Algoritmos e Estruturas de Dados

Grafos

Slides baseados em:

- **ZIVIANI, N. Projetos de Algoritmos - com implementações em Java e C++. Thomson Learning, 2007. Cap 7.**
- **CORMEN, H.T.; LEISERSON, C.E.; RIVEST, R.L. Introduction to Algorithms, MIT Press, McGraw-Hill, 1999.**
- **Slides Daniel R. Figueiredo**

Profa. Karina Valdivia Delgado

EACH-USP

03/08/2011

O que é um grafo?

- Abstração que permite codificar relacionamentos entre pares de objetos (definição informal)

Em que:

Os objetos são os vértices do grafo

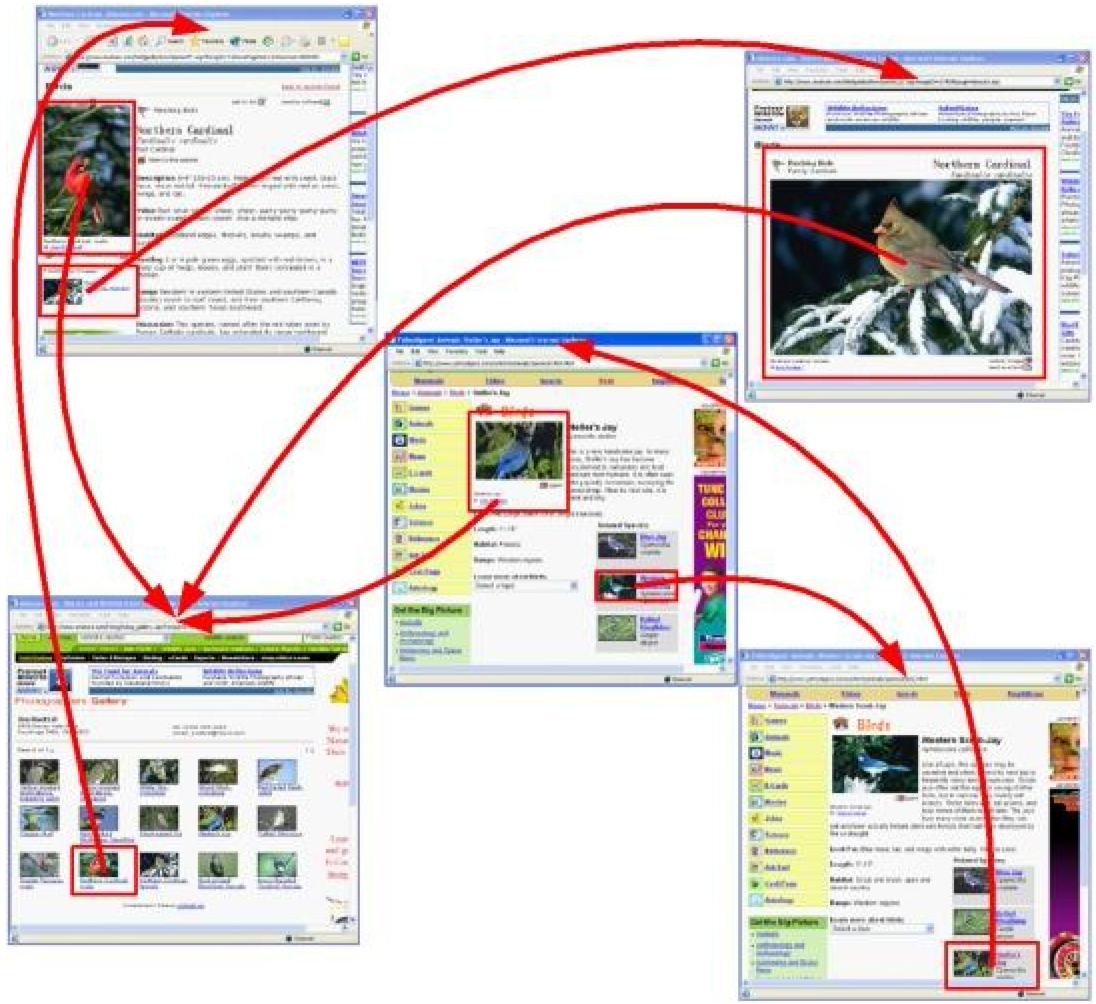
Os relacionamentos são as arestas do grafo

Motivação: Transporte aéreo

- objeto: cidades
- relacionamento: vôo comercial entre duas cidades

Motivação: Páginas web

- objeto: páginas web
- relacionamento: link de uma página para outra



Motivação: Poder da abstração

- Existe um tipo abstrato de dados (TAD=conjunto de operações associado a uma estrutura de dados) usado para modelar tais situações!!

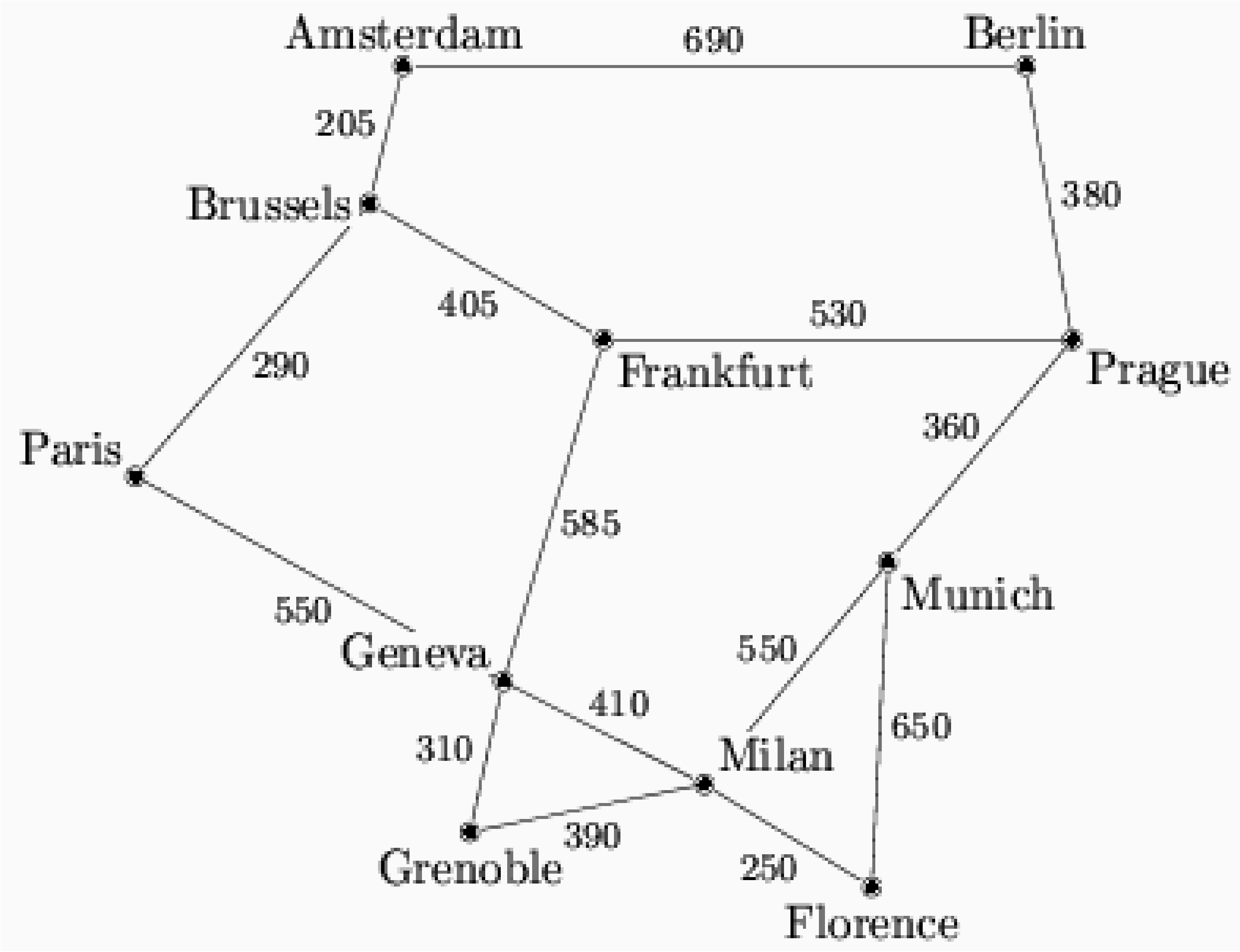
grafo

- Muitos problemas podem ser resolvidos com o mesmo algoritmo em cima da abstração

Motivação: Viagem entre cidades

- Quantos caminhos existem para ir de Prague até Amsterdam?
- Qual é o menor(melhor) caminho entre Prague e Amsterdam?
- Existe um caminho para ir de uma cidade a outra?

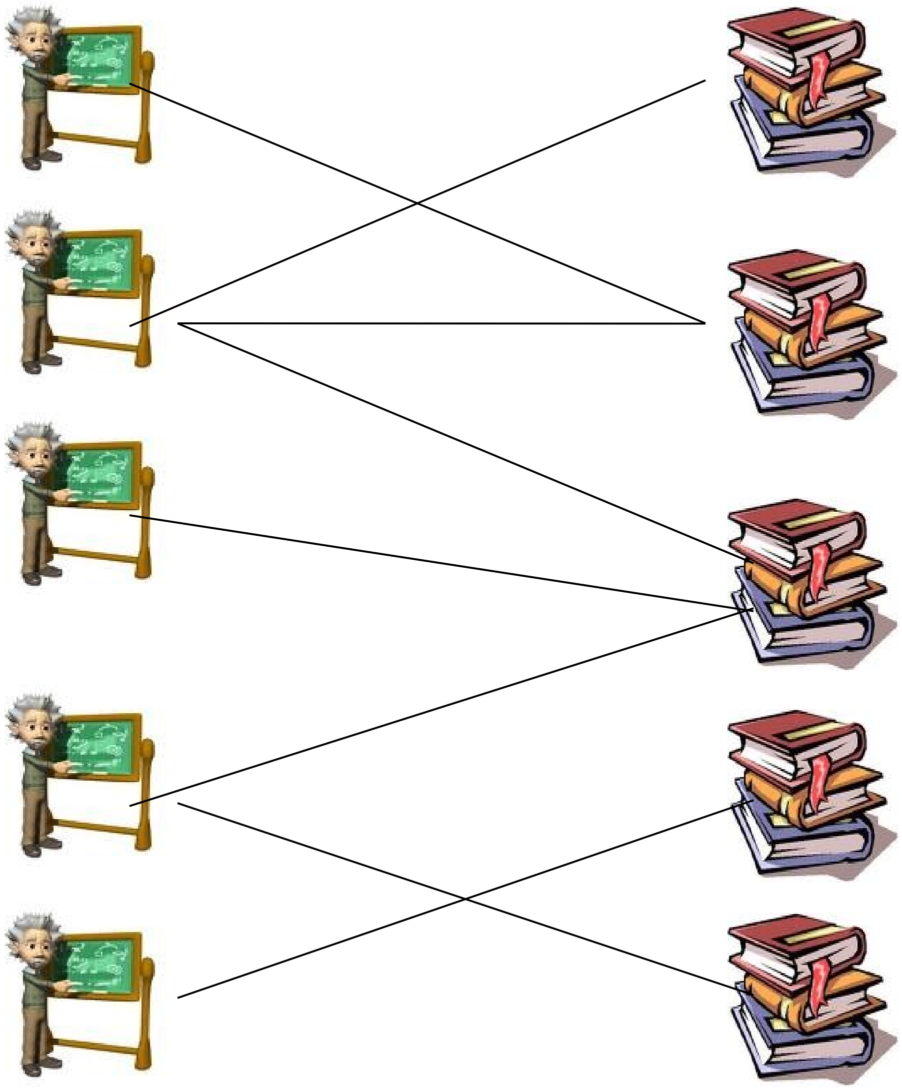
Motivação: Viagem entre cidades



Motivação: Alocação de professores

- Temos N professores e M disciplinas
- Dado o que cada professor pode lecionar, é possível que as M disciplinas sejam oferecidas simultaneamente?
- Qual o maior número de disciplinas que podem ser oferecidas?

Motivação: Alocação de professores

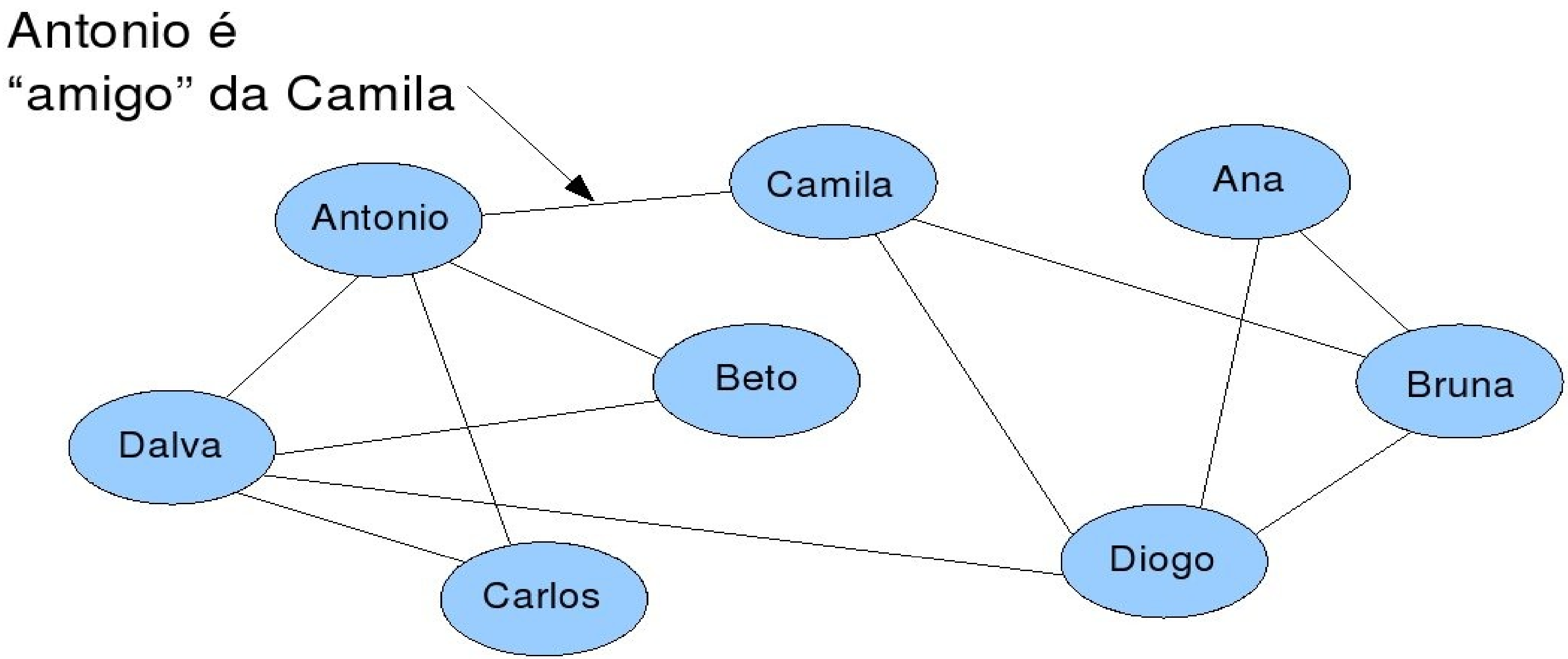


Motivação: Pesquisando no Orkut

- Como saber se duas pessoas estão conectadas (interligadas via relacionamentos declarados) através de uma sequência de relacionamentos?
- Qual é o menor caminho entre duas pessoas?

Motivação: Pesquisando no Orkut

- Vértices: profiles (pessoas)
- Arestas: relacionamentos declarados

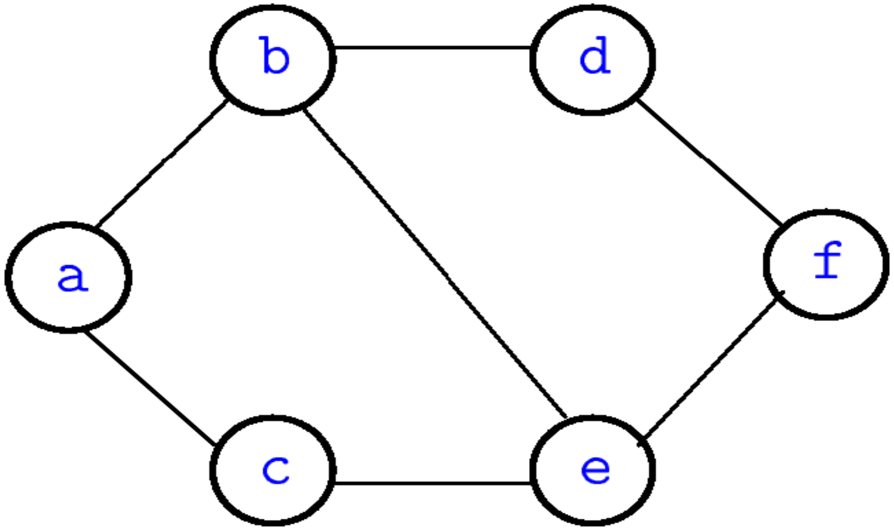


Conceitos Básicos

Conceitos Básicos

- Grafo: conjunto de vértices e arestas.
- Vértice: objeto simples que pode ter nome e outros atributos.
- Aresta: conexão entre dois vér tices.

Exemplo: representação usual

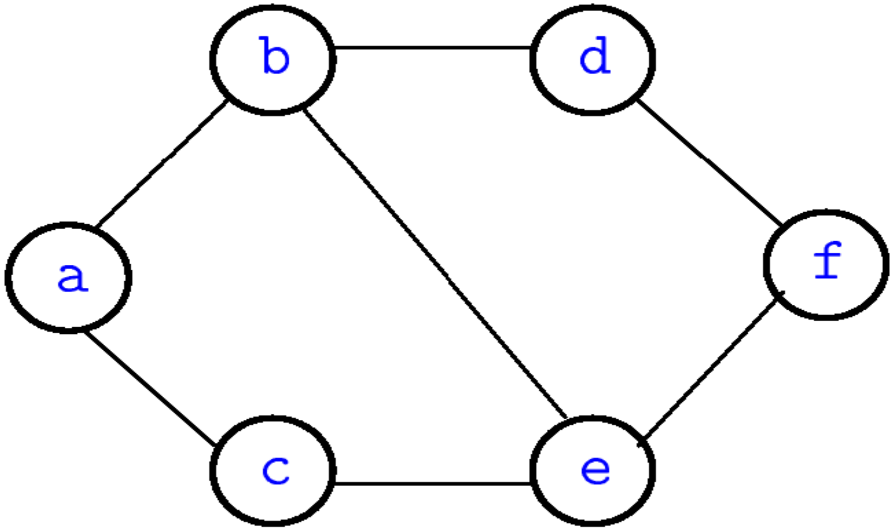


Conceitos Básicos

- Notação: G=(V,A)

Em que

- G: grafo
- V: conjunto de vértices
- A: conjunto de arestas

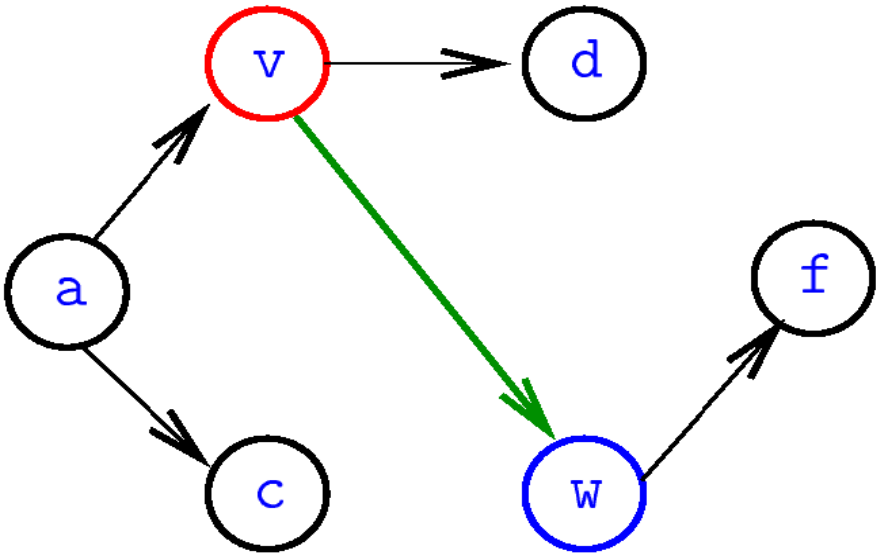


Grafos direcionados

- Um grafo direcionado G é um par (V, A), em que:
  - V é um conjunto finito de vértices
  - A é uma relação binária em V (i.e., uma aresta é um par ordenado de vértices)

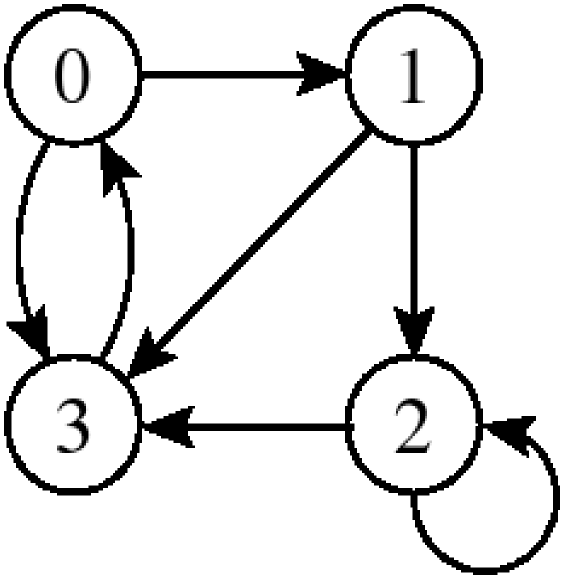
Grafos direcionados

- Uma aresta (v, w) sai do vértice v e entra no vértice w . O vértice w é adjacente ao vértice v.



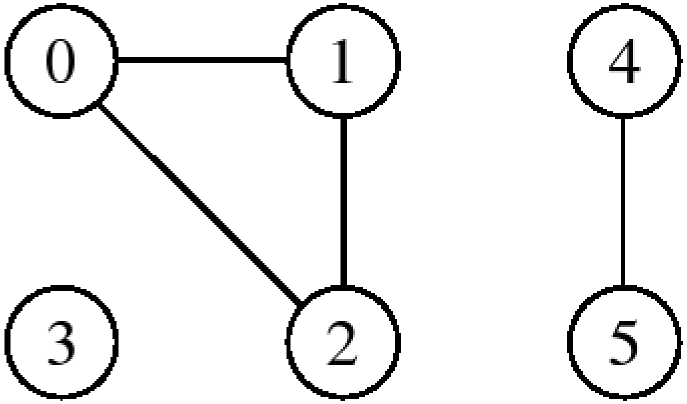
Grafos direcionados

- Podem existir arestas de um vértice para ele mesmo, chamadas de self-loops.



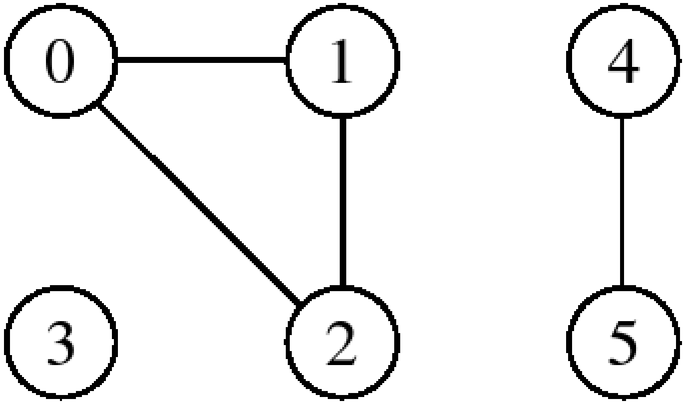
Grafos não direcionados

- Um grafo não direcionado G é um par (V, A), em que:
  - O conjunto de arestas A é constituído de pares de vértices não ordenados.
  - As arestas (u, v) e (v , u) são consideradas como uma única aresta.
  - A relação de adjacência é simétrica.
  - Self-loops não são permitidos.



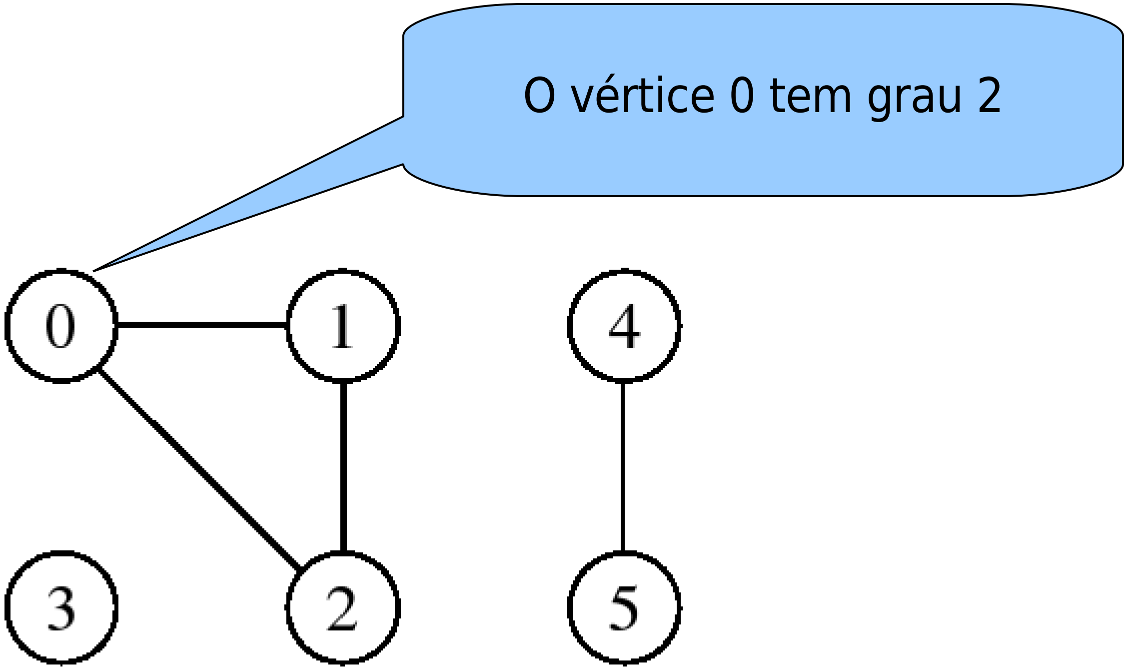
Grau de um vértice em grafos não direcionados

- O grau de um vértice é o número de arestas que incidem nele.
- Um vérice de grau zero é dito isolado ou não conectado.



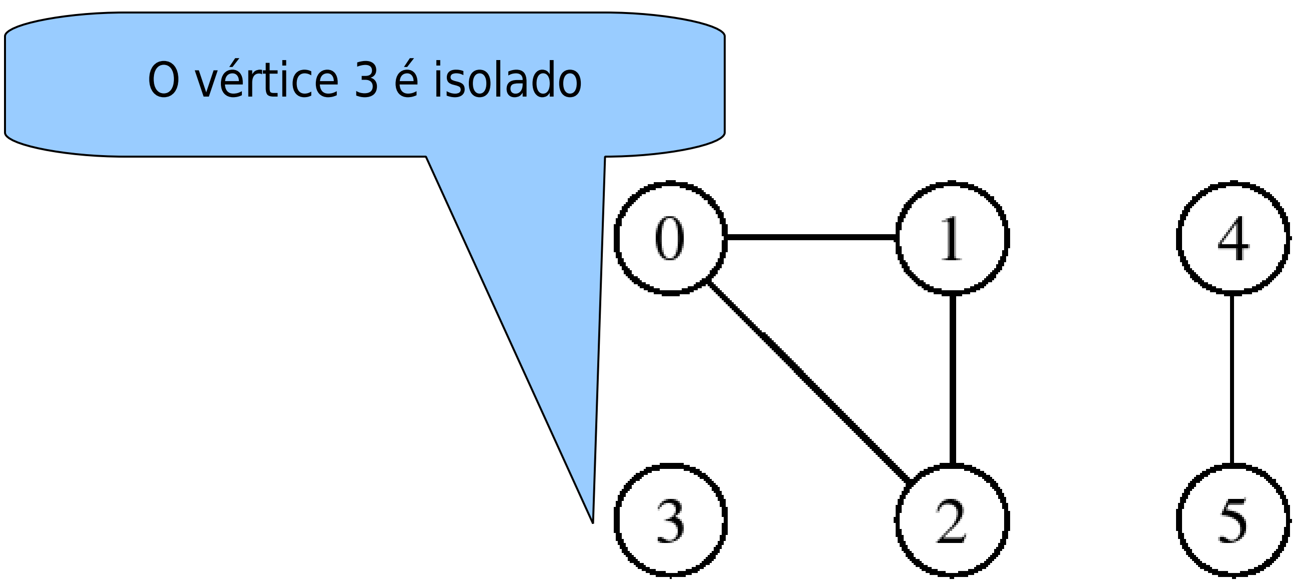
Grau de um vértice em grafos não direcionados

– O grau de um vér tice é o número de arestas que incidem nele.
– Um vérice de grau zero é dito isolado ou não conectado.



Grau de um vértice: em grafos não direcionados

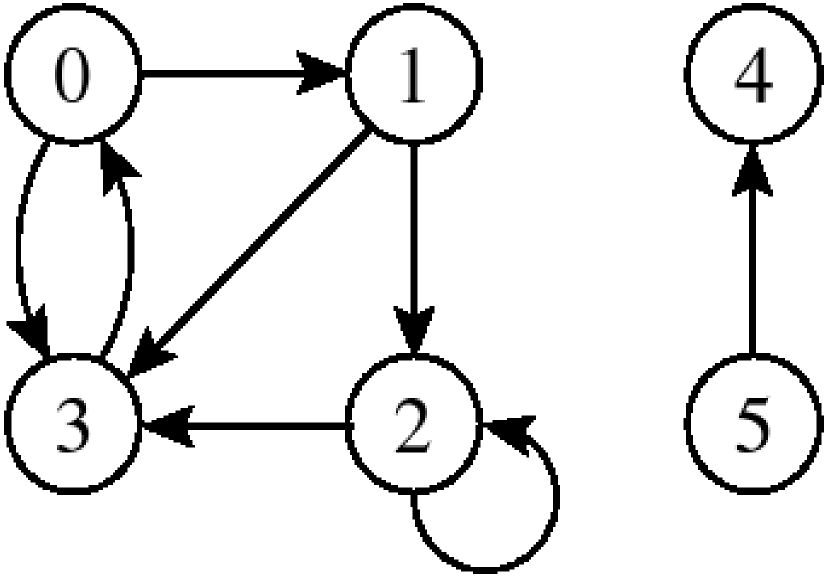
- O grau de um vér tice é o número de arestas que incidem nele.
- Um vérice de grau zero é dito isolado ou não conectado.



Grau de um vértice em grafos direcionados

- Grau de saída: número de arestas que saem do vértice.
- Grau de entrada: número de arestas que chegam no vértice.
- Grau de um vértice:

grau de saída + grau de entrada.

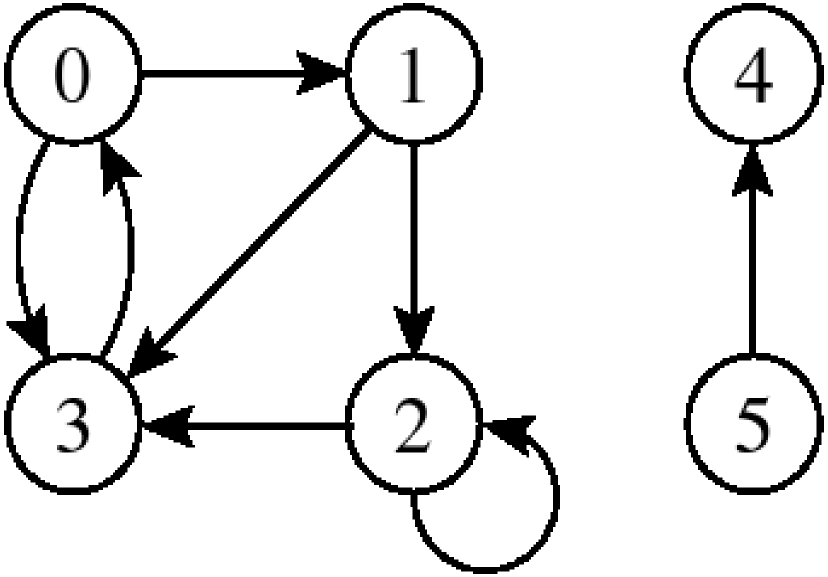


Grau de um vértice em grafos direcionados

- Grau de saída: número de arestas que saem do vértice.
- Grau de entrada: número de arestas que chegam no vértice.
- Grau de um vértice:

grau de saída + grau de entrada.

O vértice 0 tem grau de saída 2, grau de entrada 1 e grau 3.



Caminho entre vértices

- Um caminho de comprimento k de um vértice x a um vér tice y em um grafo $G = (V, A)$ é uma seqüência de vértices

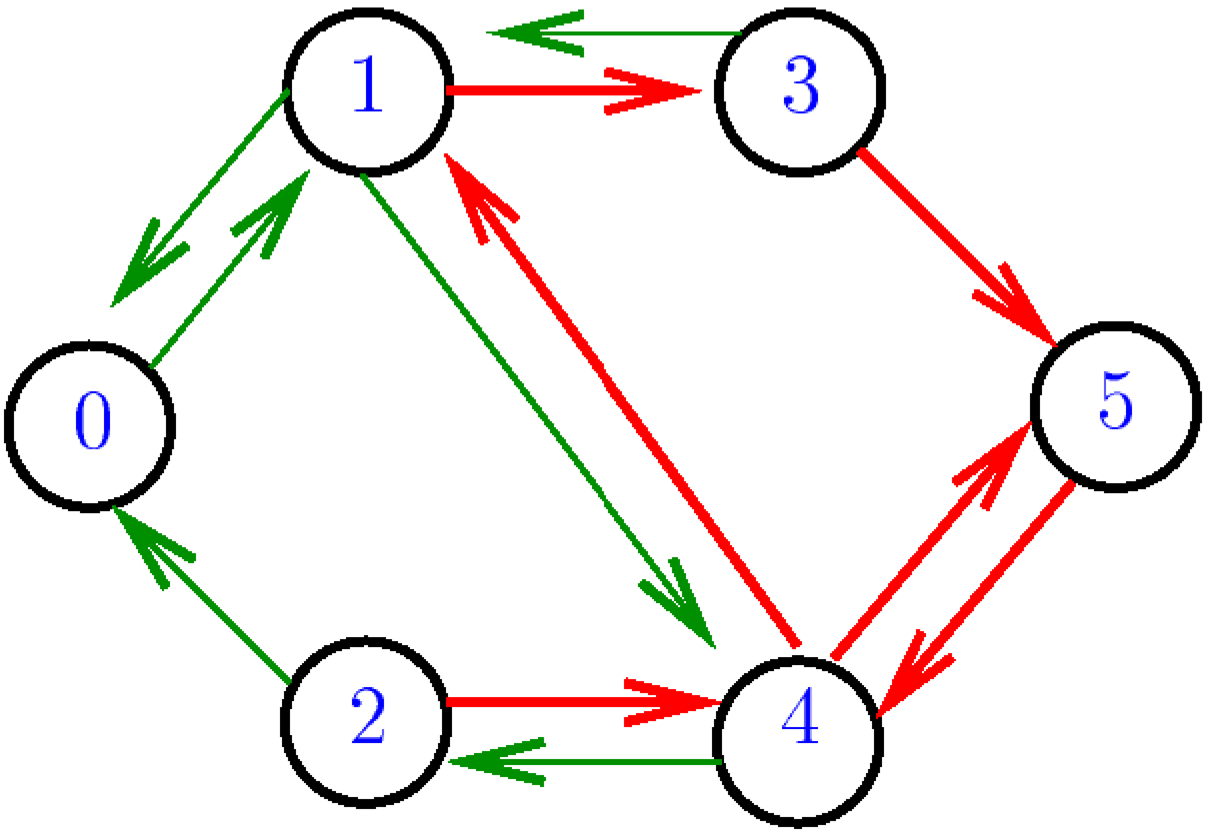
$$(v_0, v_1, v_2, \ldots, v_k)$$

 tal que $x = v_0$ e $y = v_k$, e $(v_{i-1}, v_i) \in A$ para $i = 1, 2, \ldots, k$.

- O comprimento de um caminho é o número de arestas nele

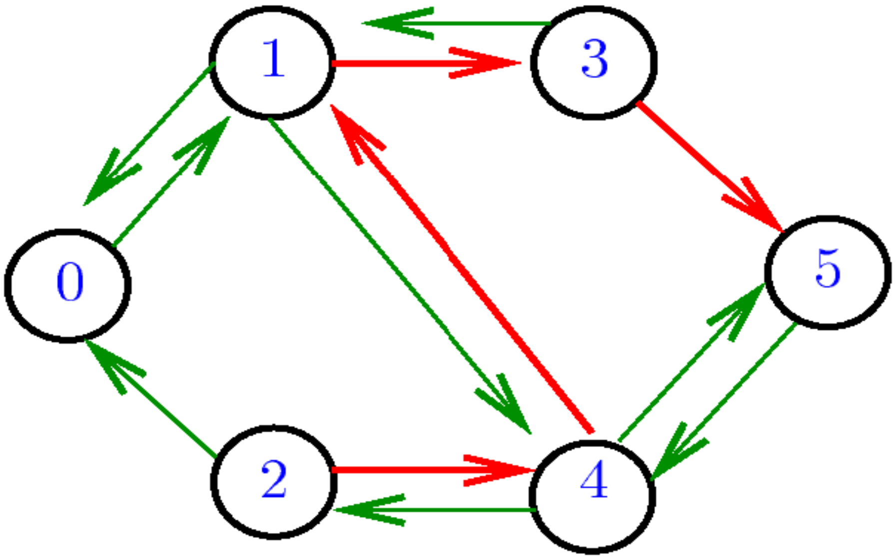
Caminho entre vértices

(2,4,1,3,5,4,5) é um caminho do vértice 2 até o vértice 5 de comprimento 6



Caminho entre vértices

- Se existir um caminho c de x a y então y é alcançável a partir de x via c.
- Um caminho é simples se todos os vértices do caminho são distintos. Exemplo: (2,4,1,3,5)

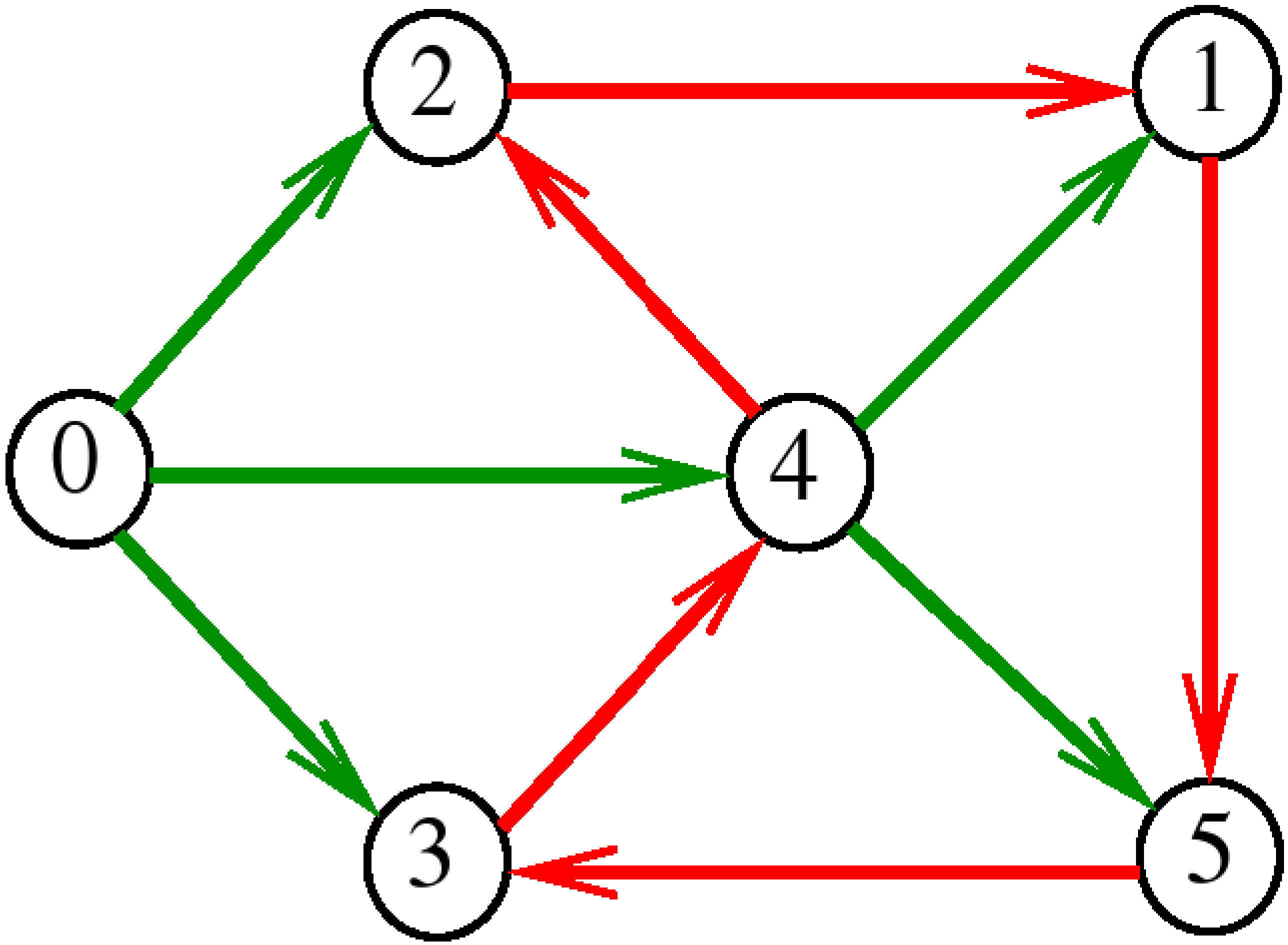


Ciclos em um grafo direcionado

- Um caminho $(v_0, v_1, \ldots, v_k)$ forma um ciclo se $v_0 = v_k$ e o caminho contém pelo menos uma aresta.
- O ciclo é simples se os vértices $v_1, v_2, \ldots, v_k$ são distintos.
- O self-loop é um ciclo de tamanho 1.

Ciclos em um grafo direcionado

- Exemplo: (2,1,5,3,4,2) é um ciclo simples

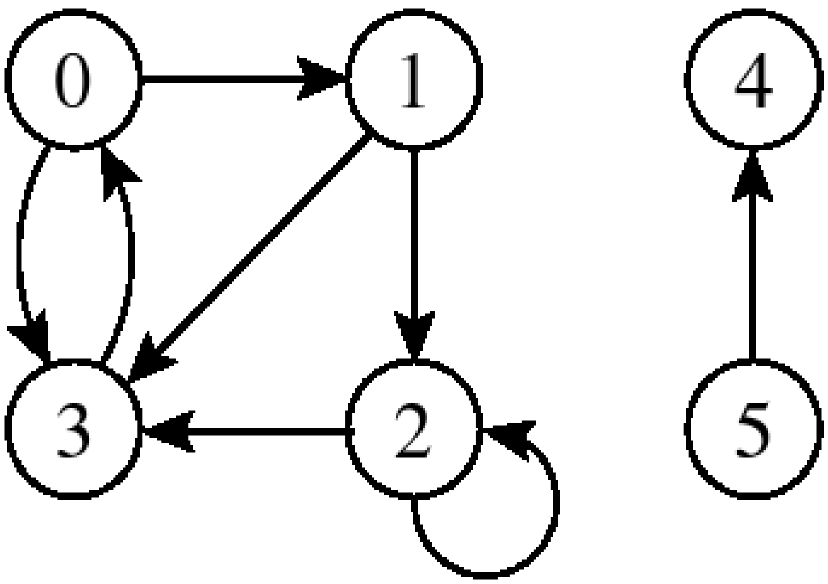


Ciclos em um grafo direcionado

- Dois caminhos $(v_0, v_1, \ldots, v_k)$ e $(v'_0, v'_1, \ldots, v'_k)$ formam o mesmo ciclo se existir um inteiro j tal que $v'_i = v_{(i+j) \bmod k}$ para $i = 0, 1, \ldots, k-1$.

Ciclos em um grafo direcionado

- Exemplo: o caminho(0, 1, 3, 0) forma o mesmo ciclo que os caminhos (1, 3, 0, 1) e (3, 0, 1, 3).

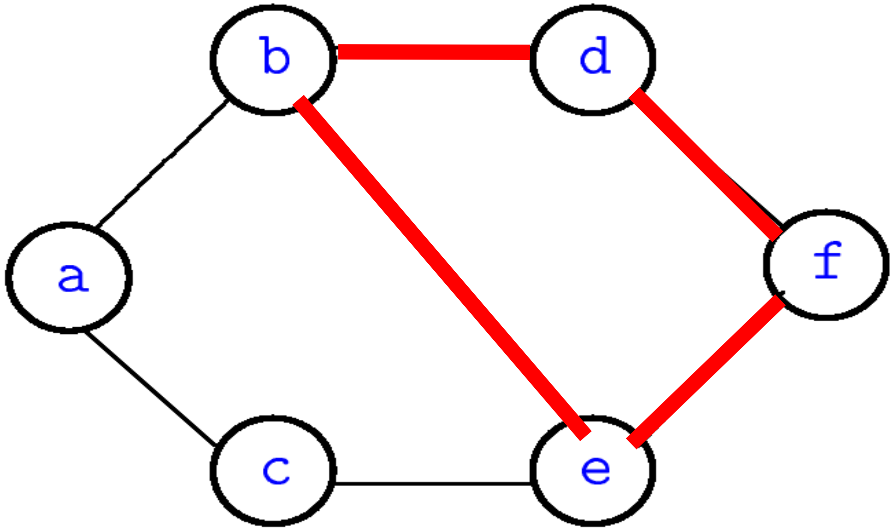


Ciclos em um grafo não direcionado

- Um caminho $(v_0, v_1, \ldots, v_k)$ forma um ciclo se $v_0 = v_k$ e o caminho contém pelo menos três arestas.
- O ciclo é simples se os vértices $v_1, v_2, \ldots, v_k$ são distintos.

