UNIVERSIDADE FEDERAL DE SANTA MARIA
Curso de Ciência da Computação e Sistemas de Informação

Disciplina de Paradigmas de Programação
Implementação de Funções sobre Grafos
Professora: Juliana Kaizer Vizzotto

0.1 Introdução: Grafos Diretos

Um grafo direto é formado por uma coleção de *nodos*, cada um contendo um identificador associado, e um conjunto de *arcos* (ligações), tal que cada arco conecta um nodo origem a um dado nodo destino (ambos especificados por seus identificadores). Como um exemplo, os nodos podem representar cidades e os arcos podem representar rodovias conectando tais cidades. Um *grafo com peso* associa um peso numérico à cada arco, o qual pode ser utilizado, por exemplo, para representar a distância, tempo, ou custo de mover de uma cidade para outra. O objetivo deste exercício é que você implemente uma *suite* de funções para manipular grafos e como desafio implementar um algoritmo que calcule o *menor caminho* entre dois nodos do grafo.

0.2 Grafos Diretos em Haskell

Grafos diretos podem ser representados de diversas maneiras em Haskell. Neste exercícios, vamos assumir que ambos os identificadores de nodos e de arcos são números inteiros. Um arco será então um par de identificadores, o primeiro elemento identificando o nodo origem e o segundo o nodo destino. Um grafo será representado por uma lista de pares `( arcos,pesos)`, então:

```
type Id = Int
type Weight = Int
type Edge   = (Id,Id)
type Graph  = [(Edge,Weight)]
```

Vamos assumir arbitrariamente que a origem de todos os nodos será um nodo com *label 0*.

O objetivo deste exercício será calcular o *menor caminho* entre um dado nodo e o nodo `0`. *O custo de um caminho é a soma dos pesos de cada arco no caminho*. Entretanto, se não existe uma maneira de atingir determinado nodo a partir da origem, então seu custo será marcado como infinito. Para fazer isto, define-se um novo tipo de dado algébrico que distingue infinito dos outros custos:

```
data Cost = Finite Weight | Infinity
            deriving (Eq, Ord, Show)
```

Finalmente, o custo mínimo de atingir um dado nodo a partir do nodo `0` será representado pelo par:

```
type PathCost = (Cost, Id)
```

0.3 O que fazer?

A primeira etapa constituí-se da implementação das seguintes funções (use o template que está disponível no site):

1. Defina a função `edges ::  Graph -> Int` a qual retorna o número total arcos em um dado grafo.

2. Defina a função `weightTotal :: Graph -> Int`, a qual retorna o peso total de cada arco em um dado grafo.

3. Defina a função `nodes :: Graph -> [Id]`, a qual retorna a lista dos identificadores de nodos em um dado grafo. A lista não deve conter números duplicados. Dica: examine os identificadores de ambos origem e destino de um arco, pois pode ocorrer que um nodo tenha um único arco, somente partindo ou chegando.

4. Defina a função `addCosts :: Cost -> Cost -> Cost` que adicionará dois valores do tipo custo.

5. Defina a função `lookUp :: Edge -> Graph -> Cost` a qual procura pelo custo de um dado arco em um dado grafo.

Para a segunda etapa você deve resolver:

- Defina a função `allPaths :: Graph -> [PathCost]`, a qual retorna uma lista de pares da forma `(c,i)`, onde `i` é o identificador do nodo (destino) e `c` é o custo do menor caminho do nodo 0 até o nodo `i`. Para o Grafo `e1` no template, o resultado deve conter: `(Finite 3,0), (Finite 1,1), (Finite 2,2), (Finite 3, 3), (Finite 5, 4)` e `(Infinity, 5)` em alguma ordem. Como um exemplo, o menor caminho do `0` até `4` é obtido atravessando o caminho `0-1-2-3-4`, com custo `5`, em preferência a rota `0-4` a qual tem custo `6`.

- O **Algoritmo de Dijkstra** resolve este problema da seguinte maneira:

  – A função `allPaths` começa utilizando as funções `lookUp` e `nodes` para construir uma lista contendo o custo de mover do nodo `0` para qualquer outro nodo (incluindo o `0`) em *um* passo (i.e, atravessando somente um arco). Se um nodo não pode ser alcançado a partir do nodo `0` em um passo, o custo deve ser marcado como `Infinity`. Esta lista inicial tem o tipo `[PathCost]`.

  – Agora você precisa de uma função auxiliar (coloque um nome com significado nesta função, não chame de auxiliar), a qual recebe esta lista inicial e gera a partir dela a solução requerida.

  – Assumindo que o argumento da função auxiliar é chamado `ps`, ela deve fazer o seguinte:

  – Procure o elemento de `ps` (`(cmin,j)`) com o menos custo e chame ele de `minp`.

  – `minp` (i.e, a cabeça) faz parte da solução requerida. Em outras palavras o menor caminho do nodo `0` até o nodo `j` é `cmin`. O resto da solução é encontrada aplicando uma função de *relaxamento* a cada um dos elementos de `ps` que restam depois que `minp` é removido dela. A função auxiliar é então aplicada recursivamente na lista resultante. O processo completa quando `ps` está vazia.

  – A função de *relaxamento* pega um par `(c,i)` e gera um novo par `(c',i)` onde `c'` é o mínimo de `c` (o menor caminho de `0` até `i`) e o custo de pegar um rota alternativa de `0` até `j` e então um único arco de `j` até `i` (se ela existe). A função de *relaxamento* requer um `lookUp` no grafo original e o uso de `addCost`. Note que se não existe um arco entre `i` e `j` o custo associado será `Infinity` e o par `(c,i)` será retornado sem modificação.