

Identificação do número máximo de caminhos disjuntos em arestas*

Model - Magazine Abakós - ICEI - PUC Minas

Bruno Rodrigues Faria¹
Guilherme Dantas Caldeira Fagundes²
Laura Iara Silva Santos Xavier³

Resumo

Há vários problemas a serem tratados com a descoberta do número máximo de caminhos disjuntos em arestas presentes em um grafo. O trabalho irá tratar de um método que faz a descoberta de quais são os caminhos além de mostrar os mesmos para o usuário com o intuito de auxiliar em futuros desenvolvimentos. Para isso, basta criar um grafo direcionado e falar qual a origem e destino, pois um caminho deve sair de um lugar e chegar a outro para que assim possa descobrir os caminhos disjuntos, o trabalho foi realizado na linguagem de programação Python e está disponibilizado para todos terem acesso.

Palavras-chave: Grafos. Arestas. Busca. Caminhos. Disjuntos.

*Identificação de caminhos, disjuntos, arestas

¹Programador, E-mail:bruno.faria@sga.pucminas.br
Graduação Ciências da Computação - PUC Minas, Brasil.

²Programador, E-mail:gdcfagundes@sga.pucminas.br
Graduação Ciências da Computação - PUC Minas, Brasil.

³Programador, E-mail:laura.iara@sga.pucminas.br
Graduação Ciências da Computação - PUC Minas, Brasil.

1 INTRODUÇÃO

Atualmente a aplicação de grafos vem sendo cada vez mais utilizado no mundo da computação, por ser bastante versátil na resolução de problemas. Um exemplo disso é o aplicativo Waze que utiliza de grafos para traçar o caminho a ser percorrido para chegar de um local a outro. Pode-se observar com isso, que a descoberta dos caminhos disjuntos teria uma boa aplicação nesse aplicativo, pois caso o motorista não queira utilizar do caminho disponibilizado pelo aplicativo ele pode solicitar um novo caminho. Com isso, utilizando o método proposto no trabalho pode-se descobrir um novo caminho completamente diferente para o motorista, caso esse caminho exista, para que ele possa ir da origem até o destino de interesse.

2 DESENVOLVIMENTO

2.1 Entrada de dados

Para a entrada de dados, foi utilizada a linha de comando para passar os argumentos. Os argumentos que são passados para o programa são respectivamente: o arquivo contendo a criação do grafo ou nome do arquivo que deseja criar o grafo; o vértice de origem para a busca dos caminhos; o vértice de destino para a busca chegar até ele.

A partir disso, para criar um grafo, basta fazer um arquivo *.txt* passando na primeira linha do arquivo a quantidade de vértices, após isso cada linha do arquivo deve ser uma aresta presente no grafo direcionado, sendo da seguinte forma *vértice_saída vértice_entrada*, onde vértice de saída é de onde a aresta sai e o vértice de entrada é onde a aresta entra, mostrado na Figura 1. Essa situação se encaixa caso o grafo que deseja seja diferente dos grafos de teste abordados na Seção 3 do artigo.



Figura 1 – Exemplo de aresta direcionada

Com isso, ao passar por parâmetro o vértice de origem e o vértice de destino, eles serão utilizados para calcular o número máximo de caminhos disjuntos para o grafo desejado. Mostrando também todos os caminhos percorridos, utilizando os vértices passados, a Figura 2 onde exemplifica cada caminho disjunto em aresta, que é um caminho que sai de um vértice de destino até um vértice de origem sem repetir a aresta novamente.

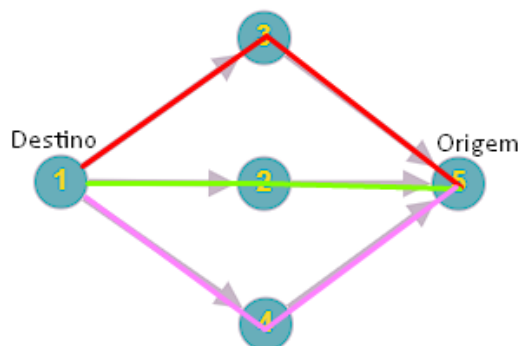


Figura 2 – Caminhos disjuntos em arestas em um grafo

2.2 Descobrir caminhos disjuntos

O algoritmo desenvolvido para encontrar a maior quantidade de caminhos possíveis em um grafos direcionado partindo de um vértices s e até um vértice t utilizando os n vértices do grafo.

O método executa uma busca em profundidade tentando encontrar o vértice t . Ao encontrá-lo, o caminho utilizado é salvo e as arestas pertencentes a esse caminho são invertidas em um grafo auxiliar, ou seja, se ela era (v, w) ela se torna (w, v) . Tal alteração parte da mesma ideia presente nos métodos de cálculo de fluxo máximo que é de consumir toda a capacidade da aresta e invertê-la.

