Traveling Salesman report

Robbie Merillat

29 January 2017

1 Implementations

In this project, two different algorithmic approaches to traveling salesman problem were analyzed; the nearest neighbor approach, and the exhaustive brute force method. Both methods were implemented using Python 2.7 and read in data points from an external file called test.txt. The data is used to create an array of Point objects that will be later used in the TSP calculations.

The Nearest Neighbor approach while incorrect is a decent estimate of the total cost of travel traversing the given points. Making use of my Point class' distance function, the distance between the starting point and each of the other points in the array are compared to find the next point nearest to the current one. The current point is updated to the next nearest, the distance between them is added to a total, and the updated location is removed from the initial list of points. This continues until the initial list of points is exhausted, at which point the path and running total at the end will be the nearest neighbor solution to the TSP.

The Exhaustive approach is a more correct, however much more inefficient method of solving the TSP. For this method, I created an external function called cost which calculated the cost of a single permutation of the list of points. The itertools framework was used to iterate through all permutations of the list of points following the starting point. The total cost of a tour is returned from the cost function and is plugged into a simple minimum search algorithm to find the tour with the shortest cost. When a new minimum is found, the path of that tour is also saved and when all permutations have been traversed, the final path and cost are printed to the console.

2 Worst-Case Complexity

(For the purposes of examining just the two approaches, excess code will be ignored.)

In the worst case scenario, the nearest neighbor algorithm that I implemented runs at a time complexity of appx:

$$(n-1)(n-1) \rightarrow O(n^2)$$

In the worst case scenario, the exhaustive search algorithm has a time complexity of appx:

$$(n-1)!(n-1)(n-1) \rightarrow O(n^2 n!)$$

3 Input generation/experimental testing

A separate script named generate.py was used to generate n random x, y coordinates with a range between 0 and 100. Using randomized for both the nearest neighbor approach and the exhaustive approach, the runtimes of various values of n are displayed in tables 1 and 2 below.

n	$iterations$	$time(seconds)$
10	100	0.000399
100	10000	0.021869
1000	1000000	1.038595
5000	25000000	27.184408

Table 1: Cost of Nearest Neighbor algorithm at various large n

n	$iterations$	$time(seconds)$
5	120	0.001822
7	5040	0.05634
8	40320	0.289989
9	362880	2.248186
10	3628800	24.74709

Table 2: Cost of Exhaustive algorithm per n

The values of n were chosen to demonstrate the complexity of each algorithm. For the nearest neighbor, increments of factors of 10 seemed like it would demonstrate the n^2 complexity well, however the nearest neighbor search took almost 2 minutes for 10,000 points so 5000 was used instead. For the exhaustive search, an n of less than 6-7 isn't very exciting, however points building up to an n of 10+ shows where things start to get much more complex.

4 Match theory and practice

Given the runtimes displayed in Table 1 and Table 2, an analysis of the complexity of these algorithms can be made. The Nearest Neighbor costs show that as n becomes n^2 and again n^3 and so on, the cost also increases by a similar factor of Cost*(n^x). When analyzing the Exhaustive algorithm, it appears as though the cost is increasing by some exponential amount.

Sources

The GitHub repository from which we found our original csv files is linked in a thread from this forum: `giantitp.com/forums/showthread.php?279077-Free-Pathfinder-Offline-Database-Program`)