Nicholas Crothers

CS 3353

Lab 3 Report

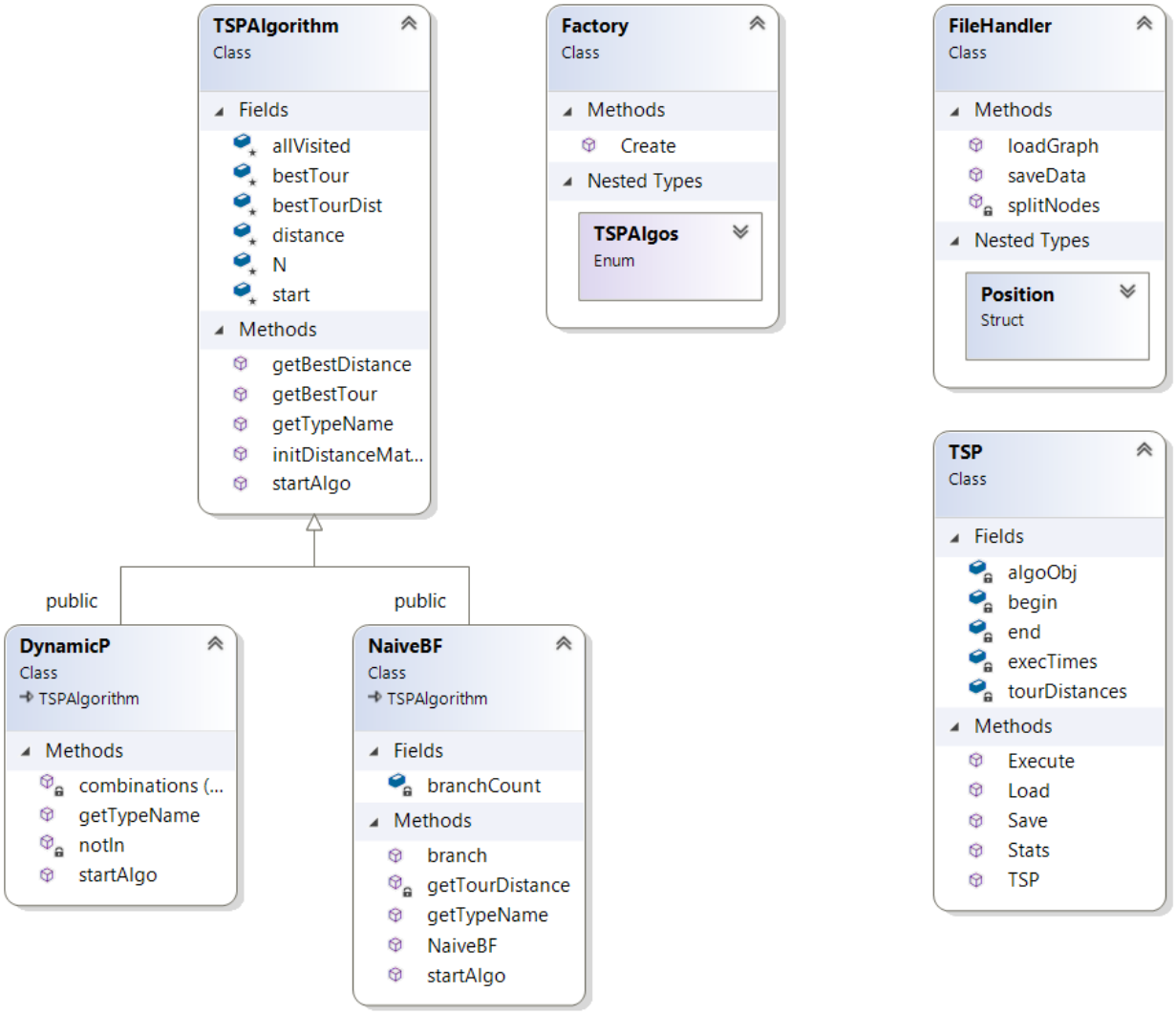
Code Structure

I chose to structure my code using the Factory pattern with inheritance for each specific algorithm, passing in the appropriate enum value for the desired algorithm in `Create()`. The Factory creates a TSP object, which has a TSPAlgorithm pointer to a specific algorithm. The TSPAlgorithm interface allows me to add more algorithms which implement the interface. All you need to do is call `startAlgo()` on the object to begin the algorithm.

