# Informatics 2 – Introduction to Algorithms and Data Structures

Coursework 3: Heuristics for the Travelling Salesman Problem

*This document is the specification for Coursework 3 of Inf2-IADS. It is being released Friday 6th March (Friday of week 7), and your submissions will be due at 4pm on Monday 23th March 2020 (Monday of week 10). This is a* summative *coursework, and your grade for this coursework will contribute 10% of your overall mark for Inf2-IADS.*

*Your implementation for this coursework should be developed and written on an individual basis. It is fine to discuss the task with classmates, and to help each other understand what need to be done. However you should not copy code from classmates or anywhere else, and you should not give or take key details for the implementation. If you discover literature which gives you ideas for (the final part of) the coursework, please make sure you* credit these sources *in your report.*

## 1 The travelling salesman problem

In this coursework we are going to explore the *Travelling Salesman Problem* (TSP). In this problem, we are given an undirected graph on $n$ nodes where the underlying graph structure is assumed to be complete [1] together with a weight function $w : E \to \mathbb{R}^+$. The nodes of the graph represent a hypothetical collection of cities that a salesman (or saleswoman!) is required to visit, and the weight $w(u, v)$ represents the cost of travelling directly from city $u$ to city $v$ (we will assume $w(u, v) = w(v, u)$ always). The salesman's task is to visit each city *exactly* once before returning to the starting point, and although the tour is restricted to visit every city exactly once, there is freedom to choose the *order* in which the cities are visited. Therefore a (potential) tour can be identified with a permutation $\pi : V \to \{0, \ldots, n-1\}$ of the nodes/cities, and the cost of such a tour is the sum of all the distances between the adjacent cities along this tour, including the final edge back to the start:

$$\sum_{i=0}^{n-1} w(\pi(i), \pi((i+1) \bmod n)).$$

The *Travelling Salesman Problem* (TSP) is to find the shortest tour that visits each city once and returns to the starting city. It one of the classical *NP-complete* problems[2], having been shown to be NP-complete in Karp's famous 1971 paper "Reducibility Among Combinatorial Problems" [3].

The definition of TSP allows any combinations of (positive) weights to define distances between cities, including very unrealistic combinations. If we are to believe that the distances correspond in some way to a natural setting, we might assume that the distances correspond to the Euclidean ("straight line") distances from some embedding of nodes in the plane. This is maybe too idealistic (and indeed a salesman taking public transport would not be travelling straight-line routes) - however, most natural TSP problems would have distances which at least satisfy the *triangle inequality*, where we insist that $w(u, v) \leq w(u, x) + w(x, v)$ for all $u, v, x \in V$. This restriction of the general TSP problem is called the *metric TSP* problem. The case where the distances are given by a planar embedding is called the *Euclidean TSP* problem. It is interesting that both these restricted cases remain *NP-complete*.

[1] Note that it is not difficult to adjust the weights to model missing edges - we just add a very high weight to that edge.

[2] Of course, the NP-completeness proof will be set-up in terms of the decision version of the problem, where we query whether there is a tour of cost less than some given $d$.

[3] Karp's paper, following "hot on the heels" of Cook's breakthrough result, demonstrated that 21 extra problems were NP-complete. One of the 21 was "Hamiltonian Cycle", and it is an easy inference that when HC is NP-complete, TSP must also be.

In this coursework we will experiment with some *heuristic* approaches to TSP - a heuristic method is a method which does not necessarily give any rigorous performance guarantees for the worst-case, but which may do fairly-well on many instances of the problem.
Your implementation should be in Python.
The relevant files for this project are available as `IADS_cwk3.tar`, and this contains some test files, a `graph.py` file for implementation of the required methods, plus an (empty!) `test.py` file where you should set up your experiments. You will be required to submit `graph.py`, `test.py` plus also a short report `report.pdf`. Details for the submission will follow.

## Part A [25 marks]: Dealing with Graphs

The tasks in this Part of the coursework are simple setting-up/evaluation tasks involving reading-in files and building the corresponding graphs, plus implementing a simple method to score the cost of the current tour.
`graph.py` has one class `Graph`, and includes the declaration of the `__init__` method:

```
def __init__(self,n,filename):
```

We will allow for two kinds of input files, and your implementation of `__init__` will need to identify and handle these two cases appropriately to build the "table" (list of lists) storing the distances between each pair of nodes:

- We will allow instances of *Euclidean TSP*, these being flagged by calling the initialisation with `n` set to the flag `-1`.

