Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores
Algoritmos e Estruturas de Dados
Semestre de Verão 2021/22
Terceira série de problemas

Observações:

- Data de entrega: **17 de Junho de 2022**.
- **Podem ser utilizadas as estruturas de `java.util` ou `kotlin.collections` no Problema.**
- **Para a função edgesIterator da pergunta 2 da primeira parte da série terão de ser desenvolvidos e entregues testes unitários.**

1 Exercícios

1. Realize a classe `TreeUtils`, contendo as seguintes funções:

   1.1. A função

   ```
   fun <E> equalTrees(root1: Node<E>?, root2: Node<E>?): Boolean
   ```

   que retorna `true` se e só se as árvores binárias de pesquisa com raíz `root1` e `root2` são iguais.

   1.2. A função

   ```
   fun kSmallest(root: Node<Int>?, k: Int): Int?
   ```

   que retorna o k-ésimo menor elemento presente na árvore binária de pesquisa com raíz `root`, ou `null` caso não exista.

   1.3. A função

   ```
   fun <E> isBalanced(root: Node<E>?): Boolean
   ```

   que verifica se a árvore binária, referenciada por `root`, é balanceada.

   Para as implementações destas funções, considere que o tipo `Node<E>` tem 3 propriedades: um `value` e duas referências `left` e `right`, para os descendentes respetivos.

2. Pretende-se realizar uma implementação do tipo de dados abstratos `graph`, que representa um grafo. Este tipo de dados é parametrizado pelos tipos genéricos `I` (tipo do identificador do vértice) e `D` (tipo dos dados associados ao vértice). A interface do tipo de dados `Graph` é da seguinte forma em Kotlin:

```
interface Graph<I, D>{
   interface Vertex<I, D> {
        val id: I
        val data: D
        fun setData(newData: D): D
        fun getAdjacencies(): MutableSet<Edge<I>?>
    }
    interface Edge<I> {
        val id: I
        val adjacent: I
    }
    val size: Int
    fun addVertex(id: I, d: D): D?
    fun addEdge(id: I, idAdj: I): I?
    fun getVertex(id: I): Vertex<I, D>?
    fun getEdge(id: I, idAdj: I): Edge<I>?
    operator fun iterator(): Iterator<Vertex<I, D>>
    fun edgesIterator(): Iterator<Edge<I>>
}
```

Os componentes da interface `Graph` têm a seguinte funcionalidade:

- interface `Vertex`: representa um vértice de um grafo composto por um identificador e outros dados associados. Através da função `setData`, é possível atribuir novos dados ao vértice;
- função `getAdjacencies`: devolve um conjunto com todas as arestas adjacentes de um vértice (a ordem é indiferente). Se não existir nenhuma aresta devolve um conjunto vazio;
- interface `Edge`: representa uma aresta adjacente a um vértice. Uma aresta é composta pelo identificador do vértice origem e do vértice destino (adjacente);
- propriedade `size`: armazena o número de vértices existentes no grafo;
- função `addVertex`: adiciona um novo vértice ao grafo indicando o seu identificador e dados. Se o vértice já existir, devolve `null` sem adicionar, caso contrário, devolve os dados do vértice;
- função `addEdge`: adiciona uma nova aresta ao grafo indicando os identificadores dos vértices origem e destino (adjacente). Se o vértice origem ainda não existe devolve `null` sem adicionar, caso contrário, devolve o identificador do vértice adjacente;
- função `getVertex`: obtém um vértice dado o seu identificador. Se este não existir devolve null;
- função `getEdge`: obtém uma aresta dados os identificadores do vértice origem e destino (adjacente). Se esta não existir devolve null;
- função `iterator`: retorna um objeto `Iterator<Graph.Vertex<I,V>>` que permite a iteração dos vértices existentes no grafo.
- função `edgesIterator`: que retorna um objeto `Iterator<Edge<I>>` que permite a iteração das arestas existentes no grafo

2 Problema: A importância dos indivíduos nas redes sociais - degree e closeness

Durante a última década tem existido um aumento do interesse no estudo e na extração de informação de redes complexas, tais como as redes sociais *Twitter* e *Facebook*. Estas redes são normalmente apresentadas sob a forma de grafos, nos quais os indivíduos são representados por vértices e os arcos correspondem a relações existentes entre dois indivíduos presentes na rede. Por exemplo, no caso do Twitter, a relação de seguidor (*follower*) é uma das relações que pode ser considerada para definir um arco, mas também a existência de retweet poderá ser considerada para definir um arco, caso se pretenda analisar a rede de influência.

