Departamento de Engenharia Eletrónica e de Telecomunicações e Computadores

Licenciatura em Engenharia Informática e de Computadores

Algoritmos e Estruturas de Dados

3ª Série de Exercícios

Alun@s:
Pedro [redacted]
Roberto

Docentes:
Cátia Vaz

Junho 2022

Conteúdo

1 Introdução

Neste relatório iremos apresentar a nossa solução para o problema do trabalho prático 3. Neste problema é necessário utilizar estruturas de dados como árvores e grafos para a sua implementação. Árvores são uma estrutura de dados que representa uma estrutura hierárquica que contém nós. Cada nó está conectado a um parente superior e pode estar conectado a nenhum ou vários filhos. Quando se trata de uma árvore binária, cada nó está limitado a ter apenas 2 filhos.

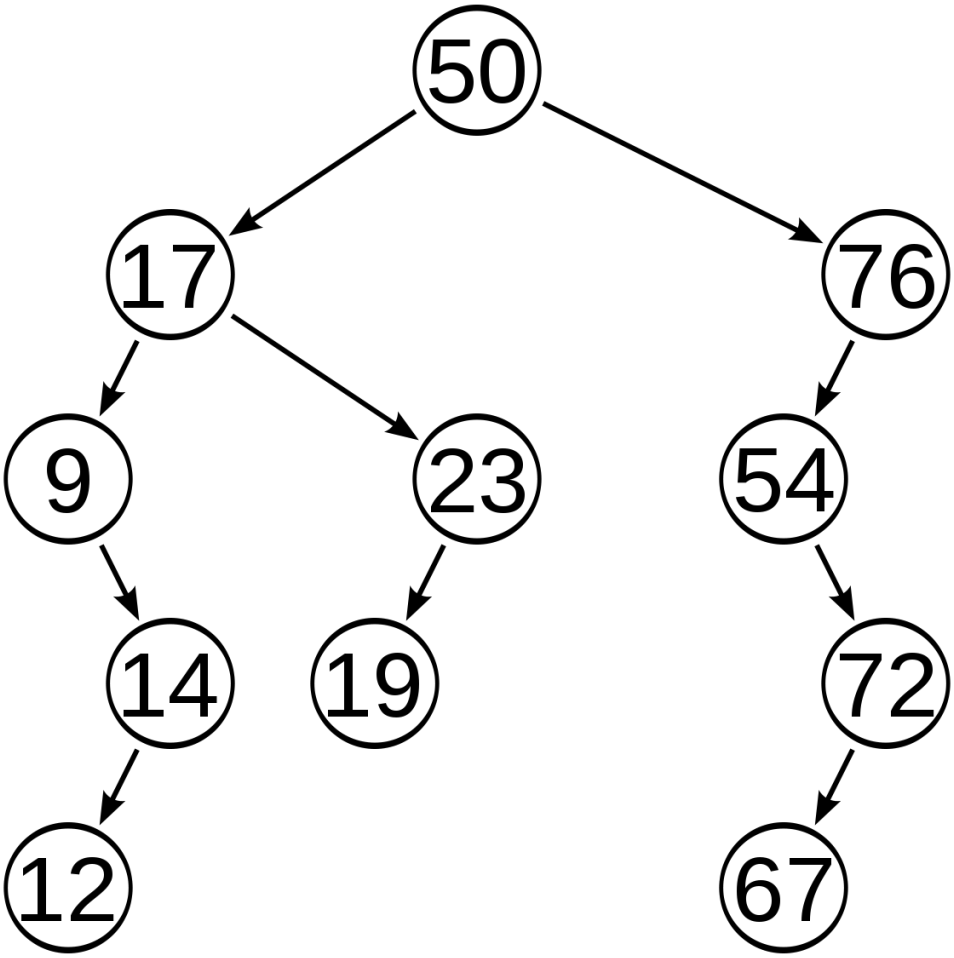

Figura 1: Exemplo de uma estrutura de uma árvore BST

2 Problema: A importância dos indivíduos nas redes sociais

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.

2.1 Paramêtros de execução

Para iniciar a execução da aplicação a desenvolver, terá de executar:

```
kotlin CentralityKt fileName
```

2.2 Comandos

Durante a sua execução, a aplicação deverá processar os seguintes comandos:

- `degree vertexId` : retorna o grau o vértice identificado por vertexId;
- `smallestDistance vertexId1 vertexId2` : retorna o valor da menor distância entre o par de vértices iden- tificados por vertexId1 e vertexId2;

- `closeness vertexId` : retorna o valor da métrica closeness para o vértice identificado por vertexId;
- `degrees` : retorna o grau de todos os vértices;
- `closeness` : retorna o valor da métrica closeness para todos os vértices.

