Analysis of Traveling Salesman Problem Solution Algorithms

Robb Dooling
Rochester Institute of Technology

Analysis of Algorithms (CSCI-261)
Professor Roxanne Canosa
Section: 03

December 11, 2013

Table of Contents

I. Overview

A. Problem and Algorithms

The classic Traveling Salesman Problem (TSP) attempts to find the shortest path that visits every node, or possible destination, in a graph exactly once, then returns home to the source. For example, if an ice cream truck wants to visit the Rochester suburbs of Chili, Greece, Henrietta, Irondequoit, and Pittsford from its home in Brockport, what is the least cost path that starts in Brockport, passes through each of five suburbs exactly once, and then ends in Brockport? This report explores four algorithms for solving TSP: an optimal solution, a greedy solution, a minimum spanning tree solution, and a dynamic programming solution searching for a bitonic tour.

The optimal algorithm generates every single path possible (visiting each node only once) and then chooses the path that requires the least distance. As seen later, this approach may always return the best possible path, but it becomes extremely impractical and even impossible as the number of nodes in consideration becomes greater and greater. The term “approximation algorithm” therefore describes the remaining three algorithms: they may not always return the absolute best possible path, but they strive to come as close as possible.

Greedy approaches to TSP lie in continually choosing the next nearest node until the tour is complete. Of course, we never choose a node that has already been visited. Greedy path-finding lies in selecting “edges,” or individual paths between

pairs of nodes. We also improve the greedy algorithm by not choosing any edge that is the third edge from any node that can be visited at another point in the path.

Approximation algorithms also include the minimum spanning tree (MST) solution: construct a tree data structure connecting together all nodes using the least number of edges possible. In many instances of TSP, creating a minimum spanning tree - of the original graph of nodes - significantly reduces the number of edges that we need to consider. Afterwards, we may construct a path using this MST.

Finally, one may also attempt to construct a bitonic tour via dynamic programming. A bitonic tour starts at the leftmost node, travels left-to-right in the graph of nodes, and then travels right-to-left to return to the leftmost node once more. We save a set of nodes for the right-to-left return trip by exploring all nodes left-to-right and assigning nodes to either the left-to-right traversal of the first part of the nodes, or to the right-to-left traversal of the second part of nodes.

B. Execution and Testing

All four algorithms were implemented and tested in Java throughout the fall semester of 2013 with results detailed below. Each implementation was created and tested on a Mid-2009 Macbook OS X with the following specifications:

- Processor: 2.8 GHz Intel Core 2 Duo
- Memory: 4 GB 1067 MHz DDR3
- Software: OS X 10.8.5 (12F45)

Each implementation was also tested to ensure compatibility on a Sun Workstation (Siren) under Rochester Institute of Technology's Computer Science department. The implementations are imperfect in handling certain cases of the Greedy TSP and Bitonic TSP, but this report still allows reasonable comparisons of each algorithm.

II. Experimental Design and Input

The four main Java programs were named OptimalTSP.java, GreedyTSP.java, MstTSP.java, and BitonicTSP.java. One main method in each file (named above) received user input in the below format:

% java [solution method]TSP [number of nodes] [random seed value]

For example, to execute the MST solution to TSP with 5 randomly-generated nodes and a seed value of 100000 to generate the x-y coordinates for each node, one would type on the command line:

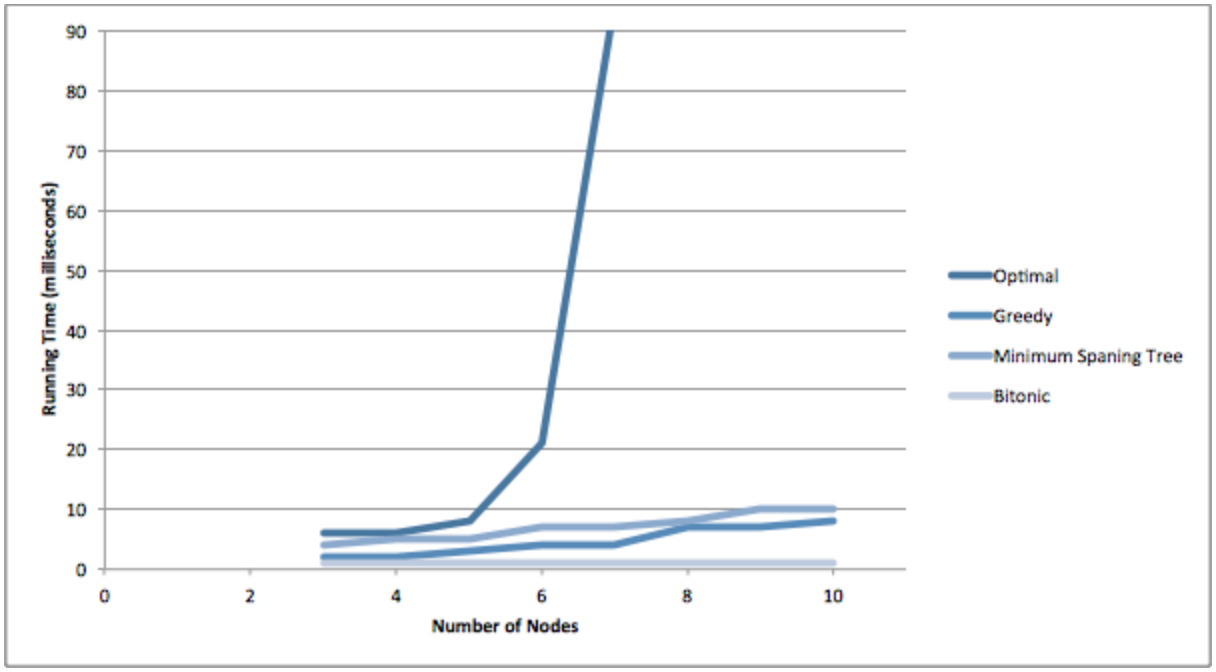
% java MstTSP 5 100000

OptimalTSP, GreedyTSP, MstTSP, and BitonicTSP were each tested with each value for the number of nodes from 3 to 13 and a seed value of 100,000. The approximation algorithms of GreedyTSP, MstTSP, and BitonicTSP were also tested with 100, 500, and 1,000 nodes. It was not possible to test OptimalTSP with more than 13 nodes because as explained earlier, the path generation consumes too much time and memory.

Future implementations could support pre-created graphs: instead of Java's random number generator creating graphs, the program could accept a list of nodes and their coordinates as input then run the algorithm on this graph.

III. Results

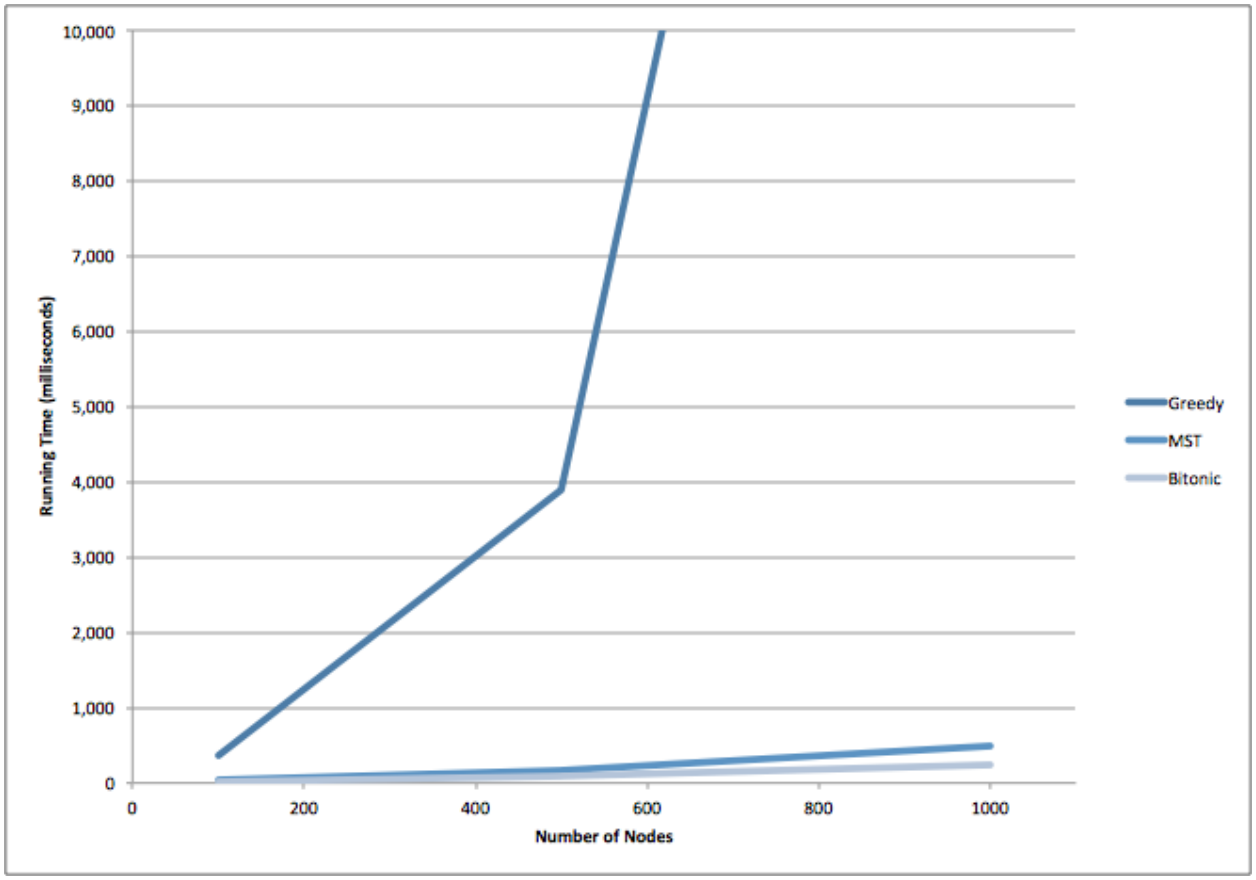
Figure 1: TSP Algorithm Runtime with Few Nodes



The inefficiency of OptimalTSP is very apparent here: it literally goes off the charts, peaking at 2,443 milliseconds for 10 nodes. Fortuantely, GreedyTSP, MstTSP, and BitonicTSP fare much better, and BitonicTSP even emerges as a

perfect champion in this race, with a 1-millisecond runtime for every node value from 3 to 10.

