Minimum Spanning Tree (Scheme+Haskell - 7+7 Points)

May 25, 2022

Given a connected weighted graph $G = (V, E)$ with a weight function $w\colon E \to \mathbf{N}$ assigning to each edge e its weight $w(e)$, its minimum spanning tree is a graph $T = (V, E')$ such that $E' \subseteq E$, T is a tree (i.e., a connected graph without cycles) and $\sum_{e \in E'} w(e)$ is minimum possible among such trees. Figure 1 shows an example of a connected weighted graph and its minimum spanning tree.

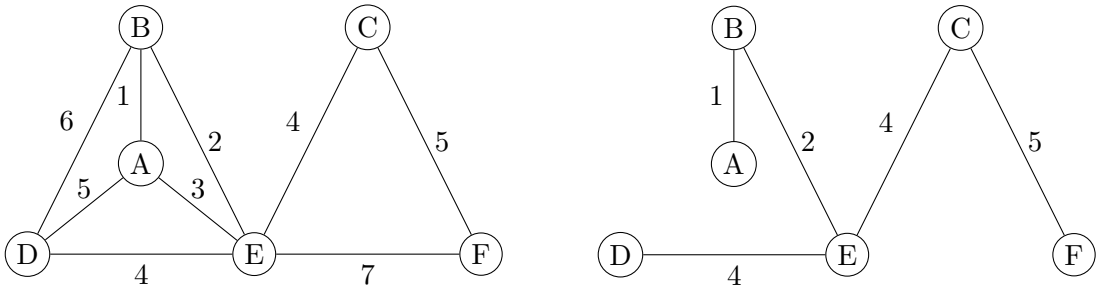

Figure 1: (Left) A connected weighted graph. (Right) Its minimum spanning tree of weight 16.

Your task is to implement an algorithm computing the minimum spanning tree, i.e., a function returning for a given connected weighted graph (V, E) the subset E' of edges in the minimum spanning tree. There are various greedy algorithms computing the minimum spanning tree. You can use, for instance, Jarník's algorithm, whose pseudocode is below.

```
vertices = [list of graph vertices]
edges = [list of graph edges]

covered = [v0] # select an arbitrary vertex as the initial one
tree_edges = []

until covered == vertices:
  find the minimum-weight edge e=(u,v) connecting a vertex u in covered
    with a vertex v not in covered
  add v to covered
  add e to tree_edges

return tree_edges
```

1 Task 3 - Scheme

In Scheme, implement a function (`minimum-spanning-tree gr`) that accepts a connected weighted graph `gr` and returns a list of edges forming the minimum spanning tree. The graph and weighted edges are represented by the following structures:

```
(struct edge (u v weight) #:transparent)
(struct graph (nodes edges) #:transparent)
```

Your file has to be called `task3.rkt` and must provide the function `minimum-spanning-tree` and the above structures so it should start like this:

```
#lang racket
(provide minimum-spanning-tree (struct-out edge) graph)
(struct edge (u v weight) #:transparent)
(struct graph (nodes edges) #:transparent)

; your code goes here
```

Example The graph from Figure 1 is represented as follows:

```
  (define gr (graph '(A B C D E F)
                    (list (edge 'A 'B 1)
                          (edge 'D 'E 4)
                          (edge 'E 'F 7)
                          (edge 'A 'D 5)
                          (edge 'B 'E 2)
                          (edge 'C 'F 5)
                          (edge 'D 'B 6)
                          (edge 'E 'C 4)
                          (edge 'A 'E 3))))

> (minimum-spanning-tree gr)
(list (edge 'C 'F 5) (edge 'E 'D 4) (edge 'E 'C 4)
      (edge 'B 'E 2) (edge 'A 'B 1))
```

Note that the structure `(edge x y w)` represents a bidirectional edge so it can be used in Jarník's algorithm as `(x,y)` and also as `(y,x)`.
The returned list of edges might be ordered arbitrarily. Each edge might be ordered arbitrarily as well. For instance, it does not matter if your output contains `(edge 'C 'F 5)` or `(edge 'F 'C 5)`. However, do not include both variants in your output.

Hint To find the minimum-weight edge, you may want to sort a list of edges by their weight. This can be done by the function `sort` allowing sorting w.r.t. a given comparing function, e.g.,

```
> (sort (list (edge 'a 'b 3) (edge 'b 'c 1) (edge 'c 'a 2))
        (lambda (e1 e2) (< (edge-weight e1) (edge-weight e2))))
(list (edge 'b 'c 1) (edge 'c 'a 2) (edge 'a 'b 3))
```

2 Task 4 - Haskell

In Haskell, we represent the weighted graph and edges by the following types:

```haskell
data Edge a b = Edge { u :: a,
                       v :: a,
                       weight :: b } deriving (Eq,Show)

data Graph a b = Graph { nodes :: [a],
                         edges :: [Edge a b] } deriving Show
```

Implement a function `minSpanningTree :: (Eq a, Ord b) => Graph a b -> [Edge a b]` that accepts a connected weighed graph and returns a list of edges from the minimum spanning tree.

```haskell
  gr :: Graph Char Int
  gr = Graph{ nodes = ['A'..'F'],
              edges = [Edge 'A' 'B' 1,
                       Edge 'D' 'E' 4,
                       Edge 'E' 'F' 7,
                       Edge 'A' 'D' 5,
                       Edge 'B' 'E' 2,
                       Edge 'C' 'F' 5,
                       Edge 'D' 'B' 6,
                       Edge 'E' 'C' 4,
                       Edge 'A' 'E' 3] }

> minSpanningTree gr
[Edge {u = 'C', v = 'F', weight = 5},Edge {u = 'E', v = 'D', weight = 4},
 Edge {u = 'E', v = 'C', weight = 4},Edge {u = 'B', v = 'E', weight = 2},
 Edge {u = 'A', v = 'B', weight = 1}]
```

The returned list of edges might be ordered arbitrarily. Each edge might be ordered arbitrarily as well. For instance, it does not matter if your output contains `Edge {u='C', v='F', weight=5}` or `Edge {u='F', v='C', weight=5}`. However, do not include both variants in your output.
Your file has to be called `Task4.hs` and must export the function `minSpanningTree` and the data types `Graph a b`, `Edge a b` so it should start like this:

```haskell
module Task4 (minSpanningTree, Graph (..), Edge (..)) where
import Data.List -- for sortOn

data Edge a b = Edge { u :: a,
                       v :: a,
                       weight :: b } deriving (Eq,Show)

data Graph a b = Graph { nodes :: [a],
                         edges :: [Edge a b] } deriving Show

-- your code goes here
```

Hint To find the minimum-weight edge, you may want to sort a list of edges by their weight. This can be done by the function `sortOn :: :: Ord b => (a -> b) -> [a] -> [a]` provided by `Data.List`. Below is an example.

```
import Data.List

> sortOn weight [Edge 'A' 'B' 4, Edge 'B' 'C' 3, Edge 'C' 'A' 1]
[Edge {u = 'C', v = 'A', weight = 1},Edge {u = 'B', v = 'C', weight = 3},
 Edge {u = 'A', v = 'B', weight = 4}]
```