Trabalho Prático 2 - Algoritmos em Grafos

Aluno: João Victor Borges Carvalho

1. Introdução

Este projeto implementa um programa em Java capaz de carregar grafos (orientados ou não-orientados, ponderados ou não) a partir de arquivos de texto. O programa foi desenvolvido em conformidade com a arquitetura-base fornecida, utilizando Programação Orientada a Objetos para encapsular a lógica dos grafos e dos algoritmos.

O usuário pode escolher carregar o grafo em memória usando uma das três estruturas de dados clássicas:

- Lista de Adjacência
- Matriz de Adjacência
- Matriz de Incidência

Uma vez carregado, o programa oferece uma interface (tanto via console quanto via interface gráfica - GUI) para a execução de sete algoritmos fundamentais da Teoria dos Grafos, cujos resultados são exibidos ao usuário.

2. Arquitetura do Projeto

O código-fonte foi estruturado de forma modular, respeitando os princípios de POO e a separação de responsabilidades.

- **Interfaces:**
    - Grafo.java: Define o contrato que todas as estruturas de dados de grafo devem seguir (ex: adicionarAresta, adjacentesDe, criarGrafoTransposto).
    - AlgoritmosEmGrafos.java: Define o contrato para a classe de lógica principal.
- **Classes de Dados (Base):**
    - Vertice.java: Representa um vértice, contendo seu id.
    - Aresta.java: Representa uma aresta, contendo origem, destino e peso.
- **Implementações de Estrutura:**
    - GrafoListaAdjacencia.java: Implementa Grafo usando um ArrayList de ArrayLists de Arestas.
    - GrafoMatrizAdjacencia.java: Implementa Grafo usando uma matriz V*V de Doubles, onde null representa a ausência de aresta.
    - GrafoMatrizIncidencia.java: Implementa Grafo usando uma matriz V*A, onde os valores -1, 1 e 2 indicam a incidência (saída, entrada ou loop).
- **Lógica Principal:**

    - MeusAlgoritmosEmGrafos.java: Classe principal que implementa a interface AlgoritmosEmGrafos e contém a lógica para todos os 7 algoritmos solicitados.
- **Interfaces de Usuário (Drivers):**
    - Main.java: Ponto de entrada para a versão em modo console (menu de texto).
    - MainFrame.java: Ponto de entrada para a versão com Interface Gráfica (GUI) feita em Java Swing.
- **Utilitários:**
    - FileManager.java: Classe auxiliar para leitura de arquivos de texto.
    - TipoDeRepresentacao.java: Enum para a seleção da estrutura de dados.

3. Descrição das Soluções Implementadas

A seguir, descreve-se a lógica de implementação de cada um dos 7 algoritmos presentes na classe MeusAlgoritmosEmGrafos.

3.1. Carregamento do Grafo (carregarGrafo)

O método utiliza o FileManager para ler as linhas do arquivo. A primeira linha é lida para determinar o número V de vértices, que são instanciados. Um switch-case com base no TipoDeRepresentacao escolhido pelo usuário instancia a implementação correta do Grafo. As linhas restantes são processadas, usando split() para extrair a origem, o destino e o peso de cada aresta, que é então adicionada ao grafo.

3.2. Busca em Profundidade (DFS)

A DFS foi implementada de forma recursiva (método dfsVisit). Um "relógio" (tempo) global é usado para calcular os tempos de descoberta d[v] e finalização f[v] de cada vértice. A classificação das arestas (Árvore, Retorno, Avanço, Cruzamento) é realizada durante a busca, com base nas cores (BRANCO, CINZA, PRETO) e nos tempos d dos vértices u (origem) e v (destino) da aresta.

3.3. Busca em Largura (BFS)

A BFS foi implementada de forma iterativa, utilizando um ArrayList como Fila (FIFO). A busca parte de um vértice s e explora os vizinhos por níveis, utilizando o sistema de cores para controle. O algoritmo preenche os arrays de distância d[v] (número de arestas de s até v) e de pais pai[v] (paiBFS).