De modo a compreender melhor estas redes, por exemplo para efeitos de publicidade, são calculadas sobre as mesmas algumas métricas que permitem identificar vértices ou facilitar na localização de subgrupos.

Uma medida de centralidade normalmente utilizada é a do *Degree (Grau)* de um vértice, isto é, o número de arcos que estão directamente conectados ao vértice.

A medida de centralidade *Closeness* é também uma medida que nos dá uma informação relevante, pois reflete o quanto um indivíduo está próximo dos outros indivíduos presentes na rede. Por definição, a medida de centralidade *closeness* corresponde ao cálculo, para cada indivíduo, do inverso da média do comprimento dos caminhos mais curtos entre o indivíduo e os restantes indivíduos presentes na rede e atingíveis pelo mesmo.

O problema é descrito por:

- um conjunto I de n indivíduos;
- uma lista L de ligações entre indivíduos, em que cada ligação é descrita por um par composto pela identificação de dois indivíduos.

O objetivo deste trabalho é portanto a realização de um programa que determine:

- o cálculo do *Degree* para cada indivíduo pertencente à rede social;
- o cálculo da *smallestDistance* entre cada par de indivíduos pertencente à rede social;
- o cálculo da medida *Closeness* para cada indivíduo pertencente à rede social.

Parâmetros de execução

Para iniciar a execução da aplicação a desenvolver, terá de executar: `kotlin CentralityKt fileName` Durante a sua execução, a aplicação deverá processar os seguintes comandos:

1. `degree vertexId` que retorna o grau o vértice identificado por `vertexId`;
2. `smallestDistance vertexId1 vertexId2` que retorna o valor da menor distância entre o par de vértices identificados por `vertexId1` e `vertexId2`;
3. `closeness vertexId` que retorna o valor da métrica closeness para o vértice identificado por `vertexId`;
4. `degrees` que retorna o grau de todos os vértices;
5. `closeness` que retorna o valor da métrica closeness para todos os vértices.

Exemplo

Considere que do ficheiro de entrada, são inferidos os seguintes arcos e que o grafo é não orientado **(note que quando é não orientado, não é necessário estar no ficheiro a ligação contrária)**:

```
1 -> 2
2 -> 3
3 -> 5
5 -> 4
4 -> 3
5 -> 2
4 -> 1
6 -> 7
6 -> 8
6 -> 9
9 -> 10
10 -> 6
```

Então,

```
degree(1) = 2
degree(2) = 3
degree(3) = 3
degree(4) = 3
degree(5) = 3
degree(6) = 4
degree(7) = 1
degree(8) = 1
degree(9) = 2
degree(10) = 2
```

closeness(1) = $\frac{1}{1+2+1+2}$ = $\frac{1}{6}$ = 0,167
closeness(2) = $\frac{1}{1+1+2+1}$ = $\frac{1}{5}$ = 0,2
closeness(3) = $\frac{1}{2+1+1+1}$ = $\frac{1}{5}$ = 0,2
closeness(4) = $\frac{1}{1+2+1+1}$ = $\frac{1}{5}$ = 0,2
closeness(5) = $\frac{1}{2+2+1+1}$ = $\frac{1}{6}$ = 0,167
closeness(6) = $\frac{1}{1+1+1+1}$ = $\frac{1}{4}$ = 0,25
closeness(7) = $\frac{1}{1+2+2+2}$ = $\frac{1}{7}$ = 0,143
closeness(8) = $\frac{1}{1+2+2+2}$ = $\frac{1}{7}$ = 0,143
closeness(9) = $\frac{1}{1+2+2+1}$ = $\frac{1}{6}$ = 0,167
closeness(10) = $\frac{1}{1+2+2+1}$ = $\frac{1}{6}$ = 0,167

Relatório

O trabalho realizado deverá ser acompanhado de um relatório, que deverá incluir a avaliação experimental dos algoritmos desenvolvidos. Podem utilizar a classe `NumberFormat` para formatarem os resultados.

Dados

Deverão usar os dados de redes não orientadas presentes em `https://snap.stanford.edu/data/` **e interpretar o estudo realizado. Note que cada rede tem um formato próprio.**

Relatório

O trabalho realizado deverá ser acompanhado de um relatório, que deverá incluir a avaliação experimental e análise do espaço ocupado em memória.

ISEL, 16 de Maio de 2022.