Universidade Federal de Santa Catarina
Centro Tecnológico
Departamento de Informática e Estatística
INE5413 – Grafos
Graduando 1: Cainã Correa Caldas
Graduando 2: Vinicius Guedes dos Santos

Atividade I

1 - Grafo) Para a primeira questão, que foi a elaboração da classe que representaria o grafo, utilizou-se duas estruturas de dados, uma delas sendo uma matriz de adjacências e outra uma lista com os rótulos dos vértices, nesta última estrutura, o índice da lista representa o vértice e o valor na posição o seu rótulo, além disso, também utilizou-se uma variável contendo o número de arestas do grafo. A estrutura de matriz foi escolhida pois ela tem complexidade *O(1)* para todas as seguintes operações exigidas: "qtdVertices", "qtdArestas", "rotulo", "haAresta" e "peso", para as demais operações, "grau" e "vizinhos", a complexidade é *O(|V|)*, onde *|V|* é o número de vértices, apesar de estas duas últimas operações terem complexidade maior, elas não são muito inferiores às suas versões realizadas em uma lista de adjacências, já que mesmo nesta, a custo da operação no pior caso também é *O(|V|)*.
Implementamos também essa questão usando a técnica da Lista de Adjacências. As motivações foram o próprio exercício, e também para aproveitar em alguns casos o fato de que as operações “grau” e “vizinhos” geralmente são mais rápidas, pois, a matriz visita todas as possibilidades de vértices, enquanto a lista itera no máximo entre os que são realmente vizinhos do vértice em questão.
Arquivos: Grafo.py, GrafoListaAdjacencias.py

2 – Algoritmo de Busca em largura) A segunda questão não exigiu muita criatividade quanto ao uso de estruturas de dados, visto que o acesso às estruturas C_v (visitados), D_v (distâncias) e A_v (antecessores) é feito sempre através de índices, bastou utilizar o número do grafo como índice, assim, a complexidade para todas as operações realizadas neste algoritmo é *O(1)*, além disso, também foi utilizada uma fila, que é própria do algoritmo, cuja complexidade de inserção e remoção também é *O(1)*.
Arquivos: Busca.py

3 – Algoritmo de Hierholzer) Neste algoritmo foi utilizada uma matriz para indicar que arestas já haviam sido visitadas, está opção consome muita memória, porém tem complexidade *O(1)* tanto para alterações quanto para consultas a arestas específicas, portanto é muito eficiente. Para representar o ciclo, foi utilizada uma lista.
Arquivos: Hierholzer.py

4 – Algoritmo de Dijkstra) Nesta questão, utilizou-se uma estrutura de heap binária para mais facilmente encontrar o vértice cuja distância da origem é a menor, como estrutura auxiliar, também foi utilizada uma lista que mapeia de D_v (lista de distâncias) para a lista contendo a heap, de forma que seja possível manter a coerência entre elas. A complexidade da remoção do vértice de menor distância é *O(log n)*, que é mais eficiente do que *O(n)* que seria a complexidade caso se utilizasse uma lista simples.
Arquivos: heap.py e Dijkstra.py

5 – Algoritmo de Floyd-Warshall) Este algoritmo não requer nenhuma estrutura além das matrizes que são próprias dele, ainda assim, foi possível fazer a seguinte otimização: Considerando que nunca é feita uma referência a matriz $d^{(k-2)}$ não há necessidade de se manter uma referência a todas as matrizes já criadas, portanto foi possível utilizar apenas duas matrizes, uma delas

mantém a matriz $d^{(k)}$ e a outra a matriz $d^{(k-1)}$, após uma iteração do primeiro laço, a matriz $d^{(k-1)}$ recebe o valor da matriz $d^{(k)}$ e a matriz $d^{(k)}$ é reinicializada.
Arquivos: Floyd_Warshall.py