CS 3353 Algorithms: Lab 3 Report

By Xuan (James) Zhai

**1: Introduction**

In this lab, we are going to solve the traveling sales man problem (TSP) and find a Hamiltonian path in a graph. We are going to use both Naive Brute force method and Dynamic Programming method (DP) to solve this program. Moreover, starting from 4 nodes, we will try to go through all the computer acceptable number of nodes to testify the time complexity for each algorithm. Finally, in this lab report, we will explore the idea of Dynamic Programing which change the program to a polynomial time of complexity.

**2: Code**

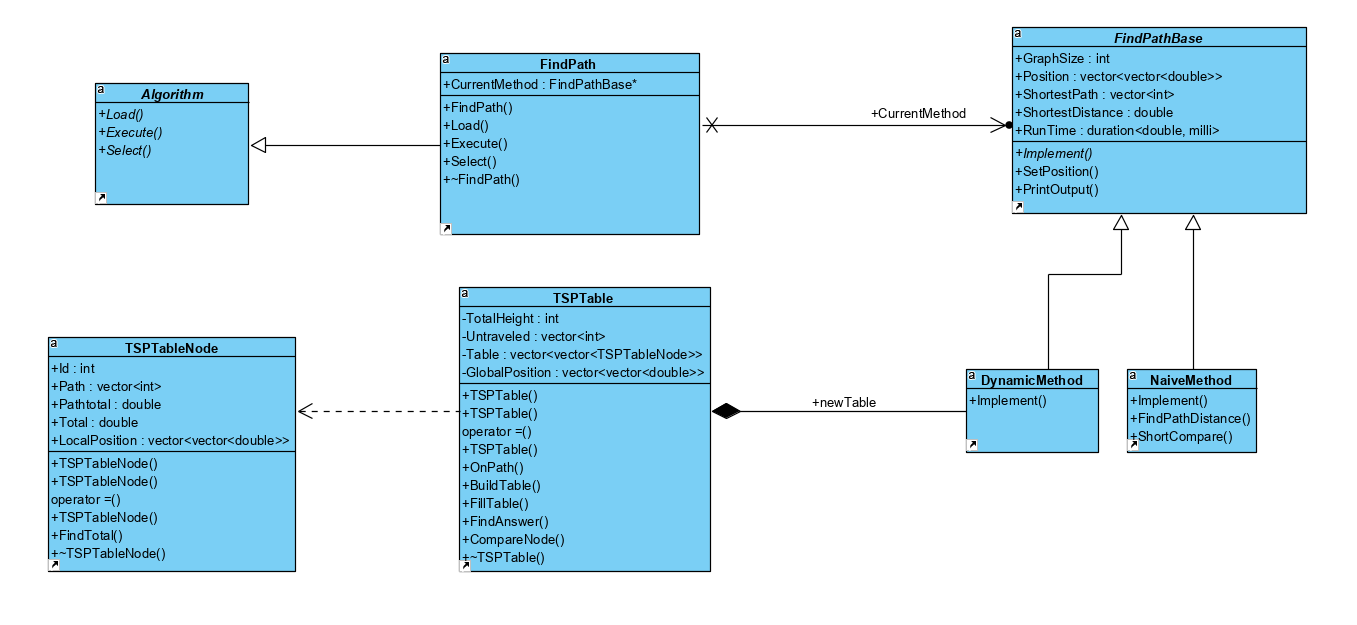


Figure 1: UML Diagram for Lab 3

1. **Pattern interface:** In the “main.cpp”, I use a “Algorithm” pattern which goes through the pattern’s implementations from Load, to Execute, and to Select. I use a “FindPath” class to inherit the interface and achieve the functionality. In main, I use a for loop that loop the implementations twice so that I can run both the Naive Brute Method and the Dynamic Programming method. In the FindPath, I use a “FindPathBase” calls “CurrentMethod” to point to each algorithm; the default algorithm for CurrentMethod is the Naive Brute Method.
2. **Pattern-Load:** In the Load function in FindPath which is inherited from Algorithm, I pass a string file name and load the positions.txt which is the same as we did in Lab2.

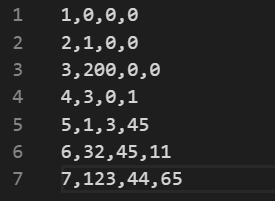


Figure 2: Sample Load file - “positions.txt”

Each line contains a node ID and its axis. I firstly use a temporary 2D double vector to store all axis. Since the first column is always in the form (1, 2, 3...), so the row x in the temporary vector represents the axis for number x+1. Next, we create a N by N matrix (2D double vector) to store the position from one number to another. For instance, (xi, yi) in the matrix contains the distance from xi + 1 to yi + 1. To find the distance, we apply the Distance Formula and use it to fill out our Position matrix. Eventually we set the Position matrix into each “FindPathBase” algorithm pointer.

