ABAKÓS
PUC Minas
Instituto de Ciências Exatas e Informática

Identificação do número máximo de caminhos disjuntos em arestas*

Model - Magazine Abakós - ICEI - PUC Minas

Bruno Rodrigues Faria[1]
Guilherme Dantas Caldeira Fagundes[2]
Laura Iara Silva Santos Xavier[3]

Resumo

Há vários problemas a serem tratados com a descoberta do número máximo de caminhos disjuntos em arestas presentes em um grafo. Este trabalho irá tratar de um método que faz a descoberta de quantos e quais são esses caminhos para o usuário com o intuito de auxiliar em futuros desenvolvimentos. Para isso, basta criar um grafo direcionado e selecionar o vértice de 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

[1]Programador, E-mail:bruno.faria@sga.pucminas.br
Graduação Ciências da Computação - PUC Minas, Brasil.
[2]Programador, E-mail:gdcfagundes@sga.pucminas.br
Graduação Ciências da Computação - PUC Minas, Brasil.
[3]Programador, 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 utilizadoa no mundo da computação, por ser bastante versatil na resolução de problemas. Um exemplo é o aplicativo Waze, que utiliza grafos para traçar o caminho a ser percorrido de um ponto a outro. Pode-se observar com isso, que a descoberta de caminhos disjuntos teria uma boa função nesse aplicativo, pois permitiria que o motorista que não queira utilizar o caminho inicial disponibilizado pelo aplicativo, possa solicitar um novo. Assim, utilizando o método proposto neste trabalho, é possível realizar uma busca para descobrir caminhos diferente para o usuário, caso exista algum..

2 DESENVOLVIMENTO

2.1 Entrada de dados

Para a entrada de dados, foi utilizada a linha de comando para passar os argumentos, que são respectivamente: o arquivo contendo o grafo criado 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, a quantidade de vértices, e a lista de arestas nas próximas linhas, da seguinte forma *vértice_saída vértice_entrada*, onde vértice de saída é de onde a aresta sai e o vértice de entrada e 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 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.

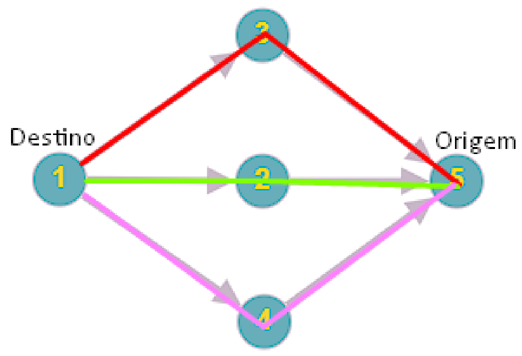

Figura 2 – Caminhos disjuntos em arestas em um grafo

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

$auxGraph \leftarrow copy(graph)$
$parent \leftarrow []$
$maxPaths \leftarrow 0$
$paths \leftarrow NULL$
while $auxGraph.search(origin, destiny, parent)$ **do**
  $reversedPath \leftarrow NULL$
  $v \leftarrow destiny$
  while $v \neq origin$ **do**
    $u \leftarrow parent[v]$
    $reversedPath.append(v)$
    $auxGraph.removeEdge(u, v)$
    $auxGraph.addEdge(v, u)$
    $v \leftarrow parent[u]$
  end while
  $reversedPath.append(origin)$
  $path \leftarrow reversedPath.popAllElements()$
  $paths.append(path)$
  $maxPaths \leftarrow maxFlow + 1$
end while
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.

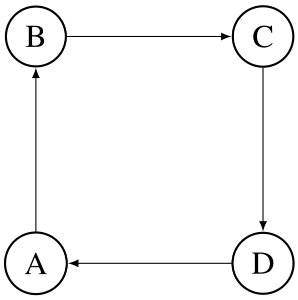

Figura 3 – Grafo cíclico sem arestas antiparalelas

Algorithm 2 Gerar grafo circular

Require: $n > 0$
  $i \leftarrow 0$
  $edges \leftarrow NULL$
  for $i < n$ **do**
    if $i \neq (n-1)$ **then**
      $edges \mathrel{+}= (i, i+1)$
    end if
  end for
  $edges \mathrel{+}= (n-1, 0)$
  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.

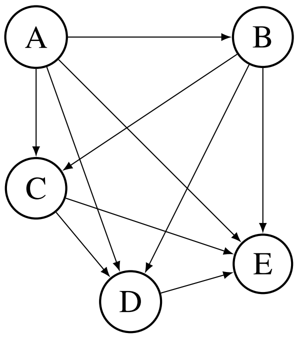

Figura 4 – Grafo formado por 5 vértices seguindo regra específica de formação

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**
        $edges \mathrel{+}= (i, j)$
      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.

