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

Experimentos com Caminhos Disjuntos: Teoria dos Grafos e Computabilidade

Luiz Fernando Oliveira Maciel[1]
Vinicius Ferreira de Souza[2]

Resumo

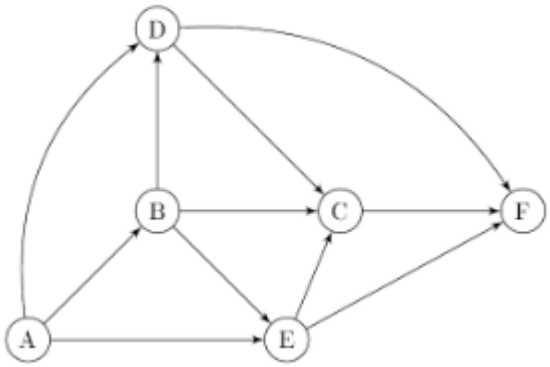
Este artigo tem como objetivo apresentar o algoritmo desenvolvido para encontrar caminhos disjuntos em grafos e analisar o comportamento desse algoritmo em diferentes tipos de grafos. Artigo feito como parte do Trabalho Prático 2 da matéria Teoria dos Grafos e Computabilidade, do 4º período do curso de Ciência da Computação da PUC Minas PPL.

Palavras-chave: Teoria dos Grafos e Computabilidade. Caminhos Disjuntos.

[1]Aluno do Programa de Graduação em Ciência da Computação, Brasil – lfomaciel@sga.pucminas.br.
[2]Aluno do Programa de Graduação em Ciência da Computação, Brasil – vfsouza@sga.pucminas.br.

1 INTRODUÇÃO

Um caminho simples em um grafo direciado é um caminho sem repetição de vértices. Ou seja, um caminho simples é uma sequência de vértices e arestas sem repetir o mesmo vértice mais de uma vez.



No gráfico acima, as sequências de vértices A, B, E, F e a sequência A, D, C, F são caminhos simples. Essas duas sequências também são caminhos disjuntos em arestas, pois não possuem arestas iguais entre si.

O problema de achar os caminhos disjuntos em arestas em um grafo possui vários usos em diversas áreas. Para isso, nós implementamos um algoritmo capaz de achar quantos e quais caminhos disjuntos existem em um grafo gerado de forma automática.

2 RESOLUÇÃO

Para resolvermos o problema, inicialmente assumimos que todas as arestas no grafo possuem capacidade de fluxo unitária. Considerando que um grafo possua um número k máximo de caminhos disjuntos de um vértice s à um vértice t, podemos dizer que o fluxo máximo do vértice s ao vértice t será k (já que a capacidade de fluxo é unitária em todos as arestas). (WAYNE, 2001) Para encontrarmos o número máximo de caminhos disjuntos precisamos, portanto, apenas encontrar o fluxo máximo do grafo, e a partir do corte das arestas que possuem fluxo maior que 0 utilizarmos força bruta para encontrar um conjunto de caminhos disjuntos que possua o mesmo número de elementos que o valor máximo encontrado.

3 IMPLEMENTAÇÃO

Nosso algoritmo foi desenvolvido em Python devido à sua alta abstração e consequente facilidade de implementação. Link do código-fonte: <https://github.com/lfnand0/Grafos-TP2>

Os grafos foram representados como matrizes de adjacência, onde a coluna representa o vértice de saída e a fileira o vértice de entrada, e o valor nessa posição é 1 caso exista uma aresta

e 0 caso contrário. Para encontrar o fluxo máximo, utilizamos o algoritmo de Edmonds-Karp (GOLDREICH et al., 2006). Ao fim da execução do algoritmo, temos uma matriz com o fluxo residual - como precisamos apenas do corte onde existe fluxo, comparamos a matriz do grafo com o fluxo residual e geramos uma nova matriz (chamada de dif) seguindo as seguintes regras: para cada posição em uma matriz n x n: caso o valor seja 0 nessa posição na matriz do grafo, o elemento na posição na matriz dif será 0; caso seja 1, o elemento na matriz dif receberá o resultado da operação xor entre 1 e o elemento na matriz do fluxo residual. Visualização:

