TEG

Gilmário B. Santos

gilmario.santos@udesc.br

http://www.joinville.udesc.br/portal/pagina/gilmario

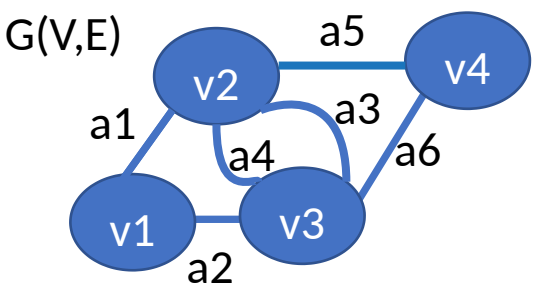
Percursos

Passeio *P* em um grafo G(V,E):

Uma sequência de vértices $v_1, \ldots, v_k$ tal que $(v_j, v_{j+1}) \in E$, $1 \leq j < |k|$, é denominado passeio de $v_1, \ldots, v_k$.

O comprimento de um passeio corresponde ao seu número de arestas.

Passeio aberto de *n* vértices → comprimento = n-1.



Exemplos de passeios em G:

P1: v1, a2, v3, a4, v2, a5, v4, a6, v3

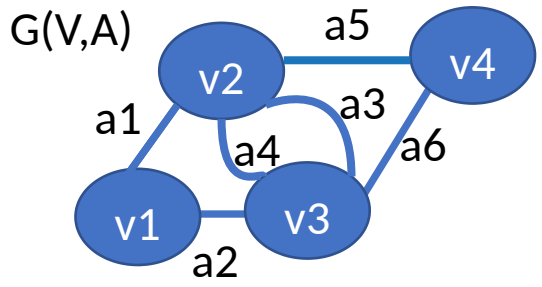
P2: v1, a1, v2, a3, v3, a4, v2, a1, v1

P3: v3, a6, v4, a5, v2, a1, v1

Percursos

Trilha *T* em um grafo:

Corresponde a um passeio sem a repetição de arestas.



Exemplos:

P1: v1, a2, v3, a4, v2, a5, v4, a6, v3, a3 v2

P2: v1, a1, v2, a3, v3, a4, v2, a1, v1

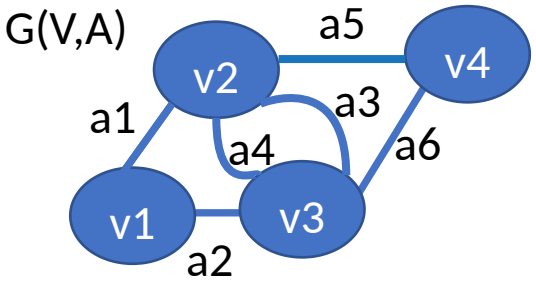
P2 não é uma trilha!

Percursos

Caminho *C* em um grafo:

Corresponde a um passeio sem a repetição de vértices.

Um caminho v_1, v_2, v_3, ..., v_k, v_{k+1} no qual v_1, v_2, v_3, ..., v_k são todos distintos entre si é chamada caminho simples;



Exemplos:

P1: v1, a2, v3, a4, v2, a5, v4, a6, v3

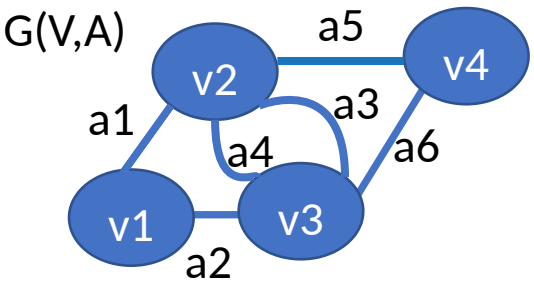
P2: v3, a6, v4, a5, v2, a1, v1

P1 não é um caminho

Percursos

Se $v_1=v_{k+1}$ e $k \geq 3$, então o caminho $v_1, v_2, v_3, \ldots, v_k, v_{k+1}$ é fechado e chamado de ciclo (circuito)

Ciclo→ não há repetições de vértices (exceto nas extremidades do caminho).



P1: v3, a6, v4, a5, v2, a1, v1, a2, v3

P4: v1, a2, v3, a6, v4, a5, v2, a1, v1

P3 e P4 são ciclos

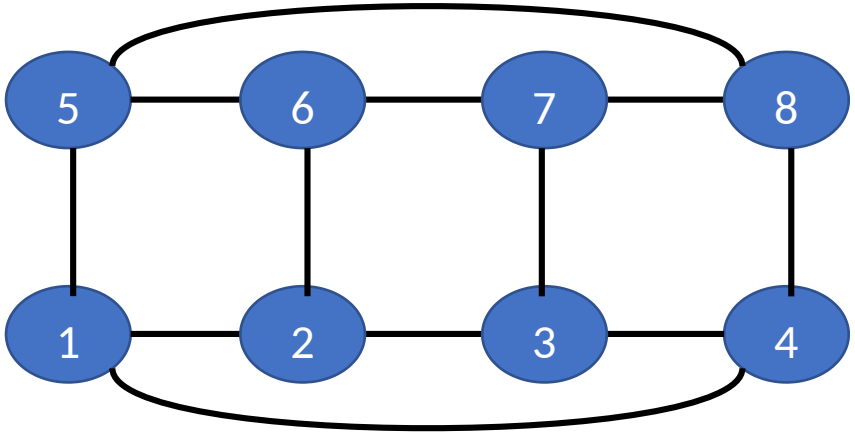
Percursos

Um grafo que não possui ciclos é acíclico.

Um triângulo é um ciclo de comprimento 3.

Dois ciclos são considerados idênticos se um deles puder ser obtido do outro, através de uma permutação circular de seus vértices.

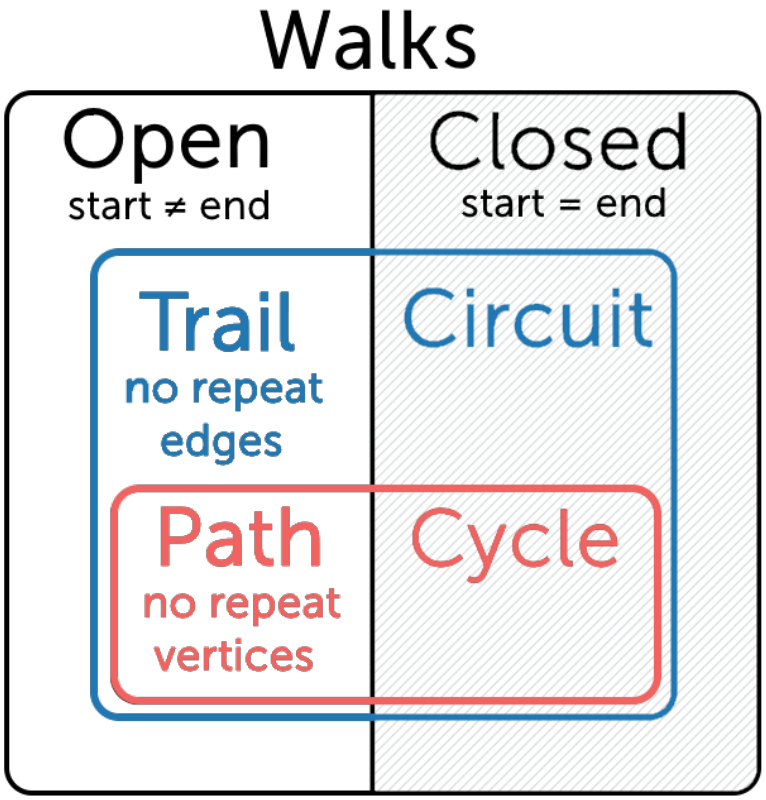
Por exemplo, no grafo da figura, os percursos *2*, *3*, *7*, *6*, *2* e *7*, *6*, *2*, *3*, *7* são ciclos idênticos,



Percursos - Resumo

1) Passeio: sequência de vértices e arestas;
2) Trilha: passeio sem repetição de arestas;
3) Caminho: passeio sem repetição de vértices;
4) Ciclo: caminho fechado;

Passeio: walk
Trilha: Trial
Caminho: Path



https://mathspace.co/textbooks/syllabuses/Syllabus-1030/topics/Topic-20297/subtopics/Subtopic-266706/
https://www.javatpoint.com/walks-trails-path-circuit-and-cycle-in-discrete-mathematics

Percursos - Resumo

1)Passeio: sequência de vértices e arestas;

2)Trilha: passeio sem repetição de arestas;

3)Caminho: passeio sem repetição de vértices, todos os vértices são distintos entre si;

Perceba que a parte “distinta” da definição de caminho é o que proíbe arestas repetidas, pois se uma aresta é repetida, um vértice é necessariamente repetido. Na verdade, podemos definir um caminho como uma trilha com a propriedade adicional de que os vértices são distintos e então restringir explicitamente arestas repetidas.

4)Ciclo: caminho fechado.

Percursos

Dois percursos particularmente especiais:

Trilha fechada Euleriana: contém todas as arestas do grafo sem repeti-las (é permitido repetir vértices);

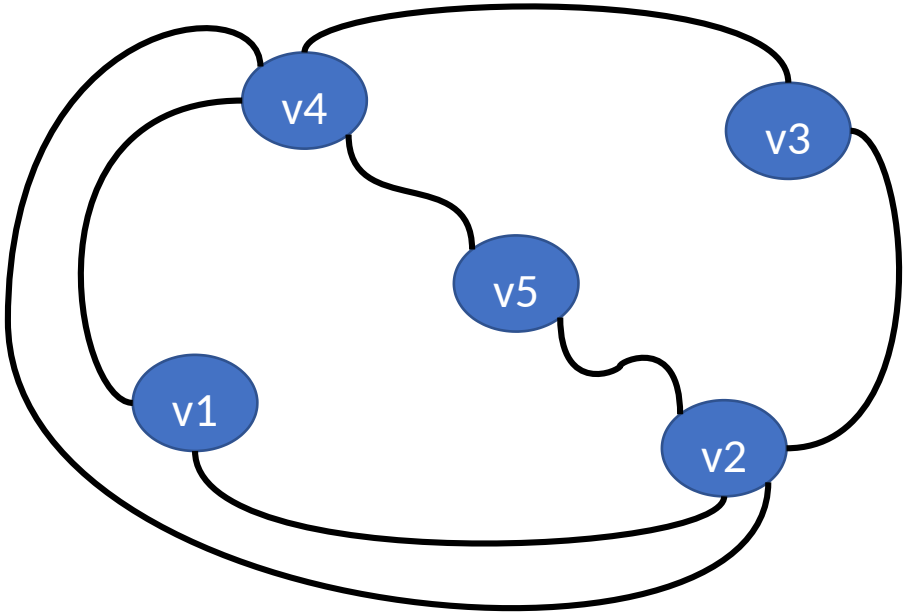
Ciclo Hamiltoniano: contém todos os vértices do grafo sem repetições, ou seja, todos os vértices são distintos entre si.

Percursos

Seja G um grafo que possui uma trilha contendo todas as arestas de G sem repetições, então G possui uma trilha Euleriana.

Seja G um grafo com uma trilha fechada contendo todas as arestas de G sem repetições, então G possui uma trilha fechada Euleriana e é dito um grafo Euleriano.

{v4, v5, v2, v3, v4, v1, v2, v4} é uma trilha fechada euleriana, G é euleriano. Nesse caso pode ocorrer repetições de vértices.



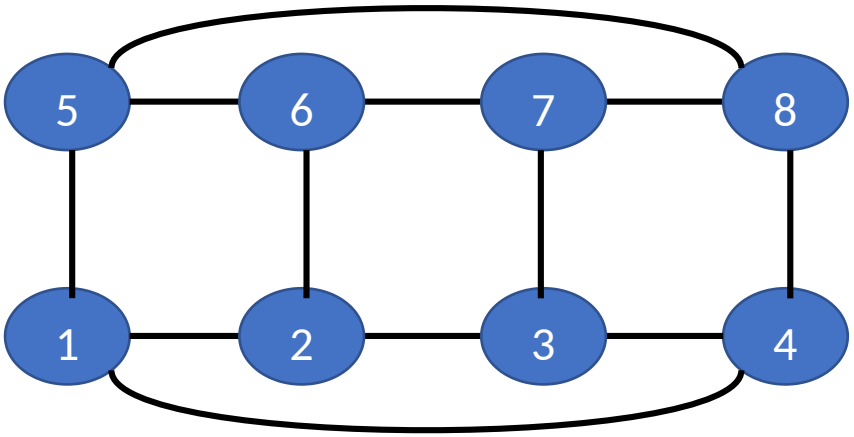
Percursos

Um percurso que contenha todos os vértices do grafo sem repetições é chamado caminho Hamiltoniano. O caminho *1*, *5*, *6*, *2*, *3*, *7*, *8*, *4* é Hamiltoniano.

