Travelling Salesman Investigation

Zoe Wall
40182161@napier.ac.uk
Edinburgh Napier University - Algorithms and Data Structures (SET09117)

Keywords – Algorithms, Travelling Salesman, Nearest Neighbour, Two-Opt

1 Introduction

In the Travelling Salesman Problem, or 'TSP', a salesman is given a list of cities in which he has to travel between every one, once and only once, and loop back to the starting position. It is a very common problem that is used in researching optimisation techniques. The key is to finding a tour or route length that is the shortest distance between all of the points, but to find this most optimal solution would be to check every possible permutation. This method is called Brute Force.[1] However this method is just simply not feasible on even modest datasets as the total number of permutations to be checked can be calculated with equation 1,

$$\frac{(n-1)!}{2} \tag{1}$$

where n is the dimension of the problem. This means that for even just 10 cities, 181440 possible permutations are to be found. Yes, this will yield an exact solution to the problem, but could take an extraordinary amount of time. Without brute force as an option, the issue then becomes finding a balance between tour length and the time taken to find it. For a good solution, an optimal result should be found in a reasonable amount of time.

The TSP, is thought to be an NP problem, which means that it cannot be solved in polynomial time and therefore the complexity of any algorithm used to solve it would be exponential. [2] As the dimension of the problem increases, the time taken to solve the problem would increase exponentially. The two heuristics chosen to experiment with are Nearest Neighbour and the Two-Optimisation for it.

2 Method

An experiment was conducted into the performance of certain algorithms solving for different Travelling Salesman problem sets. For this experiment, Nearest Neighbour and an optimisation for it was implemented in c#.

2.1 Nearest Neighbour

The Nearest Neighbour algorithm is probably the most intuitive starting point when solving a TSP. The salesman starts at a random point and then visits the nearest city, they continue to visit the next nearest city from where they currently are until they reach the end. Once they have reached the final city, the salesman loops back to the starting point. However, this algorithm, sometimes referred to as "greedy" produces a non-optimal route, as some cities can be "forgotten" and left to expensive insertions into the route at the end, see figure 1.

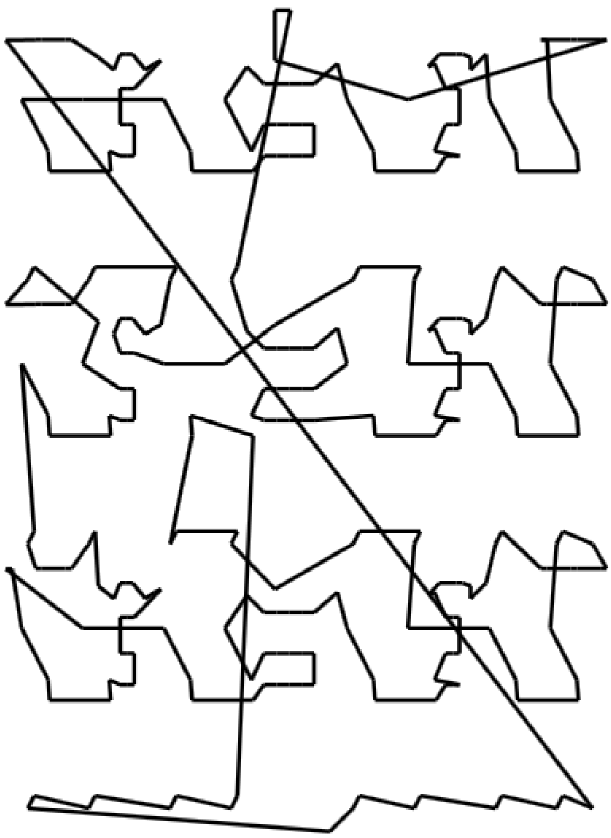

Figure 1: **Nearest Neighbour Route** - Image of route containing 318 cities calculated by Nearest Neighbour algorithm. Note cross over paths as some cities are left out yielding a suboptimal route.

Theoretically the complexity of this algorithm is $O(n^2)$. Which means that at it's worst case scenario, where the next closest point is found at the end of the iteration, it has to iterate through the dataset n*n times. Which means the time taken to run this algorithm will increase exponentially with the dimension of the problem. However, it is fairly consistent with it's results being sub-optimal and it's speed is relatively quick compared to others. [3]

2.2 Two-Opt

Starting from the Nearest Neighbour, a optimisation algorithm was implemented to improve the route by getting rid of the expensive cross-overs. It works by iteratively swapping two points until the optimal route is found, see algorithm 1.

```
while no improvement is made < 5times do
    best_distance = calculateDistance(existing_route);
    for i = 0 ;
    number of nodes to be swapped - 1 do
        for k = i+1 ;
        number of nodes to be swapped do
            new_route = 2-OptSwap();
            new_distance =
             calculateDistance(new_route);
            if new_distance < best_distance then
                existing_route = new_route;
                best_distance = new_distance;
                reset while loop
            end
        end
    end
end
```

Algorithm 1: Two-Opt Swap

Due to the iterative process of this particular algorithm, it is not efficient for larger data-sets. It first has to calculate the Nearest Neighbour route, and then for a worst case scenario it can take up to $O(n)$ to compute one swap. This can be optimised further, however the algorithm used in the experiment was simplified therefore the expected result from this algorithm will be quite costly for larger dimension problems. [4]

2.3 Tour

In this experiment the algorithms were run on several different problem sets. One of the main goals of the experiment was to investigate the run-times of each algorithm, so a range of dimensions were chosen. As the two algorithms implemented both have an exponential growth, they both begin to become inefficient at larger problem sets. Due to the nature of the Two-Opt algorithm, the data sets used were spread out from between small to reasonable large - around 1000 cities. Any larger, the Two-Opt algorithm would have taken too long to complete to comfortably repeat for this experiment.

