COMP 3958 Assignment

Instructions:

- Submit a zip file containing the source files `digraph.mli`, `digraph.ml` and `main.ml`. Additional instructions will be announced.
- To focus on the functional features of OCaml, **you are not allowed to use any of its imperative or object-oriented features in this assignment. In particular, this means you cannot use references, arrays, records with mutable fields, for-loops or while-loops.** If your submission does not meet this restriction, you will get litte, or even no, credit for this assignment.
- Your files must build without warnings or errors using `ocamlbuild` (see last paragraph); otherwise, you may receive no credit for the assignment as there may be no way to test your program.
- Provide comments to clarify your code.

For this assignment, you will be implementing an algorithm to find the shortest path between two given vertices in a graph.

For our purpose, a graph consists of vertices (labelled by strings) and edges, with each edge having a length (a positive integer) & going from one vertex to another vertex in the graph. We'll use the triple (v1, v2, l) to represents an edge from vertex v1 to vertex v2 with length l. Note that an edge has a direction. (Technically, we are dealing with weighted digraphs. However, in what follows, we'll call them graphs if there is no confusion.)

First, you are asked to implement an abstract data type to represent the type of graphs described above in a module named `Digraph`. This means that the implementation should be in a file named `digraph.ml` and the corresponding signatures in a file named `digraph.mli`.

The signatures in `digraph.mli` are basically as follows:

```
type t                          (* the abstract digraph type *)
type edge = string * string * int (* type of an edge *)
exception Inv_edge              (* raised by add_edge and of_edges (see below) *)
exception Inv_graph             (* raised by add_edge and of_edges (see below) *)

val empty : t                   (* the empty digraph *)
val add_edge : edge -> t -> t   (* add an edge to a digraph; may raise exception *)
val of_edges : edge list -> t   (* returns a digraph from a list of edges;
                                   may raise exception *)
val edges : t -> edge list      (* returns sorted list of distinct edges *)
val vertices : t -> string list (* returns list of all distinct vertices in
                                   alphbetical order *)
val neighbors : string -> t -> (string * int) list  (* returns list of neigbors of a
                                                       specific vertex (see below) *)
```

Note that a vertex B is a neighbor of A if there is an (valid) edge going from A to B.

A valid edge has positive length and its two vertices are labelled by distinct non-empty strings. Otherwise, the edge is invalid. The `Inv_edge` exception is raised when an invalid edge is encountered by `add_edge` (or `of_edges`). Even when an edge is valid, adding it to a graph may raise an `Inv_graph` exception. This happens when the edge we are adding is a "duplicate" edge with a different length. For example, trying to add both `("A", "B", 1)` and `("A", "B", 2)` to the same graph will result in an `Inv_graph` exception. However, adding `("A", "B", 1)` more than once is allowed. (In this case, the `edges` function should return a list containing `("A", "B", 1)` only once.) `of_edges` behaves similarly and can raise either the `Inv_edge` or the `Inv_graph` exception.

For the implementation, you must use a map to map a vertex to its neighbors together with information about the edge joining them. (This is basically the "ajacency list" representation.)

As an example, if g is bound to `Digraph.of_edges [("B,"C",7); ("A","C",4); ("A","B",5); ("A","C",4)]`, then

- `Digraph.edges g` would return `[("A", "B", 5); ("A", "C", 4); ("B", "C", 7)]`
- `Digraph.vertices g` would return `["A"; "B"; "C"]`
- `Digraph.neighbors "A" g` would return `[("B", 5); ("C", 4)]` (note: the pairs in the list may be in a different order; in this case, the returned list could be `[("C", 4); ("B", 5)]`)
- `Digraph.neighbors "C" g` would return `[]` (edges are directed)

Note that `Digraph.edges` returns the list of edges in asending order of their first vertices. For edges with the same first vertex, they are then in ascending order of their second vertices. Duplicate edges appear only once.

To find the shortest path from one vertex to another, we use Dijkstra's algorithm. Refer to the following for a description of the algorithm:

```
https://en.wikipedia.org/wiki/Dijkstra's_algorithm#Algorithm
```

Pseudocode is also available at:

```
https://en.wikipedia.org/wiki/Dijkstra's_algorithm#Pseudocode
https://en.wikipedia.org/wiki/Dijkstra's_algorithm#Using_a_priority_queue
```

The above pseudocode find the shortest path from a source vertex to all other vertices; they can be modified to find the shortest path to a specific destination.

Note that you can use the OCaml `Set` module to implement a priority queue.

Implement Dijkstra's algorithm using the techniques of functional programming. Put your implementation of the algorithm in `main.ml`. There must be a function named `dijkstra` with signature

```
val dijkstra : string -> string -> Graph.t -> int * string list
```

that takes a starting vertex, a goal vertex, a graph and returns a pair consisting of the length of the shortest path and the actual shortest path (a list of vertexes from the starting vertex to the goal vertex). The function should throw an exception (using `failwith`) if there is no suitable path.

`main.ml` must also contain a function that reads graph data from a file:

```
val read_graph : string -> Graph.t
```

The `read_graph` function expects the name of a file that contains information about a graph. The implementation may assme that each line has at least 3 words, the third of which is an integer. These three items specify an edge (which may or may not be valid). Any words that follow are regarded as comments and are ignored. As an example, assume the file `data` contains:

```
A C 2  # an edge
B  A  4
C  B  6
 C D  5
 C  E  1
 D  E  3
  E B 2
  E F 1
  F D 2
```

Then the first line represents an edge from vertex "A" to vertex "C" of length 2. We can similarly interpret the other lines. Your implementation should call `Digraph.of_edges` which can decide whether the data constitutes a valid graph.

For the above file content, `dijkstra "A" "F" @@ read_graph "data"` returns `(4, ["A"; "C"; "E"; "F"])`, indicating that the shortest path from vertex `"A"` to vertex `"F"` has length 4 and the shortest path goes from vertex `"A"` to vertex `"C"`, then to vertex `"E"` and finally to vertex `"F"`.

We will use `ocamlbuild digraph.cmo` to build `digraph.cmo`. Then we will use `main.ml` inside `utop` (after loading `digraph.cmo`) to test. Make sure this works fine with your implementation.