Um caminho Hamiltoniano fechado é um ciclo Hamiltoniano.

Se G é um grafo que possui ciclo Hamiltoniano, então G é denominado um grafo Hamiltoniano.

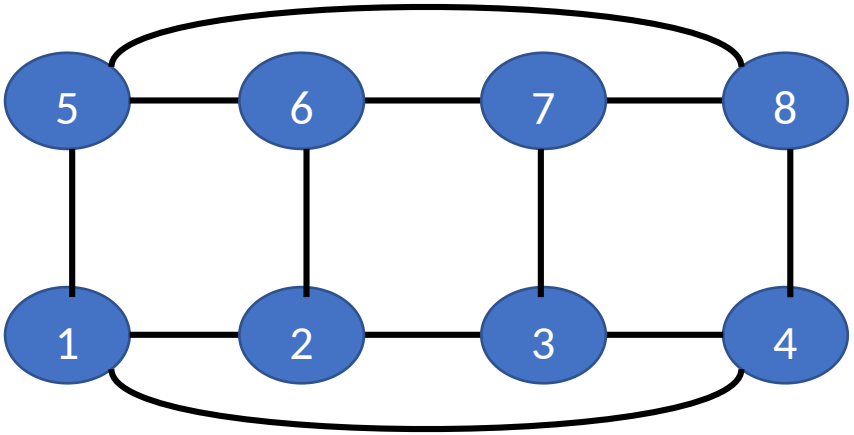
O percurso *1*, *5*, *6*, *2*, *3*, *7*, *8*, *4*, *1* é um ciclo Hamiltoniano, G é Hamiltoniano.



Distância

Denomina-se distância d(v,w) entre dois vértices v,w de um grafo ao comprimento do menor caminho entre v e w (menor contagem de arestas entre v e w).

A distância entre os vértices 1 e 8, no grafo da figura é igual a 2, isto é, d(1,8) = 2.

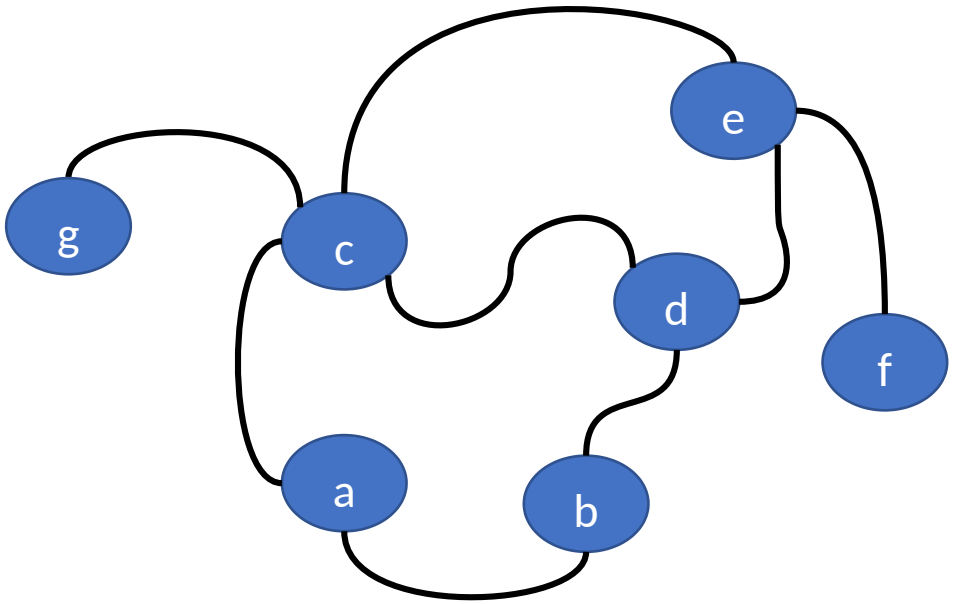


Distância

Seja G(V,E), a excentricidade de um vértice *v* é igual à máxima distância entre v e w, para todo v ∈ V e w ∈ V.

O centro de G é o subconjunto dos vértices de excentricidade mínima.

A tabela abaixo apresenta as distâncias entre *a* e os demais vértices do grafo



	distâncias de *a* (comprimento do menor caminho)
b	1
c	1
d	2
e	2
f	3
g	2

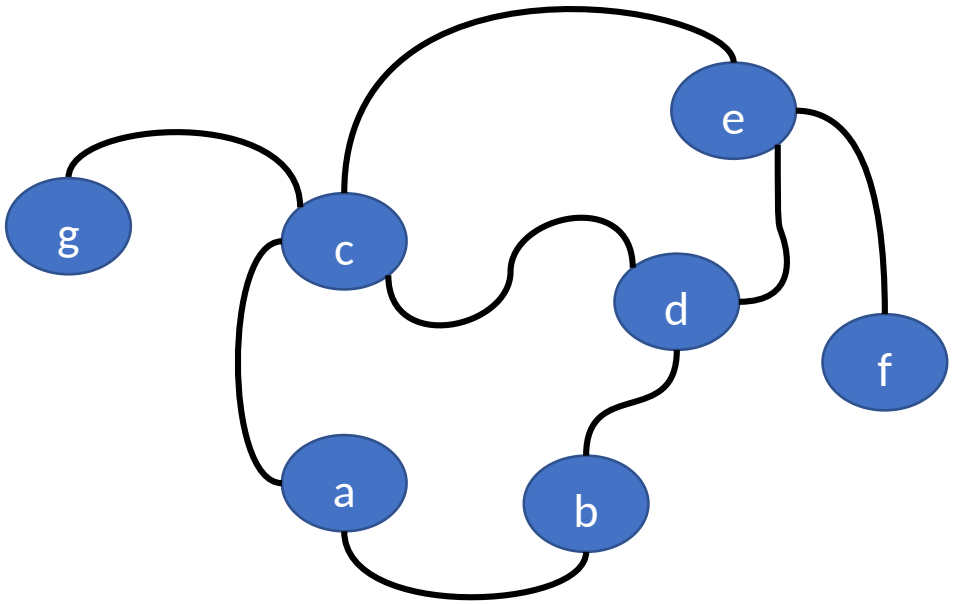
Excentricidade de *a*

Distância

Seja G(V,E), a excentricidade de um vértice *v* é igual à máxima distância entre v e w, para todo v ∈ V e w ∈ V.

O centro de G é o subconjunto dos vértices de excentricidade mínima.

A tabela abaixo apresenta as excentricidades dos vértices do grafo cujo centro corresponde ao subconjunto dos vértices {c,d,e}.



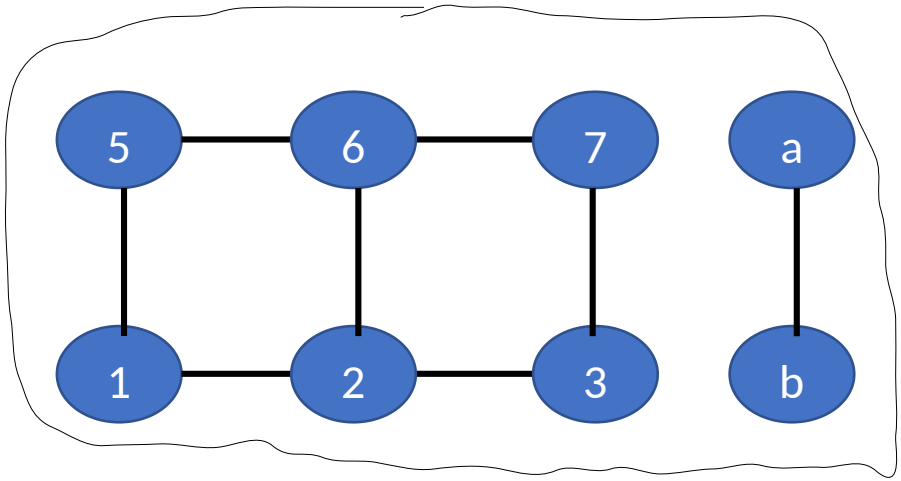
	excentricidades
a	3
b	3
c	2
d	2
e	2
f	3
g	3

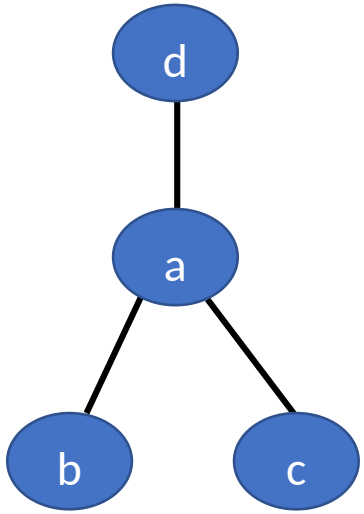
Componente Conexo

Um grafo G(V,E) é denominado conexo quando existe caminho entre cada par de vértices de G. Caso contrário G é desconexo.

A representação geométrica de um grafo desconexo é necessariamente não contígua.

Um grafo G sem arestas é totalmente desconexo.





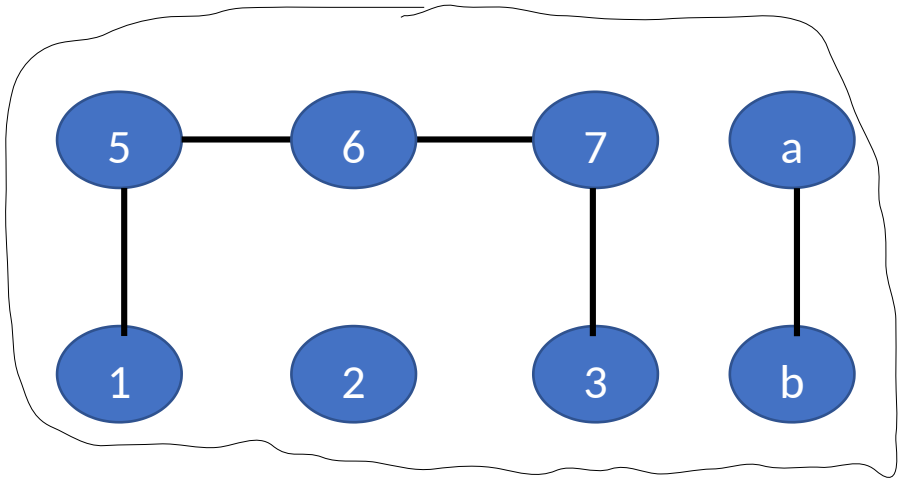
Componente Conexo

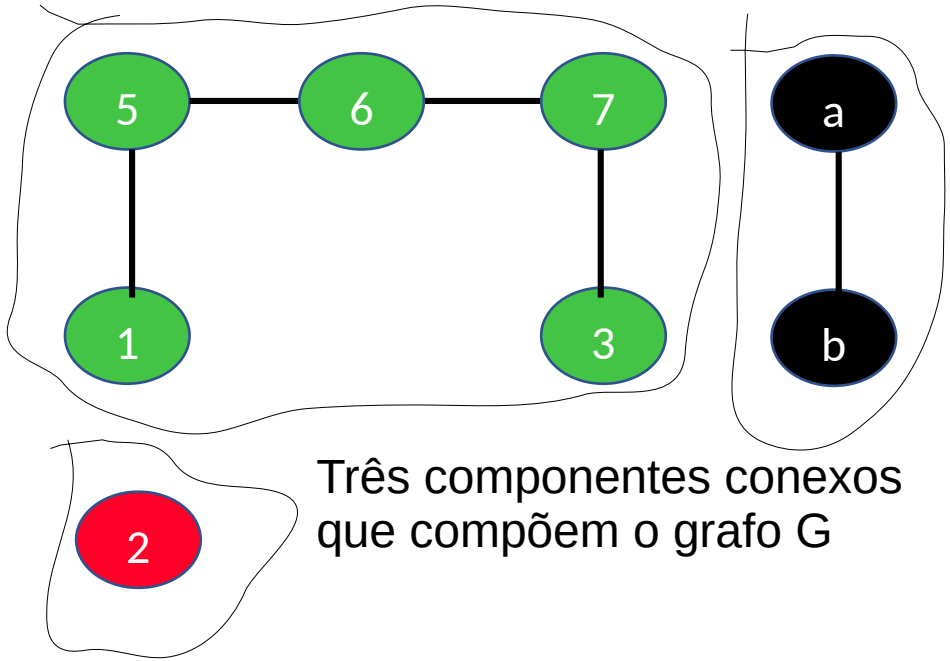
Um grafo G(V,E) é denominado conexo quando existe caminho entre cada par de vértices de G. Caso contrário G é desconexo.

Um componente conexo de um grafo é um subgrafo maximal.

Um componente conexo é o maior subgrafo possível que mantém a conectividade entre seus vértices: caso você acrescente um novo nó do grafo ao subgrafo, este subgrafo deixará de ser um componente conexo.

