**Algorithms and Data Structures - SET09117**

**Calum Templeton – 40168316**

**Introduction**

The Traveling Salesman Problem is a problem where a Salesman must travel around different cities (in this case points), visiting each city only once and returning to the starting city. The aim is to choose the shortest route possible. Solving this problem is a difficult challenge.

Starting with a dataset of n points, we are given an X and Y value which is stored in a Point2D Array. To start, the data is sorted through an Algorithm called Nearest Neighbour, which starts at the first value of the data set, then finds the nearest point surrounding it, moves to the point and continues through the dataset until all the points have been visited. Once the results are returned from the Nearest Neighbour Algorithm, they are then put through another algorithm, Five-Point-Switch Algorithm (my own algorithm) which takes five points, starting from the start of the data set, swapping two of these values, doing that for the ten possible combinations, finding the smallest distance between the five points through the various swaps and applying it if it is smaller than the original distance between the five points and it reduces the overall length of the journey. This will be done with the whole data set. Using the results from Nearest Neighbour once more, they are applied to another algorithm called 2-Opt, which removes any crossovers in the journey chosen for the travelling Salesman from Nearest Neighbour Algorithm. This will be done by reordering the points, so the journey has no crossovers. This will give three different journey lengths, one from each of the different algorithms used. There will be different advantages and disadvantages to each algorithm, all which will be analysed and determined.

The expected results for Nearest Neighbour Algorithm are that the algorithm will not give very accurate results, as the Algorithm only finds the point closest to the current point. This is flawed as if one data point is at one side of the data set, the next available point which has yet to be visited might be at the other side of the data set, which means that the journey might have to travel across the data set, creating a/many crossovers meaning the most efficient route will not be found. Nearest Neighbour Algorithm will, however, take a small amount of time to execute, as it only runs the data through a single For Loop which is nested inside a single While Loop. The While condition is only removing data from the data list, meaning Nearest neighbour algorithm will run extremely quickly and will take very little time to execute.

The expected results for Five-Point-Switch Algorithm (my own algorithm) will produce a better journey than the Nearest Neighbour Algorithm as the Five-Point-Switch Algorithm improves upon the Nearest Neighbour Algorithm. Five-Point-Switch Algorithm will hopefully improve the journey length by reordering 5 points, to get a shorter distance between them. This will only be implemented if it is the shortest distance between the 5 points and is smaller than the original distance between the non-reordered 5 points. On top of that, it must make the overall journey length shorter. The advantage to this approach will be that it makes some improvements over short distances, however, the disadvantage is that due to only 5 points being reordered at once, it can’t have an impact on the whole data set, only over a small part of the journey. Five-Point-Switch Algorithm uses a single For Loop which is nested in a Single While Loop. The While loop condition is to only stop when a shorter route cannot be found. This means, that Five-Point-Switch Algorithm will vary in the time taken to execute, depending on the size of the data set.

The expected results for 2-opt Algorithm is that it will improve upon the result produced from Nearest Neighbour as it uses the data return from the Algorithms. 2-opt Algorithm should produce the very efficient result, as it removes any crossovers in the journey, meaning the Traveling Salesman will not have to go back on himself at any stage. 2-opt will, however, not be very efficient in terms of time, as 2-opt Algorithm uses two For Loops, one nested inside the other, all while being in a While Loop. Having two For Loops, one nested inside the other will be very costly in terms of time if the data set is large due to the fact it will take a long amount of time to go through all the cities. The While Loop will also cause problems, because if the two For Loops keep finding shorter distances, then the While Loop will keep going making for an extremely long run time.

References to 2-Opt and Polygon Drawing.

# Andy (2012). C++ Implementation of 2-opt to the “Att48” Travelling Salesman Problem. Available at: http://www.technical-recipes.com/2012/applying-c-implementations-of-2-opt-to-travelling-salesman-problems/

# Kroukamp, D (2013). [about drawing a Polygon in java](http://stackoverflow.com/questions/15188238/about-drawing-a-polygon-in-java) Available at: http://stackoverflow.com/questions/15188238/about-drawing-a-polygon-in-java

**Experimental Methods**

Running the data set through the Nearest Neighbour Algorithm would produce a journey which wasn’t very efficient. Nearest Neighbour Algorithm would, however, give a starting point. Looking at the output for Nearest Neighbour Algorithm, there was lots of crossovers, but also inefficient journeys between small groups of points. There were a few instances where it was clear, that the reordering of a few points could cut the journey length by a small amount. Doing this multiple times in a large data set could cut a large distance from the journey. From this observation, it was best to create an algorithm which would solve this problem.

To start, the algorithm will load in the results from Nearest Neighbour Algorithm, to improve upon those results and make the journey length smaller. It will then put the cities into a loop, allowing for points to be switched around easily using their index values. Using 5 points, A, B, C, D, E, these will allow for multiple different calculations to be performed, hopefully leading to a shorter journey. The algorithm will start at the top of the data set, using the first 5 cities. These starting 5 cities would have the distance between them calculated. This would be done by calculating the distance between A and B, B and C, C and D, then finally D and E. These will then all be added together to give the collective distance between the 5 points.  Along with that, the distance of the whole journey, not just between these 5 points, will be calculated, as there will be possibilities when changing the order of the 5 points, to get the shortest distance between them, that it may actually increase the length of the overall journey, which is going against what is trying to be achieved. Once the initial distances have been calculated, every single other combination, which in this case will be 9 additional combinations, will also have their distances between the 5 points and overall journey length calculated. The different combinations, between the 5 points will then have all their distances to an array, to find the smallest. Similarly, the different combinations for the overall journey distances will also have their distances added to a separate array. The smallest distance between the 5 points will then be found and using the index, the corresponding value for the overall distance will be found. The smallest distance between the 5 points will then be compared against the original distance between the five points. If the new distance between the 5 points is less than the original distance between the 5 points and the overall new distance for the total journey is less than the overall original distance for the total journey then the points in the new distance will set to the data set, overwriting the old ones. This will mean that swaps in points will only be made if the make the overall journey a shorter distance. This gives an accurate result as there is no way possible to make the journey longer. This will then continue through all the points in the dataset, incrementing by 1 each time, until it reaches the end of the data. This will then be placed in a while loop and only exit once a shorter distance for the journey can’t be achieved.

One worry is by setting points and overwriting them with different points, some points might be lost, leading to an inaccurate journey. No points will be lost in this algorithm, as points overwrite each other by setting through an index. Every time two cities are switched, a temporary variable is made, to make sure no data is lost during the switch. The pseudocode in Figure A shows how the temporary variable works.

Create a temp point

Make temp equal citiesA

Set CitiesB in the index of CitiesA

Set temp, which holds CitiesA to the index of CitiesB

Return the altered data set

Figure A showing how switching points is done.

The example in Figure A shows CitiesA and CitiesB being swapped around. It shows that if one city has another city overwriting its location in the list/has another city being set to its index, then a city is not lost, as it is stored in the temporary variable.

To ensure repeatability in the results, the same computer will be used to run all the test. The test will all be carried out at the same time, with little to no other programs running in the background. For time calculations, the time will be recorded 3 times for each algorithm and the average will then be used, to give a more accurate result.

2-Opt will be implemented alongside the Five Point Switch Algorithm. 2-Opt is an already established algorithm, meaning it can reliably give an arcuate result, therefore is near enough guaranteed to work.

**Experimental Results**

Before starting with any data set, the assumption is that a small data set should take a shorter amount of time to run compared to a larger data set. In general, the more cities there are in the data set, the longer the program takes to execute.

For the time taken to run the program, the program will be run has three separate algorithms, all with their own individual tunes calculated along with the speed of the overall program. This will allow for analysation and cost benefit with time on the individual algorithms. Another reason why the overall speed is separate is that the program not only runs all three algorithms but also has to load the data, print the data and assign and display to a Polygon. The program will also remove any duplicates in the data at the start of the program, which will take up a small amount of time, though this will depend on the size of the data set. The Polygon will consist of vertex’s (lines) which connect the cities, which are mainly white dots. There are some different coloured dots, however. The green dot is the start of the data set, yellow is the second city and the red is the last, just to show what direction the Travelling Salesman takes.

Note that times will be recorded three times. This is due to external factors in with the computer that could alter the run time, so by recording 3 times across multiple program runs, this will produce more accurate and reliable data.

Starting with a small data set, Berlin52 holds 52 cities. Due to the small data size, it will be easy to identify improvements between the different algorithms. The run times will also be incredibly short.

