# Task 3: Cheap Flights (Haskell - 7 Points)

May 24, 2021

In this task your goal is to find the cheapest flights from one airport to another. You are given an undirected graph that is represented by a list of nodes (airports) and a list of edges (connections from airport to airport). Each edge contains the two nodes that it connects as well as the cost of traveling along the edge. The graph in Fig. **??** is represented as shown in the code snippet below.

```haskell
type Node = Int
type Cost = Float
type Edge = (Node,Node,Cost)
type Graph = ([Node],[Edge])
type Path = [Node]

nodes :: [Node]
nodes = [1..6]

edges :: [Edge]
edges = [(1,2,0.5), (1,3,1.0), (2,3,2.0), (2,5,1.0), (3,4,4.0), (4,5,1.0)]

graph :: Graph
graph = (nodes,edges)
```

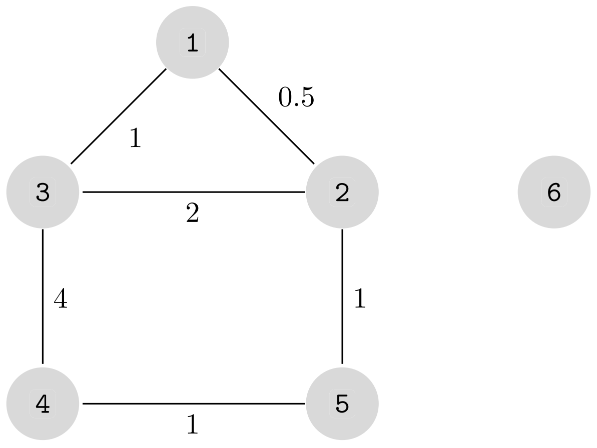

Figure 1: Exemplary graph with 5 nodes. The cost to travel along an edge is written next to the corresponding edge.

A `Path` represents your travel plan. For example, there are three paths from node `2` to node `3`:

```haskell
p1 = [2,3]     -- cost: 2
p2 = [2,1,3]   -- cost: 1.5
p3 = [2,5,4,3] -- cost: 6
```

Write a function `cheapflight :: Node -> Node -> Graph -> Maybe (Path,Cost)` which takes a starting `Node`, a destination `Node`, a `Graph`, and returns the cheapest path from start to destination, as well as the total cost.
Your solution must be in a module called `Task3.hs` and export the defined types as well was the `cheapflight` function. Your file should therefore start like this:

```haskell
module Task3 (cheapflight,Node,Cost,Edge,Graph,Path) where
import Data.List  -- needed for sorting (see hints)

-- ... your code goes here ...
```

## Example

```haskell
-- with the graph defined above:
> cheapflight 2 3 gr
Just ([2,1,3],1.5)

> cheapflight 2 6 gr
Nothing
```

## Hints

In order to find the cheapest path you can modify the breadth-first-search (BFS) algorithm that was discussed in the labs. In the lab we used a queue of partial solutions and extended the shortest path until we found a solution. Here you are not looking for the shortest but the cheapest path. To achieve this you can sort the queue of partial solutions to extend the cheapest path.

For sorting your paths according to the cost you can use the function `sortBy` which is provided by the package `Data.List`:

```haskell
import Data.List

lowcost (_,x) (_,y) | x < y = LT
                    | otherwise = GT

sortBy lowcost [([1,3],2.0) ([1,2,3],0.3)]
-- [([1,2,3],0.3) ([1,3],2.0)]
```