2.4 Testing Process

In completing the experiment, the algorithms were run a number of times and the length of the tour created and time taken to calculate it was serialised to a .csv file. This meant that the experiment could be left to complete and the data could repeated easily and averaged. A project was also created alongside the experiment to visualise the data to see if there was any problems with the created tour, see figures 1 and 6 to see the results of this. Use of in-line debugging also helped to check that the tour was valid. To ensure the accuracy and repeatability of the results, all tests were run in the same sitting on a 2.60GHz i7-6700HQ CPU with no other programs running.

3 Results

Average run-times and lengths for a range of different problem sizes can be seen in figure 2. The lengths calculated by the tour of the algorithm was the same each time for the Nearest Neighbour and Two-Optimal tours which meant that the algorithms implemented were reliable as they always produced the same result for each specific data set.

	Nearest Neighbour		Two-Opt	
Dimension	Length (units)	Time (ms)	Length (units)	Time (ms)
52	8980.92	0.00	8114.35	24.00
159	54669.03	0.00	46254.18	831.20
200	35798.41	0.40	30514.96	2204.40
318	54033.58	1.40	45464.81	6512.60
400	19168.05	3.00	16393.57	12134.80
574	46881.87	6.00	40031.74	44130.00
783	11255.07	11.60	9619.33	119372.60
1002	315596.59	18.40	276051.47	260423.60
1432	188815.01	44.60	166349.17	662562.40

Figure 2: **Table of Results**- showing the calculated tour lengths and time taken to complete each algorithm for a specific data size (dimension).

Two-Opt The results show that the Two-Opt algorithm, although a lot slower consistently achieved a considerably better tour length than the Nearest Neighbour. On average it improved the tour length by 15.99%. However around the 800 city mark, the time taken on average to solve the problem was around 2 minutes, see figure 4. Any data set larger than this, the cost of the algorithm starts to become too high, compared to the Nearest Neighbour.

Nearest Neighbour Nearest Neighbour, albeit increasing with the data set, the run-time of this algorithm was very small compared to Two-Opt. For the smallest two data sets used, a time of 0 ms was recorded as it was extremely fast. A time was only registered after the dimension of the problem was greater than 200. It took the dimension to be over 1000 before the run-time was close to the run-time of the Two-Opt algorithm for the smallest dimension.

