Artificial Intelligence Search Assignment

This is the assignment for the sub-module *Artificial Intelligence Search* of the module *Software Methodologies*. It is to be completed and handed in by **2 p.m.** on **Thursday 23rd January 2014**.

Basic instructions

You are to implement 2 or 3 different algorithms (using techniques studied during the lectures and detailed in the hand-outs, but possibly also additional algorithms you have devised or discovered for yourself) to solve the Travelling Salesman problem. Full marks can be obtained from the implementation of only 2 different algorithms but extra credit can be earned by implementing another algorithm (note that however many algorithms you choose to implement, the page-bounds, given below, on the length of the report are strict). An idea might be to implement 2 algorithms that give good results and another more complicated algorithm that might not give such good results (if any) but where implementing the algorithm is more difficult. Your implementations should seek to obtain the best Travelling Salesman tour given a collection of cities and their distances (you might have to read ahead in the hand-outs if you wish to start implementing immediately upon receiving this assignment).

The complexity of the underlying algorithms that you choose will impact upon the marks awarded. For example, if you choose the obvious 'brute-force' algorithm or a 'nearest-neighbour' algorithm then you cannot expect to be awarded as many marks as if you implemented a more complicated algorithm like an A* search, a genetic algorithm or simulated annealing. Your implementations should always detail the actual tours they find as well as the lengths of the tours (in the format described below).

You are strongly advised to develop your own implementations using the pseudo-code given in lectures and hand-outs rather than search the web for related implementations.

You should ensure that you have implemented your algorithms *at least* two working weeks before the hand-in date so as to give yourself time to experiment on the Travelling Salesman instances upon which your implementations should be tested. The answers provided by your implementations to these instances should be submitted, along with a report of your implementations and your experimentation (what is expected is described in more detail below). The two weeks allocated for experimental work will give you ample time to fine-tune your implementations, in the light of experimental experience, so as to obtain the best results you can.

The data-files

The actual instances of the Travelling Salesman problem (more precisely, the symmetric Travelling Salesman problem, where the distance from city x to city y, denoted (x, y), is always the same as the distance from city y to city x, that is, (y, x)) will be given in the following form (the cities are always named $1, 2, \ldots, n$).

> `NAME = <string` = name-of-the-data-file`>`,
> `SIZE = <integer` n `=` the number of cities in the instance`>`,
> `<list-of-integers` $d_1, d_2, d_3, \ldots, d_m$`>` where the list consists of
> the distances between cities $(1, 2), (1, 3), \ldots, (1, n)$,
> then the distances between cities $(2, 3), (2, 4), \ldots, (2, n)$,
> ...
> and finally the distance between the cities $(n - 1, n)$.

Commas ',' are used as delimitters in data-files; carriage returns, end-of-line markers, spaces, etc., should be ignored.

So, for example, the instance with 5 cities where: the city 1 is at the origin; the city 2 is 3 miles north; the city 3 is 4 miles east; the city 4 is 3 miles south; and the city 5 is 4 miles west, and all distances are the Euclidean distances between cities, is encoded as the city-file `AISearchsample.txt`:

```
NAME = AISearchsample,
SIZE = 5,
3, 4, 3, 4,
5, 6, 5,
5, 8,
5
```

With reference to the remark above, re: delimitters, the above city-file could well be presented with no carriage returns, etc., as simply

```
NAME = AISearchsample,SIZE = 5,3,4,3,4,5,6,5,5,8,5
```

Note that in general a given instance need not be based on the Euclidean distances of a collection of cities on the plane. You should assume that a given distance between two cities is a non-negative integer (and so it could well be the case that the distance between two distinct cities is 0).

First task: Your first task is to be able to read in the data from a city-file and store it internally for use by your implementation. I strongly recommend that you write some code to read the data from a city-file into a two-dimensional list whose (i, j)th entry stores the distance from city i to city j. So, with the city-file `AISearchsample.txt`, above, if the list is called *city* then $city[2][3]$ has the value 5 as does $city[3][2]$. Also, there are carriage returns and other control characters in the city-files. It is your job to ignore these characters. The reason these characters are there is because this is how data is presented in the real world: as 'dirty data'. I would advise reading the city-files character by character and stripping out characters that are not as you would expect from a well-formatted city-file. I have supplied a sample city-file, `AISearchtestcase.txt` (that is dirty), for you to experiment on.

Your tour-files

You are expected to derive the best tours that you can with your implementations and to provide the actual tour in each case. Your tour-file should be named by prefixing the name of the city-file with the prefix '`tour`' and should have the following form:

> `NAME = <string =` name-of-TSP-instance-file`>`,
> `TOURSIZE = <integer` n `=` integer-giving-the-number-of
> -cities-in-the-tour`>`,
> `LENGTH = <integer` len `=` integer-giving-length-of-tour`>`,
> `<list-of-integers` $x_1, x_2, x_3, \ldots, x_n$`>` where $x_1, x_2, x_3, \ldots, x_n$ is a list of the cities in the order of the tour (with the final hop being from city x_n back to city x_1).