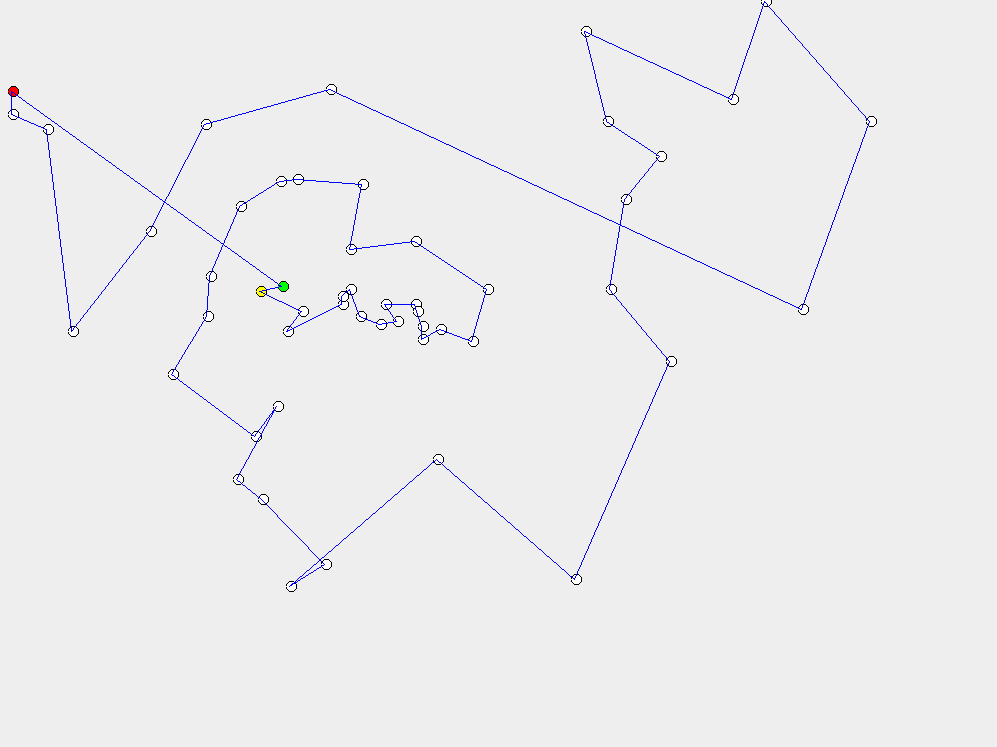
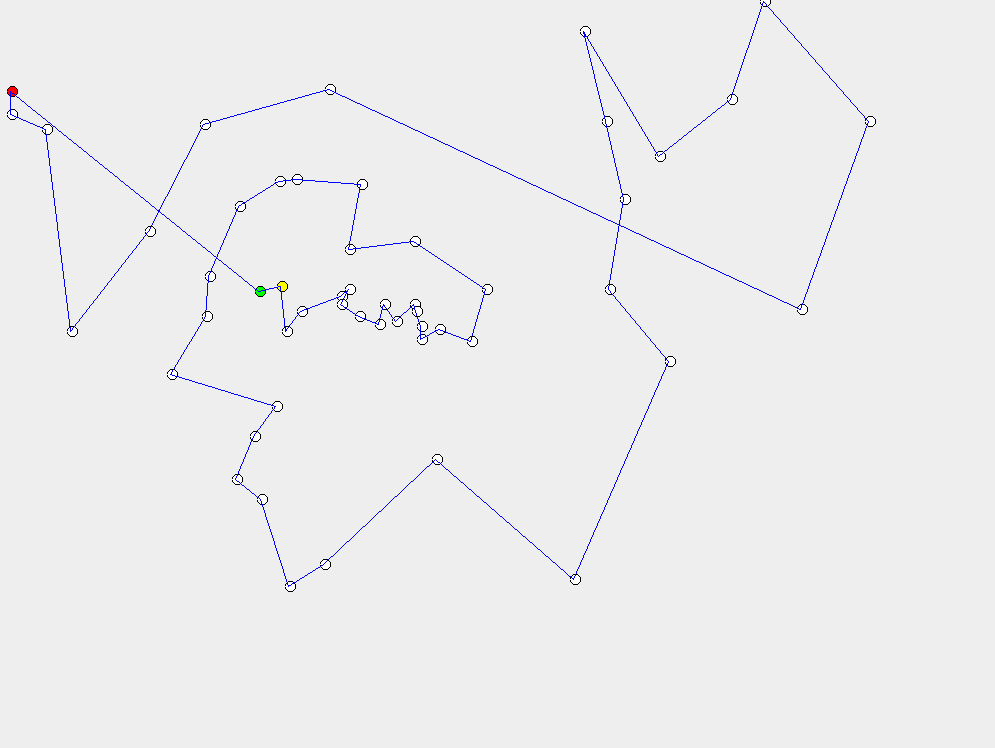
 

Figure 1 (left) showing Nearest Neighbour Algorithm and Figure 2 (right) showing Five Point Switch Algorithm running on the Berlin52 data set.

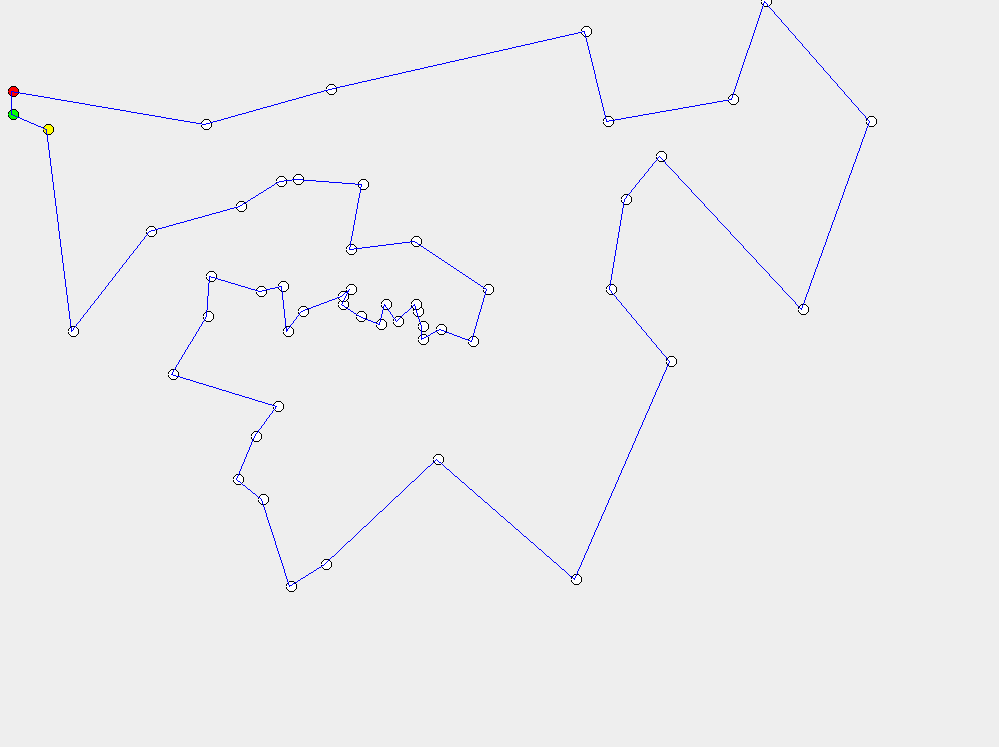


Figure 3 showing 2-Opt Algorithm running on the Berlin52 data set.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | Input Length | Nearest Neighbour | Five Point Switch | 2-Opt |
| Length | 22205 | 8980 | 8843 | 8121 |

Figure 4 showing the overall lengths of each for each Algorithm for the Berlin52 data set.

Starting with Figure 1, you can see that Nearest Neighbour Algorithm has produced a journey which has a lot of overlapping, along with vertex’s going back on each other. This creates for an inefficient route, where there is a lot of room for improvement, to make the journey length shorter. This does, however, give a good starting point, reducing the Input Length by about 60%. Looking at Figure 2, this resolves some, but not all of the issues in Figure 1 with Nearest Neighbour. Vertex’s going back on each other have been removed, which is shown at the bottom of Figure 2, which show’s a more efficient and direct route. Five Point Switch Algorithm, however, does not remove much crossover in the data, leaving room for improvement. The different in length between Nearest Neighbour Algorithm and Five Point Algorithm is only 137, which doesn’t improve the length by that much, however, every little improvement helps, as long as it doesn’t take too long to run. Finally, looking at Figure 3, you see all that all crossovers have been removed. This is thanks to the way 2-Opt works, as its main purpose is to reorder itself so that no crossovers occur. The difference between Five Point Switch and 2-Opt Algorithms is 722 which is much larger than the difference between Nearest Neighbour and Five Point Switch Algorithms, which shows that 2-opt is quite effective.

Figure 5 showing the time taken to run each Algorithm for the Berlin52 data set.

As we can see from the results in Figure 5 that the Nearest Neighbour Algorithm takes no time to run at all. This not surprising due to the small amount of data and the efficiently of the algorithm. All three runs of Nearest Neighbour all produce the same time, 0.01 milliseconds, which is extremely fast. Five Point Switch Algorithm produces a slightly longer time, which again is understandable due to the layout of the algorithm. The three times recorded for the algorithm are very similar and only vary by 2 milliseconds. Finally 2-Opt Algorithm, similarly to Five Point Switch Algorithm, produces slightly longer times than Nearest Neighbour Algorithm. This is a little unexpected, as the complexity of the 2-Opt Algorithm would suggest that it would take much longer to execute. The improvement, shown in Figure 4 from 2-Opt would also indicate a longer run time, however, this may be down to the small data size. Looking at the overall run time, it’s over 4 times the combined time of the three algorithms. This is strange, as it would be expected that the algorithms would take up most of the run time. This, again, could be put down to the small data size.

Moving on to a larger data set, ch130 has 130 cities. This is a larger data set, however, is still a small amount of cities in general.