Ciclos em um grafo não direcionado

- Exemplo: o caminho (b, d, f, e, b) é um ciclo simples

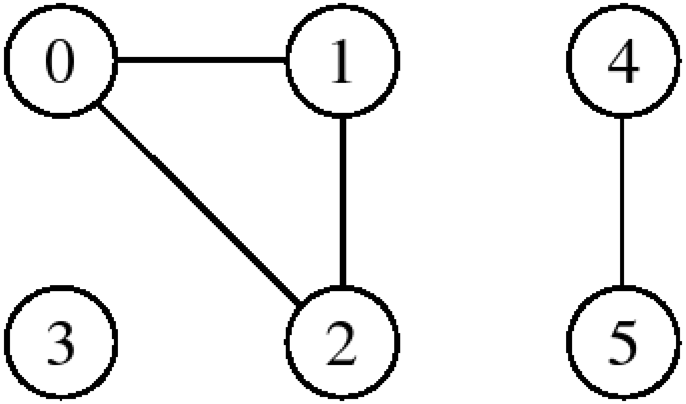


Componentes conectados

- Um grafo não direcionado é conectado se cada par de vértices está conectado por um caminho.
- Os componentes conectados são as porções conectadas de um grafo.
- Um grafo não direcionado é conectado se ele tem exatamente um componente conectado.

Componentes conectados

- Os componentes conectados são: {3} {0,1,2} e {4,5} e o grafo não é conectado uma vez que ele tem mais de um componente conectado.

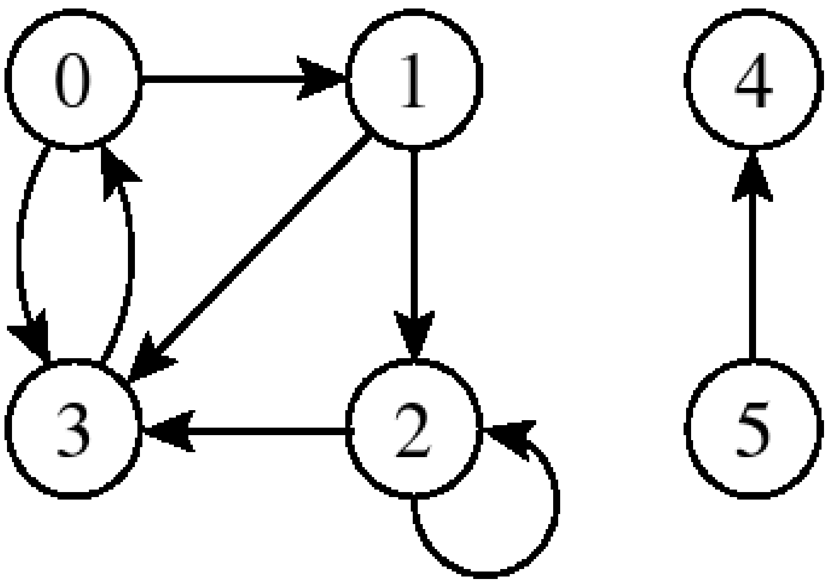


Componentes fortemente conectados

- Um grafo direcionado G = (V,A) é fortemente conectado se cada dois vértices quaisquer são alcançáveis um a partir do outro.
- Os componentes fortemente conectados de um grafo direcionado são conjuntos de vértices sob a relação "são mutuamente alcançáveis".
- Um grafo direcionado fortemente conectado tem apenas um componente fortemente conectado.

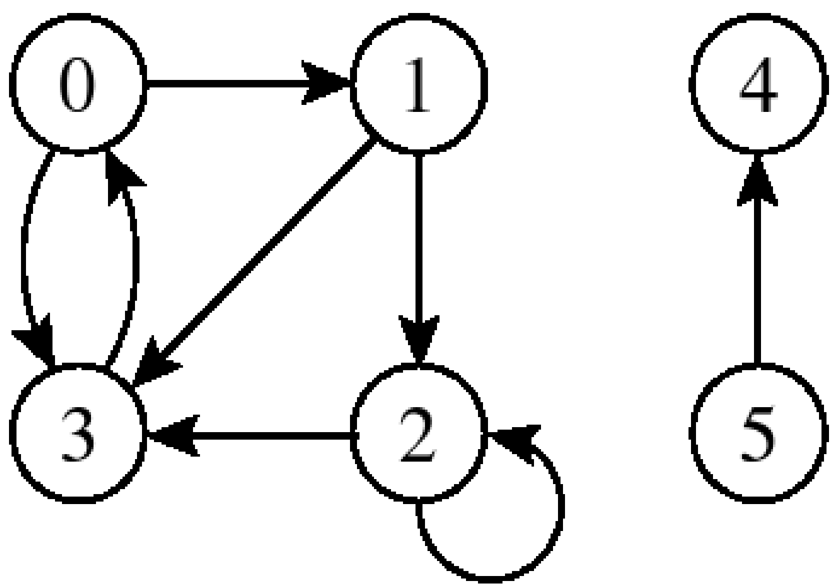
Componentes fortemente conectados

- quais os componentes fortemente conectados?



Componentes fortemente conectados

- {0, 1, 2, 3}, {4} e {5} são os componentes fortemente conectados
- {4, 5} não é um componente fortemente conectado

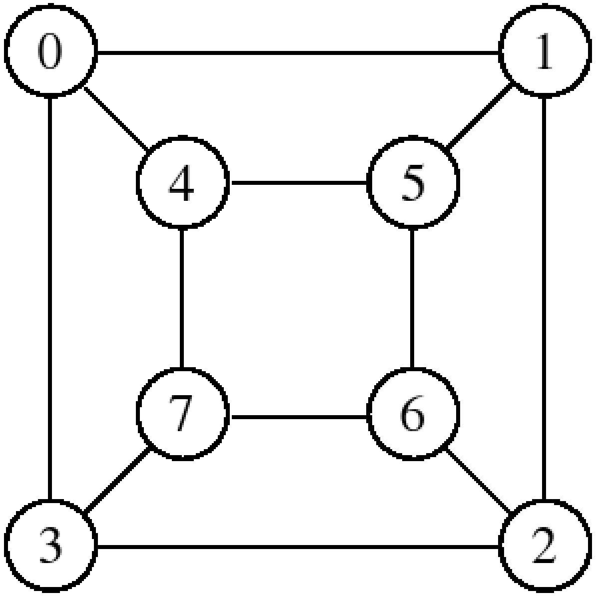


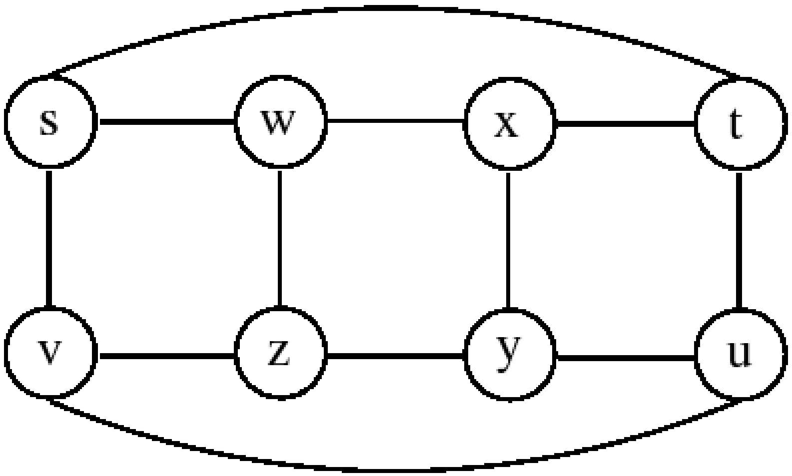
Grafos Isomorfos

- G = (V , A) e G'= (V', A') são isomorfos se existir uma bijeção f : V → V' tal que

  $(u, v) \in A$ se e somente se $(f(u), f(v)) \in A'$.

É possível re-rotular os vértices de G para serem rótulo de G'?

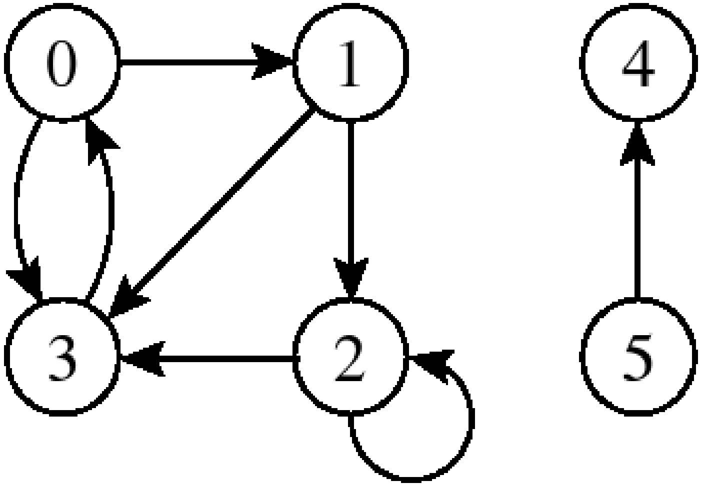




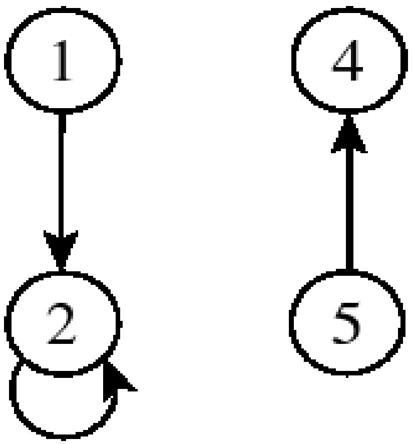
Subgrafos

- Um grafo $G' = (V', A')$ é um subgrafo de $G = (V, A)$ se $V' \subseteq V$ e $A' \subseteq A$.
- Dado um conjunto $V' \subseteq V$, o subgrafo induzido por V' é o grafo $G' = (V', A')$, em que $A' = \{(u, v) \in A \mid u, v \in V'\}$.

Subgrafos



- O subgrafo induzido pelo conjunto de vértices V'={1,2,4,5} é:

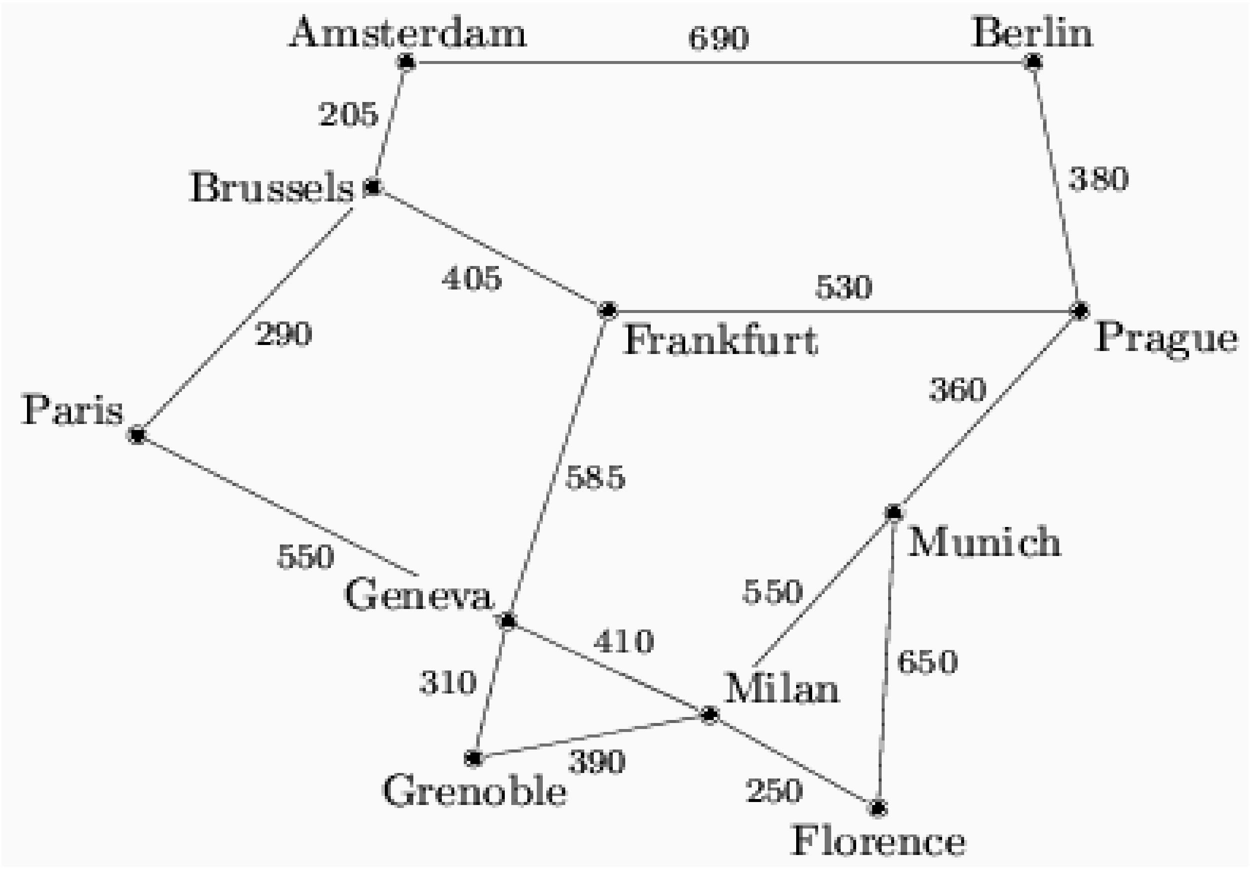


Outras clasificações de grafos

- **Grafo ponderado:** possui pesos associados às arestas.

Outras classificações de grafos

- **Grafo ponderado:** possui pesos associados às arestas.



Outras clasificações de grafos

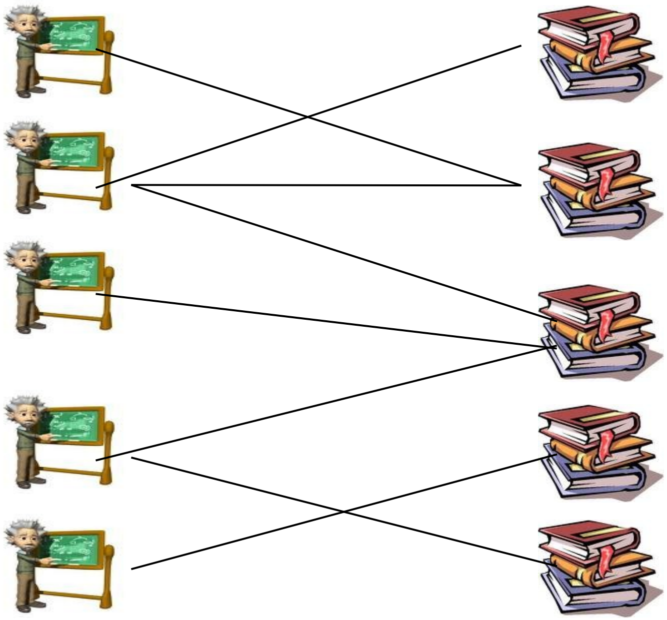
- **Grafo bipartido:** grafo não direcionado

  G = (V , A) no qual V pode ser particionado em dois conjuntos V1 e V2 tal que (u, v) ∈ A implica que u ∈ V1 e v ∈ V2 ou u ∈ V2 e v ∈ V1 (todas as arestas ligam os dois conjuntos V1 e V2).

Outras clasificações de grafos

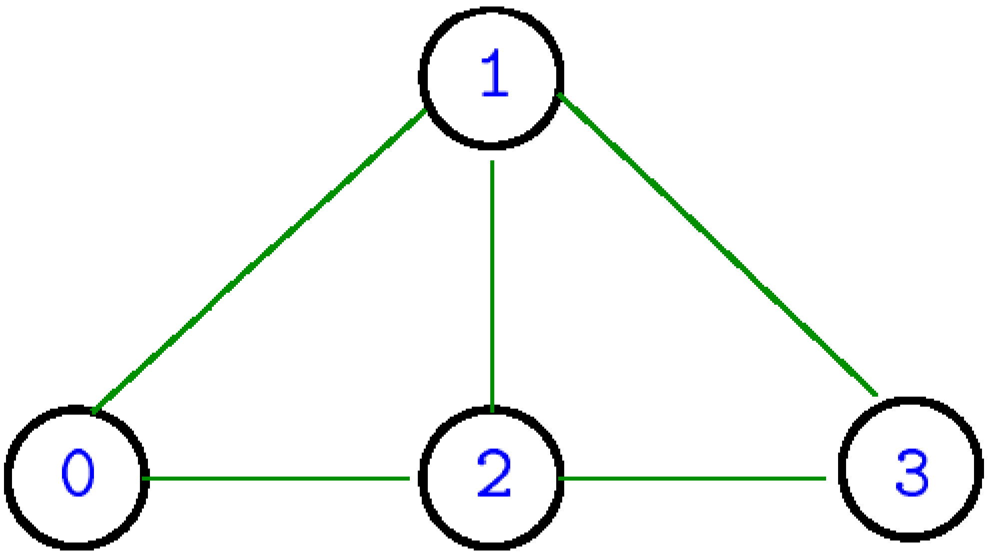
- **Grafo bipartido:** grafo não direcionado

  G = (V , A) no qual V pode ser particionado em dois conjuntos V1 e V2 tal que (u, v) ∈ A implica que u ∈ V1 e v ∈ V2 ou u ∈ V2 e v ∈ V1 (todas as arestas ligam os dois conjuntos V1 e V2).



Grafos completos

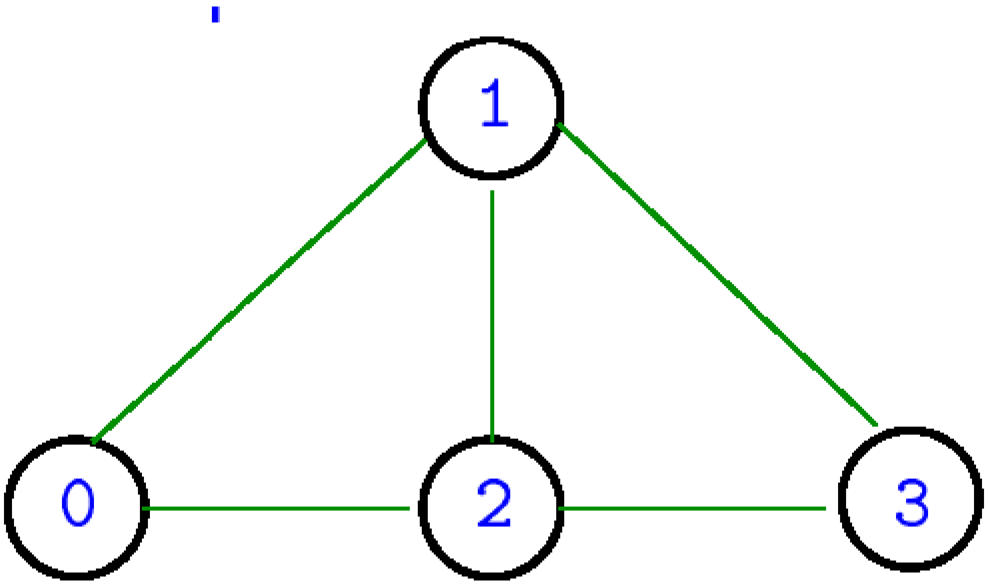
- Um grafo completo é um grafo não direcionado no qual todos os pares de vértices são adjacentes.



É completo?

Grafos completos

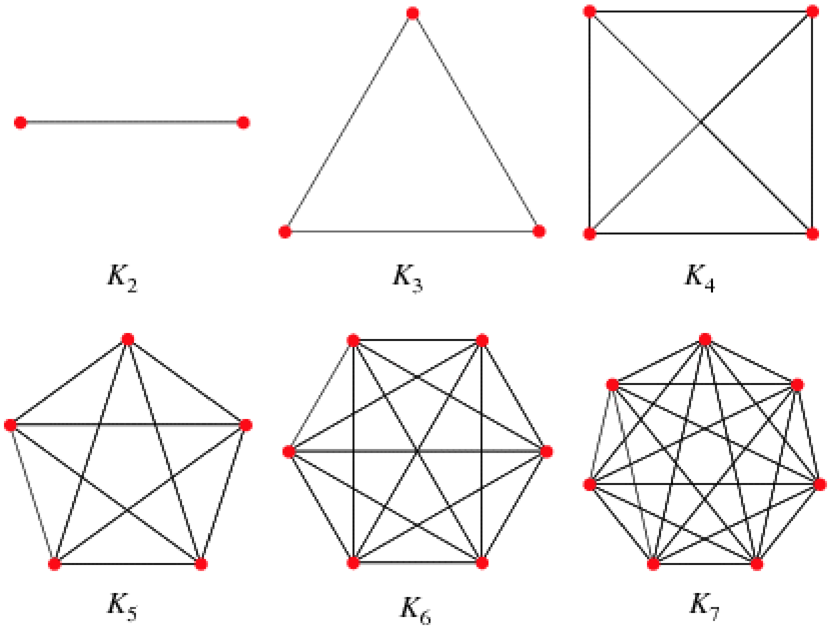
- Um grafo completo é um grafo não direcionado no qual todos os pares de vértices são adjacentes.



É completo?
não

Grafos completos

- Um grafo completo é um grafo não direcionado no qual todos os pares de vértices são adjacentes.



Grafos completos

- Quantas arestas possui um grafo completo com V vértices?

Grafos completos

- Quantas arestas possui um grafo completo com V vértices?
- Possui $(|V|^2 - |V|)/2 = |V|(|V| - 1)/2$ arestas, pois do total de $|V|^2$ pares possíveis de vértices devemos subtrair $|V|$ self-loops e dividir por 2 (cada aresta ligando dois vértices é contada duas vezes no grafo direcionado).

Grafos completos

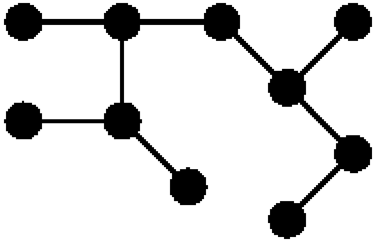
- Qual o número total de grafos diferentes com $|V|$ vértices?

Grafos completos

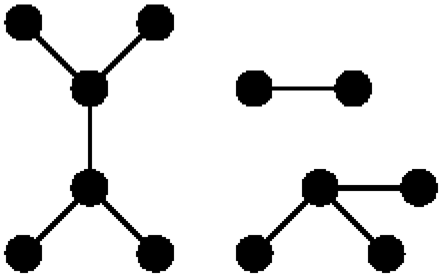
- Qual o número total de grafos diferentes com $|V|$ vértices? Ou seja, qual o número de maneiras diferentes de escolher um subconjunto a partir de $|V|(|V| - 1)/2$ possíveis arestas?

Árvores e floresta

- Árvore livre: grafo não direcionado acíclico e conectado. É comum dizer apenas que o grafo é uma árvore omitindo o “livre”.
- Floresta: grafo não direcionado acíclico, podendo ou não ser conectado. É um conjunto de árvores.



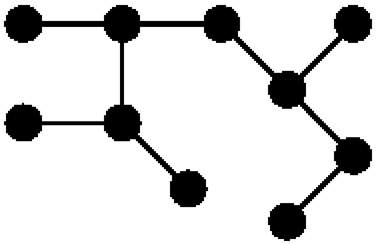
Árvore livre



Floresta

Árvores e floresta

- Cuantas arestas tem uma arvore com n vértices?



O tipo abstrato de dados Grafo

- Importante considerar os algoritmos em grafos como tipos abstratos de dados (TAD).
- TAD= Conjunto de operações associado a uma estrutura de dados.
- Independência de implementação para as operações.

Operações do TAD Grafo

- Criar um grafo vazio.
- Inserir uma aresta no grafo.
- Verificar se existe determinada aresta no grafo.
- Obter a lista de vértices adjacentes a determinado vértice.
- Eliminar uma aresta do grafo.
- Imprimir um grafo.
- Obter o número de vértices do grafo.
- Obter o número de arestas do grafo
- Obter a aresta de menor peso de um grafo.

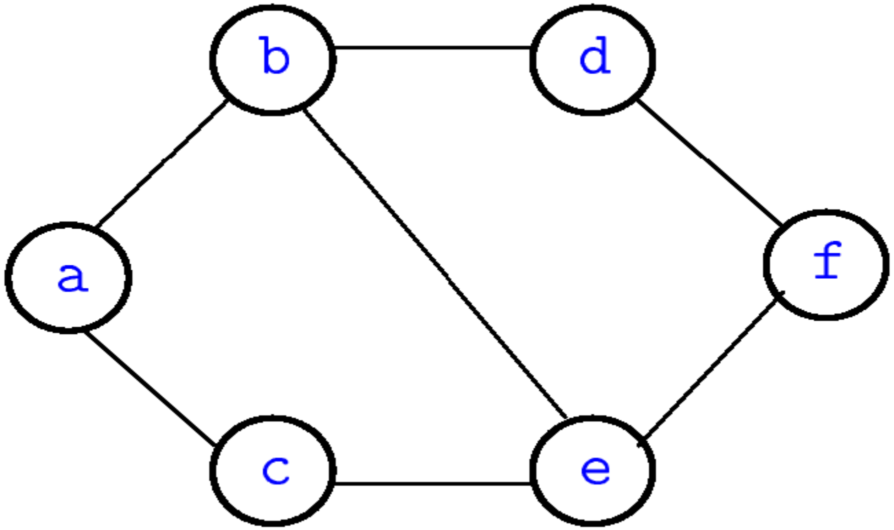
Representação de Grafos

Representação de grafos

G=(V,A)

V={a, b c, d, e f}

A={(a,b), (a,c), (b,d), (b,e), (c,e), (d,f), (e,f) }



Como um grafo pode ser representado no computador?

Representações de grafos

- Como uma matriz de adjacências
- Como uma coleção de listas de adjacências

Qual a representação mais eficiente ou mais adequada?

Representações de grafos

- Como uma matriz de adjacências
- Como uma coleção de listas de adjacências

Qual a representação mais eficiente ou mais adequada?

Depende do algoritmo

Matriz de Adjacência

- Como representar grafos utilizando matrizes?

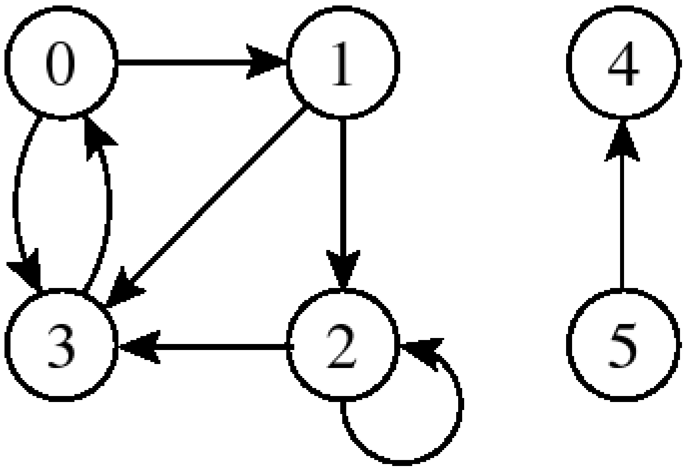
Matriz de Adjacência

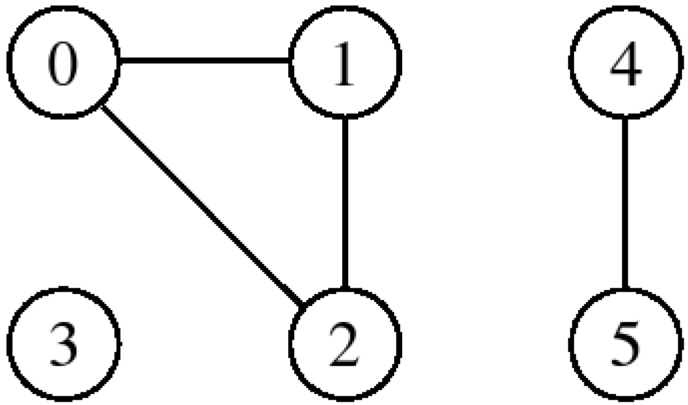
- Associar vértices à linhas e colunas da matriz e o elemento da matriz indica se há aresta

Matriz de Adjacência

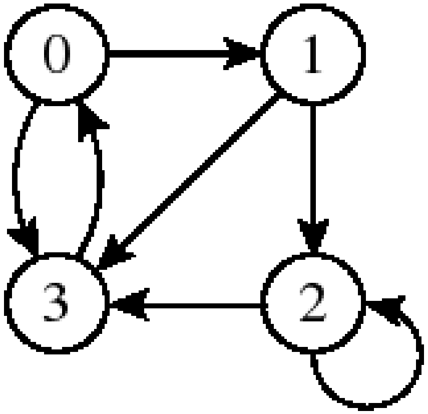
- A matriz de adjacência de um grafo $G = (V, A)$ contendo n vértices é uma matriz $n \times n$, em que:
  - A[i, j]=1 (ou verdadeiro) se e somente se existe um arco do vértice i para o vértice j .
  - A[i, j]=0 caso contrário.
- Para grafos ponderados:
  - A[i, j] contém o rótulo ou peso associado com a aresta do vértice i para o vértice j
  - Se não existir uma aresta de i para j, utilizar um valor que não possa ser usado como rótulo ou peso.

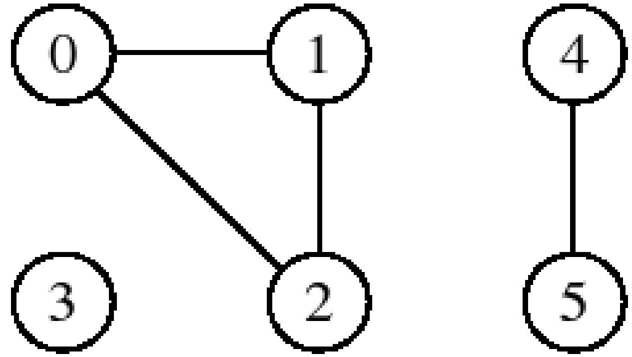
Matriz de Adjacência





Matriz de Adjacência





	0	1	2	3	4	5
0		1		1		
1			1	1		
2			1	1		
3	1					
4						1
5						

	0	1	2	3	4	5
0		1	1			
1	1		1			
2	1	1				
3						
4						1
5					1	

Matriz de Adjacência

- Considere grafos grandes e esparços
  - grande: muitos vértices
  - esparço: relativamente poucas arestas

Matriz de Adjacência

- Considere grafos grandes e esparços
  - grande: muitos vértices
  - esparço: relativamente poucas arestas

Matriz formada principalmente de zeros!

- Grande consumo de memória (desnecessário)!

Matriz de Adjacência

- Deve ser utilizada para grafos densos, em que $|A|$ é próximo de $|V|^2$.
- O tempo necessário para acessar um elemento é independente de $|V|$ ou $|A|$.
- É muito útil para algoritmos em que necessitamos saber com rapidez se existe uma aresta ligando dois vértices.
- A maior desvantagem é que a matriz necessita $\Omega(|V|^2)$ de espaço.

Como representar grafos grandes e esparços?

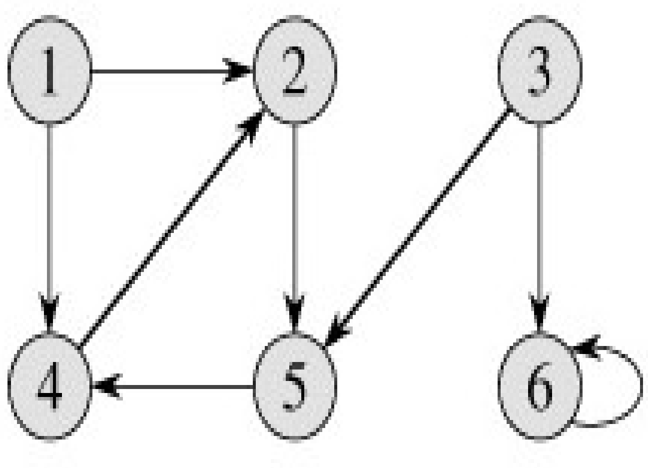
Coleção de listas de adjacências

- Associar a cada vértice uma lista de vértices adjacentes

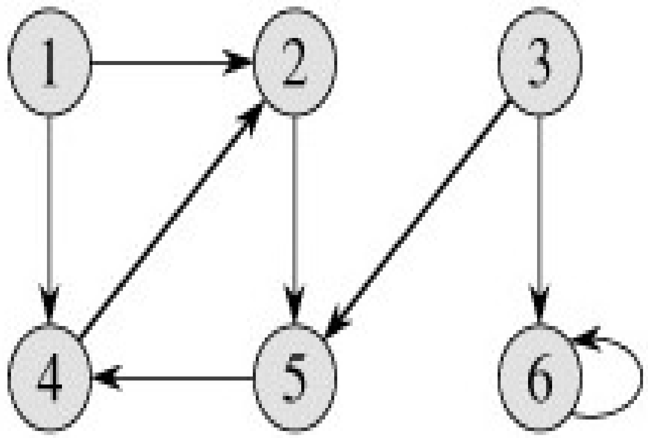
Coleção de listas de adjacências

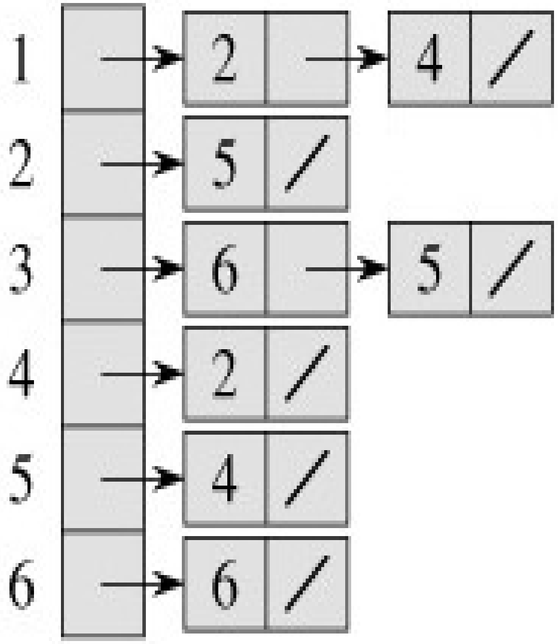
- A representação de um grafo $G = (V, A)$ usando listas de adjacências consiste de:
  - Um vetor Adj de $|V|$ listas, uma para cada vértice em V.
  - Para cada $u \in V$, a lista de adjacências Adj[u] consiste de todos os vértices adjacentes a u, i.e., todos os vértices v tais que existe uma aresta $(u,v) \in A$

Coleção de listas de adjacências

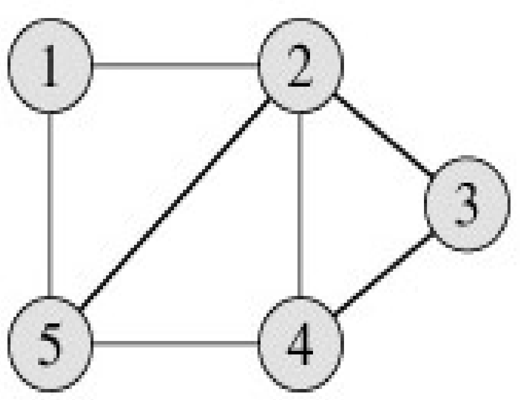


Coleção de listas de adjacências

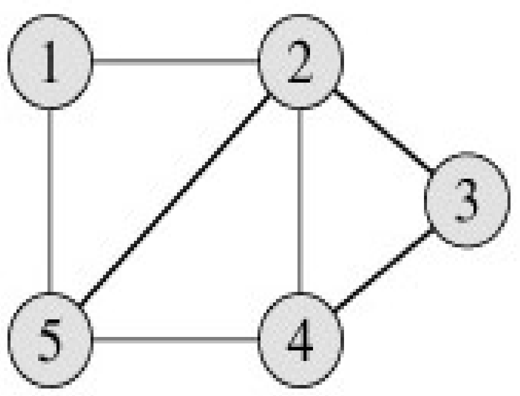


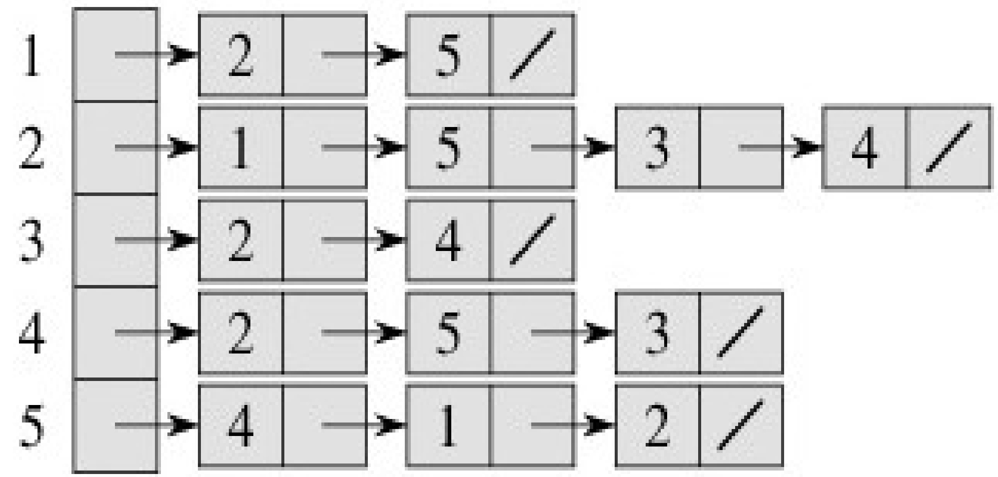


Coleção de listas de adjacências



Coleção de listas de adjacências





Coleção de listas de adjacências

- Os vértices de uma lista de adjacência são em geral armazenados em uma ordem arbitrária.
- A soma dos comprimentos de todas as listas de adjacências
  - se G é um grafo orientado é:
  - se G é um grafo não orientado é:

Coleção de listas de adjacências

- Os vértices de uma lista de adjacência são em geral armazenados em uma ordem arbitrária.
- A soma dos comprimentos de todas as listas de adjacências
  - se G é um grafo orientado é: $|A|$
  - se G é um grafo não orientado é: $2|A|$
- O espaço requerido por essa representação é:

Coleção de listas de adjacências

- Os vértices de uma lista de adjacência são em geral armazenados em uma ordem arbitrária.
- A soma dos comprimentos de todas as listas de adjacências
  - se G é um grafo orientado é: $|A|$
  - se G é um grafo não orientado é: $2|A|$
- O espaço requerido por essa representação é:

  $$O(|V| + |A|)$$

  seja o grafo orientado ou não.

Coleção de listas de adjacências

- Representação mais adequada para grafos esparsos, (|A| é muito menor do que $|V|^2$).
- É compacta e usualmente utilizada na maioria das aplicações.
- A principal desvantagem é que ela pode ter tempo O(|V|) para determinar se existe uma aresta entre o vértice i e o vértice j , pois podem existir O(|V|) vértices na lista de adjacentes do vértice i.

Busca em grafos

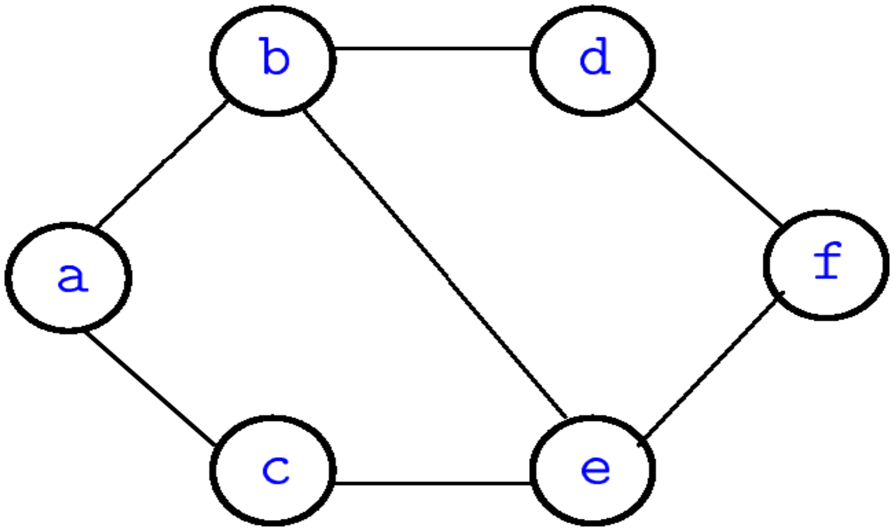
Busca em grafos

- Problema fundamental em grafos:

  Como explorar um grafo de forma sistemática?

- Muitas aplicações são abstraídas como problemas de busca
- Os algoritmos de busca em grafos são a base de vários algoritmos mais gerais em grafos.

Busca em grafos



- Como explorar o grafo?
- Por exemplo, como saber se existe caminhos simples entre dois vértices?

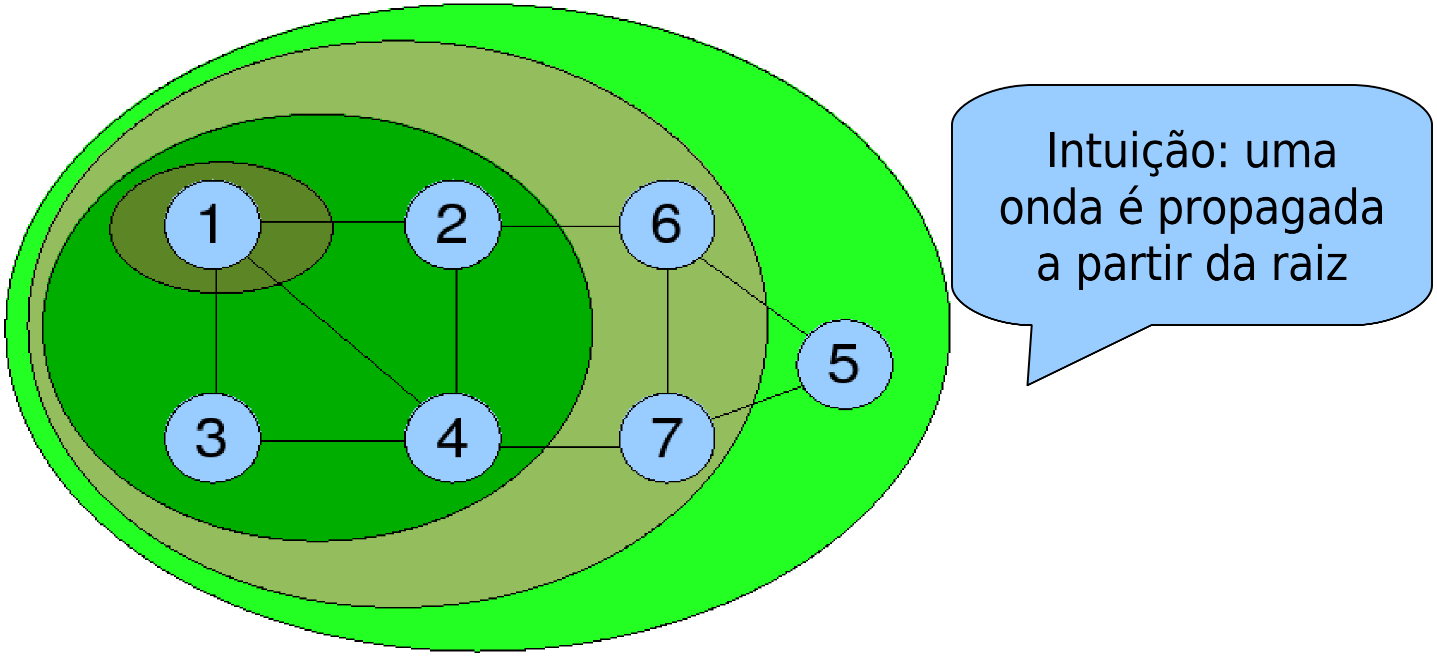
Evitar explorar vértices já explorados. Temos que marcar os vértices!

Busca em largura (BFS breadth-first-search)

- Seja $G = (V, A)$ é um vértice s de G
- BFS percorre as arestas de G descobrindo todos os vértices atingíveis a partir de s.
- BFS determina a distância (em número de arestas) de cada um desses vértices a s.

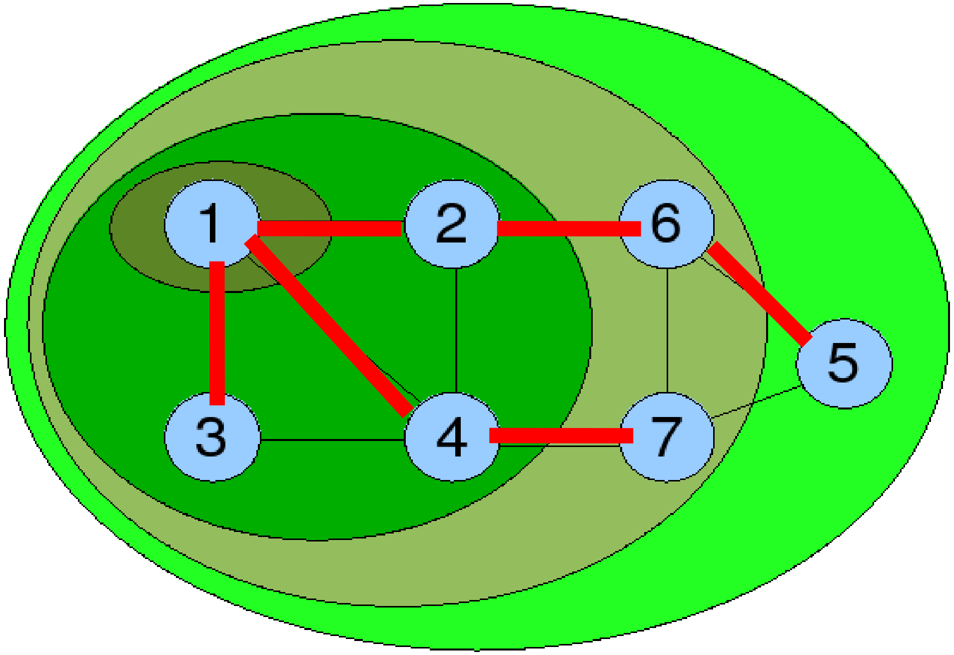
Busca em largura (BFS breadth-first-search)

- Antes de encontrar um vértice à distância k+1 de s, todos os vértices à distância k são encontrados.



Busca em largura (BFS breadth-first-search)

- BFS produz uma árvore BFS com raiz em s, que contém todos os vértices acessíveis. Determinando o caminho mínimo ou caminho mais curto (caminho que contém o número mínimo de arestas) de s a t (vértice acessível).