The TSP object itself stores the execution data from the algorithm it houses. The class has similar functionality to the Algorithm class from the strategy pattern we used previously. For example, you call `Load()` to load the data file into the TSP object for use by the algorithm object. The TSP object serves as the housing for the algorithm over multiple iterations; it stores a vector of execution times and best tour distances for writing to disk later as well as the time points used for calculating execution times. This way, the TSP algorithm only stores its current execution data, since the algorithm itself doesn't need to know it will be run multiple times.

The FileHandler class handles all reading and writing to files. It doesn't depend on any user-defined classes, so more algorithms can be added or modified and FileHandler doesn't need to be modified unless there are changes in the data being saved or read.

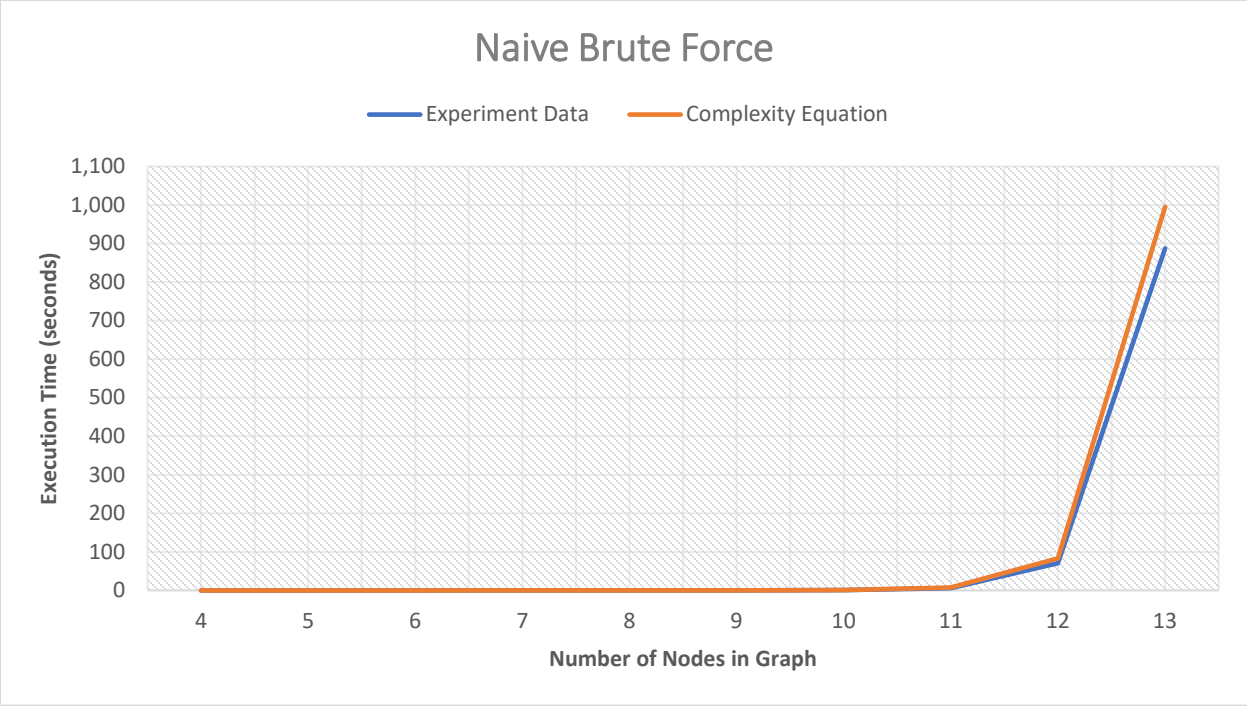