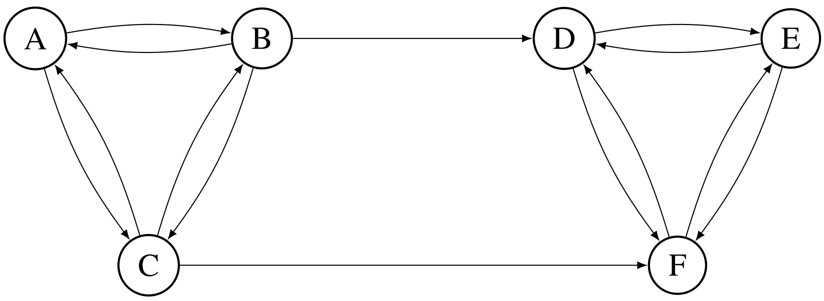

Figura 5 – Dois grafos K3 ligados por duas arestas

Algorithm 4 Gerar dois K-N grafos ligados por duas arestas

Require: $n > 0$

```
edges_k1 ← getCompleteGraph(0, n/2)
edges_k2 ← getCompleteGraph((n/2), n)
edges ← edges_k1 + edgesk2
extraEdge_1 ← (getRandomEdge(0, (n/2) − 1), getRandomEdge((n/2), n − 1))
extraEdge_2 ← (getRandomEdge(0, (n/2) − 1), getRandomEdge((n/2), n − 1))
while extraEdge_1 = extraEdge_2 do
    extraEdge_1 ← (getRandomEdge(0, (n/2) − 1), getRandomEdge((n/2), n − 1))
    extraEdge_2 ← (getRandomEdge(0, (n/2) − 1), getRandomEdge((n/2), n − 1))
end while
edges += extraEdge_1
edges += extraEdge_2
return edges
```

