SET09117 report

# Introduction

The travelling salesman problem is a mathematical problem which seeks to find the best, or ‘cheapest’ solution for a tour of cities. Cheapest refers to a solution which visits each city on the tour only once and only once with the shortest overall distance. (Applegate, et al., 2007) This problem suffers from an exponential increase in possible results as the number of points on the tour increases. This can best be represented with the equation where n equals the number of points and shows that as the number of cities or points increases, the number of possible permutations increases exponentially. While no one is sure of the origin of this problem it is believed that one of the earliest descriptions of the travelling salesman problem can be found in an 1832 German handbook “Der Handlungsreisende—wie er sein soll und was er zu thun hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiss zu sein—Von einem alten Commis-Voyageur” and Princeton university started making developments in TSP in the 1930s. (Applegate, et al., 2007)

The simplest solution in terms of complexity is to just attempt all the possible permutations and to see which one is the ‘cheapest’. This brute force search is suffers from an exponential increase in running time as the number of points on a tour increases, illustrated by the equation . this means that this method of calculating the cheapest solution is impractical for larger point sets. This therefore was the first algorithm to be implemented in the form of the nearest neighbour algorithm. This algorithm takes in an array list of Point2D points that represent cities on a tour. The first city is selected from the list, set to be the current city and added to the results Hashmap which holds the cities in the order they are visited. Then the algorithm iterates through all of the other sets of points and calculates which one is the closest to the first point. This city is then added to the Hashmap and is set to be the current city. This process is then repeated for that city, and every other city until all the cities have been added. The first city is then added again to complete the tour of the cities. Once this is complete each of the cities is iterated through using a function which calculates the distance between the selected city and the next one and these numbers are added together to give the total distance of the tour. While this algorithm does work and give an adequate result, it gives a less than optimal result and for much larger point sets it is a rather slow algorithm. For smaller point sets however. It is a useful algorithm to give a quick and useable, if not the best, result.

A possible simple improvement that was attempted was to randomise the starting point of the algorithm which may allow a shorter path depending on the start location, this however presents a problem as it may also start in a location which would create a longer route through all points. An attempted modification to this was to use the randomised starting point within a loop which allows a set amount of randomised starting points which are calculated and the cheapest one is kept. The downside to this method as it is more time intensive and on larger problems requires a very large amount of time.

The next algorithm is an improved version of nearest neighbour, which functions on the same basic principal as the normal algorithm but in this one there is a function that picks the 3 closest points to the current city and looks forward an number of steps ahead of those 3 cities. Once it has done so it calculates the distance of these 3 separate routes and returns the closest one. The algorithm then moves on to the next city and repeats this method. This method should allow the computer to pick a route that would have a shorter distance for the number of turns it looks ahead. This however, causes problems if there are a collection of close points and more outlying points or if there are some points. In those circumstances, the algorithm goes through all the close points before having to make inefficient moves which may go back through other points or intersect with other lines in order to reach further away points. This can lead to inefficiencies within the route that is calculated and in some cases can even make it longer than the standard nearest neighbour.

The final algorithm in this comparison is the Hybrid Nearest Neighbour algorithm. This algorithm utilises the normal nearest neighbour and then uses the pathfinding that is used in the previous algorithm. To start, the standard nearest neighbour algorithm is ran and the route and distance is stored. Then the pathfinding is looped through, increasing the number of steps that the algorithm looks forward. If the route distance is shorter, that route is stored instead. If it is not, then the route is not stored and the loop increases the amount of steps into the future. The loop can be stopped by entering “x” into the command prompt. Once this is entered the algorithm stops once its current calculation is complete and it returns the shortest route that it has found up to that point. This allows the user to stop the algorithm early if they want a quick solution, or leave it running to find a more efficient route. The use of the nearest neighbour algorithm as a first step means that the route returned can never by more expensive than the nearest neighbour calculation. Once the calculation has been completed or stopped the user can then opt to use a swapping algorithm that is part of Hybrid Nearest Neighbour. This looks at groups of points to assess if swapping them would provide a more efficient route. If this is so it keeps the swap and moves on to the next group of points. This allows for a correction in some of the problems of the pathfinding and is not very time intensive.

{\displaystyle O(n!)}

# Method

In order to test the effectiveness of the algorithms five different point sets with varying complexity were inputted into the algorithms and ran 10 times. The time taken to read in and calculate a solution to each problem was measured and the average times and distances for each point set was calculated. From these the overall effectiveness of the algorithms could be determined as well as how cheap the algorithm was to run.