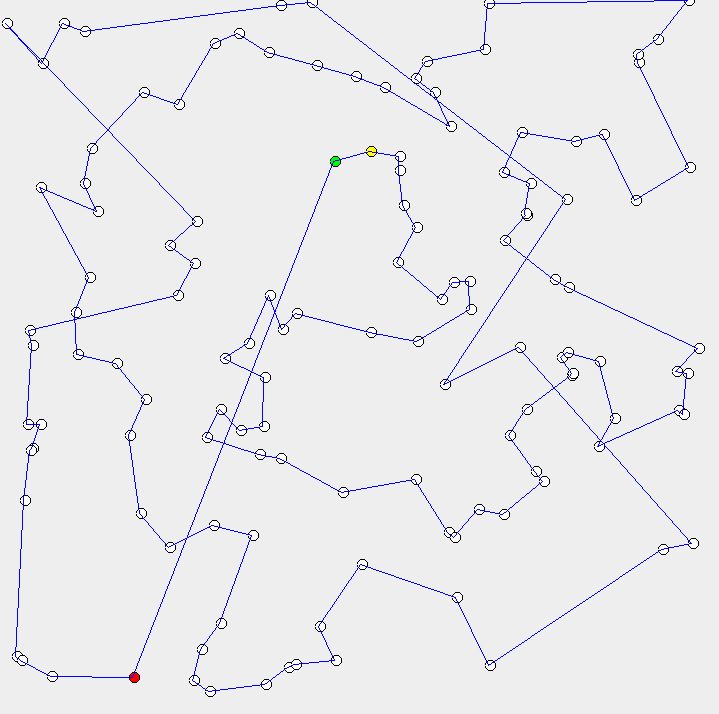
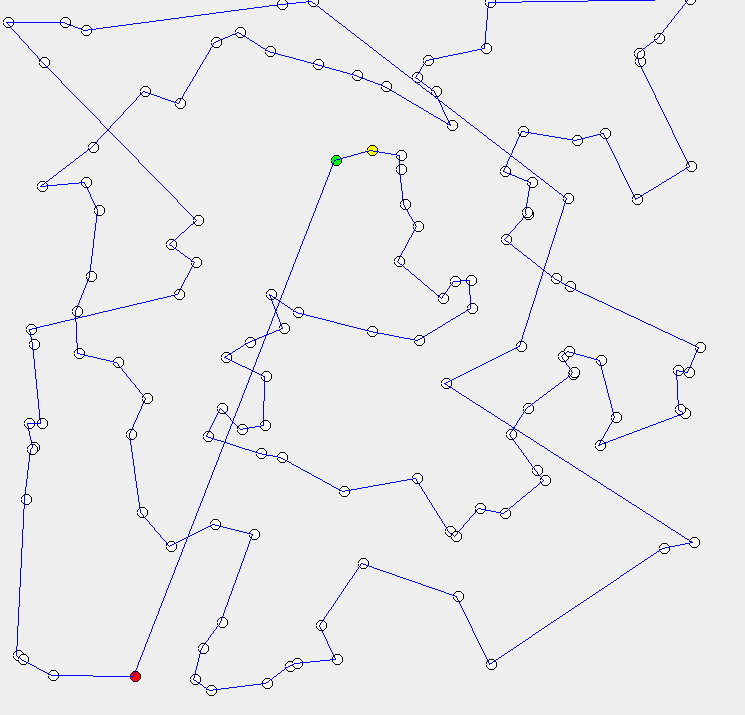
 

Figure 6 (left) showing Nearest Neighbour Algorithm and Figure 7 (right) showing Five Point Switch Algorithm running on the Cho130 data set.

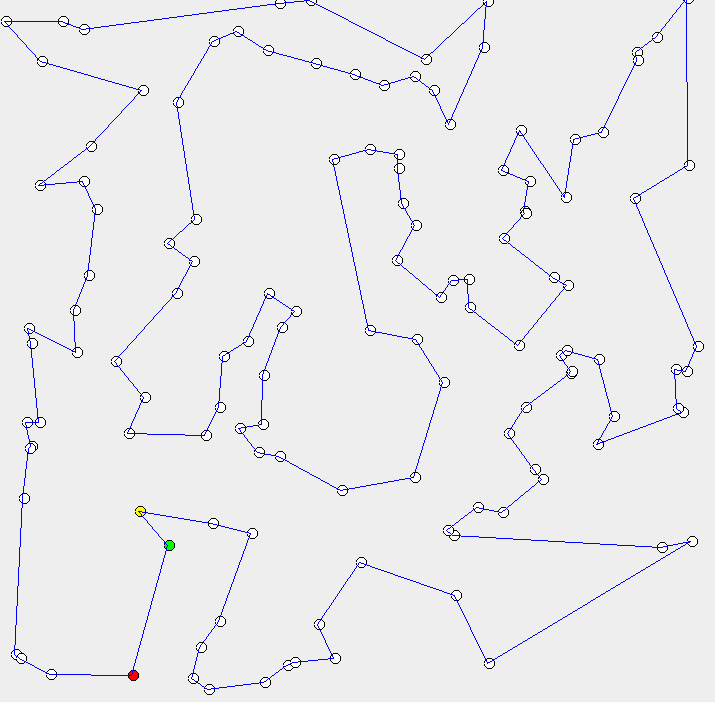


Figure 8 showing 2-Opt Algorithm running on the Ch130 data set.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | Input Length | Nearest Neighbour | Five Point Switch | 2-Opt |
| Length | 47800 | 7575 | 7467 | 6771 |

Figure 9 showing the overall lengths of each for each Algorithm for the Ch130 data set.

Starting with Figure 6, you can see that Nearest Neighbour Algorithm has produced a journey which has, again, a lot of overlapping, along with vertex’s going back on each other. This does, however, give a good starting point, reducing the Input Length by about 70% and gives areas for Five Point Switch and 2-Opt to improve on. Looking at Figure 7, no crossovers have been removed however, the order of some cities have been changed, to avoid vertex’s going back on each other. This is shown in the change between Nearest Neighbour and Five Point Switch as it only reduces the journey length by 108, which is a very small amount. You can also see, in Figures 6 and 7, that when the Algorithms reach the last city (highlighted in red), that the journey has to cut across the data set, which is very inefficient. 2-Opt Algorithm, again, resolves all the issues with Nearest Neighbour and Five Point Switch, by removing all crossovers and making an efficient route.

Figure 10 showing the time taken to run each Algorithm for the Ch130 data set.

As we can see from the results in Figure 10 that Cho130 Nearest Neighbour Algorithm takes no time to run at all. This not surprising due to the small amount of data and the efficiently of Nearest Neighbour and is similar to Berlin52 Nearest Neighbour’s results. Five Point Switch Algorithm produces a slightly longer time, which again is understandable due to the layout of the algorithm. This, however, is much larger than Nearest Neighbour Algorithm, which is expected. There is a much larger gap between the two Algorithms compared with the smaller 52 cities used in Berlin52. 2-Opt Algorithm again is much larger than Five Point Switch. This is expected and follows a trend between Nearest Neighbour and Five Point Switch Algorithms. Looking at the overall run time, this is now roughly double the length of 2-Opt. This, compared to Berlin52 times, is meaning the algorithms are taking up at least half of the run time, which is much more expected, compare to the Berlin52 data set outcome.

Moving on to a much larger data set, fl417 has 417 cities in the data set.

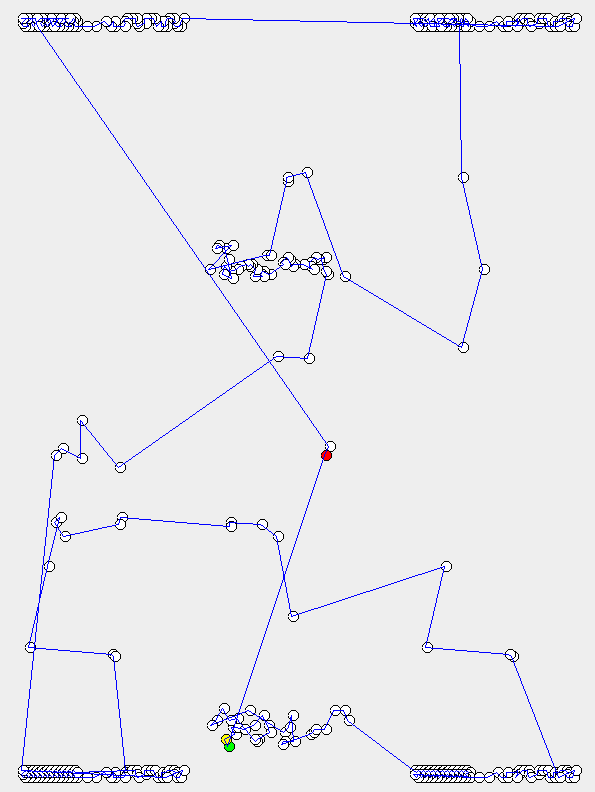
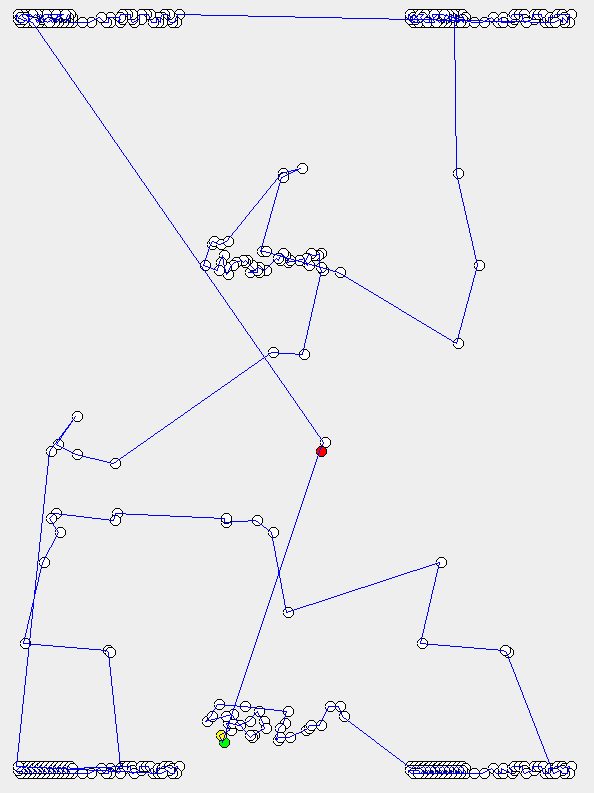


Figure 11 (left) showing Nearest Neighbour Algorithm and Figure 12 (right) showing Five Point Switch Algorithm running on the Fl417 data set.