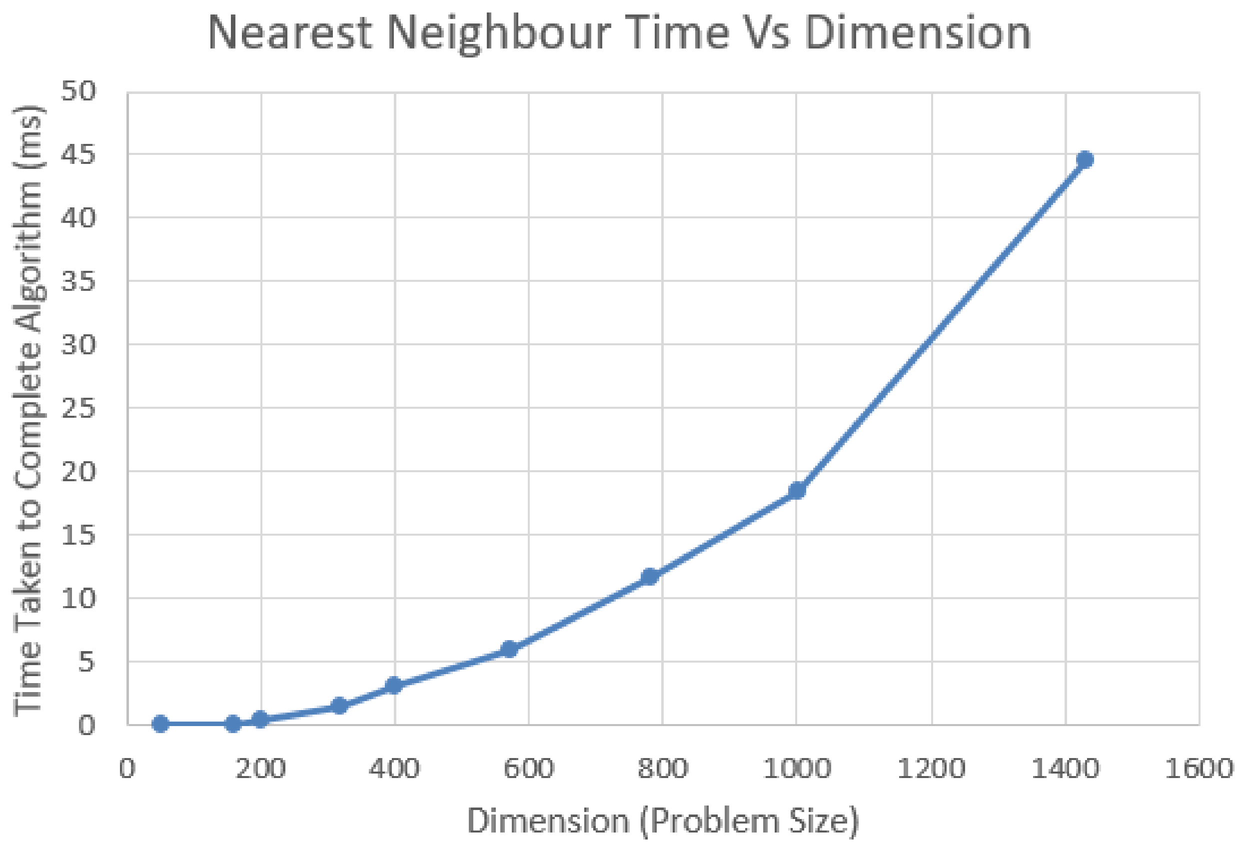

Figure 3: A graph to show the average run-time of the Nearest Neighbour algorithm against the problem size.

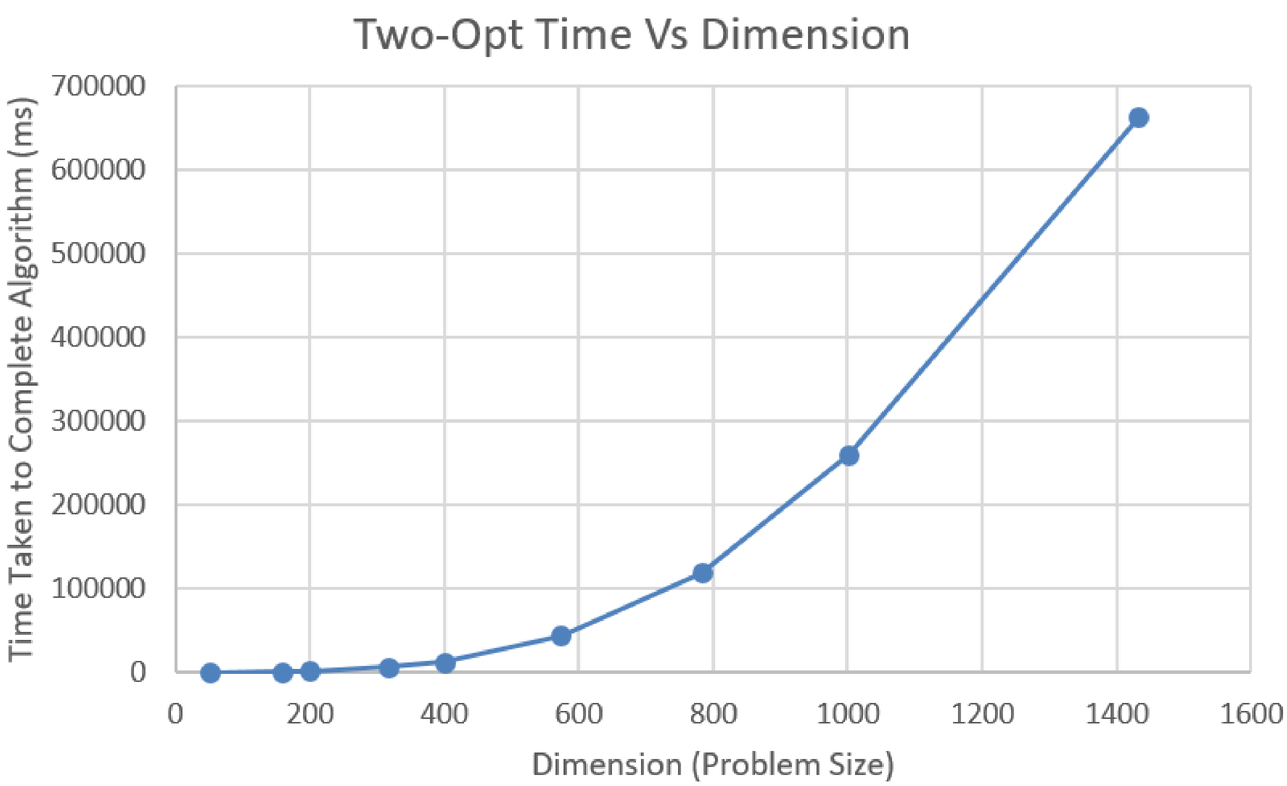

Figure 4: A graph to show the average run-time of the Two-Opt algorithm against the problem size.