Figura 1 – Um grafo direcionado qualquer e sua matriz de adjacência

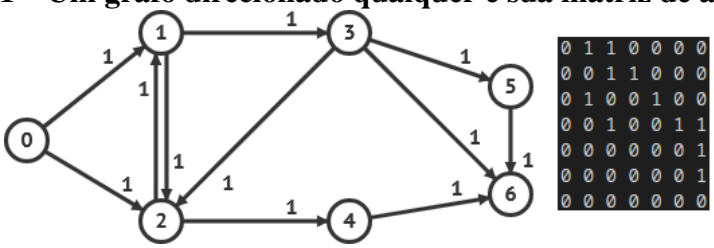


Figura 2 – A matriz do fluxo residual do vértice 0 ao 6 encontrado após a execução do algoritmo de Edmonds-Karp

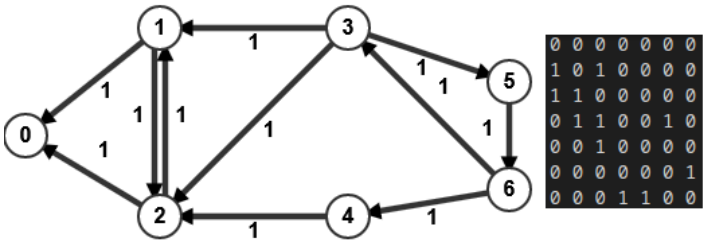
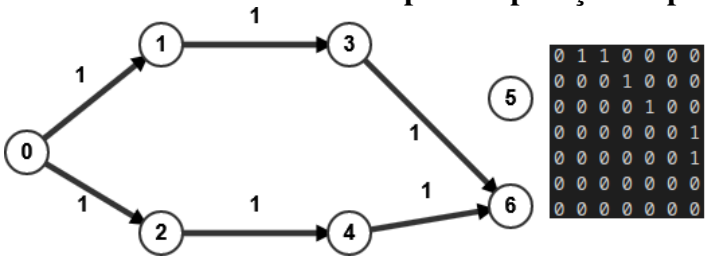


Figura 3 – A matriz "dif"encontrada após as operações explicadas acima



Partindo dessa nova matriz "dif", realizamos primeiramente uma busca em profundidade, encontrando todos os caminhos do vértice origem ao destino e os salvando em uma lista. Depois criamos uma lista vazia chamada "disjuntos"para os caminhos disjuntos e "visitadas"para todas as arestas visitadas, e executamos o seguinte código:

Algoritmo 1 - Encontrar conjunto de caminhos disjuntos

Algorithm 1: Encontrar conjunto de caminhos disjuntos

Preenche o vetor de $attributes.size + 1$ atributos com os valores dos atributos, sendo a primeira posição do vetor preenchida com o valor 1
$hidden\_layer\_size = attributes.size * 2 + 1;$
for caminhoA in caminhos **do**
    $disjuntos[] = [].append(caminhoA);$
    $visitadas = [].extend(caminhoA);$
    for $caminhoB$ en $caminhos - caminhoA$ **do**
        if $nenhumaarestadecaminhoB$ em $visitadas$ **then**
            $disjuntos.append(caminhoB);$
            $visitadas.extend(caminhoB);$
        end if
    end for
    if $disjuntos.length$ == $maxflow$ **then**
        return $disjuntos$;
    end if
end for

Após a execução desse algoritmo, temos a lista com o conjunto de caminhos disjuntos.

4 UTILIZAÇÃO DO CÓDIGO

Para utilizar o código, é necessário ter instalado o Python em uma versão igual ou superior à 3. Nosso código apresenta diversas flags para chamada inline, para que o usuário consiga utilizar da maneira que deseja. O usuário pode utilizar quantas e quais flags quiser. As flags que podem ser utilizadas são as seguintes:

4.1 -prob [float]

A probabilidade que dois vértices tenham uma aresta em comum na geração de grafos aleatórios. Caso não seja utilizado a probabilidade será 0.3.

