Analysis of Algorithmic Solutions to the Traveling Salesman Problem

Robin Li

Analysis of Algorithms

Professor Roxanne Canosa

CSCI-261-01

May 13, 2015

Table of Contents

1.0. Overview

The purpose of this project is to implement and compare the optimal and several approximate solutions to the Traveling Salesman Problem. The accuracy of the approximate solutions to the optimal solutions on small problem sizes will be compared to approximate solutions to large problem sizes. The second goal is to discover how accurate the theoretical estimates of complexity are when compared to real execution times. The Traveling Salesman Problem is: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city? In this project, four different path algorithms are implemented to try to solve the problem.

1) Optimal solution: generate an optimal tour of the Traveling Salesman Problem by considering all possible permutations of cities and compute the cost of each permutation to find the tour of least cost.
2) Greedy solution: generate an approximately optimal tour by continually choosing an edge that has the lowest cost of those edges that remain, provided that a cycle is not formed (unless it is the last edge chosen) and the chosen edge is not the third edge from any city.
3) Minimum spanning tree solution: generate an approximately optimal tour by constructing the minimum spanning tree of the graph and computing the pre-order number of each vertex in the tree during a depth-first search.
4) Dynamic programming solution: generate an approximately optimal TSP by finding the optimal bitonic tour using a dynamic programming approach.

The four algorithms were implemented in Java. Each implementation was developed on:

Lenovo Yoga 2 Pro Specifications: Windows 8.1

Processor: Intel® Core™ i5-4200U CPU 1.60 GHz 2.30 GHz

RAM: 4.00 GB

System Type: 64-bit Operating System, x64-based processor

The algorithms were also tested on the CS Ubuntu Machines under local host name: Hendrix.cs.rit.edu.

Specifications on the CS Ubuntu Machines:

Processor: Intel Core i5-2400 CPU 3.10 GHZ

Memory 15.6 GiB

Disk 476.0 GB

2.0. Experimental Design and Input

The OptimalTSP.java file holds the main method for running the optimal solution algorithm and uses the Graph.java and Permutation.java files.

The GreedyTSP.java file holds the main method for running the greedy solution algorithm and uses the DisjointSet.java, Edge.java, Graph.java and QuickSort.java files.

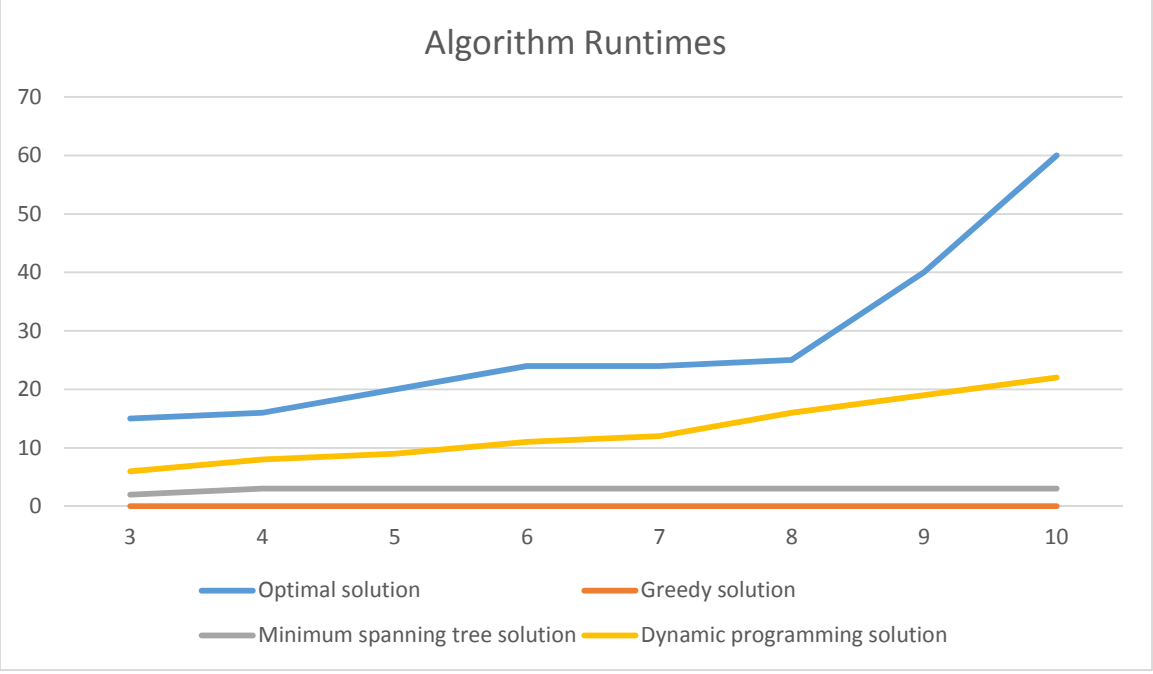
The MstTSP.java file holds the main method for running the minimum spanning tree solution and uses the BinaryHeap.java, Edge.java, Graph.java, and Key.java files.