UML Diagram

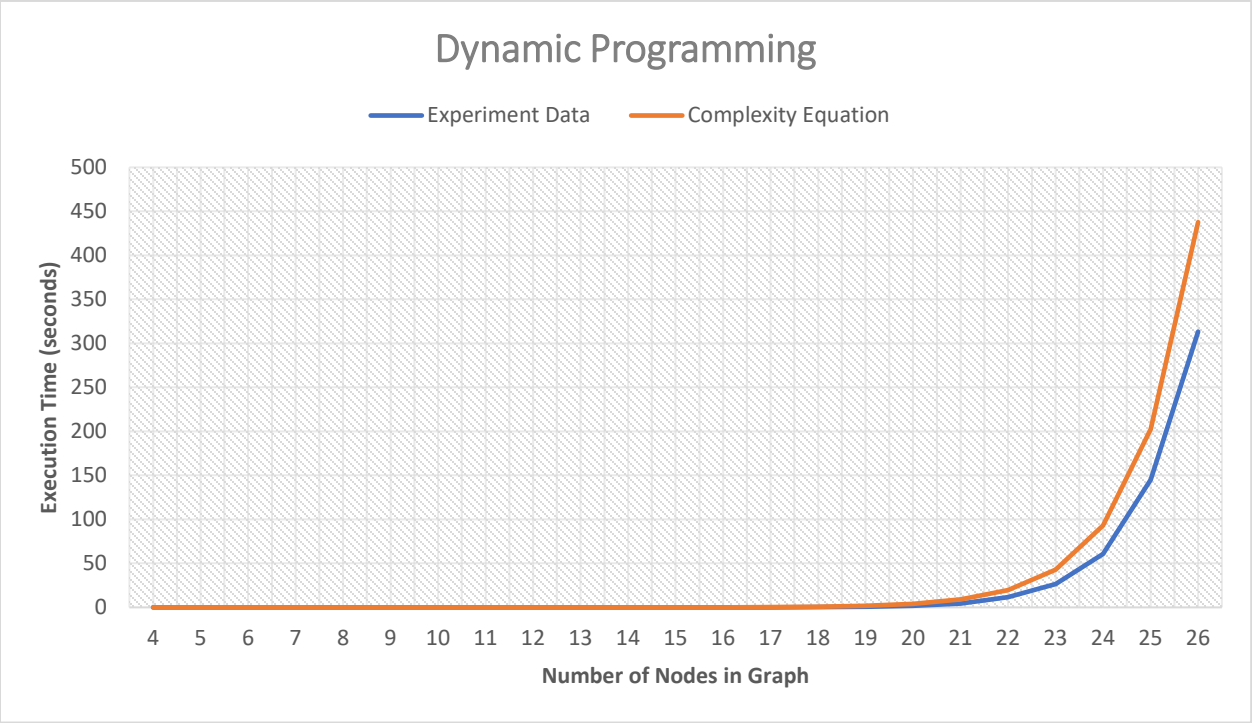

TSPAlgorithm
Class
Fields
allVisited
bestTour
bestTourDist
distance
N
start
Methods
getBestDistance
getBestTour
getTypeName
initDistanceMat...
startAlgo
Factory
Class
Methods
Create
Nested Types
TSPAlgos
Enum
FileHandler
Class
Methods
loadGraph
saveData
splitNodes
Nested Types
Position
Struct
TSP
Class
Fields
algoObj
begin
end
execTimes
tourDistances
Methods
Execute
Load
Save
Stats
TSP
public
public
DynamicP
Class
TSPAlgorithm
Methods
combinations (...
getTypeName
notIn
startAlgo
NaiveBF
Class
TSPAlgorithm
Fields
branchCount
Methods
branch
getTourDistance
getTypeName
NaiveBF
startAlgo

Data

Naïve Brute Force		
Number of Nodes	Execution Time (seconds)	Normalized Complexity Equation Values
4	1.70E-05	1.2E-05
5	5.48E-05	4.82E-05
6	0.000281	0.000241
7	0.001654	0.001445
8	0.011606	0.010113
9	0.072219	0.080907
10	0.598455	0.728163
11	5.99126	7.281633
12	70.8648	80.09796
13	887.157	961.1755

