PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Instituto de Ciências Exatas e de Informática

Trabalho Prático 2*

Model - Pratical Exercise - ICEI - Puc Minas

Henrique Castro[1]
Leonardo Gomide[2]

Resumo

Este exercício prático contem a documentação do Trabalho Prático 2, feita em LaTeX. Em grafos simples, pode-se encontrar sequências de vértices que vão de um ponto a outro, essa sequência, quando não repete arcos, é denominada de caminho. Tendo isso em vista, um problema é encontrar o número de caminhos disjuntos de um grafo simples qualquer, sendo caminhos disjuntos aqueles que não possuem arestas em comum. Dessa forma este trabalho propõe uma forma de lidar de com esse problema, por meio de algoritmos conhecidos dentro da área de estudo de grafos.

Palavras-chave: LaTeX. Exercício Prático. Java.

*Artigo apresentado como documentação ao Instituto de Ciências Exatas e Informática da Pontifícia Universidade Católica de Minas Gerais para disciplina Teoria dos Grafos e Computabilidade.

[1]Curso de Ciência da Computação PUC Minas, Brasil– https://github.com/HenriqueCastros
[2]Curso de Ciência da Computação PUC Minas, Brasil– https://github.com/GomideLeo

1 INTRODUÇÃO

Dentro de grafos simples existem diversos conceitos e aplicações que vêm sido estudadas ao longo de décadas. Dentro dos conceitos mais clássico, temos a definição de caminho, que resumidamente é uma sequência de vértices onde arestas não são repetidas.

Um comportamento interessante de caminhos são os caminhos disjuntos, que são dois ou mais caminhos de um mesmo grafo, que não possuem nenhuma aresta em comum entre si. Uma vez entendido esse conceito, torna-se interessante encontrar o número máximo de caminhos disjunto de um grafo.

Para tentar solucionar esse problema, foi proposto pensar em um grafo ponderado, onde todas arestas possuem peso igual. Tendo esse novo grafo, ao calcular o fluxo máximo aparenta ser possível encontrar esse número máximo. As seções seguintes descrevem a implementação, os testes e conclusões sobre resultados encontrados.

2 IMPLEMENTAÇÃO

Para desenvolver a solução do trabalho, utilizamos a linguagem Java. Ela foi escolhida por ser uma linguagem que permite implementar facilmente orientação de objetos e ser relativamente rápida.

Foram criadas diferentes classes que se complementam na solução, das quais se destacam a classe *Grafo*, que modela um grafo qualquer e possui os métodos necessários para resolver o problema e as classes utilitárias *Edge* e *Tuple*, usadas para encapsular os conceitos de vértice e tuplas, facilitando a implementação. Todas as classes foram escritas de forma a rodar em uma única *Thread* de execução.

2.1 Classe Grafos

A classe Grafo foi desenvolvida para ser a principal estrutura de dados utilizada durante o trabalho. Apesar dos exercícios serem para grafos direcionados, implementamos a classe de forma que ela também para grafos não direcionados, o atributo *isDirectional* faz o controle dessa característica do grafo, tendo o *default* como verdadeiro.

Utilizamos de uma matriz de adjacência para armazenar as arestas e seus respectivos pesos, dentro da classe a matriz pode ser acessada por meio do atributo *edgesWeights*. Fora a facilidade de utilização e implementação, a matriz de adjacência permite extrair todas as arestas que saem de um dado vértice, por meio do método *getAllEdgesFromNode*. dada uma origem é possível atingir outro vértice, por meio da busca em largura.

Para os construtores da classe, definimos uma estrutura de arquivo que pode ser importada pelo construtor para facilitar os testes, neste arquivo, temos o cabeçalho, com uma linha

e três valores que ditam quantos vértices o grafo possui e de qual à qual vértice os caminhos serão procurados. No corpo, cada linha representa uma aresta direcional, com dois valores representando quais vértices a aresta liga, sendo o primeiro a origem e o segundo o destino.