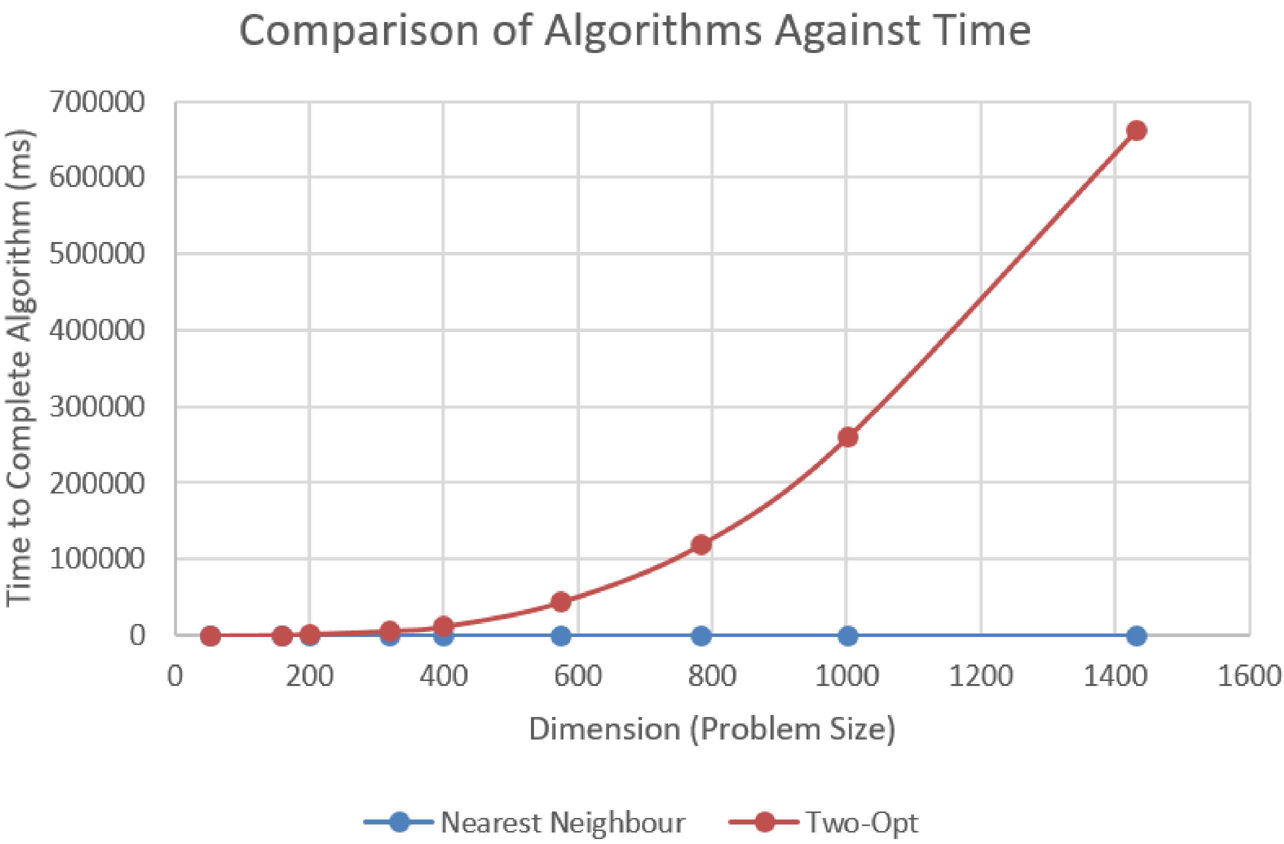

Figure 5: A graph to compare the average run-times of both algorithms against the problem size.

Validity To demonstrate that the solutions were valid several different checks were used. Firstly, a check to see if the new tour contained the correct number of cities. Then a check to see if there was any duplicates within the data was performed. This was implemented by attempting to add each city to a HashSet, each element within a HashSet must be unique so would return false if a duplicate value was added. A final check was also completed to see if every element within the original data set appeared somewhere within the tour. If all of these checks passed, the method returned true and it was printed to the console window.

Another way to check the algorithms were working correctly was to use the visualiser. By using a WPF canvas each point was added from the tour and lines were drawn between each city. This was a simple way to compare the results of the algorithms by eye. Figure 6 shows the Two-Optimal route found, with no paths crossing over.

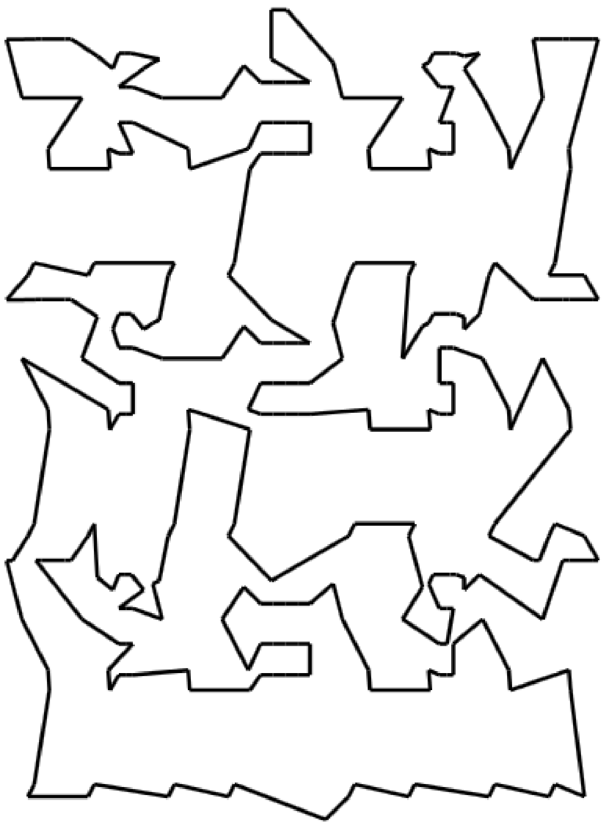

Figure 6: **Two-Opt Route**- Image of Two-Optimal Nearest Neighbour route from same dataset as figure 1.

