DIOGO OLIVEIRA CARVALHO – 202120533

LUCAS SILVA MEIRA - 202120807

BRUNO DE ALMEIDA DE PAULA – 201920350

PROFESSOR: MAYRON MOREIRA

DISCIPLINA: ALGORITMOS EM GRAFOS

TRABALHO PRÁTICO – ALGORITMOS EM GRAFOS

AGOSTO/2024

SUMÁRIO

1. Introdução

Este trabalho implementa diversas funções para manipulação e análise de grafos, incluindo a leitura de grafos, verificação de propriedades como conexidade, bipartição, eulerianidade, detecção de ciclos, contagem de componentes conexas e fortemente conexas. As operações são realizadas utilizando algoritmos clássicos de busca em largura (BFS) e profundidade (DFS).

2. Definições e Estruturas

- **Vértice:** Ponto de conexão em um grafo.
- **Aresta:** Conexão entre dois vértices, podendo ter um peso associado.
- **Grafo Direcionado:** As arestas têm uma direção (u -> v).
- **Grafo Não Direcionado:** As arestas não têm direção (u <-> v).
- **Lista de Adjacência:** Representação do grafo onde cada vértice aponta para os vértices adjacentes.

3. Tipos e Estruturas Utilizadas

- **struct aresta:**
  - id: Identificador único da aresta.
  - u: Primeiro vértice da aresta.
  - v: Segundo vértice da aresta.
  - peso: Peso associado à aresta.
- **enum TipoGrafo:**
  - direcionado: Grafo com arestas direcionadas.
  - nao_direcionado: Grafo com arestas não direcionadas.

4. Funções Implementadas

4.1 lerGrafo

- **Função:** void lerGrafo(int n, int m, enum TipoGrafo tipo, vector<aresta>* LA)

- **Descrição:** Leia os dados do gráfico e preencha a lista de adjacências. A função primeiro lê o número de vértices e arestas e, em seguida, processa cada aresta fornecida, adicionando-a à lista de adjacência correspondente. Dependendo do tipo de gráfico (direcionado ou não direcionado), esta função ajusta as conexões de acordo com a direção das arestas. Retorna uma lista de adjacências preenchida com arestas válidas.

4.2 bfsConexidade

- **Função:** void bfsConexidade(int s, vector<int>& cor, vector<aresta>* LA)
- **Descrição:** Verifique a conectividade do gráfico
- usando pesquisa em largura (BFS). A função começa nos vértices de origem, explora todos os seus vizinhos e os marca como visitados. Repita este processo para todos os vértices conectados ao vértice inicial. Após a exploração, a função verifica se todos os vértices foram visitados. Retorna 1 se todos os vértices foram visitados (indicando que o grafo está conectado se não for direcionado ou fracamente conectado se for direcionado); retorna 0 se nenhum vértice tiver sido visitado (indicando que o grafo está desconectado);

4.3 conexidadeGrafo

- **Função:** int conexidadeGrafo(int n, enum TipoGrafo tipo, vector<aresta>* LA)

- **Descrição:** Verifique a conectividade do gráfico. Para gráficos direcionados, esta função cria um gráfico auxiliar com arestas invertidas. Em seguida, executa um BFS no gráfico original e outro BFS no gráfico auxiliar. Se todos os vértices em ambos os gráficos forem alcançáveis a partir do vértice inicial, os gráficos serão considerados fracamente conectados. Retorna 1 se o gráfico estiver conectado (não direcionado) ou fracamente conectado (direcionado);

4.4 bipartido

- **Função:** int bipartido(int n, vector<aresta>* LA)

- **Descrição:** Verifique se o grafo não direcionado é um grafo bipartido. Ele usa pesquisa em amplitude (BFS) com coloração para realizar a verificação. A função começa em um vértice e atribui uma cor (por exemplo, 0). Em seguida, ele colore todos os vizinhos com a cor oposta (1). Continue este processo para todos os vértices do gráfico. Se não houver vértices adjacentes da mesma cor no final, o grafo é um grafo bipartido. Retorna 1 se o gráfico for bipartido, 0 caso contrário.

