Travelling Salesman Problem Report

Conner Weatherston *
Edinburgh Napier University
Algorithms and Data Structures (SET09117)

Abstract

The purpose of this report is to find out the efficiency of the nearest neighbour algorithm in order to solve a variety of travelling salesman problems. This report is also looking how possible variations and optimization techniques such as just searching on an individual axis would impact on the performance of the algorithm. From the results it can be concluded that is quicker to compare on a single axis and to reduce the number of mathematical operations used. Reducing code in the loop also improves the algorithm significantly. Out of the different variations of the nearest neighbour the most practical one to use would be the nearest neighbour squared.

Keywords: travelling salesman problem, nearest neighbour, optimisation, algorithms

1 Introduction

This report is looking at possible improvements of the nearest neighbour algorithm in order to solve the travelling salesman problem (commonly referred to as tsp). The tsp essentially is a question which asks 'Given a set of cities, what is the shortest route possible that visits each city only once and will return to the origin[contributors 2016].

The main issue with the travelling salesman problem is the possible routes available to traverse grows exponential with size. An example of this is a tsp with 10 cities there is 181,400 possible routes. This is calculated using the formula:

$$P = \frac{(N-1)!}{2} \tag{1}$$

where P is number of possibilities and N is number of cities (points).

Only half of the possible routes are counted as each route has an equal reverse route that has the exact same distance. The P-1 is there since the starting city is defined and the other cities can have different permutations.

Size	Possible routes
5	12
10	181400
12	19958400
14	3113510400

As seen from the table above it is clear that the is an exponential growth in the number of possible routes as the number of cities in the list increases.

2 Method

One of the most common ways to get a general solution to the travelling salesman problem is to utilise the nearest neighbour algorithm. The aim of this algorithm is to sort the list so that the next element is the closest one to the current city. The limiting factor of this algorithm is how fast it can do the comparisons between the current city to the other cities to find the closest one.

Pros

- Easy to implement.
- Very quick results for small data sizes.
- Simple to calculate the big O notation.

Cons

- Requires large storage of data.
- Large searching problems (Have to continually iterate over list until it is empty.)
- Assumptions are made about distance (Some routes may be infeasible).
- Brute force method.

Pseudocode

In this section the pseudo code for all of the different variations of the algorithm that are aimed to be implemented are available.

```
Data: ArrayList input
Result: returns Nearest Neighbour list
current city = input first value
while cities in input do
    add current city to result
    distance = max value
    foreach city in input do
        if distance(current city, city) < distance then
            closest city = city
            distance = distance(current point,city)
        end
    end
    remove closest city from input
    current city = closest city
end
```

Algorithm 1: Nearest neighbour algorithm

An improvement to the algorithm can be made by using the distance squared function instead of the distance function. To improve upon this more the result from distance squared is stored so it is not needed to be computed again.

An attempt of rewriting the algorithm was done. The aim of this was to try and minimise the number of iterations in the for loop by comparing the current distance against the next city in the list.

Java has a random function available. One variation was crossing a random starting position in order to provide a better result.

*e-mail:40167111@live.napier.ac.uk

```
Data: ArrayList input
Result: returns Nearest Neighbour squared list
current city = input first value
while cities in input do
    add current city to result
    distance = max value
    foreach city in input do
        current Distance = distanceSquared(current city, city) if
         current Distance < distance then
            closest city = city
            distance = current Distance.
        end
    end
    remove closest city from input
    current city = closest city
end
```

Algorithm 2: Nearest Neighbour squared

```
Data: ArrayList input
Result: returns Nearest Neighbour rewritten list
current city = input first value
while cities in input do
    add current city to result
    distance = max value
    foreach city in input do
        if distance(current city, city) < distance then
            closest city = city
            distance = distance(current point,city)
        end
        if (another city after city) then
            if distance(current city, next city) < distance then
                closest city = next city
                distance = distance(current point, nextcity)
                continue loop from next city
            end
        end
    end
    remove closest city from input
    current city = closest city
end
```

Algorithm 3: Nearest Neighbour Rewritten algorithm

Taking advantage of the collections type it is possible to randomise the entire input arrayList before using the nearest neighbour algorithm.

