COMP 3958: Asignment 1

Submit a zip files containing your source files `anneal.ml`, `anneal.mli`, `tsp.ml`, `tsp.mli` and `main.ml`. Submission deadline and instructions will be provided. Your files must build without warnings or errors; otherwise, you may receive no credit for the assignment. Note that this assignment is worth 15% of your grade.

For this assignment, you are asked to implement simulated annealing using functional techniques in OCaml and apply it to find approximate solutions 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.

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

Simulated annealing is a way to get an approximate solution to a global optimum of a function. The name comes from annealing in metallurgy: the heating and controlled cooling of a metal.

Suppose we want to find a solution x so that $E(x)$ is a minimum. (We think of E as the energy function, and we want to find the state with lowest energy.) We start with an initial state s and picks a random move to state s'. If the move leads to a better solution (i.e, lower energy: $E(s') < E(s)$), it is accepted. Otherwise, there is still a probabilty of accepting it. Accepting worse solutions allows for a more extensive search for the global minimum.

The probability of accepting a worse move depends on how worse the move is, i.e., it depends on $E(s)$ and $E(s')$ (actually their difference), and also on a parameter T that we think of as the temperature. As T decreases, the probability of accepting a worse solution decreases.

In a simulation, the temperature progressively decreases from an initial positive value. At each step, the algorithm randomly selects a solution close to the current one, measures its quality, and decides whether to move to it. The simulation stops after a certain number of steps.

Here is the pseudocode based on the one in Wikipedia, slightly modified:

```
/* see: https://en.wikipedia.org/wiki/Simulated_annealing#Pseudocode */
Let s = s0
For k = 0 through kmax (exclusive):
  T <- temperature(k)
  Pick a random neighbour, s' <- neighbour(s)
  If P(E(s), E(s'), T) >= random(0, 1):
    s <- s'
Output: the final state s
```

In the above, `s0` is the starting state, `neighbour(s)` returns a randomly chosen neighbour of state `s` and `E` is the energy function. These are specific to the particular problem we are working on. For example, for TSP, a state is a route that visits every city exactly once, returning to its starting city, and its energy is the length of the route.

For the annealing, we need to choose a temperature function (which should be a montonically decreasing function that depends on the iteration, `k`) and an acceptance function, P. `random(0, 1)` simply returns a random value in the range `[0, 1]`, uniformly distributed.

Our choice for P is as follows:

$$P(E(s), E(s'), T) = \begin{cases} exp((E(s) - E(s'))/T) & \texttt{if } E(s) < E(s') \\ 1 & \texttt{otherwise} \end{cases}$$

exp is the exponential function. Note that the argument to it is negative. Note also that if $E(s)$ and $E(s')$ are the same, s' is accepted.

Since we will be dealing with a very large number of steps (i.e., `kmax` is very large) in a simulation, instead of using a temperature function that decreases at every step, we choose one that decreases by

a constant factor after every n steps where n is a positive integer. We will call this n the temperature interval — the interval, in number of iterations, between temperature changes.

In a file named `anneal.ml` (corresponding to module Anneal), implement a function `run` that runs the simulation specified by the pseudocode above. This function has 3 parameters (although the first and second parameters are themselves triples – tuples of 3 elements) and is intended to be called thus:

```
run (s, energy, next) (t, factor, interval) numsteps
```

where

- `s` is the starting state
- `energy` is the energy function; `energy s` gives the energy of state `s`
- `next` is a function with `next s` returning the next state (the `neighbour` function in the pseudocode above)
- `t` is the starting temperature
- `factor` is the factor by which the temperature decreases ($0 <$ `factor` < 1)
- `interval` is the temperature interval (explained above)
- `numsteps` is the number of iterations to run the simulation (the `kmax` above)

Note that the triples group related parameters together.

`run` returns the final state and has signature

```
val run : 'a * ('a -> float) * ('a -> 'a) -> float * float * int -> int -> 'a
```

You may use additional helper functions in `anneal.ml`. You will also need to provide a `anneal.mli` file containing the signature of the functions you want to export.

The code for the TSP portion of this assignment goes in a file named `tsp.ml` (corresponding to module Tsp). You will also need to provide a `tsp.mli` file containing the signatures of the functions you want to export. Data for the problem will come from a file containing an n-by-n matrix of floating-point numbers (the adjacency matrix of the graph). You must provide a function `read_distances : string -> float array array` that takes a file name and reads the data in the file into a matrix (array of array) and returns it. This function may assume that the input data is valid, i.e., there are n rows (for some positive integer n) and each row contains exactly n floating-point numbers separated by spaces. We regard the cities as numbered from 0 through $n - 1$ corresponding to the indices of the rows (and columns) of the matrix.

Each state is a route that visits each city exactly once and returns to the starting city. Hence it can be represented by a permutation of the cities, i.e., of the integers $0, \cdots, (n - 1)$. Its energy is the total distance of the route (including return to the starting city). Given a route, we randomly choose the next route by swapping a pair of cities in the original route. We use as the starting state the cities numbered from 0 to $(n - 1)$

Using the above information, implement a function `run` that runs simulated annealing for TSP. Its arguments are: the name of the matrix file; a triple consisting of the starting temperature, the temperature factor, and the temperature interval; and the number of iterations. It returns a pair consisting of the length of the path it finds as well as the path itself. Its signature is

```
val run : string -> float * float * int -> int -> float * int list
```

In `main.ml`, implement a complete program that can be executed on the command-line. The program takes the name of the matrix file and the number of iterations as command-line arguments. It basically calls `Tsp.run` with some fixed values: starting temperature is 100.0, the factor is 0.99, the temperature interval is calculated as (number of iterations/500) + 1. (The intention is to have the temperature changes approximately 500 times in the whole simulation; the +1 is to handle the case where the number of iterations is less than 500.) It outputs the length of the path it finds as well as the path itself as a sequence of integers.

Note that we will be using ocamlbuild to build your program: `ocamlbuild main.native`. Make sure that this command can build your progam without warnings or errors and be sure to comment what each function does. Additional requirements to facilitate testing will be discussed in class.