4.5 euleriano

- **Função:** int euleriano(int n, enum TipoGrafo tipo, vector<aresta>* LA)

- **Descrição:** Verifique se o gráfico é um gráfico de Euler. Para gráficos direcionados, esta função verifica se o grau de entrada de cada vértice é igual ao grau de saída. Para gráficos não direcionados, calcule o grau de cada vértice e verifique se todos eles possuem um número par. Retorna 1 se o gráfico for um gráfico de Euler (de acordo com condições específicas do tipo de gráfico), 0 caso contrário.

4.6 dfsCiclo

- **Função:** void dfsCiclo(int n, int u, vector<int>& cor, vector<int>& pai, vector<aresta>* LA)

- **Descrição:** pesquisa recursivamente por loops em um gráfico usando pesquisa em profundidade (DFS). Esta função executa DFS a partir do vértice u e marca os vértices visitados. Um ciclo é detectado se durante o processo DFS for descoberto que já existe um vértice na pilha recursiva (ou seja, um vértice que foi visitado e ainda não

completou a exploração). Retorna 1 se um loop for encontrado, 0 caso contrário.

4.6 ciclo

- **Função:** int ciclo(int n, vector<aresta>* LA)

- **Descrição:** Verifique se o gráfico possui ciclos. Ele explora o gráfico de cada vértice usando pesquisa em profundidade (DFS). Se algum dos vértices explorados formar um ciclo (identificado pelo retorno da função dfsCiclo), a função retornará 1. Caso contrário, se nenhum anel for encontrado após a conclusão da exploração, 0 será retornado.

4.7 qtdCompConexas

- **Função:** int qtdCompConexas(int n, vector<aresta>* LA)

- **Descrição:** Conta o número de componentes conectados em um gráfico não direcionado. Esta função usa BFS ou DFS para explorar cada componente do gráfico a partir de vértices não visitados, contando o número total de componentes conectados. Retorna o número de componentes conectados encontrados.

4.8 dfsCompFortConexas

- **Função:** void dfsCompFortConexas(int s, int& t, vector<int>& cor, vector<int>& f, vector<aresta>* LA)

- **Descrição:** Realiza uma DFS para encontrar componentes fortemente conexas em grafos direcionados. A função explora o grafo a partir do vértice de origem s, atualizando os tempos de descoberta e fechamento dos vértices. Modifica os vetores cor e f para refletir o estado e os tempos de fechamento dos vértices.

4.9 qtdCompFortConexas

- **Função:** int qtdCompFortConexas(int n, vector<aresta>* LA)

- **Descrição:** Calcula o número de componentes fortemente conexas em grafos direcionados. Utiliza a função dfsCompFortConexas para encontrar e contar as componentes fortemente conexas no grafo. Retorna a quantidade de componentes fortemente conexas encontradas.

4.10 tarjan

- **Função:** void tarjan(int u, int* d, int* low, int* pai, int& t, int& qtdFilhosRaiz, const int raiz, int& contArtic, int& contPontes, const vector<aresta>* LA, bool* articulacoes, bool* pontes)

- **Descrição:** Implementa o algoritmo de Tarjan para encontrar vértices de articulação e arestas ponte em um grafo não direcionado. A função inicializa o tempo de descoberta e o valor low para o vértice u, percorre os vizinhos e realiza chamadas recursivas para descobrir articulações e pontes. Atualiza os vetores articulacoes e pontes e os contadores contArtic e contPontes com os resultados encontrados.