For the algorithms that use random features (random start and shuffle) it is important to note that 1. They are not reliable methods to calculate a suitable route and secondly given enough time the algorithm could be looped and compared to find out if the new results beat the previous results and if it does replace the results with the new results. However as it is unrealiable it was decided to be implemented to see how much it would decrease or increase performance and was not focusing on providing the best route possible (which could happen on its first iteration or it could never happen).

Another variation of the nearest neighbour is checking for the closest point on a given axis. As the points are in two dimensional space this means checking along the x axis and y axis. The aim of this was to try and completely remove mathematical operations and only use simple boolean operations in order to get a route in quick computation time. However this may result in very long1 routes due to point distributions.

O notation

```
Data: ArrayList input
Result: returns Nearest Neighbour random start list
current city = Random value from input list
while cities in input do
    add current city to result
    distance = max value
    foreach city in input do
        if distance(current city, city) < distance then
            closest city = city
            distance = distance(current point,city)
        end
    end
    remove closest city from input
    current city = closest city
end
```

Algorithm 4: Nearest neighbour random start algorithm

```
Data: ArrayList input
Result: returns Nearest Neighbour shuffle list
input = randomise(input) current city = input first value
while cities in input do
    add current city to result
    distance = max value
    foreach city in input do
        if distance(current city, city) < distance then
            closest city = city
            distance = distance(current point,city)
        end
    end
    remove closest city from input
    current city = closest city
end
```

Algorithm 5: Nearest neighbour shuffle algorithm

An effective way to calculate the complexity of an algorithm is to find out what the O notation of the algorithm is. The O notation being used to calculate the complexity is Big O which is used to specifically used to describe the worst-case scenario of an algorithm. Figure 1 shows the rough guidelines for big O complexity.

To calculate the big O notation of an algorithm a series of rules are followed.

- Input will be called n.
- Measure the number of ”touches” (interactions with n).
- It is the worst case scenario.

The O notation if n^2

It is important to note that complexity has an effect on the performance however performance does not affect the complexity of an algorithm. The reason the specifications of the computer being used is because performance is also affected by hardware.

A limitation of this solution is the file reader. It can only read in certain travelling salesman problems which limits the variety of the results obtained.

3 Results

The algorithms were tested using Eclipse Neon.1 Java development environment. Also the machine specifications are as follows:

- Processor: Intel i7-4790K @ 4.00GHz
- Graphics card: GeForce GTX 980 4GB GDDR5

```
Data: ArrayList input
Result: returns Nearest x Neighbour list
current city = input first value
while cities in input do
    add current city to result
    closest x = max value
    foreach city in input do
        if city.x < current city.x then
            closest city = city
            closest x = city.x
        end
    end
    remove closest city from input
    current city = closest city
end
```

Algorithm 6: Nearest x neighbour algorithm

```
Data: ArrayList input
Result: returns Nearest y Neighbour list
current city = input first value
while cities in input do
    add current city to result
    closest y = max value
    foreach city in input do
        if city.y < current city.y then
            closest city = city
            closest y = city.y
        end
    end
    remove closest city from input
    current city = closest city
end
```

Algorithm 7: Nearest y neighbour algorithm

- Ram: 16.0G

When collecting the results it was important to ensure that they are all treated equally with no bias. Therefore for each data set each algorithm is ran an equal number of times. For this report 5 times was suitable due to the number of data sets and number of variations of the algorithm. The slowest and fastest times were removed and ran again to make the results a little more accurate. This was to take into account of Java garbage collection. Also the algorithms were ran one at a time to ensure there was no heap space competition. Lastly no background applications were running to attempt to minimise any performance issues due to resource competition because of background applications. Another important thing to note is the algorithms were all ran on the same computer to ensure that the hardware specifications remained the same and would not skew the results.

Data set rl5915

rl5915 has a size of 5915 and when unsorted has a total distance of 1014504.71m.

