader class adapts the standard library ifstream class by adding the ability to read in comma separated values and return a vector of those values as strings. Similarly, The File Writer class adapts the standard library ofstream class by adding the ability to write vectors of values to a file in csv format. I used these adapter classes to make reading and writing my data much easier with less code. I also used a struct called Positions that had three double

values to represent position in 3d space as well as a function to calculate the distance between two Positions.

UML



Dynamic Programming Solution

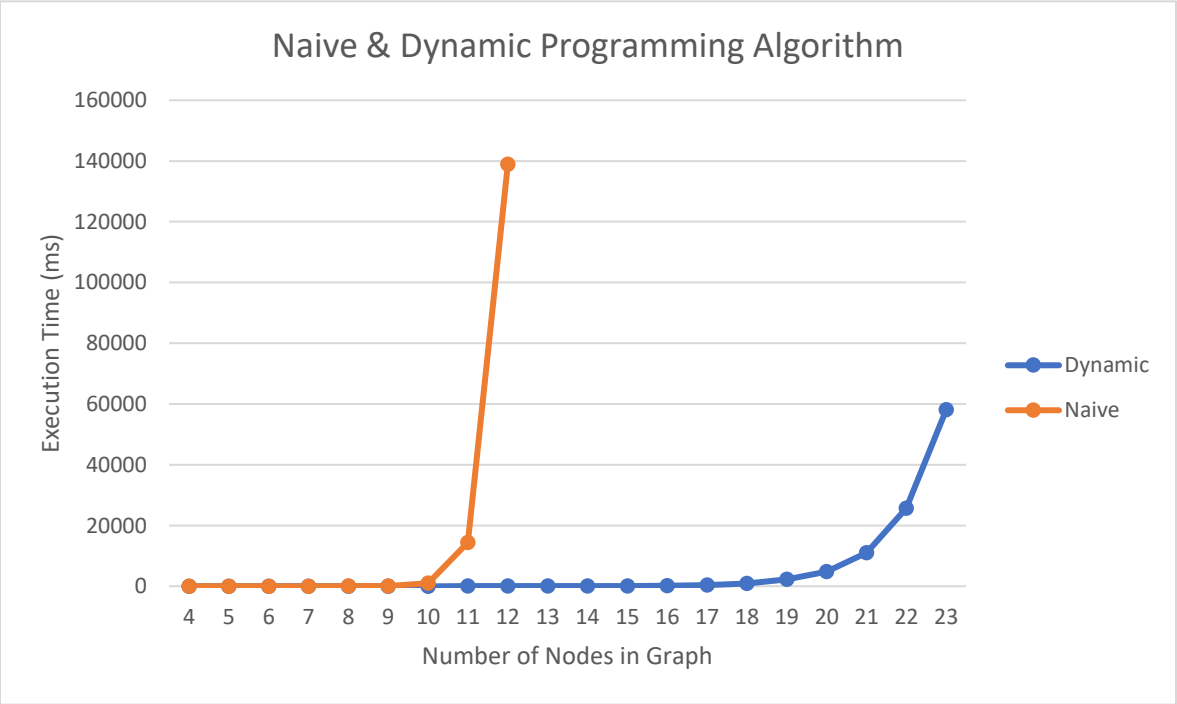
For the Dynamic Programming solution, I utilized a memoization function to solve the Traveling Salesman Problem. The memoization function solves the problem by first finding the most optimal path for 3 nodes, then 4, until it reaches the number of nodes in the graph. The algorithm utilizes a 2d array called a memo table of size $nx2^n$ with each value set to INFINITY, where n is the number of nodes in the graph. This memo table will be used to keep track of the optimal path between nodes. Then the algorithm sets the value in each row of the memo table with the distance from the starting node to each node in the graph. Each distance is stored at the index of (1 | 1<< index of the non-starting city). To do this it uses bit

