2º Trabalho de Projeto e Análise de Algoritmos

Aluno: Matheus Kersul Souza Rodrigues

Professor: Douglas Castilho

- RESUMO

Este trabalho teve como objetivo criar e mostrar algoritmos em grafos que foram ensinados durante as aulas. Ele possui uma interface gráfica para facilitar os testes e a implementação de todos os métodos propostos pelo professor, além de um algoritmo adicional para “Caminho Mínimo”.

Mudanças do código base: adição de um construtor feito exclusivamente para fluxo máximo e adição de dois métodos na classe Aresta, ambos para consulta de HashMap

1) **Tipos de Representação**: O código consegue armazenar o grafo em três tipos diferentes: Matriz de Adjacência, Matriz de Incidência e Lista de Adjacência. O tipo de representação é determinado pelo usuário durante a inicialização do código, que também devolve um erro caso a escrita do grafo esteja incorreta.

2) **Busca em Profundidade (DFS)**: O algoritmo realiza uma varredura por todos os vértices do grafo, salvando dados como tempo de descoberta/finalização, cores para simbolizar se o vértice foi visitado e devolve um conjunto de arestas simbolizando a floresta que foi sendo construída durante o processo. O algoritmo se baseia na verificação dos vértices adjacentes para realizar a busca e além disso, ele possuí vários métodos secundários, como identificação do tipo de aresta (cruzamento, avanço e retorno), devolvendo para o usuário todas as informações encontradas.

3) **Busca em Largura (BFS)**: Esse algoritmo precisa de uma origem v para ser inicializada (o default é o vértice 0), e ele utiliza uma fila (LinkedList) para buscar os vértices adjacentes de v, e em seguida ele verifica os adjacentes dos vértices encontrados (u) e continua até não achar mais nenhum vértice. Ele salva dados como cores, para verificar se o vértice já foi visitado; relação pai e filho (pi), para indicar qual é o antecessor de cada vértice; e arestas encontradas, que retornam o resultado da busca.

4) **Existe Ciclo**: Informa se o grafo possuí um ou mais ciclos. Essa informação é detectada e armazenada durante a Busca em Profundidade, que segue uma lógica bem simples: se um vértice “cinza” (já visitado porém não visitou todos os adjacentes) encontra outro vértice “cinza”, existe ciclo, já que a busca poderia dar a volta por onde começou.

5) **Árvore Geradora Mínima (AGM)**: Essa implementação usa o algoritmo de Kruskal, que funciona de forma gulosa. O algoritmo ordena todas as arestas da árvore de forma crescente e armazena os vértices em um HashMap (vértices, vértices), que de início cada chave aponta para o próprio vértice. Em seguida

ele roda todas as arestas e verifica se o grupo do vértice aresta.origem contém o Vértice da aresta.destino. Se não, ele agrupa esses vértices e adiciona essa aresta em uma ArrayList “Ramos da Árvore”, indicando que essa aresta respeitou as regras do algoritmo de Kruskal e que faz parte da árvore geradora mínima. Além disso, o algoritmo também devolve o custo dessa árvore, que é somatória dos pesos das arestas.

6) **Caminho mínimo (Dijkstra)**: O algoritmo se baseia numa busca por vértices não visitados, que determina que o vértice adjacente não visitado e que possua uma menor distância máxima acumulada desde a origem seja colocado como filho do vértice antecessor (novamente a relação pai e filho), ele necessita de dois vértices de origem e destino para funcionar. Foi usado dois HashMaps para guardar a distância e os filhos, que são alterados sempre que o algoritmo identifica um caminho com o peso menor (método de relaxamento), consequentemente rodando até que a lista de não visitados esteja vazia. No final ele retorna o caminho mínimo entre origem e destino reconstruído e o tempo gasto pelo mesmo.

7) **Caminho mínimo (Bellman Ford)**: Tendo quase a mesma base que o algoritmo de dijkstra, usando relações de pai e filhos e distâncias, o algoritmo de Bellman Ford realiza uma busca pelos grafos (repete por V - 1) e identifica arestas que podem ser relaxadas, ou seja, possuem um caminho mínimo menor que o anterior. A principal diferença entre esse e Dijkstra é que não se utiliza fila para verificar os vértices não visitados, já que o algoritmo irá necessariamente rodar por todas as possibilidades possíveis. Ele também pode identificar ciclos negativos, que no caso é o retorno de null pelo método e é informado pela interface.

8) **Fluxo máximo (Ford Fulkerson)**: Foi usado o algoritmo de Ford Fulkerson, que possuí conceitos como grafos residuais, arestas residuais e arestas transpostas. Ele determina que a capacidade inicial de cada aresta é o peso e que o fluxo da mesma é 0.0; em seguida ele realiza uma busca por largura modificada para esse método: é definido uma nova classe de dados, composta por (Aresta aresta original, Double capacidade residual, Boolean residual reversa) e realiza o mesmo processo de busca BFS explicado anteriormente, só que dessa vez verificando se o conjunto entregue pode ter alguma capacidade residual disponível. Se sim, o conjunto de informações é alterado mudando a capacidade residual e determinando ambas arestas residuais (diretas e reversas), por fim ele remonta o caminho com as arestas ajustadas e retorna o caminho construído para o looping de fluxo_maximo. Dentro do fluxo máximo ele recebe esse caminho e indica que o fluxo máximo é o valor antigo + o gargalo encontrado nessa busca BFS.

   Observações: neste algoritmo foi preciso adicionar dois métodos adicionais na classe Aresta: equals e hashCode, ambos usados para implementar os dados de fluxo máximo, que constantemente eram iniciados como nulo independentemente de ações de segurança tomadas.

9) **Componentes Fortemente Conectados (Kosaraju)**: Esse algoritmo encontra determinados grupos de vértices fortemente conectados, onde se pode dizer que é como “dividir por ciclos” dentro de uma Busca em Profundidade.
   Ele realiza uma pesquisa DFS no grafo original e guarda o tempo de descoberta/finalização de cada vértice, gerando um grafo transposto (invertendo a orientação das arestas). Depois ele realiza uma busca DFS nesse novo grafo seguindo a ordem decrescente do tempo de finalização do grafo original,

conseguindo assim filtrar os componentes principais de cada “sub-árvore”. Em seguida ele realiza diversas comparações com o tempo de descoberta/finalização de cada componente principal e resgata as arestas do grafo original, remontando um grafo reduzido e devolvendo para o usuário esse mesmo grafo.

Observação: Os vértices que compõem o grafo reduzido possuem novos ids, porém salva as arestas do grafo normal, como dito anteriormente.