The BitonicTSP.java file holds the main method for running the dynamic programming solution and uses the Edge.java, Graph.java, Key.java, and Vertex.java files.

The OptimalTSP, GreedyTSP, MstTSP, and BitonicTSP were each tested with values N = 3 to 10 each using a seed value of 100,000. The next set of tests consisted of using N = 100, 500, 1000, and 3000 each using a seed value of 100,000.

3.0. Results

Graph 1: Runtimes of all four implementation for values of N = 3, 4, 5, 6, 7, 8, 9, 10



Graph 2: Runtimes of all four implementation for values of N = 100, 500, 1000, 3000

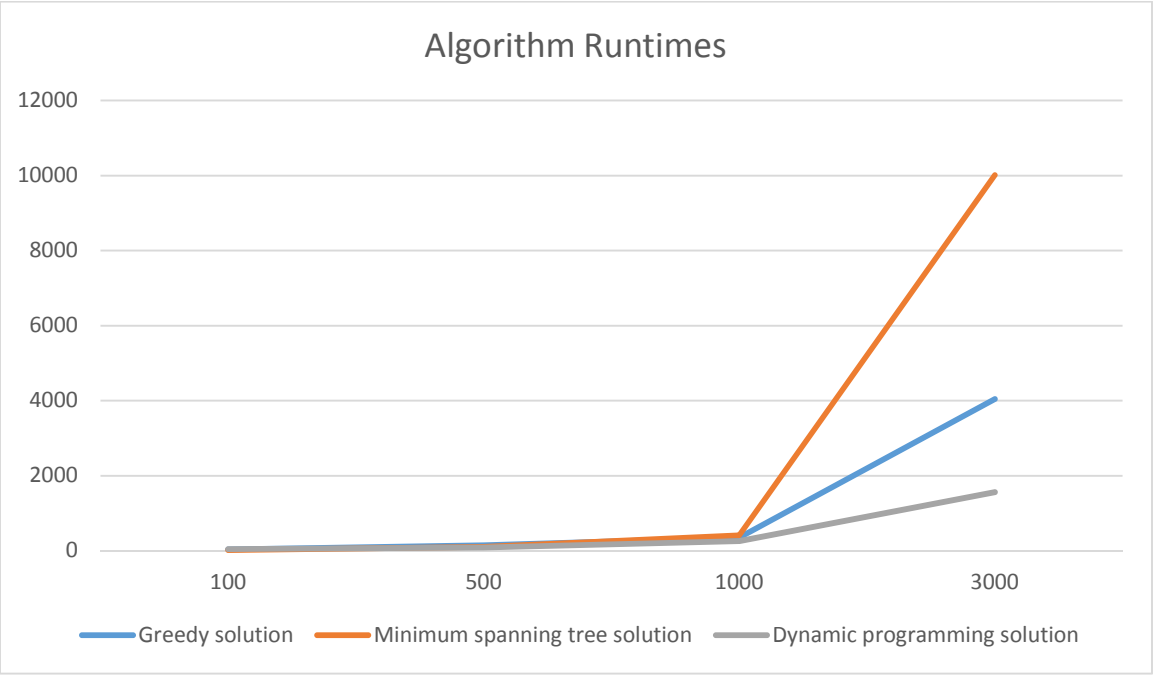


Table 1: Distances for all four implementations for values of N = 3, 4, 5, 6, 7, 8, 9, 10.

N	Optimal Distance	Greedy Distance	MST Distance	Bitonic Distance
3	4.83	4.83	4.83	4.83
4	7.71	7.81	7.71	7.71
5	10.46	11.83	12.67	10.46
6	15.54	16.94	16.94	15.54
7	18.10	20.70	20.36	18.10
8	24.66	26.92	27.62	24.66
9	24.89	35.47	35.63	24.89
10	29.00	29.00	36.93	29.00

Table 2: Distances for all four implementations for values of N = 100, 500, 1000, 3000

N	Optimal Distance	Greedy Distance	MST Distance	Bitonic Distance
100	N/A	973.03	1,062.58	1,486.79

500	N/A	9,573.90	11,722.26	37,613.90
1000	N/A	26,625.79	32,239.68	148,891.13
3000	N/A	137,135.54	167,270.40	1,345,507.54

Table 3: Average Runtimes of Each Algorithm for N = 3, 4, 5, 6, 7, 8, 9, 10

N	Optimal solution	Greedy solution	Minimum spanning tree solution	Dynamic programming solution
3	15	0	2	6
4	16	0	3	8
5	20	0	3	9
6	24	0	3	11
7	24	0	3	12
8	25	0	3	16
9	40	0	3	19
10	60	0	3	22