Grafo G

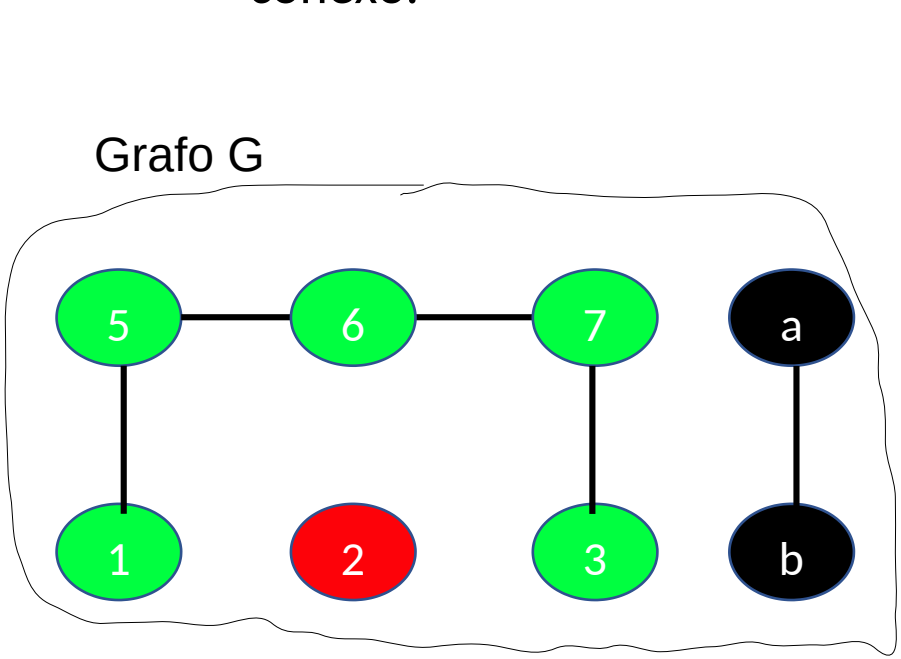


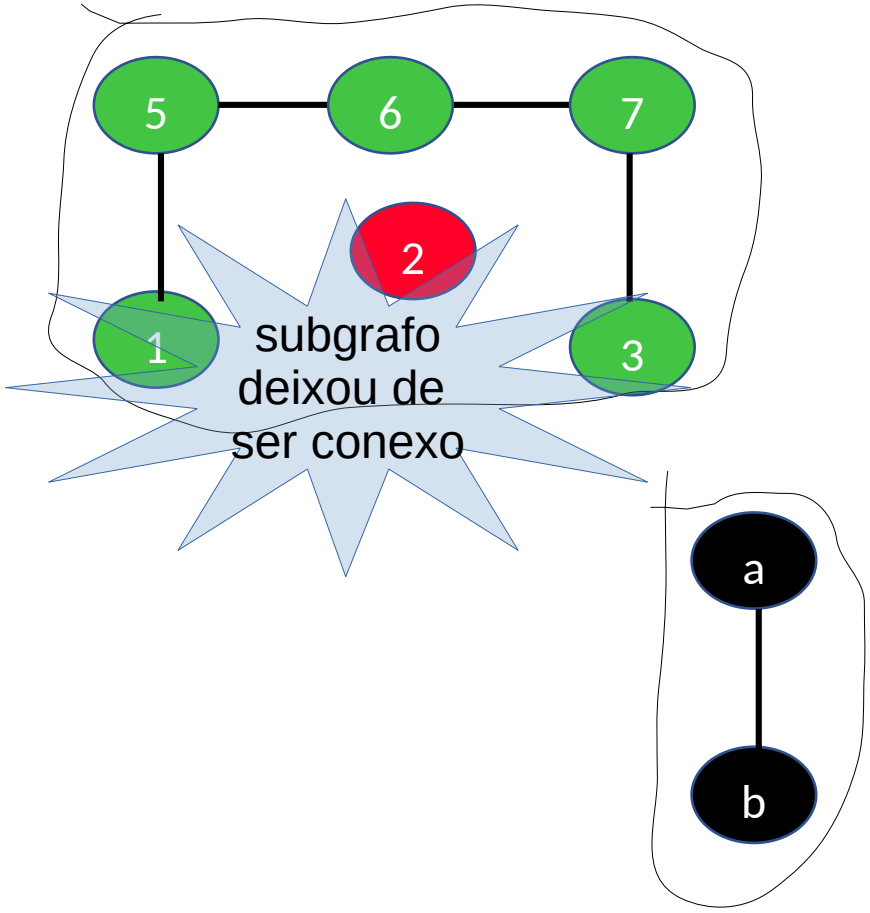


Três componentes conexos que compõem o grafo G

Componente Conexo

Um componente conexo é o maior subgrafo possível que mantém a conectividade entre seus vértices: caso você acrescente um novo nó do grafo ao subgrafo, este subgrafo deixará de ser um componente conexo.





Caminho mínimo

Em muitas situações é útil determinar o caminho mais curto entre duas localizações, quando esse problema trata de arestas de mesmo peso ou sem ponderação, esse problema se resume a buscar um caminho que percorra a menor quantidade de arestas entre a origem e o destino.

Quando as arestas do grafo são ponderadas, a determinação do caminho mínimo precisa levar em conta o "custo" desses pesos na determinação do percurso desejado.

Uma solução conhecida para esse problema é implementada pelo algoritmo de Dijkstra o qual trata um grafo cujas arestas ponderadas (pesos positivos) e considera o cálculo do caminho mínimo entre dois vértices pré-determinados.

É importante destacar que Dijkstra não é a única opção para a determinação do caminho mínimo, um outro algoritmo muito citado na literatura e que lida com o mesmo tipo de problema é o algoritmo de Bellman-Ford.

Com uma boa implementação, o tempo de execução do algoritmo de Dijkstra é inferior ao do algoritmo de Bellman-Ford (Cormen).

Caminho mínimo

Vamos agora ao algoritmo de Dijkstra apresentado na seção CAMINHO MÍNIMO E ÁRVORE GERADORA MÍNIMA do livro da Judith Gersting (disponível no formato ebook e fisicamente no acervo da biblioteca do CCT).

Fundamentos Matemáticos para a Ciência da Computação

Judith L. Gersting

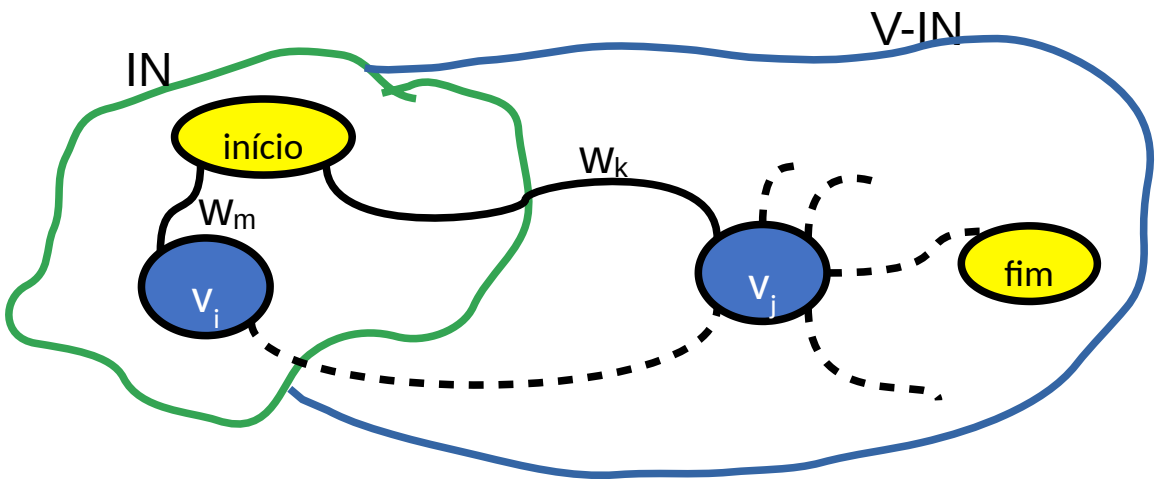
Demonstração (fonte Wikipedia)

Dijkstra

Seja G(V,E) um grafo ponderado, Dijkstra mantém um conjunto IN de vértices já processados (vértices para os quais já foram determinados os pesos finais de caminhos mínimos que partem do vértice de saída).

O algoritmo seleciona repetidamente o vértice ainda não processado pertencente a {V – IN} que tem a mínima estimativa do caminho mínimo.

A ideia é avaliar o custo para alcançar esse vértice via algum caminho mínimo atual ou por outro caminho mais promissor (em termos do custo total). Esse processo é chamado de relaxação.



Dijkstra

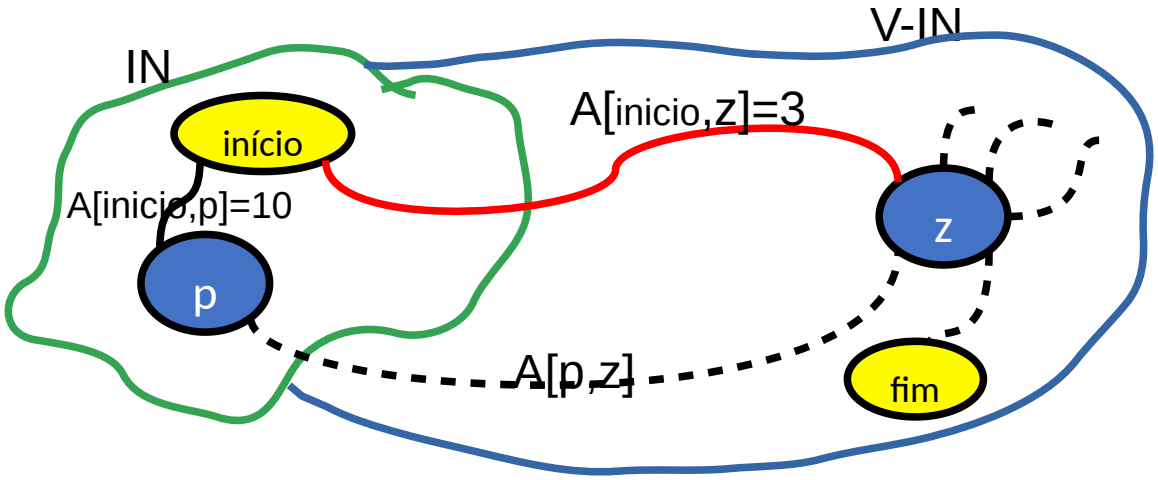
Seja a Matriz de adjacências *A* de G(V,E) e um caminho mínimo que chega em *p*, deseja-se avaliar o vértice *z* quanto ao próxima configuração do caminho ser {início, p, z} ou {início, z}, para tanto se avalia a relaxação:

Relaxação: d[z] = min(d[z],d[p]+A[p,z])

	início	p	z
d	0	10	3
s	--	início	início

?

	início	p	z
d	0	10	3
s	--	início	início

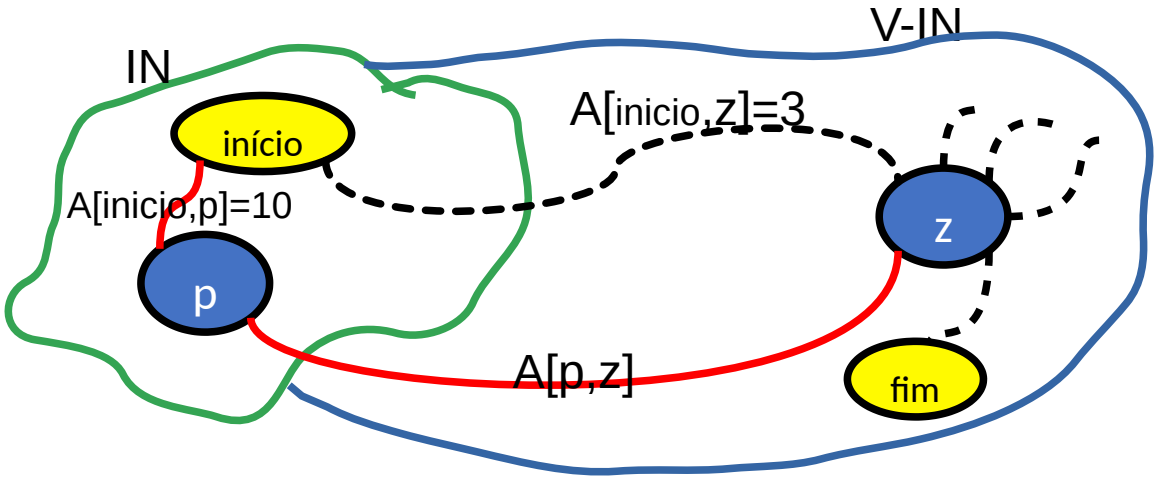


Dijkstra

Seja a Matriz de adjacências *A* de G(V,E) e um caminho mínimo que chega em *p*, deseja-se avaliar o vértice *z* quanto ao próximo trecho do caminho ser {início, p, z} ou {início, z}, para tanto se avalia a relaxação:

Relaxação: d[z] = min(d[z],d[p]+A[p,z])

	início	p	z
d	0	10	3
s	--	início	início



?

