Emily Fashenpour

CSE 3353

Lab3 Report: Traveling Salesman Problem

Naïve/Brute Force Implementation vs Dynamic Implementation

To show timing on the dynamic and naïve implementation of TSP, I found the graphs for the number of nodes vs the number of times the recursive function was called and also the number of nodes vs the actual timing in seconds each algorithm took to run. I collected the times based on the number of node values ranging from 3 to 12.

|  |  |  |  |
| --- | --- | --- | --- |
| Naïve |  | Dynamic |  |
| # of Nodes | Timing | # of Nodes | Timing |
| 3 | 5.93E-06 | 3 | 7.33E-06 |
| 4 | 9.15E-06 | 4 | 1.44E-05 |
| 5 | 2.24E-05 | 5 | 4.33E-05 |
| 6 | 7.46E-05 | 6 | 0.000199 |
| 7 | 0.00039047 | 7 | 0.001351 |
| 8 | 0.002877 | 8 | 0.008402 |
| 9 | 0.0257225 | 9 | 0.069222 |
| 10 | 0.225801 | 10 | 0.632498 |
| 11 | 2.17262 | 11 | 5.55309 |
| 12 | 23.8756 | 12 | 58.2687 |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Naïve |  | Dynamic |  |
| # of Nodes | # of Calls | # of Nodes | # of Calls |
| 3 | 3 | 3 | 3 |
| 4 | 9 | 4 | 10 |
| 5 | 33 | 5 | 41 |
| 6 | 164 | 6 | 206 |
| 7 | 979 | 7 | 1237 |
| 8 | 6851 | 8 | 8660 |
| 9 | 54801 | 9 | 69281 |
| 10 | 493206 | 10 | 623530 |
| 11 | 4932051 | 11 | 6235301 |
| 12 | 54252557 | 12 | 68588312 |

Asymptotic Timing for Naïve: O(n!)

Asymptotic Timing for Dynamic O((n^2)(2^n))

What this graph shows is that for a time, the naïve approach will theoretically be faster than the dynamic approach. However, after a certain value, the dynamic approach becomes faster and is faster when it needs to handle larger values. Summed up: the value is equal to the point when the two lines meet. When nodes < value, naïve is faster. When nodes > value, dynamic implementation is faster.

However, the timing for my algorithm implementations does not reflect the theoretical timing. Instead if shows that the naïve approach is faster from all the nodes that I tested. (Nodes 4-12) and both complexities as n factorial.

Design/Design Patterns

Utilizes Design Patterns

* Strategy Pattern
  + Utilized this design pattern to easily be able to add more derived classes to the AbsAlgorithm class (for future expandability) and because inheritance is used, the derived classes can easily be initialized in the AlgoFactory class without needing to create multiple factory classes. Inheritance cuts down on the amount of code.
* Factory Pattern
  + Used the factory pattern to initialize an AbsAlgorithm to a certain algorithm type. Then can specify which algorithm implementation to run.
* Chain of Responsibility
  + Utilizes Chain of Responsibility in the AbsAlgorithm class so that the call to select an algorithm implementation and then to run it can be done on the same line. This makes it easier on the user to understand and it cleaned up my main function.

In the main, I created an AbsAlgorithm pointer that is set using the static AlgoFactory create function. The create function take an enum from the AlgorithmOptions enum and initializes the AbsAlgorithm object to its corresponding algorithm class object. Once the AbsAlgorithm object has been set, I can select and run any sub algorithm implementations.

In the TSP class (derived from AbsAlgorithm class), I have an object of type TSPAlgorithm that will be used to call the naïve or dynamic function depending on the enum passed into the select method.

In the TSPAlgorithm class, I use a ReadFile object function (that return the map of nodes IDs and their positions) to set the TSPAlgorithm class variable map. This is then used to find the shortest path and distance.

A screenshot of a cell phone

Description automatically generated

DP Development

Will be caching the distance values from one node to another into an adjacency matrix.

Need to find the smallest distance at each level of recursion (starting at the bottom going up) and resets a distance value to the smallest distance found at the lower level. This value will be passed up to the next until the total shortest path is found.

A picture containing text, whiteboard

Description automatically generated