Quality The quality of a solution to a TSP depends on both the route length and the cost of the algorithm. The results from figure 5 show a comparison between the costs of each algorithm. The Two-Optimal tour is shown to have a very high compared cost to the Nearest Neighbour tour at higher dimensions. The figure is slightly misleading as even for the dimensions smaller than around 400 the results are not in the same order of magnitude. For the data set of dimension 400, the Two-Opt took around 12 seconds to complete whilst the Nearest Neighbour's run-time was only 3ms. Meaning the quality of the Two-Opt algorithm is bad if the costs were compared in this way. However, it consistently yields a significantly better tour length than the Nearest Neighbour. The time taken to complete a data set of size 1000 is around 5 minutes long, which is probably the limit you would put on gaining a result. Therefore, providing the dimension of the TSP is less than 1000, the quality of the Two-Opt solution is good. Whereas for datasets larger than 1000, the Nearest Neighbour algorithm is of better quality, even though it returns a sub-optimal route.

4 Conclusions

Summary of Results On reflection, the Two-Opt algorithm consistently returned a shorter, and therefore better route length than that of the Nearest Neighbour, however the run-time for the algorithm to complete larger datasets was proven to be too long. This was the expected result as discussed. The results are reliable for two reasons; for each problem set, both algorithms produced the same permutation each time respectively, and therefore the same route length, and the experiments were run a number of times on the same machine so that the run-times could be averaged.

To conclude, the Nearest Neighbour is an algorithm that always provides a valid solution quickly. How valuable the Two-Optimisation algorithm is depends on the data set and the time allowed to experiment. In a real world situation where this problem needs to be solved only once, the Two-Optimal route is definitely favoured over the Nearest Neighbour algorithm.

Performance on Assessment The Nearest Neighbour algorithm that was implemented worked extremely well in consistently finding a valid solution to a travelling salesman problem set and the increase in run-time for larger data sets is not an issue at all. The problem lies with the Two-Opt algorithm which performs poorly due to the way it has been implemented. It was expected that the algorithm's run-time would increase exponentially, however the code can be re-factored to optimise it, for further research. For instance, the way the algorithm calculates the swap is by creating a new solution to the problem every iteration. Meaning the new distance is calculated on the whole list when only 2 paths have been switched. A further optimisation could be to only calculate the route length for the changed subsection of the tour to store the best one. Another expensive part of this algorithm is the fact it is run up to 5 times after a solution has been found. Meaning that for the worst-case scenario being that there is no more improvement to be made, this process can be very costly. An interesting way to improve this algorithm further would be to also store data of the nearest avalible points, thereby cutting the cost of having to iterate through every city to find the best switch. [4]

Both algorithms produce valid solutions to the travelling salesman problem, resulting in a similar overall quality. The Nearest Neighbour is fast but the route is not optimised, whereas the Two-Opt implementation is slow but gives a consistently good result. This assessment was not to find the shortest route length for the least cost but to investigate the effects of the size of the data to the run-time of the algorithms, which was completed successfully.

References

[1] J. Malkevitch, "Sales and chips," *Accessed: October 2016*. www.ams.org.

[2] M. Freiberger, "The travelling salesman," *Accessed: November 2016*. www.plus.maths.org.

[3] D. Johnson and L. McGeoch, "The travelling salesman problem: A case study in local optimization," pp. 7–8, 1995.

[4] C. Nilsson, "Heuristics for the travelling salesman problem," pp. 1–3, 2003.