?

	início	p	z
d	0	10	3
s	--	início	início

	início	p	z
d	0	10	d[p]+A[p,z]
s	--	início	p

Dijkstra

ALGORITMO Caminho Mínimo do vértice **x** ao **y** por Dijkstra

Inicializações:

A: matriz de adjacências;

IN: conjunto de vértices; {menor caminho conhecido a partir de x}

d: vetor de inteiros; {distância a partir de x usando os vértices de IN}

s: vetor de vértices; {vértice anterior no caminho mínimo}

```
begin
(inicia o conjunto IN e os vetores d e s}
IN:= {x};
d[x] : = 0;
for todos os vértices z que não pertençam a IN do
      begin
        d[z] := A[x, z];
        s[z] := x;
end;
```

Dijkstra

Complemento do algoritmo de Dijkstra

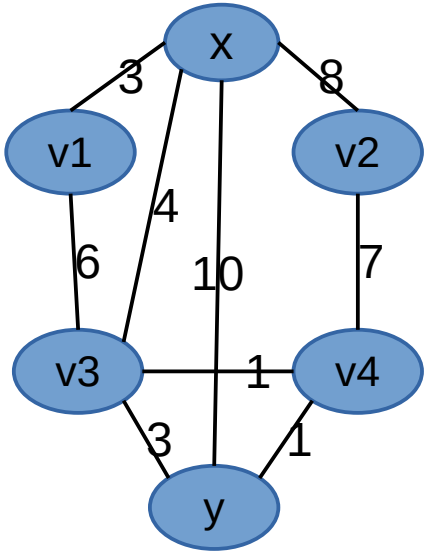
```
enquanto y não pertence a IN faça
    p = nó z não pertencente a IN com d[z] mínimo
    IN = IN ⋃ {p}
    para todos os nós não pertencentes a IN faça
        DistânciaAnterior = d[z]
        d[z] = min(d[z], d[p] + A[p, z])
        se d[z] < DistânciaAnterior então
            s[z] = p
escreva (“Em ordem inversa, os nós do caminho são”)
escreva (y)
z = y
repita
    escreva (s[z])
    z = s[z]
até z = x
escreva (“A distância percorrida é”, d[y])
```

Simulação do algoritmo de Dijkstra (*CaminhoMinimo em* Judith Gersting)

Inicialização*:*

x é o vértice de partida *(s=x)*

Ao final da fase de inicialização teremos IN={x} e *d* contendo todas as distâncias de *x* até os demais vértices em {*V-IN}*.



Matriz de adjacências (A)						
	x	v1	v2	v3	v4	y
x	∞	3	8	4	∞	10
v1	3	∞	∞	6	∞	∞
v2	8	∞	∞	∞	7	∞
v3	4	6	∞	∞	1	3
v4	∞	∞	7	1	∞	1
y	10	∞	∞	3	1	∞

IN | V-IN

	x	v1	v2	v3	v4	y
d	0	3	8	4	∞	10
s	–	x	x	x	x	x

Simulação do algoritmo de Dijkstra (*CaminhoMinimo em* Judith Gersting)

Laço principal (1ª iteração):

ENQUANTO o destino *y* não pertencer a *IN*:

// 1) determine *p*, o vértice com menor valor d[ ] que pertença a V-IN
No caso: p=v1

IN | V-IN

	x	v1	v2	v3	v4	y
d	0	3	8	4	∞	10

// 2) atualize *IN = IN U p*
No caso: IN={x,v1} → V-IN={v2,v3,v4,y}

IN | V-IN

	x	v1	v2	v3	v4	y
d	0	3	8	4	∞	10

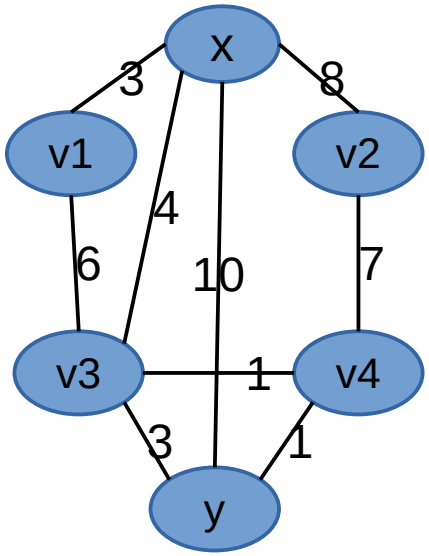
// 3) verifique se há um caminho mínimo chegando em *z em V-IN*
PARA todo *z* em *V-IN*

DistânciaAnterior = d[z]
calcule a relaxação: d[z] = min(d[z],d[p]+A[p,z])
SE d[z] < DistânciaAnterior:
ENTÃO: s[z]=p

Simulação do algoritmo de Dijkstra (*CaminhoMinimo em* Judith Gersting)

Laço principal (1ª iteração):

ENQUANTO o destino *y* não pertencer a *IN*:
  No caso: p=v1
    PARA todo *z* em V-IN={v2,v3,v4,y}, d[z] = min(d[z],d[p]+A[p,z])
      z=v2: d[v2] = min{d[v2],d[v1]+A[v1,v2]}= min{8,3+∞}=8
      z=v3: d[v3] = min{d[v3],d[v1]+A[v1,v3]}= min{4,3+6}=4
      z=v4: d[v4] = min{d[v4],d[v1]+A[v1,v4]}= min{∞,3+∞}=∞
      z=y: d[y] = min{d[y],d[v1]+A[v1,y]}= min{10,3+∞}=10



Matriz de adjacências (A)						
	x	v1	v2	v3	v4	y
x	∞	3	8	4	∞	10
v1	3	∞	∞	6	∞	∞
v2	8	∞	∞	∞	7	∞
v3	4	6	∞	∞	1	3
v4	∞	∞	7	1	∞	1
y	10	∞	∞	3	1	∞

IN V-IN

	x	v1	v2	v3	v4	y
d	0	3	8	4	∞	10
s	–	x	x	x	x	x

Simulação do algoritmo de Dijkstra (*CaminhoMinimo em* Judith Gersting)

Laço principal (1ª iteração):

No caso: p=v1 e *z* em V-IN={v2,v3,v4,y}, d[z] = min(d[z],d[p]+A[p,z])

d[v2] = 8

→ manteve a mesma *DistânciaAnterior* → não altere a tabela (d,s)

d[v3] = 4

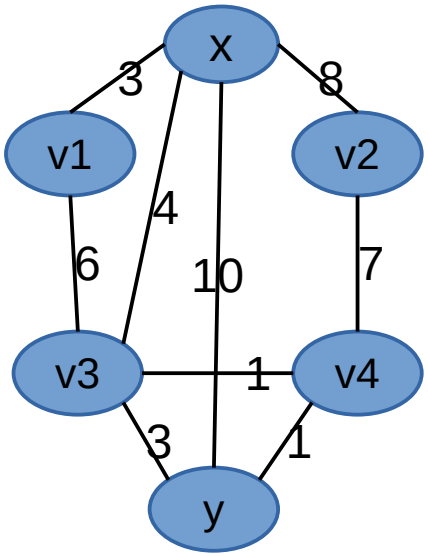
→ manteve a mesma *DistânciaAnterior* → não altere a tabela (d,s)

d[v4] = ∞

→ manteve a mesma *DistânciaAnterior* → não altere a tabela (d,s)

d[y] = 10

→ manteve a mesma *DistânciaAnterior* → não altere a tabela (d,s)



Os valores na tabela se mantêm inalterados:

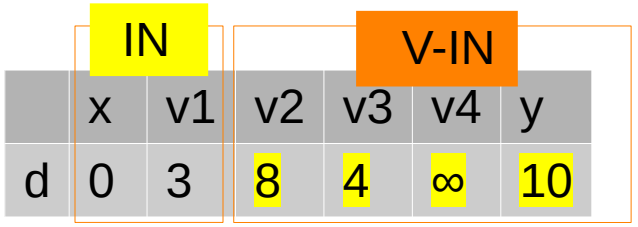
	IN		V-IN			
	x	v1	v2	v3	v4	y
d	0	3	8	4	∞	10
s	–	x	x	x	x	x

Simulação do algoritmo de Dijkstra (*CaminhoMinimo em* Judith Gersting)

Laço principal (2ª iteração):

// 1) determine *p*, o vértice com menor valor d[ ] que pertença a V-IN

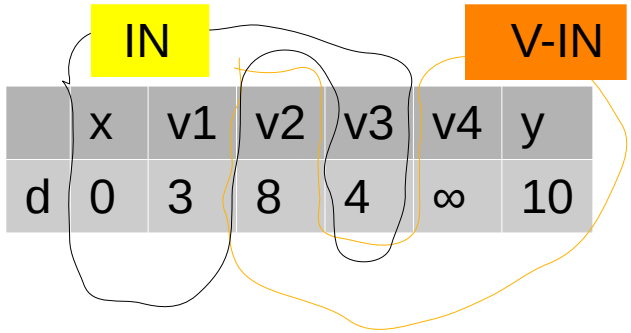
No caso: p=v3



	x	v1	v2	v3	v4	y
d	0	3	8	4	∞	10

// 2) atualize *IN = IN U p*

No caso: IN={x,v1,v3} → V-IN={v2,v4,y}



	x	v1	v2	v3	v4	y
d	0	3	8	4	∞	10

// 3) verifique se há um caminho mínimo chegando em *z em V-IN*

PARA todo *z* em *V-IN*

DistânciaAnterior = d[z]

calcule a relaxação: d[z] = min(d[z],d[p]+A[p,z])

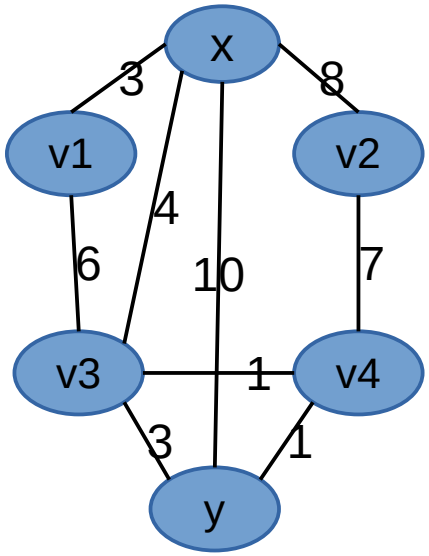
SE d[z] < DistânciaAnterior:

ENTÃO: s[z]=p

Simulação do algoritmo de Dijkstra (*CaminhoMinimo em* Judith Gersting)

Laço principal (2ª iteração):

ENQUANTO o destino *y* não pertencer a *IN*:
  No caso: p=v3
    PARA todo *z* em V-IN={v2,v4,y}, d[z] = min(d[z],d[p]+A[p,z])
      z=v2: d[v2] = min{d[v2],d[v3]+A[v3,v2]}= min{8,3+∞}=8
      z=v4: d[v4] = min{d[v4],d[v3]+A[v3,v4]}= min{∞,4+1}=5
      z=y: d[y] = min{d[y],d[v3]+A[v3,y]}= min{10,4+3}=7



Matriz de adjacências (A)						
	x	v1	v2	v3	v4	y
x	∞	3	8	4	∞	10
v1	3	∞	∞	6	∞	∞
v2	8	∞	∞	∞	7	∞
v3	4	6	∞	∞	1	3
v4	∞	∞	7	1	∞	1
y	10	∞	∞	3	1	∞

IN

	x	v1	v2	v3	v4	y
d	0	3	8	4	∞	10
s	–	x	x	x	x	x

V-IN

Simulação do algoritmo de Dijkstra (*CaminhoMinimo em* Judith Gersting)

Laço principal (2ª iteração):