2.3 Formato do ficheiro

Claro que, para construirmos a estrutura do grafo com o programa, o ficheiro tem que ter um input válido, portanto, decidimos em grupo que o formato seria o seguinte:

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

Em que a coluna um é o grafo de partida, e a coluna dois o de chegada. Portanto a leitura e criação da estrutura é feita da seguinte maneira:

```
val graph = GraphStructure<String, String>()

//...

File(args[0]).readLines().forEach { line ->
    val values = line.split(" ")
    val (from, to) = values[0] to values[1]
    graph.addVertex(from, from)
    graph.addVertex(to, to)
    graph.addEdge(from, to)
}
```

3 GraphStructure

Para a resolução do problema, tivemos que utilizar a class `GraphStructure` anteriormente implementada, através da interface `Graph`. Esta class ajuda-nos a construir um grafo, contendo assim várias funções utéis e também a possibilidade de genéricos. A sua implementação encontra-se no ficheiro `GraphStructure.kt`.

```
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>>
}
```

4 Solução

4.1 Leitura dos comandos

Para a leitura dos comandos, simplesmente iteramos infinitamente até o utilizador utilizar o comando `exit`.

```
do {
    println(); print("command: ")
    val fullCommand = readln().split(" ")
    when (fullCommand[0]) {
        "degree" -> cmdDegree(fullCommand.elementAtOrNull(1))
        "smallestDistance" -> cmdSmallestDistance(
            fullCommand.elementAtOrNull(1),
            fullCommand.elementAtOrNull(2)
        )
        "closeness" -> cmdCloseness(fullCommand.elementAtOrNull(1))
        "degrees" -> cmdDegrees()
        "exit" -> break
        else -> println("Command ${fullCommand[0]} not found!")
    }
} while (true)
```

4.2 Degree

O cálculo do `degree` foi feito apartir da função `getDegree` da class `GraphStructure`, implementamos a função como membro da class pois achamos o mais adequado.

```
// GraphStructure.kt : GraphStructure
fun <I, D> getDegree(node: Graph.Vertex<I, D>)
    = node.getAdjacencies().size

// Centrality.kt
fun cmdDegree(vertexId: String?) {
    if (vertexId == null) {
        println("Usage: degree vertexId")
        return
    }

    val v = graph.getVertex(vertexId)
    if (v == null) {
        println("Vertex with vertexId $vertexId not found")
        return
    }

    println("degree($vertexId) = ${graph.getDegree(v)}")
}
```

4.3 Degrees

O mesmo que o comando `degree`, mas para todos os pontos:

```
fun cmdDegrees() {
    for (v in graph)
        println("degree(${v.id}) = ${graph.getDegree(v)}")
}
```

4.4 Closeness

O objetivo deste comando é percorrer todos os vértices e ir somando com a sua profundidade. Para tal, o algoritmo que desenvolvemos utiliza uma lista dos vértices já visitados (`visited`), uma lista dos próximos pontos a visitar (`queue`) e duas variáveis que vão dar jeito para guardar a soma e a profundidade atual (`sum` e `depth`).

```
fun cmdCloseness(vertexId: String?) {
    if (vertexId == null) {
        for (v in graph) cmdCloseness(v.id)
        return
    }

    val v = graph.getVertex(vertexId)
    if (v == null) {
        println("Vertex with vertexId $vertexId not found")
        return
    }

    val visited = mutableListOf(v) // visited nodes
    val queue = mutableListOf<GraphVertices>() // queue to visit nodes
    var depth = 1 // current depth
    var sum = 0 // sum of each node and its depth
```

Portanto, começamos por iterar as adjacências do ponto atual e adicionamos à `queue` caso já não tenha sido visitado para mais tarde visitar. Obviamente que temos que ir somando a profundidade atual e após adicionar os próximos a visitar, temos que a incrementar.

```
    for (e in v.getAdjacencies()) {
        if (e == null) continue
        val a = graph.getVertex(e.adjacent)!!
        if (visited.contains(a)) continue
        queue.add(a)
        sum += depth
    }
    depth++
```