1. **Pattern-Execute:** In the Execute function, the program will run each algorithm to find the shortest path with the shortest distance, and it will print out the result to the screen. At last, the program will also print the runtime (the time for finding the shortest path) to the screen. The time includes the time not only for find path in a space but may also the time for setting the space.
2. **FindPathBase:** In the FindPathBase class, it has 1: GraphSize to store the number of nodes; 2: Position to store the position map we created before; 3: a vector of integer called “ShortestPath” to store the shortest Hamiltonian path; 4: ShortestDistance initialized with INT\_MAX to store the distance for the Hamiltonian path; 5: RunTime for the implementation of High Resolution Clock. Furthermore, FindPathBase has a visual function that was inherited by DynamicMethod and NaiveMethod.
3. **Naive-Brute-Method (NativeMethod):** In the implementation of Naive Method, I use a vector stack to store the current path. I create two arrays; the bool array “visited” reflects which number is on the current and the int array “lastloop” stores the last loop number for each node. For example, for 4 nodes 1, 2, 3, 4, the vector will be filled with firstly all the nodes using multiple for loop.

1 → 2 → 3 → 4

Every time the function finds such a path, it pushes back the last node 1 and use a “ShortCompare” method to compare and update the shortest path.

If the last digit in the vector has ran through all the numbers (lastloop at that number is equal to the number of nodes), it will pop back and reset it lastloop to 0.

1 → 2 → 3

Next, the function loads the number from the lastloop and keep running the last number’s loop.

1 → 2 → 4

1 → 2 → 4 → 3

…

…

1 → 4 → 3 → 2

Eventually, the whole loop will visit all the permutation and pop all the numbers; the shortest path stored the shortest distance for all the permutations. We will add 1 at the end.

1. **Dynamic Programming Method:** In the Dynamic Programming method, I build and implement a TSP table object to store the TSPTableNode terms (similar to the term in <https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/>). For example, if we use four nodes 1, 2, 3, 4, the TSP table would be