5 Appendix

Listing 1: TSPInstance script containing loading and algorithms

```
ïż£using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Drawing;
using System.IO;
using System.Text.RegularExpressions;
using System.Diagnostics;

namespace TravellingSalesman
{
  // TSP Instance class contains methods for reading in, and creating tours for a specific tsp problem set
  class TSPInstance
  {
    private string filename;               // store filename of dataset
    public List<PointF> originalCitiesData;    // list to store the original cities read from the data
    private int dimension;                 // dimension stores problem size

    // Constructor, takes in file name, adds path to resource folder, stores a reference to it, and runs the file Loader
    public TSPInstance(String fn)
    {
      // relative path for resource folder
      string path = "..\\..\\Resources\\";
      filename = path + fn + ".tsp";

      LoadTSPLib();
    }

    // Load reads from the given file. Checks for errors, parses data. Returns list of points (cities to visit on tour) and size of the problem
    public void LoadTSPLib()
    {
      List<PointF> result = new List<PointF>(); // for storing result

      StreamReader reader;

      try
      {
        // create instance of stream reader to read from a file
        reader = new StreamReader(filename);

        bool readingNodes = false; // flag to check for End of Field
        dimension = 0;          // dimension is number of points within problem

        // using closes stream when complete
        using (reader)
        {
          string line;
          // while more lines to read, print out
          while ((line = reader.ReadLine()) != null)
          {
            // Read file until end of field
            if (line.Contains("EOF"))
            {
              // set finished flag and check if dimension is correct
              readingNodes = false;

              if (result.Count != dimension)
              {
                // close app if dimension isn't correct
                Console.WriteLine("Error loading cities");
                Environment.Exit(-1);
              }
            }

            // parse nodes
            if (readingNodes)
            {
              // get rid of spaces at start of line
              line = line.TrimStart();

              // split at any number of spaces (1 or more)
              string[] tokens = Regex.Split(line, @"\s+").ToArray();

              // trim any space from values
              tokens[1].Trim();
              tokens[2].Trim();

              // token[0] is city ID and can be ignored.

              // token[1] is x coord, 2 is y coordinate of city
              float x = float.Parse(tokens[1]);
              float y = float.Parse(tokens[2]);

              // create a new point and add to list of cities
              PointF city = new PointF(x, y);
              result.Add(city);
            }

            // read dimension
            if (line.Contains("DIMENSION"))
            {
              // save expected problem ( number of cities)
              String[] tokens = line.Split(':');
              dimension = Int32.Parse(tokens[1].Trim());
            }

            // find node data
            if (line.Contains("NODE_COORD_SECTION"))
              readingNodes = true;

          }
        }
      }
      catch (Exception e) // catch all exceptions, and print message.
      {
        Console.WriteLine("Error reading file: " + e.Message);
      }

      // store the result
      originalCitiesData = result;

    }

    //Nearest Neighbour alg from pseudocode
    public List<PointF> NearestNeighbour(List<PointF> citiesIn)
    {
      // deep copy of given list
      List<PointF> cities = new List<PointF>(citiesIn);

      // Create new empty list to store re-ordered tour
      List<PointF> newTour = new List<PointF>();

      // reference to closest city
      PointF closestCity = new PointF();

      // get first city as staring point and remove from list as its been used
      PointF current = cities.ElementAt(0);
      cities.RemoveAt(0);

      double closestDistance;

      while (cities.Count > 0)
      {
        newTour.Add(current);   // add current city

        closestDistance = double.PositiveInfinity;

        // find closest city to current
        foreach (PointF possCity in cities)
        {

          // calculate distance between points
          double pointDistance = Distance(current, possCity);

          // if distance is closer, update vars
          if (pointDistance < closestDistance)
          {
```

```
                closestCity = possCity;
                closestDistance = pointDistance;
            }
        }

        // remove closest city from the list, add to tour, and set as current to loop and find closest to that
        cities.Remove(closestCity);
        current = closestCity;

    }

    // add final city to tour
    newTour.Add(current);


    return newTour;
}

// TwoOpt Algorithm: From a starting permutation, swap cities, if better, keep result
public List<PointF> TwoOpt(List<PointF> citiesIn)
{
    // deep copy of list to store result (if no swaps can improve, this is result)
    List<PointF> result = new List<PointF>(citiesIn);

    int improvement = 0;

    // stop running algorithm after 5 times with no improvement
    while (improvement < 5)
    {
        // calculate distance of current tour.
        double bestDistance = CalculateLength(result);

        // for every city in the list
        for (int i = 0; i < dimension -1; ++i)
        {
            // for every possible other city in the list, swap the values and calc new length
            for (int k = i + 1; k < dimension; ++k)
            {
                // this method creates a new permutation by swapping elements at i and k
                List<PointF> newTour = Swap(result, i, k);

                double new_distance = CalculateLength(newTour);

                // if new length of tour is an improvement, reset the counter and save new tour as best
                if (new_distance < bestDistance)
                {
                    improvement = 0;
                    result = newTour;
                    bestDistance = new_distance;

                }
            }
        }

        improvement++;     // increase improvement counter, reset at 0 if improvement has been found
    }

    // return best list
    return result;
}

// this method returns a new permutation of the list with swapped values
public List<PointF> Swap(List<PointF> tour, int i, int k)
{
    // create a new blank tour
    List<PointF> result = new List<PointF>();

    // for the first part of route add in order, tour[0] to tour[i -1]
    for (int c =0; c <= i -1; ++c)
    {
        result.Add(tour[c]);
    }

    // for when city = i, until c = k, add them in reverse order
    int count = 0;
    for (int c = i; c <= k; ++c)
    {
        result.Add(tour[k - count]);
        count++;
    }

    // for k+1 onwards, add in order to end of tour
    for (int c = k + 1; c < dimension; ++c)
    {
        result.Add(tour[c]);
    }

    // return new list
    return result;
}

// Calculate length of tour
public double CalculateLength(List<PointF> cities)
{
    double result = 0;

    // set previous city to last city in the list to measure the length of entire loop
    PointF previousCity = cities.ElementAt(cities.Count - 1);

    foreach(PointF city in cities)
    {
        // go through each city in turn summing length between neighbouring points
        result += Distance(city, previousCity);
        previousCity = city;
    }

    return result;
}

// calculate distance between two points
private double Distance(PointF p1, PointF p2)
{
    // method to calculate distance between two points

    double result = 0;

    // pythag
    PointF difference = new PointF(p1.X - p2.X, p1.Y - p2.Y);

    result = Math.Sqrt(difference.X * difference.X + difference.Y * difference.Y);

    return result;
}

// check if correct
public bool Correct(List<PointF> toCheck)
{
    // compare sizes. If wrong don't calculate anything
    if (toCheck.Count != originalCitiesData.Count)
        return false;

    foreach (PointF p in originalCitiesData)
    {
        // foreach original city, check if it is within the new permutation
        if (!toCheck.Contains(p))
            return false;
    }

    // create new hashSet to check for duplicates. Add each point into set and if it can't then it is a duplicate
    HashSet<PointF> hashSet = new HashSet<PointF>();

    for (int i = 0; i < toCheck.Count; ++i)
    {
        if (!hashSet.Add(toCheck[i]))
            return false;
    }


    // all checks passed return true
    return true;

}
}
}
```