# Result

The first point set that was used was “Ulysses 22”. This is a 22-point set where most of the points are close together with some outliers. As the graph in Figure 1 below shows. The standard Nearest Neighbour shows an average time of 0ms. This may be due to the computer being unable to read a time less than one millisecond, hence the time of nought. The Randomised Nearest Neighbour had an average time of 12.4 milliseconds and the Hybrid Nearest Neighbour had an average time of 15.5 milliseconds. When compared to the route distance results shown in Figure 2 the Random Nearest Neighbour algorithm gives the best distance with an average of 82.43. While both the normal Nearest Neighbour and the Hybrid Nearest Neighbour returned the same distance of 88.89. For this short data set the random nearest neighbour returned on average the shortest distance for the second best time,while nearest neighbour gave the shortest time and the second best distance. The hybrid nearest neighbour however, gave both the longest distance and time for this dataset.

Figure 1 Ulysses 22 Distances

Figure 2 Ulysses 22 Times in milliseconds

The second of the point sets was “berlin 52”. This is a 52-point set with well-spaced points and medium distances between them. As shown in Figure 3, both the nearest neighbour and random nearest neighbour gave similar times, with 10.8 and 18.7 milliseconds respectively. Hybrid nearest neighbour however, gave a time of 115.7 milliseconds which is quite an increase in time. This is especially true when it is compared to the route distances that the algorithms calculated. As Figure 4 shows, the random nearest neighbour algorithm gives the best average distance while both nearest neighbour and hybrid nearest neighbour worked out the same average distance. Therefore, for this point set, random nearest neighbour was the best algorithm tested when both distance and time cost were taken into consideration. The finished algorithm, Hybrid Nearest Neighbour, was the most computationally expensive for no benefit with both this point set and Ulysses 22. This leads to the assumption that hybrid nearest neighbour is not effective at small number point sets. While randomising the start point of nearest neighbour gave the best results for both these datasets.

Figure 3 Berlin 52 Times in Milliseconds

Figure 4 Berlin 52 Distances

For the three remaining point sets, it was found that the timing differences between the algorithms was so great that a standard bar chart was unable to display the results of the time accurately enough. Due to this, the timing graphs will be changed to logarithmic graphs which will better illustrate the difference in magnitude of the times recorded by the algorithms for better viewing. The average times recorded will still be displayed.

The third point set that was tested was pr144. This is a 144-point set with widely spaced points. As Figure 5 (Overleaf) Shows, the average times for nearest neighbour and random nearest neighbour show times that are in a similar order of magnitude with 15.6 milliseconds and 17.4 milliseconds respectively. The hybrid nearest neighbour algorithm however, gives an average time which is orders of magnitude greater than the other algorithms, with an average time of 4306.4 milliseconds. This can then be compared to the route distances which these algorithms produce (Figure 6) which show that nearest neighbour and the final algorithm, hybrid nearest neighbour, give the exact same average distance of 61401.735. Random nearest neighbour however gives a shorter distance of 61021.735. For this point set, random nearest neighbour is again the most cost effective algorithm when distance and time are taken into consideration. This lends weight to the hypothesis that the hybrid nearest neighbour algorithm does not work effectively with small datasets.

Figure 5 pr144 Log Chart

Figure 6 pr144 Distances

The fourth data set used for testing was a280. This is a 280 point set with groups of closly spaced points and white space seperating them. The time results, shown in Figure 7, show that the random nearest neighbour algorithm is starting to increase in time faster than nearest neighbour due to the looping nature of the algorithm with an average time of 33 milliseconds compared to the standard nearest neighbours time of 15.6 milliseconds . For comparison, the hybrid nearest neighbour algorithm completed the calculation in a time of 58941 milliseconds. The distances shown in Figure 8 show that the random nearest neighbour gives only marignal improvement over the standard nearest neighbour, with distances of 3114.04 and 3115.519 respectively. Hybrid nearest neighnour however, gives an improvemnet in this dataset, with a distance of 3026.975. This improvement does come with a large time penalty however, with average times that are orders of magnitude greater than nearest neighbour.