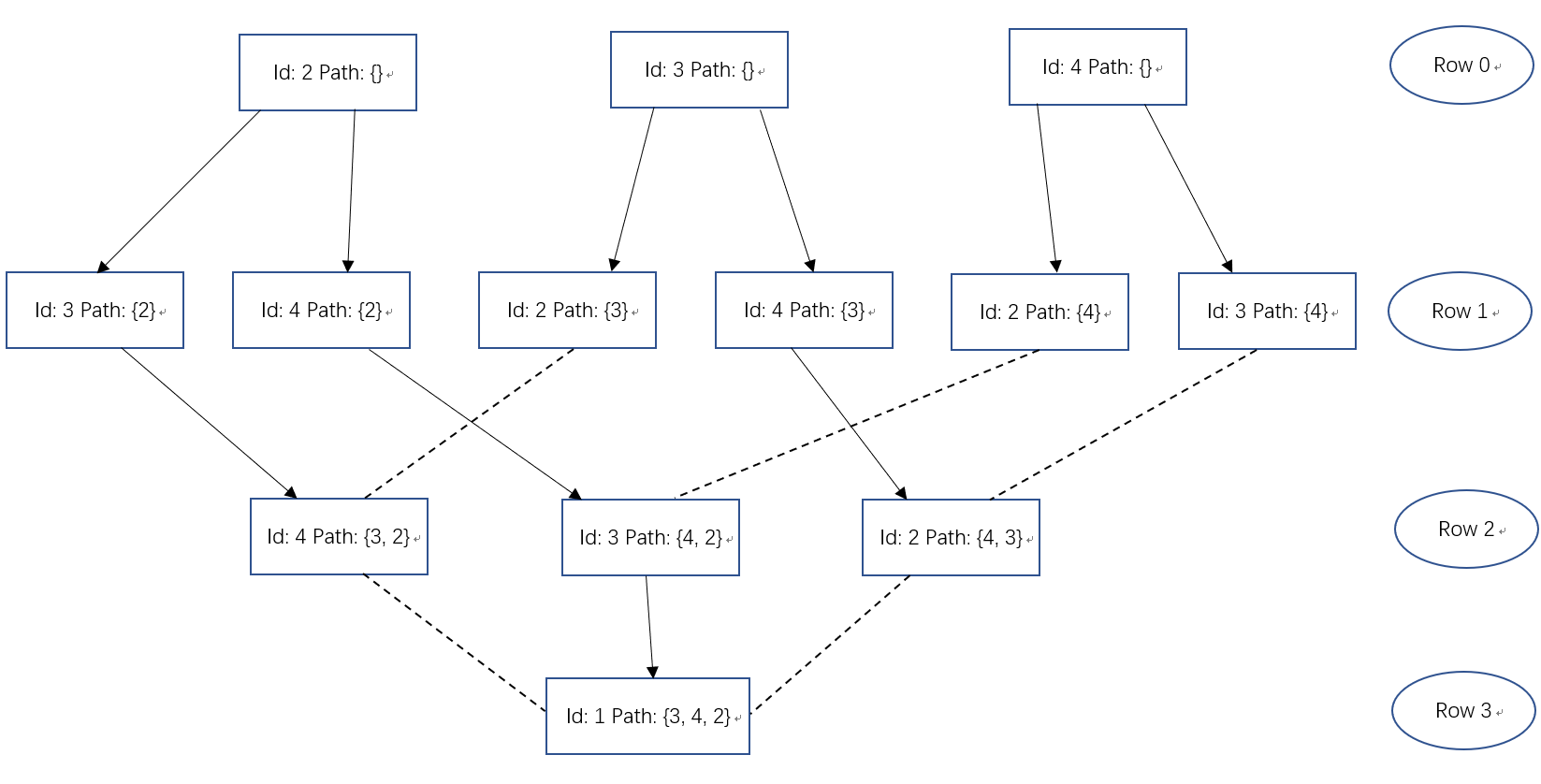


Figure 3: Sample TSP Table

Each TSPTableNode has an id; it also has a Total which reflects the distance from id to 1. The TSPTableNode has a vector of integer called Path which means the interval from id to ((2, {4,3}) means 2, 4, 3, 1, and its Total is the distance for that path). The Pathtotal is the distance for the Path in the term.

On the first row of the table, there is no Path for any term, so the Total is the distance from id direct to 1; the Pathtotal is 0 and the size of Path is 0.

Start from the second row to the last second row, we use a “FillTable” method to fill each row. It passes the upper row and the lower rom, and for each term, it creates nodes with ids that are not visited. The new nodes have path as (upper Node’s id + upper Node’s Path) and have Total as (Distance from new Id to upper Node’s Id + upper Node’s Total).

After we get our new node, the program will compare it with all the existing nodes in the lower row. If there is a node in the lower row that has same id and same path (order does not matter), the program will compare there Total. If the new Node has shorter Total, it will replace the existing node, otherwise, we will not add it to the row. If there is no such Node in the lower row, the program will push back to the last of the row.

Finally, we use a specific “FindAnswer” method to fill the last row. It has similar functionality as FillTable but it creates a Node with id “1”. Besides, after we find the shortest path, we will update the path and distance to the FindPathBase. In Figure 3, although all the new created nodes have path with 2, 3, 4, the order 1 → {3, 4, 2} has shorter distance. Therefore, the program will store it as the bottom row’s node and update the path “1 → 3 → 4 → 2 → 1”.

1. **Pattern-Select:** When the program finished one algorithm, it will call a Select method that will delete current algorithm “CurrentMethod” and create a new one.

**3: Data**

By using 4-13 nodes to find the Hamiltonian Path, we get a list of time and put it into two tables. Theoretically, the Naive Brute Method has a time complexity O(n!) and the dynamic programming Method has a time complexity . Therefore, initially the Naive Brute Method runs faster than the dynamic programming Method, but when n is at around 6 or 7, the time of using Naive Brute Method increase dramatically. On the other hand, the Dynamic Programming Method still cost a lot, but it is faster than the Naive Method.

