Solving The Travelling Salesman Problem
A Report Analysing the Algorithms Used

Beej Persson*

Edinburgh Napier University

Algorithms and Data Structures (SET09117)

Abstract

The Travelling Salesman Problem is a common algorithmic problem often discussed in computer science. The problem itself is to find the shortest route of travel between all the desired cities without going to a city twice. This report looks to analyse two algorithmic solutions to this problem and compare them against each other. The first solution is the Nearest Neighbour algorithm whilst the second is a modified and enhanced version of that algorithm. Multiple tests were run to evaluate their efficacy, including measuring run time, comparing route lengths and increasing the number of cities.

Keywords: algorithms, data structures, travelling salesman, efficacy, efficiency, problem solving.

1   Introduction

The Problem   Given a list of cities and their locations (and therefore the distances between each city), what is the shortest route from a starting city that travels to each city only once before returning to the starting city? Solving this problem is often not only about achieving the shortest possible route, there is a focus on optimisation: finding this shortest route in a reasonable time given the number of cities.

First Solution: Nearest Neighbour Basic   The Nearest Neighbour algorithm used here solves the problem by starting with the first city in the file, checking the distances between it and all the other cities, picking the shortest distance then doing the same for that closest city with all the rest, whilst removing each previous city. The pseudo-code for this algorithm is shown in Algorithm 1. This algorithm tends to return fairly short routes (although rarely the shortest) whilst also being fairly efficient. When used on a larger number of cities, the run time will still be comparatively low. This is due to it having an order of complexity of n^2, or $O(n^2)$, where the run time will increase in proportion to the problem size squared (for example the theoretical run time on a set of 10 cities will be 4 times bigger than on a set of 5, as $(10/5)^2 = 4$).

Second Solution: Nearest Neighbour Enhanced   The Nearest Neighbour Enhanced algorithm uses a fairly simple adaptation to the previous algorithm. Instead of starting with the first city in the file, it will start with a random city and find the closest city to it and carry on from there. After that it will calculate the length of that route. It will do these two steps a number of times (specifically a tenth of the number of cities in the file) and then look over the list of lengths and pick the lowest route length found. The pseudo-code for this algorithm is shown in Algorithm 2. This algorithm should consistently find lower route lengths at the expense of increased run time. As it is running the previous algorithm $n/10$ times (where n is the number of cities), its order of complexity is $O(n^3)$. This means that as the problem size increases, the run time will increase significantly (for example, the theoretical run time on a set of 10 cities will be 8 times bigger than on a set of 5, as $(10/5)^3 = 8$).

*e-mail:40183743@live.napier.ac.uk

Data: Array List of unsorted cities
Result: Empty Array List (for sorted cities)
Point2D closest = null;
Point2D currentCity = firstCityInList;
while *cities.size >0* **do**
  add currentCity to Result;
  remove currentCity from Data;
  distance = infinity;
  for *all city in Data* **do**
    if *getDistance(currentCity to city) <distance* **then**
      closest = city;
      distance = getDistance(currentCity to city);
    end
  end
  currentCity = closest;
end
return Result;

Algorithm 1: Nearest Neighbour Basic

Data: Array List of unsorted cities
Result: Empty Array List (for sorted cities)
Array List of Array Lists sortedCitiesList;
Array List lengths;
for *int i <cities.size/10, i++* **do**
  Array List tempCitiesList;
  add nearestNeighbourRandomStart(tempCitiesList) to sortedCitiesList;
  add routeLength(sortedCitiesList.get(i)) to legnths;
  distance = infinity;
  if *lengths.get(i) <distance* **then**
    distance = lengths.get(i);
    Result = sortedCitiesList.get(i);
  end
end
return Result;

Algorithm 2: Nearest Neighbour Enhanced

2   Experimental Method

Overview   The algorithms were tested against each other and on a variety of problem instances, with differing numbers of cities in each problem instance. The time it took the algorithm to return a sorted list of cities from the original unsorted list was used as the Run Time. The Route Length results were determined by calculating the length of the route between all the cities in the order of the sorted cities list. Tests were repeated for each algorithm for every problem instance. The results used in the later graphics were the averages of these repeated tests.