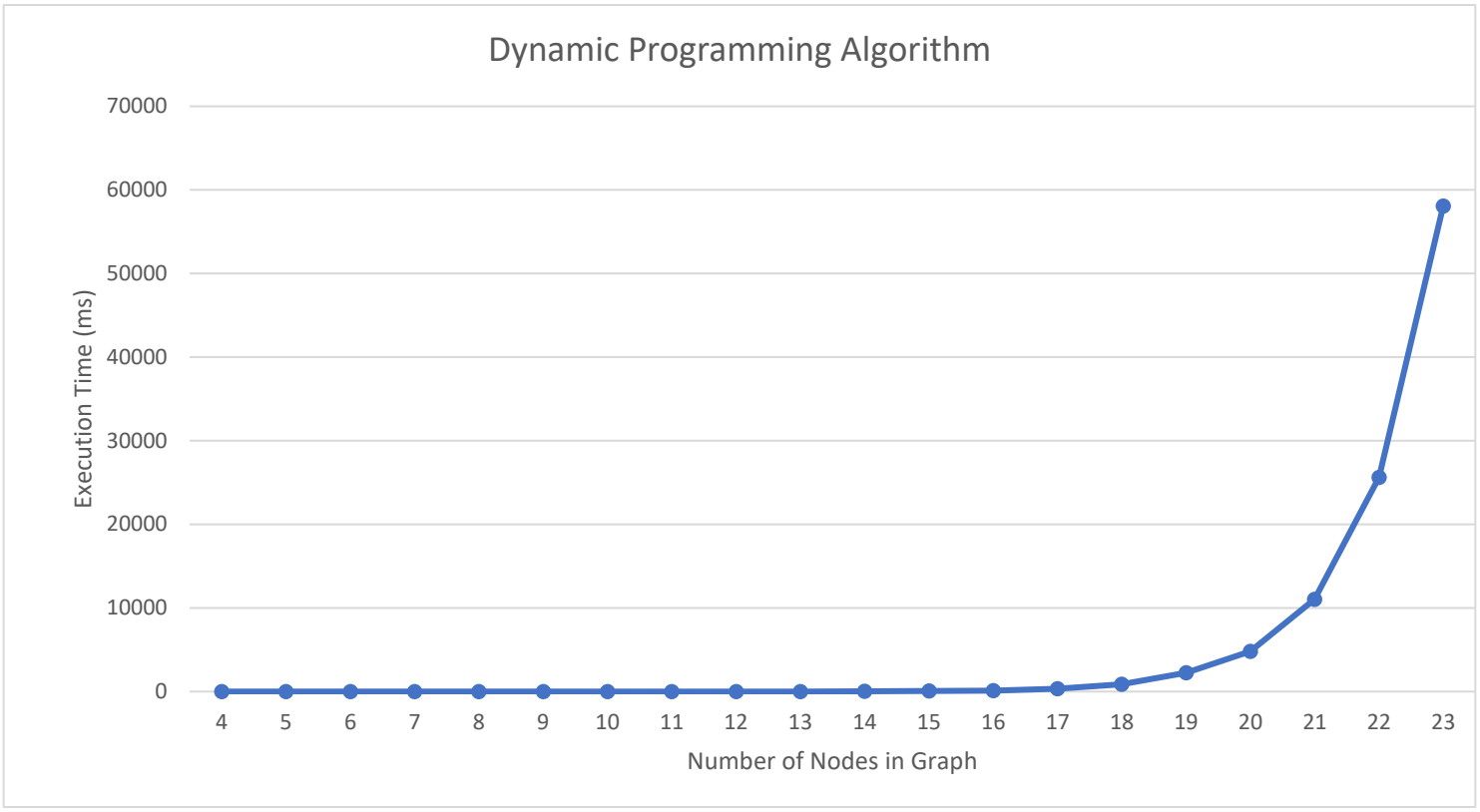
masking to keep track of which cities have been visited in binary. Now the algorithm finds the optimal distance for each number of nodes starting from 3 iterating over all the possible bitmasks were the number of bits are turned on. It then removes all combinations that do not have the starting node turned on as well as any combinations where the next is not turned on in the bitmask. Then it adds the distance between the end node and the next node to the minimum distance, and if this distance is better than the best distance, we insert the distance into the memo table. After the memo table is solved, the algorithm extracts the best path and best distance from the memo.