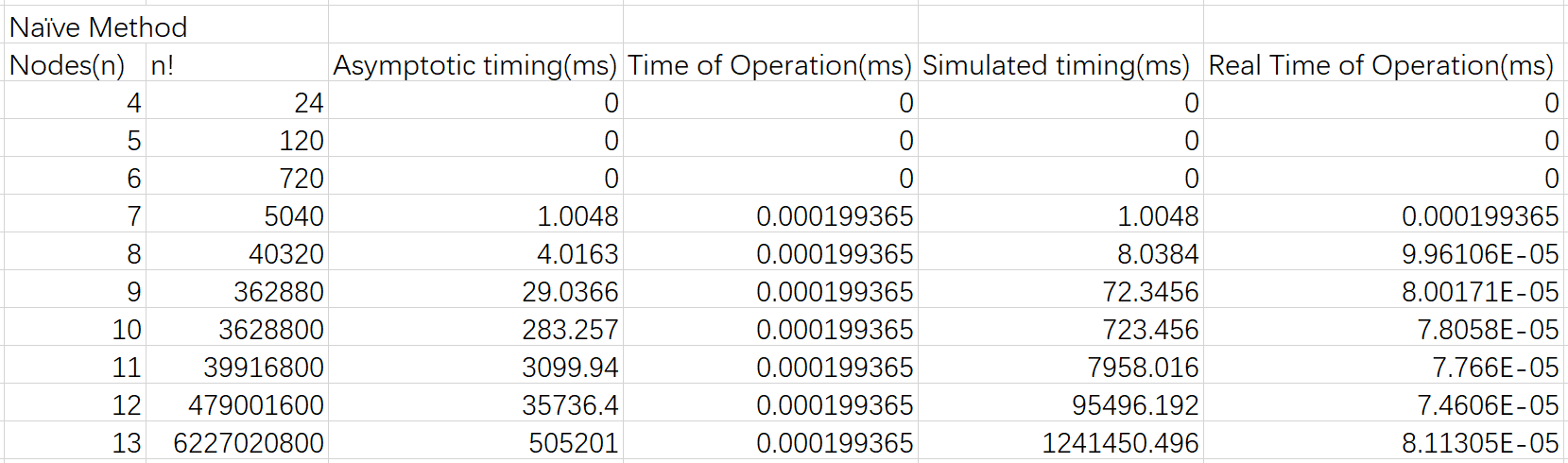


Figure 4: Naive Brute Method Timing

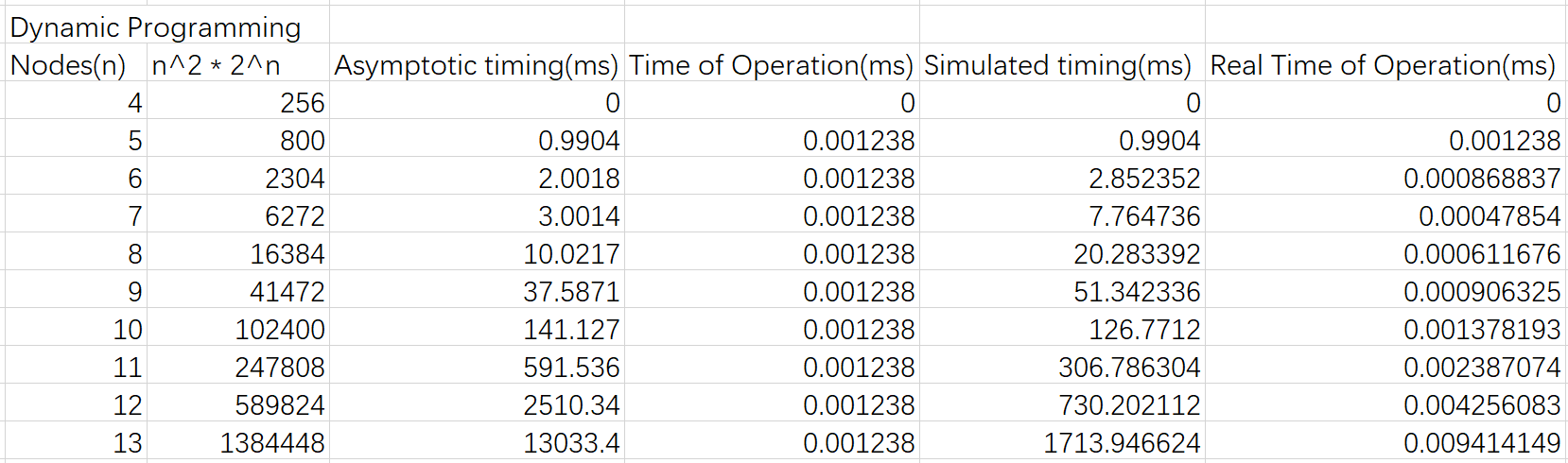


Figure 5: Dynamic Programming Method Timing

When we put the data into graphs, it shows clearer about the change of time for different algorithm.