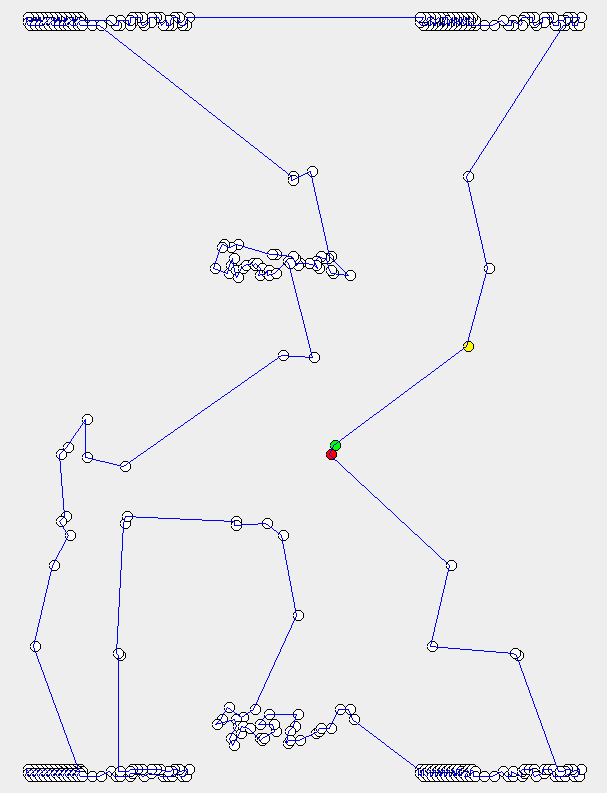


Figure 13 showing 2-Opt Algorithm running on the Fl417 data set.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | Input Length | Nearest Neighbour | Five Point Switch | 2-Opt |
| Length | 55387 | 15737 | 15616 | 12870 |

Figure 14 showing the overall lengths of each for each Algorithm for the Ch130 data set.

Starting with Figure 11, you can see that Nearest Neighbour Algorithm has produced a journey, which is very messy and unclear. This is partially to do with the data set having cities extremely close together. Figure 11 does show well, with this data set, that there are only a few improvements to be made, mainly in the middle of the diagram/data set. This backed up by the difference between Nearest Neighbour and Five Point Switch Algorithms, which is only 121. Looking at Figure 12, you can see a few crossovers have been removed but more importantly, some points which are close together (mainly in the centre of Figure 12) have been rearranged. This does not make the journey much shorter, as the points are so close together, however, it does make for a better and more efficient journey. 2-Opt Algorithm, once again, resolves all the issues with Nearest Neighbour and Five Point Switch, by removing all crossovers, however, this time improves a lot on Five Point Switch, switching many points which are close together, to make a more efficient route. This is visible by comparing the bottom centre of Figure’s 12 ad 13.

Figure 15 showing the time taken to run each Algorithm for the Fl417 data set.

As we can see from the results in Figure 15 that Fl417, Nearest Neighbour Algorithm takes so little time, compared with the other Algorithms that it is hardly shown on the graph. This, however, is not surprising, and with a few results now produced from the different data sets, there is a pattern appearing, where the more cities there are in a data, the larger the gap between the algorithms. Five Point Switch Algorithm is now roughly a twentieth the size of 2-Opt. This has increased dramatically and is another example of the growing gap in times between the algorithms. We can now see that 2-Opt Algorithm also takes up about 90% of the run time.

Continuing to increase the data set size, d657 contains 657 cities.

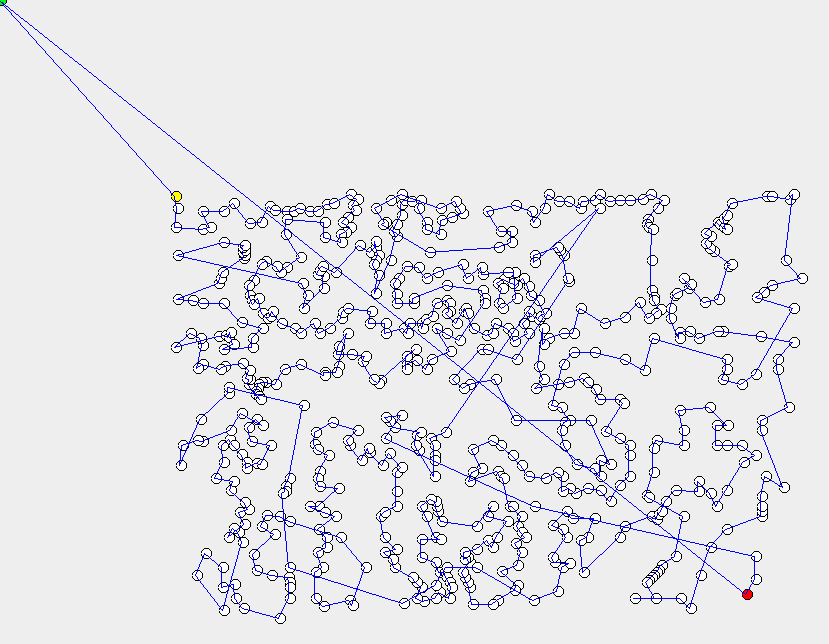
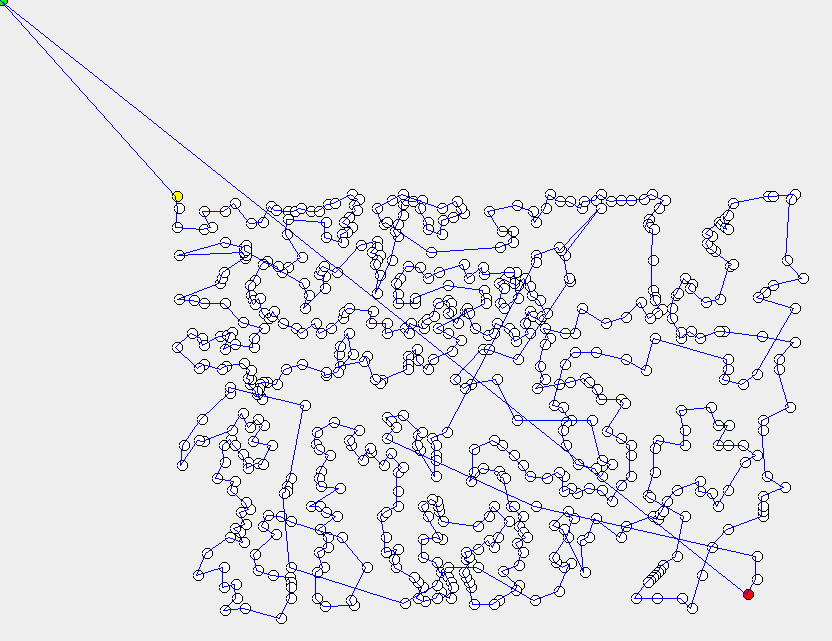
 

Figure 16 (left) showing Nearest Neighbour Algorithm and Figure 17 (right) showing Five Point Switch Algorithm running on the d657 data set.

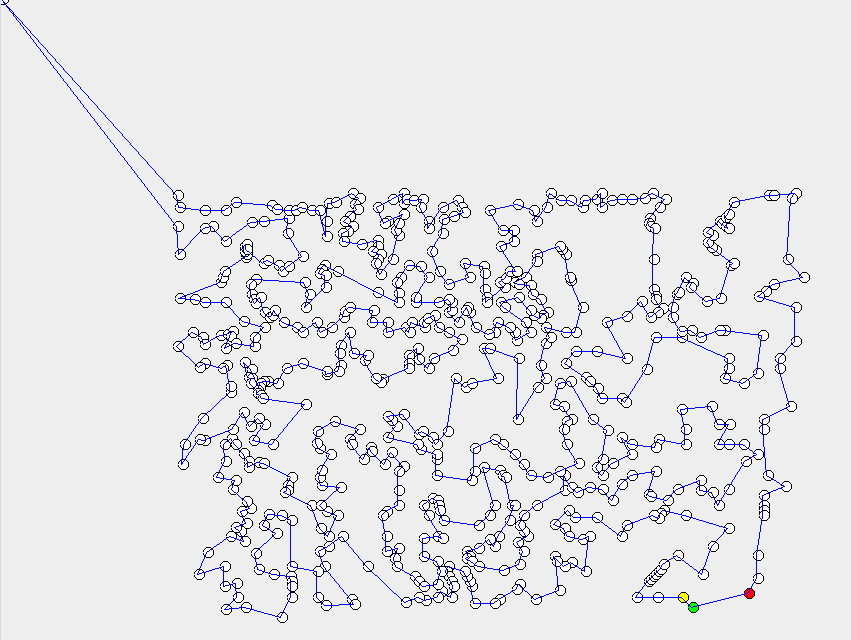


Figure 18 showing 2-Opt Algorithm running on the d657 data set.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | Input Length | Nearest Neighbour | Five Point Switch | 2-Opt |
| Length | 232140 | 61874 | 60593 | 52048 |

Figure 19 showing the overall lengths of each for each Algorithm for the Ch130 data set.

Starting by looking at Figures 16, 17 and 18, the 657 points are all closely located together, meaning that the algorithms will not be able to improve the journey distances dramatically. Looking at Figure 16, Nearest Neighbour Algorithm is actually looking relatively organised compared to previous Nearest Neighbour results over different data sets. This may, however, be down to the data. Looking at Figure 17 and comparing it to Figure 16, there are not many differences, which is backed up by Figure 19, as the difference between Nearest Neighbour and Five Point Switch is only 1281, which is not a lot considering a number of cities and the input length. A few crossovers have been removed and some points have however been arranged. Finally looking at Figure 18, we can see the 2-Opt has successfully run, removing any crossovers and making a much more efficient route. One major thing to note is that the starting point in Figures 16 and 17 is in the far top left corner, but in Figure 18, through 2-Opt Algorithm, the starting point is in the bottom right corner, the complete opposite.

