**Design Architecture:**

I designed my program to have a factory pattern as well as a strategy pattern. I chose to use a factory pattern since it allows me to add another type of algorithm, like a sorting algorithm, to my program without having to change a lot of code. I implemented the factory pattern with the virtual Algo.h class that is implemented by the TSPAlgo class. The TSPAlgo class is where the file is loaded in, output stats are printed to the console, the desired algorithm is selected, and the selected algorithm is executed. I chose to create a strategy pattern for the TSP algorithms since both algorithms had similar functionality that could be abstracted behind a class. I chose to create TSP.h that is implemented like a strategy pattern. Both my dynamicProgramming and Naïve class implement the TSP class. This will also allow me to add more TSP algorithms to the program without having to change a lot of code. My UML diagram is below.

A screenshot of a social media post

Description automatically generated

**Brute Force Algorithm:**

The brute force algorithm works by generating all permutations of the graph with 1 as the starting node. The algorithm then calculates the distance for each permutation and stores the cost and path of the minimum distance permutation.

**Dynamic Programming Algorithm:**

The dynamic programming algorithm works by breaking the problem into multiple subproblems. There are n\*2^n subproblems that take linear time to solve which makes the algorithm n^2\*2^n. The algorithm always starts with 1 as the starting node and calculates the distance to all of its neighbors + a recursive call to the same function with the neighbor node as the start node and the remaining nodes in the list. This is repeated until there are no nodes left in the set and then the minimum of all these calls is the optimal path.

The tspRecur function is the function that calculates the optimal path. Instead of passing the remaining path every time, I am using a bit number to represent the set. If there are 4 nodes in the graph the first calls of the function would be:

First call of function: tspRecur(0, 14) //14 since 2^4-2 is 14.

Recursive calls:

First call: tspRecur(1, 12) //12 since ((2^4-1) – (1 << 1)) & 14 = 12

Second call: tspRecur(2, 10) //10 since ((2^4-1) – (1 << 2)) & 14 = 10

Third call: tspRecur(3, 6) //6 since ((2^4-1) – (1 << 3)) & 14 = 6

A screenshot of a social media post

Description automatically generated**Results:**

The graph above shows the execution time of the dynamic programming algorithm and the execution of the naïve algorithm versus the number of nodes in the graph. One thing I found to be interesting is the naïve approach is actually faster with 4 nodes than the dynamic programming approach. The naïve approach starts to slow down rapidly after 5 nodes. I believe the reason that the naïve approach is faster for 4 nodes is due to the fact that the dynamic programming approach has the time complexity of 2^n \* n^2 which is greater than n! at smaller numbers. The naïve approach gets increasingly slow after 10 nodes. This is due to factorial numbers getting very large very fast. The number of permutations in a 11-node graph using the naïve approach is 39,916,800 while the number of permutations in theory of the dynamic programming approach would be 247,808. This shows how much faster the dynamic programming approach is than the naïve approach.

**Comparison to Theoretical Time:**

In the graph below I compared the number of operations the naïve took versus the !n, where n is the number of nodes. This graph matches up completely with the number of permutations done.

A screenshot of a social media post

Description automatically generated

In the graph on the next page I compared the number of the number of operations in the dynamic programming approach to n^2\*2^n, where n is the number of nodes. This graph did match up with each other. I believe this is because I am calculating the number of operations in the wrong spot or I have to add to the number of operations in another spot. This could be due to the fact that I don’t add to the number of operations when a repeated subproblem is found.

A screenshot of a map

Description automatically generated

**References:**

<https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/>

<https://medium.com/basecs/speeding-up-the-traveling-salesman-using-dynamic-programming-b76d7552e8dd>

<https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_travelling_salesman_problem.htm>

<https://github.com/evandrix/SPOJ/blob/master/DP_Main112/Solving-Traveling-Salesman-Problem-by-Dynamic-Programming-Approach-in-Java.pdf>

<https://ideone.com/tYdrK8>

[www.lucidchart.com](http://www.lucidchart.com) – for my UML diagram