Figure 7 A280 Distances

Figure 8 A280 Times

The fifth and final data set that was used for testing was ali535. This is a 535 point set with a large group of close points with a medium amount of outliers. Shown in Figure 9, nearest neighbour still gave the best time (16.7 milliseconds) while random nearest neighbour gave a longer time (62.3 milliseconds) and hybrid nearest neighbour gave a time orders of magnitude greater than both of the other algorithms (708026.3milliseconds). That time equates to a time of appoximately 11 minutes and 48 minutes. The hybrid nearest neighbour algorithm therefore, is a much more time intensive algorithm to use for this data size. The route it creates however, is more efficient than both nearest neighbour and random nearest neighbour. Figure 10 shows that hybrid nearest neighbour gives an average distance of 2524.774, a good improvement on the random nearest neighbours 2593.86 and an even better improvement of the standard nearest neighbours distance of 2670.247

Figure 9 Ali535 Times

Figure 10 Ali535 Distances

When all of these results are looked at together, showing the time differences between different data sizes, the results look like the ones presented below in Figure 11 and 12.

Figure 11 Times for Hybrid Nearest Neighbour

Figure 12 Times for Nearest Neighbour and random

These show that the finalised algorithm, hybrid nearest neighbour, increases in time required at a seeming exponential rate.

# Conclusions & Reflections

The results above show that the hybrid nearest neighbour algorithm is slower than nearest neighbour and does not produce better results on small point sets (below 280). It does however give better results above 280, but at a very large time penalty. This penalty takes the form of an exponential increase when looking at Figure 11. Meaning this algorithm would not be an effective algorithm for large datasets. A test was made for running a 1084-point set but this had to be scrapped due to the algorithm not completing the task within a sensible time frame. The algorithm does give reliable results in all data sizes. It always returns the same value when ran repeatedly and within comparable times. The main drawback to this algorithm is the time taken to complete a solution. As shown in the results it is far in excess of the nearest neighbour and random nearest neighbour algorithms and for smaller sets give the same values as nearest neighbour in terms of distance. This algorithm is best used with a function which would allow the user to stop the calculation prematurely and return the best result given at that point, as it would allow the algorithm to be used without having to leave it for the, sometimes very large, full calculation time. This function was implemented in the coursework and allowed an easier use of the algorithm.

# Appendix

## Main Class

**public** **class** Main {

**static** Thread *thread2*;

**static** String *fileLocation*;

**static** ArrayList<Point2D> *cities* = **new** ArrayList<Point2D>();

**static** Map<Integer,Point2D> *result* = **new** HashMap<Integer,Point2D>();

**static** Map<Integer,Point2D> *result2* = **new** HashMap<Integer,Point2D>();

**public** **static** **void** main(String[] args) {

//basic();

//this thread runs the algorithm

Thread thread = **new** Thread() {

**public** **void** run() {

*hybridNearestNeighbour*(**true**, **true**);

}

};

//this thread allows the user to stop the program once the path it is currently calculating is finished

*thread2* = **new** Thread() {

**public** **void** run() {

**boolean** notEntered = **true**;

**do**{

String stopped = **null**;

BufferedReader reader = **new** BufferedReader(**new** InputStreamReader(System.*in*));

System.*out*.println("Enter x to stop at any time");

**try** {

stopped = reader.readLine();

} **catch** (IOException e) {

e.printStackTrace();

}

**if** (stopped.equalsIgnoreCase("x"))

{

HybidNearestNeighbour.*stop*();

notEntered = **false**;

}

}**while**(notEntered);

}

};

thread.start();

}

**public** **static** **void** hybridNearestNeighbour(**boolean** swaps, **boolean** illustration){

BufferedReader reader = **new** BufferedReader(**new** InputStreamReader(System.*in*));

System.*out*.println("Please enter a file to load");

**try** {

//reads in the file location from the command line

*fileLocation* = reader.readLine();

*cities* = LoadCities.*loadTSPLib*(*fileLocation*);

*thread2*.start();

//runs the calculation

*result* = HybidNearestNeighbour.*calculate*(*cities*);

**if**(swaps)

{

//runs the swap sub-routine

*result* = HybidNearestNeighbour.*swap*(*result*);

}

**double** routeLength = HybidNearestNeighbour.*routeLength*(*result*);

System.*out*.println("The length of the nearest neighbour algorithm route with pathfinding is "+routeLength);

**if**(illustration)

{

//draws the final route into an image file

TSPIllustration.*IllustrateRoute*(*result*, (**float**)0.5, *fileLocation*+" Route wth pathfinding");

}

}

**catch** (IOException e)

{

e.printStackTrace();

}

}

//this runs the standard nearest neighbour and the hybrid nearest neighbour to allow a comparison

**public** **static** **void** proof(){

BufferedReader reader = **new** BufferedReader(**new** InputStreamReader(System.*in*));

