-General Information

*Recommended for use: the*[*GHC documentation*](http://lambda.inf.elte.hu/haskell/docs/latest/libraries/index.html)*, [Hoogle (Haskell API search)](http://lambda.inf.elte.hu/haskell/hoogle/) and*[*list of common functions*](http://lambda.inf.elte.hu/fp/TypeIndex_en.xml)*. Please note that there are tasks that may depend on others, so it is recommended to define the functions in the order of their introduction in the text below. Each function must have exactly the same name and type as it is given, otherwise the solution will not be accepted. Place all the function definitions in a single source file and submit the contents here, in BE-AD. In addition to that, it is highly recommended to use the same source file for the development.*

If you want to use the tests embedded into the description, add the following code snippet to the beginning of your source file:

import Control.DeepSeq

import Control.Exception

import System.IO.Unsafe

import Data.Char

import Data.Either

import Data.Function

import Data.List

raises :: NFData a => a -> String -> Bool

x `raises` s = unsafePerformIO $

either ((s ~=) . f) (const False) `fmap` (try $ evaluate $ force x)

where

f :: SomeException -> String

f = show

(~=) = isPrefixOf `on` (map toUpper . unwords . words)

{-

allTests = (and (concatMap snd tests), tests)

tests =

[ ("test\_intersections", test\_intersections)

, ("test\_outNeighbors", test\_outNeighbors)

, ("test\_step", test\_step)

, ("test\_steps", test\_steps)

, ("test\_getRoutesWithEnding", test\_getRoutesWithEnding)

, ("test\_hasRouteWithEnding", test\_hasRouteWithEnding)

, ("test\_initRoutes", test\_initRoutes)

, ("test\_routes", test\_routes)

, ("test\_routesFromPizzerias", test\_routesFromPizzerias)

, ("test\_optimalRoute", test\_optimalRoute)

]

-}

Overview

A pizzeria in Budapest tries to optimize its pizza delivery service. It does so by using shortest routes possible. They asked us to design and implement a navigation service.

Map of Budapest is represented by a graph. Intersections are graph nodes, and streets between intersections are edges between nodes. Edges are directed, meaning that from *a* we can reach *b* by an edge from *a* to *b* but we cannot reach *a* from *b* on the same edge.

Graph representation

Graph (called Map because it represents the map of Budapest) is a list of Streets. A Street consists of a starting and an ending Intersections. Intersections are represented by integers.

type Intersection = Int

type Street = (Intersection, Intersection)

type Map = [Street]

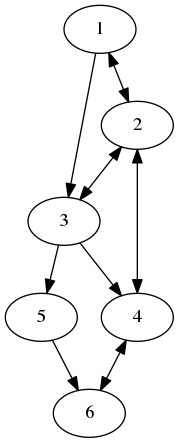
With the help of the type keyword, Int will be called Intersection, and (Int, Int) will be called Street, and list of them will be called Map(which is going to be the type for the map) for the enhanced readibility in the types.

Sample graphs

Let's introduce the following two graphs for testing:

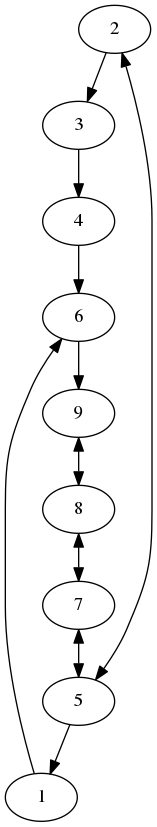
map1 :: Map

map1 = [(1,2),(2,1),(1,3),(3,4),(3,5),(4,6),(6,4),(5,6),(2,3),(3,2),(2,4),(4,2)]



map2 :: Map

map2 = [(2,3),(3,4),(5,1),(1,6),(8,7),(7,8),(9,8),(8,9),(5,2),(2,5),(7,5),(5,7),(4,6),(6,9)]



List of intersections (3 points)

Define a function that lists every intersection in a map. The result has to be free of duplicates and in an ascending order.

