Unit 16 Exercises – The Travelling Salesman Problem

Aims and objectives

- Further practice in planning and program development
- Understanding and use of algorithms

Assignment

In this exercise you will implement a program that attempts to solve an instance of the *Travelling Salesman Problem* (TSP). Your program will read an instance of the TSP from a file, construct an attempted solution (a *tour*), and display it on the screen.

The travelling salesman problem (TSP)

In the TSP, we are given a set of *cities*, which we will think of as points with coordinates. The problem is to start at a particular point, visit all other points once and only once, returning finally to the starting point, and in doing so to travel the *minimum possible total distance*.

The TSP is an infamously difficult computational challenge, but it is also an important practical problem. The following are some obvious examples: an engineer has to visit a number of customers and return to base; rubbish has to be picked up and then dumped; a robot on a factory floor has to deliver parts to machines and then return to the store room. In all such cases, a shorter round trip saves time and money.

In principle, this problem can be solved by trying out all of the possible *tours* – i.e., all of the possible orders in which the points might be visited – noting the length of each, and choosing the shortest. For example, assume that we have 4 points A, B, C and D, and that we start at point A. We might examine all possible tours ABCD, ABDC, ACBD, ACDB, ADBC, ADCB. This is an example of a *brute force* algorithm, or *exhaustive search*. If we have *n* points, with a fixed starting point, the number of possible tours is given by the formula *(n-1)(n-2)...3.2.1* – the so-called *factorial* function, *(n-1)!*. The table below gives some values of this function, and shows how long it would take a computer program to check all possible tours in each case, assuming that 10^9 (a billion) tours could be checked each second.

n	**(n-1)!**	**Time required**
10	362880	0.0004 seconds
15	8.7×10^{10}	1.5 minutes
20	1.2×10^{17}	4 years
25	6.1×10^{23}	20 million years
30	8.8×10^{30}	2.8×10^{14} years

Clearly, for anything but a small number of points, exhaustive search is infeasible. Therefore, we are usually forced to relax our requirements, and rather than demand an optimal tour, we accept a sub-optimal tour that can be computed in a reasonable amount of time. One approach that may seem reasonable is known as the *Nearest Neighbour Algorithm*, described below. It is easy to implement, but unfortunately it often finds a tour that is not very close to optimal. Your exercise is to implement the Nearest Neighbour Algorithm.

The Nearest Neighbour Algorithm

The Nearest Neighbour Algorithm constructs a particular tour. A starting point is selected, and we call this the *current point*. We then select, as the next point in the tour, the point that is closest to the *current point*. We move to that point and repeat the process, selecting next the unvisited point that is closest to our new *current point*. The process terminates when we have visited all of the points, and we then have to travel directly back to our starting point. Typically, the Nearest Neighbour Algorithm constructs a tour that starts off with short steps, but ultimately has a few long steps to pick up outlying points, and then has one very long step back from the last point visited to the starting point.

Let *Tour* be the sequence in which the points will be visited. Initially, *Tour* contains a chosen starting point. The Nearest Neighbour Algorithm might be described as follows.

> **while** at least one point remains unvisited:
> let *CurrentPoint* be the most recent point in *Tour*
> find *NextPoint*, the nearest unvisited point to *CurrentPoint*
> append *NextPoint* to *Tour*

There is a clever way to implement this by storing the sequence in a list *Cities*, and rather than having a separate list *Tour*, merely rearranging *Cities*, as follows:

1. Find the index `j` of the city in `Cities`, in position at least 1, which is nearest to `Cities[0]`; swap `Cities[1]` and `Cities[j]`
2. Find the index `j` of the city in `Cities`, in position at least 2, which is nearest to `Cities[1]`; swap `Cities[2]` and `Cities[j]`
3. Find the index `j` of the city in `Cities`, in position at least 3, which is nearest to `Cities[2]`; swap `Cities[3]` and `Cities[j]`

   `...`

N-2. Find the index `j` of the city in `Cities`, in position at least N-2, which is nearest to `Cities[N-3]`; swap `Cities[N-2]` and `Cities[j]`

This approach is somewhat similar to the second version of Selection Sort (Unit 15 Lecture). After step `I`, positions `0 ... I` of `Cities` contain the "solved" part of the list – i.e. the partial tour constructed so far.