Fora os métodos explicitados acima, construímos outros métodos auxiliares que utilizam pesos na matriz de adjacência, adicionar vértices ou arestas, e outras funções que julgamos necessárias.

2.2 Algoritmo

Para buscar os caminhos disjuntos em um grafo, interpretamos o grafo direcional como um problema de fluxo máximo e implementamos uma variação do algoritmo de Edmonds-Karp, mas como todas as arestas possuem a mesma capacidade, utilizar as arestas de retorno de uma rede residual não geraria nenhum aumento no fluxo máximo, buscar os caminhos mais curtos por meio de uma busca em largura é suficiente para obter o resultado ótimo.

Como já comentado acima, nossa implementação é uma variação do método desenvolvido por Edmonds-Karp para obter o fluxo máximo de uma rede de fluxo. É gerada uma cópia do grafo original e buscamos o menor caminho em quantidade de arestas, por meio de uma busca em largura, enquanto existirem caminhos e os adicionamos na resposta. Cada aresta pode ser usada uma única vez por conta da capacidade constante da rede de fluxo, então após utilizadas, todas as arestas de um caminho são removidas da cópia.

3 EXPERIMENTOS

Para realizar os experimentos foram desenvolvidos alguns grafos para testes, baseados em exemplos mostrados em sala de aula, para este relatório vamos focar nos representados nas figuras 1 e 2. Nas figuras, a origem foi sempre o vértice *s* e o destino o vértice *t* arestas que foram utilizadas para algum caminho foram representadas em vermelho.

No grafo *K5*, foram encontrados 4 caminhos, um direto de *s* a *t* e os outros três passando por um dos outros vértices do grafo. É interessante notar que apesar de ser possível realizar loops, como o passeio *s 1 2 1 t* ao invés do *s 1 t* por conta do uso da busca em largura, eles são evitados.

No grafo *G*, foram encontrados 3 caminhos, que novamente por conta da busca pelos menores caminhos evitou casos onde 2 ou menos caminhos seriam encontrados.

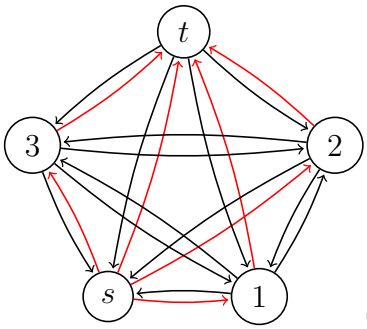

Figura 1 – K5

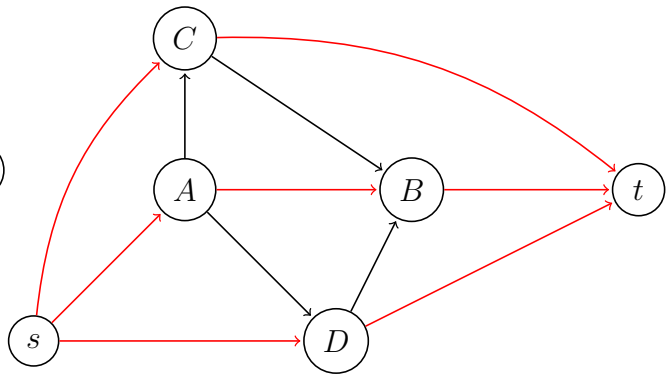

Figura 2 – G

4 CONCLUSÃO

Caminhos disjuntos possuem diversas aplicações e encontrá-los em um grafo grande pode se apresentar como um problema, portanto um algoritmo que consiga calcular esses caminhos de maneira ótima computacionalmente pode ser usado em diversas aplicações, desde redes de computadores até logística de entrega.

Utilizando conceitos que foram usados em outros problemas conseguimos transformar o problema e adaptar soluções já existentes para um novo problema de forma a facilitar a resolução garantindo a eficiência do método.