COMP 3958: Assignment 1

Submit your source code in a file named `a1.ml` to the BCIT Learning Hub. Submission information and deadline will be provided. Your file must build without warnings or errors; otherwise, you may receive no credit for the assignment. Be sure to comment each function in your code and any part of your code that may not be clear to the reader.

For this assignment, you are asked to implement a genetic algorithm using functional techniques in OCaml and apply it to find an approximate solution to the travelling salesman problem (TSP). Note that you are not allowed to use references or any of the imperative features like *for* or `while` loops in your implementation. However, you will need to use arrays for the distance matrix (see below) and the shuffle function we talked about in class.

In TSP, we are given a list of cities and the distances between each pair of cities. We want to find the shortest possible tour. (A tour is a route that visits each city exactly once and returns to the starting city.) This turns out to be a so-called NP-hard problem.

Genetic algorithms are based on the principle of evolution in biology. We start with a population of randomly generated candidate solutions (called individuals) and try to evolve them to better solutions. It is an iterative process. A new population (also called a new generation) is created from an existing population at each iteration. This terminates when either a maximum number of generations has been produced, or a satisfactory solution has been reached.

Each individual in a population is assigned a fitness which is a measure of the goodness of the solution. To create a new generation, every individual in an existing population is evaluated. Fitter individuals are selected (usually randomly) to create new offsprings using an operation called crossover (a way to combine the "genetic information" of two parents). Random mutations may also be applied to the population.

We now look at how we can use a genetic algorithm to tackle TSP. We assume the given cities are numbered from 0 to $n-1$ inclusive (for a total of n cities) and the distances between the cities are given by a distance matrix of integers. (The entry of the matrix in the `i`-th row and `j`-th column is the distance between cities `i` and `j`. The matrix is symmetric.) Each individual in a population is a tour that we represent as a list containing a permutation of the integers from 0 to $n-1$ inclusive. For example, the list [2; 3; 1; 0] (for 4 cities) represents the tour that goes from 2 to 3, then to 1, then to 0, and finally back to 2. It has four segments: 2 to 3, 3 to 1, 1 to 0, and 0 to 2. The length of the tour is the sum of the distances of its segments. (The distance of each segment is found via the distance matrix.) Note that this is different from its length as a list. The fitness of a tour is related to its length — the shorter the length, the fitter the individual. Our algorithm needs to minimize the length of tours.

At the start of the algorithm, we generate a population of random tours. This is represented by a list of tours, each tour is itself a list. Implement a function `populate` so that `populate n size` returns a list of `size` random tours, each a tour of `n` cities.

At each iteration, the population undergoes the following stages to create the next generation:

1. Evaluation: Sort the population in ascending order of tour lengths. After this, the first tour in the population has smallest length.

   Implement a function `length` such that `length distances tour` returns the length of `tour` using the distance matrix `distances`.

2. Selection: We select parents to create a new generation. In this version, the first tour in the population always goes into the next generation unchanged. We then pair consecutive tours in the remaining sorted population to generate (via crossover in the next stage) the rest of the new population. Note that if the population size is even, the last tour will not have a partner (as the first tour is not paired with anyone) and it is simply discarded. This should happen at most once if we start with an even population size.

   Implement a function `pairs` so that `pairs lst` groups the elements of the list `lst` in pairs; any extra element at the end is dropped. For example `pairs [3;2;7;6;8]` returns `[(3, 2); (7, 6)]`.

3. Crossover: Each pair from the selection stage undergoes two crossover operations to produce a pair of new tours. These new tours replace the original pair of tours in the new generation.

Implement a function `cross` so that `cross pos tour1 tour2` returns the crossover of `tour1` and `tour2` at position `pos`, i.e., the first `pos` elements of the result are the first `pos` elements of `tour1`, the rest are the elements of `tour2` that are not in the first `pos` elements of `tour1`, in the order they occur in `tour2`. For example, `cross 3 [3;1;4;2;0] [1;0;4;2;3]` returns `[3;1;4;0;2]`. Note that the first three elements of the resulting tour are the first three elements of the first tour. The rest are in the same order as in the second tour.

For each pair `(t1, t2)`, we choose a random position `pos` between 0 and n (the number of cities) inclusive, and create two new tours by calling `cross pos t1 t2` and `cross pos t2 t1`.

4. Mutation: A mutation randomly swaps a pair of cities in a tour.

   Provide a function `swap` so that `swap i j lst` returns a list that is the same as `lst` but with its `i`-th and `j`-th elements swapped. Note that `i` and `j` start from 0 and must be less than the length of `lst`.

   In this stage, we go through each tour in the population except for the first tour (which was the fittest in the previous generation) and randomly decide to mutate it or not. The probabilty of mutation is given by a floating-point number between 0 and 1 (inclusive). For each tour in the population, we generate a random floating-point number between 0 and 1. If this number is less than the probability of mutation, we mutate the tour; otherwise we don't. To mutate the tour, we randomly generate an suitable pair of integer `i` and `j` and call `swap i j tour` to get a new tour.

We can think of each of the above four stages as a sort of transformation of the population.

Write a function `algo` that implements the genetic algorithm specified above. This function takes 4 arguments and is intended to be called as follows:

```
algo niters prob distances pop
```

where

- `niters` is the number of iterations to run the simulation (at least 1)
- `prob` is the probability of mutation (a floating-point number between 0 and 1 inclusive)
- `distances` is the distance matrix that specifies the distances between cities
- `pop` is the starting population with at least 1 tour

It returns the shortest tour length found. It is up to the caller to ensure that the function is called correctly.

The distance matrix specifying the distances between cities will come from a file containing an n-by-n matrix of non-negative integers You must provide a function `read_distances : string -> int array array` that takes a file name and reads the data in the file into a matrix (array of array). This function may assume that the input data is valid, i.e., there are n rows (for some positive integer n) and each row has exactly n integers separated by spaces and the integers are non-negative. Note that the cities are then numbered from 0 to $n-1$ inclusive.

You will also need to implement a function named `run` that provides a more user-friendly interface to the `algo` function. `run` can be called as follows:

```
run file pop_size prob niters
```

where

- `file` is the name of the file containing the distance matrix (`run` will need to read this file)
- `pop_size` is the population size, a positive integer (`run` will need to generate a random population of this size)
- `prob` is the probability of mutation (between 0 and 1)
- `niters` is the number of iterations to run the simulation (at least 1)

Note the order of the arguments. Just like `algo`, it returns the length of the shortest tour found.

Put your code in a file named `a1.ml`. We will be testing it in `utop` as well as compiling it using `ocamlbuild` (`ocamlbuild -pkg=str a1.cmo`). Besides the two main functions (`algo` and `run`), the auxillary functions mentioned in this document (`length`, `populate`, `pairs`, `cross`, `swap`, `read_distances`) are needed for testing. Be sure to implement them as specified. You may also need to initialize the random number generator. More information will be provided.