*Hint*: you can use nub and sort for removing duplicates and sorting. They are in [Data.List](http://lambda.inf.elte.hu/haskell/docs/latest/libraries/base-4.9.1.0/Data-List.html).

intersections :: Map -> [Intersection]

Test cases:

test\_intersections = [

intersections [] == []

, intersections [(2,3), (3,2)] == [2,3]

, intersections [(2,3), (1,2), (3,1)] == [1,2,3]

, intersections map1 == [1,2,3,4,5,6]

, intersections map2 == [1,2,3,4,5,6,7,8,9]

]

Reachable intersections (3 points)

Define a function that returns intersections that are reachable from a specific intersection. In terms of graphs, this function returns neighbors of a node.

outNeighbors :: Map -> Intersection -> [Intersection]

Test cases:

test\_outNeighbors = [

sort (outNeighbors map1 1) == [2,3]

, sort (outNeighbors map1 3) == [2,4,5]

, sort (outNeighbors map1 5) == [6]

, sort [outNeighbors map1 i | i <- intersections map1] == [[1,3,4],[2,3],[4],[4,5,2],[6],[6,2]]

, sort [outNeighbors map2 i | i <- intersections map2] == [[1,2,7],[3,5],[4],[6],[6],[7,9],[8],[8,5],[9]]

]

Routes to customers

Since we are calculating routes to customers, it may be a good idea to introduce type Route. It is a non-empty list of intersections.

type Route = [Intersection]

*Important*: Routes are traversed from right to left (from last to first element). That is, starting point is the last element. For example, route [4,3,2,1]starts in intersection 1 and ends in intersection 4:

https://bead.inf.elte.hu/files/functionallanguages/FL_20180525_path.png

Continuations of a route (4 points)

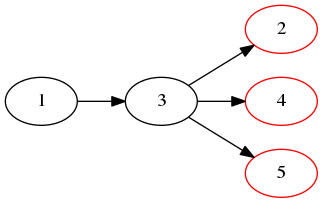
When we can choose multiple directions in an intersection, we construct a route for each choice.

Define a function that extends a route with all possible continuations.

Take care not to extend a route with an intersection that is already part of the route. That is, do not introduce cycles.

step :: Map -> Route -> [Route]

*Hint*: when route is [3,1] then we can extend it with intersections 2, 4 and 5 (drawn in red below) according to map1:



So the result consists of three routes: [[2, 3, 1], [4, 3, 1], [5, 3, 1]].

https://bead.inf.elte.hu/files/functionallanguages/FL_20180525_route1.png

https://bead.inf.elte.hu/files/functionallanguages/FL_20180525_route2.png

https://bead.inf.elte.hu/files/functionallanguages/FL_20180525_route3.png

Test cases:

test\_step = [

sort (step [(1,2)] [2]) == []

, sort (step map1 [2]) == [[1,2],[3,2],[4,2]]

, sort (step map1 [3]) == [[2,3],[4,3],[5,3]]

, sort (step map1 [1, 2]) == [[3,1,2]]

, sort (step map1 [3, 2]) == [[4,3,2],[5,3,2]]

, sort (step map1 [4, 2]) == [[6,4,2]]

]

Continuations of several routes (2 points)

Define a function that extends routes. It extends each with all possible continuations.

*Hint*: you may want to use concat here to collect all extensions into a single list. This function can also be defined without concat.

steps :: Map -> [Route] -> [Route]

Test cases:

test\_steps = [

sort (steps map1 [[1]]) == [[2,1],[3,1]]

, sort (steps map1 [[2, 1], [3, 1]]) == [[2,3,1],[3,2,1],[4,2,1],[4,3,1],[5,3,1]]

, sort (steps map1 [[3, 2, 1], [4, 2, 1], [4, 3, 1], [5, 3, 1], [2, 3, 1]]) == [[2,4,3,1],[4,2,3,1],[4,3,2,1],[5,3,2,1],[6,4,2,1],[6,4,3,1],[6,5,3,1]]

, sort (steps map1 [[4, 3, 2, 1], [5, 3, 2, 1], [6, 4, 2, 1], [6, 4, 3, 1], [2, 4, 3, 1], [6, 5, 3, 1], [4, 2, 3, 1]]) == [[4,6,5,3,1],[6,4,2,3,1],[6,4,3,2,1],[6,5,3,2,1]]

, sort (steps map2 (steps map2 (step map2 [4]))) == [[8,9,6,4]]

, sort (steps map2 (steps map2 (step map2 [9]))) == [[5,7,8,9]]

, sort (steps map2 (steps map2 (step map2 [5]))) == [[4,3,2,5],[9,6,1,5],[9,8,7,5]]

]

Calculating routes

In this part, we are going to find the shortest route (or routes, there could be many) from intersection *a* to *b*.

Filtering routes that end in an intersection (2 points)

Define a function that returns routes that end in a given intersection from a list of routes.