Nearest Neighbour

	Total Distance(m)	Time Taken (ms)
Average	707498.63	190

Nearest Neighbour Random Start

	Total Distance (m)	Time Taken (ms)
Average	707498.63	190

Nearest Neighbour Shuffle

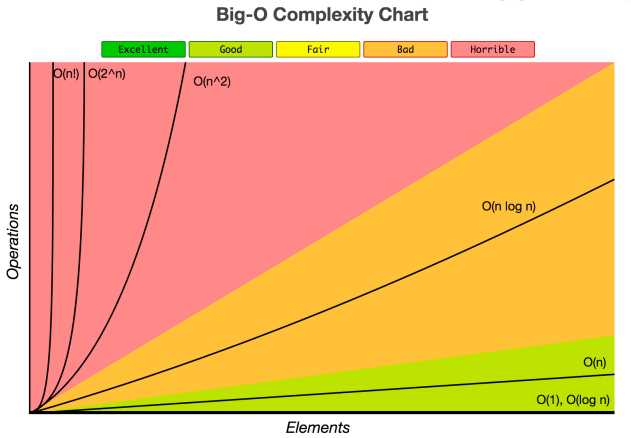


Figure 1: *Approximate graph for big O complexity.*

	Total Distance (m)	Time Taken (ms)
Average	707498.63	190

Nearest Neighbour SQ

	Total Distance (m)	Time Taken (ms)
Average	707498.63	111

Nearest Neighbour Rewrite

	Total Distance (m)	Time Taken (ms)
Average	707498.63	323

Nearest X Neighbour

	Total Distance (m)	Time Taken (ms)
Average	15429176.67	77

Nearest Y Neighbour

	Total Distance (m)	Time Taken (ms)
Average	7787705.43	81

Data set rl5934

rl5934 has a size of 5934 and when unsorted has a total distance of 9861360.32.

Nearest Neighbour

	Total Distance(m)	Time Taken (ms)
Average	683805.99	190

Nearest Neighbour Random Start

	Total Distance (m)	Time Taken (ms)
Average	687058.67	189

Nearest Neighbour Shuffle

	Total Distance (m)	Time Taken (ms)
Average	668711.52	159

Nearest Neighbour SQ

	Total Distance (m)	Time Taken (ms)
Average	683805.99	111

Nearest Neighbour Rewrite

	Total Distance (m)	Time Taken (ms)
Average	683805.99	325

Nearest X Neighbour

	Total Distance (m)	Time Taken (ms)
Average	15122594.14	78

Nearest Y Neighbour

	Total Distance (m)	Time Taken (ms)
Average	9589407.52	80

Data set rl1304

rl1304 has a size of 1304 and when unsorted has a total distance of 3231697.34

Nearest Neighbour

	Total Distance(m)	Time Taken (ms)
Average	339797.47	12

Nearest Neighbour Random Start

	Total Distance (m)	Time Taken (ms)
Average	326226.14	13

Nearest Neighbour Shuffle

	Total Distance (m)	Time Taken (ms)
Average	313703.70	16

Nearest Neighbour SQ

	Total Distance (m)	Time Taken (ms)
Average	339797.47	9

Nearest Neighbour Rewrite

	Total Distance (m)	Time Taken (ms)
Average	339797.47	23

Nearest X Neighbour

	Total Distance (m)	Time Taken (ms)
Average	3718084.59	7

Nearest Y Neighbour

	Total Distance (m)	Time Taken (ms)
Average	2786850.75	10

Data set rl1323

rl1323 has a size of 1323 and when unsorted has a total distance of 3088180.39.

