Marina Godinho Domingues (18103928)
Matheus Bruhns Bastos (15100950)
Matheus Felipe Souza Valin (14200949)

Atividade A1 - Relatório

Grafos (INE5413)

Professor: Rafael de Santiago

28 de Abril de 2019

Conteúdo

1 Exercício 1 - Representação

A linguagem de implementação escolhida foi **Python**, devido a sua amistosidade e suas práticas funções nativas com implementação bastante eficiente (funções de acesso a itens de lista, de dicionário, retornar tamanho de lista, etc). O primeiro ponto a ser comentado diz respeito ao tipo estruturado de dados ou classe para um grafo não dirigido e ponderado G(V, E, w), conforme requisitado no enunciado. Optou-se pela utilização de uma classe Grafo, com os respectivos principais atributos:

- Vértices: É um dicionário porque (considerando uma construção sem colisões, como foi garantida no trabalho) a complexidade de tempo para acessar itens do dicionário (que, em Python, é uma tabela hash) é $O(1)$. No dicionário, o número do vértice é o número de uma chave, e o conteúdo dessa chave é um outro dicionário. Esse dicionário *interno* possui 3 tipos de informação:
  - Rótulo: conteúdo dessa chave é um nome ou referência ao vértice. Há apenas uma chave desse tipo no dicionário interno.
  - Vizinho: conteúdo dessa chave é o peso da aresta que conecta o vértice original ao vizinho. Há n chaves desse tipo no dicionário interno, onde n é o número de vizinhos que o vértice original possui.
  - Índice das arestas: conteúdo dessa chave é uma lista contendo os índices dos itens que possuem o vértice original em alguma posição na lista de arestas. Há apenas uma chave desse tipo no dicionário interno.

  Vale notar que aqui cada aresta é representada duas vezes: uma no vértice original, no seu dicionário interno, apresentando outro como vizinho, contando o peso da aresta; outra no vértice vizinho, no seu dicionário interno, apresentando o original como vizinho desse, contando também o peso da aresta.
- Arestas: É uma lista porque a lista de arestas não é alterada e, portanto, requer apenas que seja devidamente acessada ou percorrida, sendo que a função para retornar um item de uma lista em Python possui complexidade de tempo $O(1)$, e a função para verificação de existência na lista (inclui percorrê-la) possui complexidade de tempo $O(n)$, sendo n o comprimento da lista. Essa lista de arestas, contudo, é composta por tuplas de três elementos, sendo eles um vértice, um vizinho desse vértice e o peso da aresta que os conecta. Vale ressaltar que, enquanto na estrutura de dados dos vértices as arestas de certa forma eram representadas de forma redundante (conexão entre um vértice v_1 e outro vértice v_2 era tanto representada em v_1 quanto em v_2, de forma equivalente), na estrutura de lista das arestas não há repetição. Portanto, o comprimento da lista de Arestas E é igual ao número de arestas no grafo G.

A seguir, a respeito da implementação das funções de representação, requisitadas na Questão 1:

- qtdVertices(): complexidade de tempo $O(1)$, dado que é o número de chaves no dicionário de vértices.
- qtdArestas(): complexidade de tempo $O(1)$, dado que é o comprimento da lista de arestas.
- grau(v): complexidade de tempo $O(1)$, dado que é o número de chaves em um determinado item (dicionário interno) no dicionário original de vértices. Retornar um elemento do dicionário tem complexidade de tempo $O(1)$ e retornar o número de chaves num dicionário também.
- rotulo(v): complexidade de tempo $O(1)$, dado que é retornar o item de uma determinada chave do dicionário.
- vizinhos(v): complexidade de tempo $O(n)$, sendo n o número de vértices, pois utiliza apenas a função de cópia de dicionário (complexidade de tempo $O(n)$) e a função de remoção de item de dicionário (complexidade de tempo $O(1)$).
- haAresta(u,v): complexidade de tempo $O(n)$, sendo n o grau do vértice u, pois é necessário verificar se existe algum elemento no dicionário do vértice u que corresponde ao vértice v.
- peso(u,v): complexidade de tempo $O(1)$, dado que apenas retorna o item de uma chave de um dicionário.
- ler(Arquivo): a função foi implementada de forma simples, percorrendo o arquivo de texto uma vez, fazendo adições e atualizações nas estruturas de dados ao longo das iterações por linha. As operações são, em suma, atribuições simples de valor, adições de elementos em listas e atualizações de chaves de dicionários.