No caso: p=v3 e *z* em V-IN={v2,v4,y}, d[z] = min(d[z],d[p]+A[p,z])

d[v2] = 8

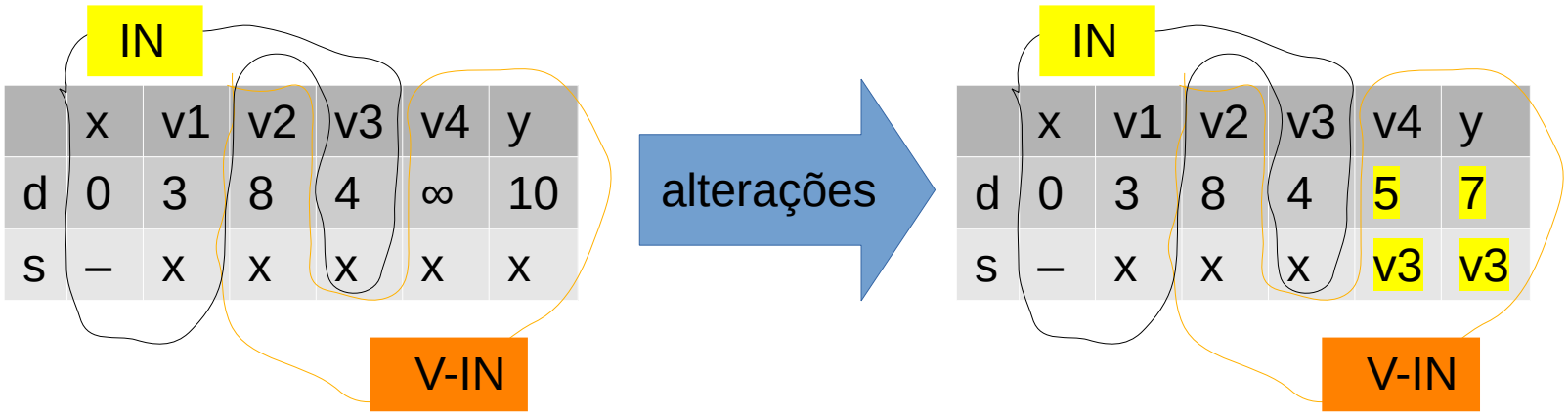
→ manteve a mesma *DistânciaAnterior* → não altere a tabela (d,s)

d[v4] = 5

→ menor do que *DistânciaAnterior* = ∞ → altera a tabela (d,s)

d[y] = 7

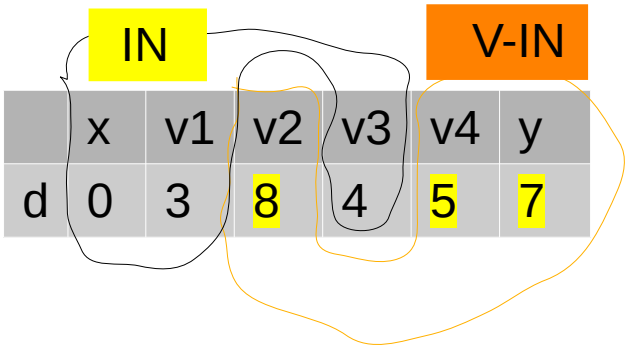
→ menor do que a *DistânciaAnterior* = *10* → altere a tabela (d,s)



Simulação do algoritmo de Dijkstra (*CaminhoMinimo em* Judith Gersting)

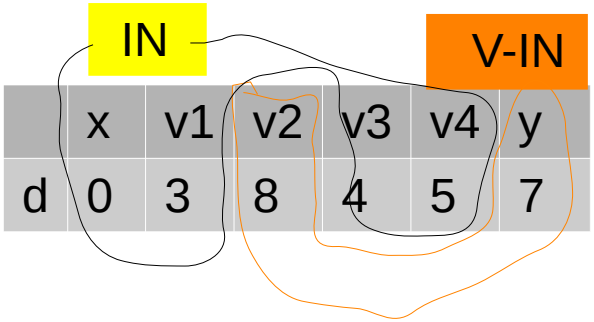
Laço principal (3ª iteração):

// 1) determine *p*, o vértice com menor valor d[ ] que pertença a V-IN
No caso: p=v4



	x	v1	v2	v3	v4	y
d	0	3	8	4	5	7

// 2) atualize *IN = IN U p*
No caso: IN={x,v1,v3,v4} → V-IN={v2,y}



	x	v1	v2	v3	v4	y
d	0	3	8	4	5	7

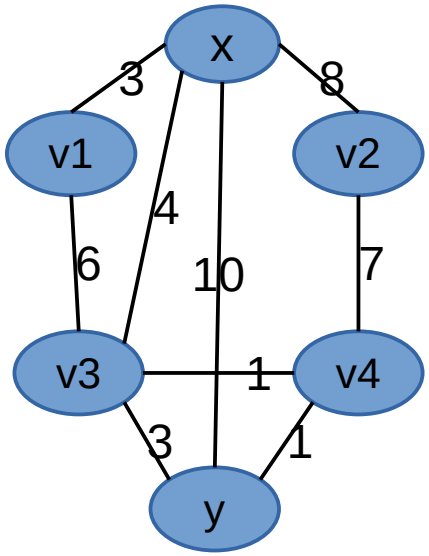
// 3) verifique se há um caminho mínimo chegando em *z em V-IN*
PARA todo *z* em *V-IN*

DistânciaAnterior = d[z]
calcule a relaxação: d[z] = min(d[z],d[p]+A[p,z])
SE d[z] < DistânciaAnterior:
ENTÃO: s[z]=p

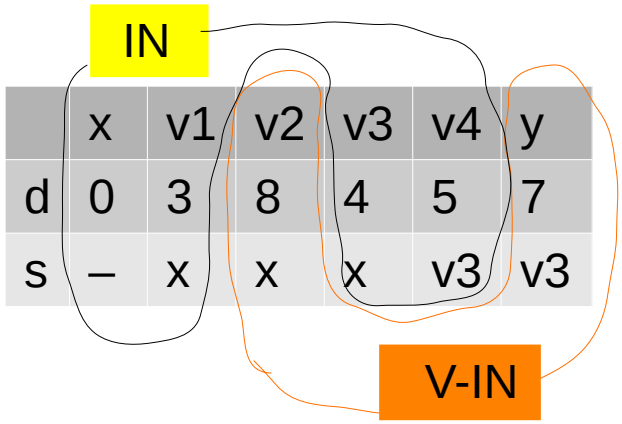
Simulação do algoritmo de Dijkstra (*CaminhoMinimo em* Judith Gersting)

Laço principal (3ª iteração):

ENQUANTO o destino *y* não pertencer a *IN*:
  No caso: p=v4
    PARA todo *z* em V-IN={v4,y}, d[z] = min(d[z],d[p]+A[p,z])
      z=v2: d[v2] = min{d[v2],d[v4]+A[v4,v2]}= min{8,5+7}=8
      z=y: d[y] = min{d[y],d[v4]+A[v4,y]}= min{10,5+1}=6



Matriz de adjacências (A)						
	x	v1	v2	v3	v4	y
x	∞	3	8	4	∞	10
v1	3	∞	∞	6	∞	∞
v2	8	∞	∞	∞	7	∞
v3	4	6	∞	∞	1	3
v4	∞	∞	7	1	∞	1
y	10	∞	∞	3	1	∞



	x	v1	v2	v3	v4	y
d	0	3	8	4	5	7
s	–	x	x	x	v3	v3

Simulação do algoritmo de Dijkstra (*CaminhoMinimo em* Judith Gersting)

Laço principal (3ª iteração):

No caso: p=v4 e *z* em V-IN={v2,y}, d[z] = min(d[z],d[p]+A[p,z])

d[v2] = 8

→ manteve a *DistânciaAnterior* → não altera a tabela (d,s)

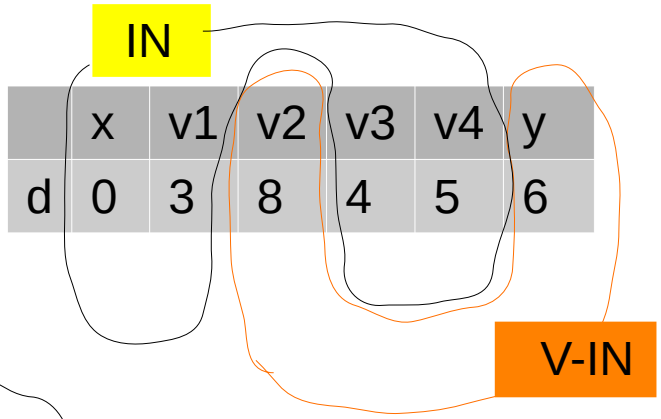
d[y] = 6

→ menor do que *DistânciaAnterior* = *7* → altere a tabela (d,s)

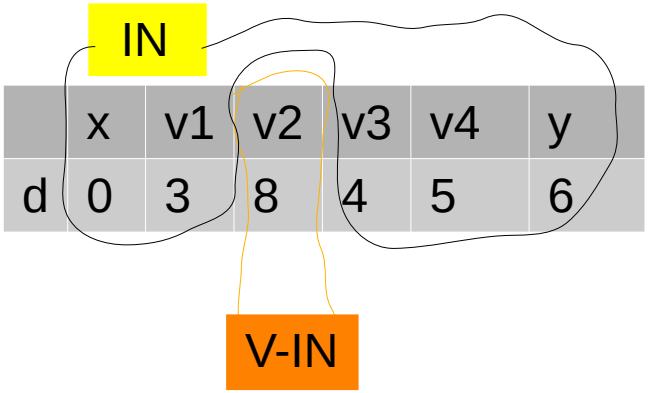
Simulação do algoritmo de Dijkstra (*CaminhoMinimo em* Judith Gersting)

Laço principal (4ª iteração):

// 1) determine *p*, o vértice com menor valor d[ ] que pertença a V-IN
No caso: p=y



// 2) atualize *IN = IN U p*
No caso: IN={x,v1,v4,v3,y} → V-IN={v2}



// 3) verifique se há um caminho mínimo chegando em *z em V-IN*
PARA todo *z* em *V-IN*

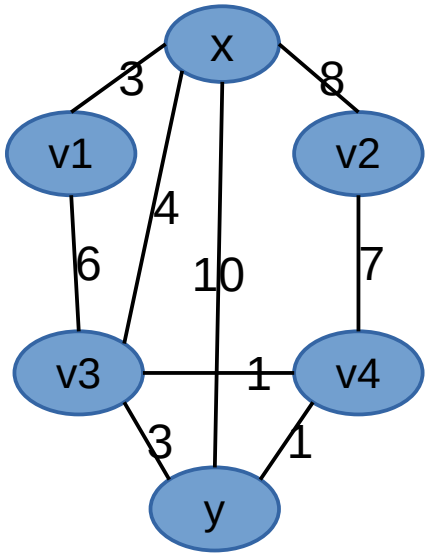
DistânciaAnterior = d[z]
calcule a relaxação: d[z] = min(d[z],d[p]+A[p,z])
SE d[z] < DistânciaAnterior:
ENTÃO: s[z]=p

Simulação do algoritmo de Dijkstra (*CaminhoMinimo em* Judith Gersting)

Laço principal (5ª iteração):

Não ocorre pois y ∈ IN

ENQUANTO o destino *y* não pertencer a *IN*:
é FALSA a condição que garantiria a iteração



IN

	x	v1	v2	v3	v4	y
d	0	3	8	4	5	6
s	–	x	x	x	v3	v4

V-IN

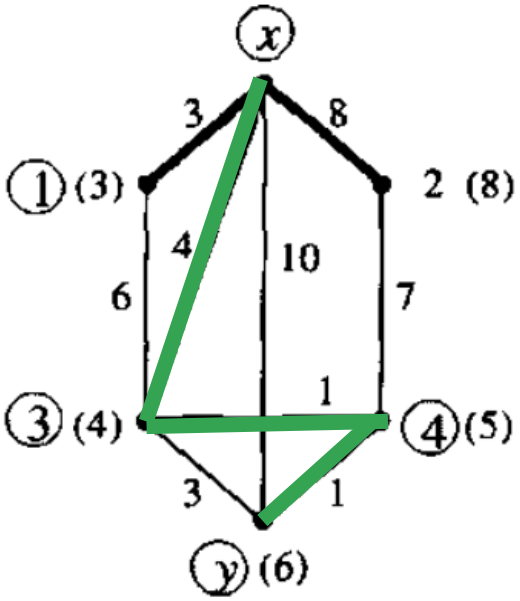
Agora que y pertence a IN, o laço while termina.

O caminho passa por y, s[y] = 4, s[4] = 3 e .v[3] = x.

Assim, o caminho usa os vértices x, 3, 4 e y. (O algoritmo nos fornece estes vértices na ordem inversa.)

A distância do caminho é d[y] = 6

	x	1	2	3	4	y
d	0	3	8	4	5	6
s	-	x	x	x	3	4



Caminho mínimo

Utilizando matriz de adjacências

n = |V|, a complexidade do Dijkstra $O(n^2)$

Podendo ser reduzida para O((V+E)*logV) usando lista de adjacências e uma fila de prioridade min-heap-priority para armazenar vértices ainda não visitados ordenados pelos pesos das arestas.