Nearest Neighbour

	Total Distance(m)	Time Taken (ms)
Average	332094.97	12

Nearest Neighbour Random Start

	Total Distance (m)	Time Taken (ms)
Average	326836.54	13

Nearest Neighbour Shuffle

	Total Distance (m)	Time Taken (ms)
Average	342522.89	11

Nearest Neighbour SQ

	Total Distance (m)	Time Taken (ms)
Average	332094.97	11

Nearest Neighbour Rewrite

	Total Distance (m)	Time Taken (ms)
Average	707498.63	23

Nearest X Neighbour

	Total Distance (m)	Time Taken (ms)
Average	3460174.47	7

Nearest Y Neighbour

	Total Distance (m)	Time Taken (ms)
Average	2940280.95	8

Data set rl1889

rl1889 has a size of 1889 and when unsorted has a total distance of 6601297.57.

Nearest Neighbour

	Total Distance(m)	Time Taken (ms)
Average	400684.64	23

Nearest Neighbour Random Start

	Total Distance (m)	Time Taken (ms)
Average	414321.08	24

Nearest Neighbour Shuffle

	Total Distance (m)	Time Taken (ms)
Average	401041.21	21

Nearest Neighbour SQ

	Total Distance (m)	Time Taken (ms)
Average	400684.64	14

Nearest Neighbour Rewrite

	Total Distance (m)	Time Taken (ms)
Average	400684.64	37

Nearest X Neighbour

	Total Distance (m)	Time Taken (ms)
Average	6024698.02	12

Nearest Y Neighbour

	Total Distance (m)	Time Taken (ms)
Average	4907965.140910832	7

Data set Berlin52

Berlin52 has a size of 52 and when unsorted has a total distance of 22205.61m.

Nearest Neighbour

	Total Distance(m)	Time Taken (ms)
Average	8980.92	0.45

Nearest Neighbour Random Start

	Total Distance (m)	Time Taken (ms)
Average	9253.75	0.53

Nearest Neighbour Shuffle

	Total Distance (m)	Time Taken (ms)
Average	9396.88	0.50

Nearest Neighbour SQ

	Total Distance (m)	Time Taken (ms)
Average	8980.92	0.46

Nearest Neighbour Rewrite

	Total Distance (m)	Time Taken (ms)
Average	8980.92	0.59

Nearest X Neighbour

	Total Distance (m)	Time Taken (ms)
Average	17074.13	0.39

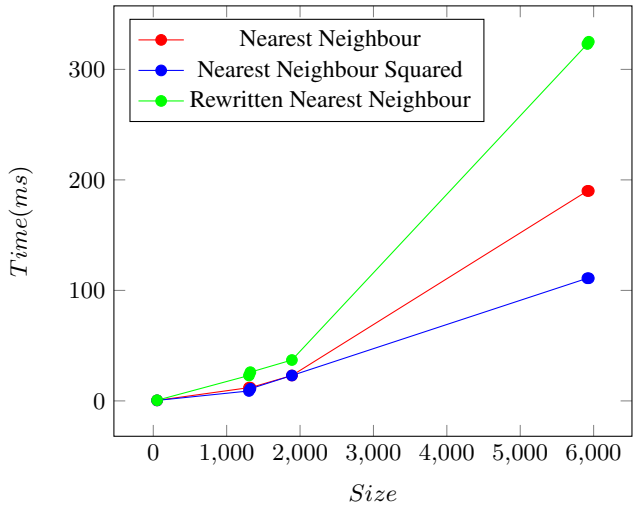
Nearest Y Neighbour

	Total Distance (m)	Time Taken (ms)
Average	24438.27	0.38

Graphs

The following graphs show the relation between the data size and taken to compute the route.