.

Figure 20 showing the time taken to run each Algorithm for the d657 data set.

As we can see from the results that d657 that Nearest Neighbour Algorithm takes so little time, compared with the other Algorithms that again it isn’t even shown on the graph. This graph has moved from milliseconds to seconds as the range difference between Nearest Neighbour and the Overall Runtime has increased so much that it was required. Again the time gaps are continuing to grow between the Algorithms, with the overall runtime now at over 12 seconds.

Finally looking at the largest data set, d1291, which contains 1291 cities.

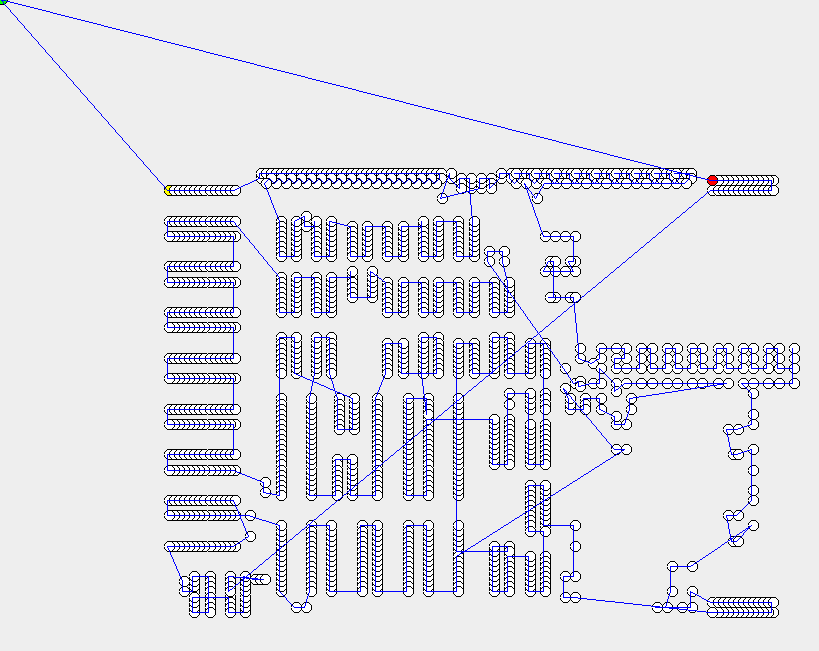
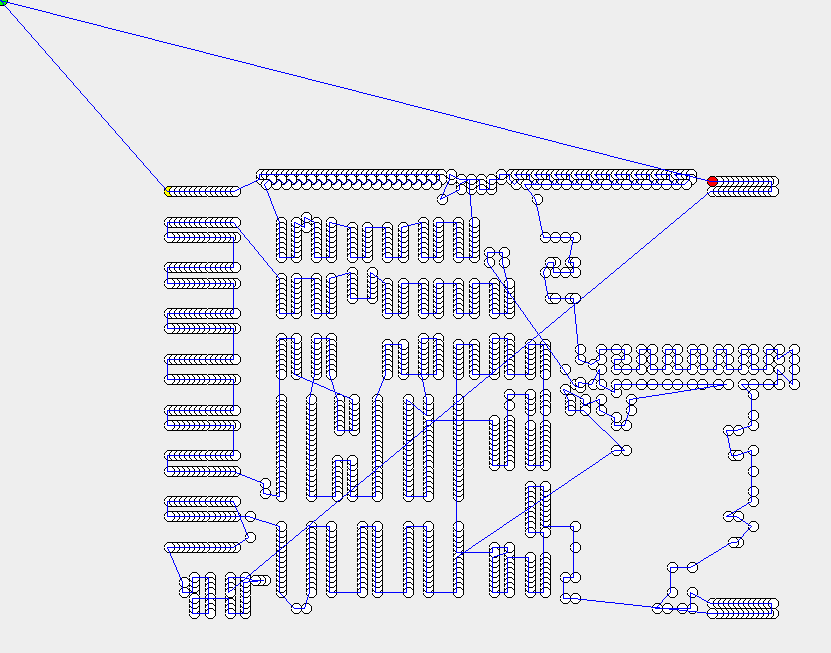
 

Figure 21 (left) showing Nearest Neighbour Algorithm and Figure 22 (right) showing Five Point Switch Algorithm running on the d1291 data set.

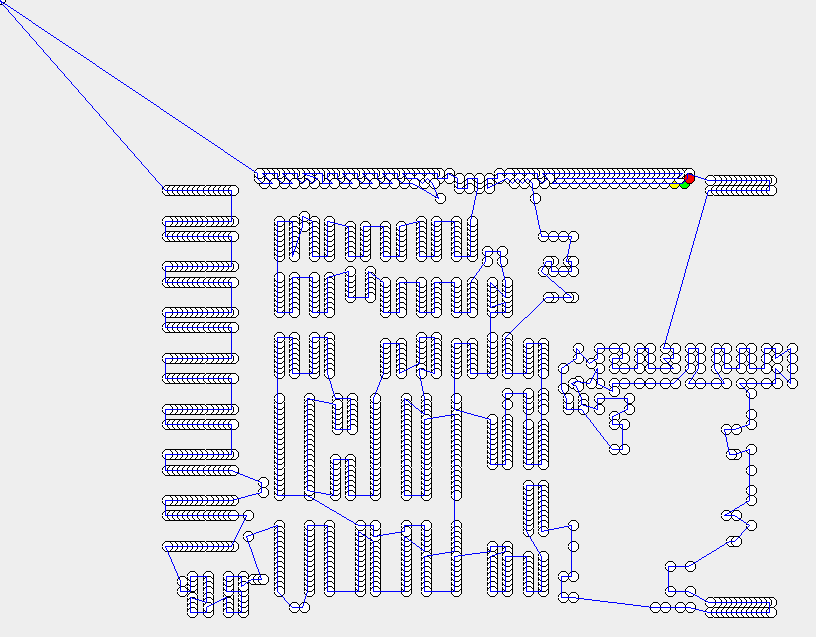


Figure 23 showing 2-Opt Algorithm running on the d1291 data set.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | Input Length | Nearest Neighbour | Five Point Switch | 2-Opt |
| Length | 150990 | 60996 | 60261 | 54771 |

Figure 24 showing the overall lengths of each for each Algorithm for the Ch130 data set.

Starting by looking at Figures 21, 22 and 23, the 1291 cities are grouped in different groups. This will mean that not many improvements will be made in the groups of cities, however, the journey in-between the groups may be altered and that is where the improvements from the Algorithms will come from in this data set. With clusters of cities, this means there will not be much difference in Nearest Neighbour and Five Point Switch, and this is backed up in Figure 24, as the difference between the 2 algorithms is less than 700. There is not much difference between Figures 21 and 22, the Nearest Neighbour Algorithm and Five Point Switch Algorithm, with only minor, small, near enough unnoticeable changes being made. Figure 23, however, looks a fair bit different from Figures 21 and 22, as the crossovers have been removed, resolving the issues with Nearest Neighbour and Five Point Switch Algorithms.

Figure 25 showing the time taken to run each Algorithm for the d1291 data set.

Finally, as we can see from Figure 25 that d1291, the range is extremely large with the Overall Runtime reaching over 100 seconds. We can see that 2-Opt is taking up 95% of the run time, with Nearest Neighbour taking up less than 1% of the run time. Due to the increase of time, which comes from the increased amount of cities, we can assume that the Overall Runtime will increase dramatically if another dataset with a lot of cities is run.

Figure 26 showing the relationship between cities and time taken to execute the program.

Looking at Figure 26, we can see the correlation between times taken to run the data sets, in seconds, against the number of cities within the data sets. We can see from the left of the graph, both the times and cities are small, then they both slowly begin to rise in the middle of the graph before skyrocketing at the right side of the graph. This is due to the way the Algorithms run, more specifically 2-Opt, which we found from the previous time graphs on the different data sets. It is because 2-Opt uses two for loops, one nested inside the other. For a data set, such as Berlin52, 2-Opt will not take long to run, as it just needs to run through the cities 52 times, 52 times over, which for a modern computer, doesn’t take very long, as you can see from Figure 26. However, 2-Opt running on the d1291 data set will have to loop around the all the 1291 cities, 1291 times, which is going to take a long time, hence the sudden increase in time at the right of the graph.

From this graph, we can confirm the assumption we made at the start of the Experimental Results section which was “the more cities there are in the data set, the longer the program takes to execute”. This is clearly shown in Figure 26, as the rise in time, from left to right, is in correlation with the rise in cities. We can confirm this assumption. If another data set was to be added with more cities than d1291, which contains 1291 cities, then the time will continue to increase dramatically.

**Conclusion and Reflections**

Overall, the results obtained are not surprising and met the analysis and expectations laid out in the Introduction and Experimental Methods. The findings were that for the more cities in the data, the longer the program will take to execute. On top of the results from the different algorithms showed that that Nearest Neighbour was very effective, as it initially organised the input values. It was often found that Nearest Neighbour length would reduce the input length by about 60%, sometimes even more while taking a very short amount of time to run. Five Point Switch Algorithm, my own personal Algorithm, was not as effective as Nearest Neighbour Algorithm, however, did lower the distance slightly, usually only by a few percent, but as Five Point Switch Algorithm took such a small amount of time to run, which is shown in the time graphs throughout this document, it was still a worth running. The improvements in length that Five Point Switch Algorithm made were very dependent on the data set. The Algorithm worked much better when points were spread out and not clustered together. This was due to Nearest Neighbour organising points in clusters well, meaning Five Point Switch Algorithm would not be very effective. 2-Opt was by far the best Algorithm, giving the shortest journeys on all the data sets, improving on both Nearest Neighbour and Five Point Switch Algorithms by some length. 2-Opt, however, takes a long time to run, especially with large data sets.

