Final Report: Lab 3

**Data table and graph outlining number of nodes versus execution time for both naïve and dynamic programming algorithms.**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Size | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| TSP Naïve(ms) | 39 | 201 | 919 | 7694 | 68716 | 627611 | 6.01E+06 | 7.23E+07 | 1.02E+09 |
| TSP DP(ms) | 11 | 19 | 39 | 79 | 228 | 475 | 762 | 1351 | 2861 |

\*ms = microseconds. Size = number of nodes run.

\*numbers 0, 11, 19, 39,… above the X-axis represent Y-values (in microseconds) of the TSP DP graph to account for visibility issues for the TSP DP graph.

**Algorithm Summary:**

Naïve approach: I used a recursive brute force algorithm that would find every single Hamiltonian path and output the shortest one. As such, my brute-force algorithm finds every single permutation of a given list of Euclidean instances and calculates their total cost, comparing them to find the path with the shortest path. This algorithm takes factorial time to execute because finding all permutations of a list of given Euclidean instances takes a factorial number of operations. Findings in my data are in accordance with this conclusion. I start at 4 Euclidean instances and expand the input all the way to 12 nodes. What I found was execution time of the naïve algorithm growing at a near factorial rate in alignment with the asymptotic factorial graph for this given algorithm. The graph and table below visually illustrate this.

|  |  |  |
| --- | --- | --- |
| Size(number of nodes) | Naïve timing(ms) | Asymptotic |
| 4 | 39 | 39 |
| 5 | 201 | 195 |
| 6 | 919 | 1170 |
| 7 | 7694 | 8190 |
| 8 | 68716 | 65520 |
| 9 | 627611 | 589680 |
| 10 | 6011470 | 5896800 |
| 11 | 72258800 | 64864800 |
| 12 | 1015840000 | 778377600 |

\*ms = microseconds.

Dynamic Programming Approach: For the Dynamic Programming algorithm, I selected an optimized version of the Held-Karp algorithm. This optimized version of Held-Karp algorithm was written originally by William Fiset in Java and re-implemented by me in C++. The function that actually executes Held-Karp is labeled solve\_dp(). Inside, two for loops using variable names “next” and “end” respectively create a O(n^2) time complexity while the two for loops that create combinations of Euclidean instances have a time complexity of O(2^n) because it creates and loops through combinations at an exponential rate. For example, when 4 nodes are run, each iteration of the combination function will give result similar to {0111, 1011, 1101, 1110} (where each binary grouping encode connectivity in nodes), which have to be compared 16 times, or 2^N times where N = 4. Combined, the algorithm as a whole demonstrates a time complexity of O(n^2\*2^n). One other thing to note about this algorithm is that the graph constructed based on the data collected for algorithm execution time with changing number of Euclidean instances illustrates a slope that is in accordance with the time complexity mentioned above but flattened by a linear factor of approximately 0.19. The table and graph below illustrate this in a more intuitive manner.

|  |  |  |
| --- | --- | --- |
| Size(number of nodes) | DP timing(ms) | Asymptotic |
| 13 | 5175 | 5175 |
| 14 | 12880 | 12004 |
| 15 | 27103 | 27561 |
| 16 | 42102 | 62718 |
| 17 | 80917 | 141606 |
| 18 | 153709 | 317512 |
| 19 | 343792 | 707543 |
| 20 | 629608 | 1567964 |
| 21 | 1.23E+06 | 3457360 |
| 22 | 2.45E+06 | 7588946 |
| 23 | 4.82E+06 | 16589060 |
| 24 | 9.63E+06 | 36125893 |
| 25 | 1.93E+07 | 78398205 |
| 26 | 3.82E+07 | 169590998 |
| 27 | 7.63E+07 | 365774668 |
| 28 | 1.55E+08 | 786741672 |

\*ms = microseconds.

\*DP timing (orange line) retains the approximate shape of an asymptotic graph of n^2\*2^n but is clearly flattened out compared to the asymptotic timing graph. William Fiset documents that this algorithm is optimized for running against 30-32 Euclidean instances, which may explain why when run against list of nodes of size less than or equal to 28, the result is favorable even compared against its asymptotic timing graph given for this algorithm.

**Design Decision**

Input/Output Interface: I decided to use an interface called file\_i\_o.h to give some abstract structure to the file input/output process. File\_i\_o is an abstract class that has three pure virtual functions load, output, and return\_it (returns the list that has been parsed and processed) respectively. These three functions are implemented in parse\_process.h/.cpp. A singleton class for parse\_process.h/.cpp called pp\_singleton.h/.cpp is created for use of the file input/output process. The singleton class ensures the relatively high-cost operations of loading the input file to a usable C++ data structure and creating an output object are done once and only once. As a result, both TSP naïve and TSP DP classes can run their algorithms back and forth without having to reload data from a file. The singleton class for parse\_process.h/.cpp is used in the TSP classes. Since different TSP algorithms only require yet one version of file loading and outputting template with minimal changes, interfacing the parsing and outputting process and defining a singleton class adds a lot of simplicity for future use of this codebase, i.e. in lab4.

TSP Interface and algorithm layout: I designed the TSP interface in a way that would seamlessly take in future variations of TSP algorithms. Knowing the next lab is an expansion on lab3 that will revolve around other TSP algorithms, this design choice made sense. The TSP interface that I called TSP\_Interface.h has two pure virtual functions, run\_tsp() and display(), which are implemented by tsp\_n.h which implements the naïve algorithm and tsp\_dp.h which implements the dynamic programming algorithm. These two implementation classes implement the functions inherited from the interface and execute their assigned algorithms and output the results via the singleton class described in the previous paragraph. This design decision allows me to simply declare a TSP interface pointer and point to whatever implementation I want and use base class functions to run different algorithms. This adds a layer of simplicity that will become more and more rewarding as more TSP algorithms are added to the codebase.

\*A UML diagram outlining the design decision is available inside /report\_files folder inside /src and is named Lab3-UML.pdf.

**Subproblems/Dynamic Programming Implementation**

The Held-Karp algorithm uses memorization techniques and the bottom up approach to implement the travelling salesman algorithm. This requires two sets of information. First, the program must find out the set of visited nodes in the sub-path and second, program must know the index of the last element visited for backtracking. If the size of the list of Euclidean instances was to be denoted by N, then there are N possible nodes that could have been visited and 2^N possible subsets of visited nodes. Therefore, the space complexity for this DP algorithm is O(N\*2^N). This algorithm encodes visited nodes in bit-fields, representing a visited node with 1 and an unvisited node with 0. These encodings start with length 3, increasing the length all the way to N while memoizing every state of bit-field encoding created. Using this memoization table, the algorithm backtracks optimal sub-paths to find the optimal path.