Laboratory 15 – Week 23

## The Travelling Salesman Problem

## 15.1 Introduction

**Firstly, this worksheet is one of the worksheets from which your laboratory worksheets portfolio of work will be assessed.**

**This worksheet is about empirically evaluating how scalable a number of single population heuristic search methods are at solving the Travelling Salesman Problem (TSP) when applied to a number of different sized problems**.

**The aim is to evaluate and demonstrate how well each of the methods performs as the size of the problem increases.**

The overall objective of this worksheet is to produce; present and report on, a Java program that is capable of solutions the TSP on a number of different sized problems using a number of different heuristic search algorithms (see below).

## 15.2 Preliminaries

Make sure that you are familiar with the material in lecture “16.1 The Travelling Salesman Problem”.

The datasets for this project are available in Appendix A in the file:

**CS2004 TSP Data (2016-2017).zip**

Within this file are a large number of TSP data files of the form TSP\_<N>.txt where *N* is the size of the data file. Within each TSP\_<N>.txt file is an *N* by *N* matrix of distances between the nodes, which is a symmetric matrix. I.e. the distance between nodes *i* and *j* is the same as the distance between node *j* and *i*.

Appendix B contains some useful information regarding the TSP problem.

## 15.3 Exercise 1: Solving the TSP Problem

The Travelling Salesman problem (TSP) has been described fully in the lectures. The purpose of this worksheet is to:

1) Implement a number of the algorithms (listed below) to solve the TSP

2) Compare the algorithms on a number of different sized datasets

3) Report on the accuracy of the methods as the problem size changes

You should try and implement more than one out of the four following methods:

1) Simple Hill Climbing (Random Mutation Hill Climbing)

2) Stochastic Hill Climber

3) Random Restart Hill Climber

4) Simulated Annealing

**Each method that is implemented should be run a number of times to remove any one off results due to the stochastic nature of the algorithms**. The exact number of times is up to the student, but **a sensible and logical number of repeat experiments should be chosen**. There are a very large number of datasets available as described in Appendix A.

Running the methods on all of these datasets for a number of repeats will take a very, very long time.

**A sample of these datasets should be chosen to represent a well thought out spread of the various sizes of the problems being solved.**

Note that each of the methods can be run for a number of iterations. **All of the experiments should be run for the same number of iterations, so that they can be compared.** **Note however, that this parameter is up to the student to choose and a sensible and logical value should be chosen.**

You should provide some form of indication of quality of your solution. **This can be done a number of ways, one suggestion is to use the *Minimum Spanning Tree* as discussed in the lectures and another is to use the optimal solution files**. These latter files are a list of integers detailing the optimal tour for a given dataset. See the *zip* file in the appendix for details.

## 15.4 Appendix A - Data Files

The following *zip* file contains the code and data you will need for this exercise sheet.

|  |
| --- |
|  |

## 15.5 Appendix B - TSP Fitness and Pseudo Code

For *N* cities, let *d*(*i*,*j*) (>0 when *i*≠*j* and =0 when *i*=*j*) be the distance between two cities numbered *i* and *j* where *d*(*i*,*j*) = *d*(*j*,*i*) (e.g. the distance between city 1 and 2 is the same as the distance between 2 and 1).

Note also that *d*(*i*,*i*) = 0, i.e. the distance between a city and itself is zero.

Let the list *T* = {*t*1,*t*2,...,*t*n} be a ***tour*** where we visit *t*1 first, then *t*2, through to *tn* in the order they appear in *T*, and then we return to *t*1. E.g. *T* = {3,1,2} means that we start at city 3, go to city 1, then to city 2 and then return to city 3.

Note that no two values in *T* are the same (we can only visit each city once) and that a tour must visit each city, i.e. *T* contains each city.

Let the length of a tour be defined as *f*(*T*) which is the sum of the distance travelled when we ***do***the tour and we note that we wish to minimise *f*(*T*), i.e. ***we want to find the shortest tour***.



If *T* = {3,1,2} as in the example above then



Or in pseudo code:

|  |  |
| --- | --- |
| **TSP Fitness Function (f)** | |
| Input: | NThe number of cities to visit  T A tour (list of integers of size N)  D An N by N matrix containing each d(i,j) |
| 1) | Let s = 0 |
| 2) | For i = 1 to (N-1) |
| 3) | Let a = ti |
| 4) | Let b = ti+1 |
| 5) | Let s = s + d(a,b) |
| 6) | End For |
| 7) | Let end\_city = tn |
| 8) | Let start\_city = t1 |
| 9) | Let s = s + d(end\_city,start\_city) |
| Output: | The tour length s |

**Note: remember that arrays and ArrayLists are zero indexed in Java!**