To conclude, if you need a journey in a short time, use Five Point Switch Algorithm, which does not take very long to execute, however, will not give the most efficient journey. If you want an efficient journey, then use 2-Opt Algorithm, however, it will take a long time to produce a journey.

To make sure the results are accurate, a check is done to make sure there are no duplicates in the data set, meaning that no city is visited twice in the journey, only the start/end point. To make sure the results are reliable, all the results were carried out on the same machine, with no other programs running in the background. Comparing the times with the cities, the average time was used, from 3 different runs/executes, to give a more reliable and accurate comparison.

Overall, reflecting on the Algorithms, Five Point Switch did make the Nearest Neighbour output shorter, however in all 5 instances tested, not by much, only a few percent. Five Point Switch had a good idea behind it, however only switching 5 points around didn’t make a large enough difference. Switching more points around would have been better, however, that would have led to more cost in terms of time. Implementing 2-opt was a good idea, as it created a short journey, with not crossovers. It may have been better to improve 2-Opt and make it less time consuming than to create a separate new Algorithm, however, it is good to have more than one option/algorithm to choose from when trying to solve the Traveling Salesman Problem.

**Source Code/Appendix**

**Demo Class**

// Libraries used for either drawing the Polygon or the Data

**import** java.awt.Color;

**import** java.awt.Dimension;

**import** java.awt.Graphics;

**import** java.awt.Polygon;

**import** java.awt.event.MouseAdapter;

**import** java.awt.event.MouseEvent;

**import** java.awt.geom.Point2D;

**import** java.util.ArrayList;

**import** javax.swing.JFrame;

**import** javax.swing.JPanel;

**import** javax.swing.SwingUtilities;

**public** **class** Demo {

// Declaring variables all used for drawing the Polygon

**static** **int** *size*;

**static** **int**[] *xvalues*;

**static** **int**[] *yvalues*;

**private** JFrame mainMap;

**private** Polygon poly;

**public** Demo(){

initComponents();

}

// Private method which create a JFrame and draws a Polygon to display the data

**private** **void** initComponents() {

// Create a new JFrame which is drawn on

mainMap = **new** JFrame();

mainMap.setResizable(**false**);

mainMap.setDefaultCloseOperation(JFrame.*DISPOSE\_ON\_CLOSE*);

// Create arrays to store the x and y values of the points

**final** **int** finalxposition[] = *xvalues*;

**final** **int** finalyposition[] = *yvalues*;

**double** scalex[] = **new** **double** [*size*];

**double** scaley[] = **new** **double** [*size*];

// Scale points

**for** (**int** i=0; i<*size*; i++)

{

// Cast point to double

scalex[i] = (**double**) finalxposition[i];

// Multiply by value to scale

scalex[i] = scalex[i]\*0.2;

// Cast back to int as data must be int to be drawn

// Repeat for Y

finalxposition[i] = (**int**)scalex[i];

scaley[i] = (**double**) finalyposition[i];

scaley[i] = scaley[i]\*0.2;

finalyposition[i] = (**int**)scaley[i];

}

// Add postions to the Polygon

poly = **new** Polygon(finalxposition, finalyposition, finalxposition.length);

// Set certain points to different colours to identify where the data starts and stops

**final** Color[] colourpoint = **new** Color[*size*];

colourpoint[0] = Color.*GREEN*;

colourpoint[1] = Color.*YELLOW*;

colourpoint[*size*-1] = Color.*RED*;

// Create a new JPanel

JPanel p = **new** JPanel() {

**private** **static** **final** **long** *serialVersionUID* = 1L;

@Override

**protected** **void** paintComponent(Graphics g) {

**super**.paintComponent(g);

**for** (**int** i =0; i < *size*; i++)

{

// Add all the points and give them appropriate colours

g.setColor(Color.*BLACK*);

g.drawOval(finalxposition[i]-3,finalyposition[i]-5, 10, 10);

g.setColor(Color.*WHITE*);

g.setColor(colourpoint[i]);

g.fillOval(finalxposition[i]-3,finalyposition[i]-5, 10, 10);

}

// Draw the Polygon

g.setColor(Color.*BLUE*);

g.drawPolygon(poly);

}

// Set Dimesnsions of the window

**public** Dimension getPreferredSize() {

**return** **new** Dimension(1000, 1000);

}

};

// Set mouse properties

MouseAdapter ma = **new** MouseAdapter() {

@Override

**public** **void** mouseClicked(MouseEvent me) {

**super**.mouseClicked(me);

**if** (poly.contains(me.getPoint())) {

System.*out*.println("Clicked polygon");

}

}

};

p.addMouseListener(ma);

mainMap.add(p);

mainMap.pack();

mainMap.setVisible(**true**);

}

// Method which adds the points to the x and y values array, which is later used to print the points and create a polygon

**public** **static** **void** printdataset(ArrayList<Point2D> dataset)

{

*size* = dataset.size();

*xvalues* = **new** **int**[*size*];

*yvalues* = **new** **int**[*size*];

**for** (**int** i=0; i<dataset.size(); i++)

{

*xvalues*[i] = (**int**)dataset.get(i).getX();

*yvalues*[i] = (**int**)dataset.get(i).getY();

}

}

// The mainmethod where the program is run from

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

// Create an arraylist and read in the input values from the file

**long** startTime = System.*currentTimeMillis*();

ArrayList<Point2D> inputvalues = **new** ArrayList<Point2D>();

inputvalues = DataLoader.*loadTSPLib*("C:\\Users\\Calum\\workspace\\Copy of ADS\\src\\d1291"+ ".tsp");

// Obtain the amount of input results

**double** sizeofinputresults = inputvalues.size();

//Remove any duplicates from the list

Object[] st = inputvalues.toArray();

**for** (Object s : st) {

**if** (inputvalues.indexOf(s) != inputvalues.lastIndexOf(s))

{

inputvalues.remove(inputvalues.lastIndexOf(s));

}

}

// Get information on the inputvalues

**double** inputvalueslength = Routelengths.*routeLength*(inputvalues);

System.*out*.println("Inputvalues Default Length = " + inputvalueslength);

// Get information on Nearest Neighbour

**long** startTimeNearestNeighbour = System.*currentTimeMillis*();

ArrayList<Point2D> Nearestneighbouroutput = Nearestniegbour.*letour*(inputvalues);

**double** Nearesyneighbourlength = Routelengths.*routeLength*(Nearestneighbouroutput);

**long** endTimeNearestNeighbour = System.*currentTimeMillis*();

**long** totalTimeNearestNeighbour = endTimeNearestNeighbour - startTimeNearestNeighbour;

System.*out*.println("Nearest Neighbour Length = " + Nearesyneighbourlength + " and the time taken = " + totalTimeNearestNeighbour + "milliseconds");

*printdataset*(Nearestneighbouroutput);

// Get information on my Algorithm

**long** startTimeMyalgorith = System.*currentTimeMillis*();

ArrayList<Point2D> Myalgorithmoutput = Myalgorithm.*Mysolution*(Nearestneighbouroutput);

**double** MyalgorithmLength = Routelengths.*routeLength*(Myalgorithmoutput);

**long** endTimeMyalgorith = System.*currentTimeMillis*();

**long** totalTimeMyalgorith = endTimeMyalgorith - startTimeMyalgorith;

System.*out*.println("My Random Length = " + MyalgorithmLength + " and the time taken = " + totalTimeMyalgorith + "milliseconds");

//printdataset(Myalgorithmoutput);

// Get information on 2-opt Algorithm

**long** startTimeTwoout = System.*currentTimeMillis*();

ArrayList<Point2D> Twooutoutput = twoopt.*twooptalgorithm*(Myalgorithmoutput);

**double** Twooptlength = Routelengths.*routeLength*(Twooutoutput);

**long** endTimeTwoout = System.*currentTimeMillis*();

**long** totalTimeTwoout = (endTimeTwoout - startTimeTwoout);

System.*out*.println("Two-opt Length = " + Twooptlength + " and the time taken = " + totalTimeTwoout + "milliseconds");

//printdataset(Twooutoutput);

// Get the sizes of all the different Algorithms and print them to make sure none are missing

**double** sizeofNearestNeighbour = Nearestneighbouroutput.size();

**double** sizeofMyalgorithm = Myalgorithmoutput.size();

**double** sizeofTwoopt = Twooutoutput.size();

System.*out*.println("Input values size = " + sizeofinputresults + " Nearest Neighbour size = " + sizeofNearestNeighbour + " My Algorthim = " + sizeofMyalgorithm + " Two-opt = " + sizeofTwoopt);

// Run Polygon creation

SwingUtilities.*invokeLater*(**new** Runnable()

{

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

{

**new** Demo();

}

});

**long** endTime = System.*currentTimeMillis*();

**long** totalTime = (endTime - startTime);

System.*out*.println("Time taken to run the whole program = " + totalTime + "milliseconds");

}

}

**Class which loads data/dataloader**

import java.awt.geom.Point2D;

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

import java.util.ArrayList;

public class DataLoader {

public static ArrayList<Point2D> loadTSPLib(String fName){

//Load in a TSPLib instance. This example assumes that the Edge weight type

//is EUC\_2D.

