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 in order to improve upon the algorithm. ADD SENTENCE SUMMARY.

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 grows exponential with size. In 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.

2 Method

One possible way of computing a tsp is by using the nearest neighbour algorithm. This involves sorting the list so that the next city in the list is the closest city to the current city.

Pros

- Easy to implement.
- Very quick results for small data sizes.
-

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

```
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.

```
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
        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

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.

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 long routes due to point distributions.

After looking more into the nearest algorithm it was noted that the distance function could be making the algorithm running slower

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

```
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
        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

```
Data: ArrayList input
Result: returns Nearest 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 4: Nearest x neighbour algorithm

```
Data: ArrayList input
Result: returns Nearest 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 5: Nearest y neighbour algorithm

than it needs to be. Therefore one of the variations of the algorithm is the exact same as the nearest neighbour apart from using distance squared instead of distance to avoid having to use a square root (best case was it would only be used once, worst case is being used twice.) Also storing this result would mean that that the function would only ever be called once in each iteration of the for loop.

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
- Ram: 16.0G
- Running from same hard drive each time.

For each data set each algorithm Also while collecting the results extremes on both sides were removed in order to create a more accurate average. This was to take into account Java garbage collection.

Data set rl5915

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

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

	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

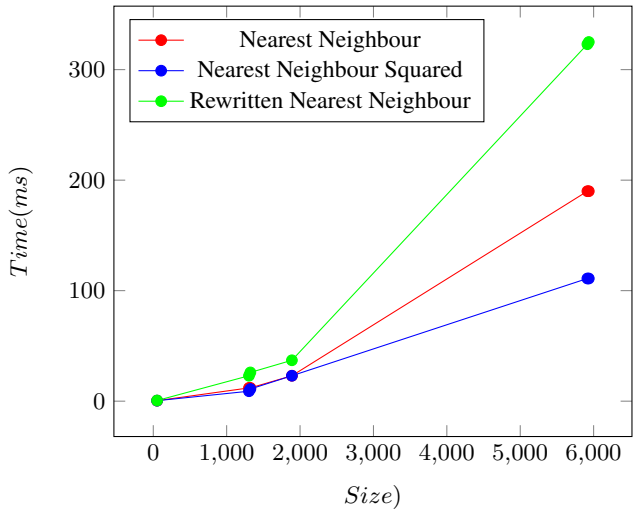
	Total Distance (m)	Time Taken (ms)
Average	17074.13	0.39

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.

nearest neighbour

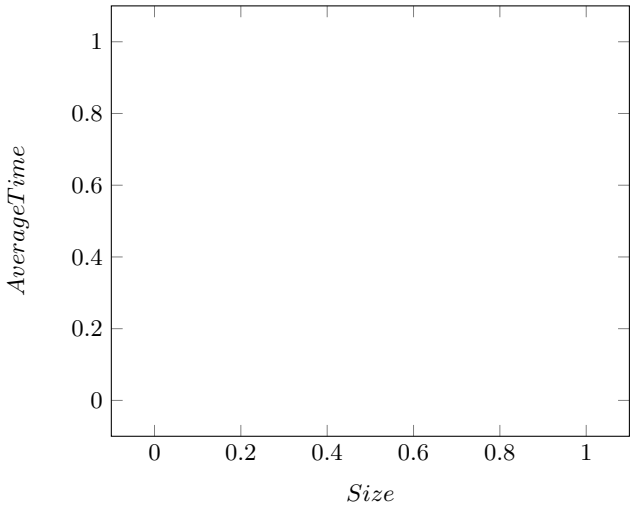


Out of the three basic nearest neighbour algorithms the fastest algorithm is the one that leaves the distances to be compared against in their squared form as well as storing the value from the function as a variable instead of computing it twice. From the graph it is clear that there is a gain in performance 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:

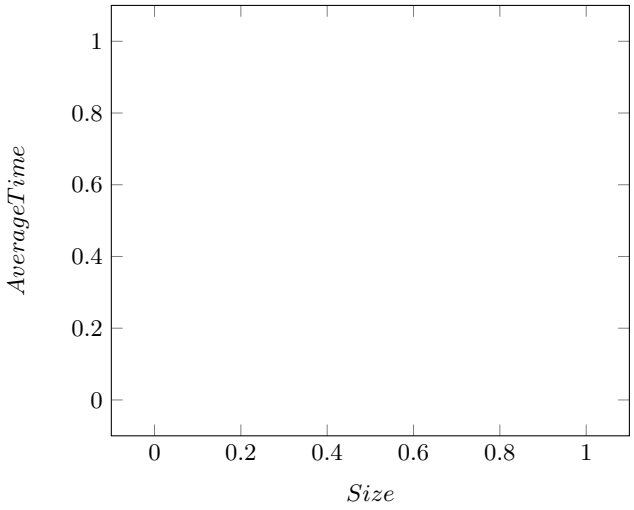
$$Distance = \sqrt{x^2 + y^2} \tag{2}$$

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

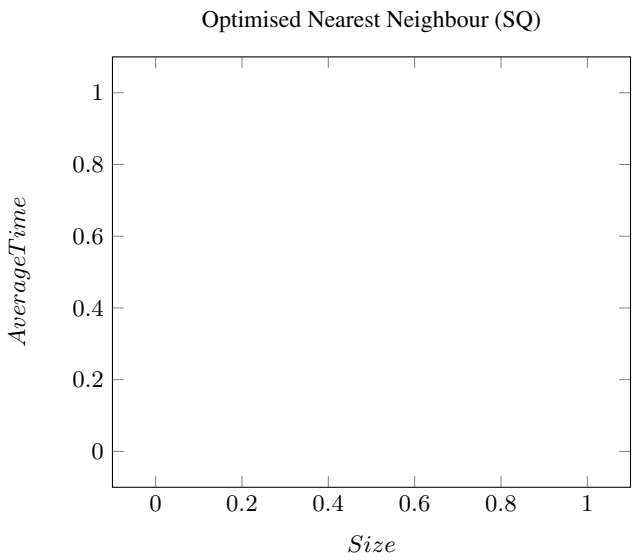
Optimised Nearest Neighbour (SQ)

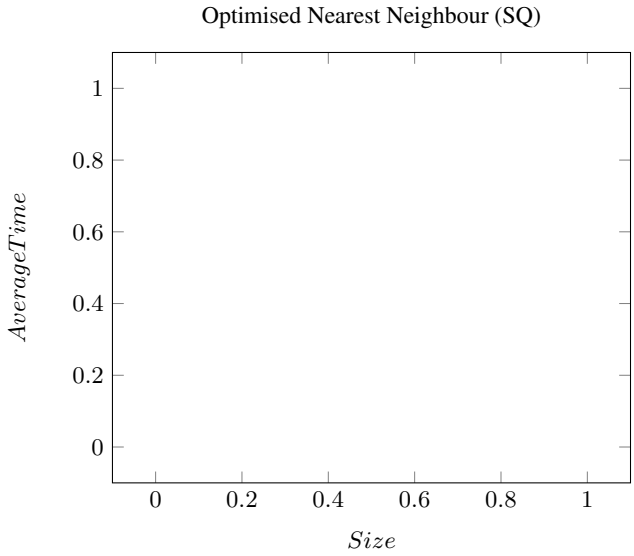


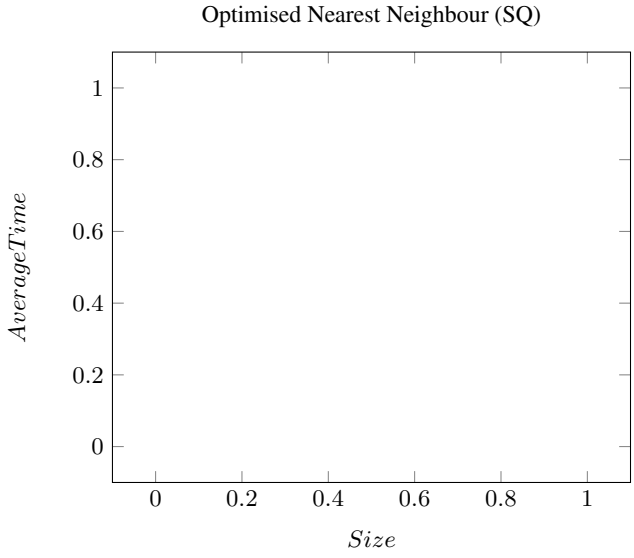
Optimised Nearest Neighbour (SQ)

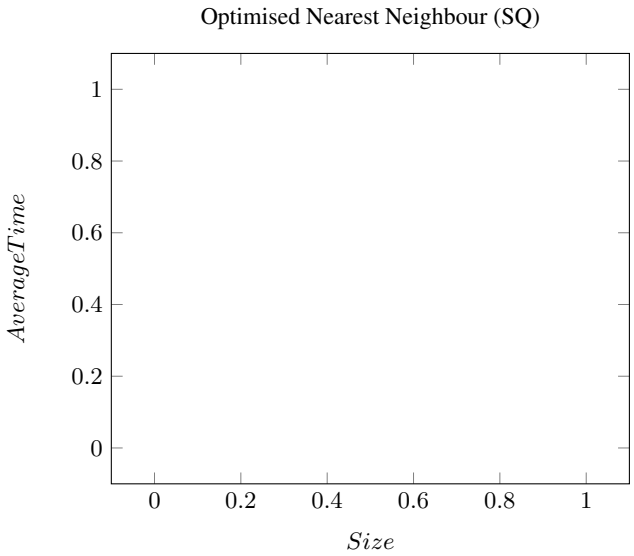


The next series of graphs are to show the data size against the total distance the route provides. As the optimised and rewritten algorithms were only aiming to improve the time taken to calculate the nearest neighbour they will be omitted as the route is the exact same as the nearest neighbour and evidence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









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

Due to the nature of the nearest neighbour algorithm some of the variations implemented would only result in a difference in time taken to compute the algorithm. However it is possible to compare the nearest neighbour with nearest x and nearest y as they produce different routes.

To summarise what has been found from this investigation of the nearest neighbour algorithm in order to solve the travelling salesman problem. The algorithm provides a decent solution in a relatively quick time as long as there is a small

References

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