Resolução de Problemas com Grafos

Prof. Vinícius M. A. Souza

Atividade 4

Nomes: Gustavo Klinfuss da Silva, Anderson Ryuuchi e Marcio Vinicius

PARTE 1

Considerando as duas estruturas de dados para a representação de grafos implementadas nos mini-projetos (matriz de adjacências e lista de adjacências), discuta e compare os custos de cada uma das estruturas para as seguintes tarefas:

a) Armazenamento da estrutura

Melhor desempenho: Lista de adjacências

Justificativa: A matriz de adjacências tem um custo de armazenamento exponencial a quantidade de vértices, porém, não há custo em armazenar arestas. Já a lista de adjacências tende a ter um custo por vértice baixo, mas também há custos para cada aresta adicionada. Em situações onde trabalha-se com grafos completos, o custo de ambas tende a ser similar.

b) Retornar os vizinhos de um vértice

Melhor desempenho: Desempenho similar

Justificativa: A matriz de adjacência tem que iterar sobre toda a linha u da matriz para realizar essa operação. Já a lista de adjacência itera a lista encadeada vinculada ao vértice.

c) Verificar se dois vértices estão conectados

Melhor desempenho: Matriz de Adjacências

Justificativa: Na matriz de adjacência só é preciso verificar os valores [u][v] e [v][u]. Já a lista de adjacências precisa iterar sobre as arestas dos dois vértices para verificar se estão conectados, sendo que, no pior caso, vai iterar sobre todas as arestas, até a última.

d) Adicionar um vértice

Melhor desempenho: Lista de adjacências

Justificativa: Na lista de adjacência é adicionado apenas mais um vértice no fim da lista. Já a matriz de adjacência, teria que iterar cada linha da matriz, adicionando uma coluna, e ao final dessa iteração, adicionar uma linha na matriz.

e) Adicionar uma aresta

Melhor desempenho: Matriz de adjacências

Justificativa: Na matriz de adjacências é preciso apenas verificar uma única posição, ou no caso de aresta não-direcionada, em duas posições. Já na lista de adjacências seria preciso iterar sobre todas as arestas de um vértice, ou, no caso de arestas não direcionadas, seria preciso iterar sobre os dois vértices dos quais deseja-se adicionar a aresta.

f) Remover um vértice

Melhor desempenho: Matriz de adjacência

Justificativa: Na matriz de adjacências é preciso apenas remover a linha N que corresponde ao vértice e as colunas M que correspondem ao vértice. Já na lista de adjacências além de remover o vértice seria preciso percorrer cada vértice para verificar se não há aresta ligando com o vértice removido

g) Remover uma aresta

Melhor desempenho: Matriz de adjacência

Justificativa: Na matriz de adjacências é preciso apenas verificar uma única posição. Já na lista de adjacências seria preciso iterar sobre o vértice no qual a aresta está saindo para procurar a ligação e remove-la.

PARTE 2

Retome a sua implementação de grafo utilizando **matriz de adjacências** e adapte o código para que o usuário possa escolher se o grafo será direcionado ou não direcionado, além de ponderado ou não ponderado, no momento da instanciação.

Por exemplo:

G = Grafo(5, direcionado=True, ponderado=True) # irá gerar um grafo com 5 vértices, considerando arestas **direcionadas** e **ponderadas**

G = Grafo(5, direcionado=True, ponderado=False) # irá gerar um grafo com 5 vértices, considerando arestas **direcionadas** e **não ponderadas**

G = Grafo(5, direcionado=False, ponderado=True) # irá gerar um grafo com 5 vértices, considerando arestas **não direcionadas** e **ponderadas**

G = Grafo(5, direcionado=False, ponderado=False) # irá gerar um grafo com 5 vértices, considerando arestas **não direcionadas** e **não ponderadas**

Realize as modificações necessárias para que os seguintes métodos* funcionem corretamente:

a. def adiciona_aresta(u, v, peso) # cria uma aresta com peso entre os vértices u e v do grafo G. No caso de um grafo não ponderado, deve ser atribuído um peso = 1 para as arestas.
b. def remove_aresta(u, v) # remove a aresta entre os vértices u e v do grafo G.

c. def tem_aresta(u, v) # verifica se existe uma aresta entre os vértices u e v do grafo G e retorna True ou False.
d. def grau(u) # retorna a quantidade de arestas conectadas ao vértice u do grafo G.
e. def grau_entrada(u) # retorna a quantidade de arestas que chegam até o vértice u do grafo direcionado G.
f. def grau_saida(u) # retorna a quantidade de arestas que saem do vértice u do grafo direcionado G.
g. def retorna_adjacentes(u) # retorna uma lista com os vértices adjacentes ao vértice u. Lembre que no caso de grafos não direcionados, todos os vértices conectados ao vértice u são considerados adjacentes. No caso de grafos direcionados, os adjacentes de u são somente os vértices em que é possível sair de u e chegar até outro vértice a partir de uma aresta.

* Caso algum destes métodos ainda não esteja implementado, implemente-o.