Exemplo: python tp2.py -prob 0.1

4.2 -arq [string]

Diretório de um arquivo txt com a matriz de adjacência de um grafo.

Exemplo: python tp2.py -arq "./grafos cíclicos/50.txt"

4.3 -ver [int]

Número de vértices a ser usado na geração de grafo. Caso não seja utilizado o número de vértices será 10.

Exemplo: python tp2.py -ver 100

4.4 -ori [int]

Vértice de origem. Caso não seja utilizado, o vértice de origem será 0.

Exemplo: python tp2.py -ori 1

4.5 -des [int]

Vértice de destino. Caso não seja utilizado, o vértice de destino será n - 1.

Exemplo: python tp2.py -des 9

4.6 -cam [string]

Diretório para salvar os caminhos disjuntos encontrados. Caso não seja utilizado, os caminhos serão salvos em "./disjuntos.txt"

Exemplo: python tp2.py -cam "./out/disj.txt"

4.7 -w [string]

Diretório para salvar o grafo gerado. Caso não seja utilizado o grafo gerado não será salvo

Exemplo: python tp2.py -w "./grafos simples/50.txt"

4.8 -ciclo

Gera um grafo cíclico.

Exemplo: python tp2.py -ciclo

4.9 -completo

Gera um grafo completo
Exemplo: python tp2.py -completo

5 GERAÇÃO DE GRAFOS ALEATÓRIOS

Nosso algoritmo também é capaz de gerar grafos cíclicos, simples e completos de forma automática para serem testados pelo nosso método de busca de caminhos disjuntos em arestas.

Para a geração de grafos simples, utilizamos o Modelo Erdős–Rényi, que adiciona arestas ao grafo a partir de uma probabilidade definida.(ERDÖS; RÉNYI, 1959). Para grafos completos, chamamos a função que gera grafos simples, porém passando uma probabilidade de 1 (100%) - ou seja, existe uma aresta saíndo de cada vértice do grafo para todos os outros. Para grafos cíclicos, apenas adicionamos uma aresta saíndo de um vértice e indo para seu sucessor, e ao chegar no maior índice retornamos ao primeiro vértice.

6 EXPERIMENTOS COM CAMINHOS DISJUNTOS

6.1 Resultados Obtidos

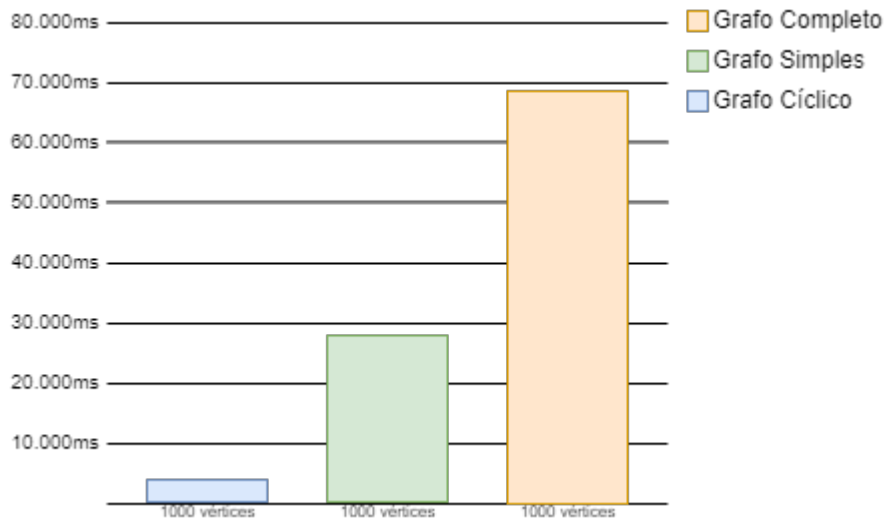
Apresentação dos resultados obtidos em cada tipo de grafo com tamanhos diferentes. Cada tabela possui qual tipo de grafo que testamos bem como o número de vértices e o tempo de execução.