System.*out*.println("Please enter a file to load");

**try** {

*fileLocation* = reader.readLine();

*cities* = LoadCities.*loadTSPLib*(*fileLocation*);

*result* = NearestNeighbour.*nearestNeighbour*(*cities*);

*cities* = LoadCities.*loadTSPLib*(*fileLocation*);

*result2* = HybidNearestNeighbour.*calculate*(*cities*);

} **catch** (IOException e) {

e.printStackTrace();

}

System.*out*.println("Calculations complete");

**double** routeLength = NearestNeighbour.*routeLength*(*result*);

**double** routeLength3 = HybidNearestNeighbour.*routeLength*(*result2*);

System.*out*.println("The length of the nearest neighbour algorithm route is "+routeLength);

System.*out*.println("The length of the nearest neighbour algorithm route with pathfinding is "+routeLength3);

TSPIllustration.*IllustrateRoute*(*result*, (**float**) 2, *fileLocation*+" Route");

TSPIllustration.*IllustrateRoute*(*result2*, (**float**)2, *fileLocation*+" Route wth pathfinding");

}

//this runs and then outputs the standard nearest neighbour

**public** **static** **void** basic()

{

BufferedReader reader = **new** BufferedReader(**new** InputStreamReader(System.*in*));

System.*out*.println("Please enter a file to load");

**try** {

*fileLocation* = reader.readLine();

*cities* = LoadCities.*loadTSPLib*(*fileLocation*);

*result* = NearestNeighbour.*nearestNeighbour*(*cities*);

} **catch** (IOException e) {

e.printStackTrace();

}

System.*out*.println("Calculations complete");

**double** routeLength = NearestNeighbour.*routeLength*(*result*);

System.*out*.println("The length of the nearest neighbour algorithm route is "+routeLength);

TSPIllustration.*IllustrateRoute*(*result*, (**float**) 5, *fileLocation*+" Route");

}

}

## Hybrid Nearest Neighbour Class

**public** **class** HybidNearestNeighbour {

**static** **boolean** *cont* = **true**;

**public** HybidNearestNeighbour() {

}

**public** **static** Map<Integer, Point2D> calculate(ArrayList<Point2D> points) {

Map<Integer,Point2D> shortestresult = **new** HashMap<Integer,Point2D>();

**double** shortestDistance = Double.*MAX\_VALUE*;

**int** totalCalc;

**int** a = 1;

**do**{

Map<Integer,Point2D> result = **new** HashMap<Integer,Point2D>();

ArrayList<Point2D> cities = **new** ArrayList<Point2D>(points);

totalCalc = cities.size();

Point2D currentCity = cities.remove(0);

**int** i = 0;

result.put(i,currentCity);

**while**(cities.size() > 0)

{

CopyOnWriteArrayList<Point2D> tempcities = **new** CopyOnWriteArrayList<Point2D>(cities);

**double** distance = Double.*MAX\_VALUE*;

Point2D[] closest = **new** Point2D[3];

//get the closest city to the current

**for** (Point2D city: tempcities)

{

**if**(currentCity.distance(city) < distance)

{

closest[0] = city;

distance = currentCity.distance(city);

tempcities.remove(closest[0]);

}

}

**double** distance2 = Double.*MAX\_VALUE*;

//get the second closest city to the current

**for** (Point2D city: tempcities)

{

**if**(currentCity.distance(city) < distance2)

{

closest[1] = city;

distance2 = currentCity.distance(city);

tempcities.remove(closest[1]);

}

}

**double** distance3 = Double.*MAX\_VALUE*;

//get the third closest city to the current

**for** (Point2D city: tempcities)

{

**if**(currentCity.distance(city) < distance3)

{

closest[2] = city;

distance3 = currentCity.distance(city);

tempcities.remove(closest[2]);

}

}

**double** routeDistance = Double.*MAX\_VALUE*;

Point2D closestCity = **null**;

//find which has the shortest route for the number of steps ahead

**for**(**int** j=0;j<=2; j++)

{

**double** pathDistance = *pathfinding*(cities, closest[j], a);

**if**(pathDistance <routeDistance)

{

closestCity = closest[j];

routeDistance = pathDistance;

}

}

//remove the best next city

cities.remove(closestCity);

currentCity = closestCity;

result.put(i,closestCity);

i++;

}

result.put(i, result.get(0));

**if**(*routeLength*(result)<shortestDistance)

{

shortestDistance = *routeLength*(result);

shortestresult = result;