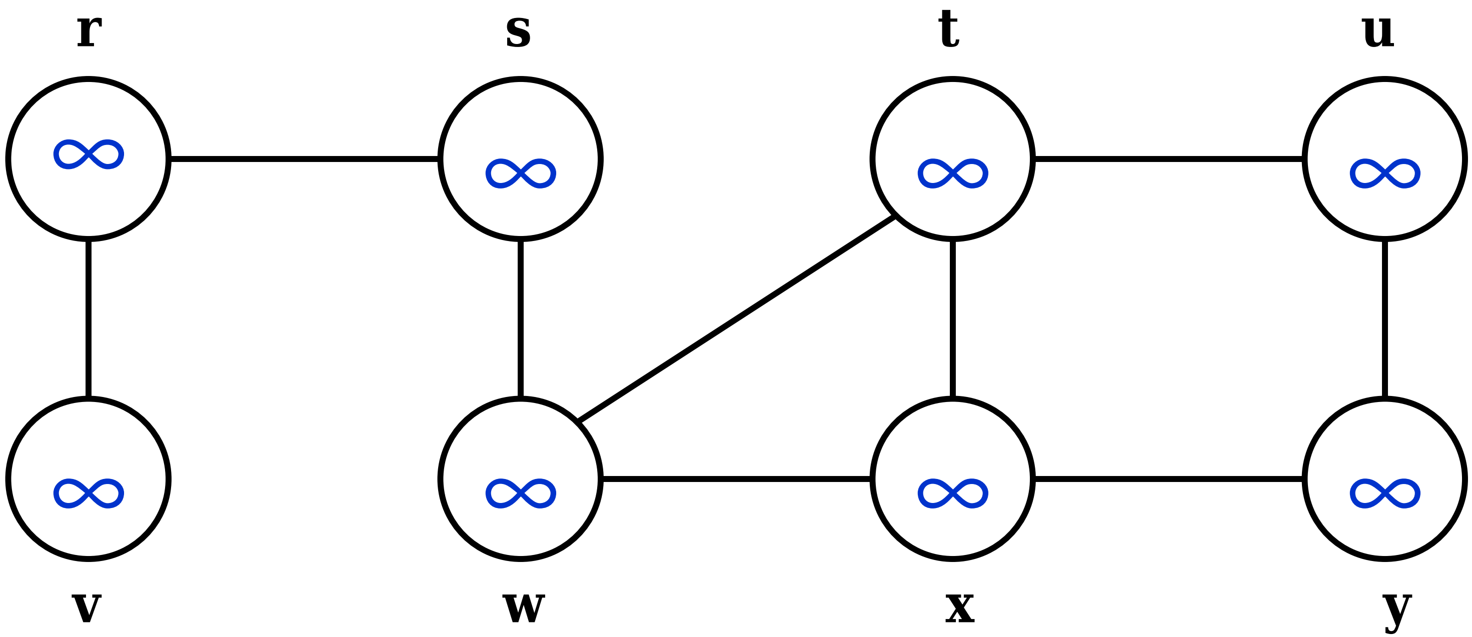


Busca em largura (BFS breadth-first-search)

- Para organizar o processo de busca os vértices são pintados:
  - branco: não foram descobertos ainda
  - cinzas: são a fronteira. O vértice já foi descoberto mais ainda não examinamos os seus vizinhos.
  - pretos: são os vértices já descobertos e seus vizinhos já foram examinados.
- É utilizada uma fila para manter os vértices cinzas

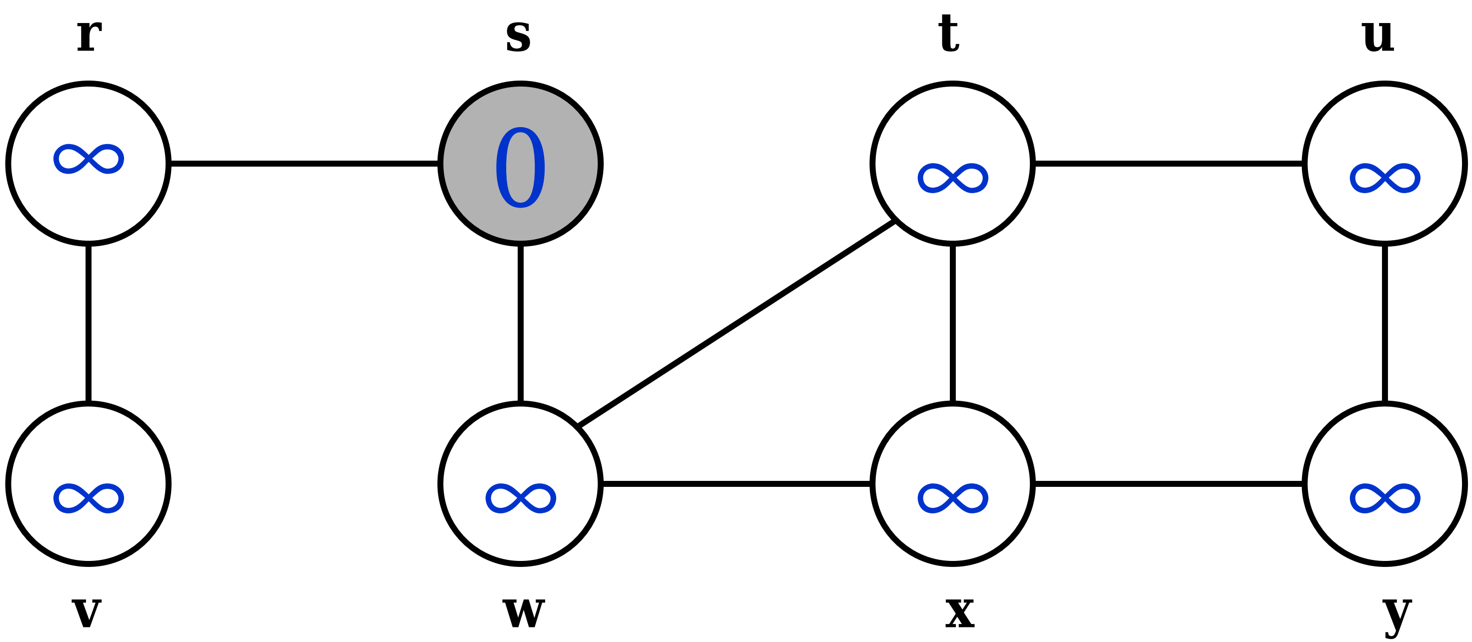
Busca em largura (BFS breadth-first-search)

No início todos os vértices são brancos e a distância é infinita



Busca em largura (BFS breadth-first-search)

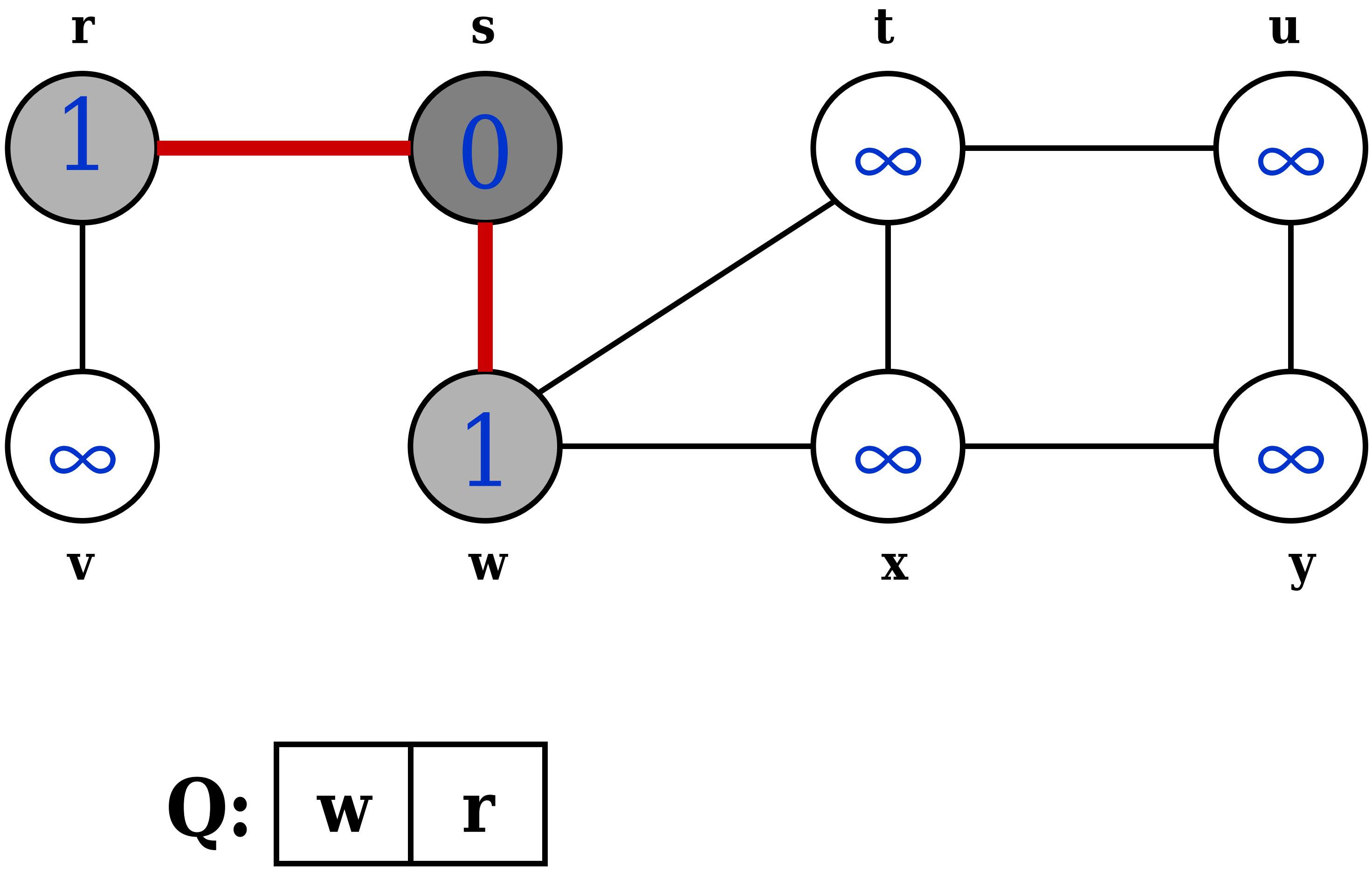
O vértice origem é pintado de cinza (ele é considerado descoberto) e é colocado na fila



Q: s

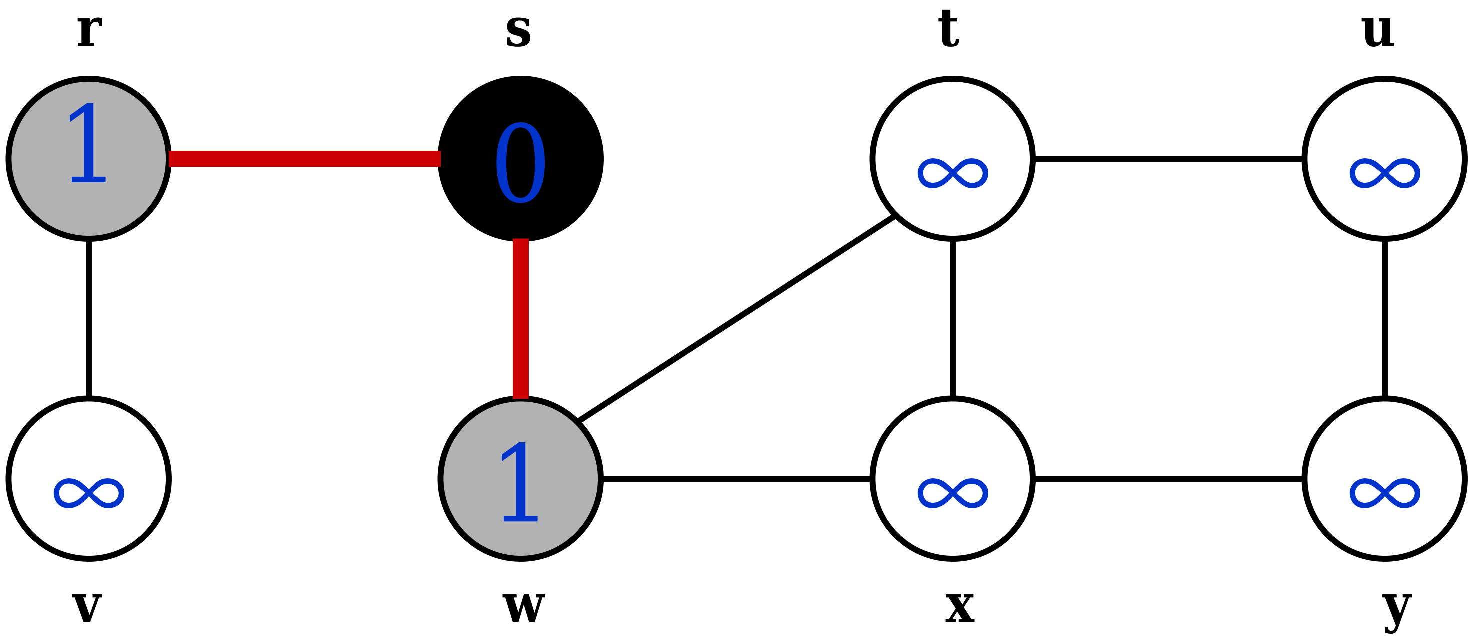
Busca em largura (BFS breadth-first-search)

É retirado o primeiro elemento da fila (s) e os adjacentes a ele saõ colocados em Q, pintados de cinza. Além disso é atualizada a distância e o pai



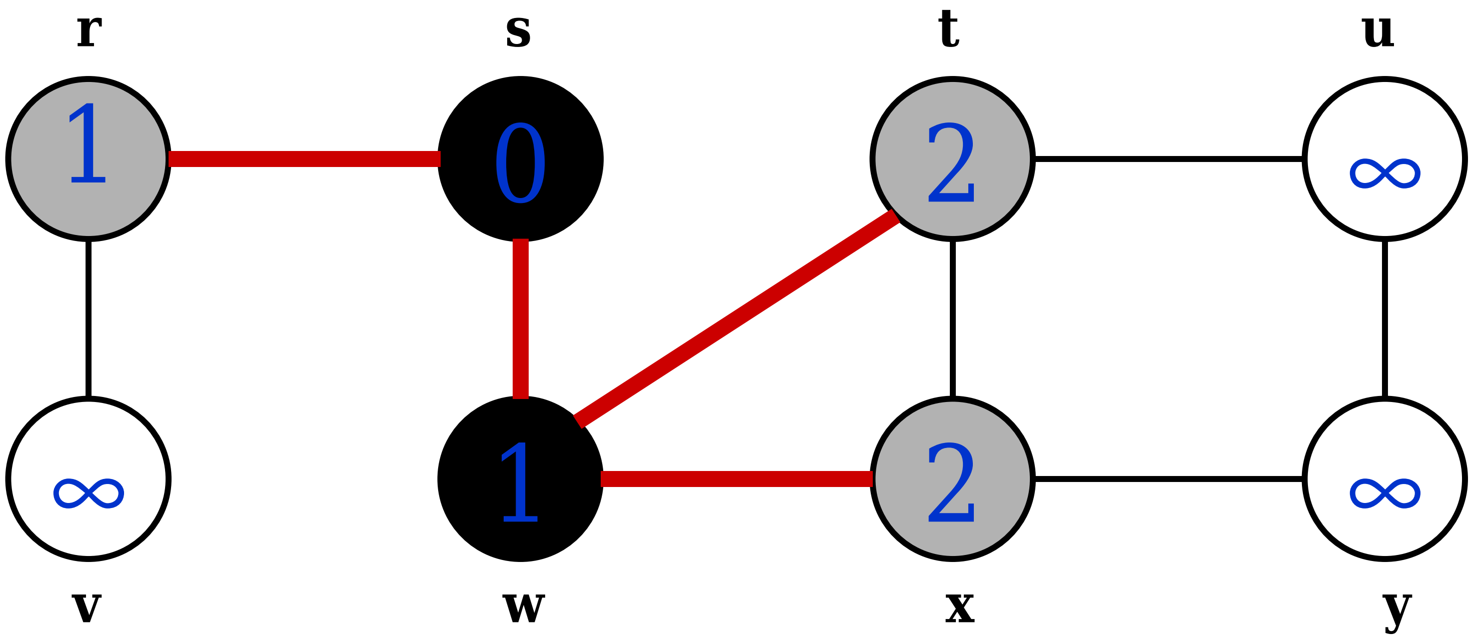
Busca em largura (BFS breadth-first-search)

colorimos o vértice com preto (os seus vizinhos já foram descobertos)



Q: | w | r |

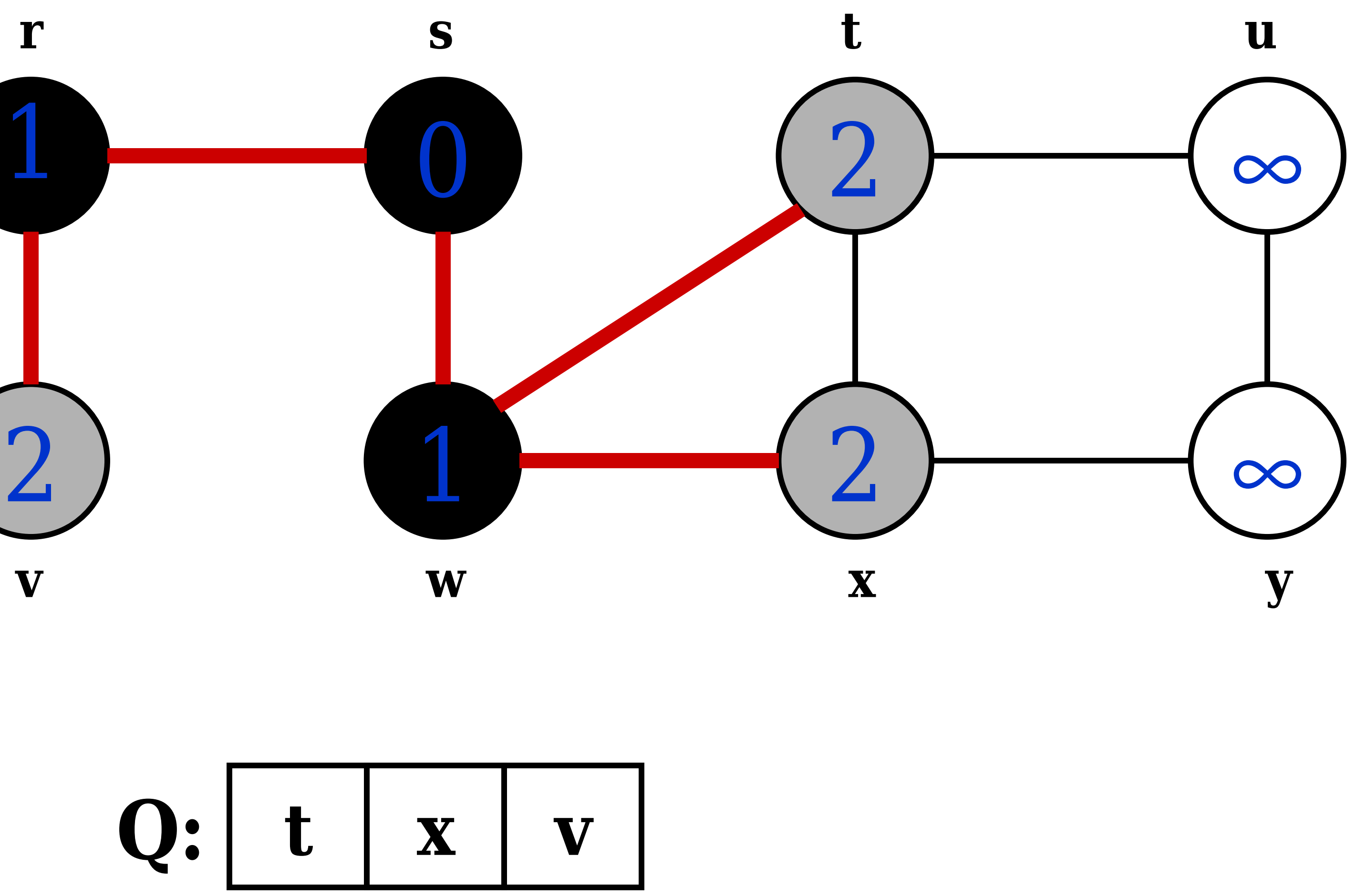
Busca em largura (BFS breadth-first-search)



Q:

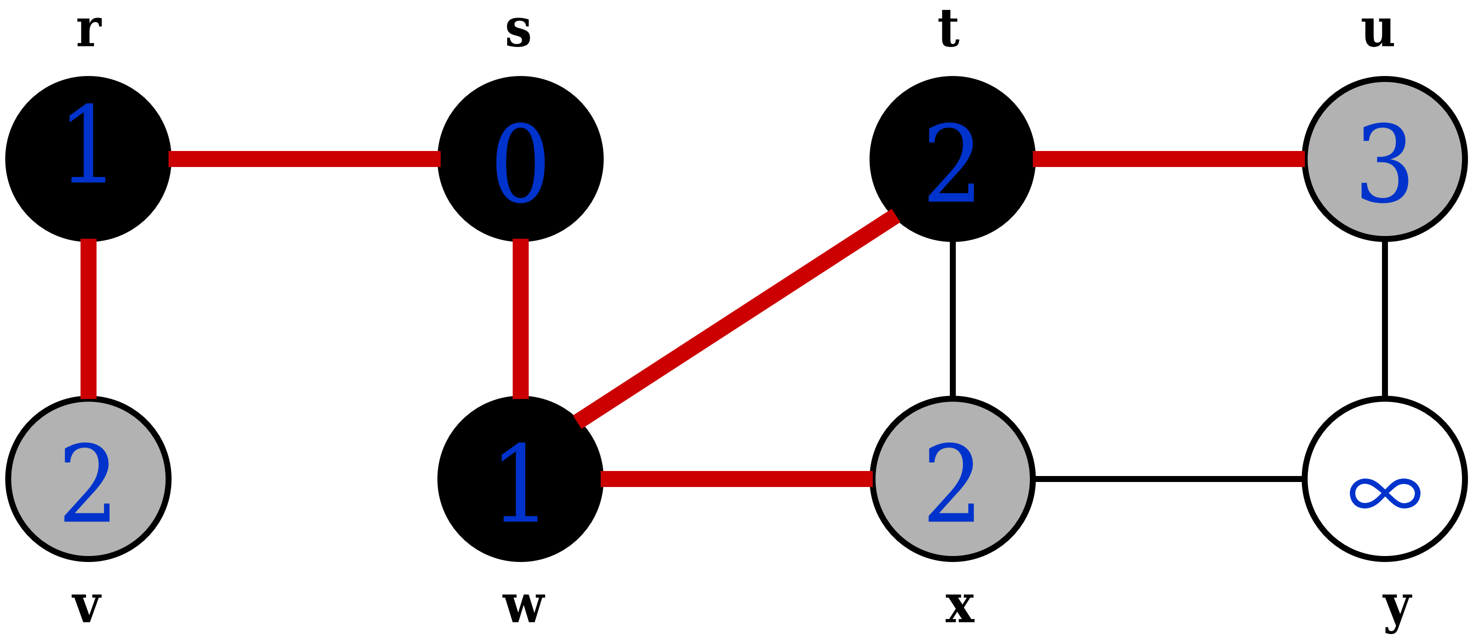
r	**t**	**x**

Busca em largura (BFS breadth-first-search)



Busca em largura (BFS breadth-first-search)

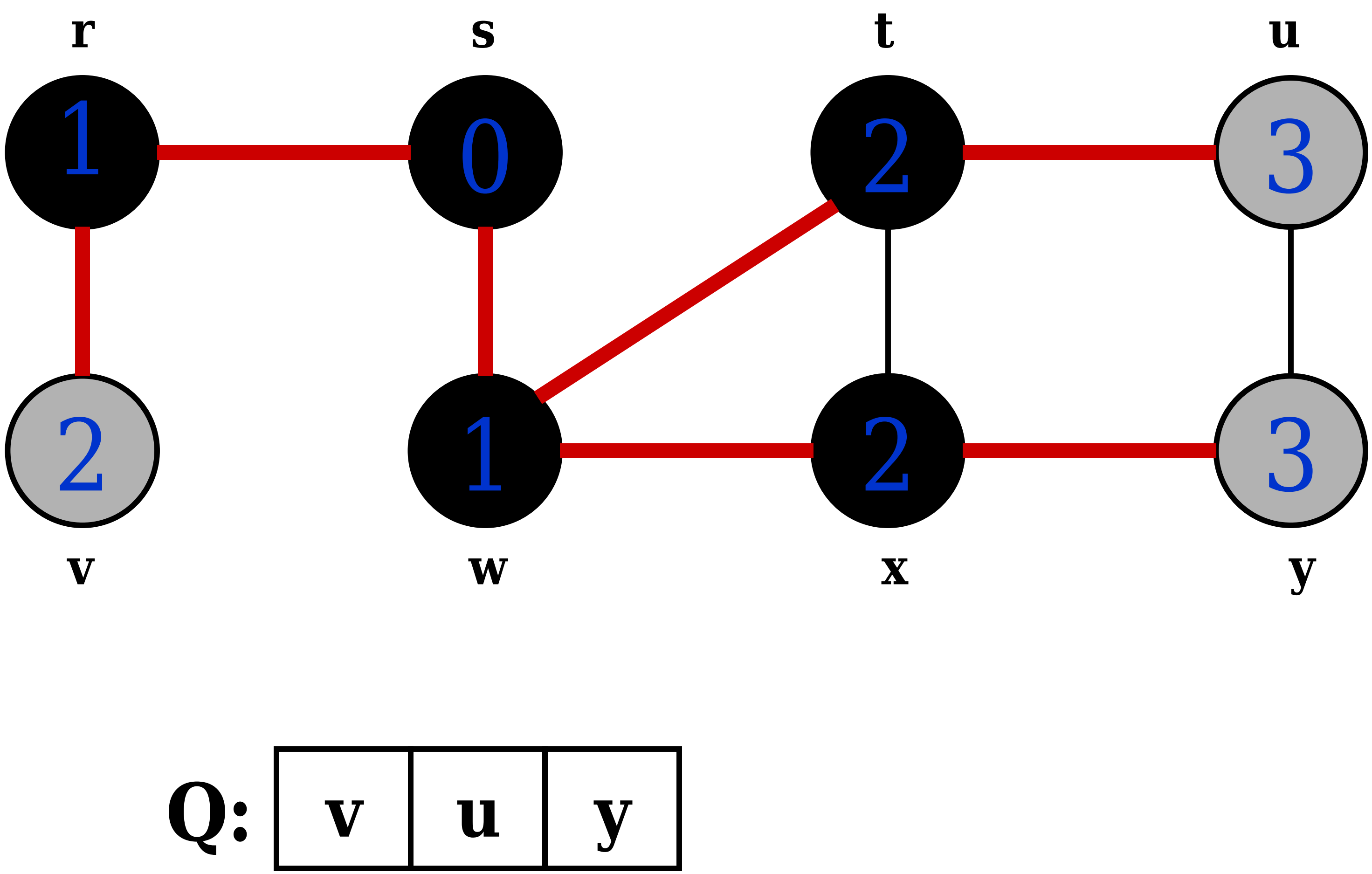
Observe que somente enfileramos vértices brancos, que são imediatamente coloridos de cinza ao entrar na fila



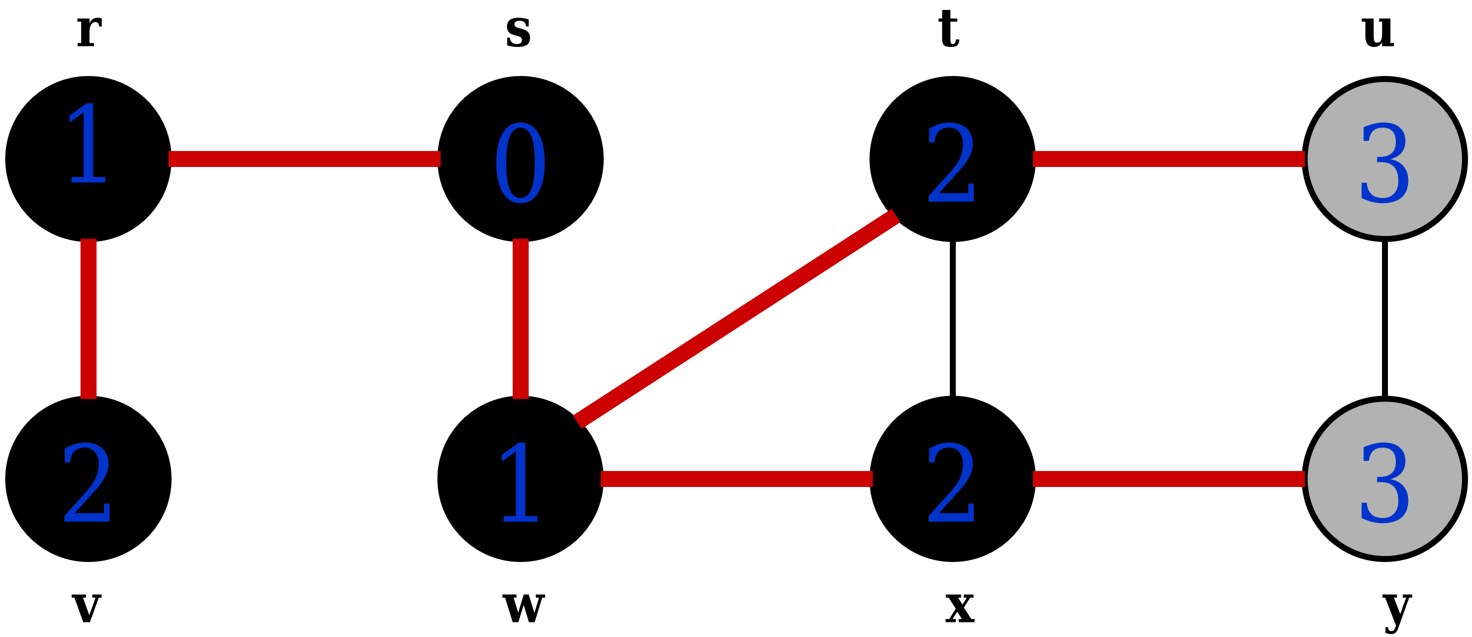
Q:

x	v	u

Busca em largura (BFS breadth-first-search)



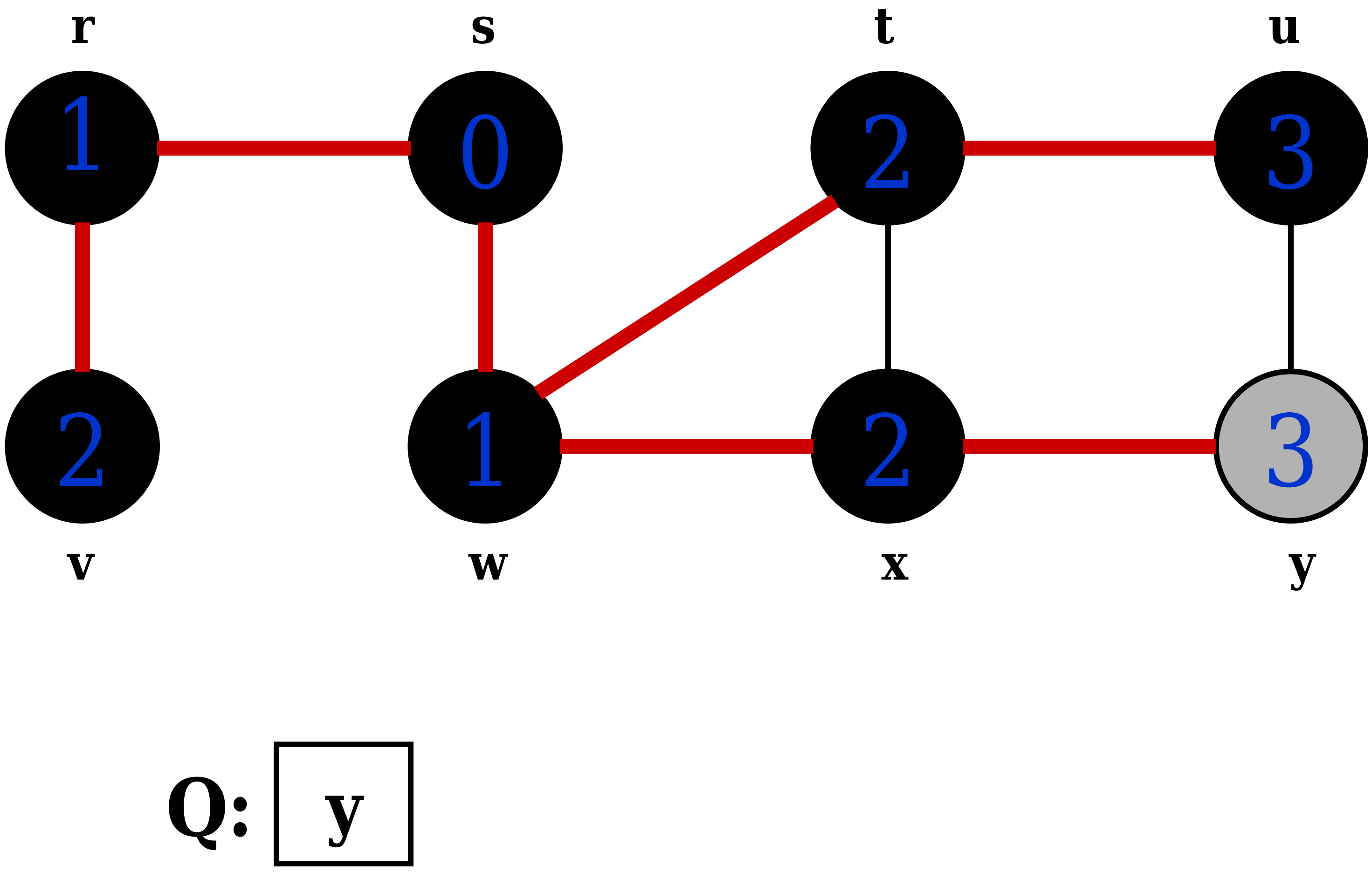
Busca em largura (BFS breadth-first-search)



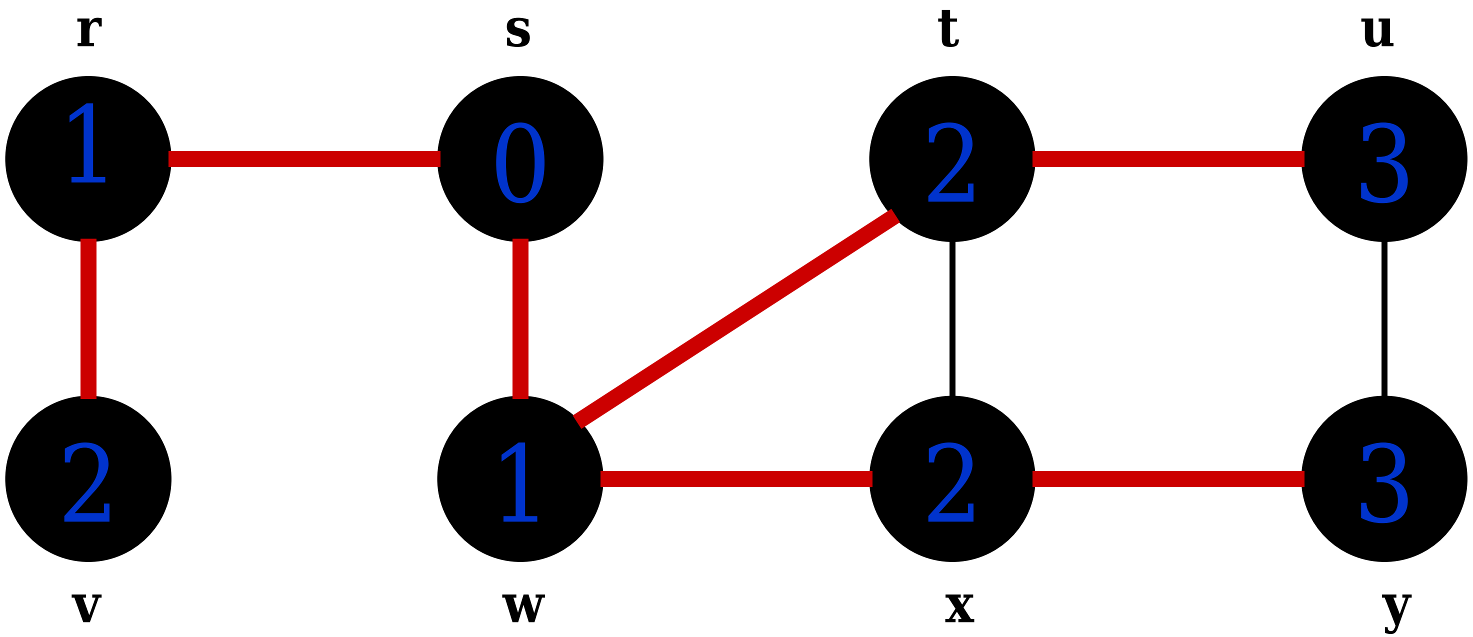
Q:

u	y

Busca em largura (BFS breadth-first-search)

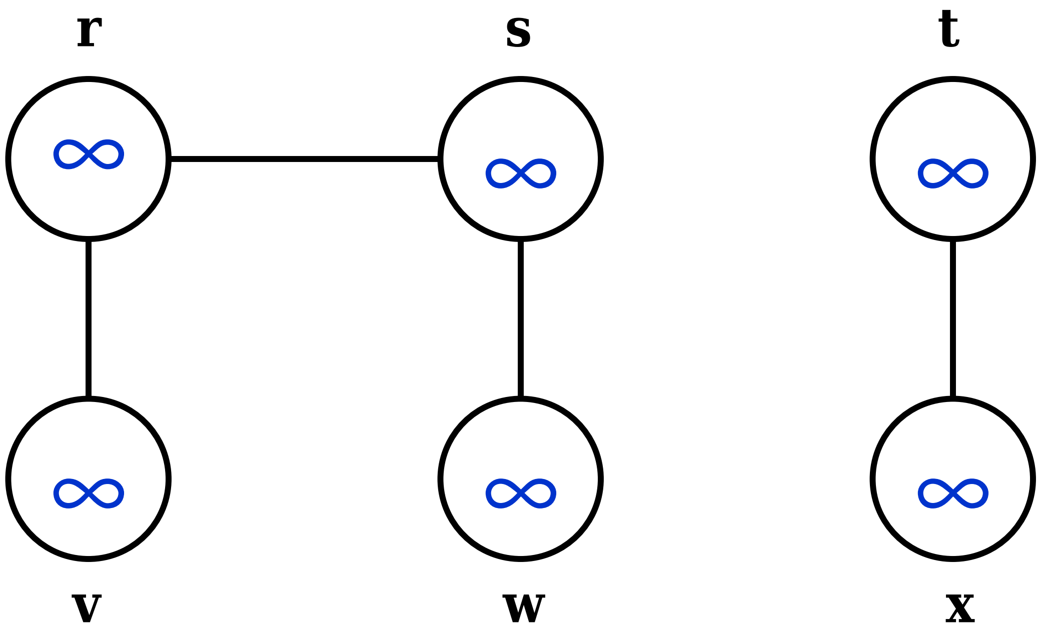


Busca em largura (BFS breadth-first-search)



Q: Ø

Busca em largura (BFS breadth-first-search)



Aplicar o algoritmo BFS,
considerar s como vértice origem

Busca em largura (BFS breadth-first-search

– Escrever o pseudocódigo de BFS

Busca em largura (BFS breadth-first-search

```
BFS(V, A, s)
    for each u in V − {s}          ▷ para cada vértice u em V exceto s
        color[u] ← WHITE           ▷ no início todos os vértices são brancos
        d[u] ← infinity
        π[u] ← NIL
    color[s] ← GRAY
    d[s] ← 0
    π[s] ← NIL
    Q ← {}
    ENQUEUE(Q, s)
    while Q is non-empty
        u ← DEQUEUE(Q)
        for each v adjacent to u
            if color[v] = WHITE
                then  color[v] ← GRAY
                      d[v] ← d[u] + 1
                      π[v] ← u
                      ENQUEUE(Q, v)
     color[u] ← BLACK
```

Busca em largura (BFS breadth-first-search

```
BFS(V, A, s)
    for each u in V − {s}              ▷ para cada vértice u em V exceto s
        color[u] ← WHITE               ▷ no início todos os vértices são brancos
        d[u] ← infinity
        π[u] ← NIL
    color[s] ← GRAY                    ▷ Vértice origem descoberto
    d[s] ← 0
    π[s] ← NIL
    Q ← {}
    ENQUEUE(Q, s)                      ▷ Colocar o vértice origem na fila Q
    while Q is non-empty
        u ← DEQUEUE(Q)
        for each v adjacent to u
            if color[v] = WHITE
                then  color[v] ← GRAY
                      d[v] ← d[u] + 1
                      π[v] ← u
                      ENQUEUE(Q, v)
      color[u] ← BLACK
```

Busca em largura (BFS breadth-first-search

```
BFS(V, A, s)
    for each u in V − {s}              ▷ para cada vértice u em V exceto s
        color[u] ← WHITE               ▷ no início todos os vértices são brancos
        d[u] ← infinity
        π[u] ← NIL
    color[s] ← GRAY                    ▷ Vértice origem descoberto
    d[s] ← 0
    π[s] ← NIL
    Q ← {}
    ENQUEUE(Q, s)                      ▷ Colocar o vértice origem na fila Q
    while Q is non-empty               ▷ Enquanto existam vértices cinzas
        u ← DEQUEUE(Q)                 ▷ i.e., u = primeiro(Q)
        for each v adjacent to u
            if color[v] = WHITE
                then  color[v] ← GRAY
                      d[v] ← d[u] + 1
                      π[v] ← u
                      ENQUEUE(Q, v)
        color[u] ← BLACK
```

Busca em largura (BFS breadth-first-search

```
BFS(V, A, s)
      for each u in V − {s}            ▷ para cada vértice u em V exceto s
            color[u] ← WHITE           ▷ no início todos os vértices são brancos
            d[u] ← infinity
            π[u] ← NIL
      color[s] ← GRAY                  ▷ Vértice origem descoberto
      d[s] ← 0
      π[s] ← NIL
      Q ← {}
      ENQUEUE(Q, s)                    ▷ Colocar o vértice origem na fila Q
      while Q is non-empty             ▷ Enquanto existam vértices cinzas
            u ← DEQUEUE(Q)                      ▷ i.e., u = primeiro(Q)
            for each v adjacent to u            ▷ para cada vértice adjacente a u
                  if color[v] = WHITE
                      then  color[v] ← GRAY
                            d[v] ← d[u] + 1
                            π[v] ← u
                            ENQUEUE(Q, v)
        color[u] ← BLACK
```

Busca em largura (BFS breadth-first-search

```
BFS(V, A, s)
    for each u in V − {s}              ▷ para cada vértice u em V exceto s
        color[u] ← WHITE               ▷ no início todos os vértices são brancos
        d[u] ← infinity
        π[u] ← NIL
    color[s] ← GRAY                    ▷ Vértice origem descoberto
    d[s] ← 0
    π[s] ← NIL
    Q ← {}
    ENQUEUE(Q, s)                      ▷ Colocar o vértice origem na fila Q
    while Q is non-empty               ▷ Enquanto existam vértices cinzas
        u ← DEQUEUE(Q)                 ▷ i.e., u = primeiro(Q)
        for each v adjacent to u       ▷ para cada vértice adjacente a u
            if color[v] = WHITE        ▷ se é branco ele ainda não foi descoberto
                then color[v] ← GRAY
                     d[v] ← d[u] + 1
                     π[v] ← u
                     ENQUEUE(Q, v)
        color[u] ← BLACK
```

Busca em largura (BFS breadth-first-search

```
BFS(V, A, s)
    for each u in V − {s}            ▷ para cada vértice u em V exceto s
        color[u] ← WHITE             ▷ no início todos os vértices são brancos
        d[u] ← infinity
        π[u] ← NIL
    color[s] ← GRAY                  ▷ Vértice origem descoberto
    d[s] ← 0
    π[s] ← NIL
    Q ← {}
    ENQUEUE(Q, s)                    ▷ Colocar o vértice origem na fila Q
    while Q is non-empty             ▷ Enquanto existam vértices cinzas
        u ← DEQUEUE(Q)               ▷ i.e., u = primeiro(Q)
        for each v adjacent to u     ▷ para cada vértice adjacente a u
            if color[v] = WHITE      ▷ se é branco ele ainda não foi descoberto
                then  color[v] ← GRAY
                      d[v] ← d[u] + 1
                      π[v] ← u       ▷ pai de v é o nó que levou a descoberta de v
                      ENQUEUE(Q, v)
        color[u] ← BLACK             ▷ os vizinhos de u já foram examinados
```

Busca em largura (BFS breadth-first-search

```
BFS(V, A, s)
    for each u in V − {s}                ▷ para cada vértice u em V exceto s
        color[u] ← WHITE                 ▷ no início todos os vértices são brancos
        d[u] ← infinity
        π[u] ← NIL
    color[s] ← GRAY                      ▷ Vértice origem descoberto
    d[s] ← 0
    π[s] ← NIL
    Q ← {}
    ENQUEUE(Q, s)                        ▷ Colocar
    while Q is non-empty                 ▷ Enqua
        u ← DEQUEUE(Q)                   ▷
        for each v adjacent to u         ▷
            if color[v] = WHITE          ▷                                to
                then  color[v] ← GRAY
                      d[v] ← d[u] + 1
                      π[v] ← u
                      ENQUEUE(Q, v)
        color[u] ← BLACK                 ▷ os vizinhos de u já foram examinados
```

Cada vértice é colocado na fila no máximo uma vez e portanto retirado da fila no máximo uma vez

Busca em largura (BFS breadth-first-search

```
BFS(V, A, s)
    for each u in V − {s}             ▷ para cada vértice u em V exceto s
        color[u] ← WHITE              ▷ no início todos os vértices são brancos
        d[u] ← infinity
        π[u] ← NIL
    color[s] ← GRAY                   ▷ Vértice origem descoberto
    d[s] ← 0
    π[s] ← NIL
    Q ← {}
    ENQUEUE(Q, s)                     ▷ Colocar
    while Q is non-empty              ▷ Enqua
        u ← DEQUEUE(Q)                ▷
        for each v adjacent to u      ▷
            if color[v] = WHITE       ▷                                   to
                then color[v] ← GRAY
                     d[v] ← d[u] + 1
                     π[v] ← u
                     ENQUEUE(Q, v)
        color[u] ← BLACK              ▷ os vizinhos de u já foram examinados
```

As operações DEQUEUE e ENQUEUE demoram tempo O(1). Assim o tempo total de operações na fila é:

Busca em largura (BFS breadth-first-search

```
BFS(V, A, s)
    for each u in V − {s}          ▷ para cada vértice u em V exceto s
        color[u] ← WHITE           ▷ no início todos os vértices são brancos
        d[u] ← infinity
        π[u] ← NIL
    color[s] ← GRAY                ▷ Vértice origem descoberto
    d[s] ← 0
    π[s] ← NIL
    Q ← {}
    ENQUEUE(Q, s)                  ▷ Colocar
    while Q is non-empty           ▷ Enqua
        u ← DEQUEUE(Q)             ▷
        for each v adjacent to u   ▷
            if color[v] = WHITE    ▷                    to
                then  color[v] ← GRAY
                      d[v] ← d[u] + 1
                      π[v] ← u
                      ENQUEUE(Q, v)
      color[u] ← BLACK             ▷ os vizinhos de u já foram examinados
```

As operações DEQUEUE e ENQUEUE demoram tempo O(1). Assim o tempo total de operações na fila é: O(V)

Busca em largura (BFS breadth-first-search

```
BFS(V, A, s)
    for each u in V − {s}              ▷ para cada vértice u em V exceto s
        color[u] ← WHITE               ▷ no início todos os vértices são brancos
        d[u] ← infinity
        π[u] ← NIL
    color[s] ← GRAY                    ▷ Vértice origem descoberto
    d[s] ← 0
    π[s] ← NIL
    Q ← {}
    ENQUEUE(Q, s)                      ▷ Coloca
    while Q is non-empty               ▷ Enqua
        u ← DEQUEUE(Q)
        for each v adjacent to u
            if color[v] = WHITE        ▷
                then color[v] ← GRAY
                     d[v] ← d[u] + 1
                     π[v] ← u
                     ENQUEUE(Q, v)
     color[u] ← BLACK                  ▷ os vizinhos de u já foram examinados
```

A lista de adjacências de cada vértice é examinado somente quando o vértice é desenfileirado, a lista de adjacências de cada vértice é examinada no máximo uma vez

Busca em largura (BFS breadth-first-search

```
BFS(V, A, s)
    for each u in V − {s}              ▷ para cada vértice u em V exceto s
        color[u] ← WHITE               ▷ no início todos os vértices são brancos
        d[u] ← infinity
        π[u] ← NIL
    color[s] ← GRAY                    ▷ Vértice origem descoberto
    d[s] ← 0
    π[s] ← NIL
    Q ← {}
    ENQUEUE(Q, s)                      ▷ Coloca
    while Q is non-empty               ▷ Enqua
        u ← DEQUEUE(Q)                     ▷
        for each v adjacent to u
            if color[v] = WHITE            ▷                              to
                then  color[v] ← GRAY
                      d[v] ← d[u] + 1
                      π[v] ← u
                      ENQUEUE(Q, v)
     color[u] ← BLACK                  ▷ os vizinhos de u já foram examinados
```

Assim, o tempo gasto na varredura total das listas de adjacências é:

Busca em largura (BFS breadth-first-search

```
BFS(V, A, s)
    for each u in V − {s}           ▷ para cada vértice u em V exceto s
        color[u] ← WHITE            ▷ no início todos os vértices são brancos
        d[u] ← infinity
        π[u] ← NIL
    color[s] ← GRAY                 ▷ Vértice origem descoberto
    d[s] ← 0
    π[s] ← NIL
    Q ← {}
    ENQUEUE(Q, s)                   ▷ Coloca
    while Q is non-empty            ▷ Enqua
        u ← DEQUEUE(Q)                  ▷
        for each v adjacent to u
            if color[v] = WHITE         ▷                                          to
                then  color[v] ← GRAY
                      d[v] ← d[u] + 1
                      π[v] ← u
                      ENQUEUE(Q, v)
      color[u] ← BLACK                  ▷ os vizinhos de u já foram examinados
```

Assim, o tempo gasto na varredura total das listas de adjacências é: O(A)

Busca em largura (BFS breadth-first-search

```
BFS(V, A, s)
for each u in V − {s}                ▷ para cada vértice u em V exceto s
        color[u] ← WHITE             ▷ no iníci
        d[u] ← infinity
        π[u] ← NIL
color[s] ← GRAY                      ▷ Vértice
d[s] ← 0
π[s] ← NIL
Q ← {}
ENQUEUE(Q, s)                        ▷ Colocar o vértice origem na fila Q
while Q is non-empty                 ▷ Enquanto existam vértices cinzas
        u ← DEQUEUE(Q)                       ▷ i.e., u = pop(Q)
        for each v adjacent to u             ▷ para cada vértice adjacente a u
                if color[v] = WHITE          ▷ se é branco ele ainda não foi descoberto
                    then  color[v] ← GRAY
                          d[v] ← d[u] + 1
                          π[v] ← u
                          ENQUEUE(Q, v)
    color[u] ← BLACK                         ▷ os vizinhos de u já foram examinados
```

A parte de inicialização é:

Busca em largura (BFS breadth-first-search

```
BFS(V, A, s)
    for each u in V − {s}              ▷ para cada vértice u em V exceto s
        color[u] ← WHITE               ▷ o iníci
        d[u] ← infinity
        π[u] ← NIL
    color[s] ← GRAY                    ▷ Vértice
    d[s] ← 0
    π[s] ← NIL
    Q ← {}
    ENQUEUE(Q, s)                      ▷ Colocar o vértice origem na fila Q
    while Q is non-empty               ▷ Enquanto existam vértices cinzas
        u ← DEQUEUE(Q)                 ▷ i.e., u = pop(Q)
        for each v adjacent to u       ▷ para cada vértice adjacente a u
            if color[v] = WHITE        ▷ se é branco ele ainda não foi descoberto
                then color[v] ← GRAY
                     d[v] ← d[u] + 1
                     π[v] ← u
                     ENQUEUE(Q, v)
     color[u] ← BLACK                  ▷ os vizinhos de u já foram examinados
```

A parte de inicialização é O(V)

Busca em largura (BFS breadth-first-search

O tempo total da busca em largura é $O(V+A)$

Busca em largura (BFS breadth-first-search

O tempo total da busca em largura é $O(V+A)$

Se é um grafo completo qual o tempo total da busca em largura?