//It will work for examples such as rl5915.tsp. Other files such as

//fri26.tsp .To use a different format, you will have to

//modify the this code

ArrayList<Point2D> result = new ArrayList<Point2D>();

BufferedReader br = null;

try {

String currentLine;

int dimension =0;//Hold the dimension of the problem

boolean readingNodes = false;

br = new BufferedReader(new FileReader(fName));

while ((currentLine = br.readLine()) != null) {

//Read the file until the end;

if (currentLine.contains("EOF")){

//EOF should be the last line

readingNodes = false;

//Finished reading nodes

if (result.size() != dimension){

//Check to see if the expected number of cities have been loaded

System.out.println("Error loading cities");

System.exit(-1);

}

}

if (readingNodes){

//If reading in the node data

String[] tokens = currentLine.split(" ");

//Split the line by spaces.

//tokens[0] is the city id and not needed in this example

float x = Float.parseFloat(tokens[1].trim());

float y = Float.parseFloat(tokens[2].trim());

//Use Java's built in Point2D type to hold a city

Point2D city = new Point2D.Float(x,y);

//Add this city into the arraylist

result.add(city);

}

if (currentLine.contains("DIMENSION")){

//Note the expected problem dimension (number ofcities)

String[] tokens = currentLine.split(":");

dimension = Integer.parseInt(tokens[1].trim());

}

if (currentLine.contains("NODE\_COORD\_SECTION")){

//Node data follows this line

readingNodes = true;

}

}

} catch (IOException e) {

e.printStackTrace();

} finally {

try {

if (br != null)br.close();

} catch (IOException ex) {

ex.printStackTrace();

}

}

return result;

}

}

**Draw Polygon Class**

import java.awt.geom.Point2D;

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

import java.util.ArrayList;

public class DataLoader {

public static ArrayList<Point2D> loadTSPLib(String fName){

//Load in a TSPLib instance. This example assumes that the Edge weight type

//is EUC\_2D.

//It will work for examples such as rl5915.tsp. Other files such as

//fri26.tsp .To use a different format, you will have to

//modify the this code

ArrayList<Point2D> result = new ArrayList<Point2D>();

BufferedReader br = null;

try {

String currentLine;

int dimension =0;//Hold the dimension of the problem

boolean readingNodes = false;

br = new BufferedReader(new FileReader(fName));

while ((currentLine = br.readLine()) != null) {

//Read the file until the end;

if (currentLine.contains("EOF")){

//EOF should be the last line

readingNodes = false;

//Finished reading nodes

if (result.size() != dimension){

//Check to see if the expected number of cities have been loaded

System.out.println("Error loading cities");

System.exit(-1);

}

}

if (readingNodes){

//If reading in the node data

String[] tokens = currentLine.split(" ");

//Split the line by spaces.

//tokens[0] is the city id and not needed in this example

float x = Float.parseFloat(tokens[1].trim());

float y = Float.parseFloat(tokens[2].trim());

//Use Java's built in Point2D type to hold a city

Point2D city = new Point2D.Float(x,y);

//Add this city into the arraylist

result.add(city);

}

if (currentLine.contains("DIMENSION")){

//Note the expected problem dimension (number ofcities)

String[] tokens = currentLine.split(":");

dimension = Integer.parseInt(tokens[1].trim());

}

if (currentLine.contains("NODE\_COORD\_SECTION")){

//Node data follows this line

readingNodes = true;

}

}

} catch (IOException e) {

e.printStackTrace();

} finally {

try {

if (br != null)br.close();

} catch (IOException ex) {

ex.printStackTrace();

}

}

return result;

}

}

**Nearest Neighbour Class**

import java.awt.geom.Point2D;

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

import java.util.ArrayList;

public class DataLoader {

public static ArrayList<Point2D> loadTSPLib(String fName){

//Load in a TSPLib instance. This example assumes that the Edge weight type

//is EUC\_2D.

//It will work for examples such as rl5915.tsp. Other files such as

//fri26.tsp .To use a different format, you will have to

//modify the this code

ArrayList<Point2D> result = new ArrayList<Point2D>();

BufferedReader br = null;

try {

String currentLine;

int dimension =0;//Hold the dimension of the problem

boolean readingNodes = false;

br = new BufferedReader(new FileReader(fName));

while ((currentLine = br.readLine()) != null) {

//Read the file until the end;

if (currentLine.contains("EOF")){

//EOF should be the last line

readingNodes = false;

//Finished reading nodes

if (result.size() != dimension){

//Check to see if the expected number of cities have been loaded

System.out.println("Error loading cities");

System.exit(-1);

}

}

if (readingNodes){

//If reading in the node data

String[] tokens = currentLine.split(" ");

//Split the line by spaces.

//tokens[0] is the city id and not needed in this example

float x = Float.parseFloat(tokens[1].trim());

float y = Float.parseFloat(tokens[2].trim());

//Use Java's built in Point2D type to hold a city

Point2D city = new Point2D.Float(x,y);

//Add this city into the arraylist

result.add(city);

}

if (currentLine.contains("DIMENSION")){

//Note the expected problem dimension (number ofcities)

String[] tokens = currentLine.split(":");

dimension = Integer.parseInt(tokens[1].trim());

}

if (currentLine.contains("NODE\_COORD\_SECTION")){

//Node data follows this line

readingNodes = true;

}

}

} catch (IOException e) {

e.printStackTrace();

} finally {

try {

if (br != null)br.close();

} catch (IOException ex) {

ex.printStackTrace();

}

}

return result;

}

}

**Five Point Switch/My Algorithm**

import java.awt.geom.Point2D;

import java.util.ArrayList;

public class Myalgorithm {

// This method gets the distance between the 5 points

public static double gettingthedistaance(ArrayList<Point2D> cities, int h){

// This method calculates the distance between points 1 and 2, 2 and 3, 3 and 4, 4 and 5 then adds them all together and returns the overall length

double thelength = 0.0, onetwolength = 0.0, twothreelength = 0.0, threefourlength =0.0, fourfivelength =0.0;

onetwolength = Point2D.distance(cities.get(h-2).getX(), cities.get(h-2).getY(), cities.get(h-1).getX(), cities.get(h-1).getY());

twothreelength = Point2D.distance(cities.get(h-1).getX(), cities.get(h-1).getY(), cities.get(h).getX(), cities.get(h).getY());

threefourlength = Point2D.distance(cities.get(h).getX(), cities.get(h).getY(), cities.get(h+1).getX(), cities.get(h+1).getY());

fourfivelength = Point2D.distance(cities.get(h+1).getX(), cities.get(h+1).getY(),cities.get(h+2).getX(), cities.get(h+2).getY());

thelength = onetwolength + twothreelength + threefourlength + fourfivelength;

return thelength;

}

// This method switches points h-2 and h-1 or A and B

public static ArrayList<Point2D> abswitch(ArrayList<Point2D> cities, int h)

{

Point2D temp;

temp = cities.get(h-2);

cities.set(h-2, cities.get(h-1));

cities.set(h-1, temp);

return cities;

}

// This method switches points h-2 and h or A and C

public static ArrayList<Point2D> acswitch(ArrayList<Point2D> cities, int h)

{

Point2D temp;

temp = cities.get(h-2);

cities.set(h-2, cities.get(h));

cities.set(h, temp);

return cities;

}

// This method switches points h-2 and h+1 or A and D

public static ArrayList<Point2D> adswitch(ArrayList<Point2D> cities, int h)

{

Point2D temp;

temp = cities.get(h-2);

cities.set(h-2, cities.get(h+1));

cities.set(h+1, temp);

return cities;

}

// This method switches points h-2 and h+1 or A and E

public static ArrayList<Point2D> aeswitch(ArrayList<Point2D> cities, int h)

{

Point2D temp;

temp = cities.get(h-2);

cities.set(h-2, cities.get(h+2));

cities.set(h+2, temp);

return cities;

}

// This method switches points h-1 and h or B and C

public static ArrayList<Point2D> bcswitch(ArrayList<Point2D> cities, int h)

{

Point2D temp;

temp = cities.get(h-1);

cities.set(h-1, cities.get(h));

cities.set(h, temp);

return cities;

}

// This method switches points h-1 and h+1 or B and D

public static ArrayList<Point2D> bdswitch(ArrayList<Point2D> cities, int h)

{

Point2D temp;

temp = cities.get(h-1);

cities.set(h-1, cities.get(h+1));

cities.set(h+1, temp);

return cities;

}

// This method switches points h-1 and h+2 or B and E

public static ArrayList<Point2D> beswitch(ArrayList<Point2D> cities, int h)

{

Point2D temp;

temp = cities.get(h-1);

cities.set(h-1, cities.get(h+2));

cities.set(h+2, temp);

return cities;

}

// This method switches points h and h+1 or C and D

public static ArrayList<Point2D> cdswitch(ArrayList<Point2D> cities, int h)

{

Point2D temp;

temp = cities.get(h);

cities.set(h, cities.get(h+1));

cities.set(h+1, temp);

return cities;

}

// This method switches points h and h+2 or C and E

public static ArrayList<Point2D> ceswitch(ArrayList<Point2D> cities, int h)

{

Point2D temp;

temp = cities.get(h);

cities.set(h, cities.get(h+2));

cities.set(h+2, temp);

return cities;

}

// This method switches points h+1 and h+2 or D and E

public static ArrayList<Point2D> deswitch(ArrayList<Point2D> cities, int h)

{

Point2D temp;

temp = cities.get(h+1);

cities.set(h+1, cities.get(h+2));

cities.set(h+2, temp);

return cities;

}

// This method finds switch between 5 points is the smallest, and then reapply the switch to the data set

public static void findwhichlengthisthesmallest(double smallestswitch, double abswitchlength, double acswitchlength, double adswitchlength, double aeswitchlength,

double bcswitchlength, double bdswitchlength, double beswitchlength, double ceswitchlength, double cdswitchlength, double deswitchlength, ArrayList<Point2D> cities, int h)