}

**if**(a == result.size())

{

*cont* = **false**;

}

a++;

System.*out*.println("completed "+a+" of "+totalCalc);

}**while**(*cont*);

**return** shortestresult;

}

**public** **static** **double** pathfinding(ArrayList<Point2D> points, Point2D startingPoint, **int** stepsAhead) {

Map<Integer,Point2D> tempresult = **new** HashMap<Integer,Point2D>();

ArrayList<Point2D> tempcities = **new** ArrayList<Point2D>(points);

**int** steps = stepsAhead;

//if the number of cities left is less than the steps to look ahead

**if**(points.size()<stepsAhead)

{

steps = points.size();

}

//if there is no point to start at

**if**(startingPoint == **null**)

{

**return** Double.*MAX\_VALUE*;

}

Point2D currentCity = startingPoint;

**int** i = 0;

tempresult.put(i,currentCity);

tempcities.remove(currentCity);

//while there are still steps to look ahead

**while**(i<steps)

{

Point2D closest = **null**;

**double** distance = Double.*MAX\_VALUE*;

**for** (Point2D city: tempcities)

{

**if**(currentCity.distance(city) < distance)

{

closest = city;

distance = currentCity.distance(city);

}

}

tempcities.remove(closest);

currentCity = closest;

tempresult.put(i,currentCity);

i++;

}

**double** result = *routeLength*(tempresult);

**return** result;

}

//calculate the length of the route

**static** **double** routeLength(Map<Integer, Point2D> route)

{

**double** distance = 0;

**for**(**int** i=0;i< route.size()-1; i++)

{

**if**(route.get(i+1) != **null**)

{

distance += route.get(i).distance(route.get(i+1));

}

}

**return** distance;

}

//swap sub-routine

**static** Map<Integer, Point2D> swap(Map<Integer, Point2D> route)

{

Map<Integer,Point2D> temproute = **new** HashMap<Integer,Point2D>(route);

Map<Integer,Point2D> switchingroute = **new** HashMap<Integer,Point2D>(route);

**int** swaps = switchingroute.size()/4;

**for** (**int** i = 2;i<swaps-1; i+=2)

{

System.*out*.println("optimizing solution "+ (i/2)+"/"+((swaps/2)-1));

//if the first point is closer to the third point than the second point

**if**(switchingroute.get(i-1).distance(switchingroute.get(i+1))<switchingroute.get(i-1).distance(switchingroute.get(i)))

{

Point2D temp = switchingroute.get(i);

switchingroute.put(i, switchingroute.get(i+1));

switchingroute.put(i+1, temp);

}

//if the fourth point is closer to the first point than the third point

**if**(switchingroute.get(i+2).distance(switchingroute.get(i-1))<switchingroute.get(i+2).distance(switchingroute.get(i+1)))

{

Point2D temp = switchingroute.get(i-1);

switchingroute.put(i-1, switchingroute.get(i+2));

switchingroute.put(i+2, temp);

}

//if the second point is closer to the fourth point than the third point

**if**(switchingroute.get(i).distance(switchingroute.get(i+2))< switchingroute.get(i).distance(switchingroute.get(i+1)))

{

Point2D temp = switchingroute.get(i+2);

switchingroute.put(i+2, switchingroute.get(i+1));

switchingroute.put(i+1, temp);

}

//if the route is shorter than the normal route

**if**(*routeLength*(switchingroute)<*routeLength*(temproute))

{

temproute=switchingroute;

}

}

**return** temproute;

}

//stops the program after the current calculation

**static** **void** stop(){

*cont* = **false**;

}

}

# References

Applegate, D. L., Bixby, R. E., Chvátal, V. & Cook, W. J., 2007. [Online]   
Available at: http://press.princeton.edu/chapters/s8451.pdf  
[Accessed 23 October 2016].

## Table of Figures

Figure 1 Ulysses 22 Distances 3

Figure 2 Ulysses 22 Times in milliseconds 3

Figure 3 Berlin 52 Times in Milliseconds 4

Figure 4 Berlin 52 Distances 4

Figure 5 pr144 Log Chart 5

Figure 6 pr144 Distances 5

Figure 7 A280 Distances 6

Figure 8 A280 Times 6

Figure 9 Ali535 Times 6

Figure 10 Ali535 Distances 7

Figure 11 Times for Hybrid Nearest Neighbour 7

Figure 12 Times for Nearest Neighbour and random 7