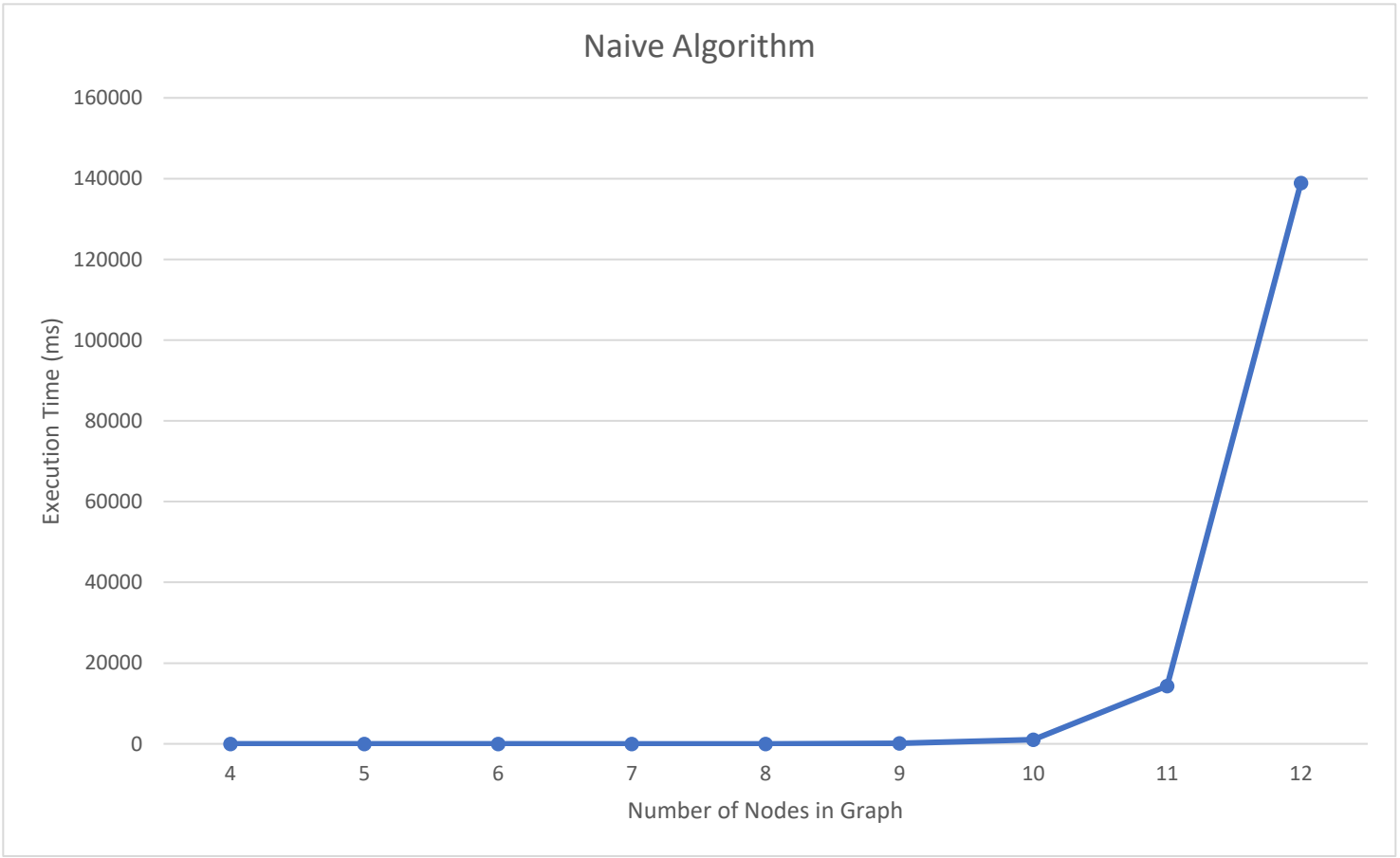
The Dynamic Programming solution improved on the brute force time by removing sub-problems from the main problem. The algorithm does this by using a memoization table that stores the shortest path for each subproblem i.e.: going from one city to another.