Caminho entre dois vértices

- Problema: Como saber se existe caminho entre dois vértices?

Exemplo: existe um caminho entre São Luis e Limeira?

Caminho entre dois vértices

- Problema: Como saber se existe caminho entre dois vértices?

Exemplo: existe um caminho entre São Luis e Limeira?

- Solução: usar BFS

Caminho entre dois vértices

- Problema: Como saber se existe caminho entre dois vértices?

Exemplo: existe um caminho entre São Luis e Limeira?

- Solução: usar BFS
  - Marcar São Luis como raiz
  - Realizar BFS
  - Ao terminar BFS se Limeira tiver distância diferente de infinito ou se o vértice está pintado de preto, então há caminho, caso contrário, não há.

Caminho entre dois vértices

- Problema: Como saber se existe caminho entre dois vértices?

Exemplo: existe um caminho entre São Luis e Limeira?

- Solução: usar BFS
  - Marcar São Luis como raiz
  - Realizar BFS

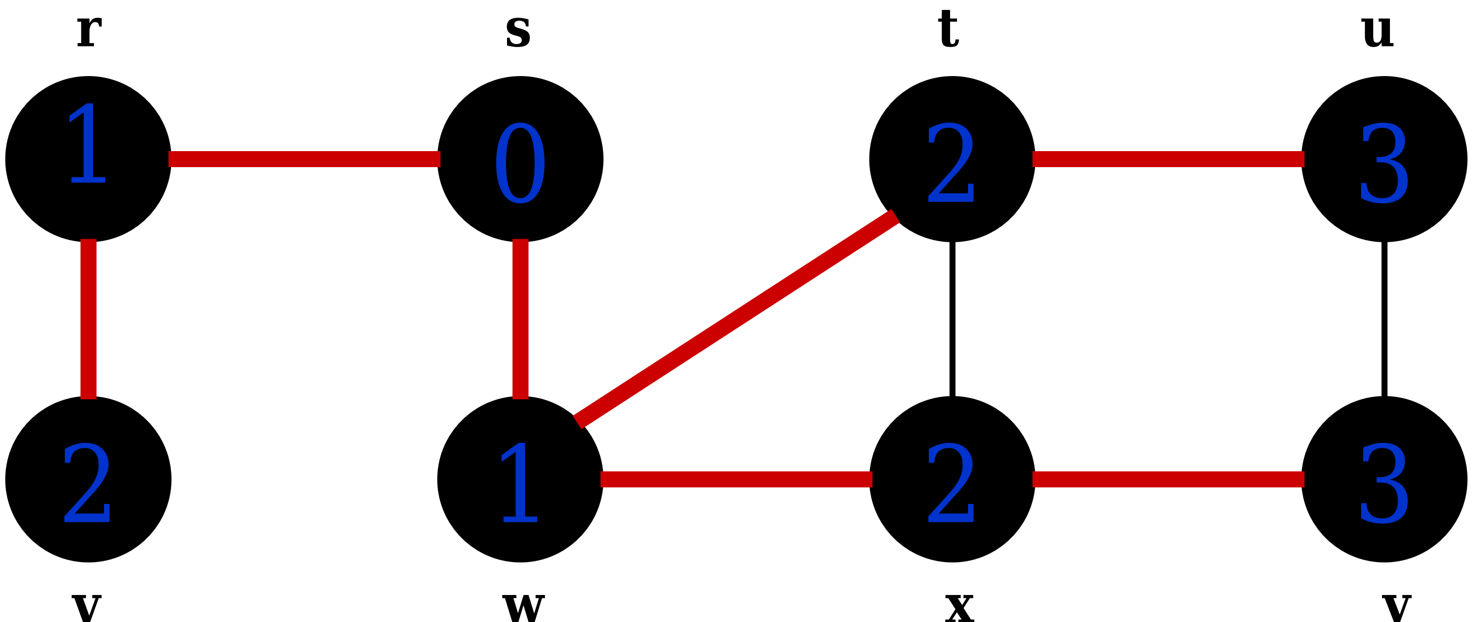
É necessário realizar o BFS completo?

Caminho entre dois vértices

```
BFS(V, A, s)
1.      for each u in V − {s}              ▷ para cada vértice u em V exceto s
2.          color[u] ← WHITE               ▷ no início todos os vértices são brancos
3.          d[u] ← infinity
4.          π[u] ← NIL
5.      color[s] ← GRAY                    ▷ Vértice origem descoberto
6.      d[s] ← 0
7.      π[s] ← NIL
8.      Q ← {}
9.      ENQUEUE(Q, s)                      ▷ C
10      while Q is non-empty               ▷ E
11.         u ← DEQUEUE(Q)                      ▷ i.e., u ← primeiro(Q)
12.         for each v adjacent to u            ▷ para cada vértice adjacente a u
13.             if color[v] = WHITE            ▷ se é branco ele ainda não foi descoberto
14.                 then  color[v] ← GRAY
15.                       d[v] ← d[u] + 1
16.                       π[v] ← u             ▷ pai de v é o nó que levou a descoberta de v
17.                       ENQUEUE(Q, v)
18.         color[u] ← BLACK                   ▷ os vizinhos de u já foram examinados
```

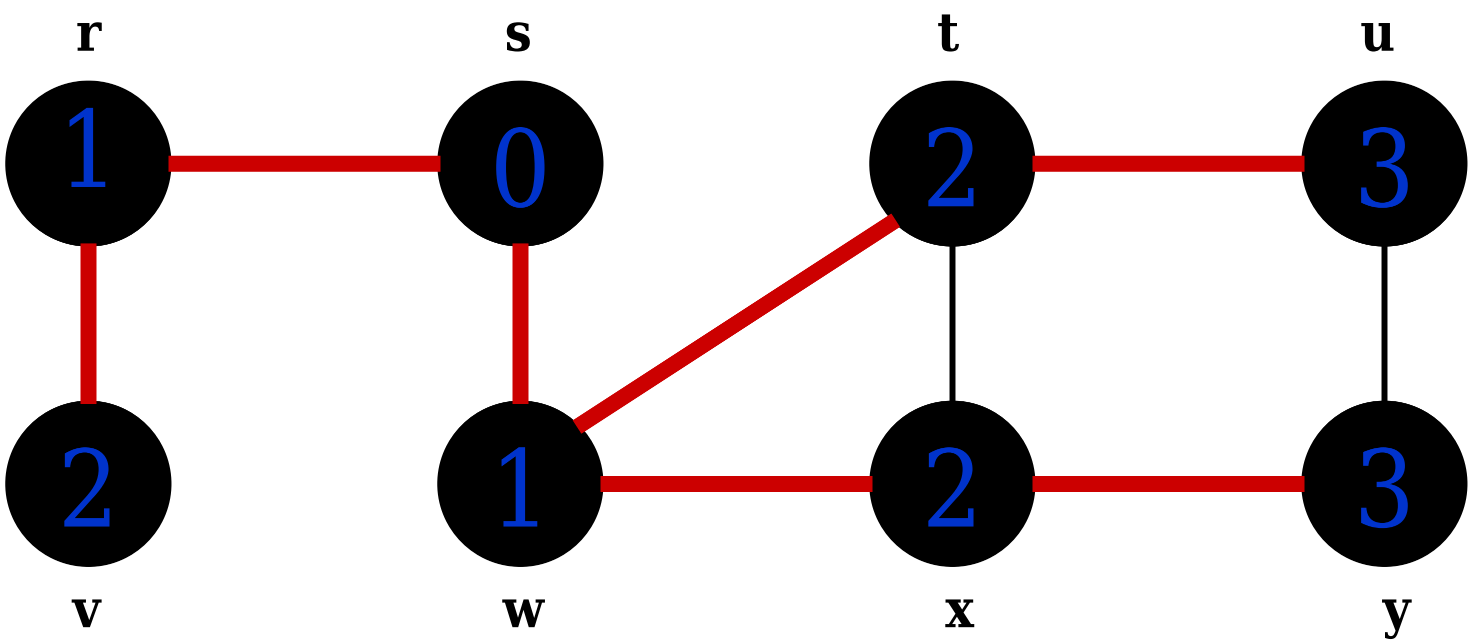
Que linhas mudar?

Caminhos mais curto



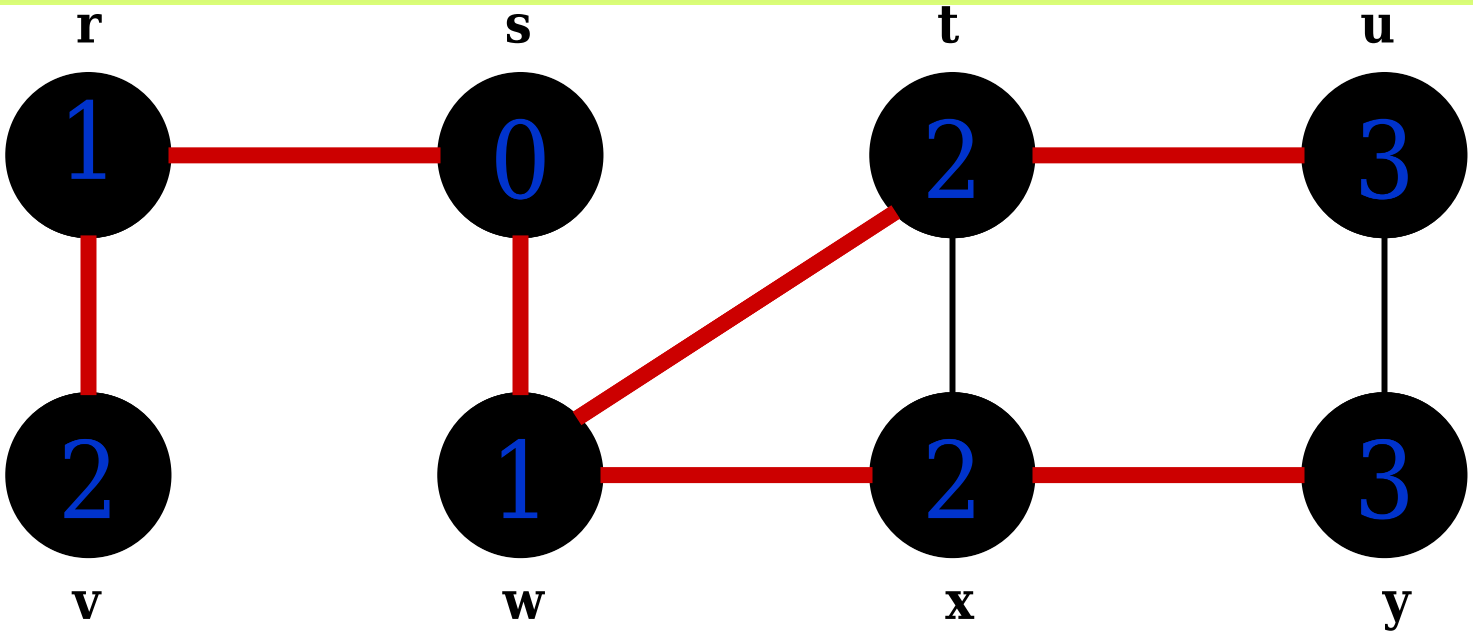
- BFS produz uma árvore BFS com raiz em s e define o caminho mais curto
- Qual é o caminho mais curto entre u e s?
- Qual é o caminho mais curto entre u e y?

Caminhos mais curto



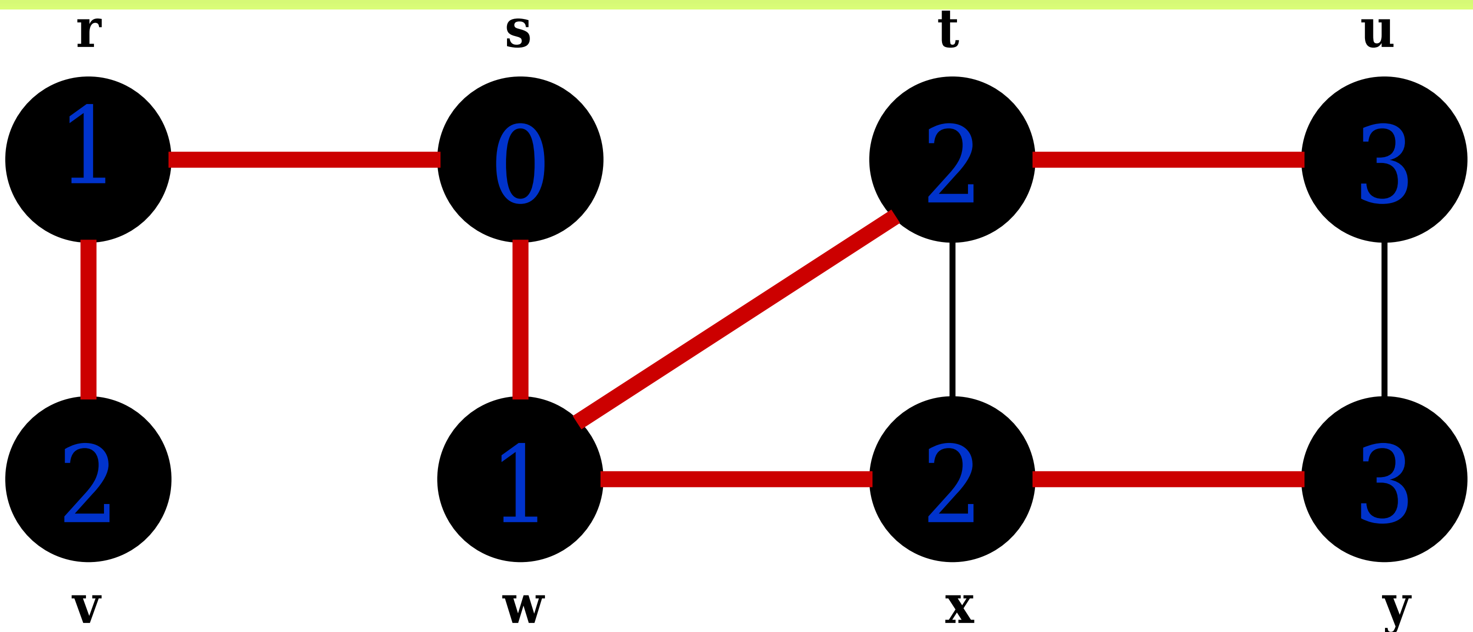
- BFS produz uma árvore BFS com raiz em s e define o caminho mais curto
- Qual é o caminho mais curto entre u e s?
- Qual é o caminho mais curto entre u e y?
- A árvore define o caminho mais curto para a raiz!!!!

Caminhos mais curto



- Suponha que já executamos a busca BFS para calcular o caminho mais curto entre s e os vértice acessíveis a partir da raiz s
- Elabore um algoritmo para imprimir o caminho mais curto entre s e um vértice v qualquer

Caminhos mais curto



– Suponha que já executar... caminho mais curto entr... tir da raiz s

Podemos usar recursão.
Print-Path(G, s, v)
Caso Base: ?

– Elabore um algoritmo para imprimir o caminho mais curto entre s e um vértice v qualquer

Caminhos mais curto entre s e v

```
Print-Path(G, s, v)

        if v = s
        then print s
        else  if π[v] = NIL
                then  print "no path exists from " s" to "v"
                else   Print-Path(G, s, π[v])
                         print v
```

Grafo conectado

- Problema: Como saber se um grafo é conectado

(i.e., se cada par de vértices está conectado por um caminho)?

Exemplo: gostaria de saber se posso voar de qualquer cidade para qualquer cidade.

Grafo conectado

- Problema: Como saber se um grafo é conectado

(i.e., se cada par de vértices está conectado por um caminho)?

Exemplo: gostaria de saber se posso voar de qualquer cidade para qualquer cidade.

- Solução: usar BFS
  - Escolher um vértice v qualquer de G
  - Executar BFS à partir de v
  - Verificar se todos vértices foram pintados de preto

Grafo conectado

- Problema: Como saber se um grafo é conectado

(i.e., se cada par de vértices está conectado por um caminho)?

Exemplo: gostaria de saber se posso voar de qualquer cidade para qualquer cidade.

- Solução: usar BFS
  - Escolher um … v qualquer de G
  - Execut…
  - Verificar se todos vértices foram pintados de preto

Precisamos usar BFS?

Grafo conectado

- Problema: Como saber se um grafo é conectado

(i.e., se cada par de vértices está conectado por um caminho)?

Exemplo: gostaria de saber se posso voar de qualquer cidade para qualquer cidade.

- Solução: usar BFS
  - Escolher um ... qualquer de G
  - Execut...
  - Verifica... de preto

Precisamos usar BFS?
Não, podemos usar qualquer algoritmo de busca.
Note que não importa a ordem.

Grafo conectado

- Problema: Como saber se um grafo é conectado

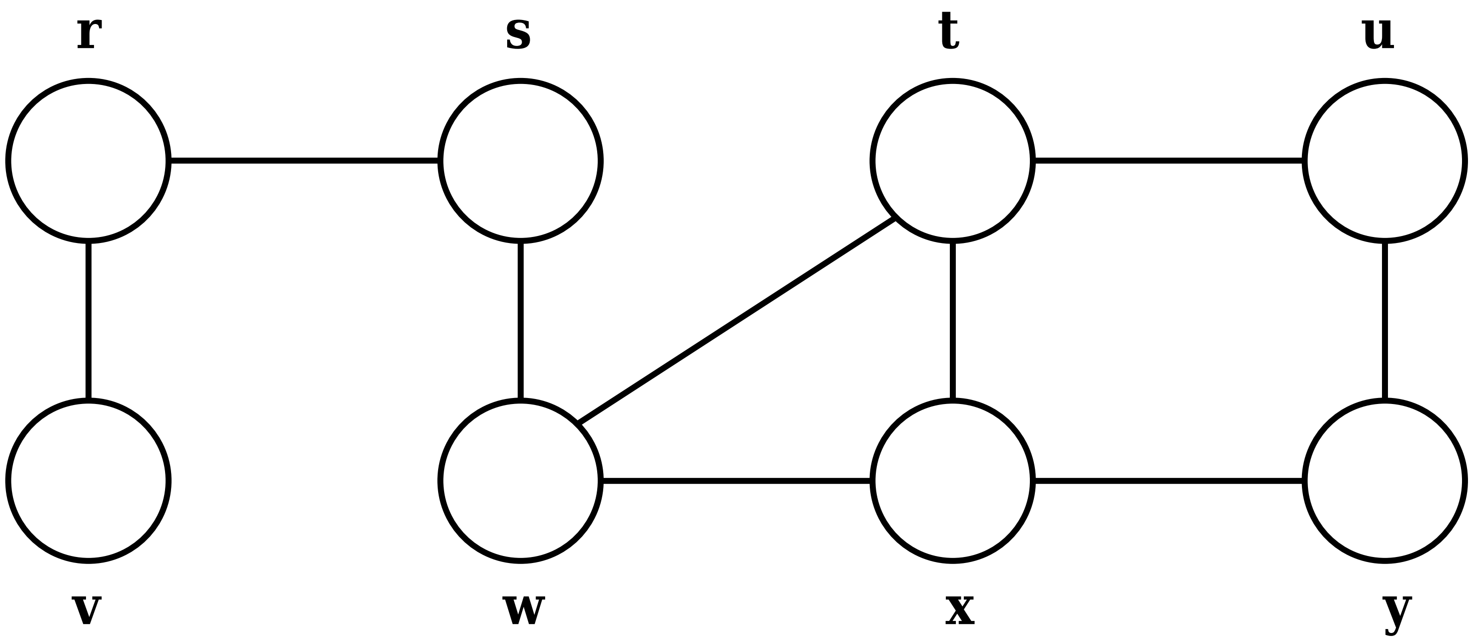
(i.e., se cada par de vértices está conectado por um caminho)?

Exemplo: gostaria de saber se posso voar de qualquer cidade para qualquer cidade.

- Podemos usar o seguinte algoritmo?

```
R = {s}
while  existir aresta (u,v) em que u pertence a R e v não pertence a R
       R = R ∪ {v}
```

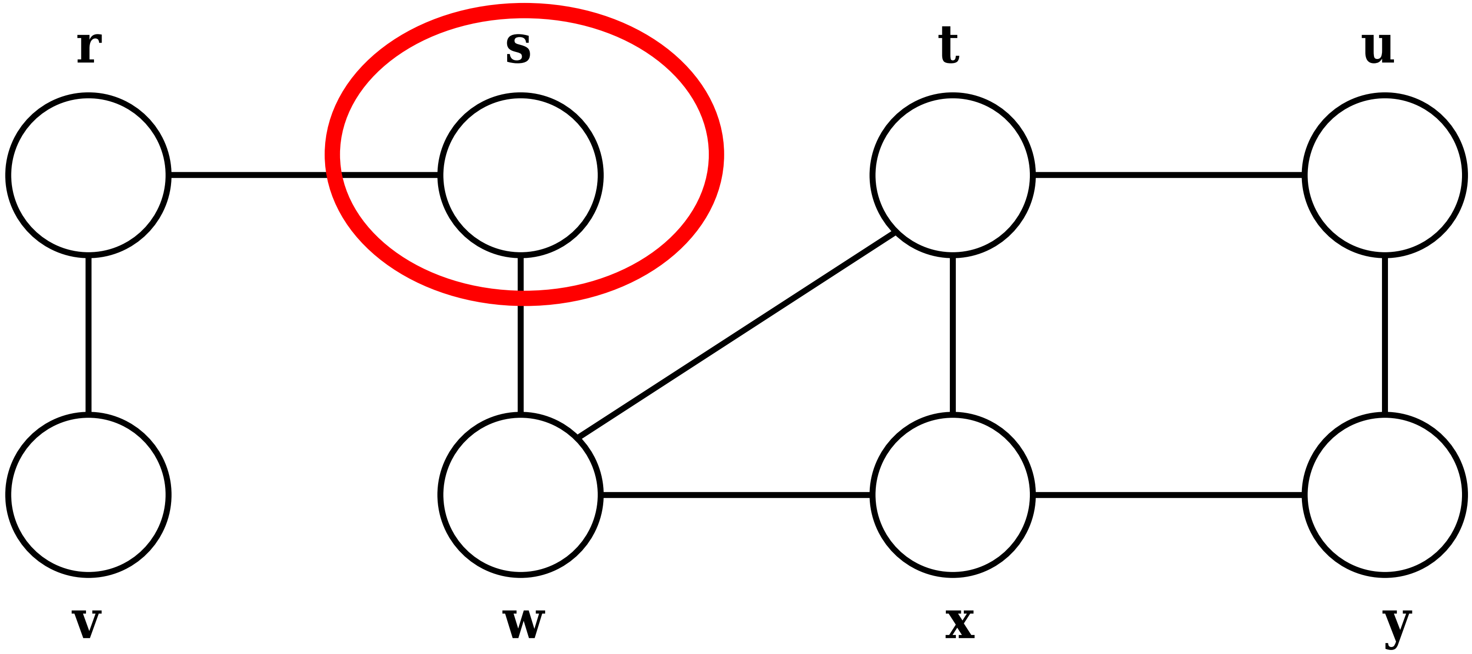
Grafo conectado



R = {s}
while existir aresta (u,v) em que u pertence a R e v não pertence a R
R = R ∪ {v}

Quais os elementos do conjunto R depois de aplicar o algoritmo, quando R é inicializado com o vértice s do grafo?

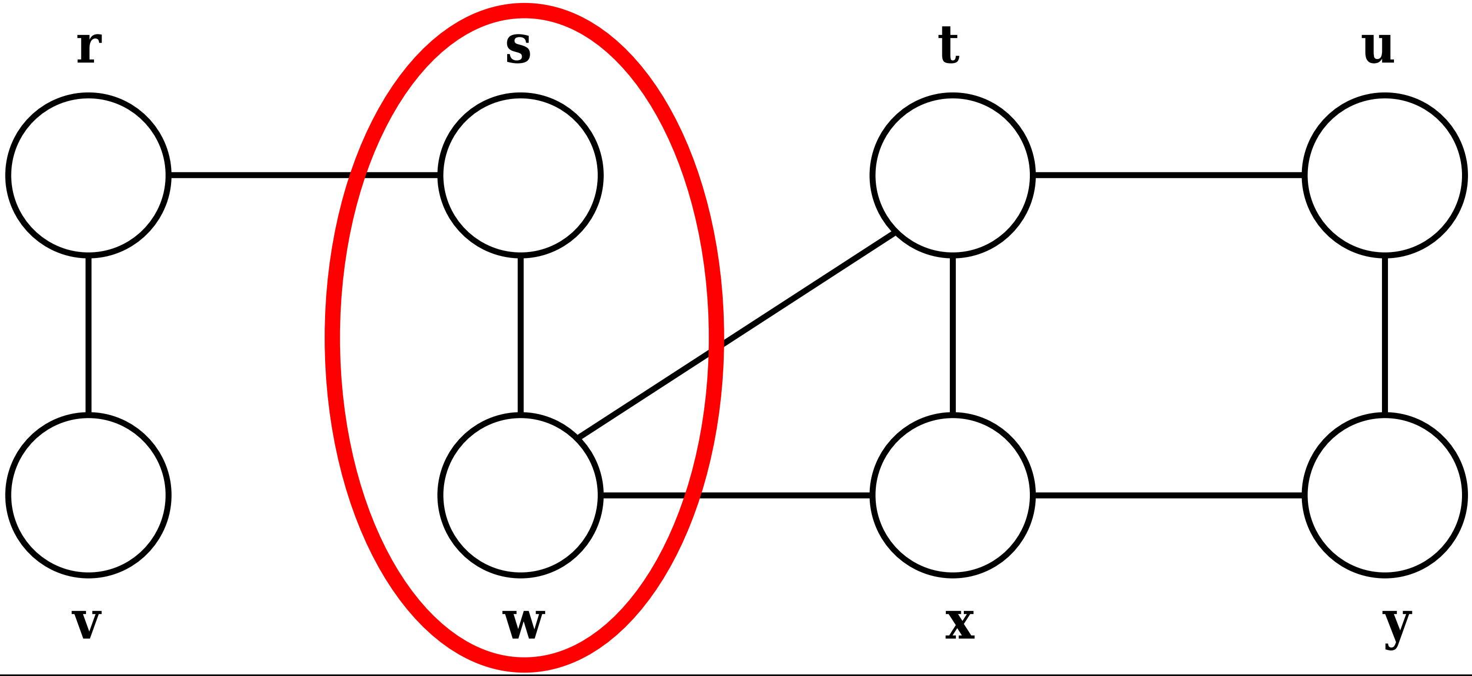
Grafo conectado



R = {s}
while existir aresta (u,v) em que u pertence a R e v não pertence a R
    R = R ∪ {v}

Quais os elementos do conjunto R depois de aplicar o algoritmo, quando R é inicializado com o vértice s do grafo?

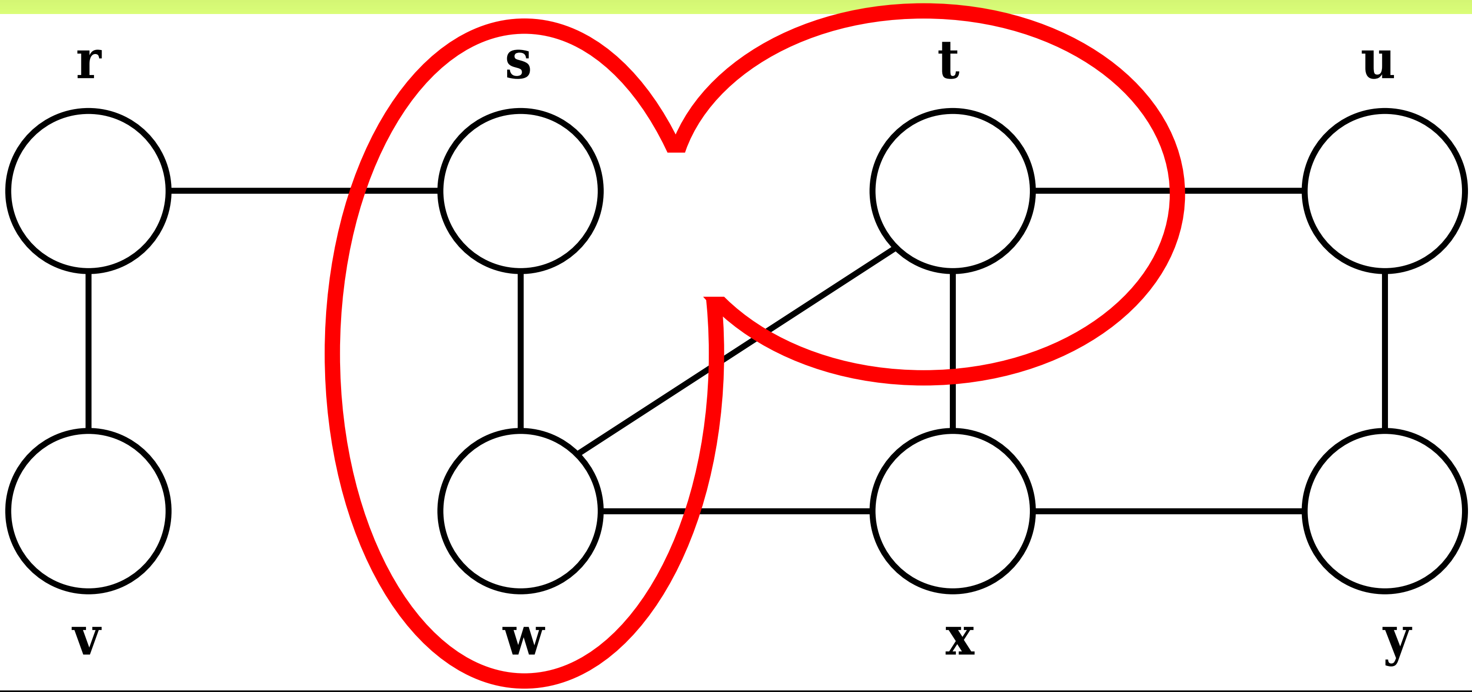
Grafo conectado



R = {s}
while existir aresta (u,v) em que u pertence a R e v não pertence a R
R = R ∪ {v}

Quais os elementos do conjunto R depois de aplicar o algoritmo, quando R é inicializado com o vértice s do grafo?

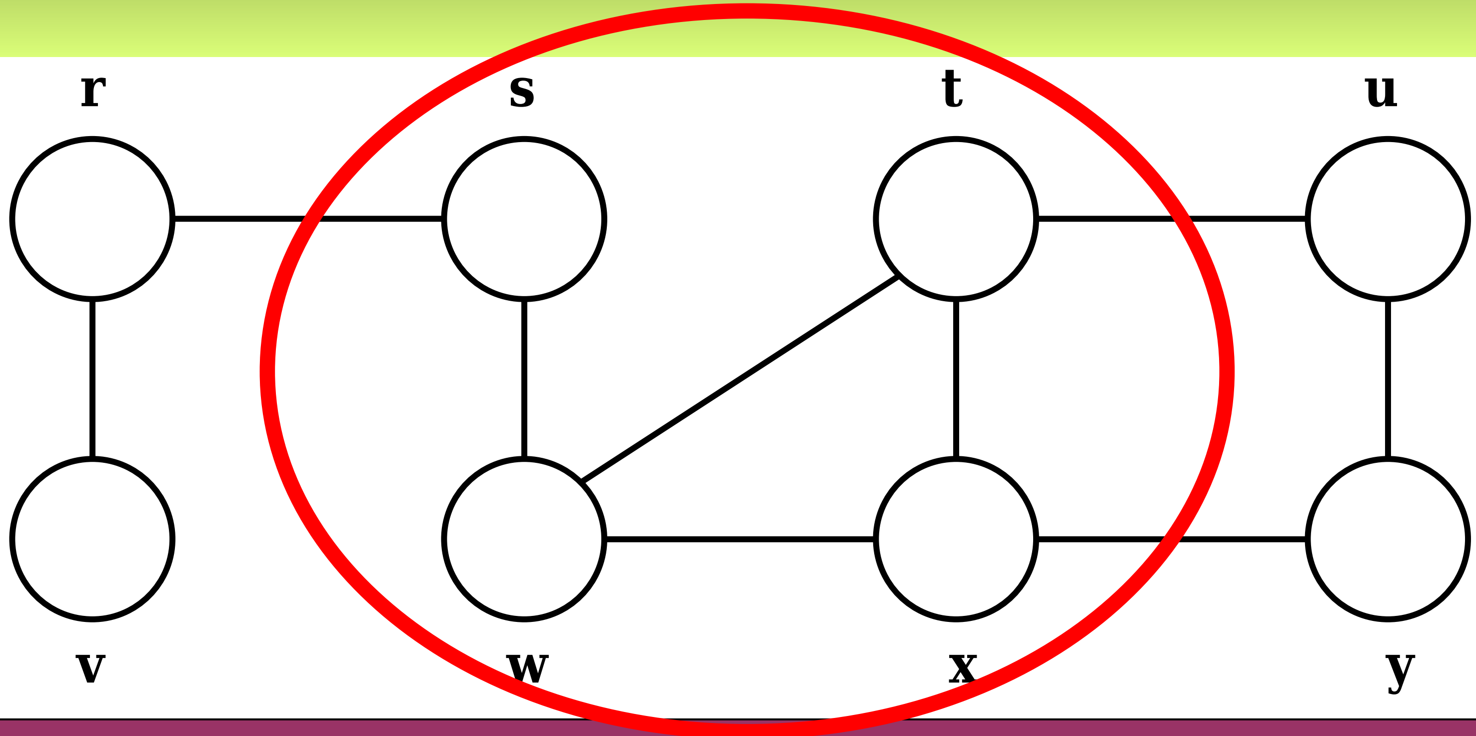
Grafo conectado



R = {s}
while existir aresta (u,v) em que u pertence a R e v não pertence a R
R = R ∪ {v}

Quais os elementos do conjunto R depois de aplicar o algoritmo, quando R é inicializado com o vértice s do grafo?

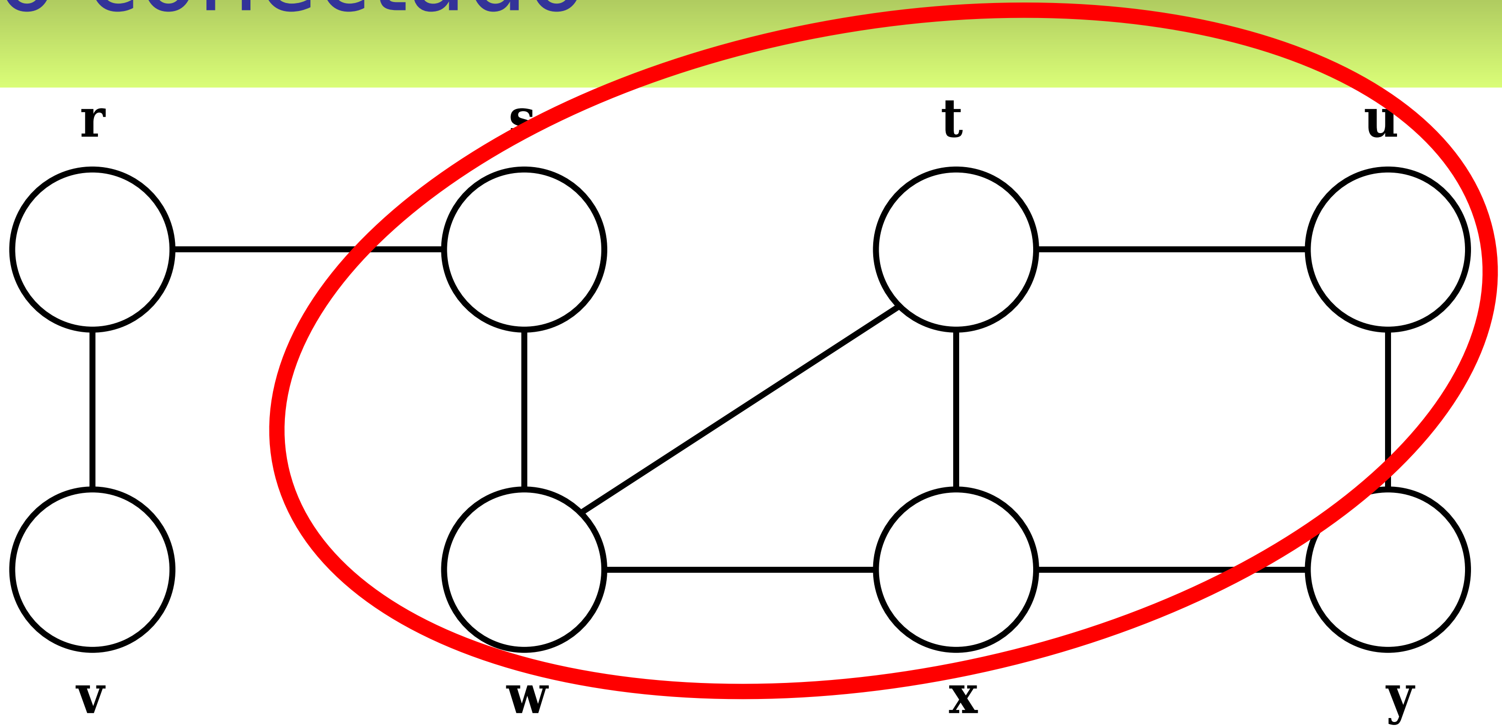
Grafo conectado



R = {s}
while existir aresta (u,v) em que u pertence a R e v não pertence a R
R = R ∪ {v}

Quais os elementos do conjunto R depois de aplicar o algoritmo, quando R é inicializado com o vértice s do grafo?

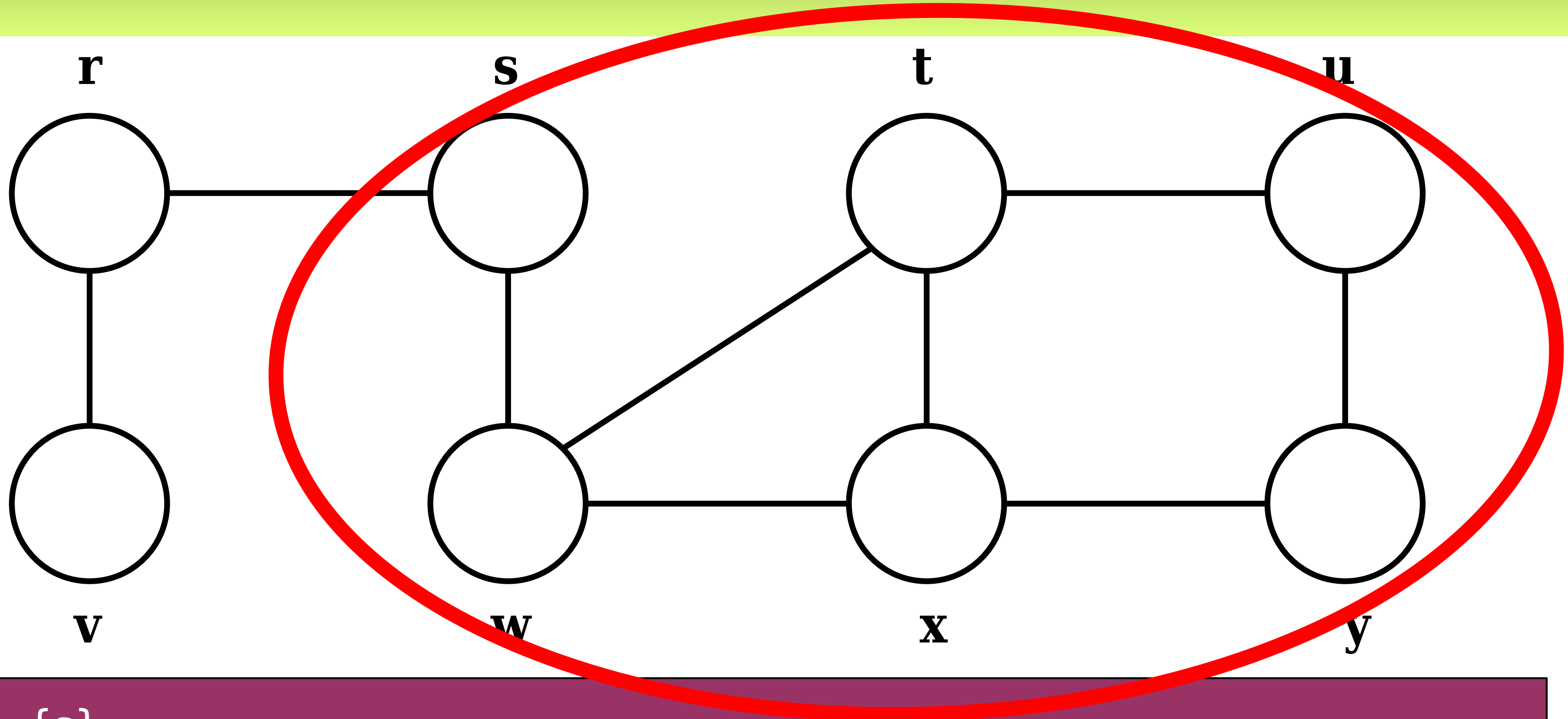
Grafo conectado



R = {s}
while existir aresta (u,v) em que u pertence a R e v não pertence a R
R = R ∪ {v}

Quais os elementos do conjunto R depois de aplicar o algoritmo, quando R é inicializado com o vértice s do grafo?