So, a tour-file for the above city-file `AISearchsample.txt` might be the file `tourAISearchsample.txt` of the form:

```
NAME = AISearchsample,
TOURSIZE = 5,
LENGTH = 24,
2,3,1,5,4
```

I have supplied a sample tour-file, `tourAISearchtestcase.txt` (for the sample city-file `AISearchtestcase.txt`).

IMPORTANT!

You are expected to hand in a folder named with your username, e.g., `dcs0ias`*, and within which:*

- *If you implement 2 algorithms then there are 2 folders named*

  `TourfileA` *and* `TourfileB`

  and if you implement 3 algorithms then there are 3 folders named

  `TourfileA`, `TourfileB` *and* `TourfileC`.

  Each folder should contain 10 tour-files, named

  `tourAISearchfile012.txt`, `tourAISearchfile017.txt`, *etc.*,

  detailing the best tours that particular implementation has found in the respective collection of cities;

- *there is a pdf file, entitled* `dcs0iasreport.pdf`*, in the folder named* `dcs0ias` *including a detailed description of your implementations (each marked as A, B or C, as appropriate), tabulations of the results produced, a comparative analysis, and an explanation of your experiments (the report should be no more than 4 pages of A4, in a single column format, and no smaller than 11 pt. font; there is no need for cover and contents pages, etc.); and*
- *there is another folder, called* `dcs0iasrest`*, in the folder* `dcs0ias` *containing your implementations and anything else of relevance.*

Ensure that your material is handed in as above with files and folder so named (you can zip everything together into one file if you like).

Mark scheme

Credit will be given for good working implementations and good experimental results as demonstrated in your detailed account, structured as follows.

- Full and clear descriptions of your implementations focussing on implementation issues (do not include your code in your report: focus on an overview of how your implementation works and on any specific implementation details such as choice of data structures or data representation). If your chosen algorithms are structurally and intellectually simple, e.g., brute-force search, 'nearest-neighbour', etc., then you are not likely to score as highly as if you had implemented a more complicated and involved algorithm [25 marks].
- A thorough (tabulated) description of your results (especially the lengths of the best tours obtained) and their quality. The better the tours you find, the better the marks [30 marks].
- Details of your experiences with running your implementations on the different input cases and a comparative analysis, together with a discussion of any significant fine-tuning and experimentation you performed in order to try and improve performance [45 marks].

You will be given marks only for the content of the report. Do not expect me to look in other files or folders for missing details. These other folders are supplied just for me to clarify your results if I choose to do so.

It is absolutely crucial that you conform to the above format. I automatically check that your tours are indeed of the lengths you state and if I can't do this because you have not formatted files properly then you will score no marks for tour quality. I provide my tour-checking Python program for you to use so that you can be sure that your files are properly formatted.

In general, I am looking for a good understanding of the algorithms you have implemented and some ingenuity in manipulating these implementations so as to try and get better results. I will reward ingenuity even if the method you have chosen to implement does not in general give good results. However, I will also reward students who get good results. If you choose to implement a brute-force search, for example, then you may not score so well, because the brute-force search is pretty useless in general. However, you might gain credit if you were to try and improve your brute-force search by some ingenuity (but your tours will not be very good whatever you do, though; trust me!). The choice is yours as to the algorithms you implement. As regards your implementation descriptions, I am looking for a well written, concise and informative presentation, and as regards running your implementations, I am looking for 'innovative tinkering' to try and get improvements.

Some remarks and hints

Essentially, the following methods are available to you (as studied in the course):

- brute-force search
- nearest-neighbour
- best-first search
- greedy best-first search
- A* search
- hill-climbing search
- simulated annealing
- genetic algorithm.

There is a little bit of work to do as regards the implementation of each method and here is a little bit of help.

Brute-force search

Here is a hint as to how all tours in an n-city Travelling Salesman instance can be generated. Each tour is stored in a 1-dimensional list T of size n, where the elements are all distinct and come from $\{1, 2, \ldots, n\}$. The tour stored in T is $T[1], T[2], \ldots, T[n], T[1]$. In fact, we can similarly store a tour of the $m \leq n$ cities $\{1, 2, \ldots, m\}$ in $T[1], T[2], \ldots, T[m]$, with $T[m+1] = T[m+2] = \ldots = T[n] = 0$.

Our procedure `gen(`T`,`m`)` takes as input a tour of m cities, $x_1, x_2, \ldots, x_m, x_1$, say, held in the list T (as above), and proceeds as follows.

- If $m = n$ then we compare the length of the tour T (of n cities) with the length of the shortest tour found so far and if T is shorter then we remember T and its length.

- If $m < n$ then `gen(`T`,`m`)` generates all tours of the cities $\{1, 2, \ldots, m, m+1\}$ by inserting the value $m+1$ in location 1, then location 2, ..., then location m, then location $m+1$ of T (so that the cities coming after $m+1$ are 'shifted' along the array). Interleaved with generating each tour, which we refer to as T', we recursively call the procedure `gen(`T'`,`$m+1$`)`.