2 Exercício 2 - Buscas

Abaixo estão dispostas as principais estruturas utilizadas para o algoritmo de Busca em Largura (BFS):

- Estrutura de dados para marcar se vértice foi visitado: representa o status do vértice, podendo ser $True$ no caso do vértice já ter sido visitado pela busca, ou $False$ no caso dele não ter sido ainda visitado pela busca. Foi escolhida uma **lista**, dado que as funções $get()$ e $set()$ para listas ambas possuem complexidade de tempo $O(1)$.

- Estrutura de dados para indicar distância do vértice de origem: também **lista**, pelo mesmo princípio anterior, inicializada com um valor arbitrário muito elevado (999999999).
- Estrutura de dados para indicar o vértice antecessor: também **lista**, pelo mesmo princípio anterior, inicializada com valor *None* (o equivalente em Python do *null* de outras linguagens).
- Informações de Nível: para armazenar informações referentes ao nível em que a busca se encontra (necessário para apresentação devida dos resultados da Busca em Largura) foi criado um dicionário em que as chaves correspondem ao nível da busca e os itens presentes correspondem aos vértices que a busca alcançou naquele nível.
- Fila de Vizinhos: usado a estrutura *deque* da biblioteca nativa *collections*, um *deque* é uma lista otimizada para inserção e remoção de itens. Em listas normais, a remoção do último item ($lista.pop()$) é feita em complexidade de tempo $O(1)$, porém a remoção de um item arbitrário ($lista.pop(k)$) possui complexidade $O(k)$, sendo k a distância até o último elemento da lista. Assim, a remoção do primeiro item da lista ($lita.pop(0)$) possui complexidade de tempo $O(n)$, sendo n o tamanho da lista. Já utilizando a estrutura *deque*, é possível utilizar a função de remoção do início com complexidade de tempo $O(1)$, em algo como $lista.popleft()$. Como apenas são realizadas adições ou remoções no último ou no primeiro item da lista/deque, a estrutura é otimizada para essa finalidade.

Outros elementos do algoritmo de Busca em Largura são apenas variações de semelhantes já apresentados. São usadas funções de atualização, adição e remoção, e listas e em dicionários, para implementação correta do algoritmo.

3 Exercício 3 - Ciclo Euleriano

Para busca do Ciclo Euleriano foi utilizado o Algoritmo de Hierholzer, pela sua característica de sempre, quando existir, encontrar um ciclo euleriano no grafo. A implementação usa estruturas semelhantes aos algoritmos anteriores, dado que requer saber quais vértices já foram visitados, e também faz uso de remoção de itens do início de listas, para isso utilizando-se a estrutura *deque*, como já justificado. Para execução desse algoritmo foi utilizada a estrutura já descrita na Seção 1, que faz parte dos itens do dicionário de vértices: a lista dos índices das arestas que conectam o vértice (na lista de arestas). Dessa forma, torna-se trivial saber se há arestas não visitadas e, ainda mais, quais são elas e que vértices elas conectam.

4 Exercício 4 - Algoritmo de Dijkstra

A implementação do algoritmo de Dijkstra faz uso exatamente das mesmas estruturas de dados referenciadas na Seção 2, dado que precisa definir uma estrutura para mapear quais vértices já foram visitados, uma para armazenar as distâncias a cada vértice e uma para definir o antecessor de cada vértice. Além disso, foi-se aproveitada a estrutura de dicionário de vértices para acessar e percorrer os vértices durante o algoritmo.

5 Exercício 5 - Algoritmo de Floyd-Warshall

A implementação do algoritmo de Floyd-Warshall faz uso exatamente das mesmas estruturas de dados referenciadas na Seção 2. Ao iniciar, cria uma estrutura de matriz que será base para a atualização das distâncias ao longo da execução do algoritmo. Vale lembrar que esta versão utiliza-se de menor espaço de memória (complexidade de espaço), que tipicamente para o algoritmo seria $O(|V|^3)$, implementando-o com complexidade de espaço $O(|V|^2)$ - conforme notas de aula.

6 Observação

É possível, também, encontrar o código, além das instruções para uso em:
`https://github.com/Matbbastos/ine5413-grafos`