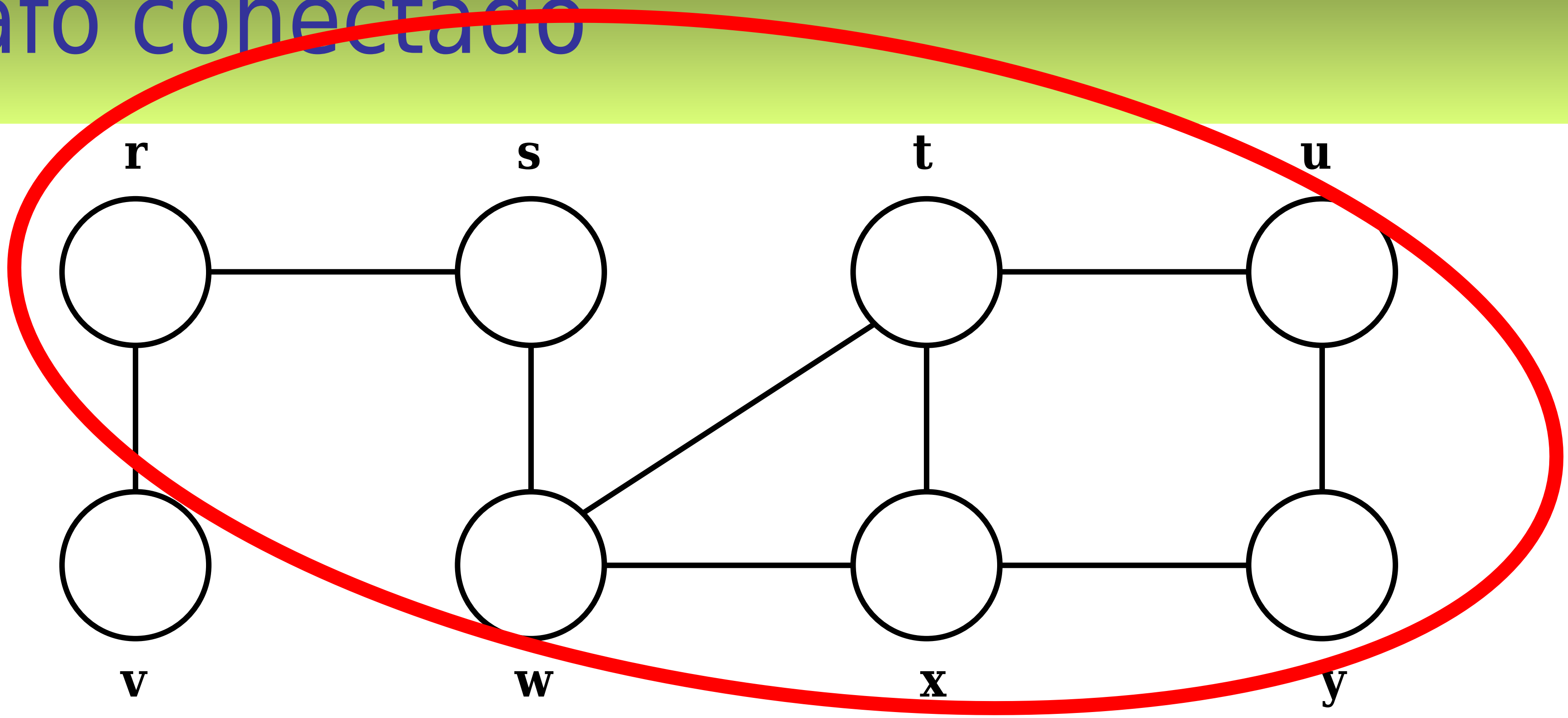
Grafo conectado



R = {s}
while existir aresta (u,v) em que u pertence a R e v não pertence a R
R = R ∪ {v}

Quais os elementos do conjunto R depois de aplicar o algoritmo, quando R é inicializado com o vértice s do grafo?

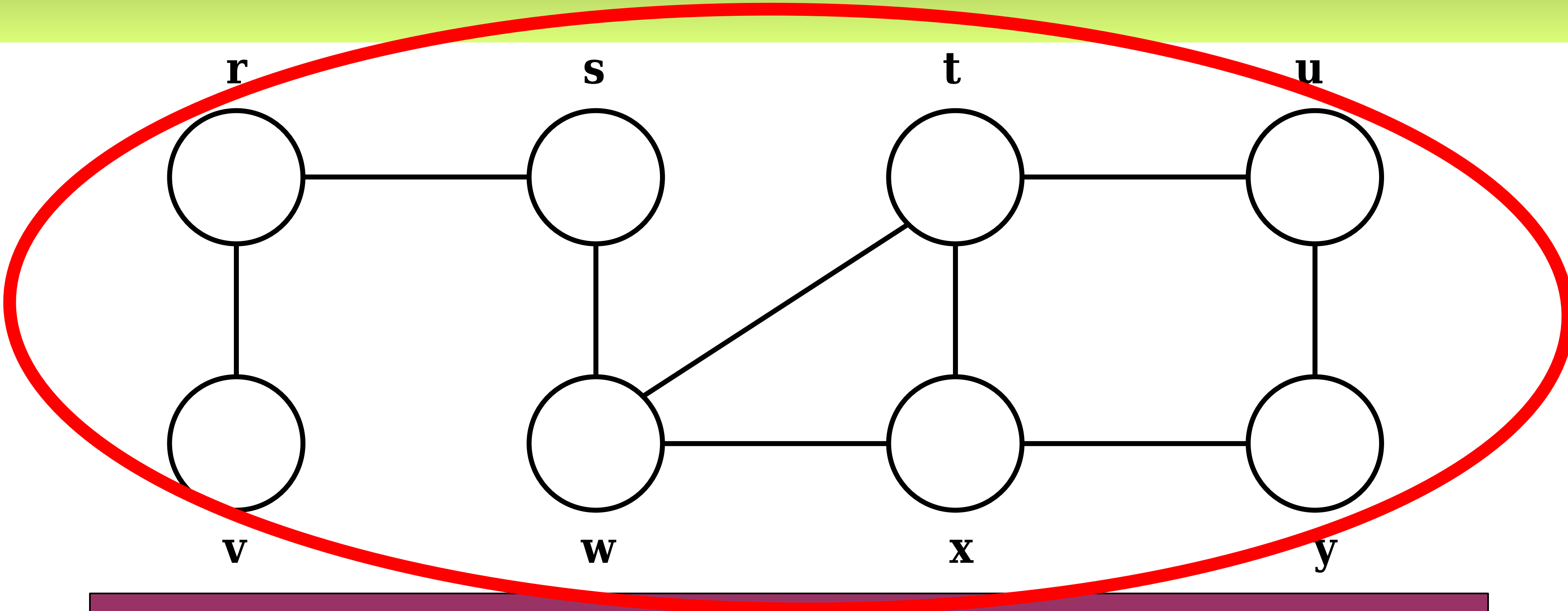
Grafo conectado



R = {s}
while existir aresta (u,v) em que u pertence a R e v não pertence a R
R = R ∪ {v}

Quais os elementos do conjunto R depois de aplicar o algoritmo, quando R é inicializado com o vértice s do grafo?

Grafo conectado



R = {s}
while existir aresta (u,v) em que u pertence a R e v não pertence a R
R = R ∪ {v}

Quais os elementos do conjunto R depois de aplicar o algoritmo, quando R é inicializado com o vértice s do grafo?

Grafo conectado

- Problema: Como saber se um grafo é conectado
(i.e., se cada par de vértices está conectado por um caminho)?
- Solução: usar o seguinte algoritmo

```
R = {s}
while  existir aresta (u,v) em que u pertence a R e v não pertence a R
        R = R ∪ {v}
```

- s é um vértice origem qualquer de G
- R é o conjunto de vértices que possuem caminho até s. Assim, R é o componente conectado que possui s.

Grafo conectado

- Problema: Como saber se um grafo é conectado

(i.e., se cada par de vértices está conectado por um caminho)?

- Solução: usar o seguinte algoritmo

```
R = {s}
while  existir aresta (u,v) em que u pertence a R e v não pertence a R
        R = R ∪ {v}
```

- s é um vértice origem qualquer de G
- R é o conjunto de vértices que possuem caminho até s. Assim, R é o componente conectado que possui s.
- Se R contém todos os vértices, então o grafo é conectado, caso contrário, não é.

Busca em profundidade (DFS depth-first-search)

- É outro método de busca
- A idéia é proseguir a busca sempre a partir do vértice descoberto mais recentemente, até que este não tenha mais vizinhos descobertos. Neste caso, volta-se na busca para o precursor desse vértice.
- Oposto de BFS que explora o vértice mais antigo primeiro
- DFS devolve uma floresta

Busca em profundidade (DFS depth-first-search)

- Intuição: Procurar uma saída de um labirinto
  - vai fundo atrás da saída (tomando decisões a cada encruzilhada)
  - volta a última encruzilhada quando encontrar um beco sem saída ou encontrar um lugar já visitado.

Busca em profundidade (DFS depth-first-search)

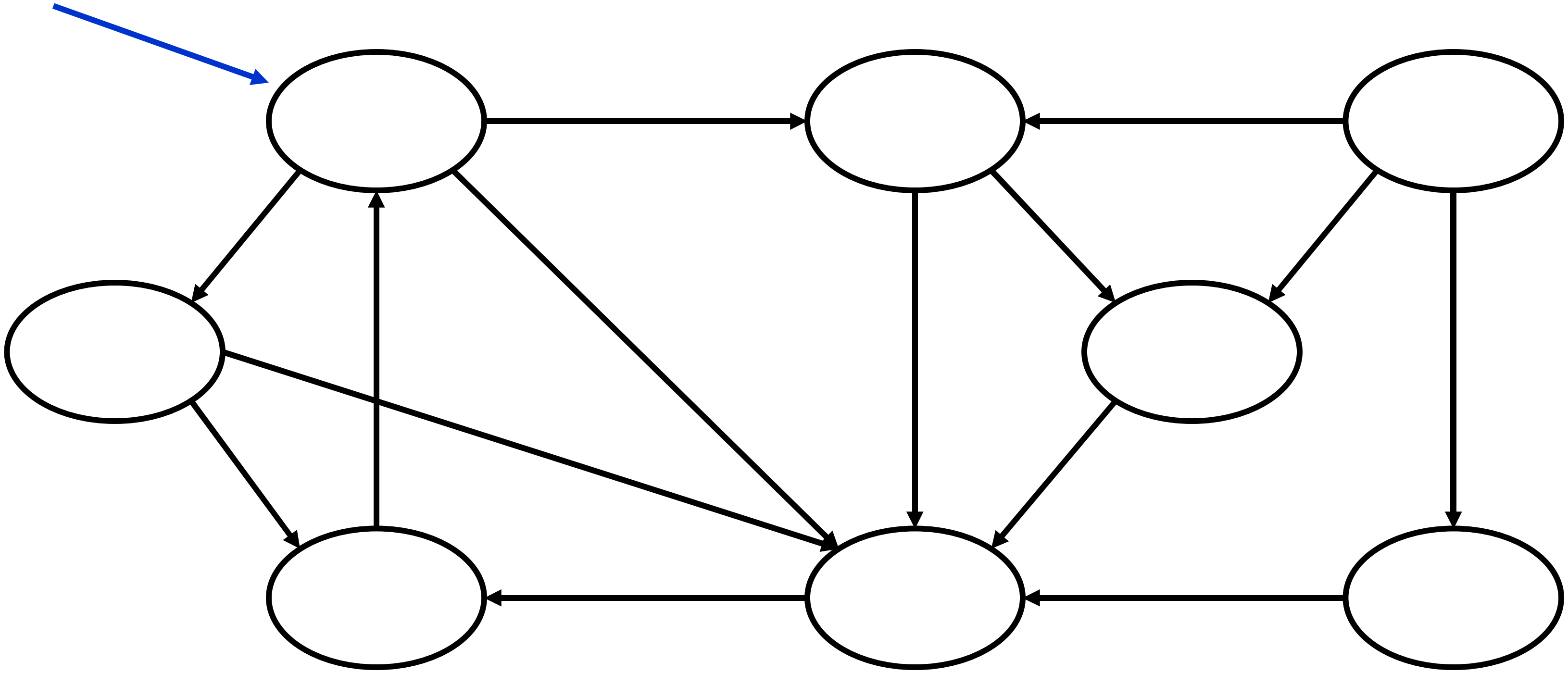
– Os vértices recebem 2 rótulos:

- d[.]: o momento em que o vértice foi descoberto (tornou-se cinza)
- f[.]: o momento em que examinamos os seus vizinhos (tornou-se preto)

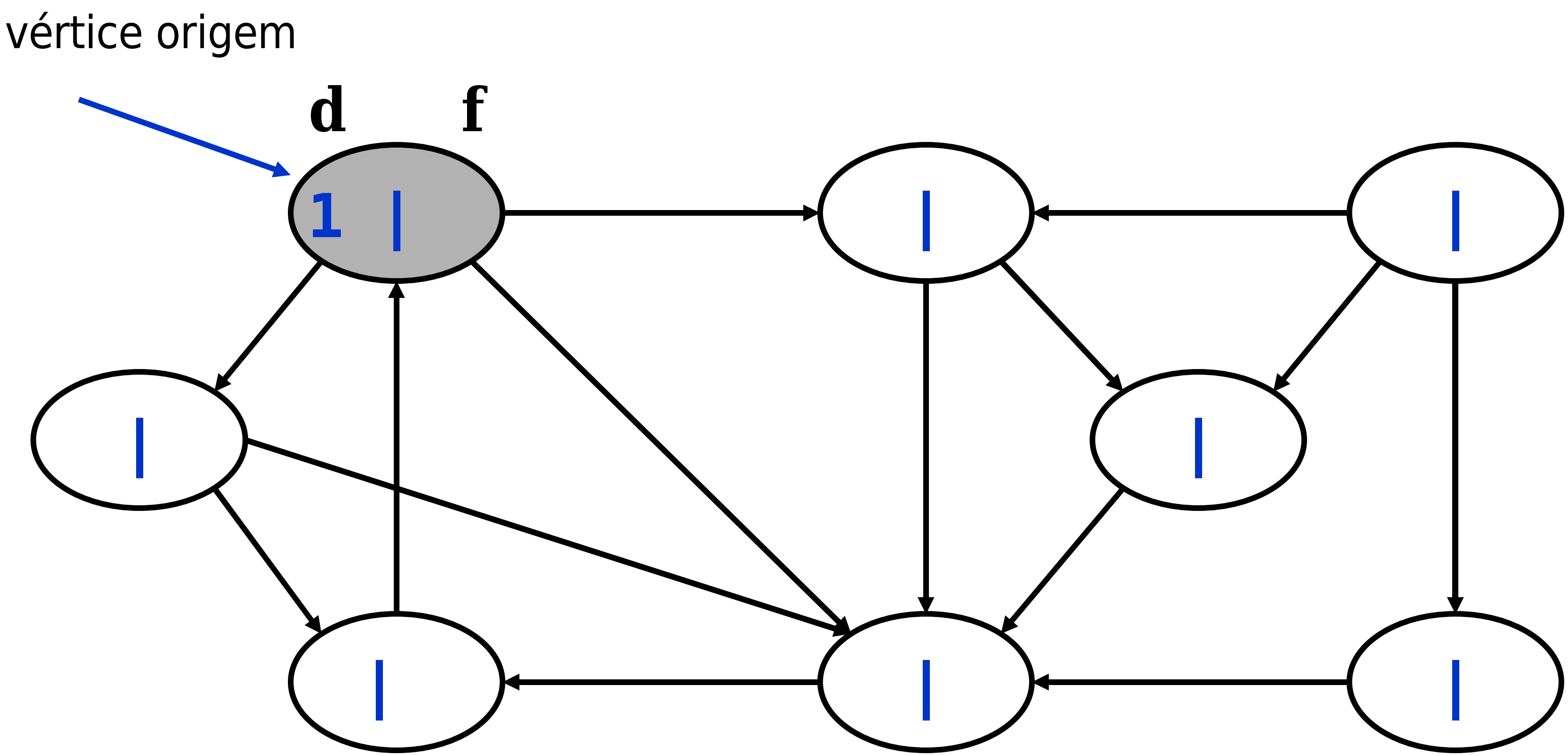
O vértice é branco até d , cinza entre d e f e preto a partir de f.

Busca em profundidade (DFS depth-first-search)

vértice origem

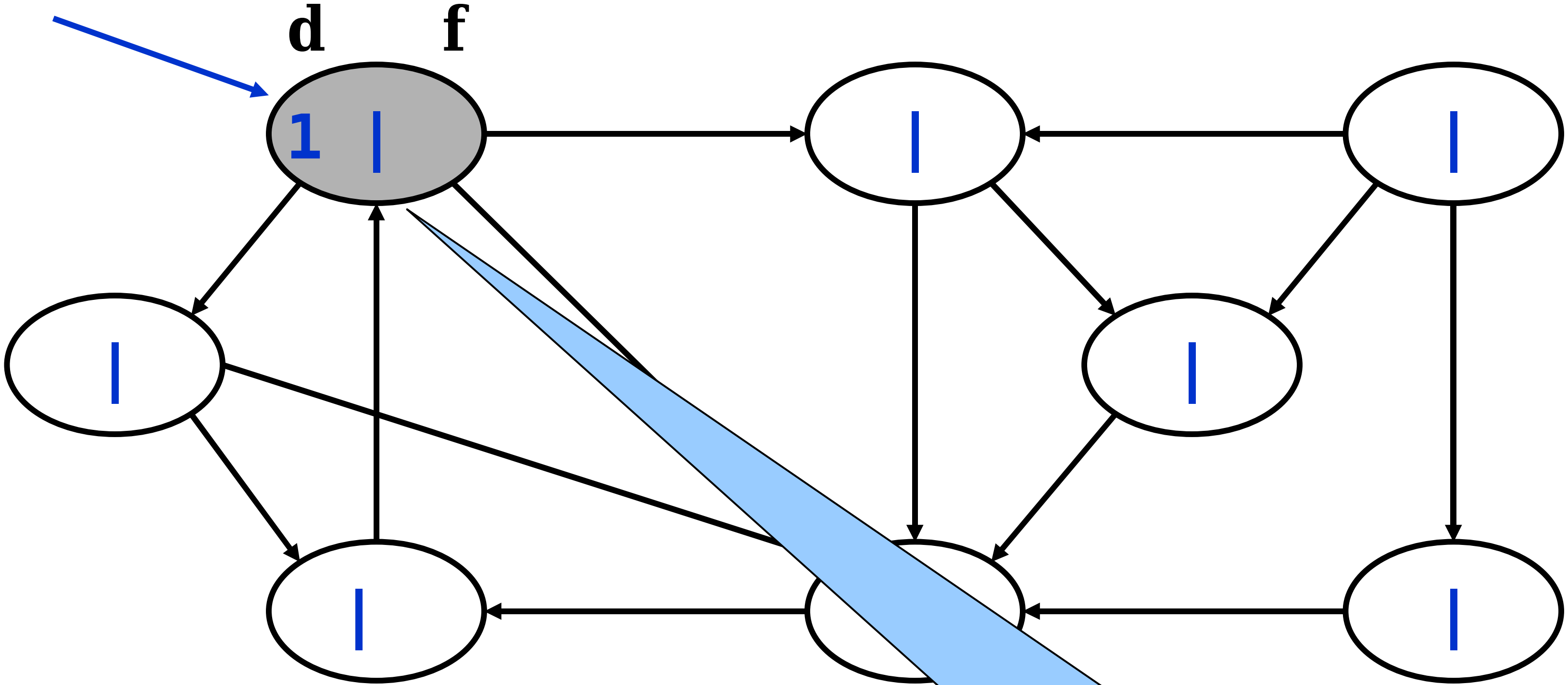


Busca em profundidade (DFS depth-first-search)



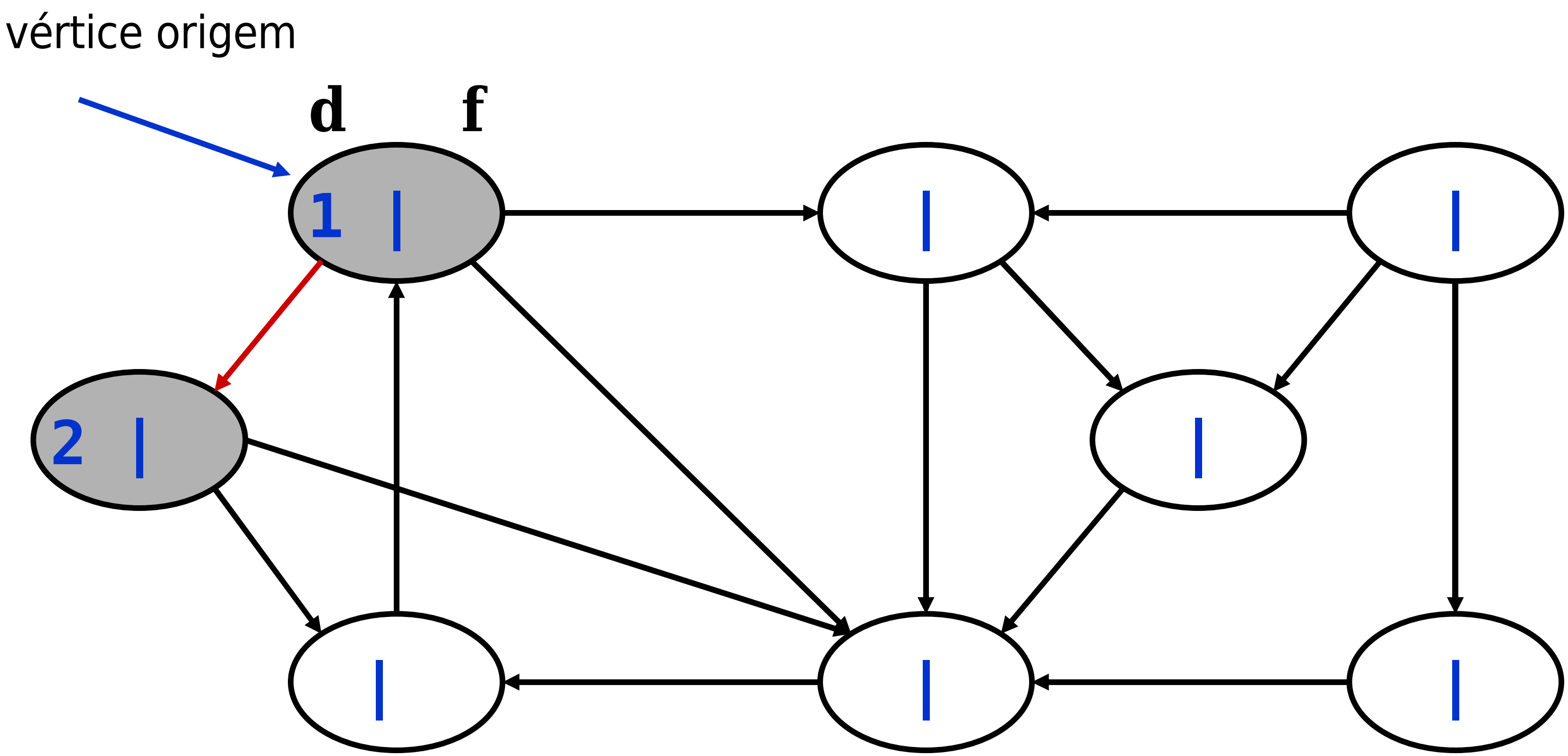
Busca em profundidade (DFS depth-first-search)

vértice origem

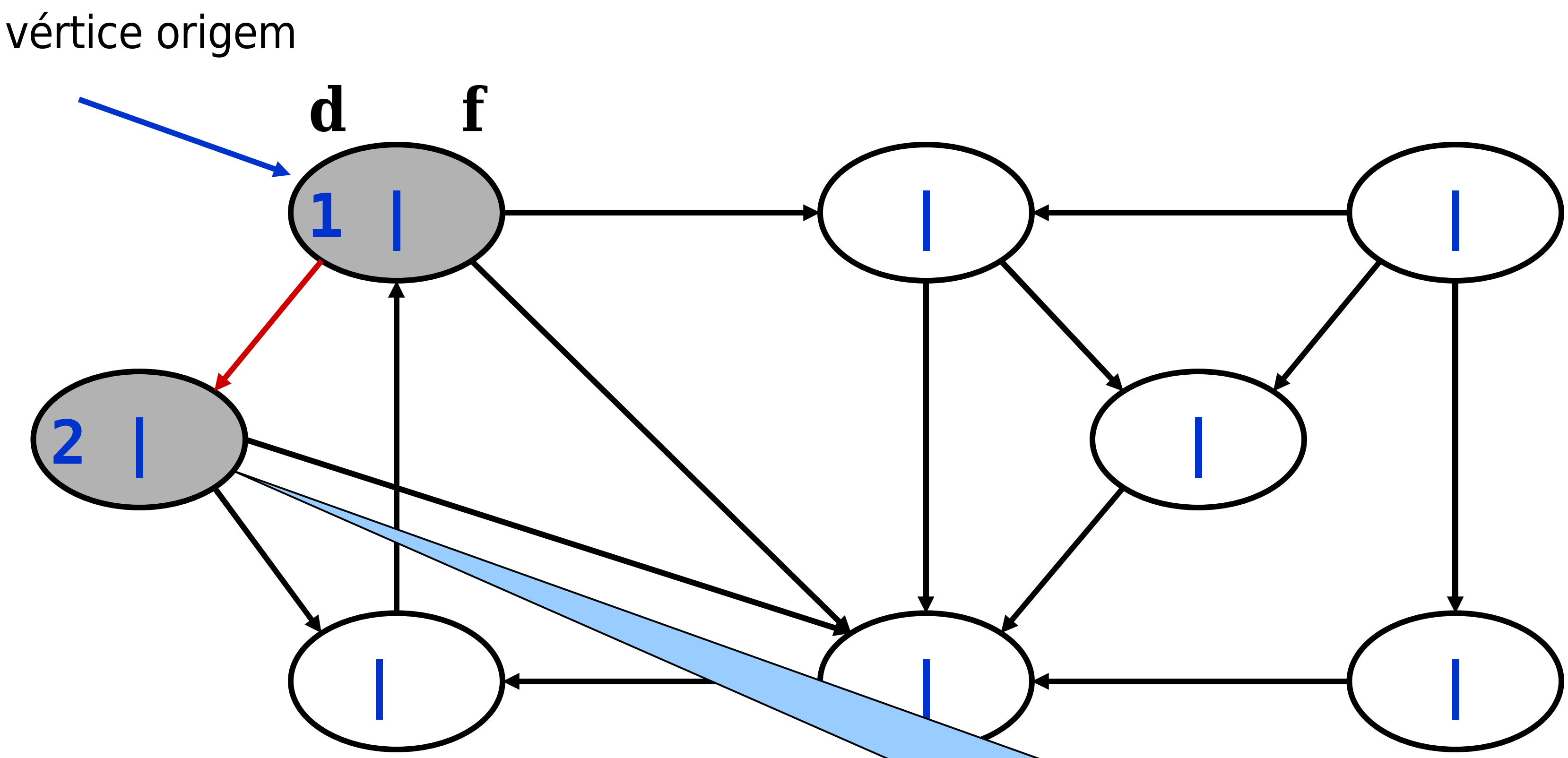


Existe algum vértice adjacente ao vértice que não tenha sido descoberto?

Busca em profundidade (DFS depth-first-search)



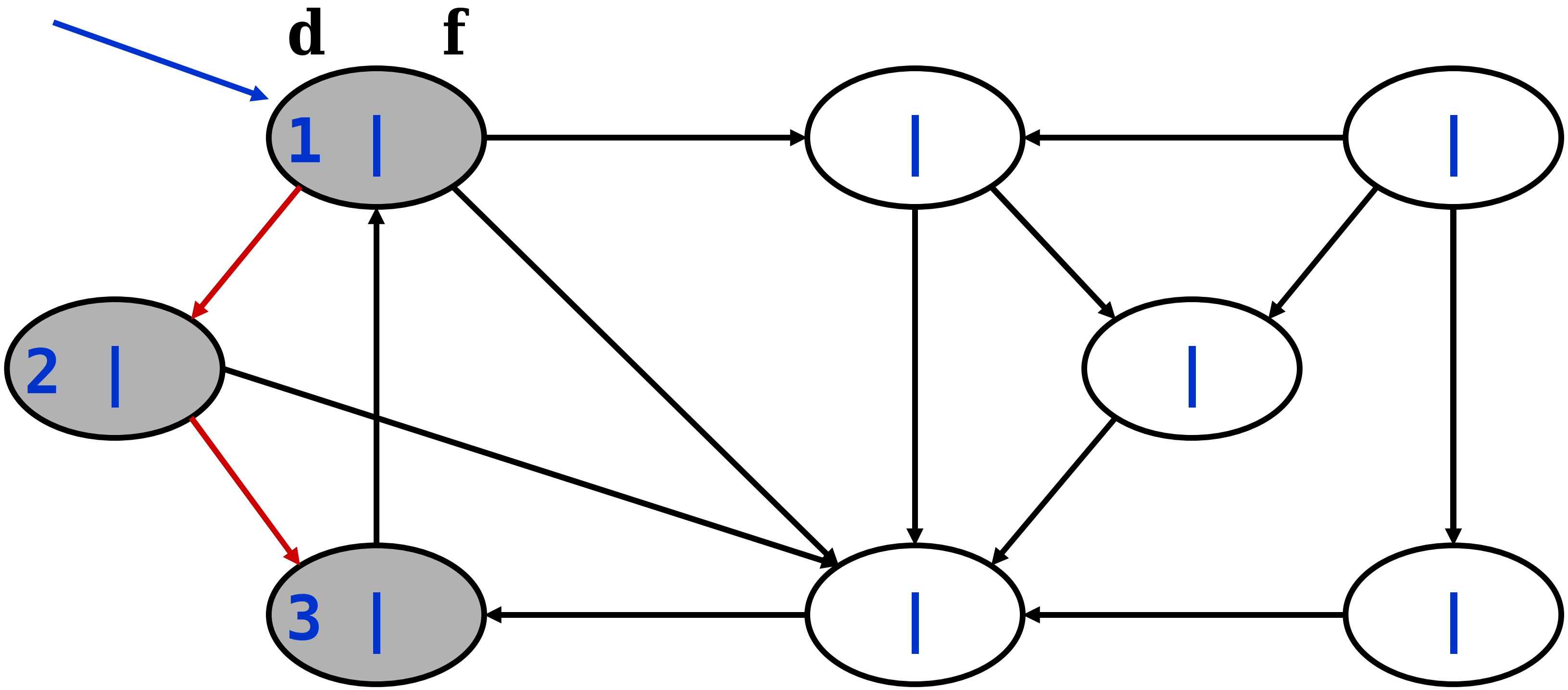
Busca em profundidade (DFS depth-first-search)



Existe algum vértice adjacente ao vértice que não tenha sido descoberto?

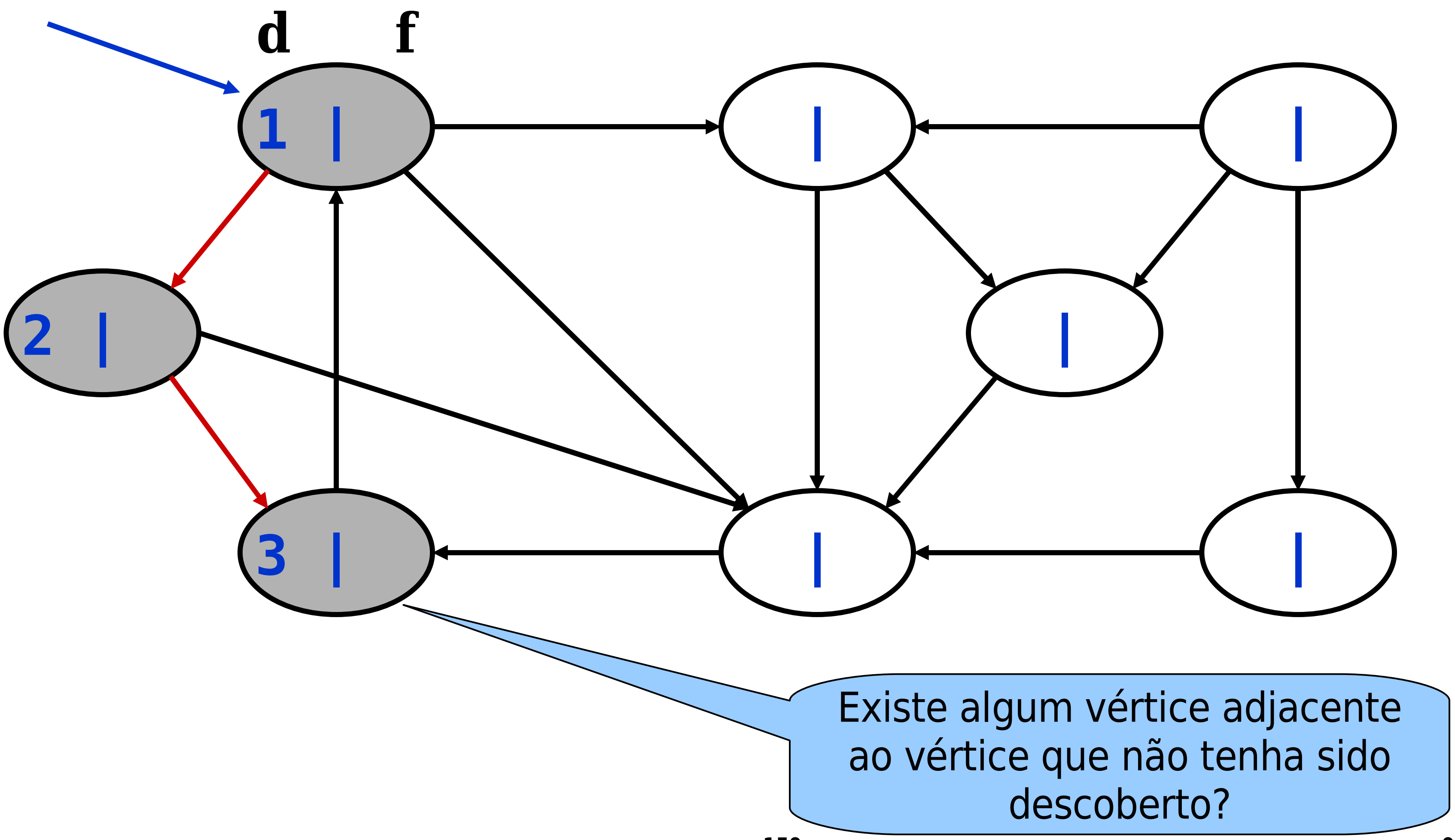
Busca em profundidade (DFS depth-first-search)

vértice origem



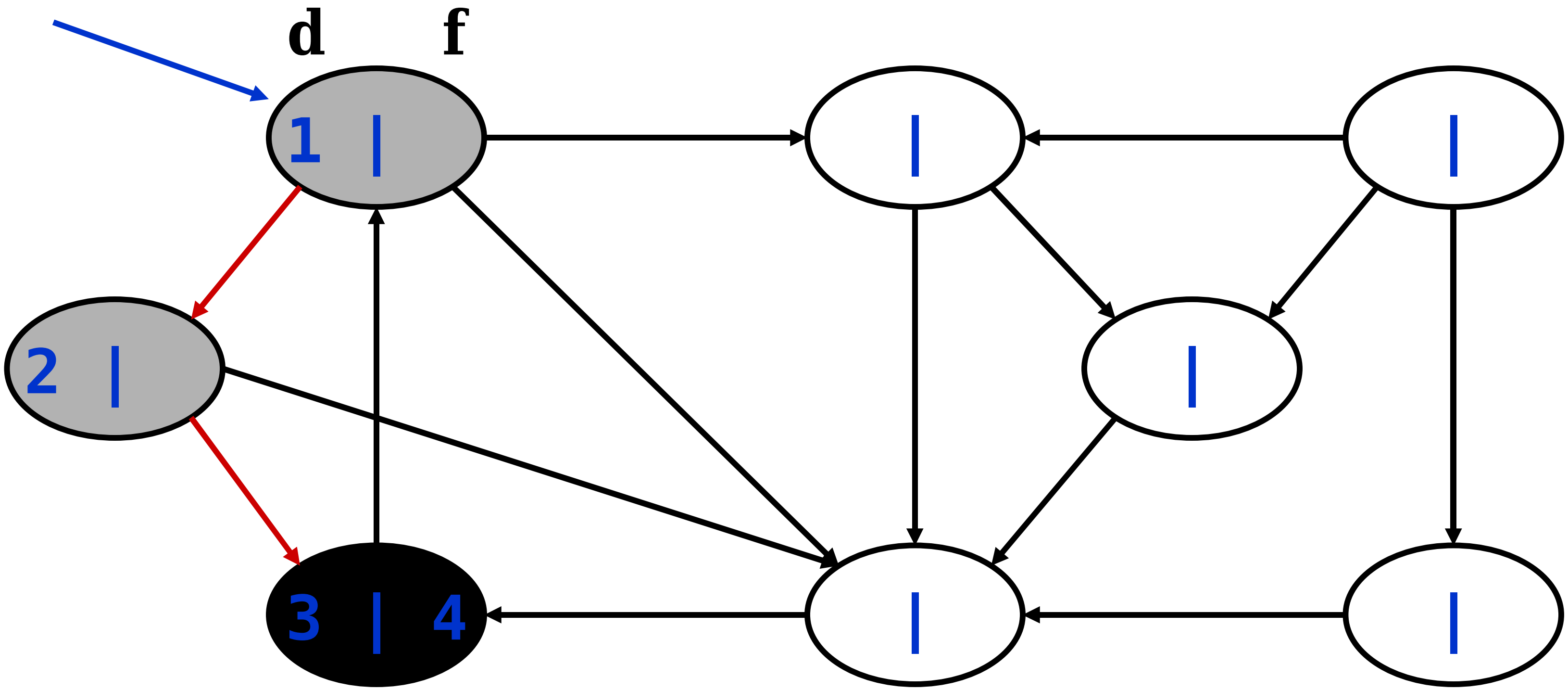
Busca em profundidade (DFS depth-first-search)

vértice origem



Busca em profundidade (DFS depth-first-search)

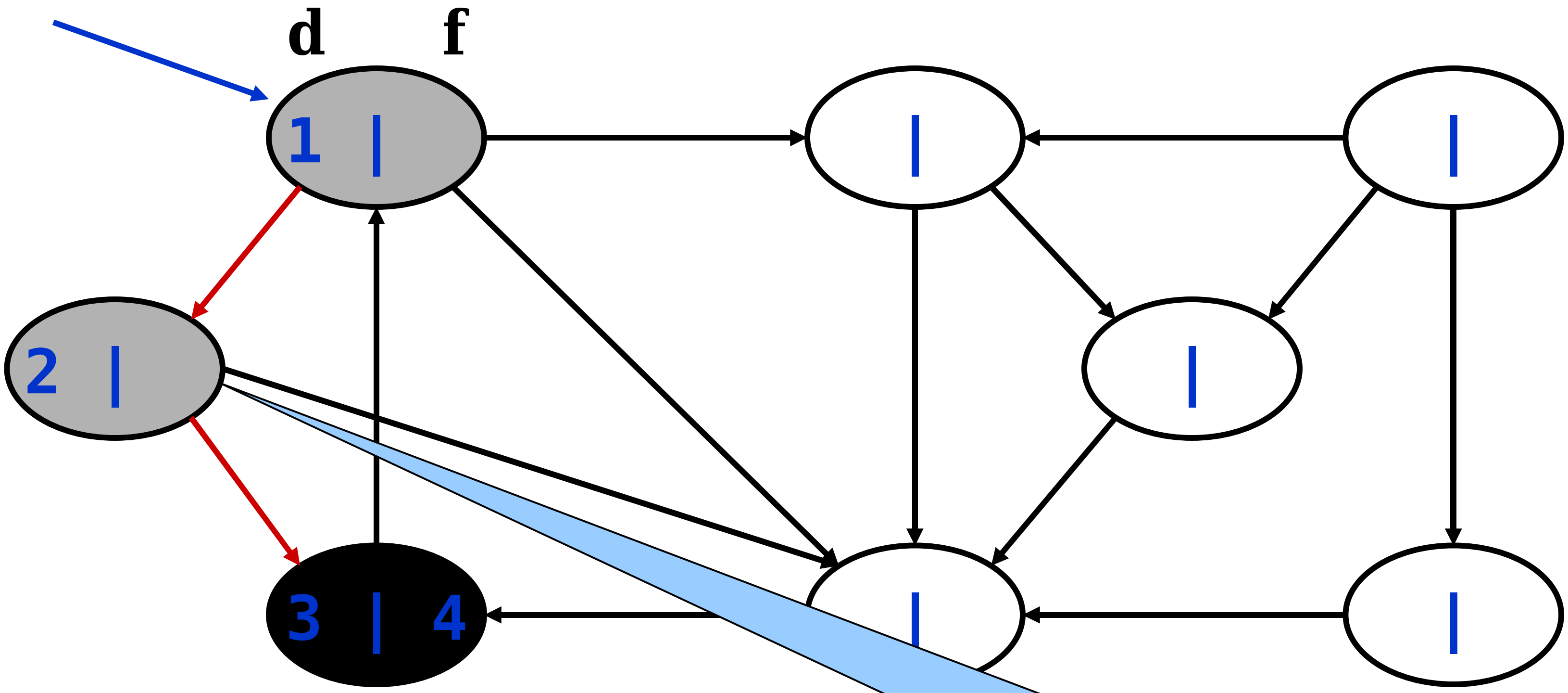
vértice origem



Não existe. Então, terminei com o vértice (pinta ele de preto)
Volta-se na busca para o precursor desse vértice.

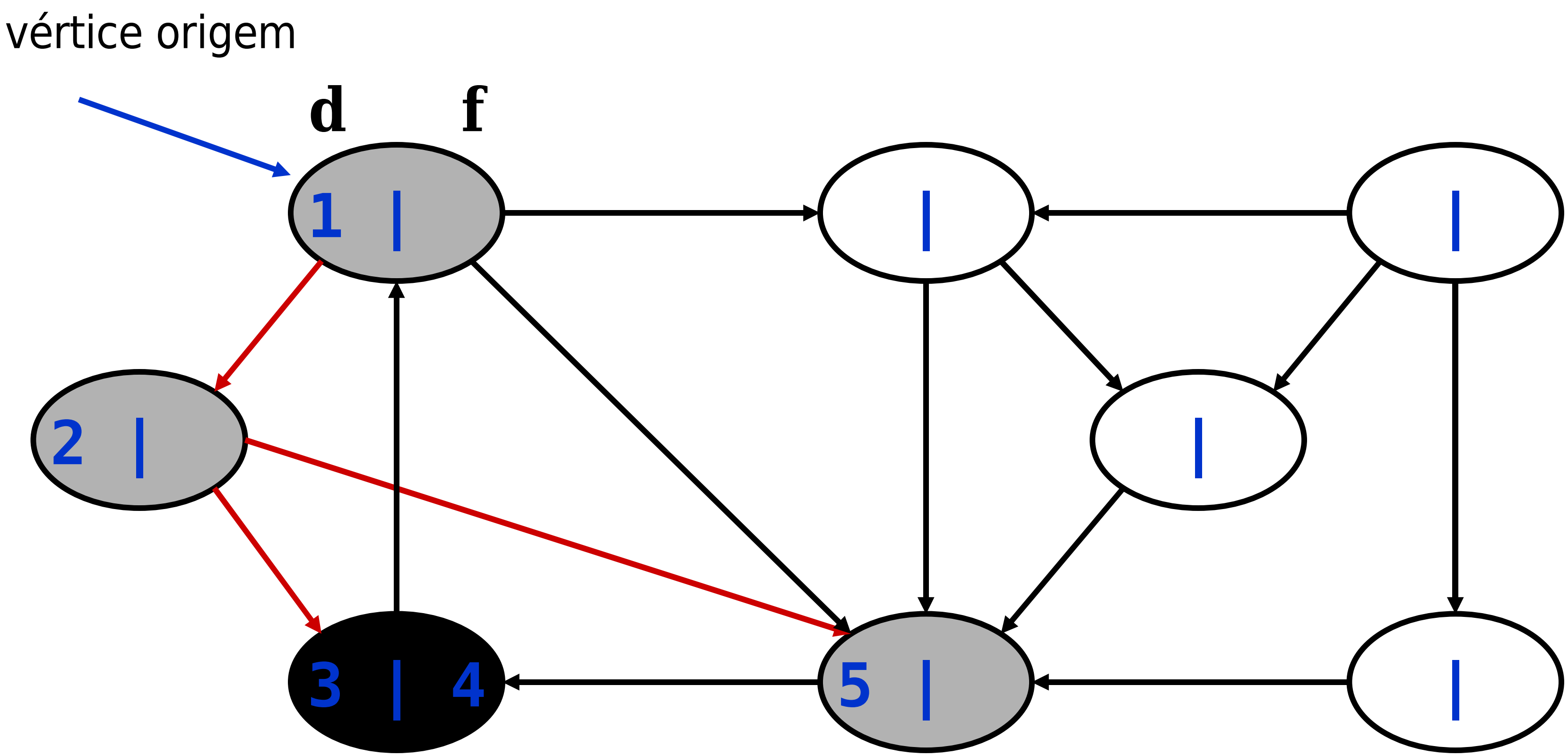
Busca em profundidade (DFS depth-first-search)

vértice origem

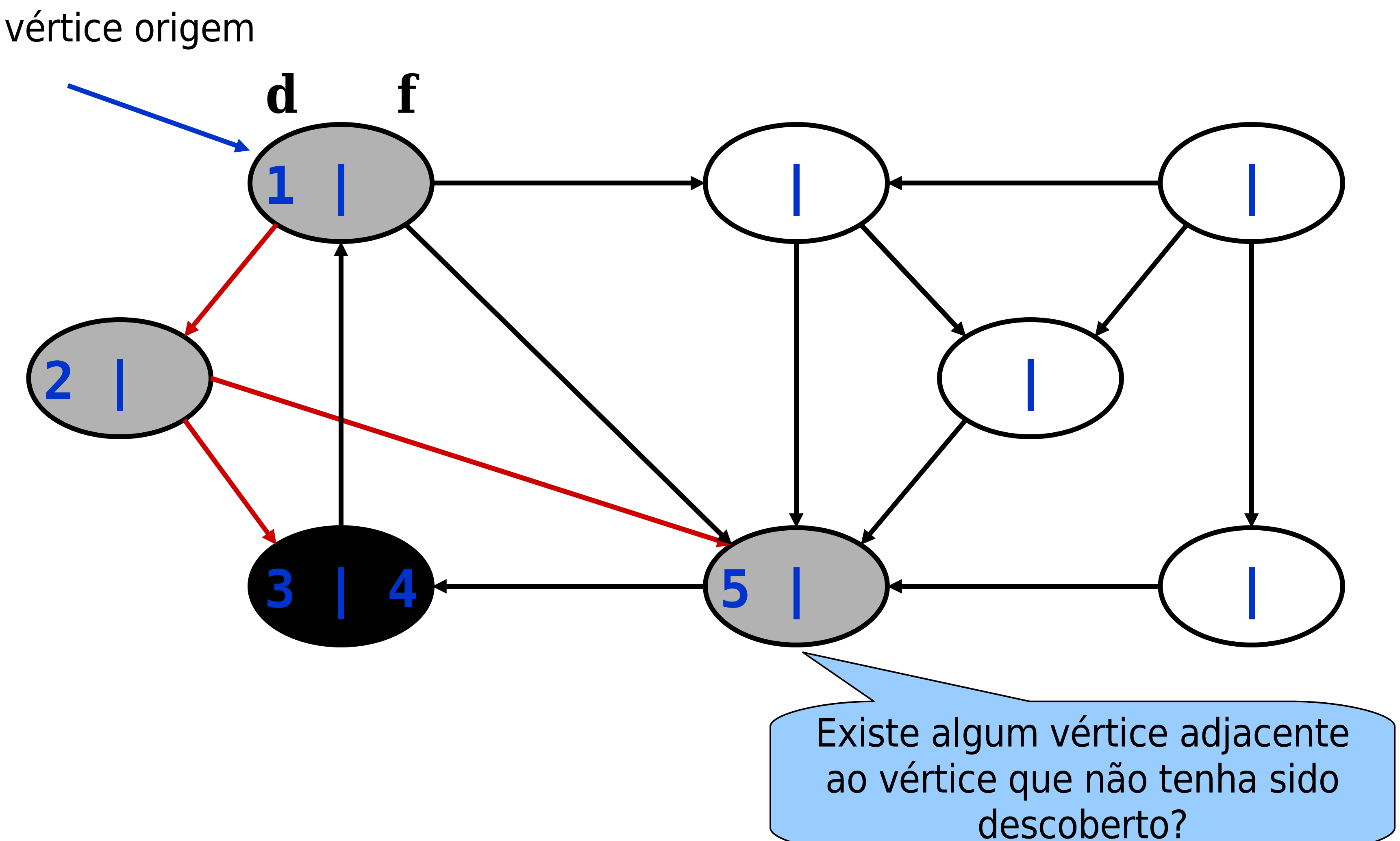


Existe algum vértice adjacente ao vértice que não tenha sido descoberto?