{

// If the smallest value out of all the switches is equal to the abswitch

if (smallestswitch == abswitchlength)

{

// Reapply the switch

abswitch(cities, h);

}

// This is then continued for ever possibility

if (smallestswitch == acswitchlength)

{

acswitch(cities, h);

}

if (smallestswitch == adswitchlength)

{

adswitch(cities, h);

}

if (smallestswitch == aeswitchlength)

{

aeswitch(cities, h);

}

if (smallestswitch == bcswitchlength)

{

bcswitch(cities, h);

}

if (smallestswitch == bdswitchlength)

{

bdswitch(cities, h);

}

if (smallestswitch == beswitchlength)

{

beswitch(cities, h);

}

if (smallestswitch == ceswitchlength)

{

ceswitch(cities, h);

}

if (smallestswitch == cdswitchlength)

{

cdswitch(cities, h);

}

if (smallestswitch == deswitchlength)

{

deswitch(cities, h);

}

}

// This method is the main method of the class and where the algorithm is executed

public static ArrayList<Point2D> Mysolution(ArrayList<Point2D> citieslist){

// The vreation of the cities arraylist which will hold the cities

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

// Make cities arraylis equal to the input data

cities = citieslist;

// Initialise variables

int loop = 0;

double startlength = 0.0, endlength = 0.0, originallength = 0.0;

// Get the start length of the overall route

startlength = Routelengths.routeLength(cities);

// The start of a do while loop which runs while the endlength is lower than the startlength

do

{

// Only make the startlength equal to the endlength after one loop or greater

if (loop > 0)

{

startlength = endlength;

}

// Create a for loop making h equal 2 as we need access to h-2 which is

for (int h =2; h < cities.size()-2; h++){

// Obtain the oringial length between the 5 cities

originallength = gettingthedistaance(cities, h);

// Carry out a switch of the 5 points - This case is an a b switch or h-2 and h-1

abswitch(cities, h);

// Get the new length between the 5 points

double abswitchlength = gettingthedistaance(cities, h);

// Get the new overall length of the whole data set with this switch applied

double abendlength = Routelengths.routeLength(cities);

// Undo the switch

abswitch(cities, h);

// Repeat for all other 9 possibilities

acswitch(cities, h);

double acswitchlength = gettingthedistaance(cities, h);

double acendlength = Routelengths.routeLength(cities);

acswitch(cities, h);

adswitch(cities, h);

double adswitchlength = gettingthedistaance(cities, h);

double adendlength = Routelengths.routeLength(cities);

adswitch(cities, h);

aeswitch(cities, h);

double aeswitchlength = gettingthedistaance(cities, h);

double aeendlength = Routelengths.routeLength(cities);

aeswitch(cities, h);

bcswitch(cities, h);

double bcswitchlength = gettingthedistaance(cities, h);

double bcendlength = Routelengths.routeLength(cities);

bcswitch(cities, h);

bdswitch(cities, h);

double bdswitchlength = gettingthedistaance(cities, h);

double bdendlength = Routelengths.routeLength(cities);

bdswitch(cities, h);

beswitch(cities, h);

double beswitchlength = gettingthedistaance(cities, h);

double beendlength = Routelengths.routeLength(cities);

beswitch(cities, h);

cdswitch(cities, h);

double cdswitchlength = gettingthedistaance(cities, h);

double cdendlength = Routelengths.routeLength(cities);

cdswitch(cities, h);

ceswitch(cities, h);

double ceswitchlength = gettingthedistaance(cities, h);

double ceendlength = Routelengths.routeLength(cities);

ceswitch(cities, h);

deswitch(cities, h);

double deswitchlength = gettingthedistaance(cities, h);

double deendlength = Routelengths.routeLength(cities);

deswitch(cities, h);

// Create a new array which stores the value of the switch combination lengths between 5 points

// Add in all the switched lengths to the array

double smallestpossibleswitch[] = new double [10];

smallestpossibleswitch[0] = abswitchlength;

smallestpossibleswitch[1] = acswitchlength;

smallestpossibleswitch[2] = adswitchlength;

smallestpossibleswitch[3] = aeswitchlength;

smallestpossibleswitch[4] = bcswitchlength;

smallestpossibleswitch[5] = bdswitchlength;

smallestpossibleswitch[6] = beswitchlength;

smallestpossibleswitch[7] = cdswitchlength;

smallestpossibleswitch[8] = ceswitchlength;

smallestpossibleswitch[9] = deswitchlength;

// Create a new array which stores the value of the overall length combinations of the switches between the whole data set

// Add all the lengths to the array

double smallestlength[] = new double [10];

smallestlength[0] = abendlength;

smallestlength[1] = acendlength;

smallestlength[2] = adendlength;

smallestlength[3] = aeendlength;

smallestlength[4] = bcendlength;

smallestlength[5] = bdendlength;

smallestlength[6] = beendlength;

smallestlength[7] = cdendlength;

smallestlength[8] = ceendlength;

smallestlength[9] = deendlength;

// Set smallestswitch to the first value of smallestpossibleswitch and set index to 0

double smallestswitch = smallestpossibleswitch[0];

int index = 0;

// Create a for loop which loops through to find the smallest possible switched

for (int i = 0; i < smallestpossibleswitch.length; i++)

{

// If the smallest possible switch is lower the the current smallest switch then

if (smallestpossibleswitch[i] < smallestswitch)

{

// Make smallest switch equal to the smallest possible switch

smallestswitch = smallestpossibleswitch[i];

// Importantly make i equal to index

index = i;

}

}

// Whatever the smallest switched length between 5 points is, use the index of that value to get the smallest overall length between all the points in the dataset

endlength = smallestlength[index];

// If the smallestlength between 5 points is less than the original length between 5 points AND if the starting length between all the points is more than the end length with the smallest switch between 5 points applied then

// Both these statements are need because even though the smallest length between 5 points is lower than the original length between 5 points, it can increase the overall distance

if(smallestswitch < originallength && endlength < startlength){

// Find the smallest switch between the 5 points and re apply the switch to the data set

findwhichlengthisthesmallest(smallestswitch, abswitchlength, acswitchlength, adswitchlength, aeswitchlength, bcswitchlength, bdswitchlength, beswitchlength, ceswitchlength, cdswitchlength, deswitchlength, cities, h);

}

}

// Get the overall endlength which is the distance between all the cities/points

endlength = Routelengths.routeLength(cities);

// Increment the loop

loop++;

// Print to help show working

//System.out.println(startlength + " - " + endlength + " so the overall distance decreases by " + (startlength - endlength));

// Loop while the endlength, which is the length of all of the points at the end of the algorithm, smaller than all of the points at the start of the algorithm

}while (endlength < startlength);

// Return the results once complete

return cities;

}

}

**2-Opt Algorithm**

import java.awt.geom.Point2D;

import java.util.ArrayList;

public class twoopt {

// The 2-opt Algorithm

public static ArrayList<Point2D> twooptalgorithm(ArrayList<Point2D> citieslist)

{

// Create an arraylist called results

ArrayList<Point2D> results = new ArrayList<Point2D>(citieslist.size());

// Get the size of the data

double size = citieslist.size();

// Set improve to zero

int improve = 0;

// Create a while loop and keep going until improve is less than a value - Depends on the datasize the value may want to be larger

// as more than n amount of loops may be required to give an accurate result

while (improve < 2)

{

// Get the best distance from the list of cities

double best\_distance = Routelengths.routeLength(citieslist);

// Create a for loop to loop through the data

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

{

// Create a nested for loop to loop through the data

for ( int k = i + 1; k < size; k++)

{

// Call the swap method, which changes the results arraylist

results = theswap(i, k, citieslist);

// Get the new distance of the alter results arraylist

double new\_distance = Routelengths.routeLength(results);

// If the altered new distance is less than the original best distance then

if (new\_distance < best\_distance)

{

// Set improve back to zero

improve = 0;

// Make the results equal to the citieslist

citieslist = results;

// Make the best original distance equal to the new distance

best\_distance = new\_distance;

}

}

}

// Increment improve for the while loop

improve++;

}

// Return citieslist once finished

return citieslist;

}

// Mehtod which carries out the swapping of points

public static ArrayList<Point2D> theswap(int i, int k, ArrayList<Point2D> citieslist)

{

// Create a arraylist called results

ArrayList<Point2D> results = new ArrayList<Point2D>(citieslist.size());

// Get the size of the citieslist input data

int size = citieslist.size();

// Take route[0] to route[i-1] and add them in order to new\_route

for (int c = 0; c <= i - 1; ++c)

{

results.add(c, citieslist.get(c));

//System.out.println(c);

}

// Take route[i] to route[k] and add them in reverse order to new\_route

int dec = 0;

for (int c = i; c <= k; ++c)

{

results.add(c, citieslist.get(k - dec));

dec++;

}

// Take route[k+1] to end and add them in order to new\_route

for ( int c = k + 1; c < size; ++c )

{

results.add(c, citieslist.get(c));

}

// Return the results once done

return results;

}

}

**Class which gets route length**

import java.awt.geom.Point2D;

import java.util.ArrayList;

public class Routelengths {

public static double routeLength(ArrayList<Point2D> cities){

//Calculate the length of a TSP route held in an ArrayList as a set of Points

double result = 0.0;//Holds the route length

Point2D prev = cities.get(cities.size()-1);

//Set the previous city to the last city in the ArrayList as we need to measure the length of the entire loop

for(Point2D city : cities){

//Go through each city in turn

result += city.distance(prev);

//get distance from the previous city

prev = city;

//current city will be the previous city next time

}

//Return results

return result;

}

}