Larissa Gremelmaier Rosa (20100531)

Raquel Cristina Schaly Behrens (20100544)

Atividade A1 – Relatório

Grafos (INE5413)

26 de Maio de 2020

Sumário

[Exercício 1 – Representação 3](#_Toc104131144)

[Exercício 3 – Ciclo Euleriano 4](#_Toc104131145)

[Exercício 4 – Algoritmo de Floyd-Warshall 4](#_Toc104131146)

# Exercício 1 – Representação

A linguagem de implementação escolhida foi Java, e, desta forma, foi criada a classe Grafo para a resolução do primeiro exercício. Juntamente, optou-se pela criação de uma ArrayList de inteiros, para representar a lista de vértices, outra ArrayList de strings para representar os rótulos de cada vértice (e, portanto, o índice *v* de um vértice da lista de vértices *V* é o mesmo índice de um rótulo *r* da lista de rótulos *R*), e uma lista linkada *E* para representar cada aresta. Cada aresta *e*, pertencente à lista de arestas *E*, é uma ArrayList de floats com dois vértices e o peso da aresta.

* Optou-se pela utilização de ArrayList em *V*, *R* e listas individuais *e*’s, porque muitos métodos da classe Grafos utilizam a função *get*, que, para LinkedLists, possui a complexidade O(n) no pior caso, e, para ArrayLists, possui complexidade O(1).
* Optou-se pela utilizaçação de LinkedLists em *E*, porque, para grafos muito grandes, pode-se haver muito mais arestas que vértices, e, ao inicializar o grafo, a função *add* é amplamente utilizada. Para LinkedLists, *add* possui complexidade O(1), e, para ArrayLists, possui O(n) no pior caso.

Complexidade das funções implementadas na classe Grafo:

* qtdVertices(): utiliza a função *size*, que possui complexidade O(1);
* qtdArestas(): utiliza a função *size*, que possui complexidade O(1);
* grau(v): possui um for que percorre a lista *E* de arestas, e, para cada lista *e* contida em *E*, verifica os dois primeiros índices para verificar se alguma aresta em *e* é igual ao vértice *v* recebido na entrada. Considerando *n* o número de arestas, a função possui complexidade O(n);
* rotulo(v): contém for que possui *n*iterações, sendo *n* igual à quantidade de vértices. Logo, possui complexidade O(n);
* vizinhos(v): contém for que possui *n*iterações, sendo *n* igual à quantidade de arestas. Logo, possui complexidade O(n);
* haAresta({u, v}): contém for que possui *n* iterações, sendo *n* igual à quantidade de arestas. Logo, possui complexidade O(n);
* peso({u, v}): contém for que possui *n* iterações, sendo *n* igual à quantidade de arestas. Logo, possui complexidade O(n);
* lerArquivo(path): contém um while que percorre todas as linhas do arquivo que está no path fornecido. Além disso, dentro do while estão contidas algumas chamadas de funções:
  + *add*
    - Para ArrayLists, cujo tamanho pode alcançar no máximo 3 índices, quando estiver capturando valores de arestas;
    - Para ArrayLists, cuja quantidade de índices pode alcançar no máximo o número da quantidade de vértices, para quando o while estiver lendo os vértices;
    - Para LinkedLists que podem possuir tamanhos muito maiores.

Dessa forma, sendo *n* a quantidade de linhas dentro do arquivo, *m* a quantidade de vértices, e, levando em consideração que add para LinkedLists possui complexidade O(1), a complexidade desta função é O(n\*(m-1)).

# Exercício 3 – Ciclo Euleriano

Para a verificação de um Ciclo Euleriano foi utilizada o Algoritmo de Hierholz. Para isso, criou-se a classe CicloEuleriano, que possui os atributos:

* ehCiclo: booleano que quando false, significa que não há ciclo euleriano no arquivo do path indicado:
* caminho: ArrayList de inteiros que indica o caminho do ciclo;
* C: ArrayList de booleanos que indica se uma determinada aresta foi visitada ou não.

Optou-se pelo uso de ArrayLists, dado que a complexidade da função *get* – muito usada nas funções da classe – para elas é O(1).

Além disso, para que fosse possível as funções dessa classe retornarem tipos de dados diferentes, foi criada uma classe ReturnWithDifferentTypes, que possui dois atributos: ehCiclo (booleano) e caminho (ArrayList).

# Exercício 4 – Algoritmo de Floyd-Warshall

Para a execução do algoritmo de Floyd-Warshall criou-se a classe FloydWarshall, a qual implementa as funções: setMatrizAdjacência e floydWarshall. Como o algoritmo usa matriz de adjacência, a função setMatrizAdjacência utiliza uma lista simples de Arraylists de floats para implementá-la. Dessa forma é possível inicializar uma matriz em java, que possua tamanho flexível. Escolheu-se ArrayLists, já que a classe utiliza muito os métodos *get* e *set* de lista, que, neste caso, possuem complexidade O(1).