  These Euclidean file instances will contain just the *points/nodes* of the graph, each point described as a pair of integers (the x and y coordinates) on a single line, without any formatting. See `cities25`, `cities50`, `cities75` for examples of the format.

  The distance between any pair of points can be calculated using the `euclid` method in `graph.py`

  The number of nodes of the graph is exactly the number of points in the file, so we lose nothing by using `n` as a flag to indicate this type of input.

- We will also allow more general TSP inputs, possibly metric TSP, but even allowing non-metric instances to be evaluated. However, for this format, we will assume the distances are integers.

  For these inputs, the initialisation must be called with `n` set to the number of nodes of the input graph (which will certainly be greater than -1). Each line of the input file describes one *edge* of the graph, each line containing three integers, these being the first endpoint `i`, the second endpoint `j` and the given weight/distance for the edge between `i` and `j`. See `sixnodes` for a tiny example of this kind of input file.

We ask you to implement `__self__` so that we initialise the following variables as described.

- `self.n` - this should be initialised to the number of nodes/cities in the graph.

  This will be the number of points in the input file for the Eulerian case, or the given `n` in the general case.

- `self.dists` - this will be a two-dimensional "table" (in Python, a list of lists), such that the entry `self.dists[i][j]` will contain the distance (either *given* in the general TSP, or calculated by `euclid` in the Euclidean TSP) between nodes `i` and `j`).

  This table should remain unchanged throughout the execution of the various methods of the class.

  In filling this table, take care to always initialise the `[j][i]` cell as well as the `[i][j]` cell.

- `self.perm` - this is a list of length `self.n` which represents the *permutation* of the cities for the *current tour*.

  We will always initialise `perm` with respect to the identity permutation - ie, you should initialise this to have `perm[i]` equal to `i`.

  Note that we *do* expect that `self.perm` will change as our heuristic methods are applied to the data.

It may be necessary to define some local variables for `__init__` (in particular, it may be helpful to have a list of (pairwise) tuples to temporarily store the input points for the Euclidean case); however, these are the only class variables which should be set up within `__init__`.

Your final task for Part A is to complete the method `tourValue` which evaluates, and `return`s, the cost of the current tour, which is defined by the order of values in `self.perm`.

```
def tourValue(self):
```

Make sure that you remember to include the cost of the "wraparound" edge from the final node of the permutation round to the first node.

**testing:** If you run this method for the `cities50` input file, *before* applying any optimisation heuristics, the result should be about `11842.557`. For example:

```
>>> g = graph.Graph(-1,"cities50")
>>> g.tourValue()
11842.557992162183
```

## Part B [35 marks] - Some Basic Heuristics

The default order given by the input can be used to specify an initial tour, but in general this will not be a particularly good route. In this section we will explore how to improve the tour by applying a few basic heuristics.

### Swap Heuristic [10 marks]

In the first of these heuristics (called the "Swap Heuristic"), we explore the effect of repeatedly swapping the order in which a pair of *adjacent* cities are visited, as long as this swap improves (reduces) the cost of the overall tour.
The method which governs the high-level control of this heuristic (the repeated swapping, until we can see no improvement from these swaps) is given to you as `swapHeuristic`. This means that we have reached a *local optimum* with respect to the swap-mechanism - this does *not* imply that we will have a global optimum with respect to our input graph (as we will see when we run other heuristics).
Your job is to write the `trySwap` method which considers the effect of a specific swap (of `self.perm[i]` and `self.perm[(i+1) % self.n]`), determines whether this change will (strictly) improve the cost of the current tour, and *if so*, swaps those two values in `self.perm`.

```
def trySwap(self,i):
```

The function should `return True` if the update improves the cost of the tour (and the swap is made) and `False` otherwise.

**testing:** If we run `swapHeuristic()` with a correct implementation of `trySwap` on the initial data (with default permutation) from `cities50`, we should get a tour of cost about `8707.056`:

```
>>> import graph
>>> g = graph.Graph(-1,"cities50")
>>> g.swapHeuristic()
>>> g.tourValue()
8707.056392932445
```

### 2-Opt Heuristic [10 marks]

The *2-Opt Heuristic* is another heuristic which repeatedly makes "local adjustments" until there is no improvement from doing these. In this case the alterations are more significant than the swaps - the method `TwoOptHeuristic` repeatedly nominates a contiguous sequence of cities on the current tour, and proposes that these be visited in the reverse order, *if* that would reduce the overall cost of the tour. The high-level method `TwoOptHeuristic` has been implemented in `graph.py`.