Listing 2: Script to run Solver

```
ïż£using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Drawing;
using System.Diagnostics;
using System.Windows;
using System.IO;

namespace TravellingSalesman
{

  // Execution of the program is handled in this class
  class Program
  {

    static StreamWriter writer;  // declaration of streamwriter to write data to a csv file
    static string delim = ",";   // delimiter for csv

    static void Main(string[] args)
    {
      // file name for data set
      string fn = "berlin52";

      // initialise TSP instance and load file
      TSPInstance berlin = new TSPInstance(fn);

      // Initialise CSV file. Create and open streamwriter for writing, and create table headings
      InitialiseCSV(fn);

      // Loop for running tests n times
      for (int i = 0; i < 5; ++i)
      {
        RunNearestNeighbour(berlin);
        RunTwoOpt(berlin);
        writer.WriteLine();
      }

      // close writer connection to file and dispose of it
      writer.Close();
      writer.Dispose();

      // stop console window from closing
      //Console.ReadLine();
    }

    // Method to run, time and print results from nearest neighbour test
    public static void RunNearestNeighbour(TSPInstance test)
    {
      // Start timer
      Stopwatch stopwatch = new Stopwatch();
      stopwatch.Start();

      // create new tour from original read-in data, using nearest neighbour algorithm
      List<PointF> nn = test.NearestNeighbour(test.originalCitiesData);

      // Stop timer
      stopwatch.Stop();
      long elaspedTime = stopwatch.ElapsedMilliseconds;

      // print results
      // calculate total length of tour
      // check if solution is correct (no duplicates/dimensions are correct/everything exists in the list)
      PrintResult(elaspedTime, test.CalculateLength(nn), test.Correct(nn));
    }

    // Method to run, time and print results from TwoOpt test
    public static void RunTwoOpt(TSPInstance test)
    {
      // start stopwatch
      Stopwatch stopwatch = new Stopwatch();
      stopwatch.Start();

      // create new tour from NearestNeighbour.
      List<PointF> twoOpt = test.TwoOpt(test.NearestNeighbour(test.originalCitiesData));

      // Stop timer
      stopwatch.Stop();
      long elaspedTime = stopwatch.ElapsedMilliseconds;

      // Print results
      PrintResult(elaspedTime, test.CalculateLength(twoOpt), test.Correct(twoOpt));

    }

    // Method to print results in same format and add to file
    public static void PrintResult(long time, double length, bool correct)
    {
      // print results to console
      Console.WriteLine("Time taken = " + time + "ms");
      Console.WriteLine("Length of tour = " + length);
      Console.WriteLine("Is valid solution: " + correct + "\n");

      // write to file, length and time within table, separated by commas
      writer.Write(length + delim + time + delim);
    }

    // Method to create csv file to store data, and create table headings.
    public static void InitialiseCSV(string fn)
    {
      try
      {
        // create new Streamwriter connection to new file
        writer = new StreamWriter("..\\..\\Solutions\\DataSet-"+ fn +"TEST.csv");

        // write table headings in file
        writer.Write("NN Length" + delim);

        writer.Write("NN Time (ms)" + delim);

        writer.Write("Two-Opt Length" + delim);

        writer.Write("Two-Opt Time (ms)");

        // new line
        writer.WriteLine();
      }
      catch (Exception e)
      {
        Console.WriteLine("Problem in writing to file: " + e);
      }
    }

  }
}
```

Listing 3: Script to draw window for visualisation.

```
ïż£using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using System.Drawing;

namespace Visualisation
{
  // enum used for radiobutton selection of problems loaded
  enum tour { berlin, lin };

  /// <summary>
  /// Interaction logic for MainWindow.xaml
  /// </summary>
  public partial class MainWindow : Window
  {
    private int multiplier = 5; // scale for points drawn to canvas

