COMP 3958 Assignment

Instructions:

- Submit a zip file containing the 5 source files `graph.ml`, `graph.mli`, `util.ml`, `util.mli` and `main.ml`.
- 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 no credit for this assignment.
- Your files must build with no 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. Additional instructions will be announced if necessary.

In this assignment, you are asked to implement Prim's algorithm for finding the minimum spanning tree of a connected, weighted and undirected graph.

For our purpose, a graph consists of vertices (labelled by strings) and edges with each edge having a weight (a positive integer) and linking two vertices. Hence there are two strings and an integer associated with an edge: the two vertices it is linking and its weight. In this write-up, all graphs are weighted and undirected, and we will use <v1, v2, w> to denote an edge linking v1 and v2 with weight w. Note that <v1, v2, w> and <v2, v1, w> denote the same edge since edges are undirected.

You are asked first to implement an abstract data type in a module named `Graph` to represent weighted undirected graphs using the "adjacency lists" representation. For this, you must use a map (from the `Map` module) to map each vertex to the list of its neighbouring vertices and their weights. For example, adding the edge <`"A"`, `"B"`, 1> to the graph will add the tuple (`"B"`, 1) to the list associated with the key `"A"` in the map, and since edges are undirected, it will also add the tuple (`"A"`, 1) to the list associated with the key `"B"`. Note that since the module is named `Graph`, its implementation must be in a file named `graph.ml` and its signatures in a file named `graph.mli`. The file `graph.mli` is provided with this assignment — *do not modify it.*

The signature of `Graph` specified in `graph.mli` is basically as follows:

```
type t                            (* the abstract graph type *)
type edge = string * string * int  (* the edge type *)
exception Error of string  (* exception raised by some of the functions *)

val empty : t             (* the empty graph *)

(* adds an edge to a graph
   * raises an exception if edge is invalid (one or both of the vertices
     are empty strings, or the weight is not positive)
   * raises an exception when a conflicting edge is being added e.g.
     empty |> add_edge ("A", "B", 1) |> add_edge ("B", "A", 2)
     or when an edge joining a vertex to itself is being added
   * a "no-op" when the "same" edge is added a second time
*)
val add_edge : edge -> t -> t

(* returns a graph from the specified list of edges; may raise an exception *)
val of_edges : edge list -> t

(* returns the list of all vertices in a graph sorted in ascending order *)
val vertices: t -> string list

(* returns whether a string is the name of a vertex in a graph *)
val is_vertex : string -> t -> bool

(* returns a sorted list of all edges in a graph; see write-up for example *)
val edges : t -> edge list

(* returns sorted list of the neighbours of a specific vertex in a graph;
   each neighbour is represented by a (vertex, weight) pair and the list
   is sorted in ascending order of the neighbour vertices;
   returns an empty list if the given vertex is not in the graph *)
val neighbours : string -> t -> (string * int) list
```

Note that not all functions in `Graph` are used in Prim's algorithm.

Prim's algorithm is used to find the minimum spanning tree given a starting vertex and a graph. Refer to Wikipedia for a description of the algorithm:

`https://en.wikipedia.org/wiki/Prim%27s_algorithm`

The basic idea is as follows:

```
Input: connected undirected weighted graph G and a starting vertex s
Output: edges of a minimum spanning tree of G
X := {s} # singleton set
T := {}  # spanning tree; empty at first
while there is an edge <u, v, w> with u in X, v not in X do
  <u', v', w'> := such an edge with minimum weight w'
  add v' to X  # note: u' is already in X
  add edge <u', v', w'> to T
return T
```

Note that the `Set` module may be useful in your implementation.

Put your implementation of Prim's algorithm in `util.ml`. The corresponding `util.mli` file specifies 2 functions:

```
  val min_tree : string -> Graph.t -> Graph.edge list * int
  val read_data : string -> Graph.t
```

The `min_tree` function returns (the list of edges in) a minimum spanning tree together with its total weight. It takes the starting vertex and a graph as its arguments. It (or a helper function) implements Prim's algorithm. If there is no minimum spanning tree (e.g., the starting vertex is not in the graph), it returns `([], 0)`.

The `read_data` function is used to read information about a graph from a file. It takes the name of a file and returns an abstract graph. (It could raise the `Graph.Error` exception.) It expects each line of the input file to contain at least three words, the third of which is a positive integer. If a line does not satisfy this condition, it is skipped. For example, the content could be

```
 A  B  1
 A  C  4  # the following line is invalid and is skipped
 A  D -4
 A  D  3
 B  D  2
 C  D  5  # the following line is also invalid & is skipped
 E  F
```

In the above, the first line represents an edge linking vertex `"A"` with vertex `"B"` with weight 1. We can similarly interpret the other valid lines. Extra words on each line are regarded as part of a comment.

Assuming that `g` is the graph returned by `Util.read_data` from the above file content and assuming we have loaded the relevant modules, then

- `Graph.vertices g` returns `["A"; "B"; "C"; "D"]`
  Note sorted order.
- `Graph.neighbours "A" g` could return `[("B", 1); ("C", 4); ("D", 3)]`
  Note sorted order.
- `Graph.edges g` returns

  ```
  [("A", "B", 1); ("A", "C", 4); ("A", "D", 3); ("B", "A", 1); ("B", "D", 2);
   ("C", "A", 4); ("C", "D", 5); ("D", "A", 3); ("D", "B", 2); ("D", "C", 5)]
  ```

  The returned list of edges is in ascending order of their first vertices. For edges with the same first vertex, they are then in ascending order of their second vertices.
- `Util.min_tree "A" g` could return `([("A", "C", 4); ("B", "D", 2); ("A", "B", 1)], 7)`
  Note that the minimum spanning tree may not be unique. Note also that since edges are undirected, `("A, "C", 4)` could also be `("C", "A", 4)`, etc.

You will also need to implement a main program (in `main.ml`). It checks for a starting vertex followed by a filename on the command-line, reads the file to create a graph object, calls `Util.min_tree` with the specified starting vertex and graph, and prints the result. Sample output for the above file content:

```
<A, C, 4> <B, D, 2> <A, B, 1> (weight: 7)
```

We will be using ocamlbuild to build your program: the main program using `ocamlbuild main.native`, the 2 other modules using `ocamlbuild graph.cmo` and `ocamlbuild util.cmo`. Make sure these commands work without warnings or errors.