COMP 3958: Assignment 2

Submit a zip file containing the source files `digraph.mli`, `digraph.ml` and `main.ml`. Submission deadline and additional instructions will be provided. You may use functions from the standard OCaml library but not from other libraries. Also, you may not use any of the imperative features of OCaml. This essentially means you are not allowed to use references, arrays, for-loops or while-loops. Your files must build without errors; otherwise, you may receive no credit for the assignment. You may also lose marks for compiler warnings. Be sure to comment your code.

In this assignment, you will be implementing the Bellman-Ford algorithm to find the shortest paths from a specified source vertex to all other vertices in a weighted digraph (directed graph).

For our purpose, a digraph consists of vertices (labelled by strings) and edges. Each edge has a weight (an integer, which can be negative) and goes from one vertex (the source vertex) to another vertex (the destination vertex). We can represent such an edge as a tuple. For example, an edge from vertex `A` to vertex `B` with weight 5 can be represented by `("A", "B", 5)`.

You'll need to read information for a digraph from a file. Each line of the file contains information about one edge; it consists of 3 words: the source vertex, the destination vertex and the weight (which is an integer). For example, the content could be:

```
A B  5
B D -1
B E  1
A C  4
C D -2
A D  3
D E  3
```

The first line shows an edge going from `A` to `B` with weight 5. The other lines can be similarly interpreted.

You'll need to implement a function to read the graph data from a file. As there are other functions needed to manipulate a weighted digraph, implement a module named `Digraph` with the following signature:

```
(* see write-up for details *)
type t                                (* the abstract digraph type *)
type edge = string * string * int     (* type of an edge *)

exception Invalid                     (* raised when graph is invalid (see below) *)
val read_graph : string -> t          (* reads digraph from file; may raise Invalid *)
val edges : t -> edge list            (* returns (sorted) list of all edges in digraph *)
val vertices : t -> string list       (* returns list of all distinct vertices (in
                                         alphabetical order) in digraph *)
val is_vertex : string -> t -> bool   (* tests whether a given string is a vertex
                                         of the digraph *)
```

The `edges` function returns a list of the edges sorted in ascending order of source vertices; edges with the same source vertex are then sorted in ascending order of their destination vertices. It is an error to have two or more edges with the same source vertex as well as the same destination vertex even though they may have the same weight. In this situation, `read_graph` should raise the `Invalid` exception. `read_graph` also raises an exception when there is an edge connecting a vertex with itself.

As an example, after reading the file content shown above to get a digraph named `g`

- `Digraph.edges g` returns `[("A", "B", 5); ("A", "C", 4); ("A", "D", 3); ("B", "D", -1); ("B", "E", 1); ("C", "D", -2); ("D", "E", 3)]`
- `vertices g` returns `["A"; "B"; "C"; "D"; "E"]`

Note that the internal representation of the digraph type is up to you. The implementation of the `Digraph` module should be in a file named `digraph.ml` and its signature (which is basically what is shown above) in a file named `digraph.mli`.

The Bellman-Ford algorithm to find the shortest paths from a specified source vertex to all other vertices in a weighted digraph is described in the Wikipedia page:

`https://en.wikipedia.org/wiki/Bellman-Ford_algorithm`

The algorithm shown there uses imperative features and returns two arrays. Since you are not allowed to use arrays, for-loops, or while-loops in this assignment, you need to modify the algorithm to return a list of pairs: the first element of each pair corresponds to the distance and the second element, the predecessor. The loops will need to be rewritten into recursive calls.

Note that step 3 of the algorithm at the Wikipedia page is used to find negative-weight cycles. For this assignment, you do not need to implement this step. However, you need to detect whether there is a negative-weight cycle (i.e., you need to be able to tell whether there is a negative-weight cycle without actually finding such a cycle). This can be done by repeating step 2 one more time. If after doing this, at least one of the entries in the distance array (in the Wikipedia version) changes, then there is a negative-weight cycle; otherwise there isn't. In case there is a negative-weight cycle, raise an exception using `failwith`. Put your implementation of the algorithm in a function named `bellman_ford` with signature

```
val bellman_ford : string -> Digraph.t -> (int * string) list
```

This function should also raise an exception if the starting vertex is not in the digraph (again via `failwith`). Put this function in a file named `main.ml`. You may also need helper functions in this file.

As an example, for the digraph `g` corresponding to the sample file content, `bellman_ford "A" g` returns
`[("A", (0, "")); ("B", (5, "A")); ("C", (4, "A")); ("D", (2, "C")); ("E", (5, "D"))]`
Since `A` is the starting vertex, its shortest distance (from `A`) is 0. Note that we use the empty string to denote no predecessor. The fourth entry indicates that the shortest distance from `A` to `D` is 2 and its predecessor is `C`. We should be able to trace predecessors back to `A`. Note that if a vertex is not reachable from the starting vertex, it will conceptually have infinite distance (`Digraph.inf`). This can happen if the digraph is not connected and consists of separate components.