IF Sul de Minas - Campus Poços de Caldas
Trabalho Prático 2 de Projetos e Análise de Algoritmos
Implementação de Algoritmos de Grafos

Aluno: Iago Ananias Silva
Professor: Douglas Castilho

1. Utilizando a classe FileManager disponibilizada pelo professor, criei uma função **lerArquivo** na classe **AlgoritmosEmGrafos** para ler o arquivo a partir do caminho que o usuário entrar no console.

2. Utilizando o enum de tipos de representação disponibilizados pelo professor, as opções de estruturas de armazenamento de grafo são apresentadas ao usuário que por sua vez escolhe pelo número referente a estrutura. Adicionei a escolha por número para facilitar a comparação durante a execução.

3. Após isso a função **escolheMetodo** é chamada e lê a entrada do usuário que deverá ser um número compatível a um algoritmo (DFS, BFS, AGM, Caminho mínimo e fluxo máximo). Um arquivo enum foi criado para representar os algoritmos a serem escolhidos.

Busca em Profundidade (DFS)

```java
@Override
public Collection<Aresta> buscaEmProfundidade(Vertice verticeInicial) {
    int numeroDeVertices = this.grafo.numeroDeVertices();
    int tempo = 0;
    Vertice antecessorPadrao = new Vertice(-1);

    for (int i = 0; i < numeroDeVertices; i++) {
        this.grafo.vertices().get(i).setCor(BRANCO);
        this.grafo.vertices().get(i).setAntecessor(antecessorPadrao);
    }

    for (Vertice vertice : this.grafo.adjacentesDe(verticeInicial)) {
        if (vertice.getCor() == BRANCO) {
            tempo = this.visitaDfs(vertice, tempo);
        }
    }
```

```java
private int visitaDfs(Vertice vertice, int tempo) {
    if (vertice.getCor() == BRANCO) {
        vertice.setCor(CINZA);
        tempo++;
        vertice.setTempoDescoberta(tempo);
        ArrayList<Vertice> adjacentes = this.grafo.adjacentesDe(vertice);

        System.out.println(vertice.toString() + " - Tempo de Descoberta: " + vertice.getTempoDescoberta());
        if (adjacentes != null) {
            for (Vertice adjacente : adjacentes) {

                if (adjacente.getCor() == BRANCO) {
                    adjacente.setAntecessor(vertice);

                    tempo = visitaDfs(adjacente, tempo);
                }
            }

            vertice.setCor(PRETO);
            vertice.setTempoFinalizacao(tempo);
            System.out.println(vertice.toString() + " - Tempo de Finalização: " + vertice.getTempoFinalizacao());
        }
```

A função buscaEmProfundidade realiza uma busca em profundidade (Depth-First Search - DFS) a partir de um vértice inicial em um grafo. Durante a busca, ela registra informações sobre os vértices visitados e as arestas percorridas, categorizando as arestas em diferentes tipos, como Arestas de Árvores, Arestas de Retorno, Arestas de Avanço e Arestas de Cruzamento.

Resumo da função buscaEmProfundidade:

1. Ela inicia configurando a cor de todos os vértices como BRANCO, indicando que nenhum vértice foi visitado.

2. Começa a busca a partir do vértice inicial, usando uma chamada à função auxiliar visitaDfs.

3. Durante a busca, a função visitaDfs é responsável por realizar a DFS a partir de um vértice, registrando informações como o tempo de descoberta e o tempo de finalização de cada vértice.

4. Enquanto visita os vértices e arestas, a função coleta informações sobre as arestas encontradas, classificando-as em diferentes categorias:

    - Arestas de Árvores: Arestas que expandem a árvore da busca em profundidade.

    - Arestas de Retorno: Arestas que formam ciclos no grafo.

    - Arestas de Avanço: Arestas que avançam para vértices já visitados e não fazem parte de ciclos.

    - Arestas de Cruzamento: Arestas que conectam diferentes ramos da busca.
5. A função imprime os resultados, mostrando as informações sobre cada tipo de aresta encontrada no grafo.

No contexto de algoritmos em grafos, a busca em profundidade é uma técnica importante para explorar e analisar a estrutura de grafos, identificando ciclos e outros padrões. Os diferentes tipos de arestas são úteis para entender a conectividade e as relações entre os vértices em um grafo.