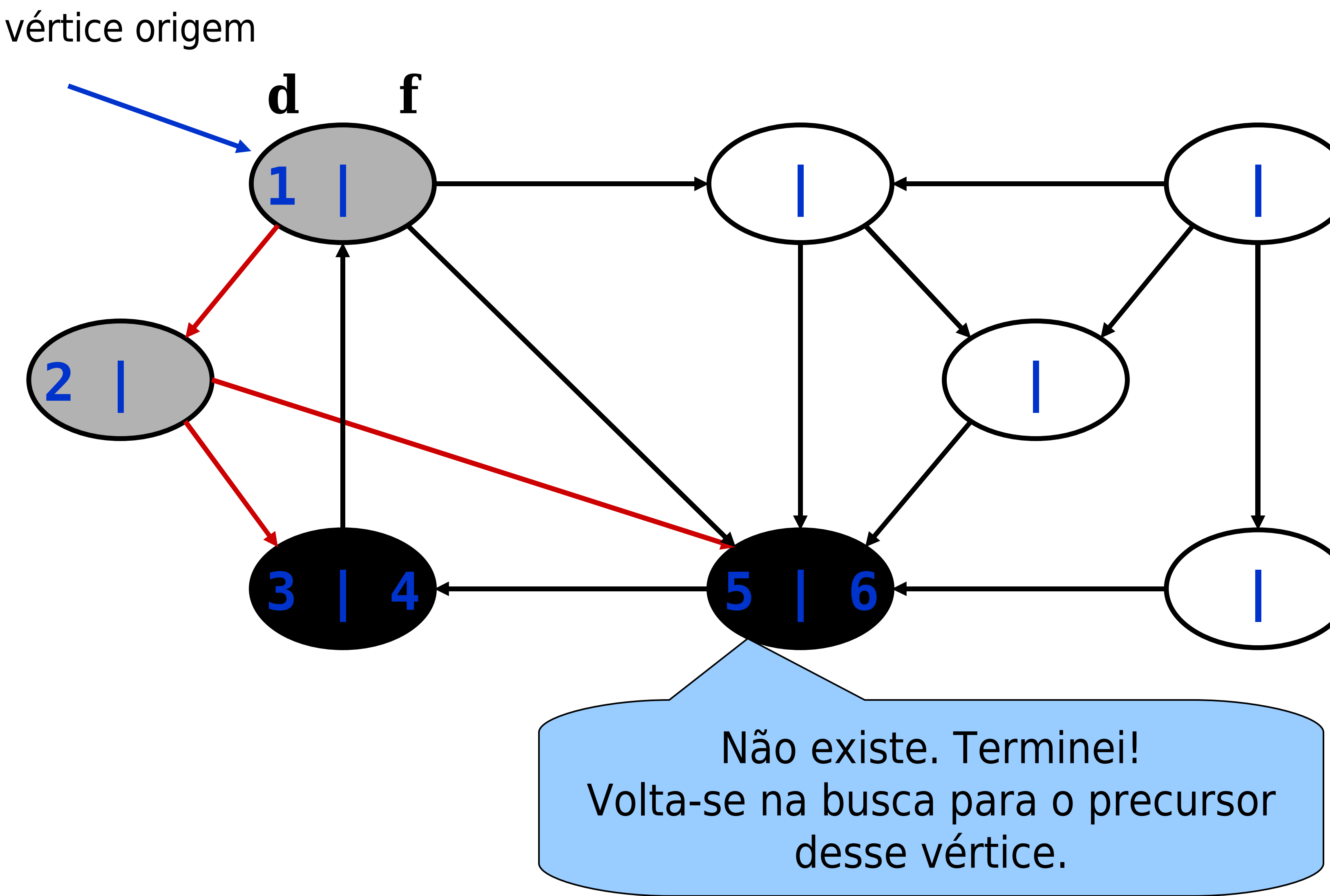
Busca em profundidade (DFS depth-first-search)



Busca em profundidade (DFS depth-first-search)

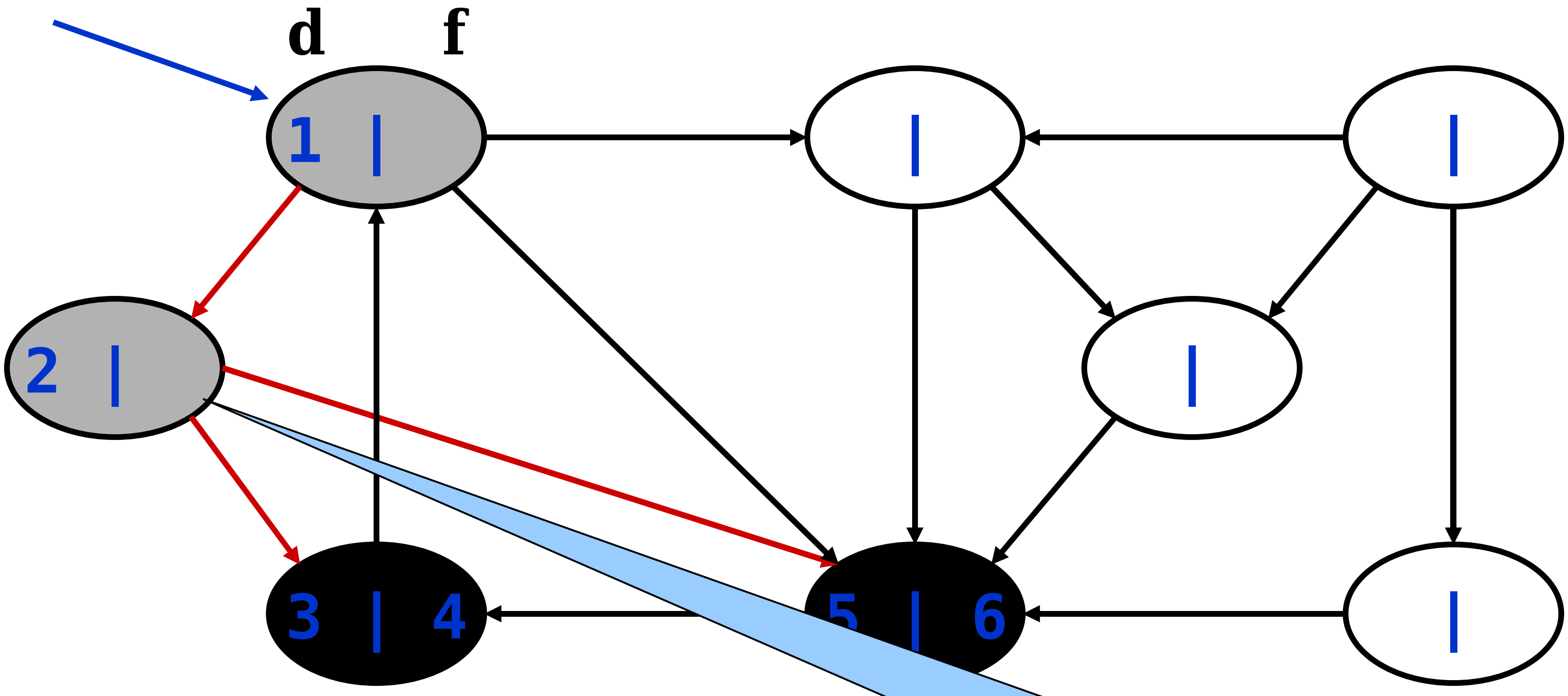


Busca em profundidade (DFS depth-first-search)



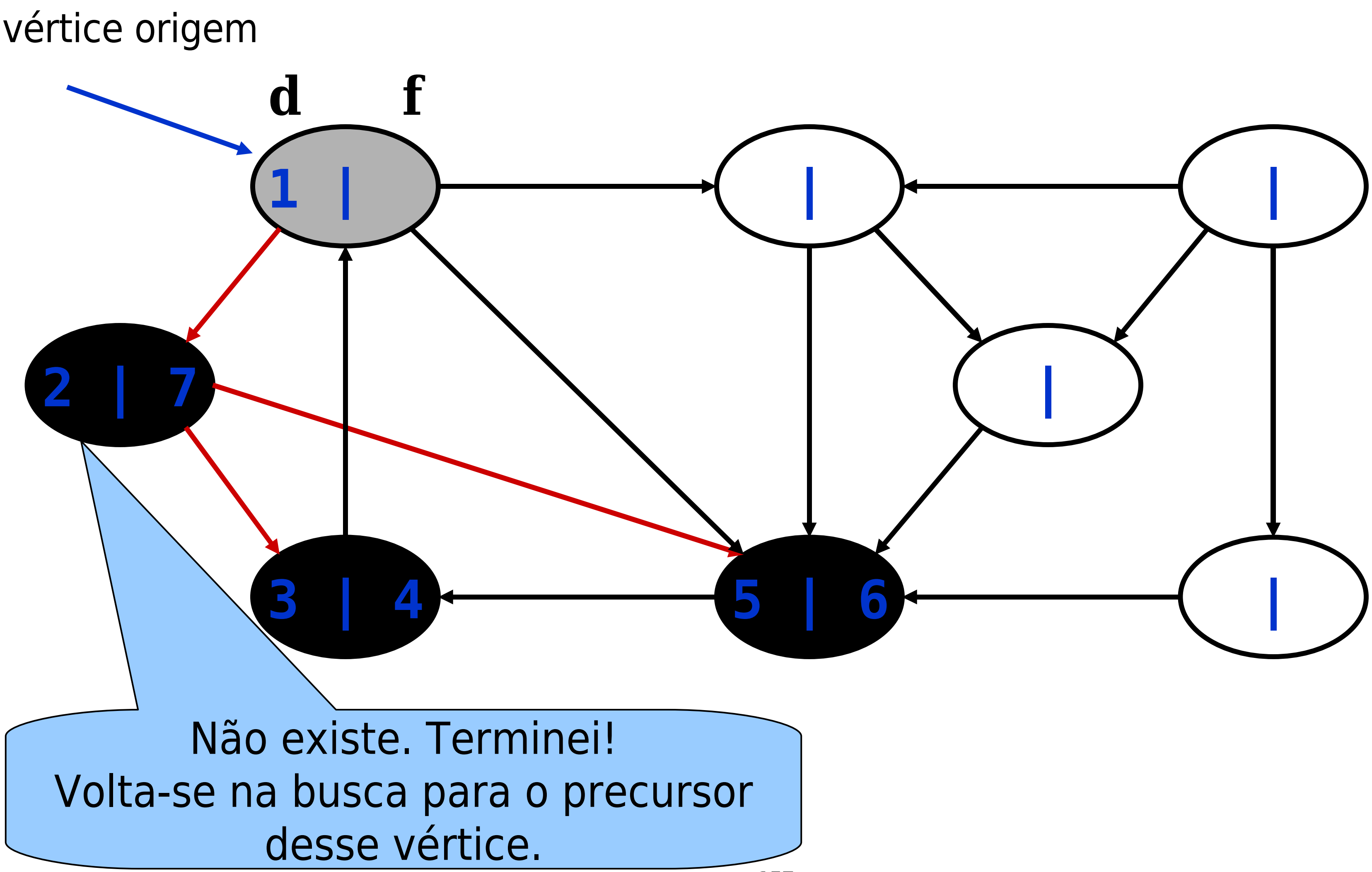
Busca em profundidade (DFS depth-first-search)

vértice origem



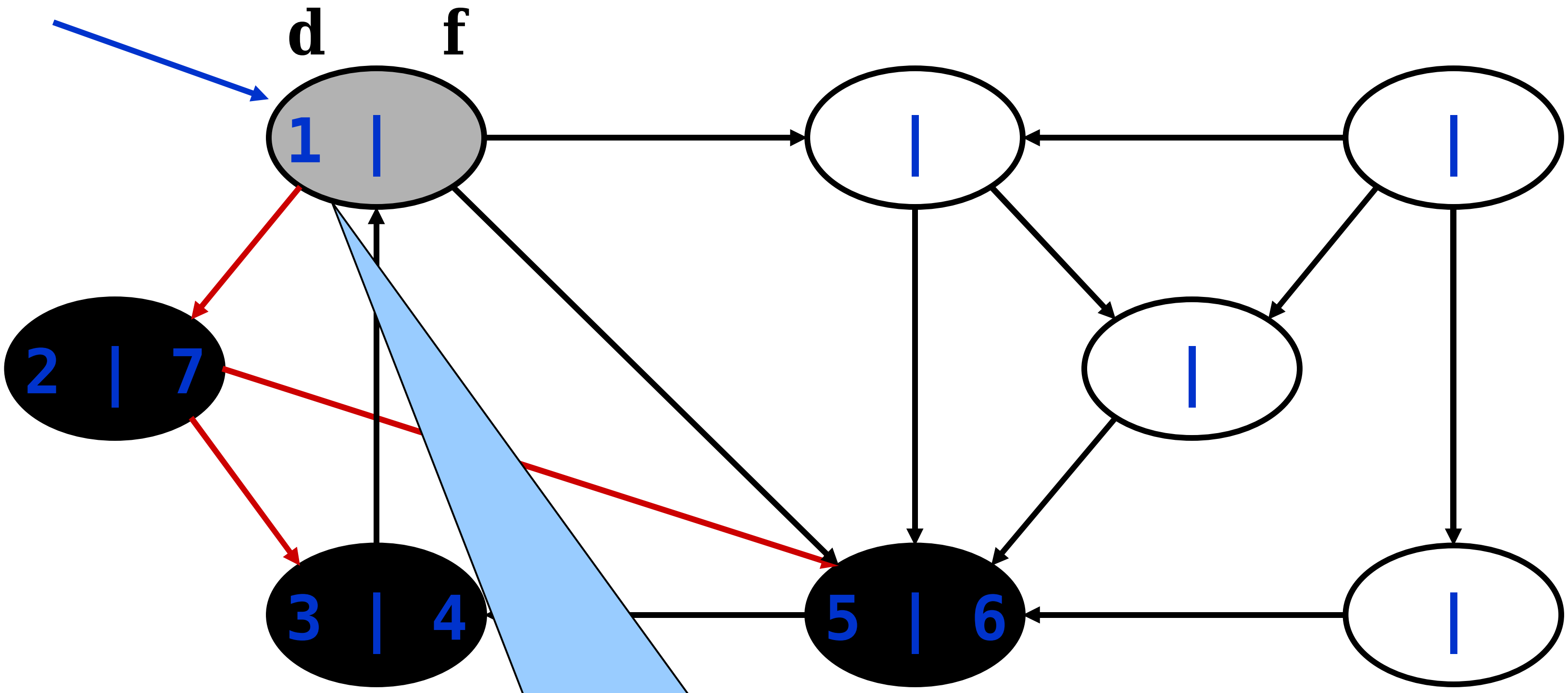
Existe algum vértice adjacente ao vértice que não tenha sido descoberto?

Busca em profundidade (DFS depth-first-search)



Busca em profundidade (DFS depth-first-search)

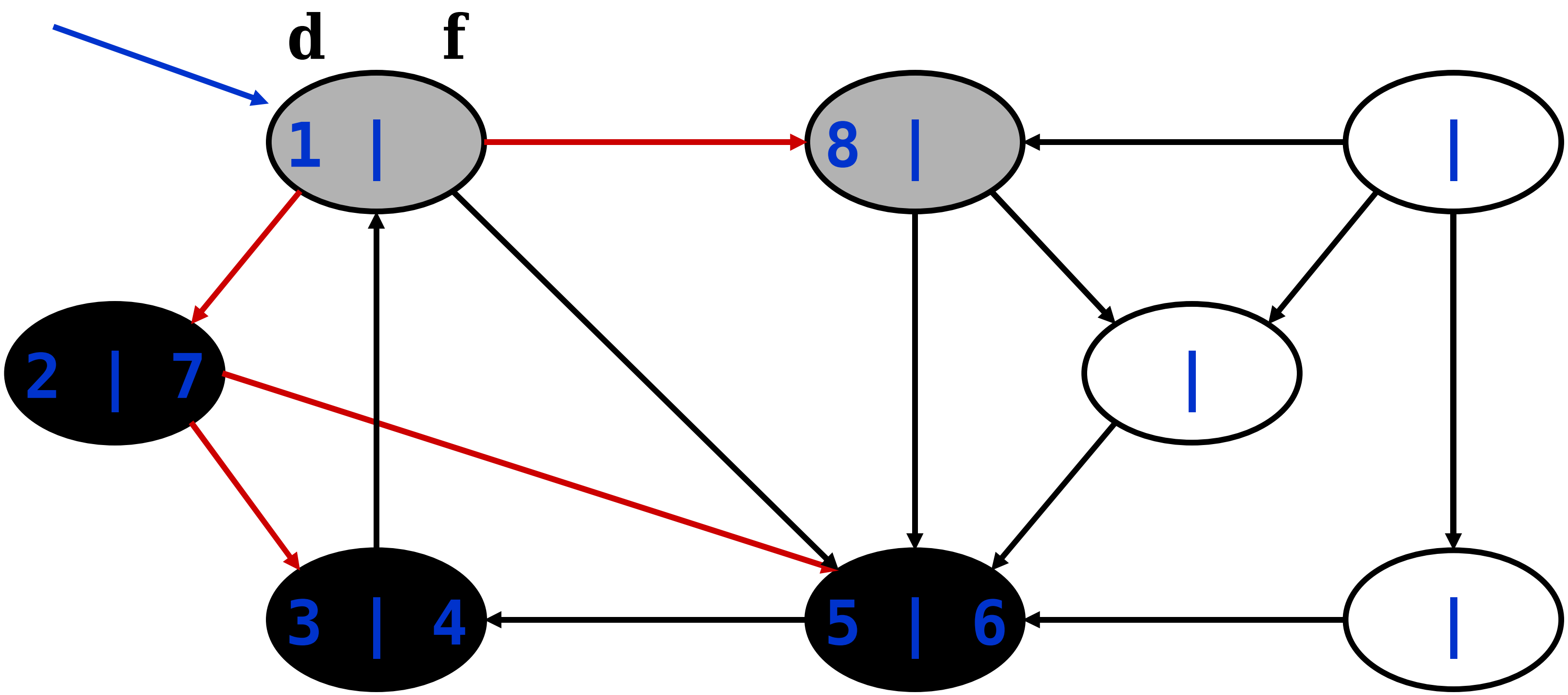
vértice origem



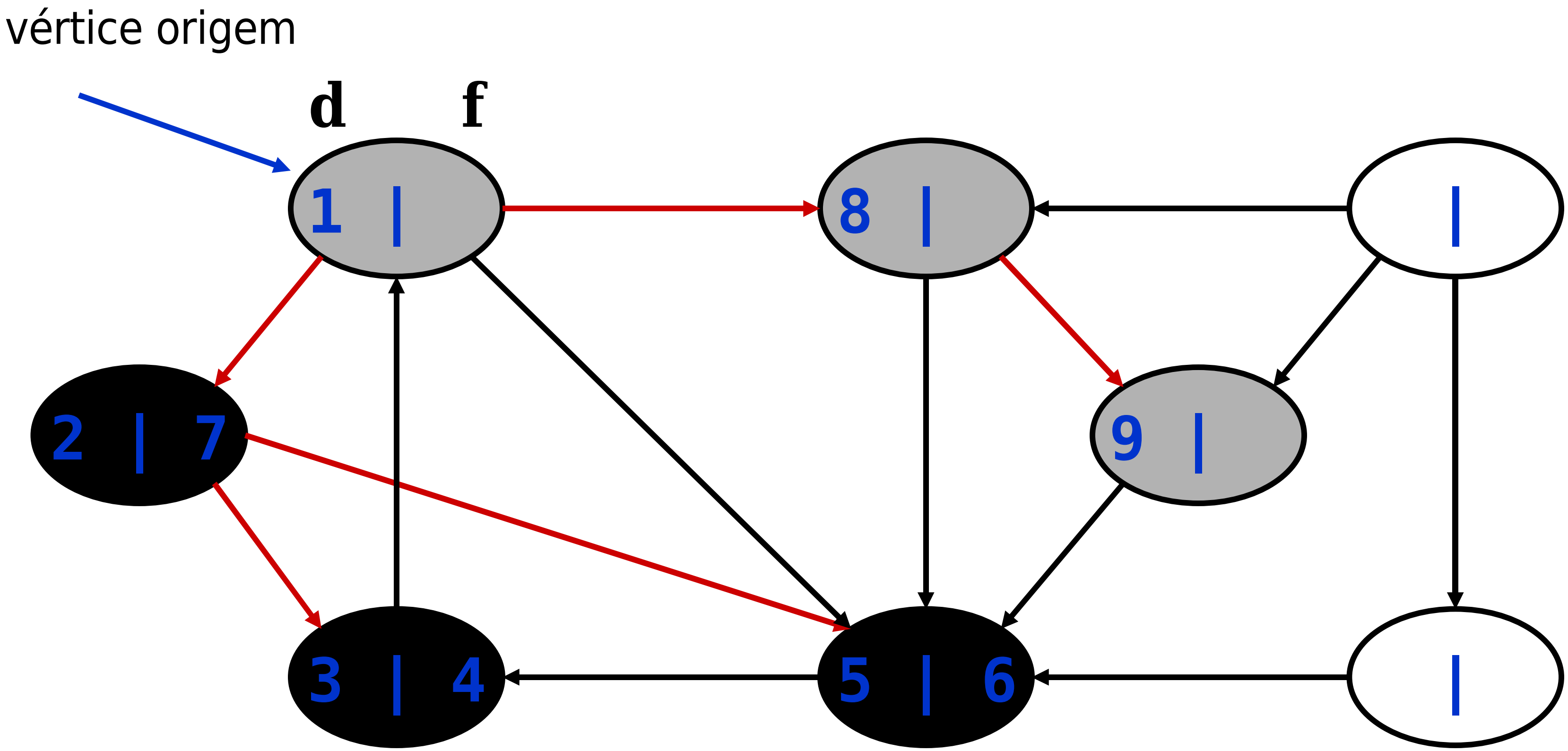
Existe mais algum vértice adjacente ao vértice que não tenha sido descoberto?

Busca em profundidade (DFS depth-first-search)

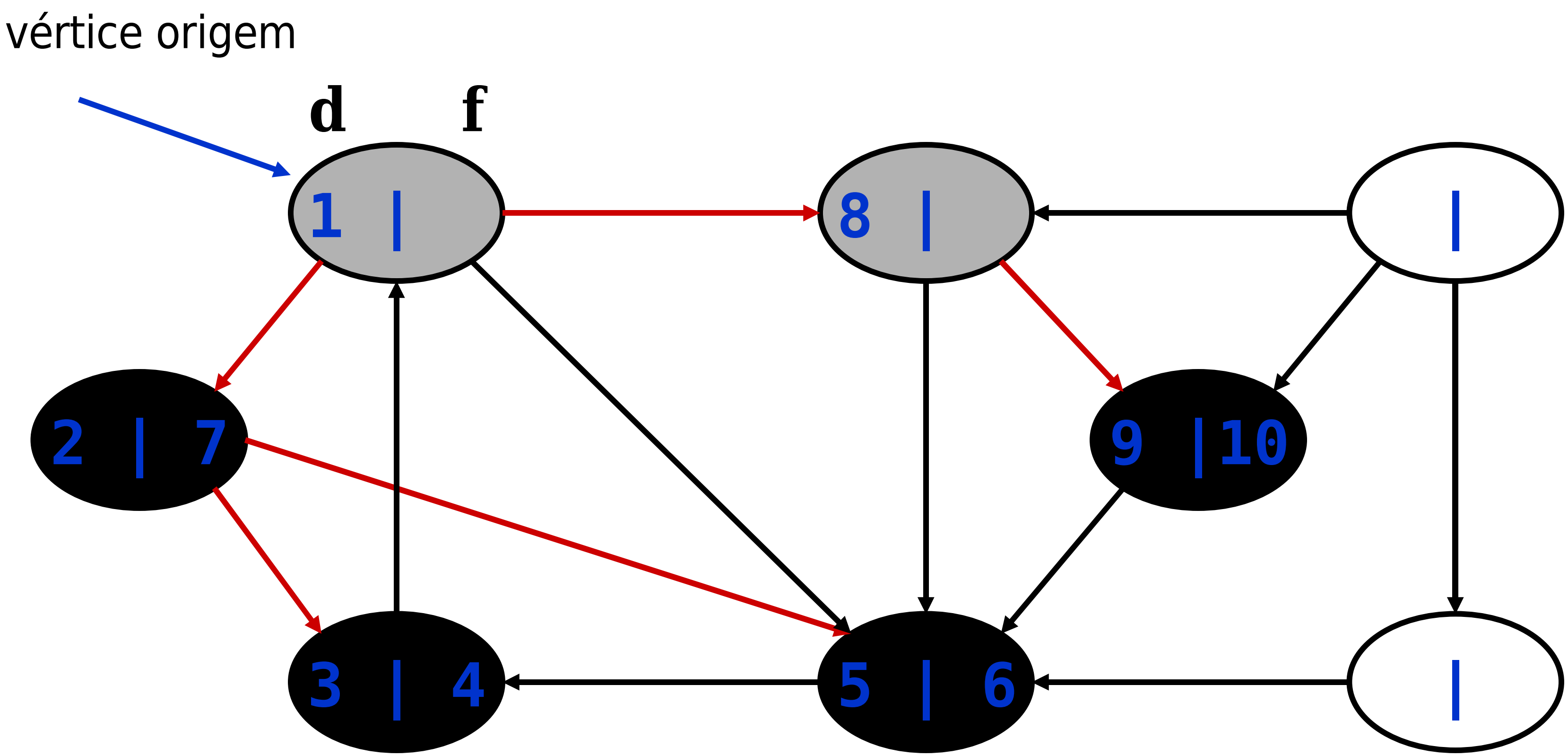
vértice origem



Busca em profundidade (DFS depth-first-search)

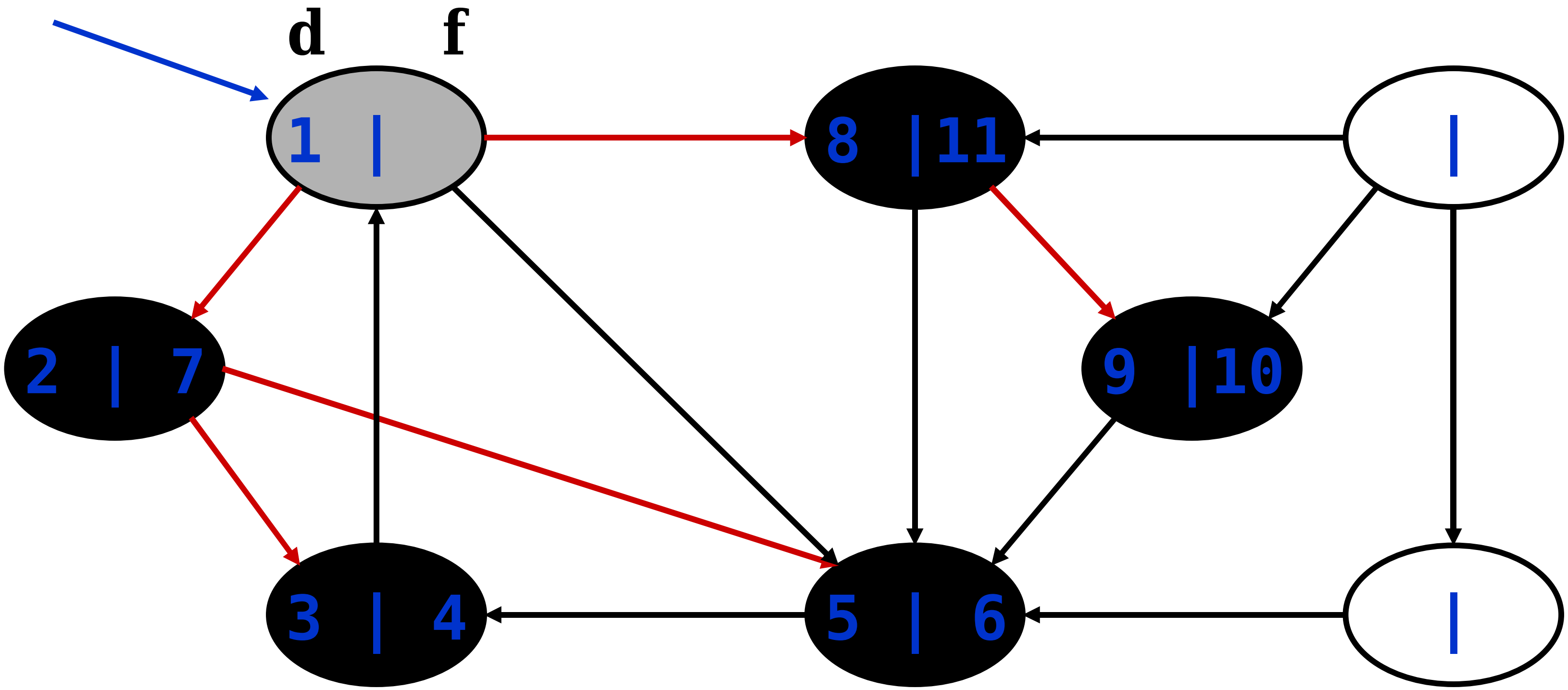


Busca em profundidade (DFS depth-first-search)

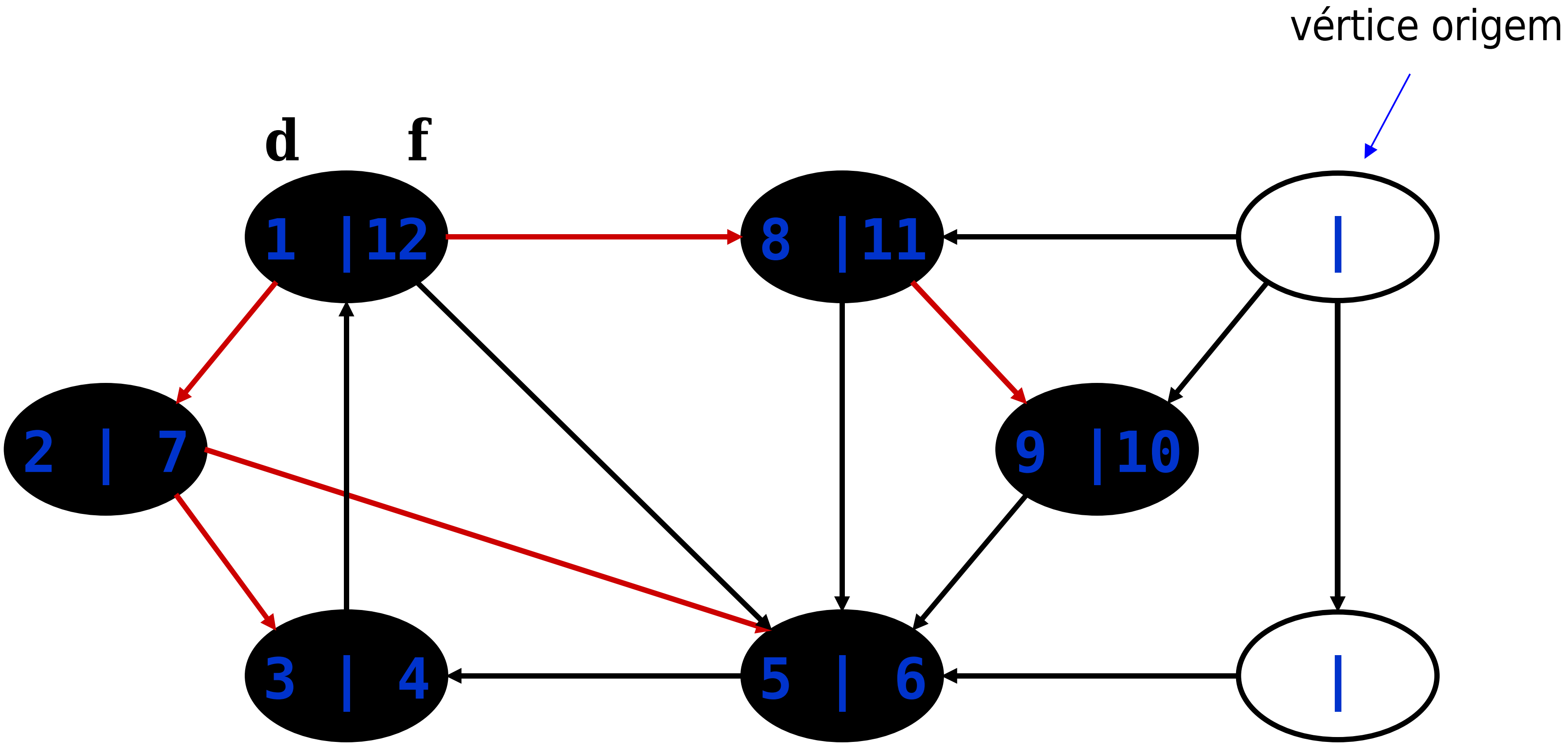


Busca em profundidade (DFS depth-first-search)

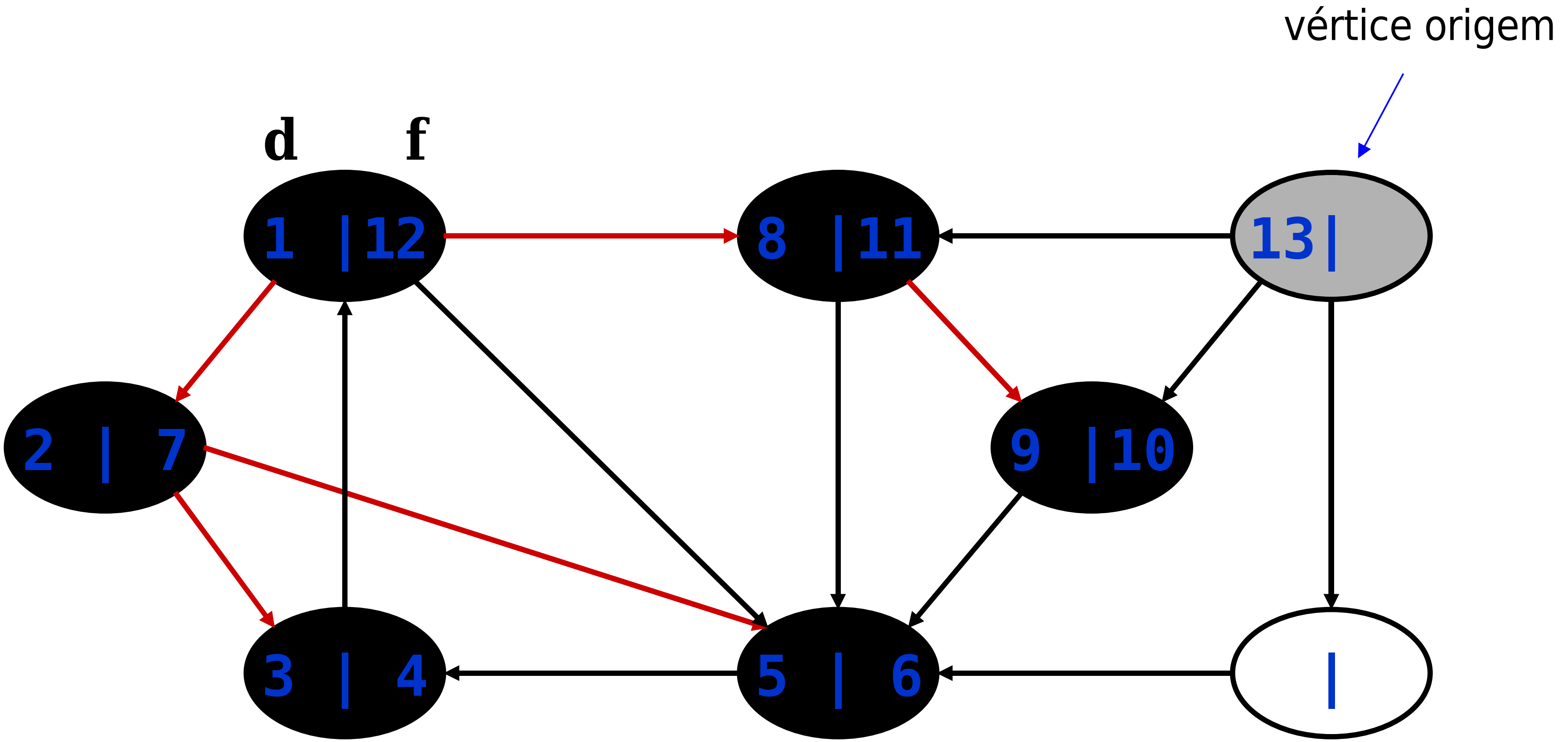
vértice origem



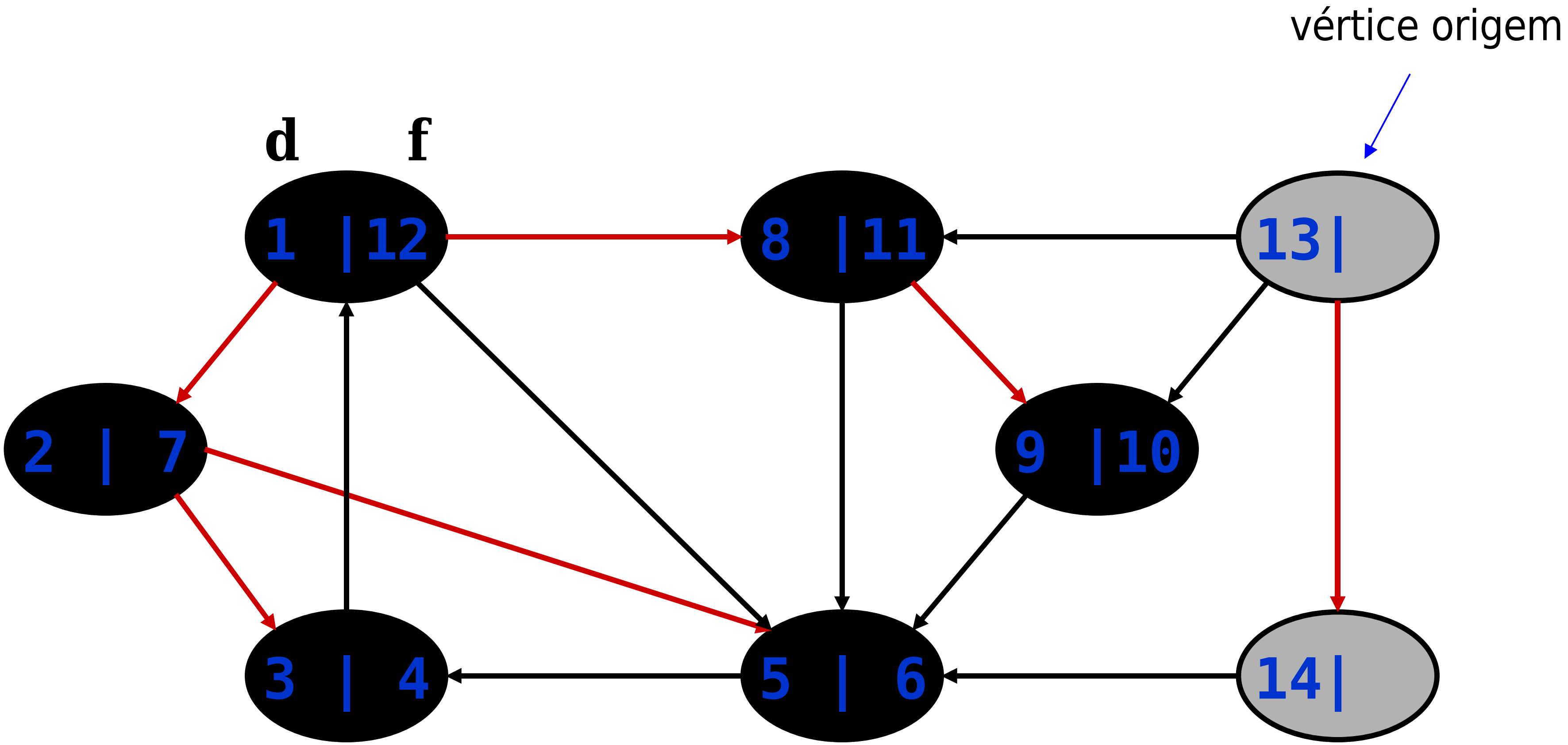
Busca em profundidade (DFS depth-first-search)



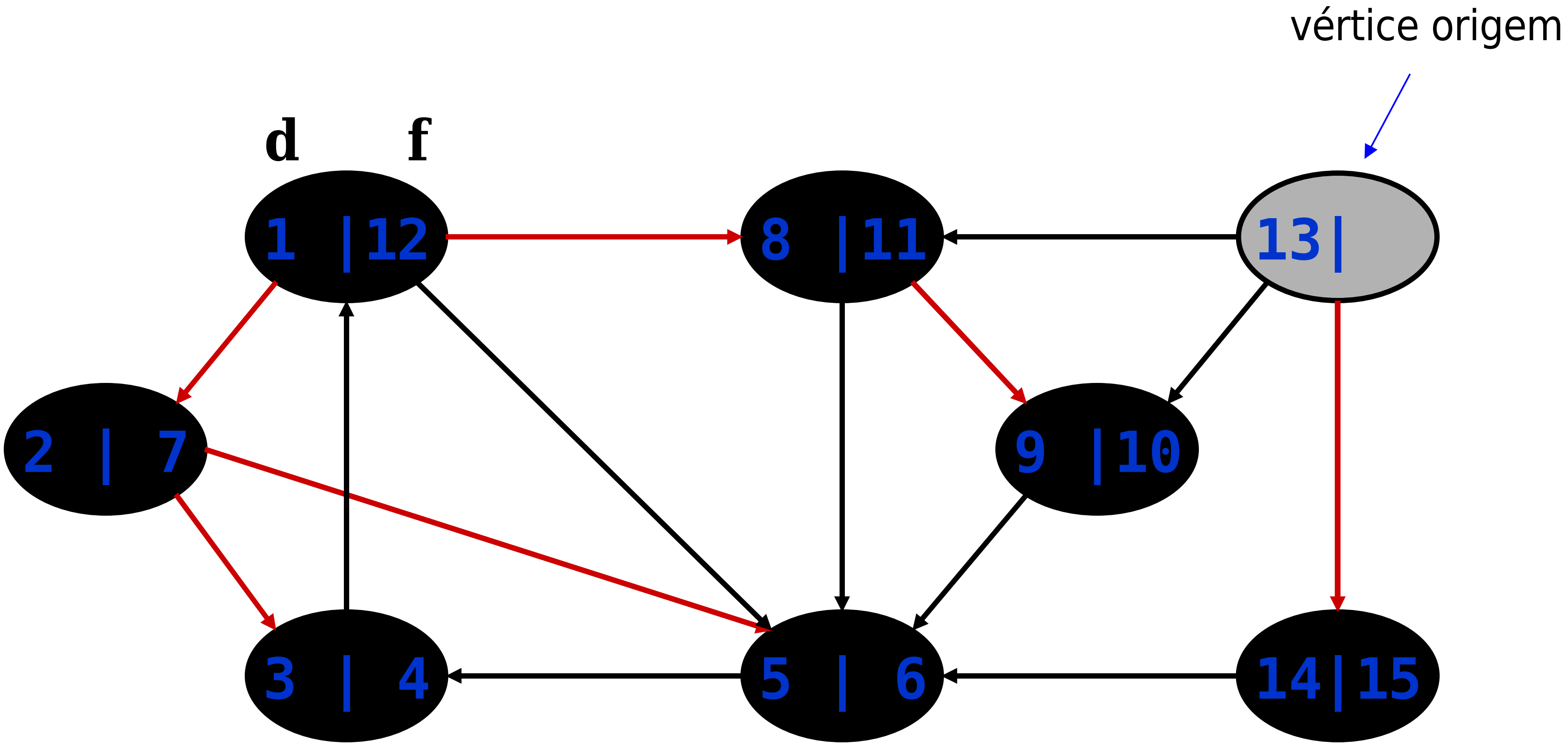
Busca em profundidade (DFS depth-first-search)