Figure 2: TSP Approximation Algorithm Runtime with Many Nodes



After testing node values of 100, 500, and 1000, we are able to investigate the differences between the three approximation algorithms in more depth. GreedyTSP becomes the next clear choice of elimination. MstTSP performs with efficiency close to that of Bitonic, but falls behind with a much more pronounced difference in the last test: with 1,000 nodes, MstTSP takes 491 milliseconds while BitonicTSP takes only 241 milliseconds.

Table 1: Small TSP Distances Found by Each Algorithm

Number of Nodes	Optimal Distance	Greedy Distance	MST Distance	Bitonic Distance
3	4.83	4.83	4.83	4.83

4	7.71	7.71	7.71	7.81
5	10.46	10.46	12.67	10.46
6	15.54	15.54	16.94	16.07
7	18.1	20.36	18.10	31.1
8	24.66	26.43	27.62	32.53
9	24.89	33.59	35.63	26.78
10	29	35.78	36.93	31.13

Table 2: Large TSP Distances Found by Each Approximation Algorithm

Number of Nodes	Greedy Distance	MST Distance	Bitonic Distance
100	1,043.26	1,062.58	2,062.35
500	10,446.74	11,725.75	63,309.78
1000	28,747.68	32,217.40	254,510.71

Table 3: Average Runtimes of Each Algorithm

Number of Nodes	OptimalTSP	GreedyTSP	MstTSP	BitonicTSP
3	6	2	4	1
4	6	2	5	1
5	8	3	5	1
6	21	4	7	1
7	94	4	7	1
8	429	7	8	1
9	958	7	10	1
10	2443	8	10	1

100	N/A	363	31	8
500	N/A	3894	175	88
1000	N/A	29819	491	241

Table 4: Theoretical Runtime of Each Algorithm

Number of Nodes	OptimalTSP	GreedyTSP O(n^2)	MstTSP O(logn)	BitonicTSP O(n(log^2)n)
3	NP	9	1	3
4	NP	16	1	5
5	NP	25	1	8
6	NP	36	1	11
7	NP	49	1	15
8	NP	64	1	19
9	NP	81	1	24
10	NP	100	1	30
100	N/A	10,000	2	3010
500	N/A	250,000	3	75257
1000	N/A	1,000,000	3	301030

IV. Analysis

The Java profiling tool hprof helped perform software profiling on each TSP solution algorithm. The statistics hprof generated included many CPU samples to show approximately how much time the program spent in each portion of each algorithm and highlighted potential areas for improvement.

For example, an analysis of BitonicTSP with 1000 nodes showed that the program spent 27.27% of its time merely generating the next node and path length tables later used to find the bitonic path. Interesting, this concern does not even exist in the analysis of BitonicTSP with 10 nodes, which shows zero CPU samples in the table generation. For higher node values, however, BitonicTSP's runtime might

be improved with a more intelligent approach to table generation: perhaps we could determine portions of the table that would absolutely not be used later in the program, and avoid filling out these areas of the table.

V. Conclusion

The trade-off between best distance found and program runtime affects our decision regarding which algorithm to use at many different levels. From the graphs and tables above, we can see that if one wishes to find the best distance and is not at all concerned about runtime, OptimalTSP is ideal whenever the problem involves 13 nodes or less. Unfortunately, the testing machines for this project could not handle the current OptimalTSP with more than 14 nodes, but perhaps other better machines could explore this opportunity further.

For TSP solving where runtime is not a concern and there are more than 14 nodes, Table 2 shows that GreedyTSP finds an only slightly better distance for 100 nodes, but finds a somewhat better distance for 500 nodes and a significantly better distance for 1,000 nodes. GreedyTSP is thus the preferred algorithm if one is unsure of node count, but absolutely wants to find the best distance without regard to runtime.

Where runtime is a concern, however, MstTSP and BitonicTSP are consistently superior to GreedyTSP and especially OptimalTSP. This difference is not very pronounced with problems involving 13 or less nodes. With higher node counts, keep in mind that OptimalTSP is completely out of the picture, leaving us with the three approximation algorithms. The runtime of GreedyTSP is only somewhat worse than those of its cousins while evaluating 100 nodes, but introducing 500 nodes shows MstTSP and BitonicTSP as vastly better in this range.

If an user places much higher priority on runtime, BitonicTSP is the algorithm of choice. For example, he or she might need to find hundreds of different TSP solutions and only need a slightly good path, making the distance differences in Table 2 more acceptable.