Your job is to implement the `tryReverse` method that considers the effect of a proposed reversal:

```
def tryReverse(self,i,j):
```

Your implementation must consider the cost of the current tour (as defined by `self.perm`), and then consider the effect of reversing the "tour segment" from `self.perm[i]` to `self.perm[j]` (in other words, to change `self.perm[i]` to the old `self.perm[j]`, `self.perm[i + 1]` to the old `self.perm[j - 1]`, and so on.)
If this reversal will make the tour length (strictly) shorter, then `tryReverse` must commit `self.perm` to this reversal, and then `return True` indicating "success"; otherwise `self.perm` must be left with its prior order, and `False` should be returned.

**testing:** If we have correct implementations of `trySwap` and `tryReverse` and run `swapHeuristic()` immediately followed by `TwoOptHeuristic` on the initial data from `cities50`, we should get a tour of cost about `2686.814`:

```
>>> g = graph.Graph(-1,"cities50")
>>> g.swapHeuristic()
>>> g.TwoOptHeuristic()
>>> g.tourValue()
2686.8136937666072
```

### Greedy [15 marks]

A commonly used approach to optimization problems is the "greedy" approach, where at each step we do what seems best, and hope that this will lead to a globally optimal solution. For the TSP problem, this approach involves taking some initial city/node (for us, we will take the one indexed 0) and building a tour out from that starting point. At the $i$-th step (for $i = 0, \ldots$), we consider the recently-assigned endpoint in `self.perm[i]` against all *previously unused* nodes, and then we take our next node (to be saved as `self.perm[i+1]`) to be the one closest in distance to `self.perm[i]`. This will eventually create a permutation within `self.perm`.
The final task in this section is to implement this method within `graph.py`:

```
def Greedy(self):
```

Test your new function. How well does the greedy heuristic perform? (it will not always do as well as the combination of `swapHeuristic`, then `TwoOptHeuristic`). On `cities50`, the result should be around 3011.593:

```
>>> g = graph.Graph(-1,"cities50")
>>> g.Greedy()
>>> g.tourValue()
3011.5931919734803
```

## Part C [20 marks] - Your Own Algorithm

The methods described in Part B are basic methods from the literature, which perform fairly well on many instances. However, there is scope for improvement!
In this section, you are asked to implement an algorithm of your own, which you should describe in your report. The algorithm should be *polynomial-time* and should be motivated by the specific details of TSP, whether the general case, or the metric/Euclidean version. The TSP problem has been worked on by many researchers, and if you want to draw on some past work, and implement a known (perhaps more interesting) algorithm for TSP than the ones in this specification, that is fine - you must credit the original authors, describe the algorithm(s) in your own words, and also you must write the implementation yourself.
This section will be assessed in terms of the *Algorithm* first section of your report, and we expect this first section to be about 1.5 pages of A4.
You should also include an implementation of your algorithm inside `graph.py` (you may name it how you like).

## Part D - Experiments

You have been provided with a small number of graphs to test your algorithms. In this section, you are asked to carry out a broad collection of comparative experiments on all your implemented algorithms (the three from Part B, and any Algorithm you have implemented for Part C). You will probably want to write some code to generate random graphs according to certain parameters for size (number of nodes) and weights (range for the weights, and probabilities).
Here there is scope to consider different settings for evaluation:

- The Euclidean setting (maybe varying the "window" of the plane to take in larger/wider values for the x and y co-ordinates).
- The Metric setting (there are different ways of generating a graph which is guaranteed to be a metric).
- The general non-Metric setting - does this change the quality of the results?

In carrying out experiments on randomly generated instances (or indeed, even the input files for the Eulerian case), the questions arises of how close we are to the best answer? That is something hard to know *in general*, as we have no polynomial-time algorithm to solve the problem! Some ideas that could be used are the following:

- We may consider implementing a simple method (branch and bound, direct enumeration) to compute an exact solution, when we are working with a small number of nodes.
- For larger number of nodes, we can consider generating a graph to ensure it will have a "planted" (but hidden) high-quality solution, then disguised by re-ordering of vertices and many distracting competing edge-weights. This can provide more insight (and more interesting coding) than blindly generating random graphs.

You should submit your code in `tests.py`; however, the most important detail is to describe your experiments in the second ("Experiments") section of your report. Please don't only show tables, give some explanation of what was interesting.

## Submission

You should submit a completed `graphs.py` file (with your own algorithm added), plus your tests in `tests.py`. You should also submit a `report.pdf` file of about 3 pages of A4, with a section on "Algorithm" and another on "Experiments".
Submission will be through Learn, details to follow.

Mary Cryan and John Longley, January 2020