Grafos Completos		
Tamanho do grafo	**Tempo de execução**	**Número de Caminhos Disjuntos**
50 vértices	215,8ms	49
100 vértices	287,8ms	99
250 vértices	1304,5ms	249
1000 vértices	69181,17ms	999

Grafos Cíclicos		
Tamanho do grafo	**Tempo de execução**	**Número de Caminhos Disjuntos**
50 vértices	18,6ms	1
100 vértices	230,48ms	1
250 vértices	311,75ms	1
1000 vértices	4658ms	1

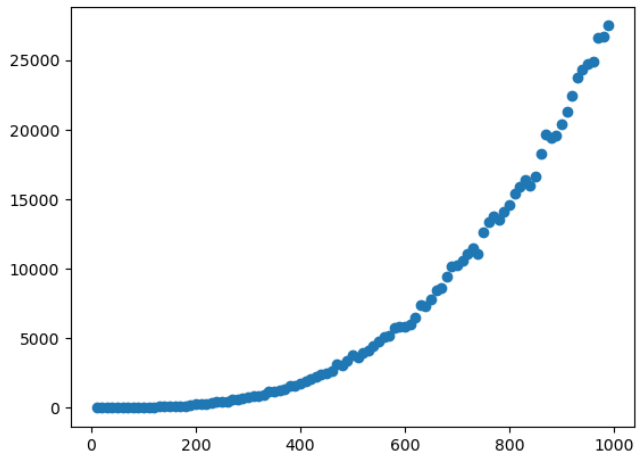
Grafos Simples		
Tamanho do grafo	**Tempo de execução**	**Número de Caminhos Disjuntos**
50 vértices	163,72ms	31
100 vértices	275,34ms	47
250 vértices	765,37ms	94
1000 vértices	27985,30ms	382

Para comparar de forma melhor, segue abaixo um gráfico com a comparação de tempo entre os diferentes tipos de grafos com mil vértices.



6.2 Regressão

Figura 4 – Grafo do nº de vértices x tempo de execução (ms)



A figura acima representa um gráfico do número de vértices pelo tempo de execução. Para encontrarmos a complexidade de tempo do programa, realizamos o teste com vários grafos

simples de tamanhos diferentes. Geramos os grafos com tamanhos incrementais de 10 a 10, gerando de tamanho 10 à 990, com a probabilidade de existir aresta entre dois vértices quaisquer igual à 50%. A função encontrada na regressão para o tempo de execução (em milisegundos) de acordo com o número de vértices foi de $T(n) = 0.04293861x^2 - 17.17452933x$, com um coeficiente muito preciso de $R^2 = 0.995024467$. Com isso, podemos dizer que nosso programa possui complexidade de tempo quadrática.

6.3 Conclusão

O programa roda com uma complexidade de $O(n^2)$. Por utilizarmos o corte do grafo encontrado com o algoritmo de Edmonds-Karp, a complexidade é reduzida em um grau de magnitude - o pior caso é em grafos completos, onde o número de arestas é reduzido de $n*(n-1)/2$ para $2n-3$. Por conseguirmos parar a execução do método de força-bruta ao encontrarmos um conjunto de caminhos disjuntos com o mesmo valor do fluxo máximo, o algoritmo também é mais rápido em casos médios.

REFERÊNCIAS

ERDÖS, P; RÉNYI, A. On random graphs i. **Publicationes Mathematicae Debrecen**, v. 6, p. 290–297, 1959. Disponível em: <https://www.renyi.hu/~p_erdos/1959-11.pdf>.

GOLDREICH, Oded; ROSENBERG, Arnold L.; SELMAN, Alan L. (Ed.). **Theoretical Computer Science: Essays in Memory of Shimon Even**. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN 3540328807.

WAYNE, Kevin. **Maximum Flow Applications**. Princeton University, 2001. Disponível em: <https://www.cs.princeton.edu/ wayne/cs423/lectures/max-flow-applications/>. Acesso em: 11 de dez. 2022. Disponível em: <https://www.cs.princeton.edu/~wayne/cs423/lectures/max-flow-applications>.