**Grafos - Relatório Atividade 1**

Alunos: Gabriel Turatti, Gabriela Furtado da Silveira, Victor Rodrigues Gouvêa

**Questão 1:**

Foi criada uma classe para o Nodo do grafo, e uma classe para o grafo em si. Na classe do Nodo, foram usadas as propriedades:

* Id, para representar o nodo;
* rótulo, para representar o nome do nodo;
* connections, para representar a quantidade de conexões que aquele nodo possui (por razões de tornar as outras funções do grafo mais eficientes no tempo de execução);
* vizinhos, para representar o dicionário que guarda a informação de que nodos são vizinhos e qual o peso da conexão deles.

Na classe do Grafo, foram utilizadas as propriedades:

* V, como o dicionário de nodos presentes no grafo, sendo a posição do dicionário o id do nodo, e o valor o objeto do nodo em si;
* nV, a quantidade de nodos (novamente, com o propósito de agilizar o funcionamento de funções que dependam do tamanho do grafo);
* E, como a lista de arestas do grafo;
* nE, como a quantidade de arestas.

Para os métodos que eram necessários programar, qtdVertices e qtdArestas utilizam do nV e nE, sendo dessa forma O(1), grau retorna o connections do nodo, rotulo retorna o rótulo do nodo, vizinhos retorna o dicionário de vizinhos do nodo, haAresta checa se o peso da conexão entre os dois nodos é menor que infinito, se for, é porque há uma aresta e ele retorna verdadeiro, peso simplesmente retorna o que tem na posição id do dicionário, que é o valor do peso da aresta.

A função de ler o arquivo, primeiramente irá escanear cada linha do arquivo em busca do caractere de ‘\*’, quando encontrar ele começara a ler as próximas linhas e criar nodos para cada uma delas, assim que encontrar outro caractere ‘\*’, ele começará a registrar as informações como arestas.

**Questão 2:**

Para a busca em largura, foram criadas duas funções: buscaLargura e printResult.

A função buscaLargura começa com a criação do grafo a partir da leitura do arquivo de entrada, seguido do algoritmo de busca em largura visto em aula. Para este algoritmo, utilizamos listas para armazenar as informações de vértice conhecido, distância e vértice ancestral, onde o índice da lista indica o id do vértice para qual aquela informação é válida. Com relação a fila da busca em largura, utilizamos também a lista padrão do python, adicionando sempre o elemento na última posição da lista e removendo sempre o elemento da primeira posição.

Na função printResult, apenas fazemos a lógica de prints com a formatação exigida pela questão, utilizando fazendo a relação dos índices das listas de retorno com os id’s dos vértices do grafo, como explicado anteriormente.

**Questão 3:**

Foi implementada uma versão do algoritmo de Hierholzer para encontrar Ciclos Eulerianos, seguindo o que foi visto em aula. Para indicar se o vértice é conhecido, foi utilizado um dicionário, cujas chaves são conjuntos (frozensets) contendo pares de vértices (já que são consideradas arestas, ou seja, a ordem não importa). Essa escolha foi realizada para que a verificação de visita do vértice tivesse complexidade constante. No entanto, para algumas operações foi necessário construir uma lista contendo as chaves desse dicionário.

O algoritmo principal realiza uma chamada a “buscaCiclo”, passando um vértice qualquer que tenha conexão, e nessa função o ciclo é salvo como uma lista, na qual os elementos são atributos “id” de objetos da classe “Node” (vértices). Escolhemos utilizar uma lista porque é a opção mais fácil para armazenar uma quantidade variável de elementos, e a inserção no final é O(1). Tenta-se montar um ciclo a partir dos vizinhos do vértice com verificações de visita realizadas a partir do dicionário também passado como argumento, e caso haja algum subciclo, os novos elementos são inseridos na lista “ciclo” nas posições adequadas.

Com o retorno obtido de “buscaCiclo”, o algoritmo principal também verifica se todas as arestas foram visitadas, buscando algum valor falso no dicionário de visitas, e realiza a impressão na tela da saída esperada.

**Questão 4:**

Para essa questão, escolhemos implementar o algoritmo de Bellman-Ford. Utilizamos duas funções para resolver a questão: BellmanFord e montaCaminho.

A função BellmanFord começa com a leitura e criação do grafo a partir do arquivo de entrada. Após isso, o algoritmo de Bellman-Ford é implementado da mesma forma como vimos em aula. Escolhemos usar dicionários para armazenar os custos dos caminhos (D) e os antecessores dos vértices de destino (A), onde a chave de cada elemento é o id do respectivo vértice. Dessa forma, tanto o acesso quanto a modificação dos valores de caminhos e antecessores são O(1).

Na função montaCaminho, fazemos a formatação de saída que a questão pede. montamos listas com o caminho mínimo até cada vértice do grafo a partir do vértice “s” escolhido na entrada. Foram escolhidas listas pois permitem armazenar quantidade variável de elementos mais facilmente. Utilizamos a função BellmanFord, descrita acima, para obter as distâncias mínimas até todos os vértices a partir de “s”, os antecessores e a variável que indica se há um ciclo negativo. Após isso criamos uma lista para cada vértice de destino, percorrendo os seus antecessores até chegar no vértice “s”, a fim de encontrar os vértices que compõem o caminho mínimo. Então, fazemos os prints destas listas com o custo do caminho mínimo.

**Questão 5:**

A função do algoritmo Floyd-Warshall foi implementada como visto na aula. Primeiramente é criada uma matriz, e preenchida a informação dela com infinito, representando a distância de cada conexão dos nodos, exceto a distância de um nodo para si mesmo, que é sempre registrada como 0. Após isso, a função irá ler a lista de arestas, e atualizar os valores de infinito de acordo.

Durante o principal looping do algoritmo, pela forma como o “for” se comporta com dicionários no Python, foi necessário resgatar o valor dos indexes de cada vértice para depois ser feita a comparação de distância.