```

```
    // two structs to store data from algorithms for faster visualisation
    private Data near;
    private Data twoOpt;

    // instance of solver
    private TSPSolver solver;

    // constructor of main window to intialse and load problems
    public MainWindow()
    {
        // initialise main window
        InitializeComponent();

        // initialise solver
        solver = new TSPSolver();
    }

    // event handler for nearestneighbour btn click, displays stored results and draws graph
    private void nnBtn_Click(object sender, RoutedEventArgs e)
    {
        typeLbl.Content = "Nearest\nNeighbour"; // change title to nn
        UpdateResults(near);                 // update graph
    }

    // event handler for twoopt click, displays new results
    private void twoOptBtn_Click(object sender, RoutedEventArgs e)
    {
        typeLbl.Content = "Two Opt";    // change title
        UpdateResults(twoOpt);        // update gui
    }

    // method to update gui with results. Displays graph and time/length
    private void UpdateResults(Data results)
    {
        // update labels
        timeLbl.Content = "Time: " + results.time + "ms";
        lengthLbl.Content = "Length: " + results.length;


        mCanvas.Children.Clear();   // ensure canvas is clear before drawing

        // for every city(point) stored in the tour list, create a line and add it to the canvas as a child
        for (int i = 0; i < results.tour.Count; ++i)
        {
            Line l = new Line();

            // ensure visible
            l.Visibility = System.Windows.Visibility.Visible;
            l.StrokeThickness = 2;
            l.Stroke = System.Windows.Media.Brushes.Black;

            // first point of line
            l.X1 = results.tour[i].X / multiplier;
            l.Y1 = results.tour[i].Y / multiplier;

            // second point of line (if not last point, draw line between next point)
            if (i < results.tour.Count - 1)
            {
                l.X2 = results.tour[i + 1].X / multiplier;
                l.Y2 = results.tour[i + 1].Y / multiplier;
            }
            else // else if last point, draw between that and starting point in list
            {
                l.X2 = results.tour[0].X / multiplier;
                l.Y2 = results.tour[0].Y / multiplier;
            }

            // add line to canvas
            mCanvas.Children.Add(l);
        }
    }

    // event handler for rb, changes choice of data set and runs solutions, shows nearest neighbour first by default
    private void berlinRBtn_Checked(object sender, RoutedEventArgs e)
    {
        // calculate selected tour nn and twoOpt
        solver.Selected(tour.berlin);
        near = solver.NN();
        twoOpt = solver.TwoOpt();

        // change line multiplier to fit lines to canvas
        multiplier = 5;

        // show nearest neighbour results first
        nnBtn_Click(this, new RoutedEventArgs());
    }

    // event handler for rb lin, changes choice of data set and runs solutions, shows nearest neighbour first by default
    private void linRBtn_Checked(object sender, RoutedEventArgs e)
    {
        // calculate selected tour nn and twoOpt
        solver.Selected(tour.lin);
        near = solver.NN();
        twoOpt = solver.TwoOpt();


        // change line multiplier to fit lines to canvas
        multiplier = 10;

        // show nearest neighbour results first
        nnBtn_Click(this, new RoutedEventArgs());
    }
  }
}
```

Listing 4: Script to store TSPInstances

```
ïż£using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Drawing;
using System.Diagnostics;
using System.Windows;

namespace Visualisation
{
  // struct to store data/results from algorithms
  struct Data
  {
      public long time;          // time taken for algorithm to complete
      public double length;      // length of completed tour
      public List<PointF> tour;   // tour (order of cities to visit)
  }

  class TSPSolver
  {
      // vars to store instances of different tsp routes
      private TSPInstance berlin;
      private TSPInstance lin;
      private TSPInstance selected;

      // constructor to load two tsp instances and store cities in list
      public TSPSolver()
      {
          // load instances of tsp for two routes
          berlin = new TSPInstance("berlin52");
          lin = new TSPInstance("lin318");
      }

      // setter for selected tour from gui radiobtn
      public void Selected(tour value)
      {
          // selected reference tsp instance to perform calulations on
          if (value == tour.berlin)
              selected = berlin;
          else
              selected = lin;
      }

      // performs then returns nearest neighbour results
      public Data NN()
      {
```

```
            // struct to store data from alg to use in GUI
            Data nearest = new Data();

            // time and complete nearest neighbour alg
            Stopwatch watch = new Stopwatch();
            watch.Start();
            nearest.tour = selected.NearestNeighbour(selected.originalCitiesData);
            watch.Stop();

            // store time and length
            nearest.time = watch.ElapsedMilliseconds;
            nearest.length = selected.CalculateLength(nearest.tour);

            return nearest;
        }

        // performs then returns two opt results
        public Data TwoOpt()
        {
            // struct to store results from twoopt
            Data twoOpt = new Data();

            // start timer, perform nn then two opt from nn
            Stopwatch watch = new Stopwatch();
            watch.Start();
            List<PointF> near = selected.NearestNeighbour(selected.originalCitiesData);
            twoOpt.tour = selected.TwoOpt(near);
            watch.Stop();

            // store time and length
            twoOpt.time = watch.ElapsedMilliseconds;
            twoOpt.length = selected.CalculateLength(twoOpt.tour);

            return twoOpt;
        }

    }
}
```