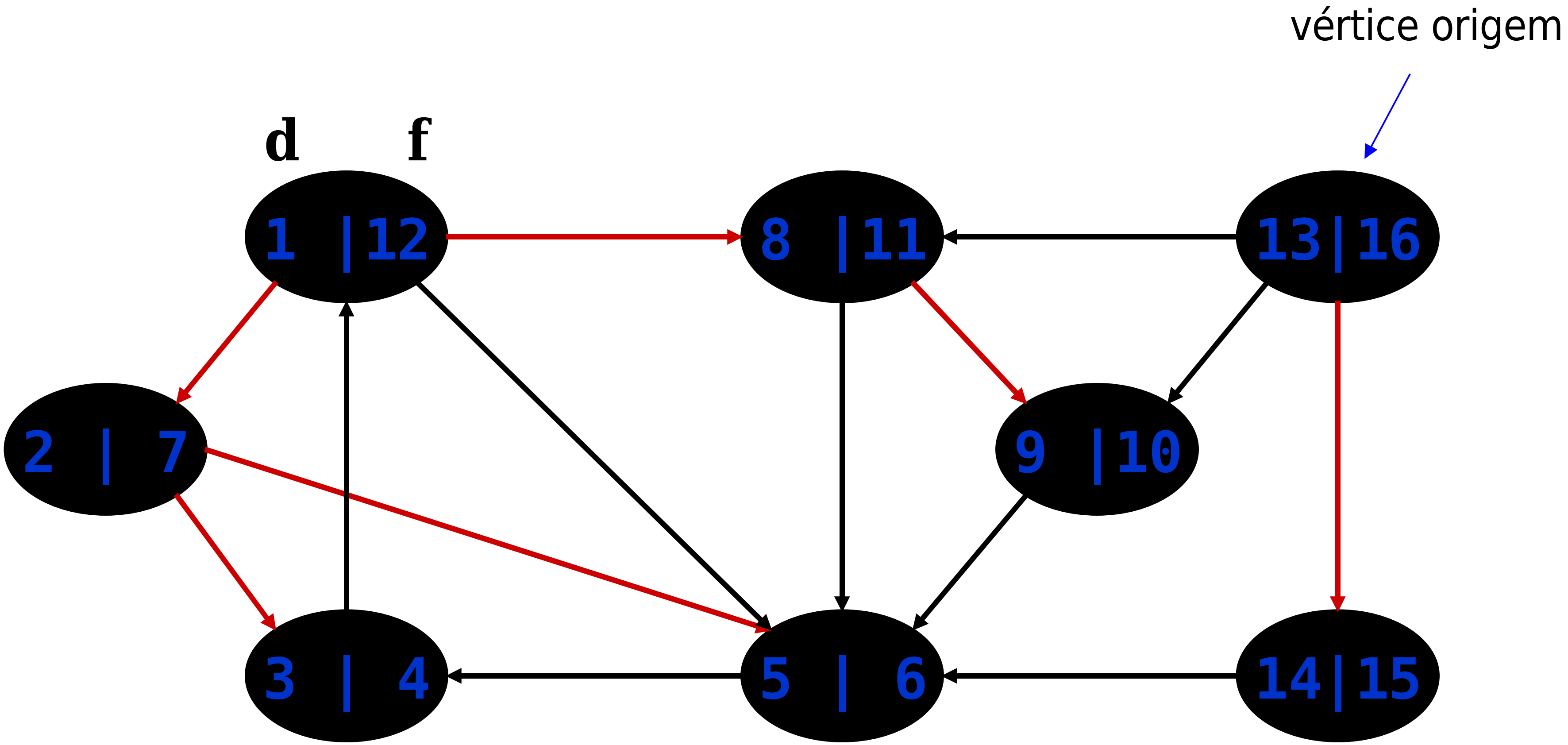
Busca em profundidade (DFS depth-first-search)



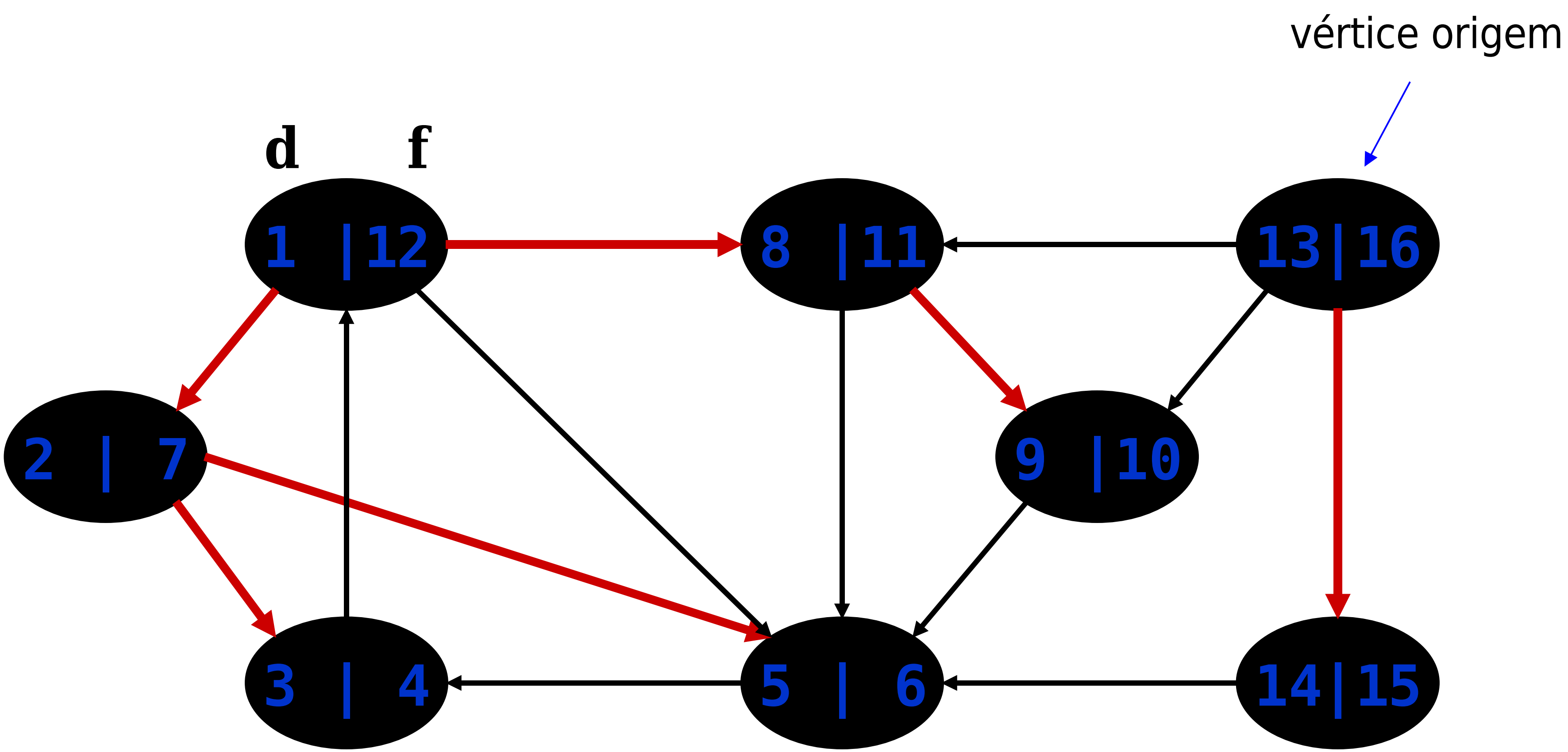
Busca em profundidade (DFS depth-first-search)



Busca em profundidade (DFS depth-first-search)

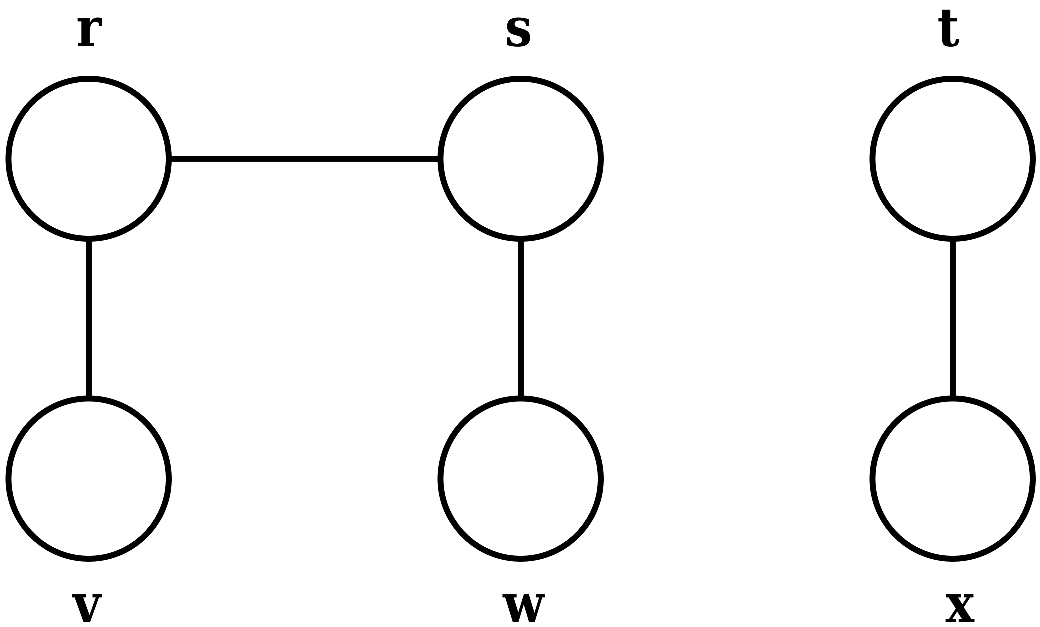


Busca em profundidade (DFS depth-first-search)



Resultado: uma floresta com 2 árvores, uma com 6 vértices e outra com 2 vértices

Busca em profundidade (DFS depth-first-search)



Aplicar o algoritmo DFS

Busca em profundidade (DFS depth-first-search)

Escrever o algoritmo recursivo **DFS-Visit(u),** que visita todos os vértices em profundidade a partir da raiz u.

Suponha que existe uma variável *time* global.

Busca em profundidade (DFS depth-first-search)

DFS-Visit(u)

```
color[u] ← GRAY                        ▷ vértice u acabou de ser descoberto
time ← time + 1
d[u] ← time
for each vertex v adjacent to u
        if color[v] = WHITE
        then π[v] ← u
                DFS-Visit(v)
color[u] ← BLACK
time ← time + 1
f[u] ← time
```

Busca em profundidade (DFS depth-first-search)

```
DFS-Visit(u)
color[u] ← GRAY                           ▷ vértice u acabou de ser descoberto
time ← time + 1
d[u] ← time
for each vertex v adjacent to u           ▷ explora a aresta (u, v)
        if color[v] = WHITE
        then π[v] ← u
                DFS-Visit(v)
color[u] ← BLACK
time ← time + 1
f[u] ← time
```

Busca em profundidade (DFS depth-first-search)

DFS-Visit(u)

```
color[u] ← GRAY                          ▷ vértice u acabou de ser descoberto
time ← time + 1
d[u] ← time
for each vertex v adjacent to u          ▷ explora a aresta (u, v)
        if color[v] = WHITE
        then π[v] ← u
                DFS-Visit(v)
color[u] ← BLACK                         ▷ os vizinhos de u já foram examinados
time ← time + 1
f[u] ← time                              ▷ terminamos com o vértice u
```

Busca em profundidade (DFS depth-first-search)

DFS-Visit(u)

1. color[u] ← GRAY ▷ vértice u acabou de ser descoberto
2. time ← time + 1
3. d[u] ← time
4. **for each** vertex v adjacent to u
5. **if** color[v] = WHITE
6. **then** π[v] ← u
7. DFS-Visit(v)
8. color[u] ← BLACK ▷ os v
9. time ← time + 1
10. f[u] ← time ▷ terminamos com o vértice u

DFS-Visit cria uma árvore DFS com raiz em u

Busca em profundidade (DFS depth-first-search)

E agora como criar a floresta?

```
DFS-Visit(u)
color[u] ← GRAY                 ▷ vértice u acabou de ser descoberto
time ← time + 1
d[u] ← time
for each vertex v adjacent to u
        if color[v] = WHITE
        then π[v] ← u
              DFS-Visit(v)
color[u] ← BLACK                ▷ os v
time ← time + 1
f[u] ← time                     ▷ terminamos com o vértice u
```

DFS-Visit cria uma árvore DFS com raiz em u

Busca em profundidade (DFS depth-first-search)

```
DFS (V, A)
for each vertex u in V                ▷inicialização
        color[u] ← WHITE
        π[u] ← NIL
time ← 0
for each vertex u in V
        if color[u] = WHITE
        then DFS-Visit(u)

DFS-Visit(u)
color[u] ← GRAY
time ← time + 1
d[u] ← time
for each vertex v adjacent to u
        if color[v] = WHITE
        then π[v] ← u
                DFS-Visit(v)
color[u] ← BLACK
time ← time + 1
f[u] ← time
```

Busca em profundidade (DFS depth-first-search)

```
DFS (V, A)
for each vertex u in V                          ▷inicialização
        color[u] ← WHITE
        π[u] ← NIL
time ← 0
for each vertex u in V
        if color[u] = WHITE
        then DFS-Visit(u)
```

Cria a floresta

```
DFS-Visit(u)
color[u] ← GRAY
time ← time + 1
d[u] ← time
for each vertex v adjacent to u
        if color[v] = WHITE
        then π[v] ← u
                DFS-Visit(v)
color[u] ← BLACK
time ← time + 1
f[u] ← time
```

Busca em profundidade (DFS depth-first-search)

```
DFS (V, A)
for each vertex u in V                    ▷inicialização
        color[u] ← WHITE
        π[u] ← NIL
time ← 0
for each vertex u in V
        if color[u] = WHITE
        then DFS-Visit(u)                 ▷ criar uma nova árvore a partir de u
```

```
DFS-Visit(u)
color[u] ← GRAY
time ← time + 1
d[u] ← time
for each vertex v adjacent to u
        if color[v] = WHITE
        then π[v] ← u
                DFS-Visit(v)
color[u] ← BLACK
time ← time + 1
f[u] ← time
```

Busca em profundidade (DFS depth-first-search)

```
DFS (V, A)
    for each vertex u in V                ▷inicialização
        color[u] ← WHITE
        π[u] ← NIL
    time ← 0
    for each vertex u in V
        if color[u] = WHITE
        then DFS-Visit(u)                 ▷ criar uma nova árvore a partir de u

DFS-Visit(u)
    color[u] ← GRAY
    time ← time + 1
    d[u] ← time
    for each vertex v adjacent to u
        if color[v] = WHITE
        then π[v] ← u
             DFS-Visit(v)
    color[u] ← BLACK
    time ← time + 1
    f[u] ← time
```

u recebe um tempo de descoberta d[u] e um tempo de término f[u] durante a execuçã de DFS-Visit(u)

Busca em profundidade (DFS depth-first-search)

```
DFS (V, A)
for each vertex u in V                 ▷inicialização
        color[u] ← WHITE
        π[u] ← NIL
time ← 0
for each vertex u in V
        if color[u] = WHITE
        then DFS-Visit(u)

DFS-Visit(u)
color[u] ← GRAY
time ← time + 1
d[u] ← time
for each vertex v adjacent to u
        if color[v] = WHITE
        then π[v] ← u
                DFS-Visit(v)
color[u] ← BLACK
time ← time + 1
f[u] ← time
```

Qual o tempo de execução de DFS?

Busca em profundidade (DFS depth-first-search)

```
DFS (V, A)
    for each vertex u in V
        color[u] ← WHITE
        π[u] ← NIL
    time ← 0
    for each vertex u in V
        if color[u] = WHITE
        then DFS-Visit(u)          ▷ criar uma nova árvore a partir de u
```

O(V)

```
DFS-Visit(u)
    color[u] ← GRAY
    time ← time + 1
    d[u] ← time
    for each vertex v adjacent to u
        if color[v] = WHITE
        then π[v] ← u
            DFS-Visit(v)
    color[u] ← BLACK
    time ← time + 1
    f[u] ← time
```

Busca em profundidade (DFS depth-first-search)

```
DFS (V, A)
for each vertex u in V                  ▷inicialização
        color[u] ← WHITE
        π[u] ← NIL
time ← 0
for each vertex u in V
        if color[u] = WHITE
        then DFS-Visit(u)               ▷ criar uma nova árvore a partir de u

DFS-Visit(u)
color[u] ← GRAY
time ← time + 1
d[u] ← time
for each vertex v adjacent to u
        if color[v] = WHITE
        then π[v] ← u
                DFS-Visit(v)
color[u] ← BLACK
time ← time + 1
f[u] ← time
```

Quantas vezes é chamado DFS-Visit para cada vértice?

Busca em profundidade (DFS depth-first-search)

DFS (V, A)

1. **for each** vertex u in V ▷inicialização
2. color[u] ← WHITE
3. π[u] ← NIL
4. time ← 0
5. **for each** vertex u in V
6. **if** color[u] = WHITE
7. **then** DFS-Visit(u) ▷ criar uma nova árvore a partir de u

DFS-Visit(u)

1. color[u] ← GRAY
2. time ← time + 1
3. d[u] ← time
4. **for each** vertex v adjacent to u
5. **if** color[v] = WHITE
6. **then** π[v] ← u
7. DFS-Visit(v)
8. color[u] ← BLACK
9. time ← time + 1
10. f[u] ← time

DFS-Visit é chamado exatamente uma vez para cada vértice v, pois ele é executado somente com vértices brancos.

Busca em profundidade (DFS depth-first-search)

```
DFS (V, A)
for each vertex u in V                ▷inicialização
        color[u] ← WHITE
        π[u] ← NIL
time ← 0
for each vertex u in V
        if color[u] = WHITE
        then DFS-Visit(u)              ▷ criar um

DFS-Visit(u)
color[u] ← GRAY
time ← time + 1
d[u] ← time
for each vertex v adjacent to u
        if color[v] = WHITE
        then π[v] ← u
              DFS-Visit(v)
color[u] ← BLACK
time ← time + 1
f[u] ← time
```

O tempo gasto na varredura total das listas de adjacências é:

Busca em profundidade (DFS depth-first-search)

```
DFS (V, A)
    for each vertex u in V              ▷inicialização
            color[u] ← WHITE
            π[u] ← NIL
    time ← 0
    for each vertex u in V
            if color[u] = WHITE
            then DFS-Visit(u)           ▷ criar um

DFS-Visit(u)
    color[u] ← GRAY
    time ← time + 1
    d[u] ← time
    for each vertex v adjacent to u
            if color[v] = WHITE
            then π[v] ← u
                    DFS-Visit(v)
    color[u] ← BLACK
    time ← time + 1
    f[u] ← time
```

O tempo gasto na varredura total das listas de adjacências é: O(A)

Busca em profundidade (DFS depth-first-search)

O tempo total da busca em profundidade é $O(V+A)$

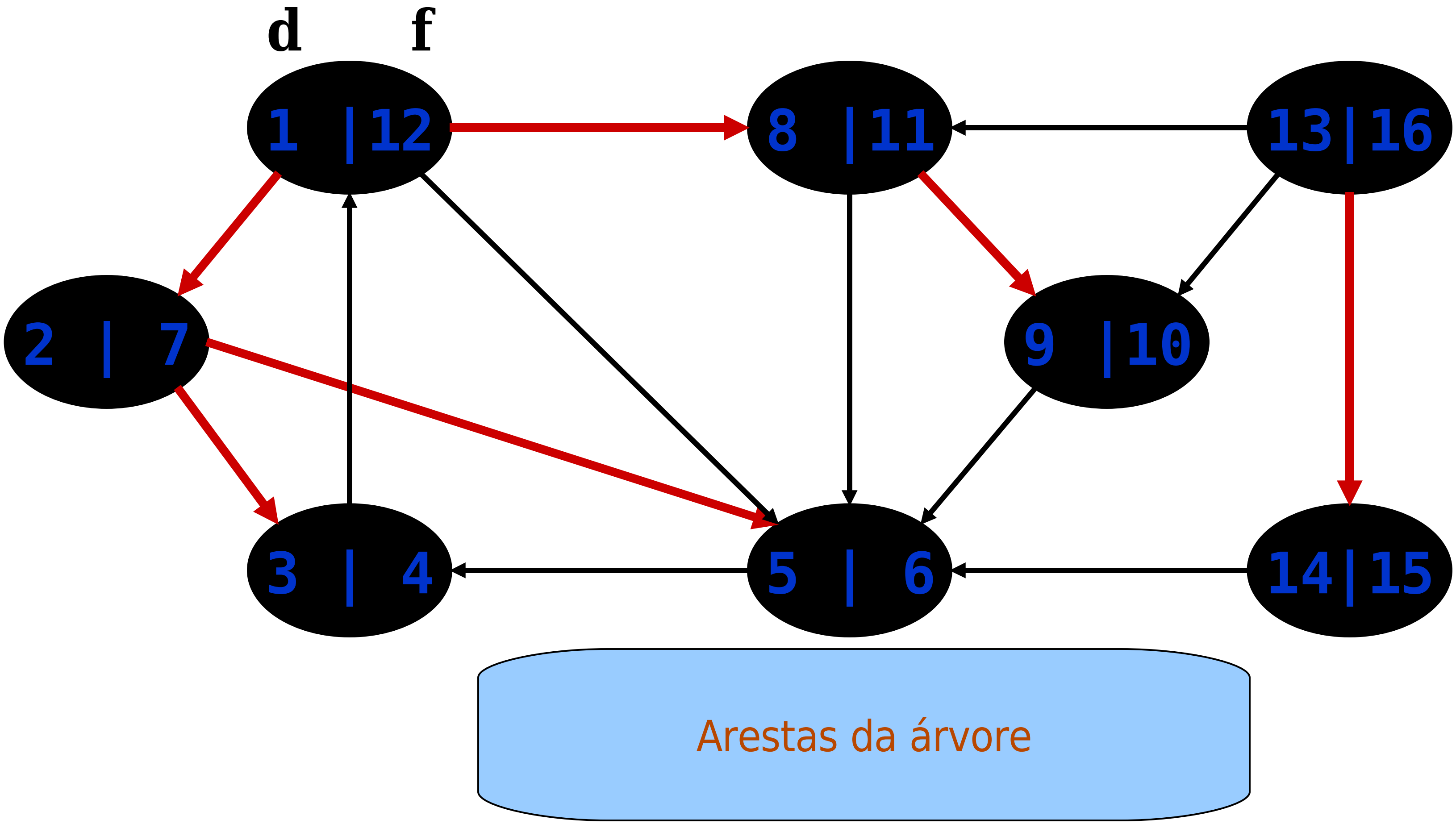
Busca em profundidade – Classificação das arestas

- Arestas da árvore
- Arestas de retorno
- Arestas de avanço
- Arestas de cruzamento

Busca em profundidade – Classificação das arestas

- Arestas da árvore: são arestas que pertencem a alguma das árvores DFS da floresta.
- Arestas de retorno
- Arestas de avanço
- Arestas de cruzamento

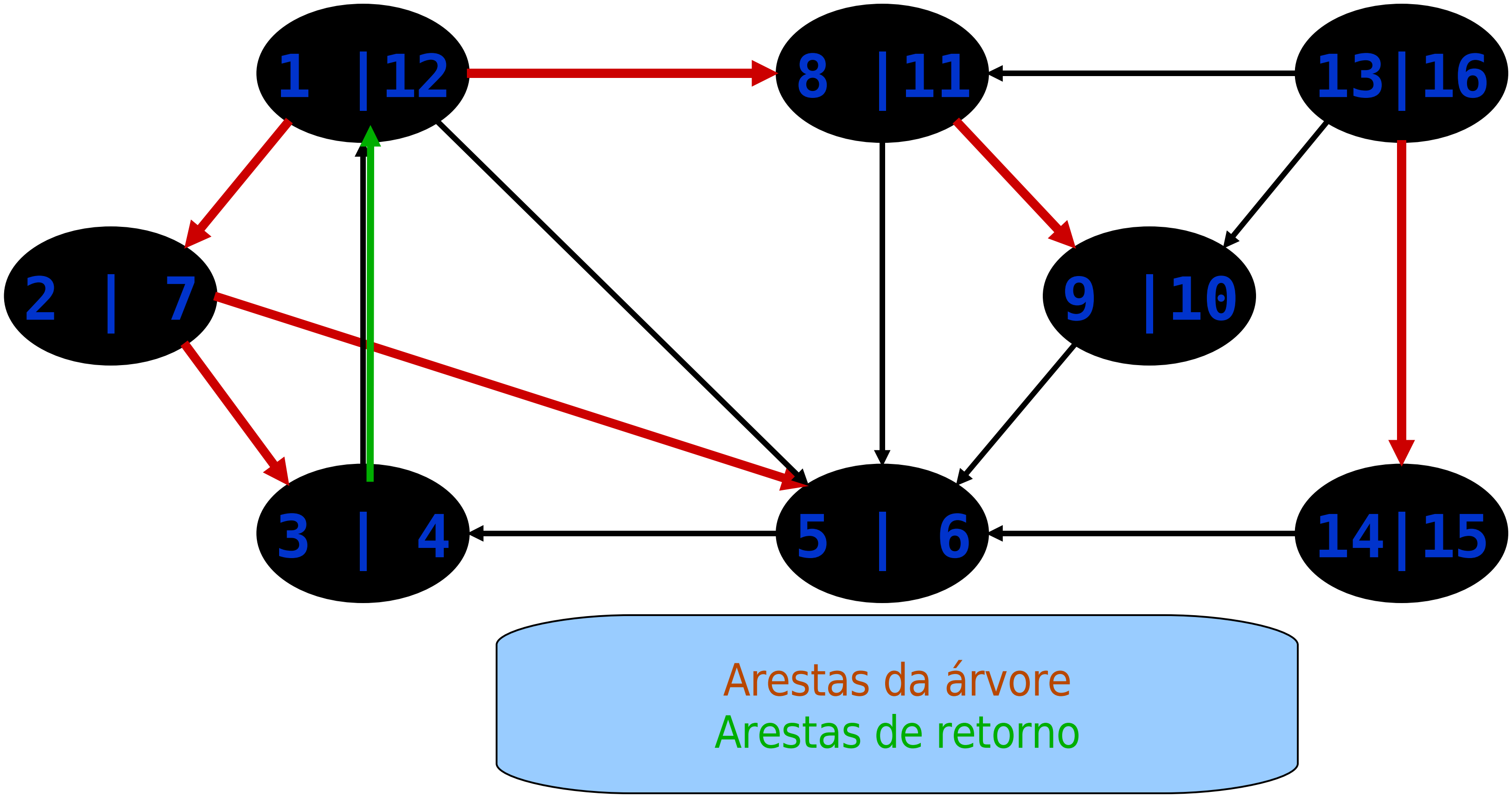
Busca em profundidade – Classificação das arestas



Busca em profundidade – Classificação das arestas

- Arestas da árvore: são arestas que pertencem a alguma das árvores DFS da floresta.
- Arestas de retorno: arestas (u,v) que não pertencem a árvore DFS. Conectam um vértice u com um ancestral v em uma árvore DFS. Self-loops são considerados arestas de retorno.
- Arestas de avanço
- Arestas de cruzamento

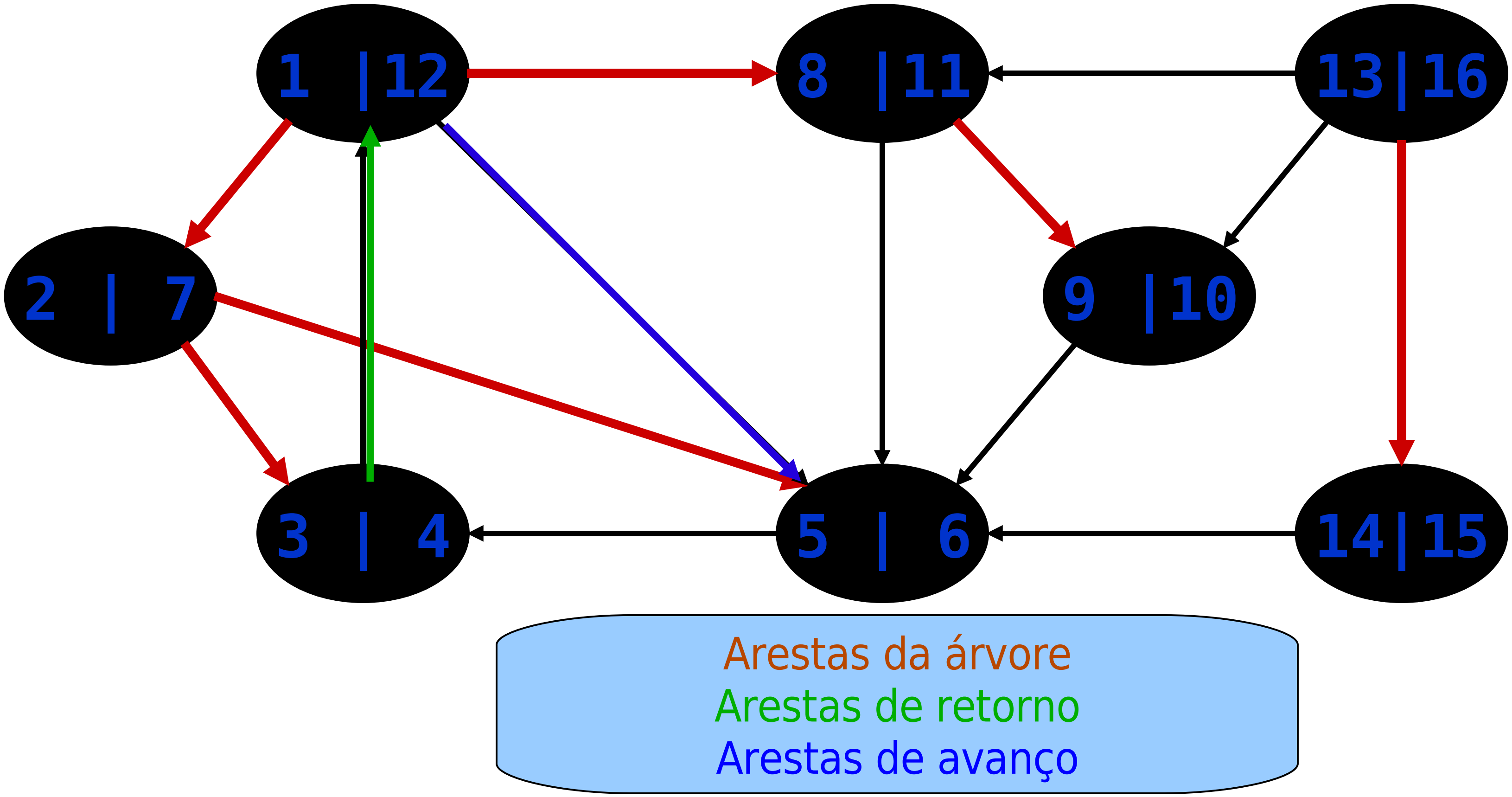
Busca em profundidade – Classificação das arestas



Busca em profundidade – Classificação das arestas

- Arestas da árvore: são arestas que pertencem a alguma das árvores DFS da floresta.
- Arestas de retorno: arestas (u,v) que não pertencem a árvore DFS. Conectam um vértice u com um ancestral v em uma árvore DFS. Self-loops são considerados arestas de retorno.
- Arestas de avanço: arestas (u,v) que não pertencem a árvore DFS. Conectam um vértice u a um descendente v em uma árvore DFS.
- Arestas de cruzamento

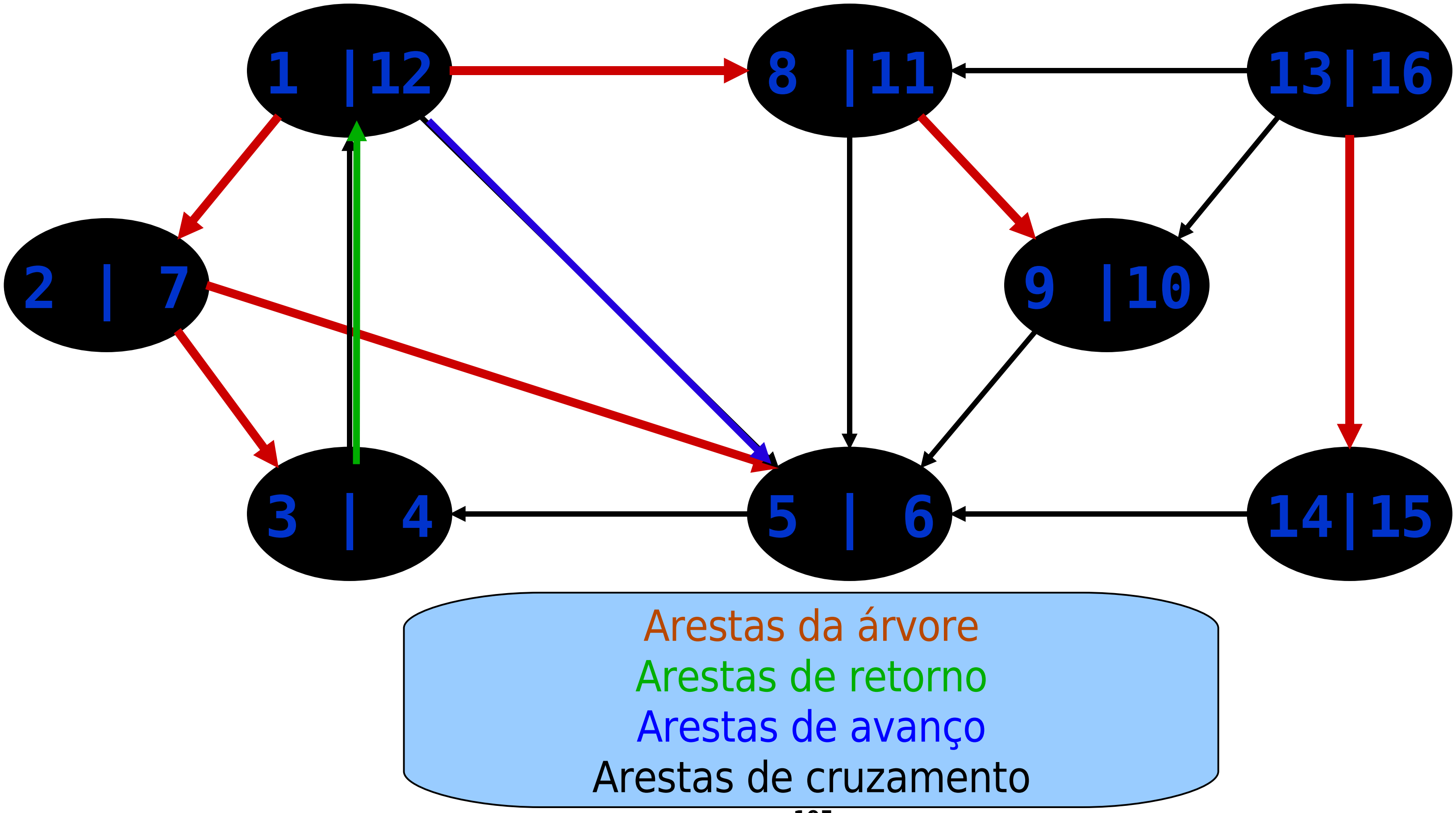
Busca em profundidade – Classificação das arestas



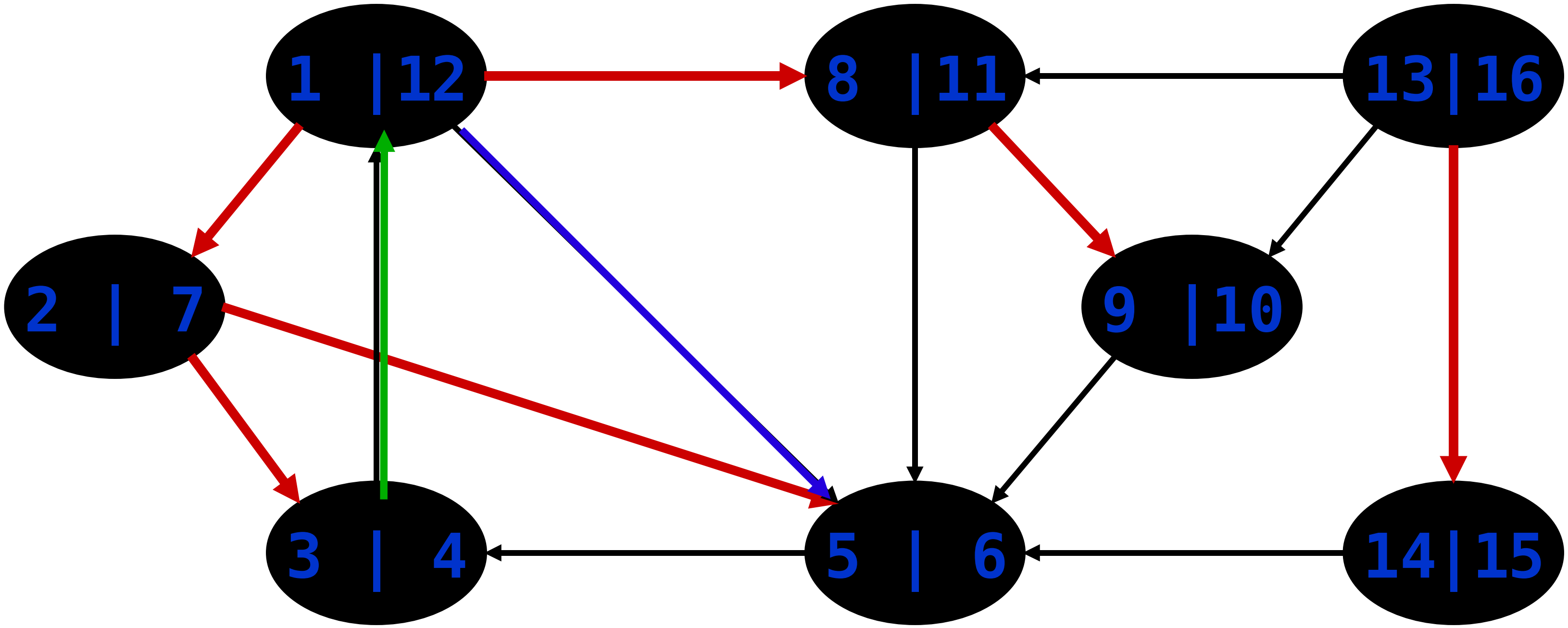
Busca em profundidade – Classificação das arestas

- Arestas da árvore: são arestas que pertencem a alguma das árvores DFS da floresta.
- Arestas de retorno: arestas (u,v) que não pertencem a árvore DFS. Conectam um vértice u com um ancestral v em uma árvore DFS. Self-loops são considerados arestas de retorno.
- Arestas de avanço: arestas (u,v) que não pertencem a árvore DFS. Conectam um vértice u a um descendente v em uma árvore DFS.
- Arestas de cruzamento: Todas as outras arestas.

Busca em profundidade – Classificação das arestas



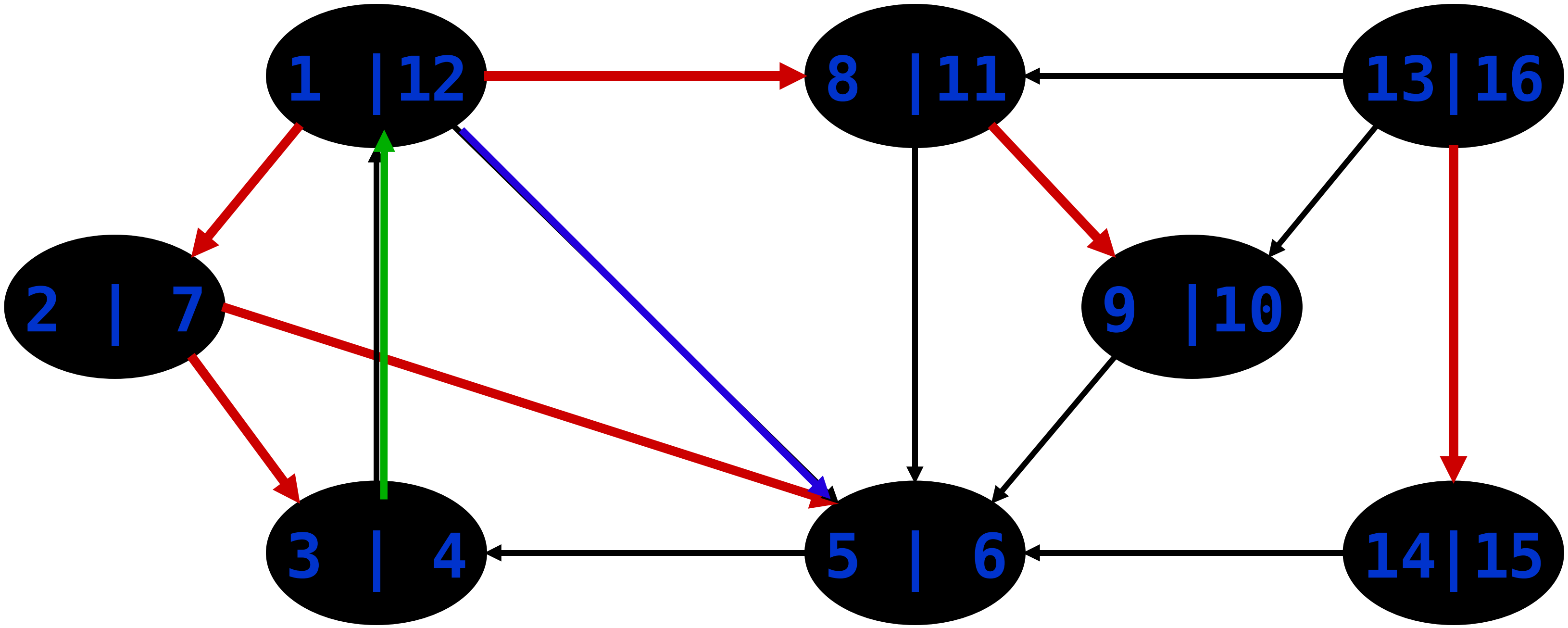
Busca em profundidade – Classificação das arestas



Arestas da árvore
Arestas de retorno
Arestas de avanço
Arestas de cruzamento

Podem as arestas de uma árvore formar ciclos?