Detailed Description

- **Functionality of One Test**: Problem instance files containing a list of a certain number, n, of cities were loaded into an array list in Java. The algorithms sort this array list into an ordered

list of cities, the order being the shortest route it found. The Run Time is calculated by finding the difference between the current time in milliseconds immediately before the algorithm is run and immediately after it returns a sorted list. This sorted list is then passed through a method that calculates the Route Length of the sorted cities in order. The Run Time and Route Length results were recorded for each test.

- **Testing Methodology**: For each of the chosen problem instances ten tests were run on each algorithm. For each test the results for the Run Time and the Route Length were then recorded in a table in an Excel file (see Figure 1). The average Run Time and Route Length for each algorithm were stored in a separate table (Figure 2) against the number of cities, n. This was the testing method used to produce the results and graphs analysed in this report.

Accuracy, Reliability and Reasoning for Testing Methodology
The number of cities in the file is initially known and is part of the file name. The number of cities in the array list (that the cities from the file are loaded into) is checked to match the number of cities in the file, and checked again after the algorithm is run. This is to ensure that the Route Length for the shortest route found is including every city in the problem instance and eliminate any artificially short routes. Nine problem instances were used, with an appropriately varying number of cities in each (ranging from 52 cities, up to 5915 cities) to help visualise the effect of n on the Run Time of the two algorithms. Each problem instance tested stored the location of cities as points in Euclidean 2D geometry (with x, y coordinates) and the way the software loaded the file's city list into an array list relied upon the city locations being stored this way. Therefore only problem instances that stored the cities in this way were tested. The Run Time results were recorded in milliseconds to best represent the time taken, with a problem instance with a small number of cities taking only a few milliseconds, and larger sample sizes taking hundreds and even thousands of milliseconds. Seconds could also have been chosen and been similarly effective due to the wide range of times achieved. The Route Length results were recorded with arbitrary units, as the location of each city was simply stored as a 2D point in the files with no indication of the units of the x and y coordinates. .Each test was run on the same PC, from the same code, with each result being recorded one at a time to reduce memory build up slowing down Run Time and improve the accuracy and repeatability of the results. Ten tests were run for each problem instance and for each algorithm and the average Run Times and Route Lengths were used for displaying the results to further improve reliability of the accuracy of the testing methodology.

3 Results

Shown below are the results of the testing.

Figure 1 shows an example table of the testing. The problem instance file name and the number of cities in that file is at the top and the test number that the results correspond to are on the left. The results for both algorithms were stored in the same table in separate columns and the averages of the results are shown at the bottom of the table. A table similar to this was produced for all nine problem instances and an averages table (see Figure 2) was produced from the averages of the results from each test of the algorithms on the separate problem instances. Shown below is that final averages table.

The table above is the averages table. The problem instances' number of cities, n are on the left with the corresponding averages of the test results for each algorithm showing the neighbouring columns.

| u1060.tsp ->n = 1060				
	Nearest Neighbour Basic		Nearest Neighbour Enhanced	
Test	Time / ms	Route Length / units	Time / ms	Route Length / units
1	13	296,543.889	1,095	282,301.279
2	14	296,543.889	1,104	281,639.607
3	13	296,543.889	1,109	281,581.385
4	14	296,543.889	1,103	281,006.855
5	13	296,543.889	1,103	281,581.385
6	13	296,543.889	1,103	281,006.855
7	13	296,543.889	1,109	281,329.829
8	13	296,543.889	1,093	281,444.456
9	14	296,543.889	1,099	282,083.731
10	13	296,543.889	1,106	281,329.829
Avg	13.3	296,543.889	1,102.4	281,530.521

Figure 1: *The table produced from the results of ten tests run for each algorithm on the file u1060.tsp, where n is the number of cities which equals 1060.*