Tempo para visitar cada vértice = O(V+E)

Tempo para processar cada vizinho de um vértice = O(logV).

total=O(V+E)∗O(logV)=O((V+E)logV)

https://www.scaler.com/topics/data-structures/dijkstra-algorithm/

CORMEN, 2002

Caminho mínimo

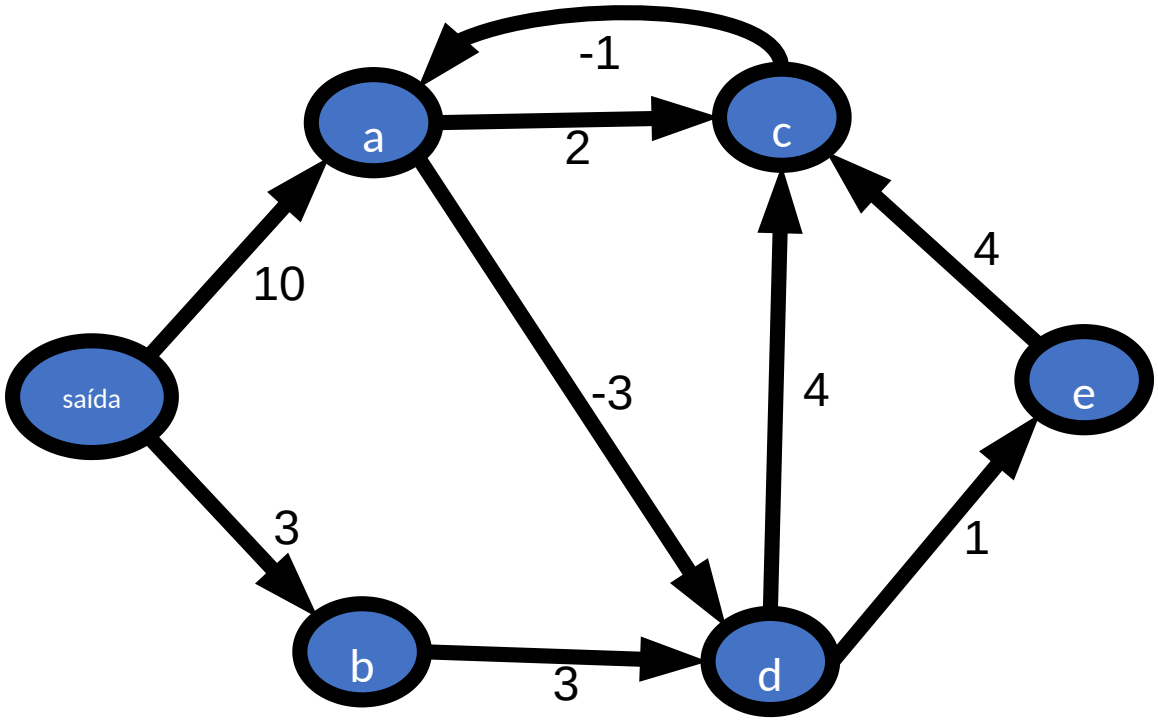
Conforme já fora mencionado, o algoritmo de Bellman-Ford também resolve o problema de caminho mínimo

Algumas diferenças entre Bellman-Ford e algoritmo de Dijkstra:

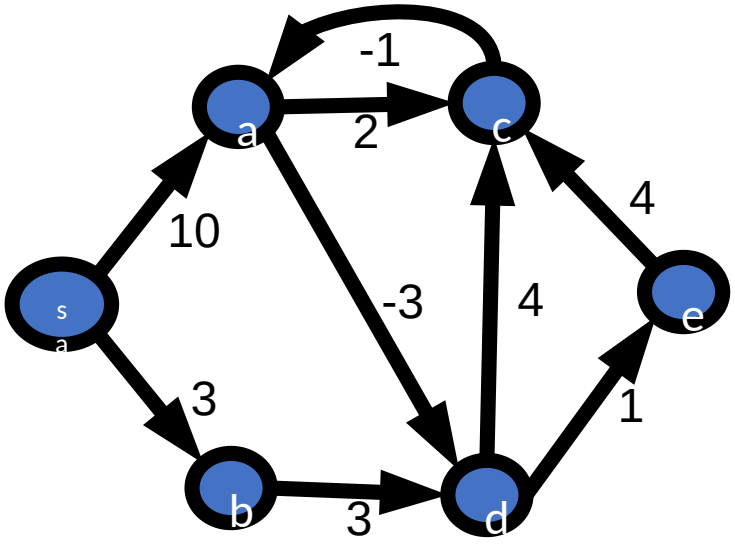
1. Dijkstra trata apenas arestas com pesos positivos. Calculando o caminho de uma fonte a um destino;
2. O algoritmo de Bellman–Ford consegue tratar arestas com pesos negativos.
3. Bellman–Ford indica se existe ou não um ciclo de peso negativo (ida e volta totaliza peso negativo) que pode ser alcançado da fonte. Se tal ciclo existe, o algoritmo sinaliza a ausência de solução. Se tal ciclo não existe, o algoritmo produz os caminhos mínimos e seus pesos de uma fonte para todos os outros vértices;

   Se num grafo existe um ciclo cuja soma dos custos das suas arestas é negativa, repetindo este ciclo, pode reduzir-se o comprimento de um caminho tanto quanto se queira [Cardoso et al.].

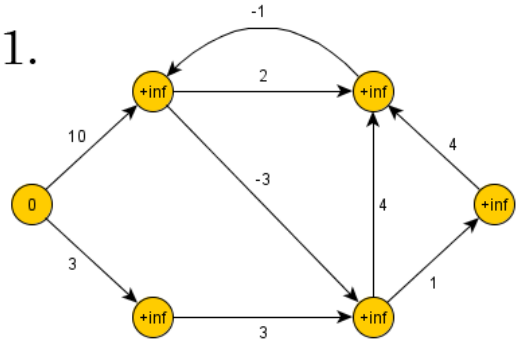
Estude o algoritmo Bellman-Ford (sugestão: pg. 538 do livro da Judith Gersting) e o aplique para encontrar o caminho mínimo entre o vértice “saída” no grafo abaixo:



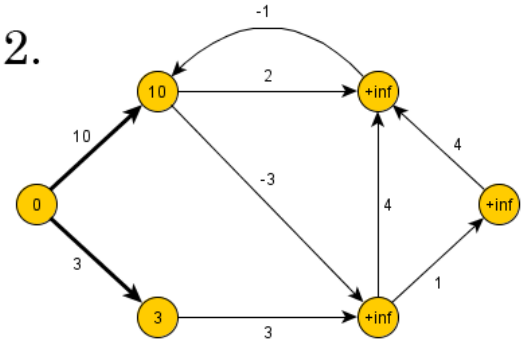
http://www.programming-algorithms.net/article/47389/Bellman-Ford-algorithm



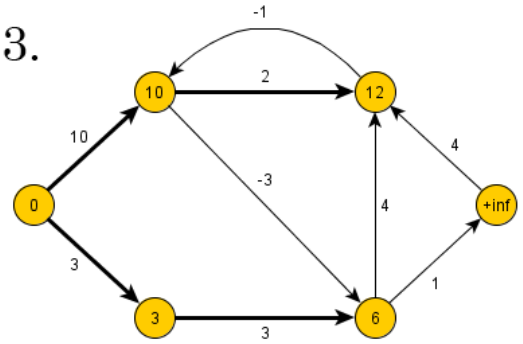
1.



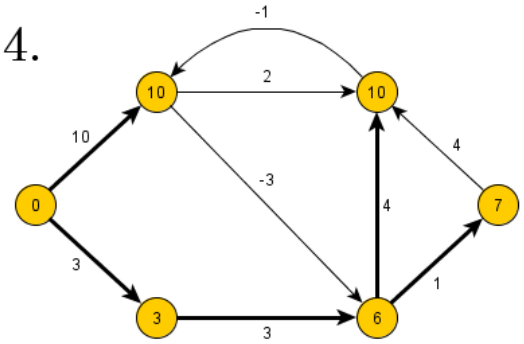
2.



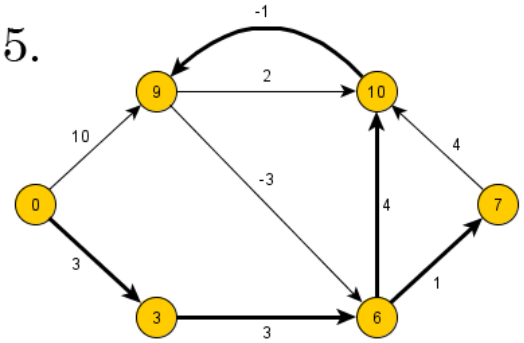
3.



4.



5.



http://www.programming-algorithms.net/article/47389/Bellman-Ford-algorithm

Percurso/Busca em um Grafo

Eventualmente queremos apenas acessar os nós de um grafo simples e conexo G(V,E) em alguma ordem sem levar em conta algum peso de aresta (como se cada aresta apresentasse peso unitário);

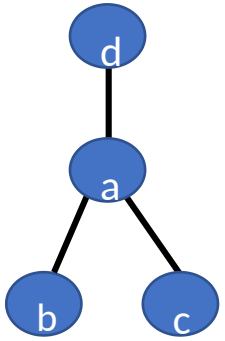
A ordem dos nós acessados determina um percurso no grafo, usualmente esse processo é conhecido como uma busca em um grafo;

Para percorrer todos os vértices os algoritmos podem precisar repetir certas arestas/vértices, porém, essas repetições não são listadas na composição final do percurso de busca;

Os processos de busca são de dois tipos: em profundidade (DFS) e em largura (BFS)

Exemplo:

busca em profundidade: no grafo ao lado, o algoritmo precisa repetir uma visita ao vértice 'a', porém, na sua listagem final teremos o seguinte resultado: d, a, b, c



Percurso/Busca em um Grafo

Vamos ao algoritmo de Busca em Largura (BFS) e Busca em Profundidade (DFS) também apresentados na seção CAMINHO MÍNIMO E ÁRVORE GERADORA MÍNIMA do livro da Judith Gersting (disponível no formato ebook e fisicamente no acervo da biblioteca do CCT).

Fundamentos Matemáticos para a Ciência da Computação

Judith L. Gersting

Busca em profundidade

Busca em Profundidade (DFS): já foi estudado em EDA para grafos do tipo árvore binária e pode ser aplicado a grafos em geral

Partindo de um *v* arbitrário, marca-se esse nó como “visitado” e se processa esse nó (um *print* do seu conteúdo, por exemplo);

A partir de *v*, visitando e processando os nós, descendo tão longe quanto possível até não existirem vértices não visitados nesse caminho. Então, retorna-se pelo mesmo caminho explorando quaisquer caminhos laterais, até voltar ao vértice inicial *v*.

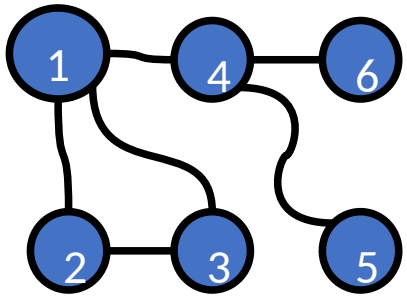
Depois exploramos possíveis caminhos ainda inexplorados restantes a partir de *v*.