3.4. Detecção de Ciclo (existeCiclo)

Esta solução reutiliza o resultado da DFS. Em um grafo orientado, a existência de um ciclo é confirmada se, e somente se, a DFS encontra uma **Aresta de Retorno** (uma aresta que aponta de um vértice para um de seus ancestrais na árvore de busca, que ainda está na pilha de recursão - cor CINZA). O método simplesmente checa se a lista de arestas de retorno está vazia.

3.5. Componentes Fortemente Conexos (SCC)

Foi implementado o **Algoritmo de Kosaraju**, que possui quatro passos:

1. Executa a DFS no grafo original G para obter os tempos de finalização f[v].
2. Calcula o grafo transposto, G^T, chamando o método criarGrafoTransposto() (que cada estrutura de grafo implementa).
3. Executa uma segunda DFS, desta vez em G^T, processando os vértices em ordem decrescente de f[v].
4. Cada árvore gerada na DFS de G^T é um Componente Fortemente Conexo. O método então constrói e retorna o **Grafo Reduzido**, onde cada vértice representa um componente.

3.6. Árvore Geradora Mínima (AGM)

Foi implementado o **Algoritmo de Kruskal**.

1. Todas as arestas do grafo são extraídas e ordenadas em ordem crescente de peso.
2. Uma estrutura de Conjuntos Disjuntos (Disjoint-Set) foi criada (usando um ArrayList de ArrayLists) para armazenar os componentes conexos.
3. O algoritmo itera pelas arestas ordenadas. Se uma aresta (u, v) conecta dois componentes diferentes (verificado pela estrutura de Disjoint-Set), ela é adicionada à AGM e os componentes de u e v são unidos.
4. O processo termina quando a AGM possui V-1 arestas.

3.7. Caminho Mínimo

Foi implementado o **Algoritmo de Dijkstra**.

1. O algoritmo utiliza um ArrayList (Q) como fila de prioridade (implementação O(V^2)), onde a extração do vértice com menor distância (extrairMinimo) é feita por busca linear O(V).
2. Os arrays de distância d[v] e pais pai[v] (dC e paiC) são inicializados (distância Integer.MAX_VALUE, pai null, e origem s com d[s] = 0).
3. O algoritmo itera, extraindo o vértice u de menor distância de Q, e executa o **relaxamento** para todos os seus vizinhos v, atualizando d[v] e pai[v] se um caminho mais curto for encontrado.
4. Ao final, o caminho de s até t é reconstruído de trás para frente, seguindo o array de pais pi[v] a partir de t.

3.8. Fluxo Máximo

Foi implementado o **Algoritmo de Edmonds-Karp** (uma especialização de Ford-Fulkerson).

1. Um grafo residual é criado (como uma matriz de capacidade).

2. O algoritmo entra em um loop, procurando repetidamente por caminhos aumentantes da origem s ao sorvedouro t no grafo residual.
3. Esta busca por caminhos é feita usando BFS (método bfsFluxo).
4. Se um caminho é encontrado, o gargalo (menor capacidade no caminho) é calculado.
5. O fluxo é atualizado: o valor do gargalo é subtraído da capacidade das arestas diretas e somado à capacidade das arestas reversas no grafo residual.
6. O loop termina quando a BFS não consegue mais encontrar um caminho de s a t, e o fluxo máximo total acumulado é retornado.

4. Como Executar

O projeto inclui duas formas de execução:

1. **Versão Gráfica (Recomendada):** Executar o arquivo MainFrame.java. Isso abrirá uma janela onde o usuário pode carregar o grafo através do menu "Arquivo" e, em seguida, selecionar o algoritmo e clicar em "Executar". Os resultados são impressos na área de texto.
2. **Versão Console:** Executar o arquivo Main.java. O programa solicitará o caminho do arquivo, o tipo de estrutura e, em seguida, exibirá um menu de opções no console.