Comparrison between three versions of nearest neighbour



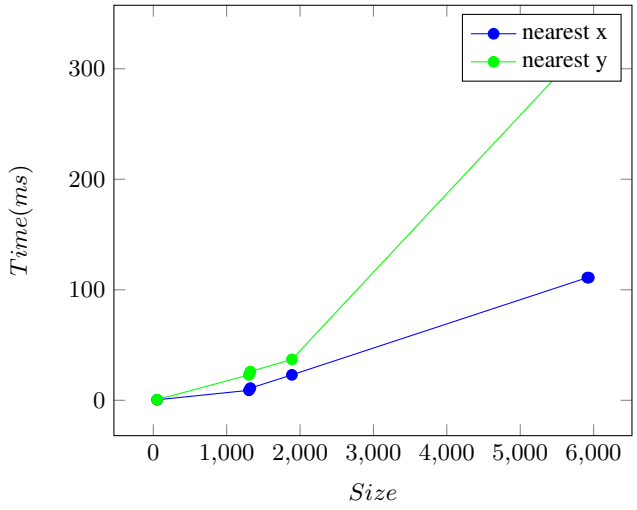
Out of the three basic nearest neighbour algorithms the fastest algorithm is the one that leaves the distances to be compared against in their squared form as well as storing the value from the function as a variable instead of computing it twice. From the graph it is clear that there is a gain in performance from 2000 points onwards. Once there are 5934 points the improved algorithm has a 41.57% improvement on computation time of the algorithm. The reason for this improvement is due to the distance function which does the following calculation:

$$Distance = \sqrt{x^2 + y^2} \tag{2}$$

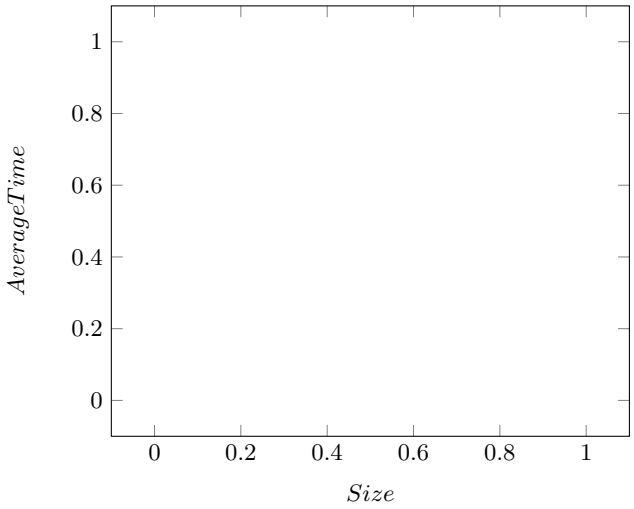
Where x is (point A.x - point B.x) and y is (point A.y - point B.y).

Therefore as the number of points increased the number of times this calculation also increased. As mathematical operations (especially square rooting) it clearly has a hit to performance like it is shown in the graph. Another reason for the nearest neighbour squared having quicker times is because the worst case scenario for the distance calculation has been reduced to one. In the original calculation the worst case scenario was two. This means that the number of times the operation is used is decreased significantly as the number of elements being sorted increases.