Busca em profundidade

Grafo G(V,E) grafo simples e conexo:
Estruturas auxiliares:

- Matriz de adjacências (MatAdj)
- Vetor de marcações (vetMarca) dos visitados inicialmente zerada

Chamada:
DFS(grafo, raiz, MatAdj, vetMarca)



Função:

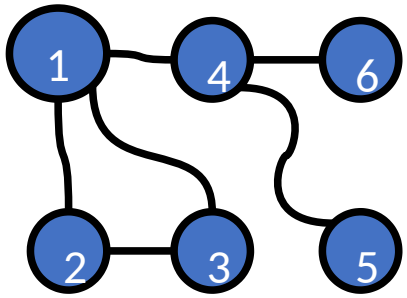
```
DFS(G,v, M, marca)
        marca[v]=1
        print(v)
        para cada w adjacente a v
                se marca[w]=0
                        DFS(G,n,M,marca)
                senão
                        retorna
        retorna
```

Busca em profundidade

Raiz= v1

vértice	1	2	3	4	5	6
Marca	0	0	0	0	0	0

Chamada:
DFS(grafo,v1,MatAdj,vetMarca)

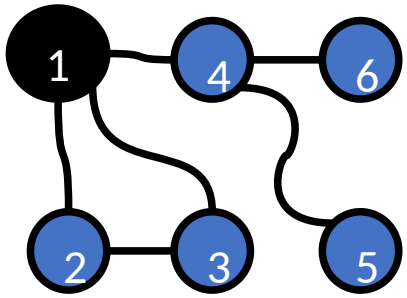


Busca em profundidade

Raiz= v1

vértice	1	2	3	4	5	6
Marca	x	0	0	0	0	0

Chamada:
DFS(grafo,v1,MatAdj,vetMarca)
… print v1
DFS(grafo,v2,MatAdj,vetMarca)



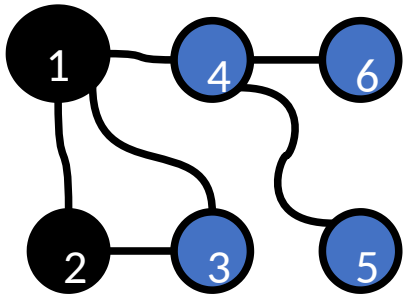
Busca em profundidade

Raiz= v1

vértice	1	2	3	4	5	6
Marca	x	x	0	0	0	0

Chamada:
DFS(grafo,v1,MatAdj,vetMarca)

DFS(grafo,v2,MatAdj,vetMarca)
...print v2



Busca em profundidade

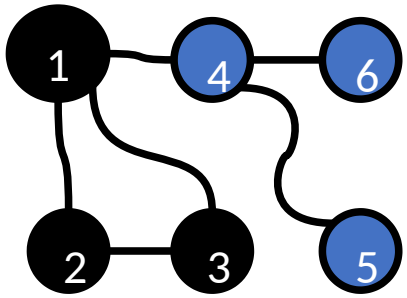
Raiz= v1

vértice	1	2	3	4	5	6
Marca	x	x	x	0	0	0

Chamada:
DFS(grafo,v1,MatAdj,vetMarca)

DFS(grafo,v2,MatAdj,vetMarca)

DFS(grafo,v3,MatAdj,vetMarca)
...print v3



Busca em profundidade

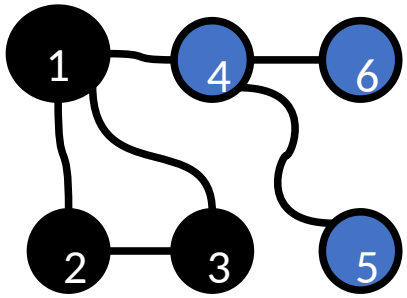
Raiz= v1

vértice	1	2	3	4	5	6
Marca	x	x	x	0	0	0

Chamada:
DFS(grafo,v1,MatAdj,vetMarca)

DFS(grafo,v2,MatAdj,vetMarca)

DFS(grafo,v3,MatAdj,vetMarca)

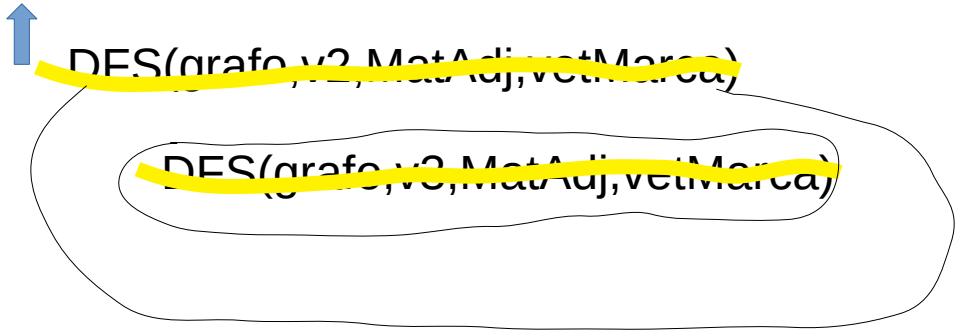


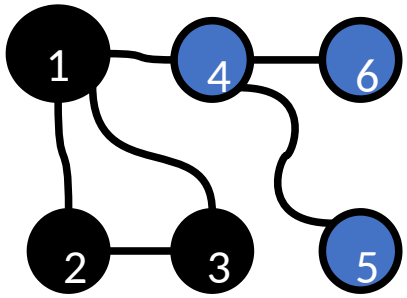
Busca em profundidade

Raiz= v1

vértice	1	2	3	4	5	6
Marca	x	x	x	0	0	0

Chamada:
DFS(grafo,v1,MatAdj,vetMarca)





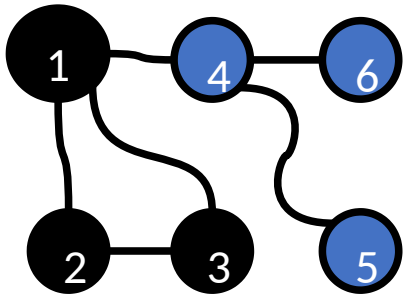
Busca em profundidade

Raiz= v1

vértice	1	2	3	4	5	6
Marca	x	x	x	0	0	0

Chamada:
DFS(grafo,v1,MatAdj,vetMarca)

para cada *w* adjacente a *v*
DFS(grafo,v4,MatAdj,vetMarca)



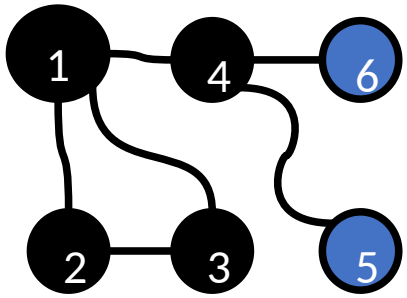
Busca em profundidade

Raiz= v1

-------- 1 2 3 4 5 6
Marca x x x x 0 0

Chamada:
DFS(grafo,v1,MatAdj,vetMarca)

DFS(grafo,v4,MatAdj,vetMarca)
...print v4



Busca em profundidade

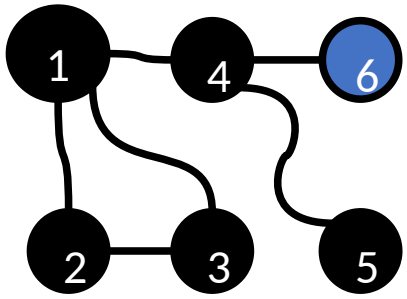
Raiz= v1

-------- 1 2 3 4 5 6
Marca x x x x x 0

Chamada:
DFS(grafo,v1,MatAdj,vetMarca)

DFS(grafo,v4,MatAdj,vetMarca)

DFS(grafo,v5,MatAdj,vetMarca)
...print v5



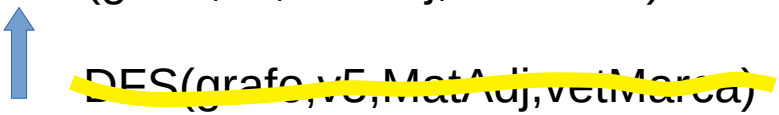
Busca em profundidade

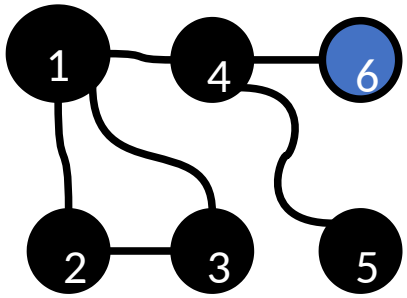
Raiz= v1

-------- 1 2 3 4 5 6
Marca x x x x x 0

Chamada:
DFS(grafo,v1,MatAdj,vetMarca)

DFS(grafo,v4,MatAdj,vetMarca)





Busca em profundidade

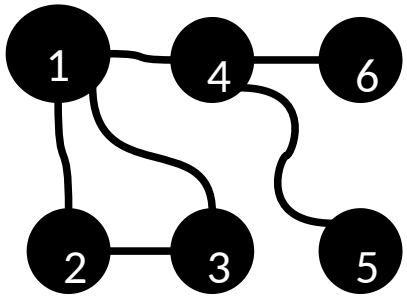
Raiz= v1

-------- 1 2 3 4 5 6
Marca x x x x x 0

Chamada:
DFS(grafo,v1,MatAdj,vetMarca)

DFS(grafo,v4,MatAdj,vetMarca)

DFS(grafo,v6,MatAdj,vetMarca)
...print v6



Busca em profundidade

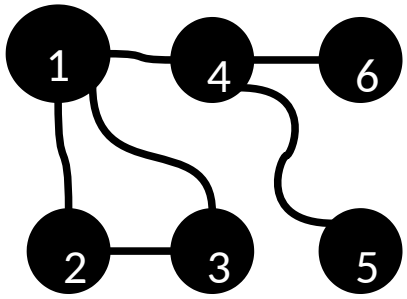
Raiz= v1

-------- 1 2 3 4 5 6
Marca x x x x x x

Chamada:
DFS(grafo,v1,MatAdj,vetMarca)

DFS(grafo,v4,MatAdj,vetMarca)

DFS(grafo,v6,MatAdj,vetMarca)

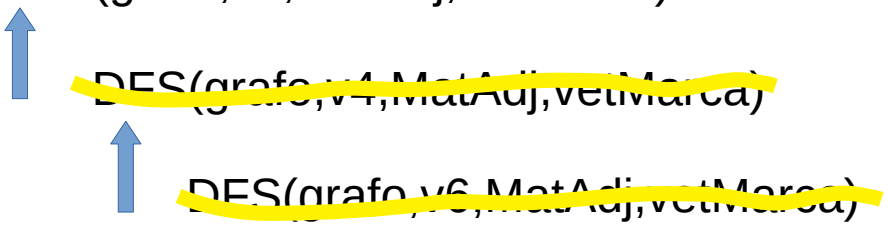


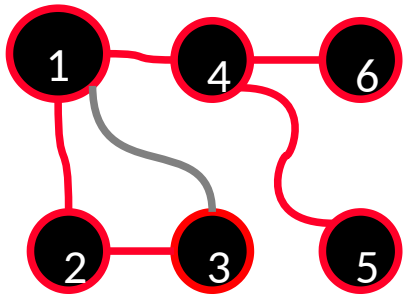
Busca em profundidade

Raiz= v1

-------- 1 2 3 4 5 6
Marca x x x x x x

Chamada:
DFS(grafo,v1,MatAdj,vetMarca)





Busca em Largura

Busca em Largura (BFS): também já foi estudado em EDA para grafos do tipo árvore binária e pode ser aplicado a grafos em geral

A partir de um vértice arbitrário *v*, procuram-se todos os seus vértices adjacentes, depois os adjacentes a esses e assim por diante, como círculos concêntricos de ondas em um pequeno lago.

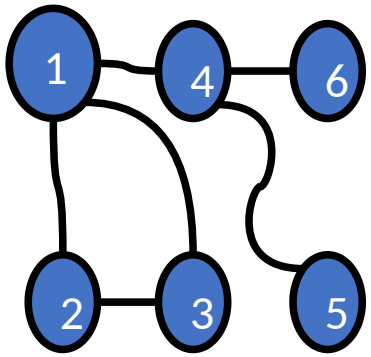
Busca em Largura

Grafo G(V,E) simples e conexo:
Estruturas auxiliares:

- Matriz de adjacências (MatAdj);
- Vetor de marcações (vetMarca) dos visitados inicialmente zerada

Chamada: BFS(grafo,raiz,MatAdj)
Função: BFS(G,v,M)

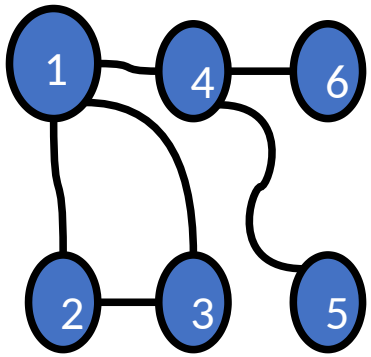
```
marca[ ] = zeros
F = cria(Fila) /* F é uma fila */
marca[v]=1
print(v)
insere(F,v)
enquanto F não é vazia:
    para cada w adjacente à frente(F):
        se marca[w]=0
            marca[w]=1
            print(w)
            insere(F,w)
    remove(F)
```



Busca em Largura

Raiz= v1

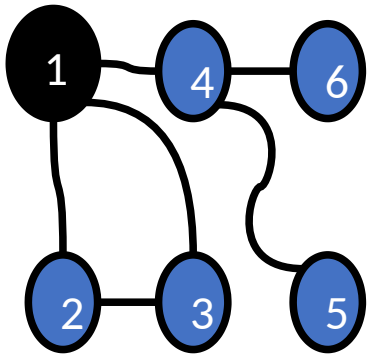
Chamada:
BFS(grafo,v1,M)



Busca em Largura

Raiz= v1	
antes	depois
vértice 1 2 3 4 5 6 Marca 0 0 0 0 0 0 Frente:-- Cauda:-- Fila: --	vértice 1 2 3 4 5 6 Marca x 0 0 0 0 0 Frente: Cauda: Fila: v1
print	v1

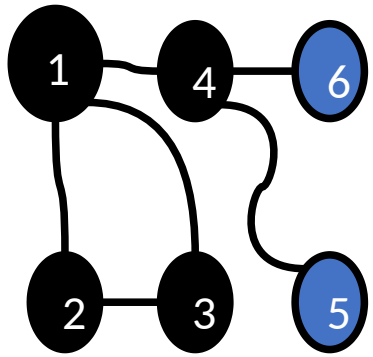
```
BFS(grafo,v,M)
    cria marca[ ] e cria
fila[ ]
    marca[v]=1
    print(v)
    insere(F,v)
```



