Jayson Werth

**Lab 3 Report – TSP and Dynamic Programming**

Brute Force vs Dynamic Programming Timing:

|  |  |  |
| --- | --- | --- |
| **nodes** | **brute (sec)** | **dynamic (sec)** |
| 4 | 0.000012620 | 0.000051579 |
| 5 | 0.00002716 | 0.000094388 |
| 6 | 0.000106651 | 0.000177028 |
| 7 | 0.000626018 | 0.00032695 |
| 8 | 0.00472562 | 0.000656288 |
| 9 | 0.035931 | 0.00122966 |
| 10 | 0.211171 | 0.00131349 |
| 11 | 1.67526 | 0.00297772 |
| 12 | 19.1431 | 0.00677971 |
| 13 | 242.074 | 0.0158074 |
| 14 |  | 0.0738069 |
| 15 |  | 0.147539 |

|  |  |  |
| --- | --- | --- |
| 16 |  | 0.302761 |
| 17 |  | 0.639005 |
| 18 |  | 1.54324 |
| 19 |  | 3.09094 |
| 20 |  | 6.47036 |
| 21 |  | 14.0575 |
| 22 |  | 30.589 |
| 23 |  | 66.2545 |
| 24 |  | 144.993 |
| 25 |  | 315.96 |

* Max Nodes for Brute Force = 13
* Max Nodes for Dynamic = 25

Brute Force Timing vs O(n!) Timing

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **nodes** | **brute timing (sec)** | **n! operations** | **n! avg** | **n! avg avg** | **n! normailized** |
| 4 | 0.000012620 | 24 | 5.25833E-07 | 1.41972E-07 | 3.40734E-06 |
| 5 | 0.00002716 | 120 | 2.26333E-07 | 1.41972E-07 | 1.70367E-05 |
| 6 | 0.000106651 | 720 | 1.48126E-07 | 1.41972E-07 | 0.00010222 |
| 7 | 0.000626018 | 5040 | 1.2421E-07 | 1.41972E-07 | 0.000715541 |
| 8 | 0.00472562 | 40320 | 1.17203E-07 | 1.41972E-07 | 0.005724324 |
| 9 | 0.035931 | 362880 | 9.90162E-08 | 1.41972E-07 | 0.051518918 |
| 10 | 0.211171 | 3628800 | 5.81931E-08 | 1.41972E-07 | 0.515189181 |
| 11 | 1.67526 | 39916800 | 4.19688E-08 | 1.41972E-07 | 5.667080994 |
| 12 | 19.1431 | 479001600 | 3.99646E-08 | 1.41972E-07 | 68.00497193 |
| 13 | 242.074 | 6227020800 | 3.88748E-08 | 1.41972E-07 | 884.0646351 |

We can derive the approximate Big O function that bound this Brute Algorithm by looking at how we have to get and compare all the permutations of the vertices:

* In order to do a brute force method of checking every possible path and finding the shortest, we have to check all permutations, which is equal to (n-1)!
* Since we have to check (n-1)! number of permutations, this functions should be of Big O order of O(n!)

Since this is the algorithm’s approximate time complexity, the timing data is graphed with the function t = !n with normalized values to be comparable to the found timing data. We get this normalization by first dividing the found timing data by the number of operations of 2nn2 for each number of nodes tests which gives the average execution of one operation for the tested system. Next we find the average of those values and finally multiply each operations number by this average of averages value.

The graph that is obtained when graphing the timing data and the normalized operations data shows that timing of both is only a little similar. The function of n! just takes off as it hits 12 and 13, while the brute force timing seems to be also trending in this way, but a big fraction of the actual function. However, even if this is the case, O(n!) still encompasses the function and could serve as its worst case scenario.

This difference in timing could be the result of multiple different aspect of both the algorithm or the code’s design. For example, the Big O function is a worst case behavior and perhaps the timing data never got to that worst case and ran on an average or sometimes best case. This seems to be the case when debugging, perhaps to the use of the std::algorithms function, next\_permutaions, the algorithm runs fairly fast and long compared to what should be shown in the n! graph.

