Lab Objective

Using both naive brute force and dynamic programming techniques, solve the traveling salesman problem for a given list of nodes and positions.

Lab Details

• Node 1 is always the start and finish point of the path
• A list of nodes is read from a text file with the format: NodeID, x, y,z
    -x,y,z refer to the nodes position in space
    -Example: 1,4.00,2.00,3.00
• The Brute Force algorithm and the Dynamic Programming algorithm are both performed on the nodes
•The path, path size, and execution time are printed

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:

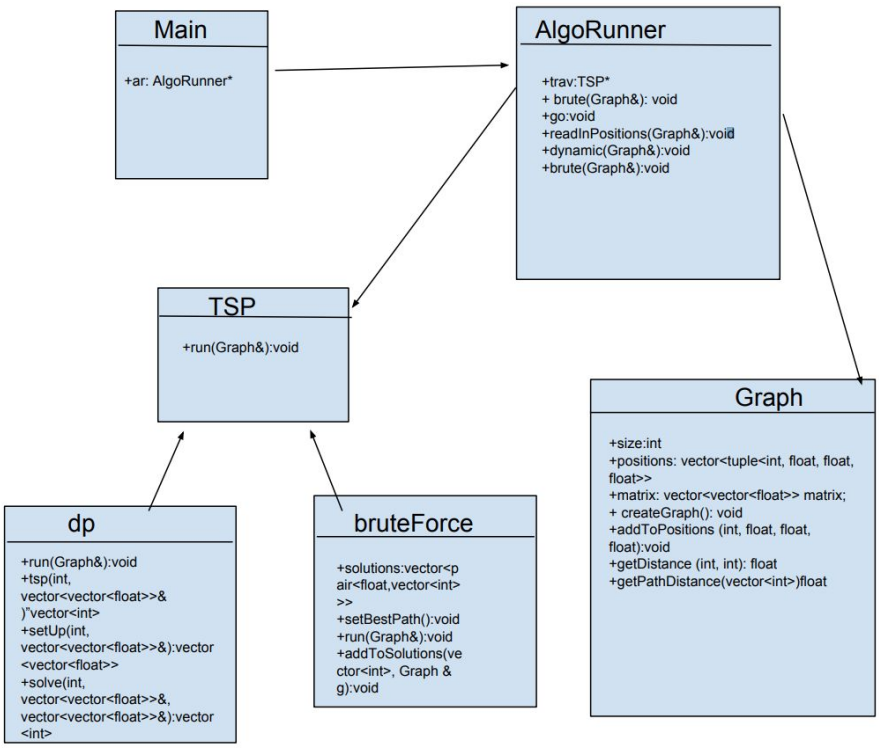
- The text file has to be read in
- The information read in needs to be organized into variables
- These variables need to be stored
- The Brute Force Algorithm needs to use these variables to find the shortest path
- The Dynamic Programming Algorithm needs to use these variables to find the shortest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

How my program is structured to achieve these tasks:

In my program I had 5 classes: AlgoRunner, Graph, TSP, bruteForce, and dp. I used the strategy design pattern by defining TSP as an abstraction of an algorithm that computes the Traveling Salesperson Problem, and then created two separate

implementations of this algorithm, which are called bruteForce and dp. AlgoRunner is where the text file input is read, and a Graph object is created based on this input. I created a AlgoRunner 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. AlgoRunner also has a pointer to a TSP object, and two different methods called ‘dynamic’ and ‘brute’, with each method taking 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.

UML Diagram



Brute Force Approach:

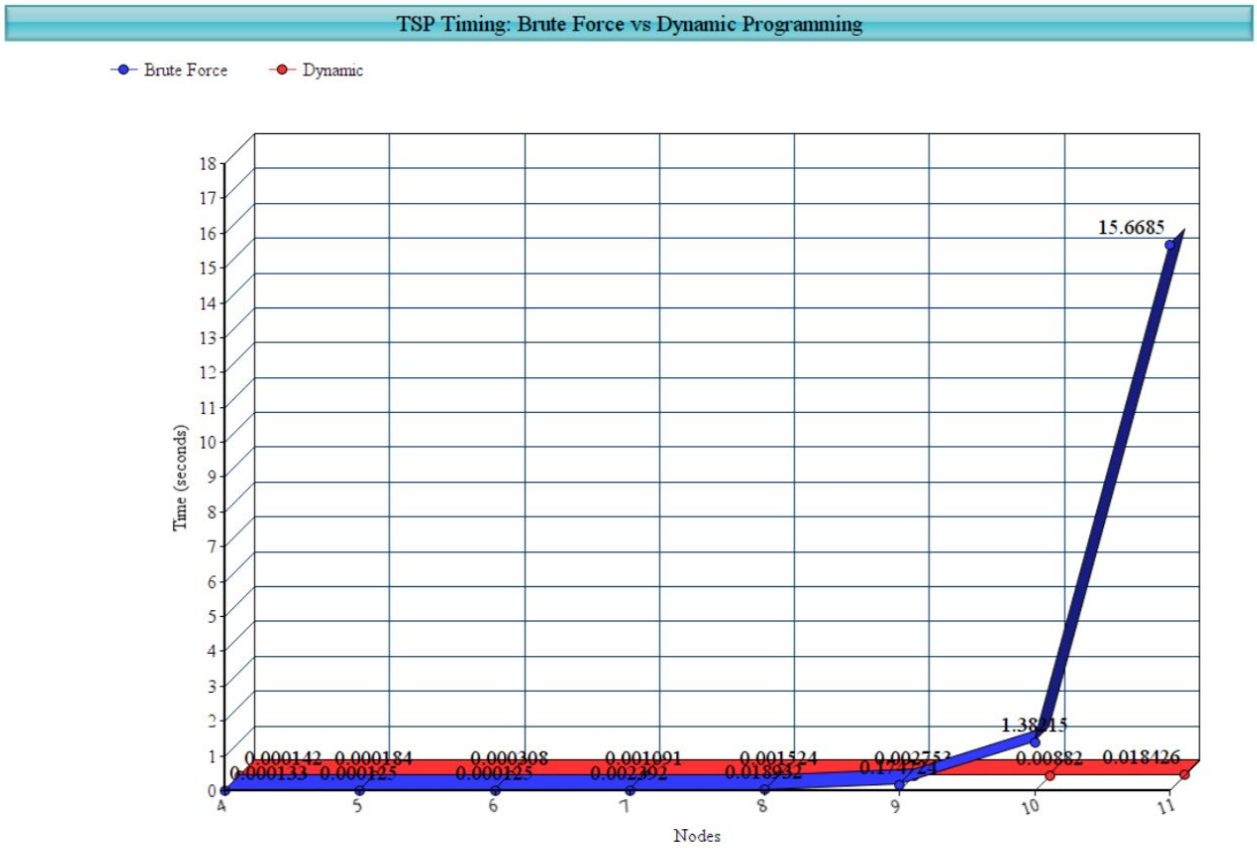
All possible permutations of nodes are examined, with the restriction that the path starts and ends with the same node. All the path lengths are compared, and the one with the shortest path is chosen.

Dynamic Programming Approach

The general dynamic programming approach is to start with the smallest possible subproblems, figures out a solution to them, and then uses these subproblem solutions to solve the overall problem. In my program I computed and stored the solutions to the smallest possible subproblems, which were the smallest subpaths for a given node count. When a solution was needed for a larger subpath, the smaller subpath solutions were used.

Complete Results of Input Testing

Nodes	Brute Force	Dynamic
4	0.000133082	0.000142434
5	0.000124888	0.000183688
6	0.000431545	0.000307553
7	0.00239156	0.00109145
8	0.0189318	0.00152406
9	0.174724	0.00275347
10	1.38215	0.00881957
11	15.6685	0.0184262



Analysis of Results

In the table, one thing I find interesting is that for 4 and 5 nodes the brute force algorithm is faster than the dynamic programming algorithm. After 5 nodes, the brute force timing increases a lot more rapidly for every node count increment, whereas the dynamic programming algorithm slowly increases. The sharp increase for the brute force approach can especially be seen from nodes 9 to 11. From node count 9 to node count 10, the brute force timing grows about 8 times larger, going from 0.17 of a second for 9 nodes to 1.3 seconds for 10 nodes. From node count 10 to node count 11 the brute force timing grows about 12 times larger, increasing

from 1.3 seconds for 10 nodes to 15.66 seconds for 11 nodes. The known run time for the brute force tsp approach is a polynomial factor of O(n!), which is a factorial of the number of nodes per path, and this polynomial factor is consistent with the data I collected. The known run time for the dynamic programming approach to tsp is $O(n^2 2^n)$. The table shows the comparison of the ratio of $(n!)/(n^2 2^n)$ to brute time/dynamic time for that node. The ratio is not the same on any node, but does show a similar trend as the node count increases.

n	n!	$n^2 2^n$	$(n!)/(n^2 2^n)$	brute time/dynamic time
4	24	256	0.09	0.93
5	120	800	0.15	0.67
6	720	2,308	0.31	1.41
7	5,040	6,272	0.80	2.19
8	40,320	16,384	2.46	12.43
9	362,880	41,472	8.75	64.44
10	3,628,800	102,400	35.44	172.5
11	39,916,800	247,808	161.08	870