Busca em Largura (BFS)

```java
@Override
public Collection<Aresta> buscaEmLargura(Vertice verticeInicial) {
    if (this.grafo == null || verticeInicial == null) {
        return new ArrayList<>();
    }

    Queue<Vertice> fila = new LinkedList<>();
    ArrayList<Vertice> visitados = new ArrayList<>();
    ArrayList<Aresta> arestasEncontradas = new ArrayList<>();

    int tempo = 0;

    fila.add(verticeInicial);
    visitados.add(verticeInicial);

    while (!fila.isEmpty()) {
        Vertice verticeAtual = fila.poll();
        tempo++;
        for (Vertice vizinho : this.grafo.adjacentesDe(verticeAtual)) {
            if (!visitados.contains(vizinho)) {
                fila.add(vizinho);
                visitados.add(vizinho);
                visitados.add(vizinho);

                System.out.println("Pai: [" + +verticeAtual.id() + "]" + " - Vertice: [" + vizinho.id()
                        + "] - Tempo de Descoberta: - " + tempo);

                arestasEncontradas.addAll(this.grafo.arestasEntre(verticeAtual, vizinho));
            }
        }
    }

    return arestasEncontradas;
}
```

A função buscaEmLargura realiza uma busca em largura (Breadth-First Search - BFS) a partir de um vértice inicial em um grafo. Durante a busca, ela registra informações sobre os vértices visitados e as arestas percorridas, com o objetivo de identificar a ordem em que os vértices são alcançados a partir do vértice inicial.

Resumo da função buscaEmLargura:

1. Ela verifica se o grafo e o vértice inicial são válidos; caso contrário, retorna uma lista vazia de arestas.

2. Inicializa uma fila (queue) para armazenar os vértices a serem visitados e cria listas para acompanhar os vértices visitados e as arestas encontradas.

3. Inicia a busca a partir do vértice inicial, adicionando-o à fila e marcando-o como visitado.

4. Enquanto a fila não estiver vazia, a função continua a busca em largura:
    - Retira um vértice da fila (o vértice atual).
    - Incrementa o tempo (um contador de tempo).
    - Itera sobre os vizinhos (vértices adjacentes) do vértice atual no grafo.
    - Para cada vizinho que não tenha sido visitado, ele é adicionado à fila, marcado como visitado e a aresta que o conecta ao vértice atual é registrada.
5. A função imprime informações à medida que encontra novos vértices e arestas, incluindo o vértice pai, o vértice atual, e o tempo de descoberta.
6. A função retorna a lista de arestas encontradas durante a busca em largura.

A busca em largura é um algoritmo importante para explorar grafos e é usado para encontrar o caminho mais curto em um grafo não ponderado. Ela trabalha de forma sistêmica, explorando os vértices em camadas a partir do vértice inicial, o que a torna útil em problemas que envolvem a descoberta de conexões ou distâncias em um grafo.

Árvore Geradora Mínima (AGM)

A função menorCaminho recebe um grafo g, um vértice de origem origem e um vértice de destino destino. Ela utiliza o algoritmo de Dijkstra para encontrar o menor caminho no grafo entre o vértice de origem e o vértice de destino. O resultado é uma lista de arestas que compõem o caminho mínimo encontrado. Essas arestas são retornadas na ordem em que devem ser percorridas para ir da origem ao destino. Se não houver um caminho entre os vértices de origem e destino, a função retornará uma lista vazia.
Caminho Mínimo

Fluxo Máximo

A classe FluxoMaximo contém a implementação do algoritmo Ford-Fulkerson para calcular o fluxo máximo em um grafo.

1. fordFulkerson(Grafo grafo, Vertice s, Vertice t): Este método é responsável por calcular o fluxo máximo em um grafo. Ele recebe o grafo, o vértice de origem e o vértice de destino. O algoritmo opera em um loop enquanto houver caminhos aumentantes no grafo. Para cada caminho aumentante encontrado, ele atualiza o fluxo máximo e ajusta o vetor de fluxo peso.

2. encontrarCaminhoAumentante(Grafo grafo, Vertice origem, Vertice destino, double[] peso): Este é um método privado que encontra um caminho aumentante no grafo a partir do vértice de origem para o vértice de destino.

Ele usa uma busca em largura (BFS) para encontrar o caminho e calcula a capacidade mínima ao longo desse caminho. O método retorna a capacidade mínima do caminho aumentante encontrado.

A ideia geral do algoritmo Ford-Fulkerson é encontrar caminhos aumentantes no grafo e aumentar o fluxo ao longo desses caminhos até que não seja mais possível encontrar caminhos aumentantes. O fluxo máximo é acumulado durante esse processo.