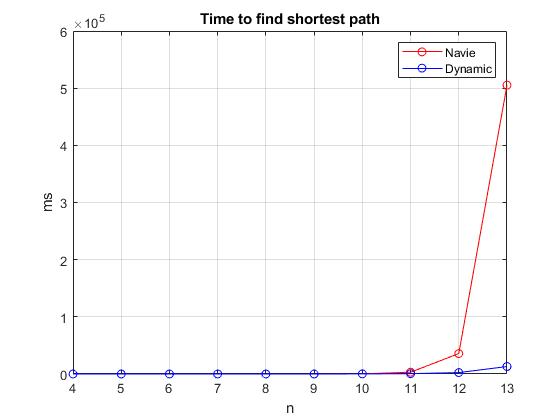


Figure 6: Naive Method and Dynamic Programming Method Timing

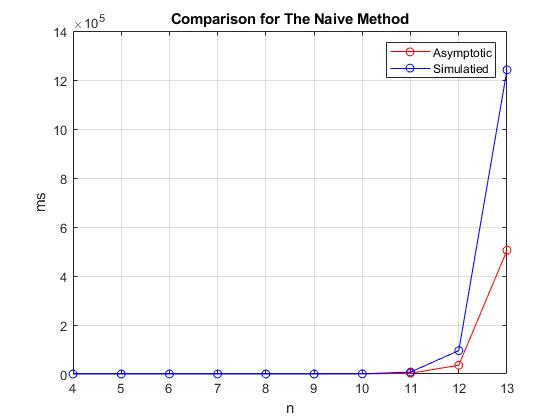


Figure 7: Naive Method Asymptotic time and Simulated time Comparison

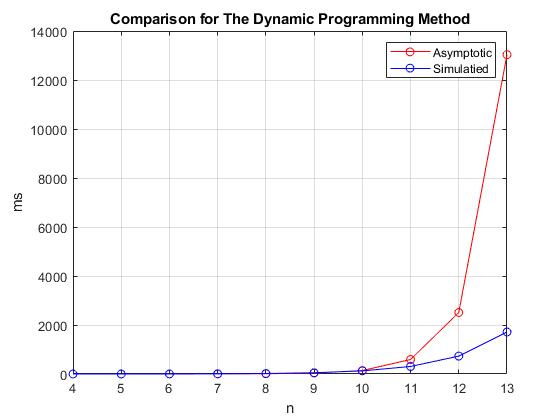


Figure 8: Dynamic Programming Method Asymptotic time and Simulated time Comparison

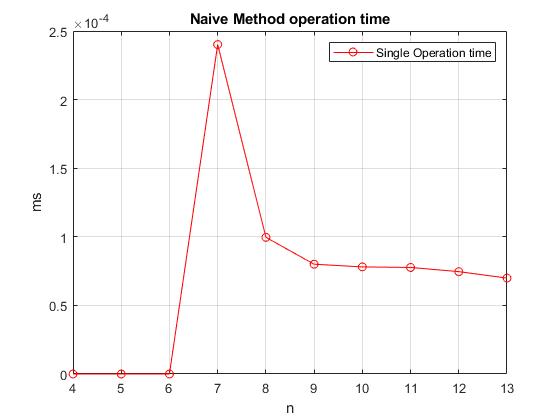


Figure 9: Naive Method operation time

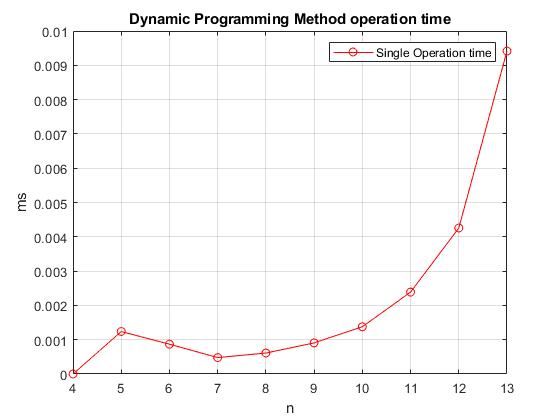


Figure 10: Dynamic Programming Method operation time

To find the Operation time for each algorithm, we use the time we find divide by n! or which n = 4, 5, 6, …

**4: Analysis**

The time complexity for the Naive Brute Method is O(n!) because it runs through all the possible permutations to find the shortest path. On the other hand, the time complexity for the Dynamic Programming is because there are at most O (n \* ) subproblems in the dynamic programming, and each one subproblem will take linear time to solve. The total running time is therefore (<https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/>). When we compute the operation time, the idea time should be constant. However, from the graphs we can find out that the operation time is not constant, so our real running time does not exactly follow the theoretical time complexity. It may cause by a variation when using the high-resolution clock. The output algorithm runtime is not constant, so when we use fewer nodes, the small changes in the operation timing will cause a dramatic difference in the final runtime. Furthermore, in the Dynamic Programming method, the program uses linear time to search, load, compare, and store in each row, so it might also increase the time if there are lots of nodes in one row. Therefore, to improve our program, we can use an more efficient Memoization or use a better search/ sort method to store our data into the table.