A cada caminho encontrado a variável responsável por contabilizar a quantidade de caminhos disjuntos é incrementada em 1. O método se repete até que mais nenhum caminho seja encontrado.

Algorithm 1 Encontrar a maior quantidade possível de caminhos disjuntos em um grafo direcionado

Require: ($origin \in n \wedge destiny \in n$)

```
1:  $auxGraph \leftarrow copy(graph)$ 
2:  $parent \leftarrow []$ 
3:  $maxPaths \leftarrow 0$ 
4:  $paths \leftarrow NULL$ 
5: while  $auxGraph.search(origin, destiny, parent)$  do
6:    $reversedPath \leftarrow NULL$ 
7:    $v \leftarrow destiny$ 
8:   while  $v \neq origin$  do
9:      $u \leftarrow parent[v]$ 
10:     $reversedPath.append(v)$ 
11:     $auxGraph.removeEdge(u, v)$ 
12:     $auxGraph.addEdge(v, u)$ 
13:     $v \leftarrow parent[u]$ 
14:   end while
15:    $reversedPath.append(origin)$ 
16:    $path \leftarrow reversedPath.popAllElements()$ 
17:    $paths.append(path)$ 
18:    $maxPaths \leftarrow maxFlow + 1$ 
19: end while
20: return  $maxPaths, paths$ 
```

3 RESULTADOS E TESTES

Os testes foram feitos em 3 tipos diferentes de topologias, cada uma seguindo sua regra específica de formação, elas serão nomeadas como *Topologia 1*, *Topologia 2*, *Topologia 3*. Além disso, cada topologia foi testada utilizando quantidades diferentes de vértices, sendo elas: 10, 100, 500, 1000 e 10.000.

3.1 Topologia 1 - Grafo Circular

A *Topologia 1* segue a ideia padrão de um grafo cíclico, direcionado, não ponderado e sem arestas antiparalelas. Ele é formado por n vértices e n arestas.

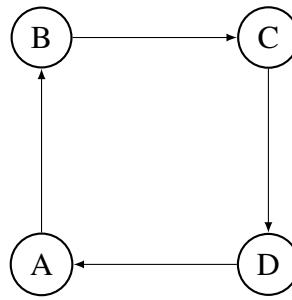


Figura 3 – Grafo cíclico sem arestas antiparalelas

Algorithm 2 Gerar grafo circular

Require: $n > 0$

$i \leftarrow 0$

$edges \leftarrow NULL$

for $i < n$ **do**

if $i \neq (n - 1)$ **then**

$edges += (i, i + 1)$

end if

end for

$edges += (n - 1, 0)$

return $edges$

No algoritmo acima, o n representa a quantidade de vértices desejado, ela deve ser maior do que 0. Já a variável $edges$ é formada por uma lista de tuplas, em cada tupla representa uma aresta (v, w) , em que ela sai de v e incide em w .

3.2 Topologia 2 - Grafo Simples

A *Topologia 2* baseia-se na ideia de possuir n vértices, em que um vértice possui um grau de saída igual a $n - 1$ e grau de entrada igual a 0 e um outro vértice que possui um grau de saída igual a 0 e um grau de entrada igual a $n - 1$. Na prática, a geração baseia-se em adicionar uma aresta (v, w) para todo par de vértices em que v preceda w em um conjunto de vértices ordenado lexicograficamente.

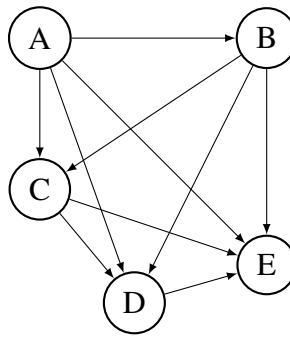


Figura 4 – Grafo formado por 5 vértices seguindo regra específica de formação

Algorithm 3 Gerar grafo com regra de formação específica

Require: $n > 0$

$i \leftarrow 0$

$edges \leftarrow NULL$

for $i < n$ **do**

$j \leftarrow i$

for $j < n$ **do**

if $i \neq j$ **then**

$edges += (i, j)$

end if

end for

end for

return $edges$

3.3 Topologia 3 - Grafo Completo

A terceira e última topologia, chamada de *Topologia 3* utiliza de dois $K-n$ grafos (grafos completos formados por n vértices). A partir desses dois grafos, que podem ser chamados de G_1 e G_2 , são adicionadas duas arestas que ligam os grafos partindo de G_1 e incidindo em G_2 .

