Colin Weil

Lab Report 3 – Travelling Salesman Problem

Brute Force Data

|  |  |  |  |
| --- | --- | --- | --- |
| Nodes | Time | O(n!) | 0.00001 O(n!) |
| 3 | 0.00002 | 6 | 0.00006 |
| 4 | 0.000057 | 24 | 0.00024 |
| 5 | 0.000192 | 120 | 0.0012 |
| 6 | 0.000918 | 720 | 0.0072 |
| 7 | 0.007179 | 5040 | 0.0504 |
| 8 | 0.046366 | 40320 | 0.4032 |
| 9 | 0.326806 | 362880 | 3.6288 |
| 10 | 2.627307 | 3628800 | 36.288 |
| 11 | 24.92624 | 39916800 | 399.168 |
| 12 | 274.720095 | 479001600 | 4790.016 |
| 13 | 3526.4484 | 6227020800 | 62270.208 |

Dynamic Programming Data

|  |  |  |  |
| --- | --- | --- | --- |
| Nodes | Time | O(n^2\*2^n) | 0.0000005 O(n^2\*2^n) |
| 3 | 0.000012 | 72 | 0.000036 |
| 4 | 0.00002 | 256 | 0.000128 |
| 5 | 0.000036 | 800 | 0.0004 |
| 6 | 0.000047 | 2304 | 0.001152 |
| 7 | 0.000114 | 6272 | 0.003136 |
| 8 | 0.000165 | 16384 | 0.008192 |
| 9 | 0.000352 | 41472 | 0.020736 |
| 10 | 0.000542 | 102400 | 0.0512 |
| 11 | 0.001202 | 247808 | 0.123904 |
| 12 | 0.002581 | 589824 | 0.294912 |
| 13 | 0.006553 | 1384448 | 0.692224 |
| 14 | 0.016305 | 3211264 | 1.605632 |
| 15 | 0.042694 | 7372800 | 3.6864 |
| 16 | 0.096524 | 16777216 | 8.388608 |
| 17 | 0.231571 | 37879808 | 18.939904 |
| 18 | 0.580149 | 84934656 | 42.467328 |
| 19 | 1.526935 | 189267968 | 94.633984 |
| 20 | 3.326841 | 419430400 | 209.7152 |

Brute Force Graphs

Dynamic Programming Graphs

Dynamic Programming vs Brute Force Timing Graph

**Results obtained:**

As the number of nodes increased, the dynamic programming function became much quicker than the brute force to solve the TSP. This is because the time complexity of the dynamic solution is O(n^2\*2^n) compared to the timing complexity, O(n!), of the brute force solution. The brute force algorithm is O(n!) because it has to create every single permutation of the n number of nodes which means that it has to travel to every node in every path every time. This is the worst possible solution because as n increases, the timing becomes basically unusable because of how long it takes. The dynamic programming solution evolves on the brute force by trying to minimize the number of paths repeated. Instead of visiting every node, every time of every single permutation of a path, the dynamic programming saves solutions to past founds sub-paths and stores the length to compare to later. This means that the program will only check a sub-path once, and if it finds it again, it will not need to run over it and waste time. This sacrifices memory space in order for the program to run faster because it needs to save the already search sub-paths. Because of this small improvement, the timing complexity is reduced from O(n!) to O(n^2\*2^n). This timing is still not good, but it is still a lot better than the simple brute force method. The time complexity is derived from the combination of finding the sub-paths where the binary string will be n long with either 0 or 1. So you have to check 2^n spots in the binary string. Then the n^2 comes from after finding the sub-paths, when you have to go through the memorization table and find the total best path.

**Design:**

Algorithm – I used a strategy pattern for this project. This allows me to pick an algorithm at run time and only create one object pointer in main. This means if I had other children of algorithm in my code folder, then I could also use them in main while easily switching between the algorithms but keep the naming conventions consistent because of the polymorphism between algorithm and its children.

HamiltonianCircuit – This class is a child of algorithm so it overrides all of algorithms methods. This class also has a function pointer in it with a vector of functions pointers that come from the static class, HamiltonianAlgorithms. When I turned in this code, there were only two methods/ solutions that I had made to this problem that were in the function vector and could be chosen by the Select function in HamiltonianCircuit, but I were to create another solution, then I could easily push it to the vector and choose select on it in order to use it in this class.

Path – Path is a simple class that contains a vector of ints that represent id’s in a created path as well as double of the length of the path. This class is used by HamiltonianCircuit and HamiltonianAlgorithsm in order to find and display the solutions.

fileReader – fileReader is the input class that can be added onto. For now, there is a 2D vector that holds the data of whatever type of file that needs to be read. The only type of file that needed to be read in for this project was the positions file. There is a function that receives a string of the file path of the positions.txt file, and then the file reader will parse the positions file and fill the 2D vector with the distance from every node to the other. More functions can be added for different file types in the future to populate the 2D vector.

output – The output class chooses a file path to send outputs to and then opens the file. The print function has a string parameter then prints the string to the output file. Since there were paths in this program, I also created a nicely formatted output for the paths (ex. 1->2->3). The output file can be expanded on for printing different types of items to a file in the future.

Diagram

Description automatically generated

**Dynamic Programming of TSP:**

The main problem when running brute force on a Hamiltonian Circuit is that you have to repeat the visitation of nodes/ sub-paths which makes the time so horrible. In order to reduce the timing, the dynamic programming of TSP makes sure that repeating the same process of checking a sub-path doesn’t happen, and a sub-path is only checked once. Since every path will need to be checked at least once to find the solution, this solution was made bottom up. It starts 3 nodes and takes every combination of 3 nodes that can happen in the problem and saves the length between those 3 nodes in order to a memorization table. Then after it finds all the solutions and saves their length, the program moves to 4 nodes. It now finds every solution with the combination of those 4 nodes, but if it contains a past sub-path within that combination, it doesn’t recalculate the whole path. Instead, it just adds take what had been saved before and adds on to create the new length and saves that to the memorization table. This process continues until the number of nodes in the combinations reaches the total number of nodes. Then once the memorization table is complete, it is very simple to backtrack through the memorization table, picking the smallest value for the length of the path. This is a perfect example of a problem/ sub-problem relationship because the sub-paths are the solution to the sub-problem to stop the overall problem of repeating the calculations of paths.