Averages				
	Nearest Neighbour Basic		Nearest Neighbour Enhanced	
n	Time / ms	Route Length / units	Time / ms	Route Length / units
52	0.6	8,980.92	1.0	8,883.10
101	1.0	825.24	2.5	759.50
159	1.6	54,669.03	5.6	50,251.97
262	5.8	3,241.47	19.9	2,920.33
493	4.9	43,646.37	125.0	41,982.41
1060	13.3	296,543.89	1,102.4	281,530.52
2103	44.2	90,517.89	8,113.1	87,664.57
3795	137.1	35,362.51	45,949.0	34,226.09
5915	336.4	707,498.63	179,763.5	672,798.03

Figure 2: *The Table of Averages produced by averaging all the results from the tests, where n is the number of cities. This is the table used for the graphics later in the report.*
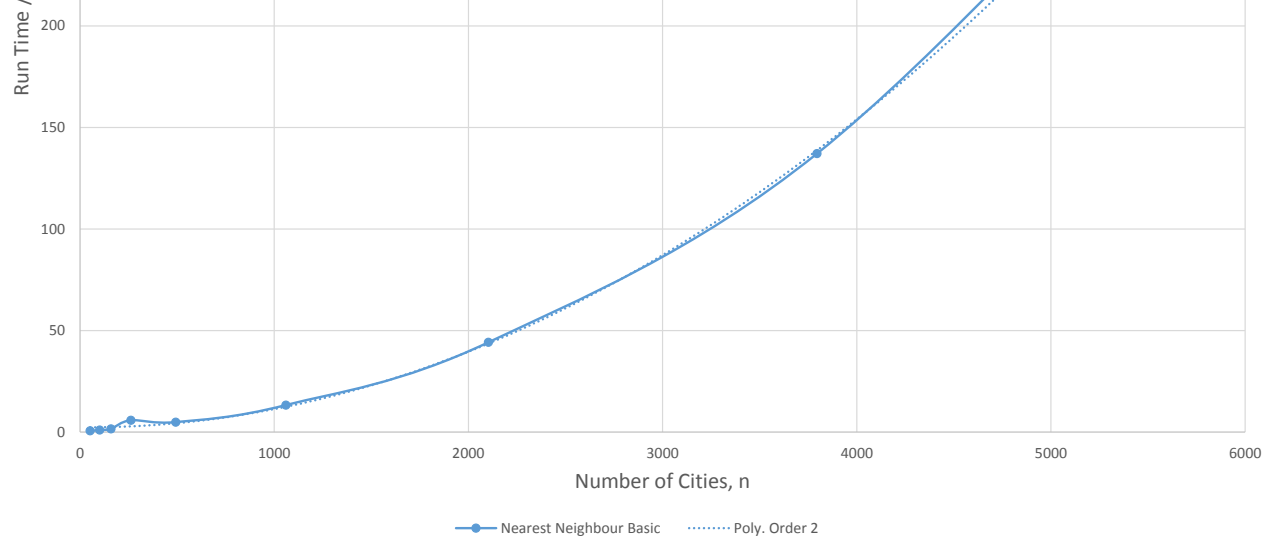
Figure 3: *A graph showing the Nearest Neighbour Basic algorithm's average run time compared against the number of cities in each problem instance. The trendline is a polynomial trendline of order 2.*
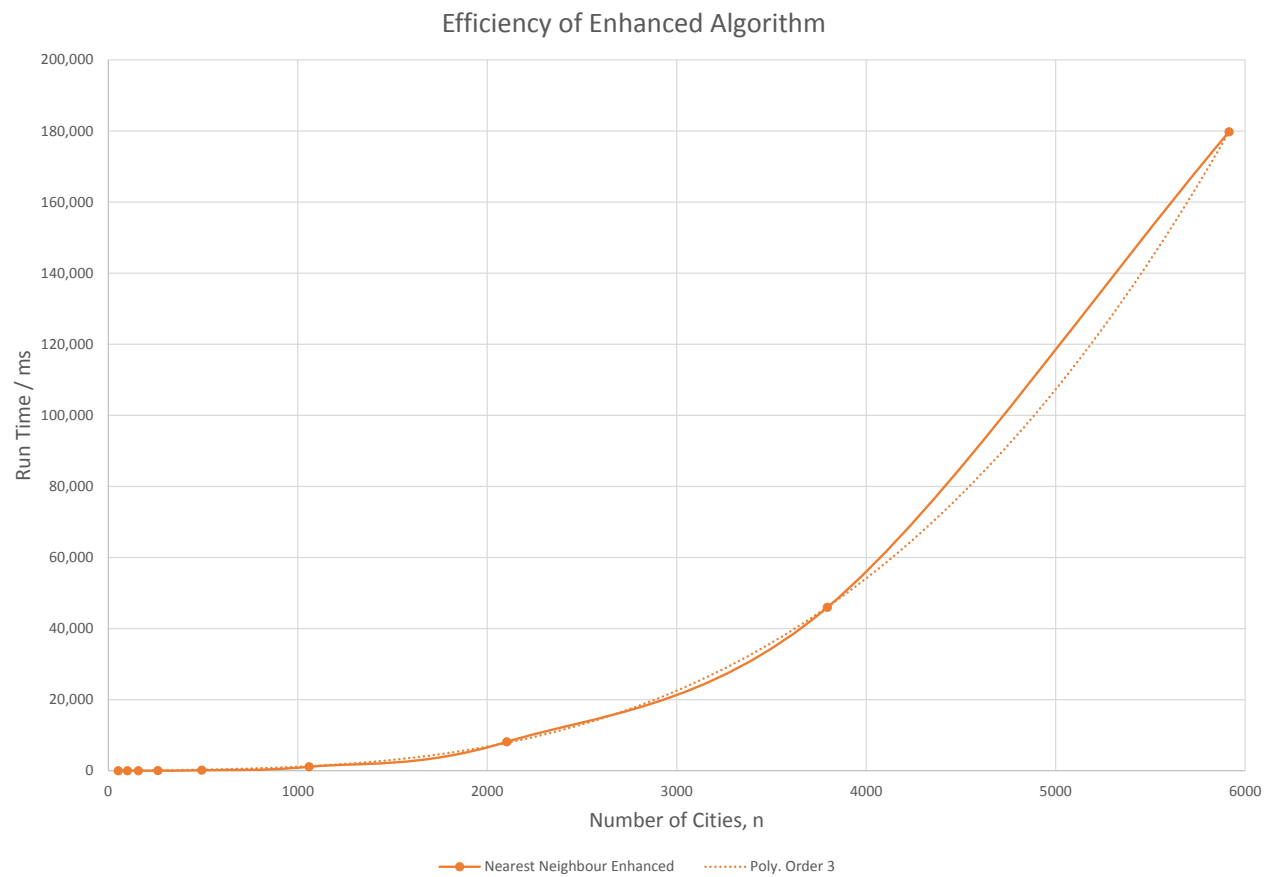
Figure 4: *A graph showing the Nearest Neighbour Basic algorithm's average run time compared against the number of cities in each problem instance. The trendline is a polynomial trendline of order 2.*

4 Guidelines

This section should be removed from your design report. Information provided here is to help you writing up your reports.

- Equations should be numbered and in the correct format, e.g., Equation 1 below:

$$\sum_{j=1}^{z} j = \frac{z(z+1)}{2} \tag{1}$$

- Furthermore, if you include an equation, ensure you explain what each of the variables are (e.g., F is force, m is mass, and a is the acceleration).

- Don't using 'I' or 'Me'.

- Each paragraph should be clear and focused, with multiple sentences that help make your point - avoid lots of single line paragraph sentence.

- Make sure the citations are done using the correct formatting (i.e., .bib file and let LaTeX generate the references).

- Every figure should have a caption, explaining what the picture is and what the reader should be looking at (i.e., what is important about the figure, what does it show)

- A figure should also be referenced in the body of the main text (e.g., see Figure **??**)

- Equations should be numbered, and referenced in the text. Furthermore, ensure each of the variables in the equation are explained (i.e., don't use F=ma and not say what F, a, and m are)

5 Conclusion and Future work

The report should finish with a summary to give a brief overview of what the reader should remember most. What was most important? The future work part only needs to be covered in your final report.

References