Comparrison between nearest x and nearest y

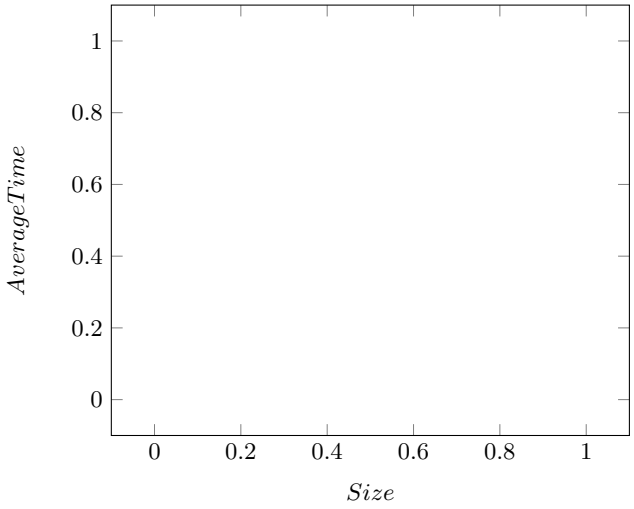


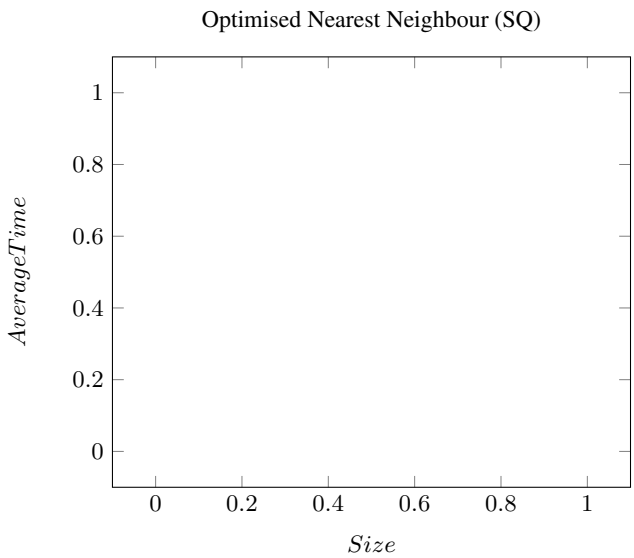
Optimised Nearest Neighbour (SQ)

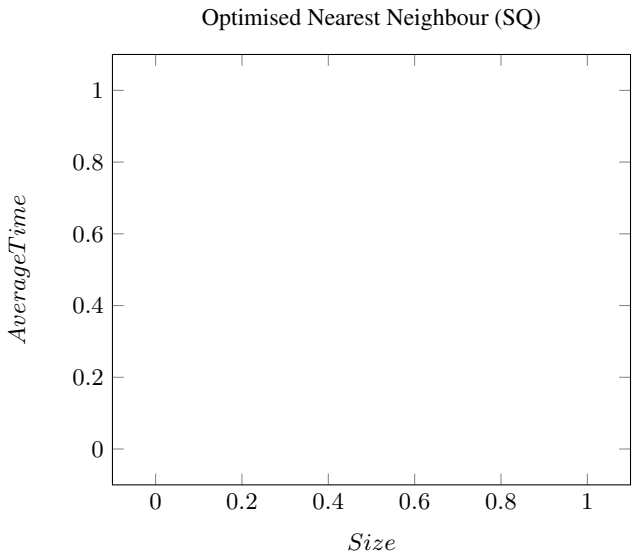


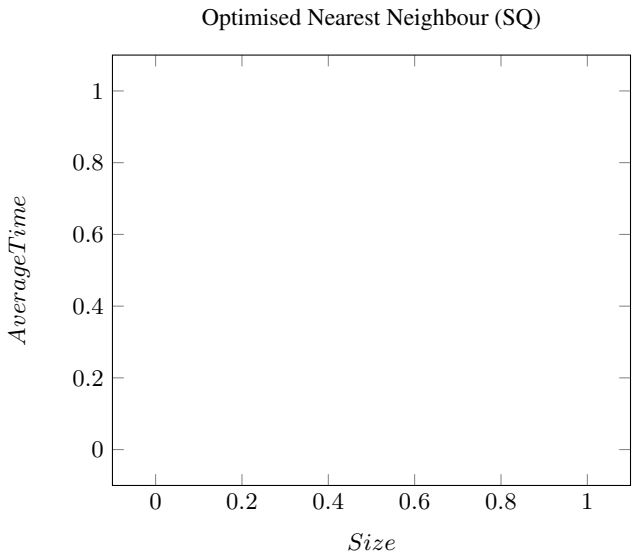
The next series of graphs are to show the data size against the total distance the route provides. As the optimised and rewritten algorithms were only aiming to improve the time taken to calculate the nearest neighbour they will be omitted as the route is the exact same as the nearest neighbour and evidence of this is provided in the tables above.

Optimised Nearest Neighbour (SQ)









Routes This section is simply showing the output of each algorithm on some of the data sets.

RL5915

The route when it is unsorted looks like:

4 Conclusion

To conclude this report, with all the variations tested the best one overall was the nearest neighbour squared. This provides the same distance as the original nearest neighbour however the improvement in the time taken to compute the route has improved significantly. What is interesting to note is that even though the nearest x also had very fast computations majority of the time the route it returned was worse than just leaving the list unsorted.

References

CONTRIBUTORS, W., 2016. Travelling salesman problem. 1