Dynamic Programming - Tabulation		
Number of Nodes	Execution Time (seconds)	Normalized Complexity Equation Values
4	1.46E-05	2.46999E-06
5	1.66E-05	7.71871E-06
6	3.35E-05	2.22299E-05
7	7.33E-05	6.05147E-05
8	0.000154	0.000158079
9	0.000372	0.000400138
10	0.000815	0.000987995
11	0.001856	0.002390947
12	0.004238	0.00569085
13	0.009847	0.013357689
14	0.02062	0.030983515
15	0.043018	0.071135622
16	0.081114	0.16187306
17	0.163674	0.365479018
18	0.371077	0.819482365
19	0.675246	1.826130456
20	1.63143	4.046826495
21	4.46691	8.923252422
22	11.5201	19.58664024
23	26.4312	42.81542432
24	60.7767	93.23888245
25	145.033	202.3413248
26	313.313	437.7047537


Naive Brute Force
Experiment Data
Complexity Equation
Execution Time (seconds)
1,100
1,000
900
800
700
600
500
400
300
200
100
0
4
5
6
7
8
9
10
11
12
13
Number of Nodes in Graph


Dynamic Programming
Experiment Data
Complexity Equation
Execution Time (seconds)
500
450
400
350
300
250
200
150
100
50
0
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
Number of Nodes in Graph

Data Summary

As expected, the dynamic programming algorithm grew much slower than the brute force algorithm, allowing my testing to get to 26 nodes using dynamic programming compared to 13 using brute force without the execution time becoming long. At the rate of growth shown, the execution time for a graph of 14 nodes using brute force would be over three hours. For a graph of 27 nodes using dynamic programming, it would likely take around 10-15 minutes.

When comparing the asymptotic graph to the real data, they are quite close. However, in both cases the asymptotic graph grows faster than the real data. This is likely happening because of the timing data for the small graph sizes. The algorithms' execution time is more influenced by the overhead for initializing the algorithm – like in creating the table, initializing values, et cetera – because there is a small number of calculations needed to solve the problem. This initialization overhead is less influential on overall execution time for larger graphs, so the average value for normalization is skewed. This discrepancy could likely be avoided by using a count of iterations/recursions; however, finding the right way to keep track of this count for growth purposes would be difficult, especially since the brute force algorithm is recursive while the dynamic programming algorithm is iterative.

Dynamic Programming

The dynamic programming algorithm I am using uses tabulation, which is a bottom-up algorithm, to solve the traveling salesman problem. The algorithm constructs a table of size $Nx2^N$, where N is the number of nodes in the graph. Each row from 1 – N represents the current node in a given path, and each column from 1 – 2^N represents the current combination of nodes that have been

visited in the current path. Meaning, if there are 4 nodes, there are 4 bits representing all of the combinations, so 1111 would mean all nodes were visited, 1010 means the first and third nodes have been visited, et cetera.

After constructing this table, the algorithm starts by calculating all possible paths and stores their distances into the table from the smallest paths up to the full tour. It begins at path length 3 continues working up until the full length of the path is reached. In order to calculate subsequent path lengths, it adds the corresponding shorter path's distance with the distance to the new next node. After the table has been populated, the algorithm starts at the column with all nodes visited, choosing the lowest distance node and pushing it to the path. Then it moves to the column without that node in the path, again choosing the lowest distance, pushing it to the path. This continues until the entire path has been pushed.

I am using the algorithm implemented by William Fiset, which I found through a YouTube video. Both the dynamic programming and the brute force algorithm use the same distance matrix format, which is calculated when the graph file is loaded.

Time Complexity

The naïve brute force algorithm solves a graph of n nodes in $O(n!)$. The execution time of the algorithm grows at a rate of $\frac{(n-1)!}{2}$, so this is the equation used for comparison in the graph shown above. The data following this equation was normalized by dividing each value in the experimental data set by $\frac{(n-1)!}{2}$ to get the amount of time spent "per path," then the average is taken of all those values. Each output value in the equation is multiplied by this average, normalizing the data. This line in the graph is labelled as "Complexity Equation."

The dynamic programming algorithm solves a graph of n nodes in $O(n^2 2^n)$. The execution time of the algorithm grows at the same rate, $n^2 2^n$, so this is the equation used for comparison in the graph shown above. The normalizing process is the same as described for the naïve brute force algorithm.

This algorithm is $O(n^2 2^n)$ because there are n possible starting nodes with 2^n possible subpaths for each node. For each subpath, a maximum of n sets of operations (one for each node) is performed, so the complexity comes to $O(n^2 2^n)$.