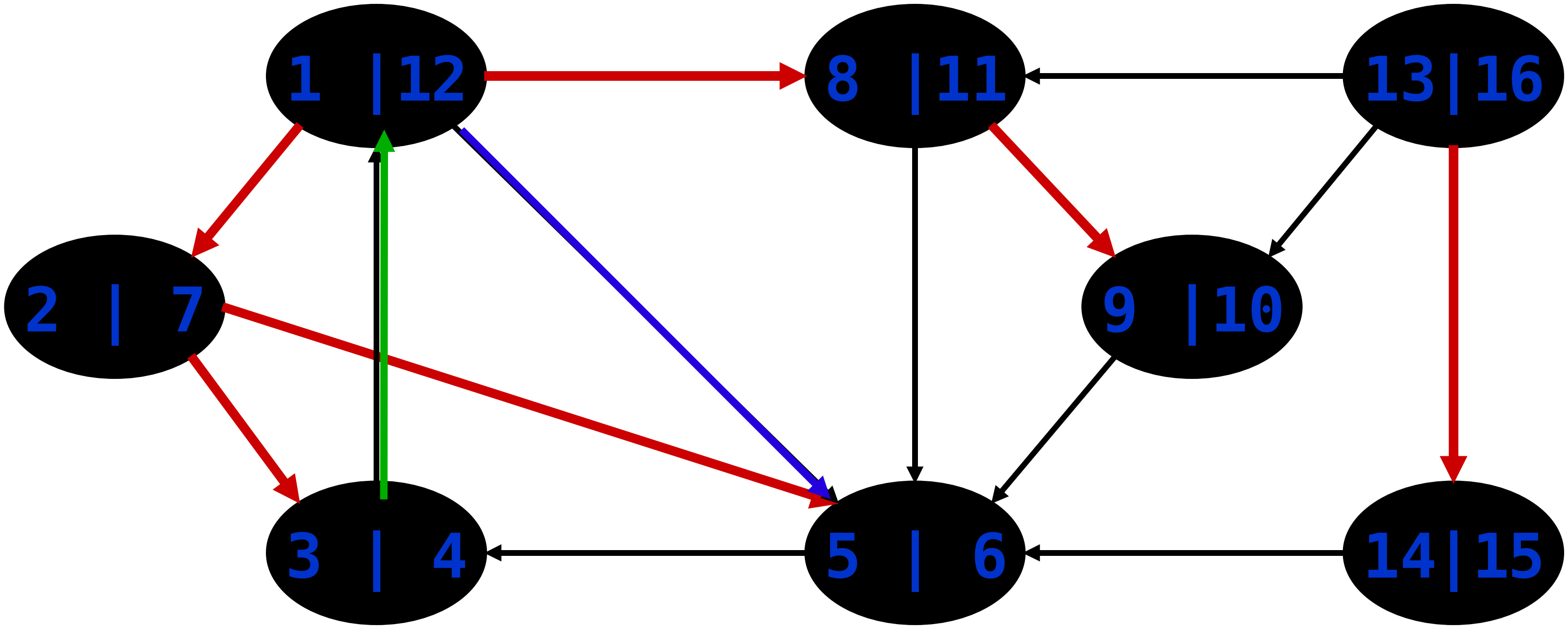
Busca em profundidade – Classificação das arestas



Arestas da árvore
Arestas de retorno
Arestas de avanço
Arestas de cruzamento

Podem as arestas de uma árvore formar ciclos? Não

Busca em profundidade – Classificação das arestas



Arestas da árvore
Arestas de retorno
Arestas de avanço
Arestas de cruzamento

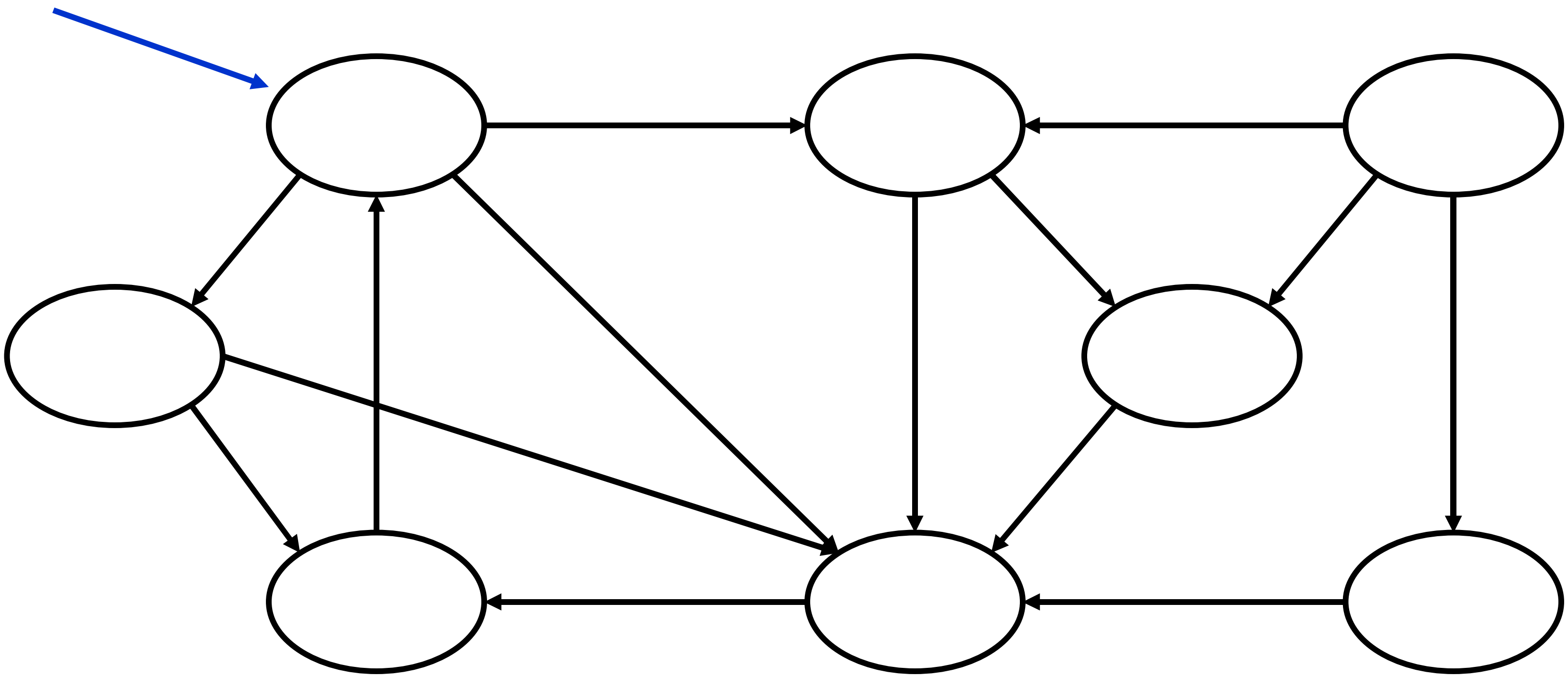
O que significa que existam aresta de retorno?

Busca em profundidade – Classificação das arestas

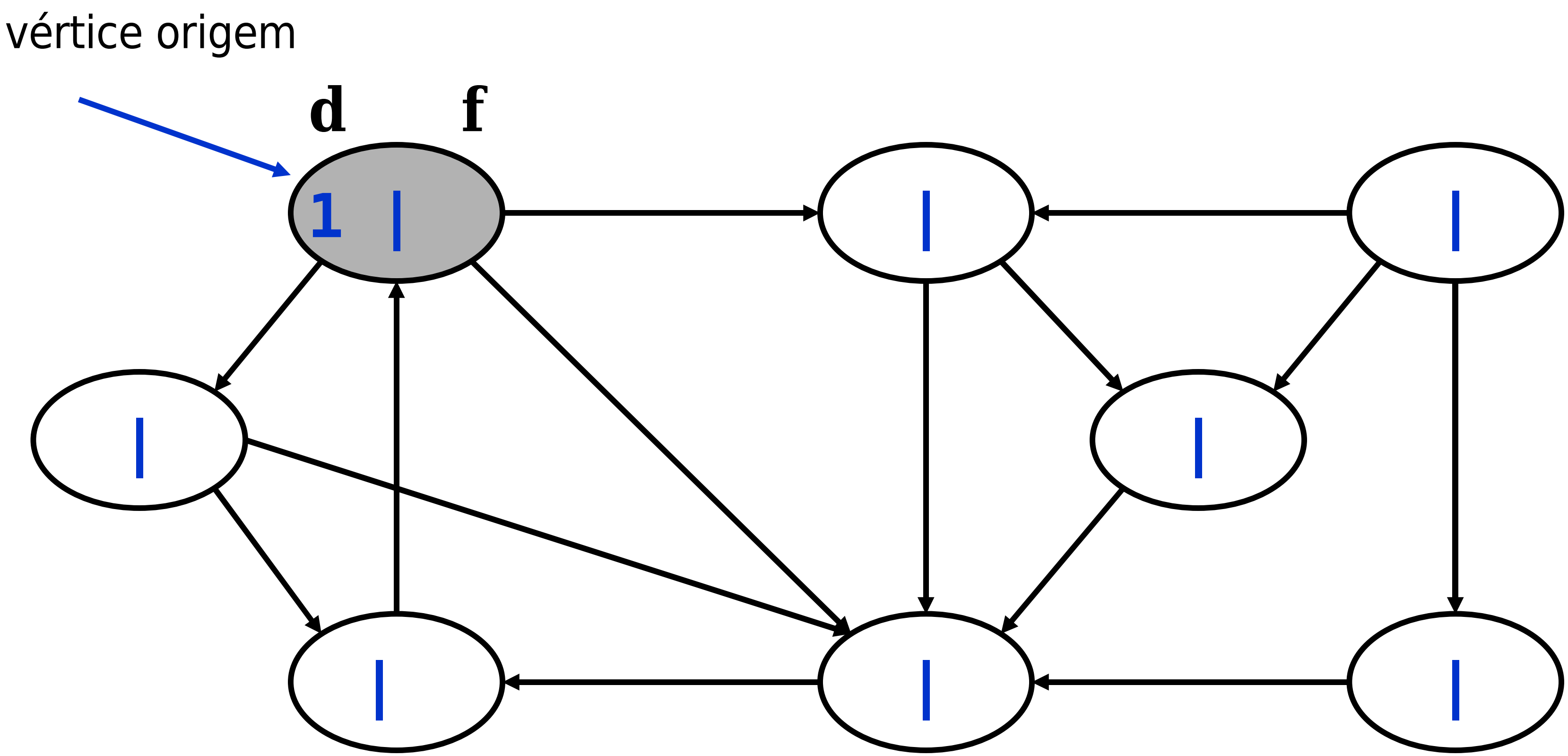
- O algoritmo DFS pode ser modificado para classificar arestas à medida que as encontra.
- Cada aresta (u,v) pode ser classificada pela cor do vértice v que é alcançado quando a aresta é explorada. Sabemos que:
  - Se v é **branco**, (u,v) é uma aresta da árvore

Busca em profundidade (DFS depth-first-search)

vértice origem

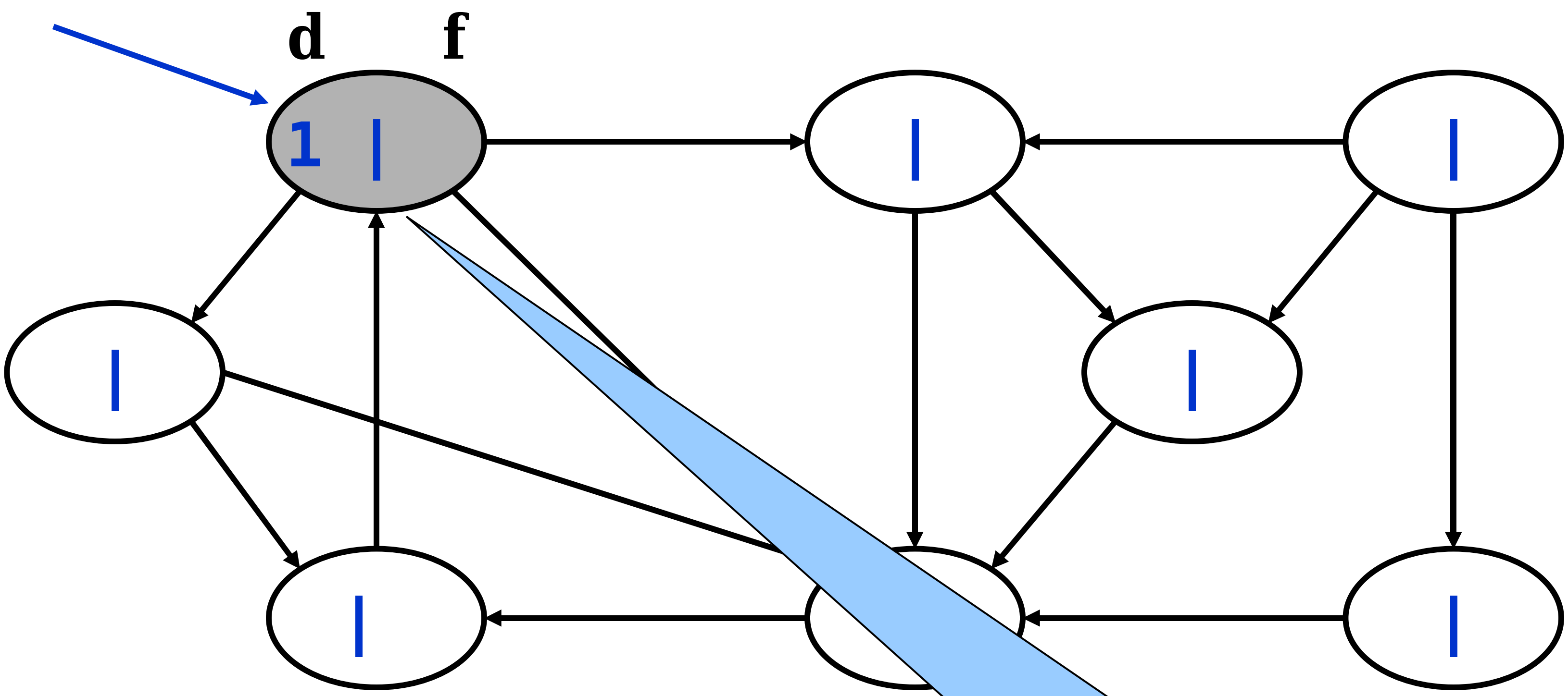


Busca em profundidade (DFS depth-first-search)



Busca em profundidade (DFS depth-first-search)

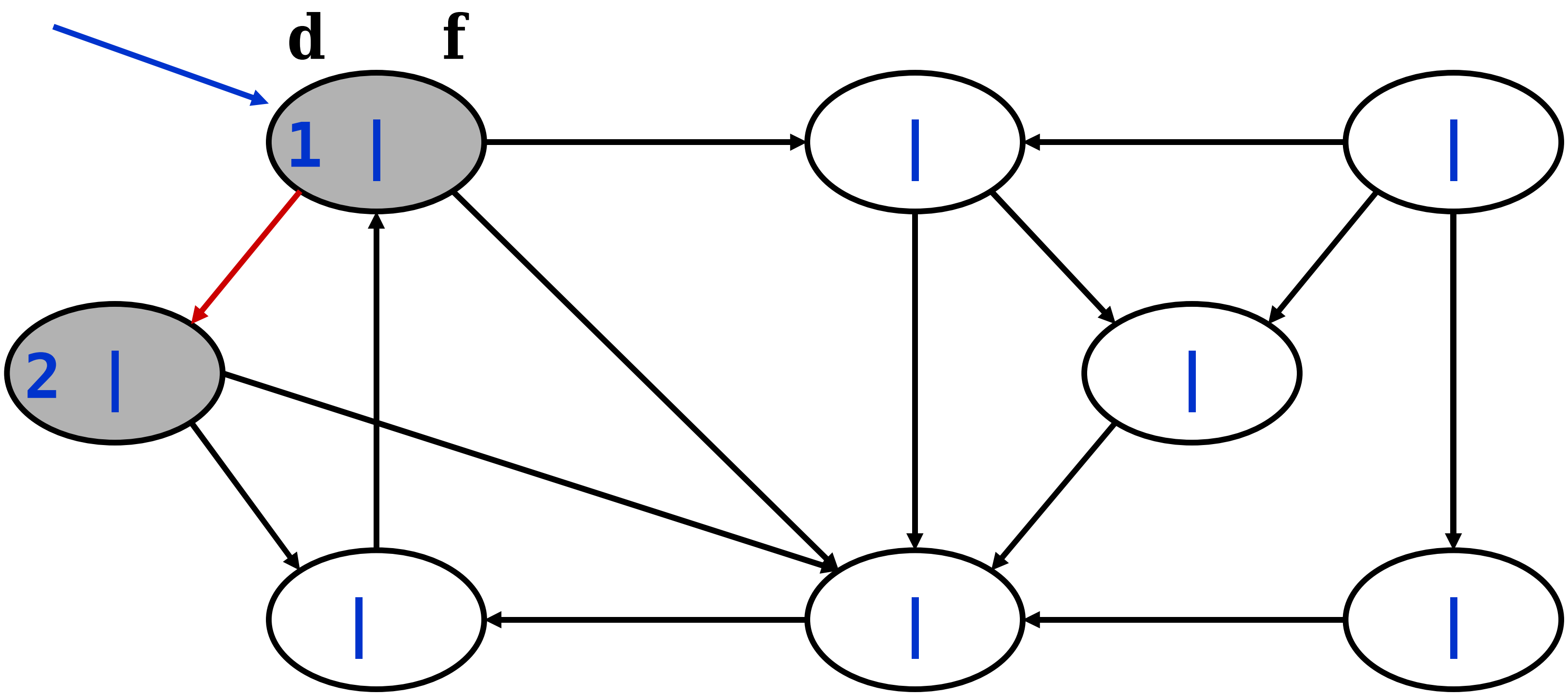
vértice origem



Existe algum vértice adjacente ao vértice que não tenha sido descoberto?

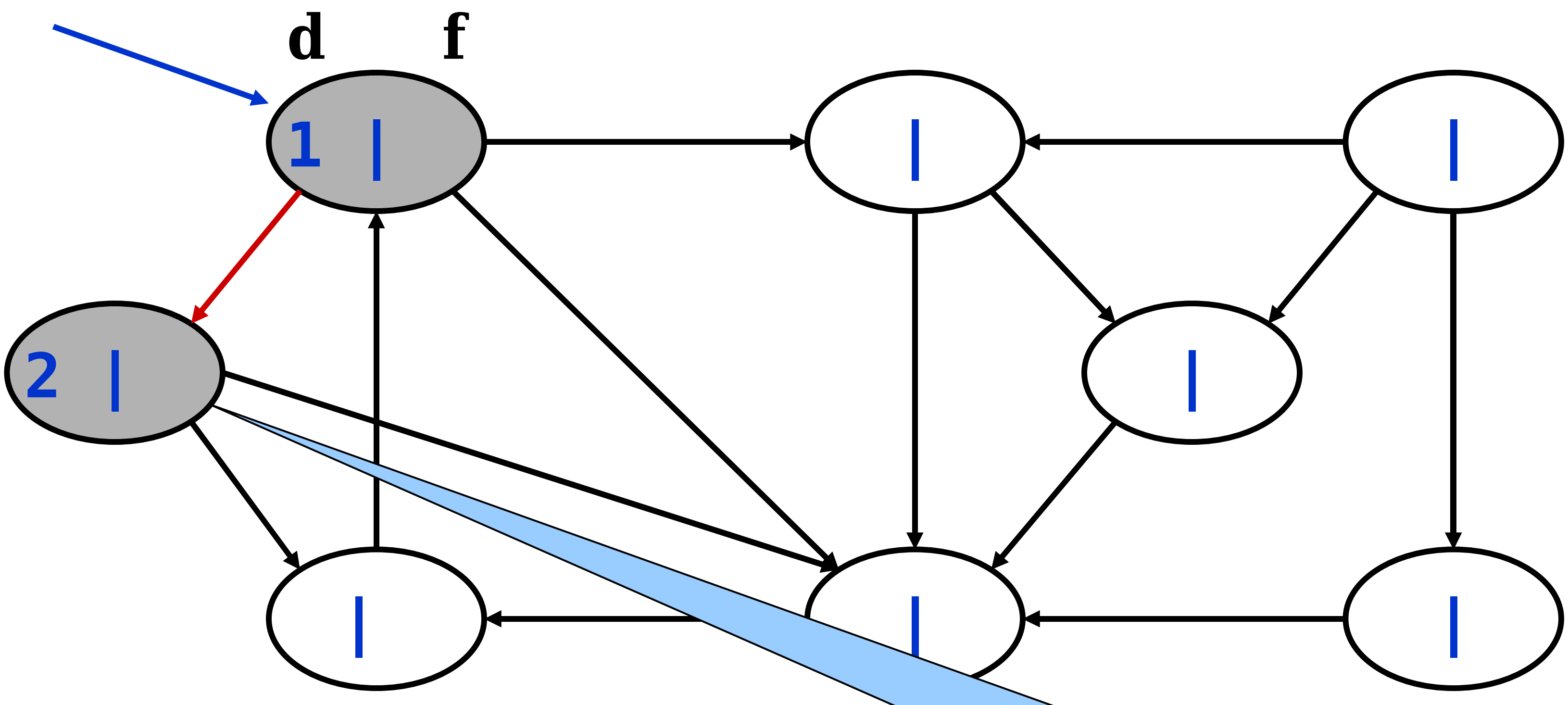
Busca em profundidade (DFS depth-first-search)

vértice origem



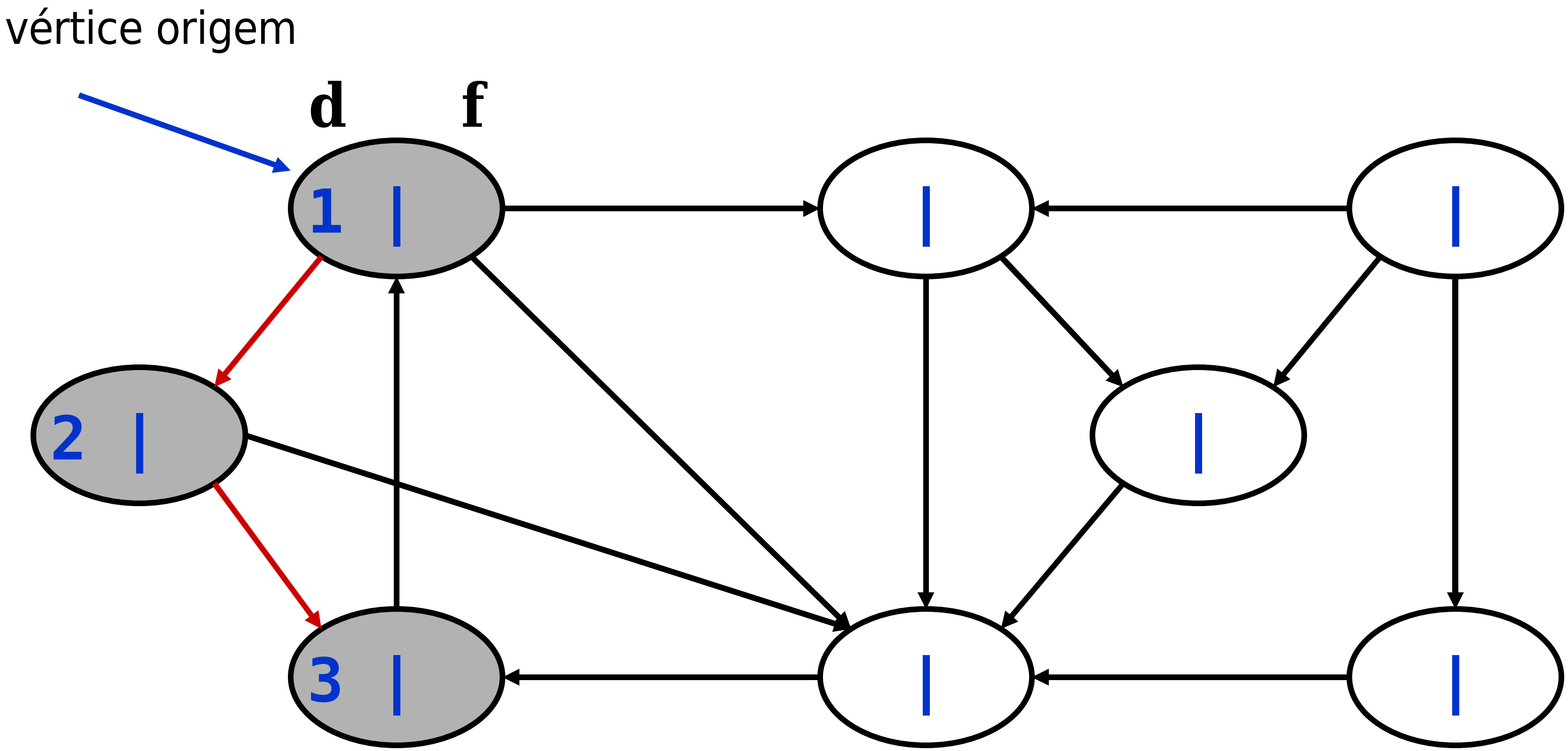
Busca em profundidade (DFS depth-first-search)

vértice origem

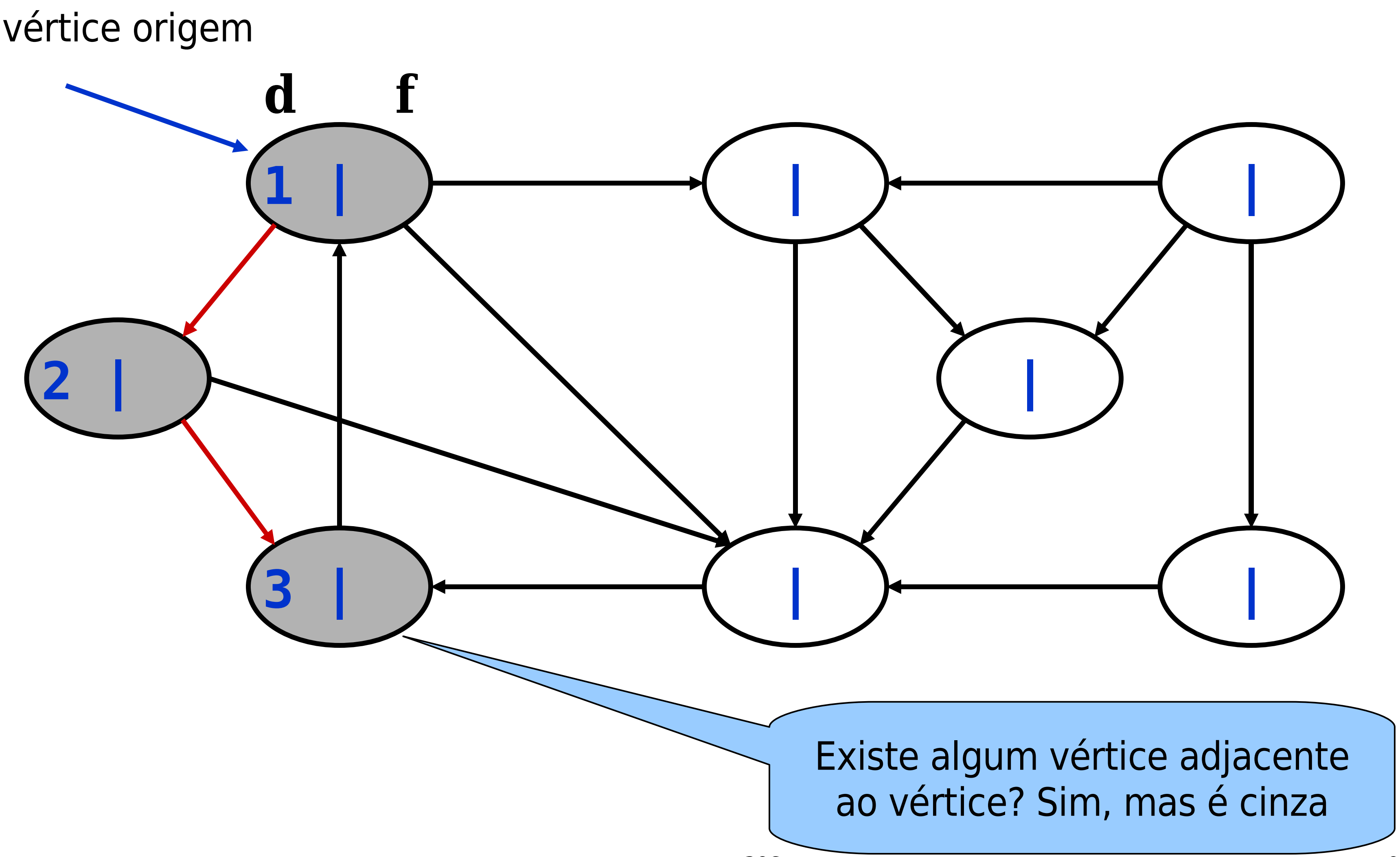


Existe algum vértice adjacente ao vértice que não tenha sido descoberto?

Busca em profundidade (DFS depth-first-search)

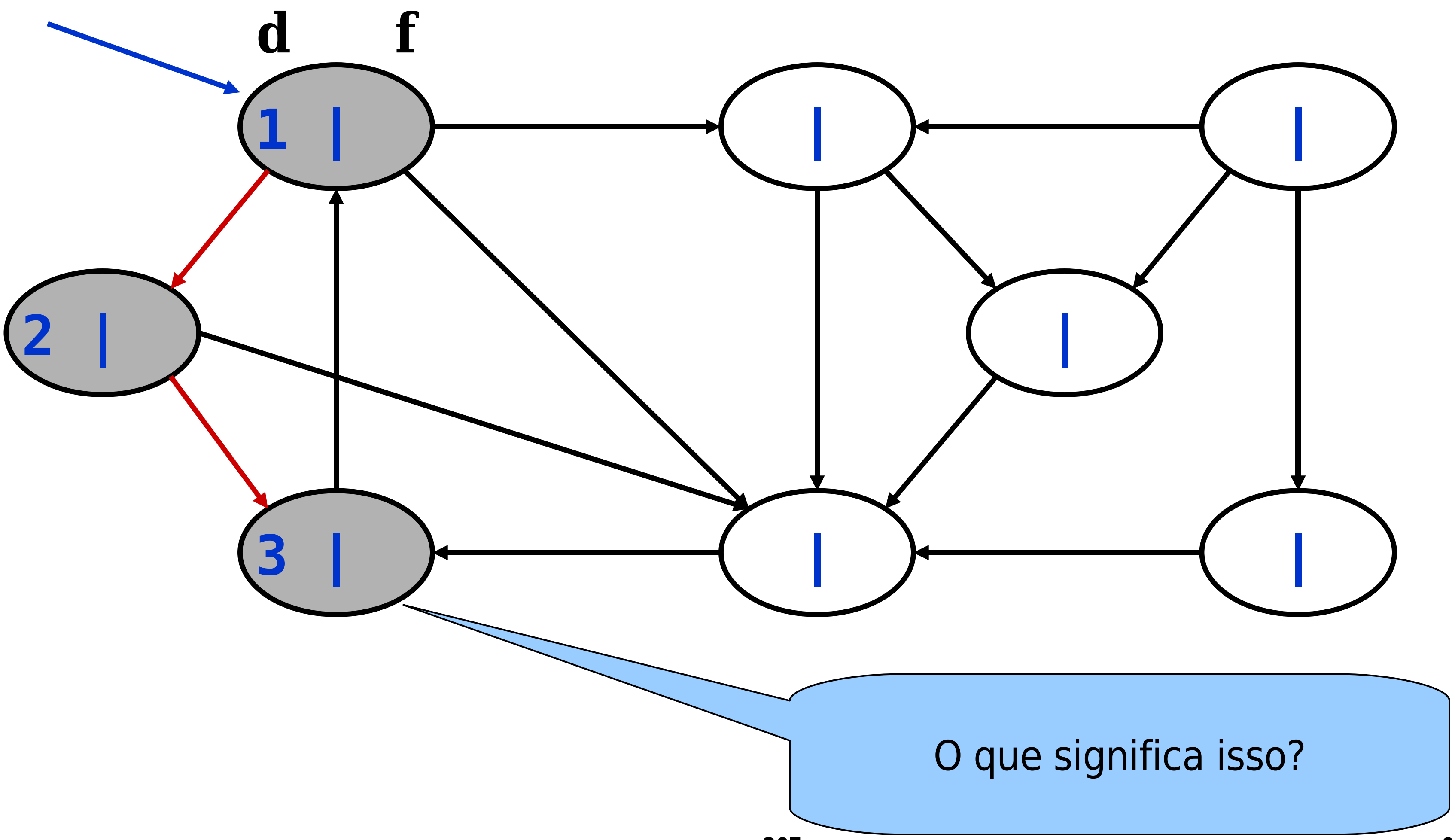


Busca em profundidade (DFS depth-first-search)

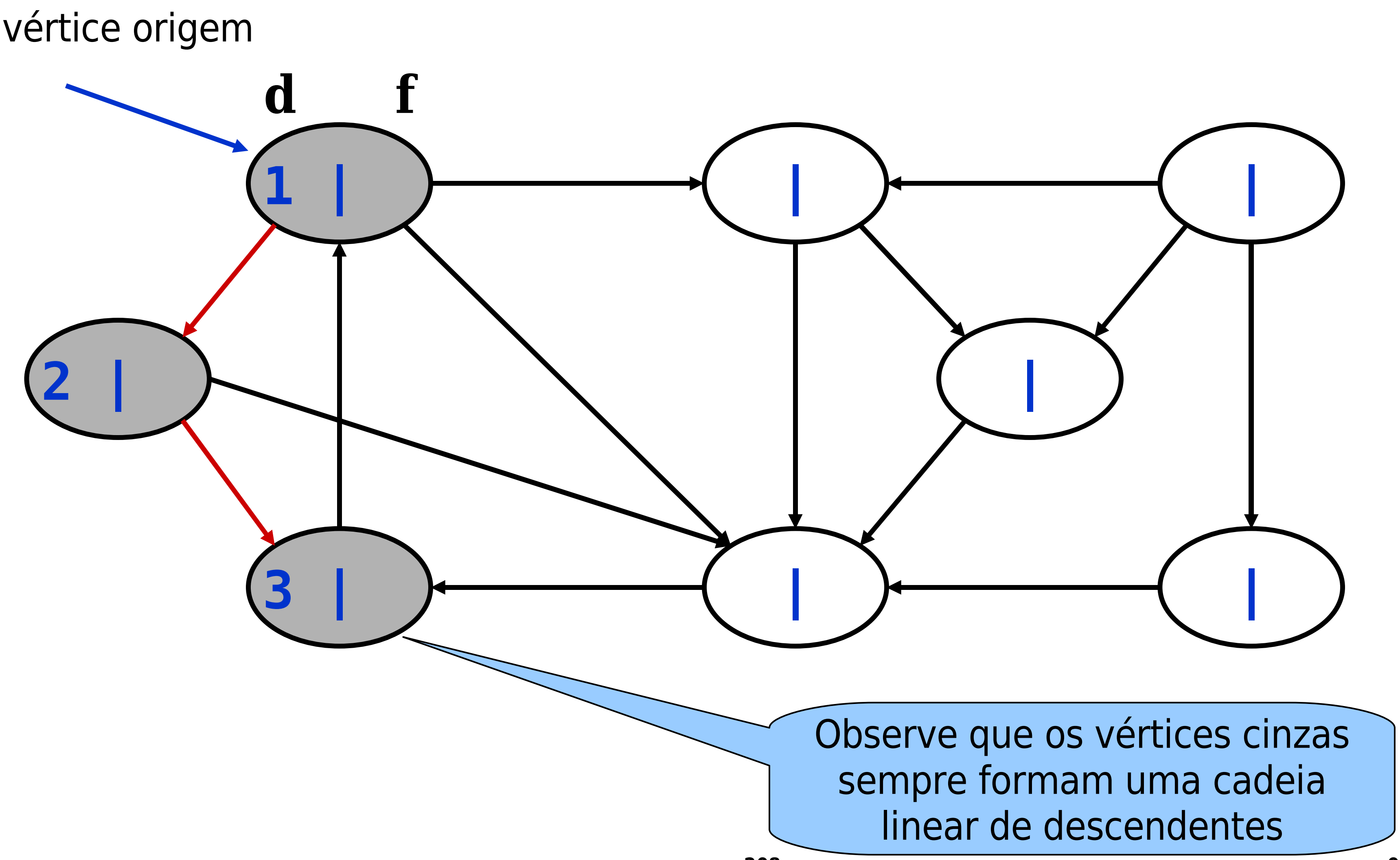


Busca em profundidade (DFS depth-first-search)

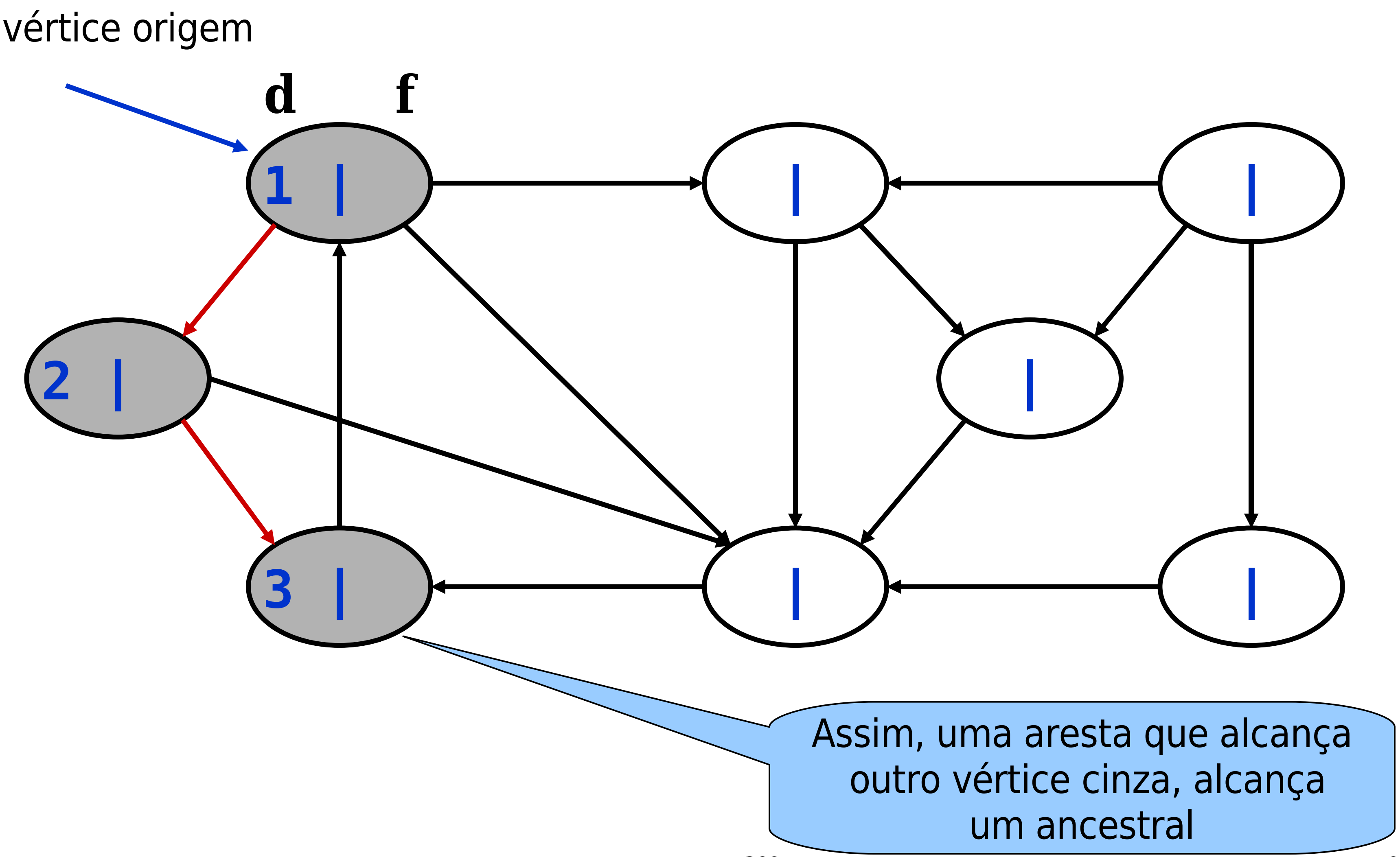
vértice origem



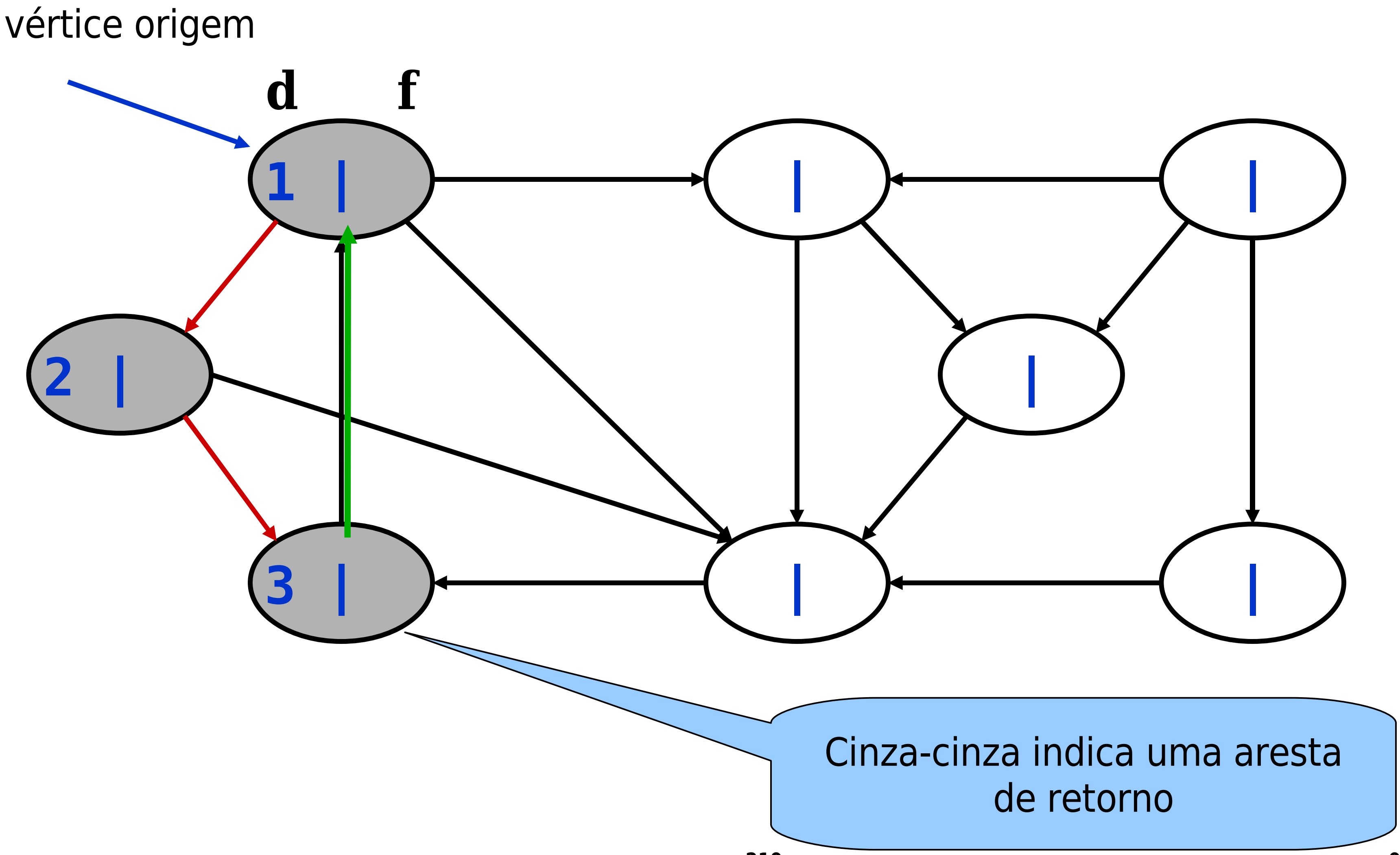
Busca em profundidade (DFS depth-first-search)



Busca em profundidade (DFS depth-first-search)



Busca em profundidade (DFS depth-first-search)



Busca em profundidade – Classificação das arestas

- O algoritmo DFS pode ser modificado para classificar arestas à medida que as encontra.
- Cada aresta (u,v) pode ser classificada pela cor do vértice v que é alcançado quando a aresta é explorada. Sabemos que:
  - Se v é **branco**, (u,v) é uma aresta da árvore
  - Se v é **cinza**, (u,v) é uma aresta de retorno
  - Se v é **preto**, (u,v) pode ser uma aresta de avanço ou uma aresta de cruzamento

Busca em profundidade – Classificação das arestas

- O algoritmo DFS pode ser modificado para classificar arestas à medida que as encontra.
- Cada aresta (u,v) pode ser classificada pela cor do vértice v que é alcançado quando a aresta é explorada. Sabemos que:
  - Se v é **branco**, (u,v) é uma aresta da árvore
  - Se v é **cinza**, (u,v) é uma aresta de retorno
  - Se v é **preto**, (u,v) pode ser uma aresta de avanço ou uma aresta de cruzamento.

Pesquisar como diferenciar uma aresta de avanço e uma aresta de cruzamento?

Exercício 1

Escrever o algoritmo para resolver o seguinte problema.

Problema: Verificar se o grafo é acíclico

Exercício 2

Escrever o algoritmo para resolver o seguinte problema.

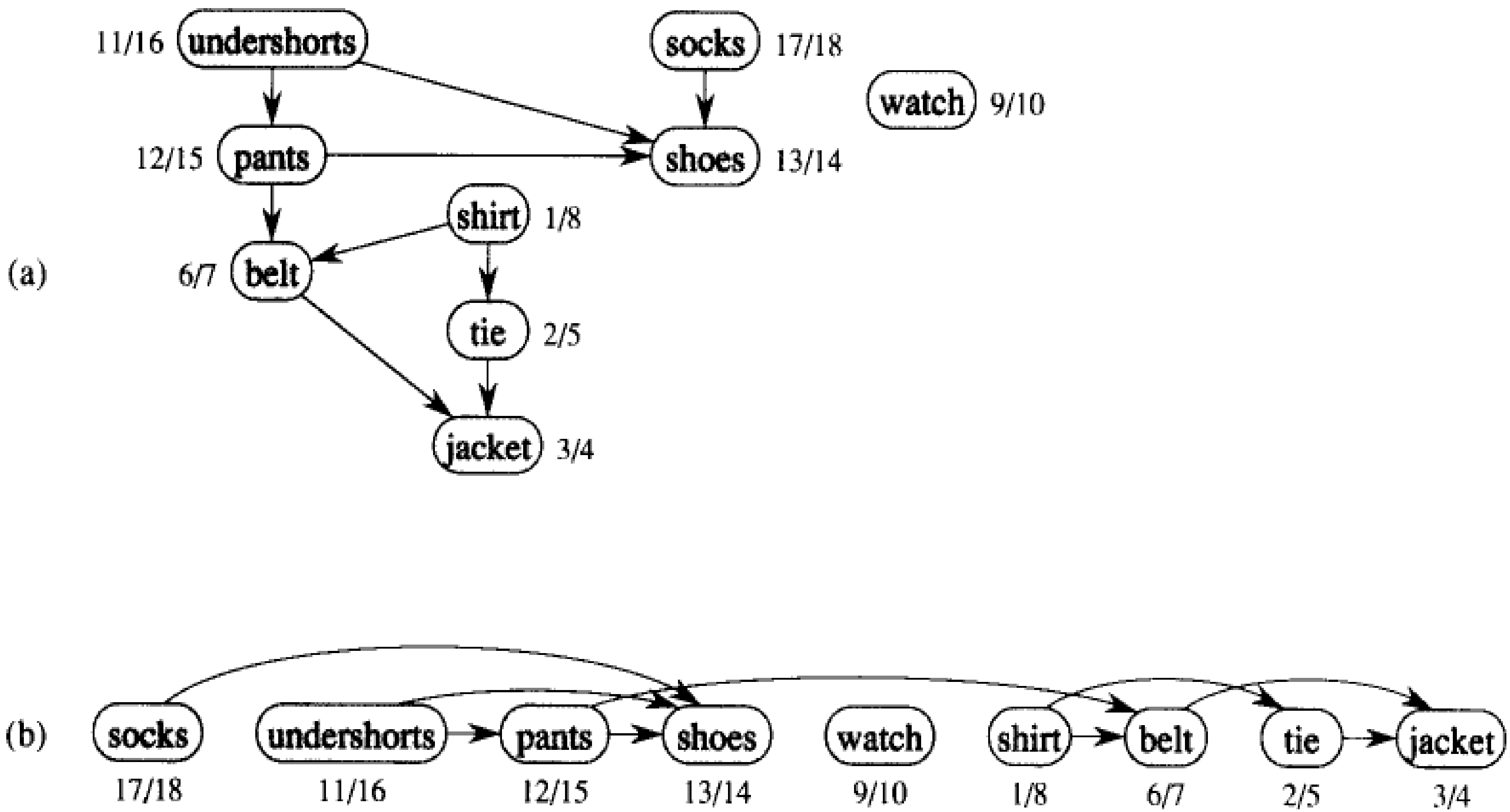
Problema: Determinar quantos componentes conectados tem um grafo não orientado.

Exercício 3

Escrever o algoritmo para resolver o seguinte problema.

Problema: Dado um grafo acíclico orientado, executar a ordenação topológica do grafo. Uma ordenação topológica é uma ordenação linear de todos os seus vértices, tal que se G contém uma aresta (u,v), então u aparece antes de v na ordenação.

Exercício 3



Cormen. Pag 437.