The requirement

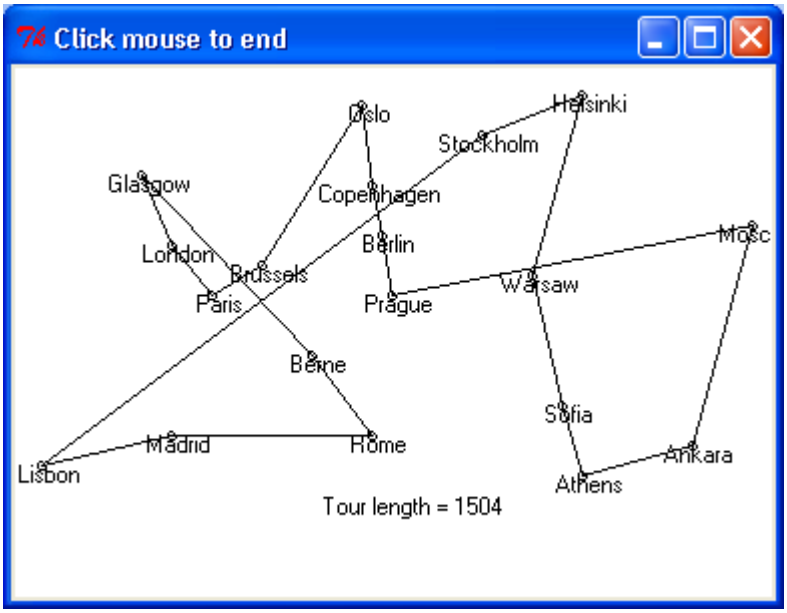
The program is to read data from a text file (Cities.txt). Each line of the file holds data about one city, namely the X and Y coordinates of that city followed by the city's name. Successive items are separated by a single space. For example:

```
436 331 Athens
528 198 Moscow
190 170 Glasgow
```

The coordinates are compatible with the drawing conventions used by the `Canvas` module: X coordinates increase from West to East, and Y coordinates increase from North to South.

The program is to read in the city data, construct a tour using the Nearest Neighbour Algorithm, and draw a picture of this tour using the `Canvas` module, reporting the length of the tour drawn (expressed as an integer, in units corresponding to the coordinate values).

The screenshot on the next page shows the kind of output that is required.



Task 1 – Understanding the algorithm; choice of data structures

Make sure you understand very clearly how the Nearest Neighbour Algorithm works. The description of the algorithm makes it clear that you should use a list of cities, but what data structure will you use to represent the data about each individual city?

Task 2 – Developing a plan

Write a top-level plan for this problem, and refinements for each major step. Think about how you will break up the program into appropriate functions. Type your plan into the file `Plan.txt`, which will form part of your solution (tutors will ask to see it).

Task 3 – Implementing and testing the program

Working from your plan, implement the program and test it. As well as testing your program with the given file of cities, you might find it useful to create your own test files in which it is obvious what the tour should be. Break this task down into the following steps:

Task 3a – reading data and drawing a tour

Start by reading the data from the file and drawing a tour in which the cities are visited in exactly the order that they occur in the file.

Task 3b – implementing the Nearest Neighbour Algorithm

Complete the implementation of the algorithm.

Task 4 (optional) – varying the starting point

The Nearest Neighbour Algorithm will give tours of different lengths for different starting points. Instead of just starting from the first city in the file, calculate the length of the Nearest Neighbour tour for each starting point in turn, and display the shortest tour that you find.