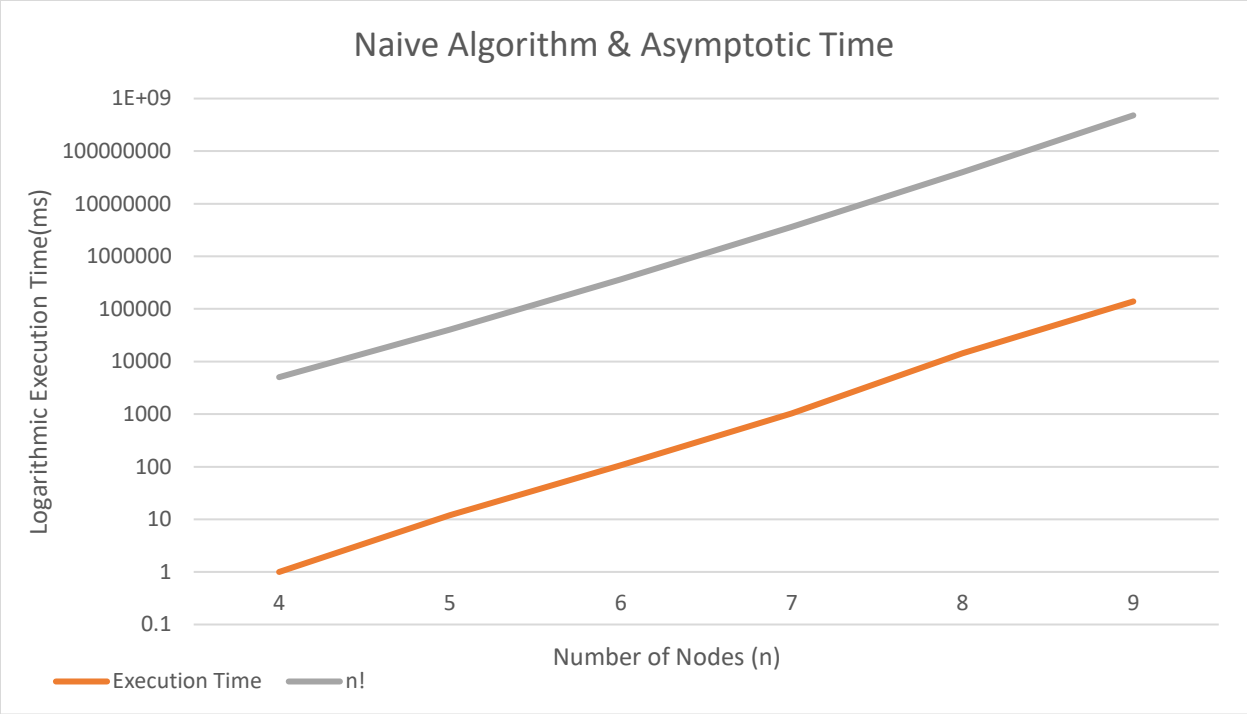
Data

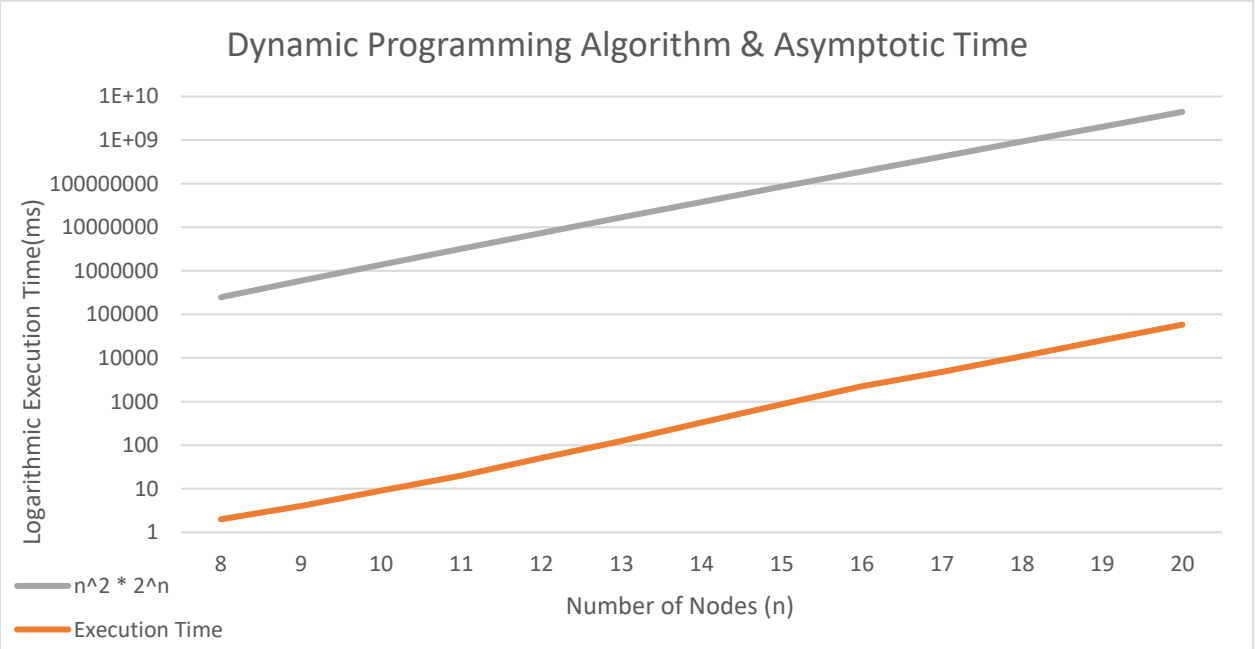
Number of Nodes	Naive(ms)	Dynamic(ms)
4	0	0
5	0	0
6	0	0
7	0.997	0
8	11.967	0
9	106.251	0
10	1024.807	0.997
11	14348.405	1.995
12	138930.199	3.992
13		8.976
14		19.947
15		50.864
16		124.695
17		333.108
18		866.738
19		2240.467
20		4817.569
21		11030.379
22		25613.131
23		58058.956




Dynamic Programming Algorithm
Execution Time (ms)
70000
60000
50000
40000
30000
20000
10000
0
4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
Number of Nodes in Graph


Naive Algorithm
Execution Time (ms)
160000
140000
120000
100000
80000
60000
40000
20000
0
4 5 6 7 8 9 10 11 12
Number of Nodes in Graph


Naive Algorithm & Asymptotic Time
1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
0.1
Logarithmic Execution Time(ms)
4
5
6
7
8
9
Number of Nodes (n)
Execution Time
n!


Dynamic Programming Algorithm & Asymptotic Time
1E+10
1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
Logarithmic Execution Time(ms)
8
9
10
11
12
13
14
15
16
17
18
19
20
Number of Nodes (n)
n^2 * 2^n
Execution Time

Analysis

When testing both algorithms, the highest node count the brute force naïve method got to was 12 while the Dynamic memoization method got all the way up to 23. The only reason the dynamic program could only get to 23 is because my program was crashing on runtime throwing error bad_alloc. When debugging I could find out what was specifically causing the issue but I assume that it is a problem with the large space required for the method as each sub problem requires n2^n space.

When comparing the results of the test of each algorithm to its respected Asymptotic time, we can see that both equations increase in execution time at the same rate as their Asymptotic time. This coincides to n! for the brute force algorithm and n^2 * 2^n for the memoization algorithm. This is exactly as expected as the execution time should increase the same as its Asymptotic time does.

When comparing the two algorithms together, it is clear that the Dynamic Programming solution is far superior to the naïve algorithm. This is again expected as the Asymptotic time for the memoization function is n^2 * 2^n which us much better than the brute force with factorial time.