Dynamic Programming Timing vs O(2nn2) Timing

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **nodes** | **dynamic timing (sec)** | **2^n\*n^2 operations** | **2^n\*n^2 avg** | **2^n\*n^2 avg avg** | **2^n\*n^2 normalized** |
| 4 | 0.000051579 | 256 | 2.0148E-07 | 3.59015E-08 | 9.19078E-06 |
| 5 | 0.000094388 | 800 | 1.17985E-07 | 3.59015E-08 | 2.87212E-05 |
| 6 | 0.000177028 | 2304 | 7.68351E-08 | 3.59015E-08 | 8.27171E-05 |
| 7 | 0.00032695 | 6272 | 5.21285E-08 | 3.59015E-08 | 0.000225174 |
| 8 | 0.000656288 | 16384 | 4.00566E-08 | 3.59015E-08 | 0.00058821 |
| 9 | 0.00122966 | 41472 | 2.96504E-08 | 3.59015E-08 | 0.001488907 |
| 10 | 0.00131349 | 102400 | 1.28271E-08 | 3.59015E-08 | 0.003676313 |
| 11 | 0.00297772 | 247808 | 1.20162E-08 | 3.59015E-08 | 0.008896678 |
| 12 | 0.00677971 | 589824 | 1.14945E-08 | 3.59015E-08 | 0.021175565 |
| 13 | 0.0158074 | 1384448 | 1.14178E-08 | 3.59015E-08 | 0.049703757 |
| 14 | 0.0738069 | 3211264 | 2.29838E-08 | 3.59015E-08 | 0.115289189 |
| 15 | 0.147539 | 7372800 | 2.00113E-08 | 3.59015E-08 | 0.264694566 |
| 16 | 0.302761 | 16777216 | 1.8046E-08 | 3.59015E-08 | 0.602327191 |
| 17 | 0.639005 | 37879808 | 1.68693E-08 | 3.59015E-08 | 1.359941861 |
| 18 | 1.54324 | 84934656 | 1.81697E-08 | 3.59015E-08 | 3.049281406 |
| 19 | 3.09094 | 189267968 | 1.6331E-08 | 3.59015E-08 | 6.795003626 |
| 20 | 6.47036 | 419430400 | 1.54265E-08 | 3.49544E-08 | 14.66095343 |
| 21 | 14.0575 | 924844032 | 1.51999E-08 | 3.59015E-08 | 33.20328642 |
| 22 | 30.589 | 2030043136 | 1.50682E-08 | 3.59015E-08 | 72.88159014 |
| 23 | 66.2545 | 4437573632 | 1.49303E-08 | 3.59015E-08 | 159.3155421 |
| 24 | 144.993 | 9663676416 | 1.50039E-08 | 3.49544E-08 | 337.788367 |
| 25 | 315.96 | 20971520000 | 1.50661E-08 | 3.49544E-08 | 733.0476714 |

We can derive the approximate Big O function that bound this Dynamic Programming Algorithm by looking at the loops and recursion within the tsp method:

* There are n different nodes for the method that could have been visited last and each one can have 2n subgraphs of visited nodes (because of the 2n combinations of state that can occur for each index) which gives a complexity of O(n2n) for each subproblem.
* We also have to do these subproblems n times in order to get the min cost for a path of length n, so if we solve these subproblems n times to get this min cost, we get the actual time complexity of O(2nn2).

Since this is the algorithm’s approximate time complexity, the timing data is graphed with the function t = 2nn2 with normalized values to be comparable to the found timing data. Similar to the brute force, we get this normalization by first dividing the found timing data by the number of operations of 2nn2 for each number of nodes tests. Next we find the average of those values and finally multiply each operations number by this average of averages value.

The graph that is obtained when graphing the timing data and the normalized operations data shows that timing of both is fairly similar. Although the actual timing data seems to be trending at lower values than the Big O function, the slopes when inspecting the graph towards the upper edge of the graph seems to show that both functions are trending similarly. It appears the timing data is running at some fractional multiple of the Big O function (since they both seem to have similar slopes) which would still mean that the function is of Big O order of 2nn2.

This difference in timing could be the result of multiple different aspect of both the algorithm or the code’s design. For example, the use of vectors for the memo and prev tables might put an effect on the timing, however it is unlikely it would make that drastic of a change. More realistically, the Big O function is a worst case behavior and perhaps the timing data never got to that worst case and ran on an average or sometimes best case. This seems to be the case when debugging, since the memo table is never really that full and also the usage of the memo table itself probably helps reduce timing due to the removal of a handful of repeated operations.

Design Implementation:

For this lab I used a mix of the Strategy Design pattern with the use of a similar Algorithms class to the first two labs and a Factory Builder Pattern to facilitate the creation of the TSPBrute objects and TSPDynamic objects. A UML Class Diagram is located in the Data folder as Lab3UML.pdf.