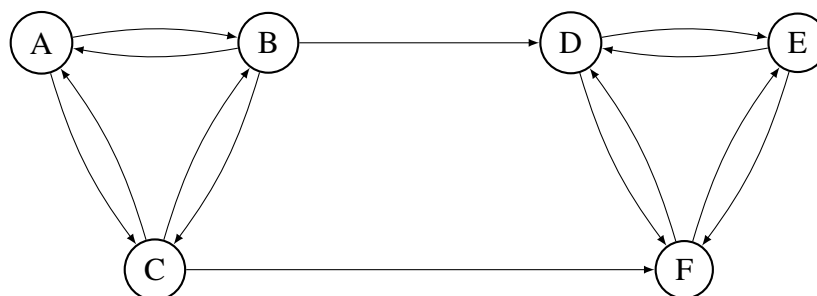


Figura 5 – Dois grafos K3 ligados por duas arestas

Algorithm 4 Gerar dois K-N grafos ligados por duas arestas**Require:** $n > 0$

```

1:  $edges_{k1} \leftarrow getCompleteGraph(0, n/2)$ 
2:  $edges_{k2} \leftarrow getCompleteGraph((n/2), n)$ 
3:  $edges \leftarrow edges_{k1} + edges_{k2}$ 
4:  $extraEdge_1 \leftarrow (getRandomEdge(0, (n/2) - 1), getRandomEdge((n/2), n - 1))$ 
5:  $extraEdge_2 \leftarrow (getRandomEdge(0, (n/2) - 1), getRandomEdge((n/2), n - 1))$ 
6: while  $extraEdge_1 = extraEdge_2$  do
7:    $extraEdge_1 \leftarrow (getRandomEdge(0, (n/2) - 1), getRandomEdge((n/2), n - 1))$ 
8:    $extraEdge_2 \leftarrow (getRandomEdge(0, (n/2) - 1), getRandomEdge((n/2), n - 1))$ 
9: end while
10:  $edges += extraEdge_1$ 
11:  $edges += extraEdge_2$ 
12: return  $edges$ 

```

Algorithm 5 Gerar K-N grafo**Require:** $(start \in Z) \wedge (end \in Z \wedge start \neq end)$

```

1:  $edges \leftarrow NULL$ 
2: for  $i \leftarrow start, end$  do
3:   for  $j \leftarrow start, end$  do
4:     if  $i \neq j \wedge (i, j) \notin edges \wedge (j, i) \notin edges$  then
5:        $edges += (i, j)$ 
6:        $edges += (j, i)$ 
7:     end if
8:   end for
9: end for
10: return  $edges$ 

```

3.4 Tempo de execução

Para os resultados dos testes, achando os determinado tempos de execução para cada situação foi levado em consideração, que os vértices usados como origem e destino seriam respectivamente o primeiro vertice do grafo, ou seja, 0 e o vértice $n - 1$, sendo n a quantidade de vértices presentes no grafo. Para que assim, tenha uma maior quantidade de possibilidades de caminhos disjuntos entre a origem e o destino. Os experimentos foram feitos em uma máquina com as seguintes configurações:

- **Processador:** i7-11390H - 3.4GHz - 2918Mhz - 4 núcleos
- **Memória:** 16GB - DDR4 - 3200MHz

- **Windows 11**
- **SSD: 500GB**

Tempo para execução por topologia (segundos)			
<i>Quantidade de Vértices</i>	<i>Topologia 1</i>	<i>Topologia 2</i>	<i>Topologia 3</i>
<i>10</i>	0.000s	0.000s	0.000s
<i>100</i>	0.001s	0.010s	0.002s
<i>500</i>	0.008s	1.930s	0.014s
<i>1000</i>	0.010s	14.130s	0.210s
<i>10000</i>	0.022s	98.913s	X

Com isso, uma análise dos gráficos de tempo em relação a quantidade de arestas dos grafos foram feitas, para cada topologia criada no trabalho.

3.4.1 Topologia 1 - Grafo Circular

Pode-se observar que no grafo circular o tempo de execução no início, onde a quantidade de arestas é menor o tempo de execução acompanha a quantidade de arestas. Com isso, pode-se concluir que ao aumentar o grafo o tempo de execução também aumenta.



Figura 6 – Análise de tempo x quantidade de arestas - Grafo Circular

3.4.2 Topologia 2 - Grafo Simples

Observa-se que no grafo simples o tempo de execução aumenta linearmente conforme a quantidade de arestas aumenta no grafo, para isso você deve levar em consideração a busca no grafo, que irá continuar sendo executada enquanto o algoritmo encontrar um caminho entre a origem e o destino.