In more detail, `gen(`T`,`m`)` is as follows.

```
if $m$ == $n$ then
   calculate the length of the tour $T$ and if it is shorter
   than the best tour found so far, remember $T$ and its
   length as the best tour found so far
else
   for $i$ = 1 to $m+1$ do
      $T'$ = $T$ with the city $m+1$ inserted into location $i$
      call gen($T'$,$m+1$)
      $T'$ = $T$ with the city $m+1$ removed from location $i$
fi
```

Thus, the following pseudo-code generates and tests all possible tours, given the distance-file of n cities.

```
$T$ = $[1,0,0,\ldots,0]$      %initialize tour $T$ as [1]
$m$ = 1                 %initialize number of cities of tour $T$ as 1
call gen($T$,$m$)
output the shortest tour found
```

Although I don't recommend that you implement a brute-force search, brute-force searches are good for checking optimal values in small cases; also generating all combinatorial possibilities comes up regularly and it is useful to know how to do this.

Best-first, greedy best-first and A* search

The Travelling Salesman Problem needs to be realised as a search problem. One way of doing this is to have the set of all lists of distinct cities as the states together with the lists of n distinct cities augmented with the start city (and so a state is a list of between 0 and n cities, or a list of $n+1$ cities where the first n are distinct and the last city equals the first). There is one action with a state t' being a successor of a state t if the list t' is the partial tour t extended with one new city (not appearing in t), or if t has length n and t' is t augmented with the first city of t. The step-cost associated with any transition from state t to state t' is the cost of moving from the final city in the list t to the final city of the list t'. The initial state is the list t_0 consisting of just the start city and a goal state is a list of $n+1$ cities. An optimal solution is thus a path from the initial state to a goal state of minimal cost.

There are a number of heuristic functions available for the Travelling Salesman Problem. One of these is the heuristic h where, given a state $t = x_1, x_2, \ldots, x_r$, $h(t)$ is defined to be the minimal step-cost of moving to a state of the form $t' = x_1, x_2, \ldots, x_r, x_{r+1}$ (that is, always move to the nearest legal city from where you are; if there is no city to move to then $h(t) = 0$).

Another heuristic is as follows. Given some state t, your heuristic function $h(t)$ is the sum of the distance of the closest city c that has not been visited to the last city of the partial tour t plus the distance of any other unvisited city (different from c) to the start city (if there aren't enough unvisited cities to apply this rule then the heuristic value is 0). This heuristic reflects that you want your next city to be visited to be close to the current city but that you don't want to be left with a city that is a long way from the start city.

Yet another heuristic h is as follows. Given a state $t = x_1, x_2, \ldots, x_m$, $h(t)$ is defined to be the minimal step-cost of moving to a state t', where t' is t with some new city inserted somewhere within t, e.g., if $t = 1, 5, 4, 7$ then t' might be $1, 5, 6, 4, 7$. If this heuristic is to be used then the transition function (above) needs to be amended to allow such transitions.

Bear in mind that A* search gives optimal solutions (under mild circumstances) and so unless you have a brilliant heuristic function (which is unlikely) then this method will only work on small instances. I have no idea how the above heuristics will pan out on the instances given!

Hill-climbing search and simulated annealing

Here, the states might be the set of all possible tours of n cities, and one state t' might be a successor of another state t if a swap of the positions of two (or more!) of the cities in the tour t results in the tour t'. The heuristic cost function of a state might be the length of the tour. There are numerous other definitions of a successor function.

Genetic algorithms

In order to formulate the Travelling Salesman Problem for solution by a genetic algorithm, we need to define our population. One way of doing this is to define the population as a set of tours of n cities, represented as strings of length n. The fitness of a member of the population might be just the length of the tour. We now need to come up with a notion of mutation and crossover. One way of defining a mutation is just to randomly swap the positions of two cities within a tour (though there are many others). Defining a notion of crossover is more difficult. However, given two tours $t = x_1, x_2, \ldots, x_n$ and $t' = x'_1, x'_2, \ldots, x'_n$, we could define a new tour as follows.

- Randomly choose some $i \in \{1, 2, \ldots, n-1\}$ and form the strings

$$s = x_1, x_2, \ldots, x_i, x'_{i+1}, x'_{i+2}, \ldots, x'_n$$

and

$$s' = x'_1, x'_2, \ldots, x'_i, x_{i+1}, x_{i+2}, \ldots, x_n$$

(note that these might not be tours as some cities might be missing and some repeated).

- Scan through s and make a list of the cities not appearing in s and a list of the locations containing repeated cities (these lists have the same length). Replace every repetition with a missing city (according to some user-defined strategy). Do the same for s'.
- Hence, we obtain two tours s and s', and we take the crossover as the shortest one.

For those really interested, there is a paper:

- P. Larranaga, C.M.H. Kuijpers, R.H. Murga, I. Inza and S. Dizdarevic, Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators, *Artificial Intelligence Review* **13** (1999) 129–170

that discusses genetic algorithms for the TSP.

Please note: the above hints are just suggestions and you might care to come up with your own ideas. Also, it will be up to you to (experimentally) vary parameters (e.g., the different probabilities in a genetic algorithm or a simulated annealing algorithm) to improve your solutions.