*Hint*: Don't forget that ending point is the first element in Route and that Routes are never empty.

getRoutesWithEnding :: Intersection -> [Route] -> [Route]

Test cases:

test\_getRoutesWithEnding = [

sort (getRoutesWithEnding 6 []) == []

, sort (getRoutesWithEnding 1 [[1]]) == [[1]]

, sort (getRoutesWithEnding 1 [[2,1]]) == []

, sort (getRoutesWithEnding 2 [[3,2,1]]) == []

, sort (getRoutesWithEnding 6 [[4,3,2,1],[5,3,2,1],[6,4,2,1],[6,4,3,1],[2,4,3,1],[6,5,3,1],[4,2,3,1]]) == [[6,4,2,1],[6,4,3,1],[6,5,3,1]]

, sort (getRoutesWithEnding 6 [[4,3,2,1],[3,2,1],[4,2,1],[4,3,1],[4,3,1],[5,3,1],[4,2,3,1]]) == []

]

Is there a route that ends in an intersection? (1 point)

Define a function that returns True when there is a route among list of routes that ends in an intersection.

hasRouteWithEnding :: Intersection -> [Route] -> Bool

Test cases:

test\_hasRouteWithEnding = [

hasRouteWithEnding 3 [] == False

, hasRouteWithEnding 3 [[1, 2], [3, 2], [4, 2]] == True

, hasRouteWithEnding 3 [[2, 1], [3, 1]] == True

, hasRouteWithEnding 2 [[2, 1], [3, 1]] == True

, hasRouteWithEnding 5 [[2, 1], [3, 1]] == False

, hasRouteWithEnding 1 [[2, 1], [3, 1]] == False

]

Initial route (2 points)

Define a function that contructs a list of routes from a starting intersection. The result should be a list with only one route. This route has only one intersection (the given one).

When the intersection is not found on the map, the function should signal an error using function error.

initRoutes :: Map -> Intersection -> [Route]

Test cases:

test\_initRoutes = [

initRoutes map1 1 == [[1]]

, initRoutes map1 4 == [[4]]

, initRoutes map2 7 == [[7]]

, initRoutes map1 7 `raises` "initRoutes: invalid intersection node"

]

Shortest paths (5 points)

Define a function that returns all shortest paths from an intersection to another.

Idea:

* Repeatedly use steps starting from the initial intersection until at least one route ends in the target intersection (hasRouteWithEnding).
* Select routes that ends in the target intersection (getRoutesWithEnding). Only some of the routes actually end in the target intersection. The rest should be dropped.

*Hints*:

* We presume that graphs are connected, so sooner or later we reach the target intersection.
* The method above is basically a breadth-first search.

routes :: Map -> Intersection -> Intersection -> [Route]

Test cases:

test\_routes = [

routes map1 2 2 == [[2]]

, routes map1 1 5 == [[5,3,1]]

, routes map1 5 3 == [[3,2,4,6,5]]

]

Optimal route

Let's introduce type Pizzeria. This denotes intersections where a pizzeria is located.

type Pizzeria = Intersection

In this part, we are going to find the closest pizzeria to a customer.

Routes from every pizzerias (2 points)

Define a function that constructs shortest routes from all pizzerias to a customer.

routesFromPizzerias :: Map -> [Pizzeria] -> Intersection -> [Route]

Test cases:

test\_routesFromPizzerias = [

sort (routesFromPizzerias map1 [1] 4) == [[4,2,1],[4,3,1]]

, sort (routesFromPizzerias map1 [4,5] 6) == [[6,4],[6,5]]

, sort (routesFromPizzerias map1 [2,5] 4) == [[4,2],[4,6,5]]

, sort (routesFromPizzerias map1 [2,1,5] 4) == [[4,2],[4,2,1],[4,3,1],[4,6,5]]

, sort (routesFromPizzerias map1 [1] 6) == [[6,4,2,1],[6,4,3,1],[6,5,3,1]]

]

An optimal route (4 points)

Given a non-empty list of pizzerias and a customer, define the function that finds the shortest route from one of the pizzerias to the customer. This route should consists of the least number of intersections.

When there are many shortest routes, select the first one.

*Hint*: calculate length of routes from each pizzeria and select routes that have minimum length. Then return the first route (presumably it exists).

optimalRoute :: Map -> [Pizzeria] -> Intersection -> Route

Test cases:

test\_optimalRoute = [

optimalRoute map1 [1,6] 3 == [3,1]

, optimalRoute map1 [1,6] 4 == [4,6]

, [optimalRoute map1 [1,6] j | j <- intersections map1] == [[1],[2,1],[3,1],[4,6],[5,3,1],[6]]

, [optimalRoute map1 [1,3] j | j <- intersections map1] == [[1],[2,1],[3],[4,3],[5,3],[6,4,3]]

, [optimalRoute map2 [1,3] j | j <- intersections map2] == [[1],[2,5,7,8,9,6,1],[3],[4,3],[5,7,8,9,6,1],[6,1],[7,8,9,6,1],[8,9,6,1],[9,6,1]]

]