Busca em Largura

Raiz= v1	
antes	depois
vértice 1 2 3 4 5 6 Marca x 0 0 0 0 0 Frente:1 Cauda:1 Fila: v1	vértice 1 2 3 4 5 6 Marca x x x x 0 0 Frente: 1 Cauda: 4 Fila: v1 v2 v3 v4
print	v1 v2 v3 v4

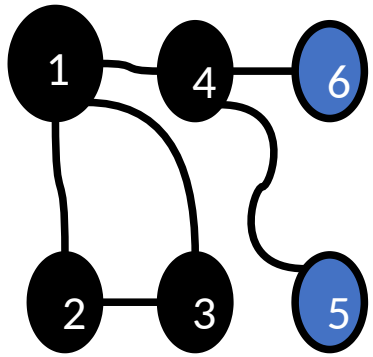
```
BFS(grafo,v,M)
    …
    enquanto fila F não vazia (1ª iteração)
        para cada w adjacente à frente(F)
            se marca[w]=0
                marca[w]=1;
                print(w);
                insere(F,w)
```



Busca em Largura

Raiz= v1	
antes	depois
vértice 1 2 3 4 5 6 Marca x x x x 0 0 Frente: 1 Cauda: 4 Fila: v1 v2 v3 v4	vértice 1 2 3 4 5 6 Marca x x x x 0 0 Frente: 2 Cauda: 4 Fila: ~~v1~~ v2 v3 v4
print	v1 v2 v3 v4

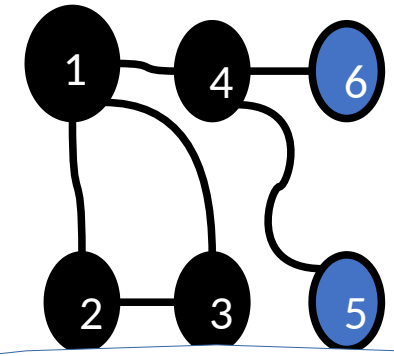
```
BFS(grafo,v,M)
    …
    enquanto fila F não vazia (1ª iteração)
        para cada w adjacente à frente(F)
            se marca[w]=0
                marca[w]=1;
                print(w);
                insere(F,w)
        remove(F)
```



Busca em Largura

Raiz= v1	
antes	depois
vértice 1 2 3 4 5 6 Marca x x x x 0 0 Frente: 2 Cauda: 4 Fila: v2 v3 v4	vértice 1 2 3 4 5 6 Marca x x x x 0 0 Frente: 3 Cauda: 4 Fila: ~~v2~~ v3 v4
print	v1 v2 v3 v4

```
BFS(grafo,v,M)
    ...
    enquanto fila F não vazia (2ª iteração)
        para cada w adjacente à frente(F)
            se marca[w]=0
                marca[w]=1;
                print(w);
                insere(F,w)
        remove(F)
```

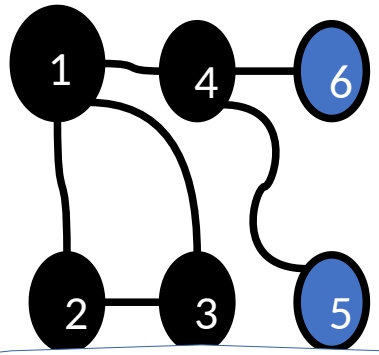


v2: seus adjacentes *w* já foram todos marcados. Ocorre apenas a remoção de v2 da fila

Busca em Largura

Raiz= v1	
antes	depois
vértice 1 2 3 4 5 6 Marca x x x x 0 0 Frente: 3 Cauda: 4 Fila: v3 v4	vértice 1 2 3 4 5 6 Marca x x x x 0 0 Frente: 4 Cauda: 4 Fila: ~~v3~~ v4
print	v1 v2 v3 v4

```
BFS(grafo,v,M)
    ...
    enquanto fila F não vazia (2ª iteração)
        para cada w adjacente à frente(F)
            se marca[w]=0
                marca[w]=1;
                print(w);
                insere(F,w)
        remove(F)
```

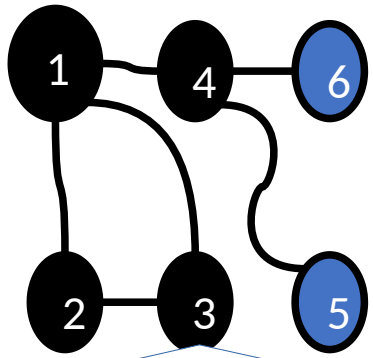


v3: seus adjacentes *w* já foram todos marcados. Ocorre apenas a remoção de v3 da fila

Busca em Largura

Raiz= v1	
antes	depois
vértice 1 2 3 4 5 6 Marca x x x x 0 0 Frente: 4 Cauda: 4 Fila: v4	vértice 1 2 3 4 5 6 Marca x x x x x x Frente: 4 Cauda: 6 Fila: v4 v5 v6
print	v1 v2 v3 v4 v5 v6

```
BFS(grafo,v,M)
    …
    enquanto fila F não vazia (2ª iteração)
        para cada w adjacente à frente(F)
            se marca[w]=0
                marca[w]=1;
                print(w);
                insere(F,w)
```

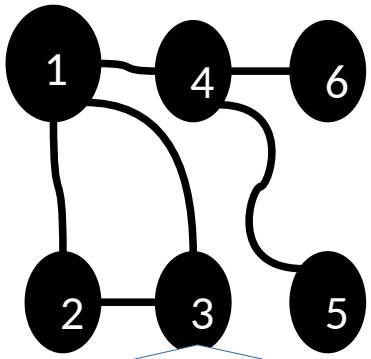


v4: há adjacentes não marcados.

Busca em Largura

Raiz= v1	
antes	depois
vértice 1 2 3 4 5 6 Marca x x x x x x Frente: 4 Cauda: 6 Fila: v4 v5 v6	vértice 1 2 3 4 5 6 Marca x x x x x x Frente: 5 Cauda: 6 Fila: ~~v4~~ v5 v6
print	v1 v2 v3 v4 v5 v6

```
BFS(grafo,v,M)
    ...
    enquanto fila F não vazia (2ª iteração)
        para cada w adjacente à frente(F)
            se marca[w]=0
                marca[w]=1;
                print(w);
                insere(F,w)
        remove(F)
```

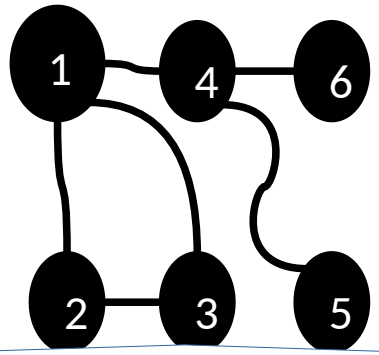


v4: há adjacentes não marcados.

Busca em Largura

Raiz= v1	
antes	depois
vértice 1 2 3 4 5 6 Marca x x x x x x Frente: 5 Cauda: 6 Fila: v5 v6	vértice 1 2 3 4 5 6 Marca x x x x x x Frente: 6 Cauda: 6 Fila: ~~v5~~ v6
print	v1 v2 v3 v4 v5 v6

```
BFS(grafo,v,M)
    ...
    enquanto fila F não vazia (3ª iteração)
        para cada w adjacente à frente(F)
            se marca[w]=0
                marca[w]=1;
                print(w);
                insere(F,w)
        remove(F)
```

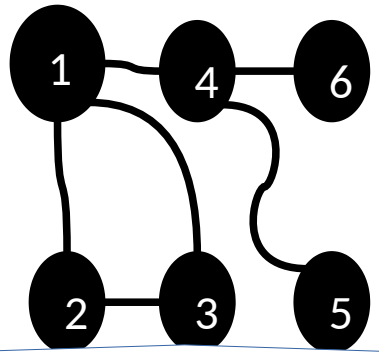


v5: seus adjacentes já foram todos marcados. Ocorre apenas a remoção de v5 da fila

Busca em Largura

Raiz= v1	
antes	depois
vértice 1 2 3 4 5 6 Marca x x x x x x Frente: 6 Cauda: 6 Fila: v6	vértice 1 2 3 4 5 6 Marca x x x x x x Frente: 7 Cauda: 6 Fila: ~~v6~~
print	v1 v2 v3 v4 v5 v6

```
BFS(grafo,v,M)
    ...
    enquanto fila F não vazia (3ª iteração)
        para cada w adjacente à frente(F)
            se marca[w]=0
                marca[w]=1;
                print(w);
                insere(F,w)
        remove(F)
```



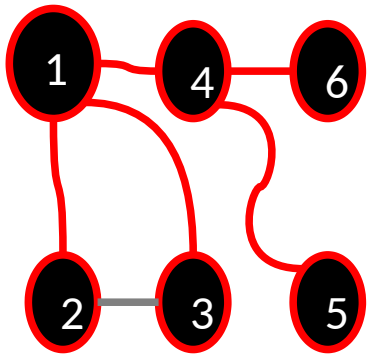
v6: seus adjacentes já foram todos marcados. Ocorre apenas a remoção de v6 da fila

Busca em Largura

Raiz= v1	
vértice 1 2 3 4 5 6 Marca x x x x x x Frente: Cauda: Fila: v a z i a	
Percurso BFS	v1 v2 v3 v4 v5 v6

Função:
BFS(grafo,v,M)

…
Fila vazia:
Fim do “enquanto fila F não vazia”



Exercício

Como você poderia aplicar percursos (DFS ou BFS) para determinar se um grafo é desconexo? Descreva uma estratégia.

Suponha que se deseja conhecer a distribuição (tamanho e quantidade) de cada componente conexo em um grafo, como você aplicaria BFS ou DFS para obter esse histograma de componentes conexos de um grafo?

Suponha que se deseje levantar uma medida que descreve a frequência na qual cada aresta se encontra no caminho mais curto entre pares de nós em uma rede (grafo). Descreva uma abordagem para tal computação. O que essa frequência significa em termos da intermediação operada por essas arestas na estrutura do grafo como uma rede de comunidades? O que acontece se for operada uma remoção sucessiva dessas arestas em termos do isolamento de comunidades subjacentes da rede?

TEG

Bibliografia

Básica

LUCCHESI, C. L. et alli. Aspectos Teóricos da Computação, Parte C: Teoria dos Grafos, projeto Euclides, 1979.

SANTOS, J. P. O. et alli. Introdução à Análise Combinatória. UNICAMP; 1995.

SZWARCFITER, J. L. Grafos e Algoritmos Computacionais. Campus, 1986.

GERSTING, Judith L. Fundamentos Matemáticos para a Ciência da Computação. Rio de Janeiro. 3a Ed. Editora.

Complementar:

1.) CORMEN, T. Introduction to Algorithms, third edition, MIT press, 2009

2.) ROSEN, K. Discrete Mathematics and its applications, seventh edition, McGraw Hill, 2011.

3.) WEST, Douglas, B. Introduction to Graph Theory, second edition, Pearson, 2001.

4.) BONDY, J.A., MURTY, U.S.R., Graph Theory with applications , Springer, 1984.

5.) SEDGEWICK, R. Algorithms in C - part 5 - Graph Algorithms, third edition, 2002, Addison-Wesley.

6.) GOLDBARG, M., GOLDBARG E., Grafos: Conceitos, algoritmos e aplicações. Editora Elsevier, 2012.

7.) BONDY, J.A., MURTY, U.S.R., Graph Theory with applications , Springer, 1984

8.) FEOFILOFF, P., KOHAYAKAWA, Y., WAKABAYASHI, Y., uma introdução sucinta à teoria dos grafos. 2011. (www.ime.usp.br/~pf/teoriadosgrafos)

9.) DIESTEL, R. Graph Theory, second edition, springer, 2000

10.) FURTADO, A. L. Teoria de grafos. Rio de janeiro. Editora LTC. 1973.

11.) WILSON, R.J. Introduction to Graph Theory. John Wiley & Sons Inc., 1985

12.) BOAVENTURA NETTO , P. O. Grafos: Teoria, Modelos, Algoritmos. Edgard Blucher, SP, quinta edição

Tutoriais, artigos, notas de aula...

Vários livros podem ser acessados np formato eletrônico (e-book) via

https://www.udesc.br/bu/acervos/ebook

Exemplos:

Teoria Computacional de Grafos - Os Algoritmos

Jayme Luiz Szwarcfiter

Fundamentos Matemáticos para a Ciência da Computação

Judith L. Gersting

Grafos

Marco Goldbarg

Algoritmos - Teoria e Prática

Thomas Cormen