Agora, temos que visitar todos os pontos da queue e fazendo a mesma lógica que no ponto 1, neste caso utilizamos uma lista nova `toAdd` que serve para sabermos que vértice adicionar depois à queue (isto porque o Kotlin não nos deixa mexer numa lista que estamos a iterar).

```
    // keeps track of what vertices should be added
    // to the queue after calculating actual node depth
    val toAdd = mutableListOf<GraphVertices>()
    toAdd.addAll(queue)

    while (queue.isNotEmpty()) {
        queue.forEach { node ->
            visited.add(node)
            for (e in node.getAdjacencies()) {
                if (e == null) continue
                val a = graph.getVertex(e.adjacent)
                if (a == null || visited.contains(a) ||
                    queue.contains(a) || toAdd.contains(a))
                        continue
                toAdd.add(a)
                sum += depth
            }
        }
        queue.addAll(toAdd)
        queue.removeAll(visited)
        depth++
    }

    /**
     * Rounds [this] number with [decimals] places.
     */
    fun Double.round(decimals: Int)
        = "%.${decimals}f".format(this).toDouble()

    val closeness = (1 / sum.toDouble()).round(3)
    println("closeness(${v.id}) = $closeness")
}
```

4.5 Smallest distance

A implementação da função `smallestDistance` acaba por ser muito parecida à da `closeness`, porém, em vez de ir somando a depth a cada iteração por cada aresta, só somamos enquanto temos valores na queue para iterar e só somamos a profundidade uma vez.

```
fun cmdSmallestDistance(vertexId1: String?, vertexId2: String?): Int {
    if (vertexId1 == null || vertexId2 == null) {
```

```
    println("Usage: smallestDistance vertexId1 vertexId2")
    return 0
}

val (v, v2) = graph.getVertex(vertexId1) to graph.getVertex(vertexId2)
if (v == null || v2 == null) {
    println("Vertex with vertexId $vertexId1 or $vertexId2 not found")
    return 0
}

val visited = mutableListOf(v) // visited nodes
val queue = mutableListOf<GraphVertex>() // queue to visit nodes
var depth = 1 // current depth
var sum = depth // sum of each node and its depth

for (e in v.getAdjacencies()) {
    if (e == null) continue
    val a = graph.getVertex(e.adjacent)!!
    if (visited.contains(a)) continue
    queue.add(a)
    if (a == v2) {
        println("smallestDistance(${v.id}, ${v2.id}) = $sum")
        return sum
    }
}
sum += ++depth

// keeps track of what vertices should be added to the queue
val toAdd = mutableListOf<GraphVertex>()
toAdd.addAll(queue)

// loop queue and check if we find the closest path to v2
// (by using [depth])
while (queue.isNotEmpty()) {
    queue.forEach { node ->
        visited.add(node)
        for (e in node.getAdjacencies()) {
            if (e == null)
                continue
            val a = graph.getVertex(e.adjacent)
            if (a == null || visited.contains(a) ||
                queue.contains(a) || toAdd.contains(a))
                    continue
            toAdd.add(a)
            if (a == v2) {
                println("smallestDistance(${v.id}, ${v2.id}) = $sum")
```

```
                    return sum
                }
            }
        }
        queue.addAll(toAdd)
        queue.removeAll(visited)
        depth++
    }
```

5 Avaliação experimental

Nesta secção do trabalho é onde iremos proceder à avaliação experimental deste problema. Aqui é onde irão ser apresentados os resultados experimentais. Para esta avaliação experimental, recorremos aos dados de redes não orientadas presentes em https://snap.stanford.edu/data/. Primeiro, foi realizada a avaliação experimental para o ficheiro facebook-combined.txt, que contém 4038 nodes. Ao correr este programa, foram utilizados os vários comandos e foi medido o seu tempo de execução. De seguida, foi também realizada a mesma avaliação experimental mas desta vez para o ficheiro ca-HepTh.txt, que contém 51971 nodes. As funções degree e degrees, em ambos os txt files são de rápida execução, na ordem dos milisegundos. Já a execução dos comandos closeness e smallestDistance é mais demorada, por vezes na ordem dos segundos (1 a 5 segundos). Infelizmente não foi possível fazer uma medição precisa, e por consequência não irá haver uma apresentação gráfica dos resultados dos tempos de execução de cada comando no prazo de entrega.

6 Conclusão

Com este problema, foi possível aprofundar o conhecimento sobre algoritmos utilizando árvores e grafos e perceber a sua importância no desenvolvimento de programas que contenham grande quantidade de dados, de modo a que sejam mais rápidos e ocupem menos memória.