4.11 articulacoesEPontes

- **Função:** void articulacoesEPontes(int n, int m, enum TipoGrafo tipo, int& contArtic, int& contPontes, int** verticesArticulacao, int** idArestasPonte, const vector<aresta>* LA)

- **Descrição:** Calcula os vértices de articulação e as arestas ponte de um grafo não direcionado usando o algoritmo de Tarjan. A função verifica se o grafo é não direcionado, inicializa os vetores necessários e executa o algoritmo de Tarjan para cada componente conexo, armazenando os resultados nos vetores de vértices de articulação e arestas ponte.

4.12 dfsArvoreVisit

- **Função:** void dfsArvoreVisit(int u, int* cor, queue<int>& filaArvore, const vector<aresta>* LA)

- **Descrição:** Realiza uma busca em profundidade (DFS) e armazena as arestas da árvore de busca em uma fila. A função marca o vértice atual como visitado e percorre seus vizinhos, adicionando as arestas da árvore de busca à fila.

4.13 dfsArvore

- **Função:** queue<int> dfsArvore(int n, const vector<aresta>* LA)

- **Descrição:** Executa uma busca em profundidade (DFS) a partir do vértice 0 e retorna uma fila contendo os IDs das arestas da árvore de profundidade. A função utiliza DFS para explorar o grafo e organiza as arestas encontradas em uma fila.

4.14 bfsArvore

- **Função:** queue<int> bfsArvore(int n, const vector<aresta>* LA)

- **Descrição:** Executa uma busca em largura (BFS) a partir do vértice 0 e retorna uma fila contendo os IDs das arestas da árvore de largura. A função utiliza BFS para explorar o grafo e organiza as arestas encontradas em uma fila.

4.15 obtemMenorChave

- **Função:** int obtemMenorChave(int* chave, bool* pertence_agm, int n)

- **Descrição:** Retorna o vértice com a menor chave entre os vértices não pertencentes à árvore geradora mínima parcial. A função utiliza os vetores de chave e de pertença à árvore para identificar o vértice com a menor chave.

4.16 agmPrim

- **Função:** int agmPrim(int n, enum TipoGrafo tipo, int conexidade, const vector<aresta>* LA)

- **Descrição:** Calcula o valor da árvore geradora mínima (AGM) usando o algoritmo de Prim. Retorna o peso total da árvore geradora mínima ou -1 se o grafo não for não direcionado ou se for desconexo. A função utiliza o algoritmo de Prim para encontrar a árvore de custo mínimo.

4.17 ordenacaoTopologicaKahn

- **Função:** queue<int> ordenacaoTopologicaKahn(int n, enum TipoGrafo tipo, int possuiCiclo, const vector<aresta>* LA)

- **Descrição:** Calcula a ordenação topológica dos vértices de um grafo direcionado usando o algoritmo de Kahn. A função utiliza o algoritmo de Kahn para encontrar uma ordenação linear dos vértices que respeita as relações de precedência do grafo.

4.18 caminhoMinimoBellmanFord

- **Função:** int caminhoMinimoBellmanFord(int n, int s, int t, const vector<aresta>* LA)

- **Descrição:** Calcula o caminho mínimo entre dois vértices usando o algoritmo de Bellman-Ford. A função retorna a distância do caminho mínimo entre os vértices s e t, ou -1 se não houver caminho.

4.19 caminhoAumentante

- **Função:** void caminhoAumentante(int s, int v, int& fluxo, int capacidade_fluxo, int* pai, int** MatFluxo)

- **Descrição:** Calcula o fluxo de um caminho aumentante entre o vértice s e v e atualiza a matriz de fluxo residual. A função atualiza o fluxo e a matriz de fluxo com base no caminho encontrado.

4.20 fluxoMaximoEdmondsKarp

- **Função:** int fluxoMaximoEdmondsKarp(int n, int s, int t, int** MatCapacidade)

- **Descrição:** Calcula o fluxo máximo entre dois vértices usando o algoritmo de Edmonds-Karp. A função retorna o valor do fluxo máximo entre s e t, considerando as capacidades das arestas.

