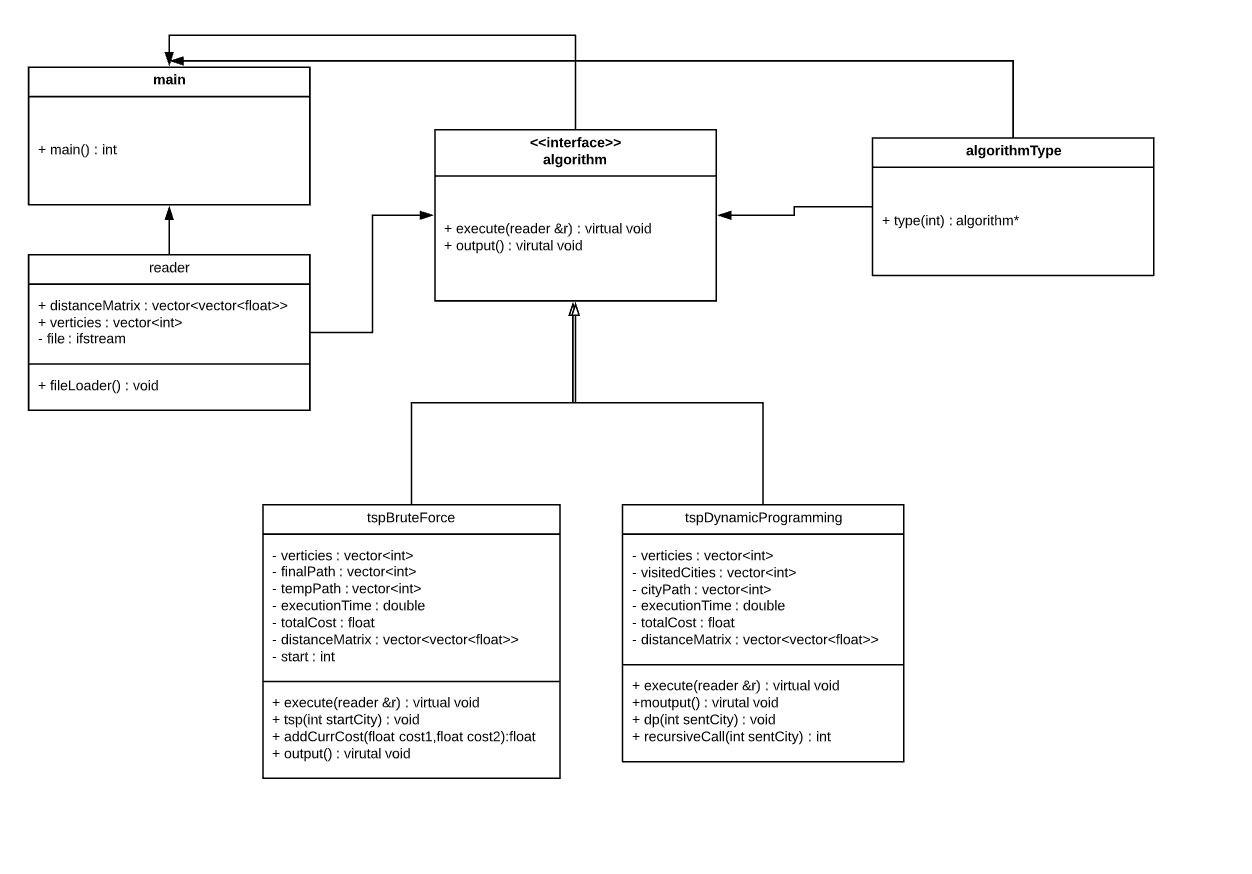
**LAB 3 REPORT**

**Excel Timing Table and Graph:**

|  |  |  |
| --- | --- | --- |
| Number of Nodes | TSP Brute Force | Tsp Dynamic Programming |
| 4 | 3.03E-06 | 2.27E-06 |
| 6 | 3.95E-05 | 3.35E-06 |
| 8 | 0.00108724 | 3.76E-06 |
| 10 | 0.0951507 | 4.40E-06 |
| 12 | 11.6153 | 7.03E-06 |
| 13 | 144.19 | 2.49E-05 |

**Analysis of Excel Timing Table and Graph:**

Initially, when obtaining the data for only 4 nodes, the results for both dynamic and naïve were very similar. Both the amount and the path returned were the same for both algorithms with 4 nodes for any position I gave the program. Although they were similar for 4 nodes in these aspects, the timing proved that the naïve approach was slower than the dynamic programming. For all node values higher than 4, the path returned, and the cost were always different between both algorithms, but one thing that did stay the same was the fact that dynamic programming was always faster. In fact, the higher up the number of nodes tested went, the bigger the difference in the time it took for the algorithms to work. This can be attributed to the algorithms time complexities. The time complexity of the naïve approach was O (n^n), because in order for the algorithm to work, it has to run through all permutations of the paths which is (n-1)!. But for the dynamic programming approach, the time complexity is O (n2^n), because of how it uses recursive equations and sub-problems which I will later discuss in this report. But on my computer, I could only get 13 nodes to work because when I bumped it to 14, I could never get it to finish running.

**UML Class Diagram:** 

**Analysis of Design:**

For this lab I decided to create a design structure of both factory and strategy design patterns. Together I use these to create a single algorithm interface which contains an execution and printing function for the derived classes that can continually be modified and added just like tsp brute force and dynamic programming. The factory design comes with my algorithmType class as I create an abstract object based on the choice from main.cpp. This design pattern helped just keep my code less clustered around the main and the derived classes overall. It also allowed for just the use of one simple for loop in main to run all the functions of the different algorithms. The strategy design came in as now because of my algorithm class, I have a single interface that can continually be added to for more algorithms.

**Analysis of Dynamic Programming and Subproblems:**

The way I structured my dynamic programming was that given a bunch of vertices representing cities S = {1,2,3,…,n} where x is any city in S, then the length of the shortest path is length(S, c). Using a directions matrix, M = (C, E), where C is the cities and E is the edges, where and edge can be represented as E = (a,b) where a and b are connected cities and the distance between a and b is d(a,b). My algorithm assumes you start at position 1 and from here it gets to any random city x. Because it has not returned back, you know that this is only a partial path. But you still need to know x because from here you have to find the best path and continue repeating this same loop. So, my algorithm crates subproblems for each of the paths in order to find and optimal solution.