Reid Russell

CS 3353

12 November, 2019

Lab 3 Report

Introduction

The task for Lab 3 was to solve the Travelling Salesman problem in two different ways: a brute force approach and a dynamic programming solution. The idea here was to demonstrate how the brute force method being O(n!) can be sped up using a dynamic programming approach to lessen the time needed to find the shortest path.

While doing research on what dynamic programming approach I should try, I came across several options to aid in the TSP. I ended up choosing to go with bit-shifting or bit-masking with memoization for the simple reason that it was the approach that I understood the best. Memorization speeds up the TSP by eliminating overlapping subproblems. This means that if the program has already found the minimum distance path for a given subtree, it doesn’t have to repeat the calculation, and can use the results from the previous calculation. Bit-shifting provided a way to keep track of state during the recursive calls. When a node was visited, the bitmask would left shift 1 a distance of the index of the newly visited node. This bitmask is then used as an index to store subtree calculations inside a lookup table so that given a certain state, if a solution to the subproblem already existed in the lookup table it would simply return that. You can clearly see the resulting increase in efficiency given the results.

Results

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Dynamic Programming Approach** | Number of Nodes | Runtime (s) | **Naïve Approach** | Number of Nodes | Runtime (s) |
|  | 4 | 2.59E-05 |  | 4 | 4.15E-05 |
|  | 5 | 4.56E-05 |  | 5 | 2.05E-04 |
|  | 6 | 8.12E-05 |  | 6 | 6.11E-04 |
|  | 7 | 3.51E-04 |  | 7 | 3.63E-03 |
|  | 8 | 0.00034 |  | 8 | 3.51E-02 |
|  | 9 | 0.000595 |  | 9 | 3.63E-01 |
|  | 10 | 0.003548 |  | 10 | 2.99232 |
|  | 11 | 0.003377 |  | 11 | 28.2869 |
|  | 12 | 0.006672 |  | 12 | 339.448 |
|  | 13 | 0.020319 |  | 13 | 4850.46 |
|  | 14 | 0.062887 |  |  |  |
|  | 15 | 0.137489 |  |  |  |
|  | 16 | 0.321433 |  |  |  |
|  | 17 | 0.582171 |  |  |  |
|  | 18 | 1.42448 |  |  |  |
|  | 19 | 2.39501 |  |  |  |
|  | 20 | 4.95861 |  |  |  |

In this first figure you can barely see the dynamic approach (DP) because the naïve solution blows up very quickly. This is because the brute force of the TSP is O(n!). I actually had to get rid of the 13 node solution from the naïve approach because it took 4859 seconds and make the DP solution invisible. For a little better picture, let’s look at the comparison only showing 11 nodes of naïve.

In this figure you can more clearly see just how much better the dynamic programming solution is as the number of nodes increases.

This is what an n! graph looks like:

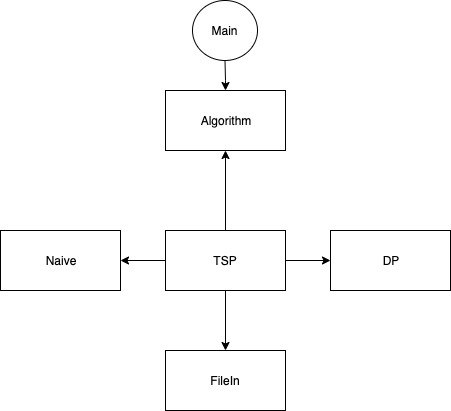
As you can see looking at the two graphs, the naïve approach displays an almost identical characteristic of increasing magnitude per new nodes.

From my research on memoization and bit shifting, I found that the complexity of my dynamic programming approach is O(n^2 \* 2^n). This is by no means good, but it is still better than n!, as we can see from the data. Here is what an n^2\*2^n graph looks like:

Just like the graph for the DP results, the n^2 \* 2^n graph is slightly more gradual in its increase than n!.

Design

For this program I decided to design my code using an overall strategy pattern. I chose the strategy pattern because if offered the best compartmentalization of code so that in the future it will be simple to add other solutions for the TSP.



The strategy pattern also forces code to be more elegant because each algorithm added conforms to the exact same interface, making the kickstart period of time for adding a new algorithm minimal once the interface has been set up. In this design, Algorithm is a pure virtual function that provides the layout for the interface. TSP is where that interface is implemented in a way that it can use the same code for different algorithms. The main function with this design is the best part, because it’s super compact- around 10 lines of code.

Aside from the strategy pattern implementation, the rest of my code followed the object-oriented approach, that allows the program to abstract features and variables into a bundle that can be reused. For the file input I created a “FileIn” class that was responsible for taking in the file-path, loading a particular file, storing the contents of that file, and if I decided to add file output it would be in charge of that as well. I chose to do this so that TSP merely has to hold a FileIn object and interact with that object on all things files instead of having to handle file reading, storage, output itself. I also built a simple Node class, that is used in storing the contents of the file and having information on each node ready to return when needed. The data stored in the nodes was a node id and a vector of the x, y, and z positions of that particular node. This way I merely had to keep a vector of nodes and the nodes themselves kept track of their positions and ids.

Dynamic Programming Development

The idea behind the dynamic programming approach that results in the efficiency increase is that of eliminating overlapping sub-trees. The way that I decided to do this was memoization. Memoization is a top-down approach where you start at the top of the tree and recursively iterate down the tree, calculating distances for sub-trees and saving the smallest distance for every sub-path until that sub-tree has all been explored and the minimum distance for that sub-tree gets cached. This helps because before going into recursive iteration, a check is performed to see if the subtree for a particular state and current position has already been evaluated, and if it has it just returns that value up to the higher level of recursion until all of the subtrees have been evaluated and the one with the shortest path is returned.

The more difficult part of this is storing the path. The solution I came up with to track the path through the recursion was to have two vectors that were parallel (values at each index were correlated). Every time I told the program to cache the results of evaluating a sub-tree, I also pushed back the minimum distance onto one of the vectors and pushed back the next node that was used to find that particular minimum distance of that subtree. Each value is set every-time the recursion call returns a distance value that is less than the current known minimum distance, and at the end of iterating through all of the possible next nodes the program exits a for loop and pushes the two values onto their respective vectors and sets the minimum distance value in the lookup table for the particular state and current position. Whenever the final minimum distance is passed out of the recursive function, I use that to track backwards through the parallel vectors and find the next node that correlates with a particular minimum distance value of the subtree. Doing this it works backwards and builds the shortest path. I went through several different attempts at saving the path correctly, but this method proved to be the simplest and therefore least prone to errors.