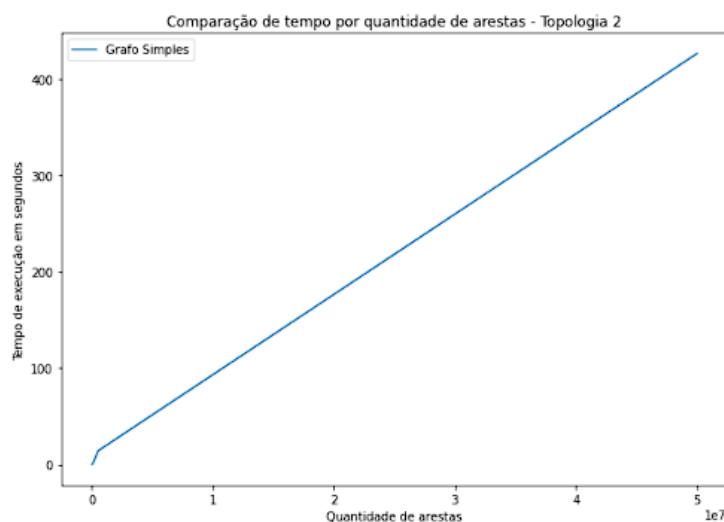


Figura 7 – Análise de tempo x quantidade de arestas - Grafo Simples

3.4.3 Topologia 3 - Grafo Completo

Tendo em vista o grafo completo e sua ligação dos K-N grafos completos por apenas dois caminhos, observa-se que o tempo de execução aumenta conforme a quantidade de arestas presentes no grafo, o que é algo esperado já que os caminhos ficariam mais longos para a busca percorrer, sendo linear o tempo de acordo com a quantidade de arestas.

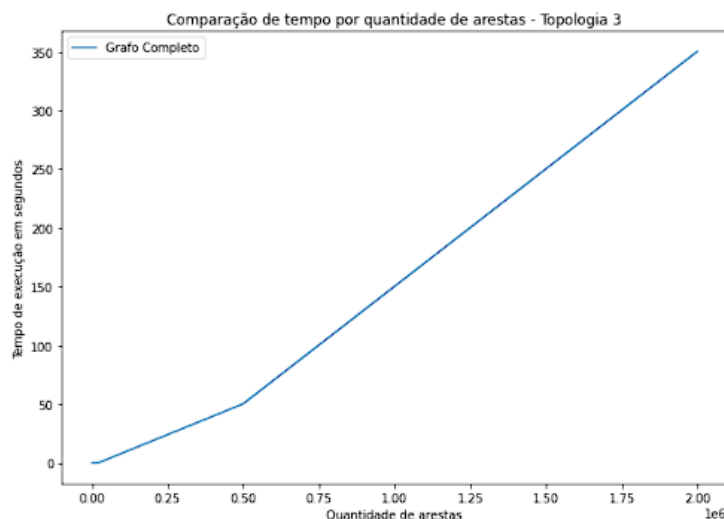


Figura 8 – Análise de tempo x quantidade de arestas - Grafo Completo

3.4.4 Análise tempo por quantidade de vértices

Para essa análise foi feita a quantidade utilizada a quantidade de vértices presentes no grafo. Com isso, teremos a análise de qual o grafo que tem o maior tempo de execução com certa quantidade de vértices, a mesma quantidade citada no início da Seção 3 para todos os testes.



Figura 9 – Análise de tempo x quantidade de vértices

Podemos observar no gráfico da Figura 9 que o grafo simples o qual tem uma possibilidade maior de caminhos possui o tempo de execução maior que os demais, já o grafo completo que possui 2 caminhos, possui o tempo de execução maior que o grafo circular que tem apenas 1 caminho presente no grafo. Com isso, pode-se ter certeza que o tempo de execução está diretamente relacionado com a quantidade de arestas e caminhos presentes entre uma origem e o destino.

4 CONCLUSÃO

Após a aplicação e testes do método podemos concluir que a aplicação da identificação da quantidade máxima de caminhos disjuntos é importante para o cenário de grafos, um exemplo disso é a aplicação desse resultado no *Teorema de de Menger*, onde o número mínimo de vértices que separam A de B é igual ao número máximo de caminhos disjuntos que ligam A a B (SEYMOUR, 1980).

Visto isso, observamos que o algoritmo se baseia em uma busca e os conceitos de caminhos aumentantes onde, após escolher o caminho você deve inverter as arestas. Com isso, observa-se que o algoritmo tem uma implementação simples e obtém resultados muito bons para grafos com poucos caminhos entre uma origem e um destino, porém caso o grafo tenha uma quantidade muito grande de caminhos disjuntos entre um par de vértices o tempo de execução irá aumentar consideravelmente como mostrado nas análises dos gráficos dos testes presentes na Seção 3.

Referências

SEYMOUR, P.D. Disjoint paths in graphs. **Discrete Mathematics**, v. 29, n. 3, p. 293–309, 1980. ISSN 0012-365X. Disponível em: <<https://www.sciencedirect.com/science/article/pii/0012365X80901582>>.