5. Caso de Teste

5.1 Caso de Teste 1: O grafo é conexo, bipartido, mas não é euleriano e não possui ciclos

Entrada:

```
5 5 nao_direcionado
0 0 1 1
1 1 2 1
2 1 3 1
3 1 4 1
4 2 0 1
```

Saída Esperada:

```
1  // Conexo
```

1 // Bipartido
0 // Não é euleriano
0 // Não possui ciclo
```

**5.2 Caso de Teste 2: O grafo é conexo em termos de conexidade fraca (considerando a falta de direção), não é bipartido, é euleriano e possui ciclos.**

Entrada:
6 7 direcionado
0 0 1 1
1 1 2 1
2 1 3 1
3 1 4 1
4 1 5 1
5 1 2 1
2 1 0 1

Saída Esperada:

1  // Conexidade fraca
0  // Não bipartido
1  // É euleriano
1  // Possui ciclo

**5.3 Caso de Teste 3: O grafo é conexo, não é bipartido, é euleriano e possui ciclos**.

Entrada:
4 4 nao_direcionado
```

0 0 1 1

1 1 2 1

2 1 3 1

3 2 0 1

```

**Saída Esperada:**

```

1 // Conexo

0 // Não bipartido

1 // É euleriano

1 // Possui ciclo

```

### 5.4 Caso de Teste 4: Grafo Não-Direcionado e Não-Ponderado

**Entrada:**

4 4 nao_direcionado

0 0 1 1

1 1 2 1

2 1 3 1

3 2 3 1

**Descrição:**

- Grafo com 4 vértices e 4 arestas.
- Grafo não-direcionado e não-ponderado.
- Representa um ciclo quadrado: 0-1-2-3-0.

**Saída esperada**

1  (Grafo conexo)

0  (Grafo não bipartido)

1  (Grafo euleriano)

1  (Grafo possui ciclo)
```

5.5 Caso de Teste 5: Grafo Direcionado e Ponderado

Entrada:

5 6 direcionado
0 0 1 3
1 1 2 4
2 2 3 5
3 3 4 6
4 4 0 2
5 2 4 1

Descrição:

- Grafo com 5 vértices e 6 arestas.
- Grafo direcionado e ponderado.
- É um ciclo completo: 0 -> 1 -> 2 -> 3 -> 4 -> 0, com uma aresta adicional 2 -> 4.

Saída Esperada:

1 (Grafo fracamente conexo)
0 (Grafo não bipartido)
0 (Grafo não euleriano)
1 (Grafo possui ciclo)

5.6 Caso de Teste 6: Grafo Não-Direcionado e Ponderado com Componentes Conexas

Entrada:

6 5 nao_direcionado
0 0 1 1
1 1 2 1
2 3 4 1

3 4 5 1

4 5 3 1

Descrição:

- Grafo com 6 vértices e 5 arestas.
- Grafo não-direcionado e ponderado.
- Possui duas componentes conexas: {0, 1, 2} e {3, 4, 5}.

Saída Esperada:

0 (Grafo desconexo)

0 (Grafo não bipartido)

0 (Grafo não euleriano)

1 (Grafo possui ciclo)

5.7 Caso de Teste 7: Grafo Simples com Árvore

Entrada:

4 3 nao_direcionado

0 0 1 1

1 1 2 1

2 2 3 1

Descrição:

- Grafo com 4 vértices e 3 arestas.
- Grafo não-direcionado e não-ponderado.
- Forma uma árvore (linear): 0-1-2-3.

Saída Esperada:

1 (Grafo conexo)

1 (Grafo bipartido)

0 (Grafo não euleriano)

0 (Grafo não possui ciclo)

6. Conclusão

Este trabalho prático apresenta a implementação de algoritmos fundamentais para análise de grafos, cobrindo propriedades críticas como conexidade, bipartição, eulerianidade e detecção de ciclos. Através das estruturas de dados e algoritmos aplicados, os estudantes obtêm uma compreensão profunda sobre o comportamento dos grafos e suas propriedades