* Strategy Aspect – Algorithms Abstract Base Class
  + Similar to the first two labs, I used a abstract base class similar to the provided implementation from the first lab in order to separate the user interface from the actual implementation of the TSP Search algorithms. In this way, I could use very similar implementations for the Brute Force Method and the Dynamic Programming Method of the TSP and only have to use the single interface of the Algorithms pointer to interact with both.
  + For extensibility of the classes/patterns, the abstract base class allows me to be able to add new methods that would be common to algorithms and then provide distinct implementations of that method in any base class, all that can be accessed by a single call by the Algorithms pointer.
  + Also, the Algorithms class still contains the Configure function that could be used to change specific things about the algorithms themselves. For example, for the heuristic and metaheuristic searches, we can implement this function to change the calculation of the heuristic, to change the neighborhood identification and Tabu List size, or to change the techniques for Selection, Mutation, or Crossover.
* Factory Aspect – AlgoFactory Class
  + Since we had to implement two different types of TSP Searches, I decided to create a factory class that would handle the creation of these specific instances of Search objects rather than having the user (in main.cpp) directly create a new TSPBrute or TSPDynamic. In this way, I could simplify the interface in main even further, so now the user just has to call the Create() method with one of the enum values (which are labelled to help identify which algorithm you will create) rather than having to call new on all the different Search Algorithms in main. This also helps remove more dependencies in the main function as the are only included in the actual factory.
  + Also with is class, I tied the Create() method functionality to be in accordance with the Algorithms Strategy Class as it returns an Algorithms pointer with the address of a new algorithm (similar to, in the previous lab, where you’d handle this at the top of main). In this way, the Factory is extensible to all classes that inherit from Algorithms, such as Sort from lab 1 or the general Search from lab 2 or possibly the heuristic/metaheuristic searches in lab 4, simply by adding new entries to the enum and adding a few more case statements in the Create() method.
* In addition to the Strategy and Factory Pattern classes, I also used the functionality of a FileHandler class, which served as the interface for loading files and saving data from algorithms to an output file.
  + For this lab, both of the TSP algorithms also contained an instance of this class, for which they used to load their positions file and to print the name, path, cost, timing, etc. that was collected during the actual execution of the algorithms. This allowed for both of those files to be much shorter than the implementations of the Sort class and Search class.
  + By using this interface, another level of separation is created from the actual user’s interface. However, it also allows the loading and saving of files to simpler for the algorithms. For this lab, the handler is set up to handle the loading of data from a positions file and saving data from TSP algorithms, but by simply adding another load function for a different set of files can extend its functionality.
  + By taking the reading and writing from the actual algorithms, the classes become simpler and easier to read/update since there won’t be 90 lines of reading to sift through in each algorithm, and updating the methods in the handler will distribute changes to all the algorithms that use that loader rather than having to go and change each individual one.

Explanation of the Dynamic Programming Algorithm:

For the Dynamic Programming Algorithm, I found a top-down memoization, recursive algorithm that utilized techniques such as bit masking, bit-wise operations, and memo and index tables for path restoration. The links to the GitHub repo and YouTube explanation video for the algorithm I utilized can be found at the top of the TSPDynamic header file.

For the general TSP problem which we are trying to solve, we need to find optimal solutions for all the subpaths of length N while using the known information from the optimal partial tours of length N-1. We can first do this by creating a base case of our recursion which will return the optimal path between two nodes, which is simply obtained by using the two nodes’ positions and find the distance between them. Next, to find the optimal solution of length = 3, we need to remember 1) the set of visited nodes from the subpath of length = 2 through the use of a 32-bit integer as a bit field and 2) the index of the last visited node which we’ll store in a table called prev. And finally for length > 3, we use this information from subpaths of length n-1 and simply add another node that hasn’t been visited from the last visited node.

This Dynamic Programming Algorithm goes as follows:

* Make a memo table of size N x 2N and a prev table of the same size. The memo table will be used to hold previous solutions of that algorithm to reduce redundancy in recursion as well as help reconstructing the path later. The prev table will have parallel data locations to the memo table but instead will hold the index of the last visited city for reconstruction of the path later.
* Calling the tspDynamic method, use a “bit field” as a state variable to show which cities have been visited and use the final state (1 << N) -1 to see if all the cities have been visited (e.g. with 5 nodes, the final state would be 11111).
  + If this final state is reached, then return the distance from that node to starting node to complete the circuit
* If not all cities have been visited, check the memo table to see if the combination of city and state has been calculated beforehand (as to reduce redundancy of operations) and return its cost if the value isn’t -1 (i.e. hasn’t been calculated before).
* If it’s a new city/state combination, loop through all cities to check each combination of new city/new state.
  + Check if the current next city (as given by the loop counter) is a non-visited city by AND-ing the current state and the counter, which will give 0 if it’s not visited.
  + If it has been visited, move to next iteration of the loop to check another combinations, else create a new state variable by OR-ing the current state with the counter (new city)
* Add the distance from the current city to the new city to the optimal subtour cost of length n-1.
  + The recursion will move down till it finds a full circuit, and then move back up adding its distance from the start to the distance from the previous node to that node. It will loop this until it finds the absolute minimum cost for each subpath, which then it will store that value and its index and move up again.
* As in the recursion, after finding this cost, if it’s the absolute minimum cost of the paths explored of length n, then update the minimum cost and index.
* After finding this absolute minimum value, we will add it to the memo table at the combination of the current city and its state and then store the index value (latest visited node from the loop) to the prev table in the same spot.
* After finding this minimum cost, the path needs to be reconstructed from the prev and memo table that was filled in the search method. To do this we will loop until state gets to large or if the path is fully constructed.
  + First we add the first node and then get the next index based on the value of the index in the prev table at (index, state).
    - If the value retrieved was -1, then we found the end of the circuit
  + Next we get the new state by OR-ing the current state with the value of the next index, showing we are going to visit that node. Following we update the locals and reloop to the top.
* Finally, we add the start node to the end (and in my case the min\_cost as to store it for stats) and then return this to the calling location.