COMP 3958: Assignment 2

Submit a zip files containing your 7 source files: **graph.mli**, **graph.ml**, **prim.mli**, **prim.ml**, **utils.mli**, **utils.ml** and **main.ml**. Submission deadline and instructions will be provided. Your files must build without errors; otherwise, you may receive no credit for the assignment. Note that this assignment is worth 15% of your grade.

In this assignment, you will be implementing an algorithm to find the minimum spanning tree of a connected, weighted and undirected graph using Prim's algorithm.

For our purpose, a graph consists of vertices (labelled by strings) and edges with each edge having a weight (a positive integer) and connecting two vertices. Hence there are 2 strings and an integer associated with an edge: the two vertices it is connecting and its weight. In this write-up, we'll use <v1, v2, w> to denote an edge connecting v1 and v2 with weight w. If we are not interested in the weight, we'll use just <v1, v2>. Note that in this write-up, all graphs are assumed to be weighted and undirected. Hence <v1, v2, w> and <v2, v1, w> are the same edge.

In the previous assignment, you dealt with the distance matrix representation of a graph. For this assignment, you are first asked to implement an abstract data type in a module named `Graph` to represent weighted undirected graphs using the adjacency lists representation. For this, you'll need to use a map (from the `Map` module) to map each vertex to the list of its neighbouring vertices with corresponding weights. For example, adding the edge <"A", "B", 1> to the graph will add the pair ("B", 1) to the list associated with the key "A" in the map (and, since edges are undirected, possibly adding the pair ("A", 1) to the list associated with the key "B"). Note that the implementation should be in a file named **graph.ml** and the corresponding signatures in a file named **graph.mli**.

The signature of `Graph` is as follows (i.e., this is the content of **graph.mli** which you must not modify):

```
type t                            (* the abstract graph type *)
type edge = string * string * int (* the edge type *)

exception Except of string  (* exception raised by some of the functions *)

val inf : int    (* special large value returned by weight function *)

(* the empty graph *)
val empty : t

(* adds an edge to a graph
   * 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_list : edge list -> t

(* returns the list of all vertices in a graph *)
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 list of all edges in a graph *)
val edges : t -> edge list

(* returns a list of all neighbours of a specified vertex in a graph;
   each pair in the list represents (neighbour_vertex, weight);
   returns an list if the vertex is not in the graph *)
val neighbours : string -> t -> (string * int) list
```

```
(* [weight v1 v2 g] returns the weight of an edge between vertices v1 & v2
   in graph g; it assumes that there is at most 1 edge connecting 2 vertices;
   it returns the special value inf if there is no edge between v1 & v2 *)
val weight : string -> string -> t -> int
```

We will use Prim's algorithm 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 here.

Put your implementation of the algorithm in `prim.ml`. The corresponding `prim.mli` file should only contain:

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

i.e., only `min_tree` is "exported". You can have other helper functions in `prim.ml`. The `min_tree` function implements Prim's algorithm. It takes a starting vertex and a graph, and returns a list of the edges in the minimum spanning tree in the form (vertex1, vertex2, weight) if such a tree exists containing the starting vertex. Otherwise, it returns an empty list.

You'll also need a few utiltity functions. Put them in `utils.ml` and `utils.mli`. The `Utils` module must provide at least the following 2 functions:

```
  val sum_weights : Graph.edge list -> int
  val read_graph : string -> Graph.t
```

`sum_weights` just returns the total of all the weights in list of edges.

`read_graph` is used to read information about a graph from a file. It takes the name of a file and returns an abstract graph. It expects each line of the file to contain three words, the third of which is an integer. For example, the content could be

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

In the above, the first line represents an edge connecting vertex `"A"` with vertex `"B"` of weight 1. We can similarly interpret the other lines. For simplicity, you may assume each line has the correct format.

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

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

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

- `Prim.min_tree "A" g` could return `[("A", "B", 1); ("B", "D", 2); ("A", "C", 4)]`

Note that in general the order of the elements in the above lists does not matter. Note also that in this particular implementation of `min_tree`, the returned list contains (`"A"`, `"C"`, 4) but not (`"C"`, `"A"`, 4) although they are the same edge. You can easily implement `min_tree` to contain both if desired.

You'll 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 `Prim.min_tree` with the specified starting vertex on the graph and prints the result. Sample output for the above input content:

```
minimum weight: 7
edges: <A, B, 1> <B, D, 2> <A, C, 4>
```

Note that the minimum weight is the sum of the weights in the edges.

We will be using ocamlbuild to build your program: `ocamlbuild prim.native`. Make sure that this command can build your program. Additional information may be provided in class.