Table 4: Average Runtimes of Each Algorithm for N = 100, 500, 1000, 3000

N	Optimal solution	Greedy solution	Minimum spanning tree solution	Dynamic programming solution
100	N/A	30	21	45
500	N/A	146	106	83
1000	N/A	352	413	259
3000	N/A	4,047	10,010	1559

4.0. Analysis

Table 5: Theoretical Runtime for each Algorithm

N	OptimalTSP	GreedyTSP $0(N^2)$	MstTSP O(MlogN)	BitonicTSP $O(N^2)$
3	N/A	9	1	9
4	N/A	16	1	16
5	N/A	25	1	25
6	N/A	36	1	36
7	N/A	49	1	49
8	N/A	64	1	64
9	N/A	81	1	81
10	N/A	100	1	100
100	N/A	10,000	2	10,000

500	N/A	250,000	3	250,000
1000	N/A	1,000,000	3	1,000,000

Table 6: Theoretical Runtimes for each implementation along with hidden constants for each.

N	OptimalTSP	Constant	GreedyTSP	Constant	MstTSP	Constant	BitonicTSP	Constant
3	N/A	N/A	9	0	1	2	9	0.
4	N/A	N/A	16	0	1	3	16	0.5
5	N/A	N/A	25	0	1	3	25	0.36
6	N/A	N/A	36	0	1	3	36	0.30
7	N/A	N/A	49	0	1	3	49	0.245
8	N/A	N/A	64	0	1	3	64	0.25
9	N/A	N/A	81	0	1	3	81	0.235
10	N/A	N/A	100	0	1	3	100	0.22
100	N/A	N/A	10,000	0.003	2	10.5	10,000	0.0045
500	N/A	N/A	250,000	$5.84*10^{-4}$	3	35.	250,000	$3.32*10^{-4}$
1000	N/A	N/A	1,000,000	$3.52*10^{-4}$	3	137.	1,000,000	$2.59*10^{-4}$

Profiling:

Since the algorithms were implemented in Java, the hprof Java profiling tool was used to profile each TSP solution algorithm. It showed how much time the program spent in each method of each algorithm and highlighted areas for improvement.

For the optimal TSP, we run the hprof only with N = 10 because N can't go higher than 13 in this project. The hprof statistics showed and even distributed of workload for the methods. 28.57% in the main method in OptimalTSP and 14.29% for each different section. The bottleneck appears in the Permutation class with generating the next permutation and calling the swap function.

For greedy TSP, for hprof run with N = 10, there was an even distribution of 20% workload throughout the algorithm. No significant unnecessary workload found here. For hprof run with N = 3000, there seems to be a significant 38.75% workload done in the graph initialization. There are many trace calls for quicksort partitions which is the other time accumulation cases.

For Mst TSP, for hprof run with N = 10, there seemed to be an even distribution of 20% workload throughout the algorithm shown in the CPU=samples. For CPU=time, the java.lang.Object.wait with ReferenceQueue.remove were the bottlenecks taking up 25.84%.

For N = 3000 using CPU=samples, the bottlenecks were initializing the graph which took up 45.16% and the rest were small percentages of other data structures. Using the CPU=time, there was a lot of time taking up using the Arraylist functions that all add up to 64%, which is significant.

Overall, after seeing all the statistics, some more work can be done in analyzing the N = 3000 case versus the N = 10 case to find hidden insignificant work done by the algorithm to improve the time complexity of the algorithm such as finding unused initialized data structures, looking at the bottlenecks to remove inefficiencies, and comparing the theoretical time against the real time found in the hprof report.

5.0. Conclusion and Recommendations

There are many different trade-offs between the algorithm in terms of optimal distance and optimal runtime which affects our decision in which algorithm to use.

If we were to find the best distance and not be concerned with runtime it would be the OptimalTSP, but for N = 13 or less.

For TSP solving for more than N = 13, where runtime is not a concern, would be to use the GreedyTSP. GreedyTSP would be recommended if N is random and to find the optimal distance without a concern for runtime.

When runtime is a priority, MstTSP and BitonicTSP should be theoretically faster than GreedyTSP and OptimalTSP. This can be demonstrated by testing for values of N greater than 500. The BitonicTSP demonstrates a faster runtime than GreedyTSP and OptimalTSP however the results received for MstTSP do not appear to be accurate because MstTSP has a consistently larger runtime than GreedyTSP. This means there are some inefficiencies within the MstTSP part. The GreedyTSP runtime should be very slow compared to MstTSP and BitonicTSP after evaluating for N greater than 500.

When runtime is the highest priority, the BitonicTSP should be the algorithm to pick because it searches for many different TSP solutions and can find a good path.