Algorithm 5 Gerar K-N grafo

Require: $(start \in Z) \wedge (end \in Z \wedge start \neq end)$

```
edges ← NULL
for i ← start, end do
    for j ← start, end do
        if i ≠ j ∧ (i, j) ∉ edges ∧ (j, i) ∉ edges then
            edges += (i, j)
            edges += (j, i)
        end if
    end for
end for
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

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

Tempo para execução da topologia 1 (segundos)	
Quantidade de Vértices	*Tempo de execução*
10	0.000s
100	0.001s
500	0.008s
1000	0.010s
10000	0.022s

Tabela 1 – Tempo para execução da topologia 1 (em segundos)

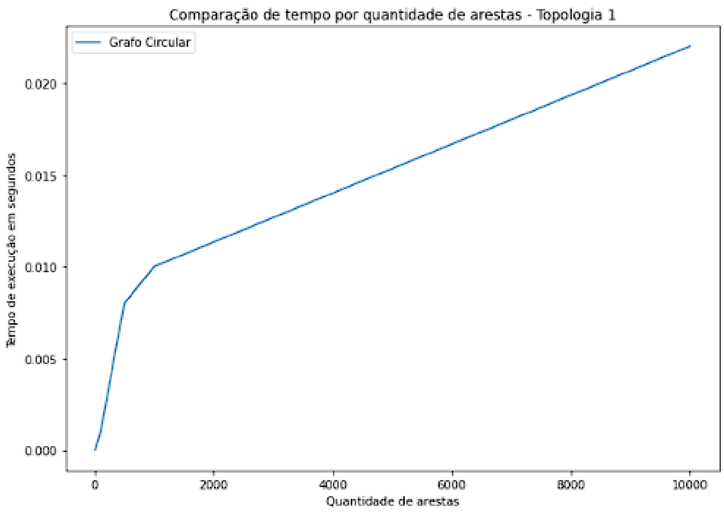

Figura 6 – Analise de tempo x quantidade de arestas - Grafo Circular

Pode-se observar que no grafo circular o tempo de execução no inicio, onde a quantidade de arestas e 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.

3.4.2 Topologia 2 - Grafo Simples

Tempo para execução da topologia 2 (segundos)	
Quantidade de Vértices	*Tempo de execução*
10	0.000s
100	0.001s
500	1.930s
1000	14.130s
10000	98.913s

Tabela 2 – Tempo para execução da topologia 2 (em segundos)

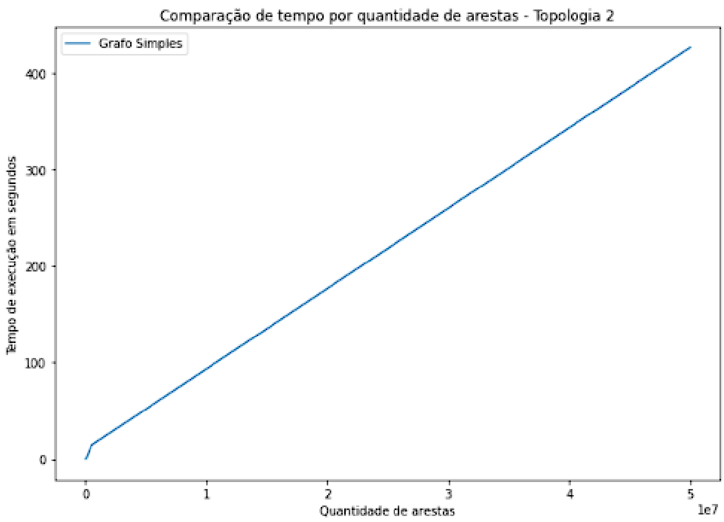

Figura 7 – Analise de tempo x quantidade de arestas - Grafo Simples

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.

3.4.3 Topologia 3 - Grafo Completo

Tempo para execução da topologia 3 (segundos)	
Quantidade de Vértices	*Tempo de execução*
10	0.000s
100	0.002s
500	0.014s
1000	0.210s
10000	X

Tabela 3 – Tempo para execução da topologia 3 (segundos)

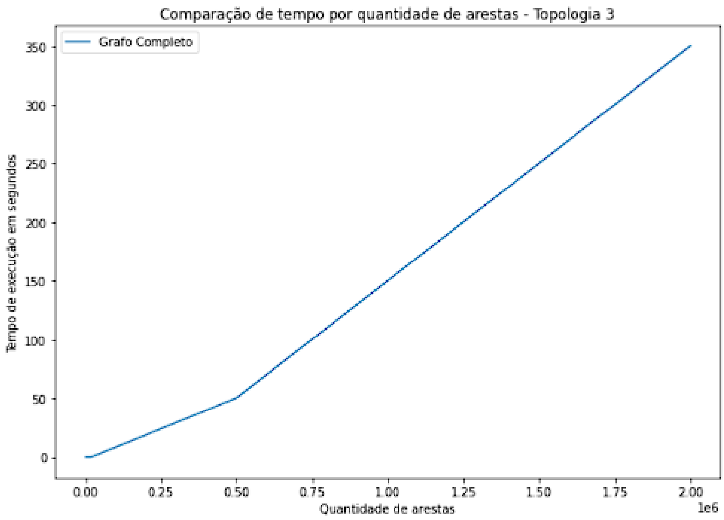

Figura 8 – Analise de tempo x quantidade de arestas - Grafo Completo

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.

3.4.4 Analise tempo por quantidade de vértices

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

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

Tabela 4 – Tempo para execução por topologia e quantidade de vértices (em segundos)

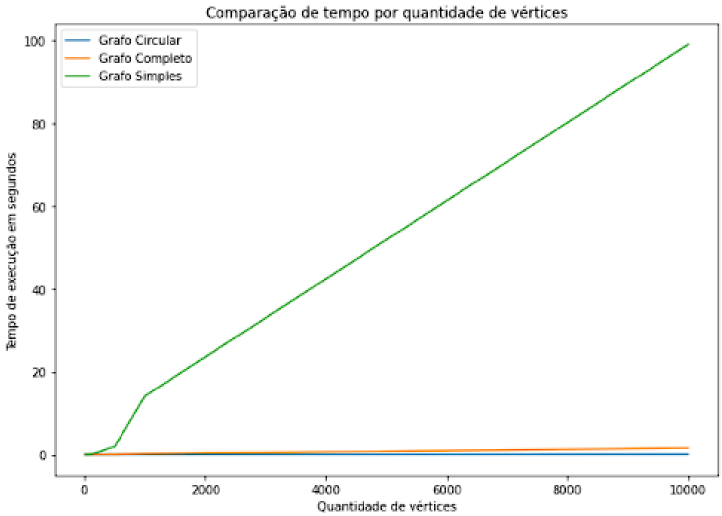

Figura 9 – Analise de tempo x quantidade de vértices

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 obtem 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 consiferavelmente como mostrado nas analises 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>.