AULA 02

Algoritmos de busca em largura e profundidade em grafos

Karina Valdivia Delgado

Operações com grafos

- 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.

Roteiro

Motivação
Algoritmo de busca em largura
Algoritmo de busca em profundidade

Motivação

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.

Motivação

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!

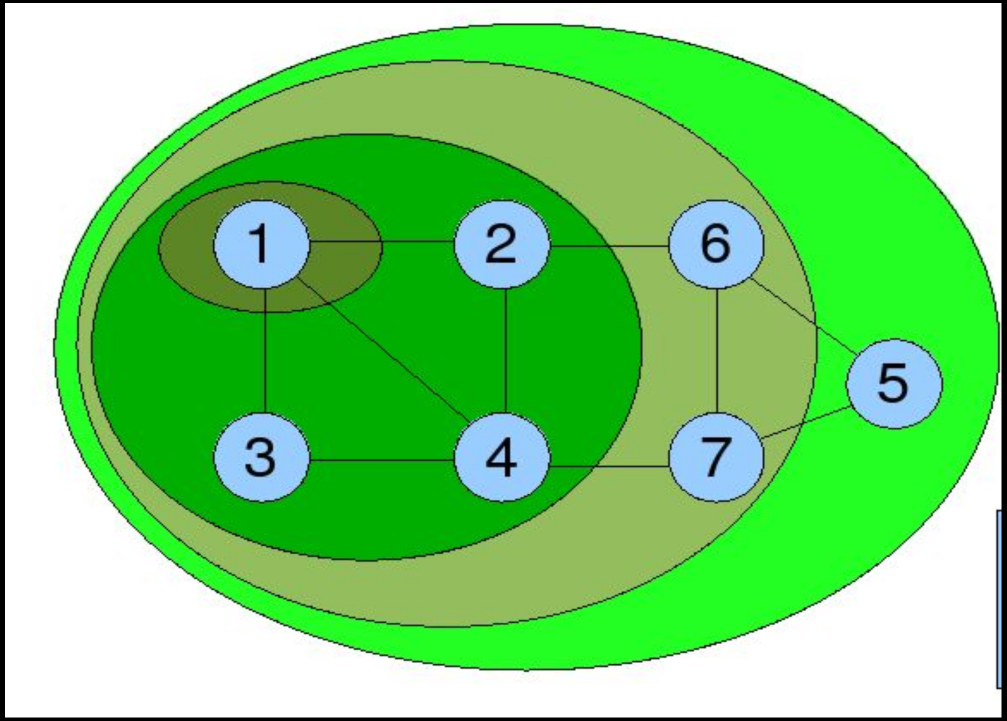
Busca em largura

Seja G = (V , A) e um vértice s, o algoritmo de busca em largura (breadth-first-search - 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.

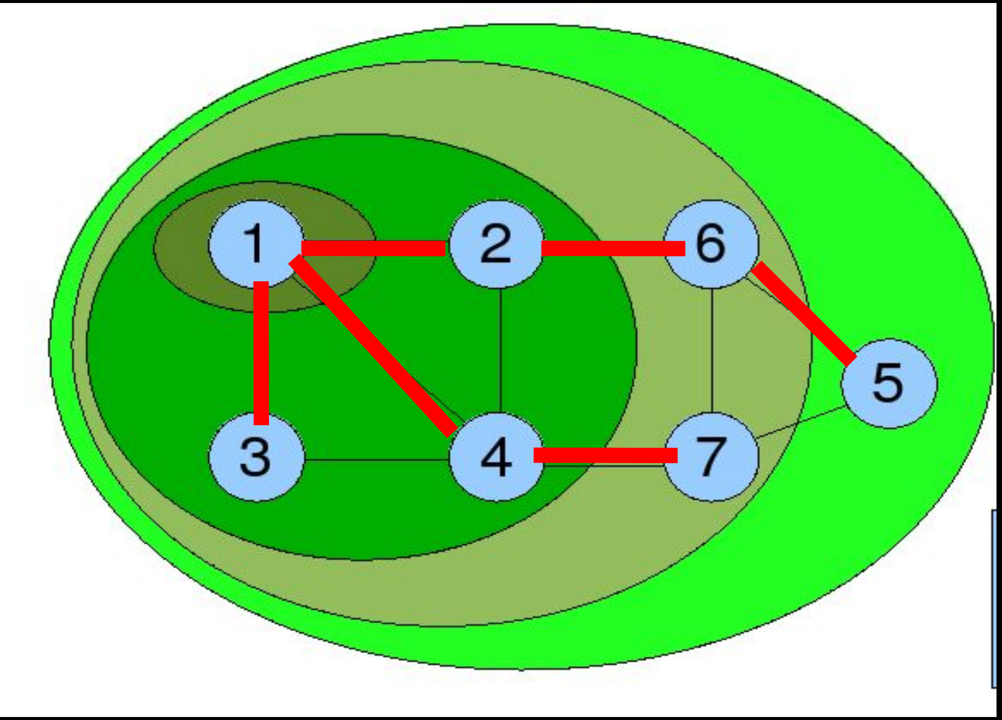
Busca em largura

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



Busca em largura

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



Busca em largura

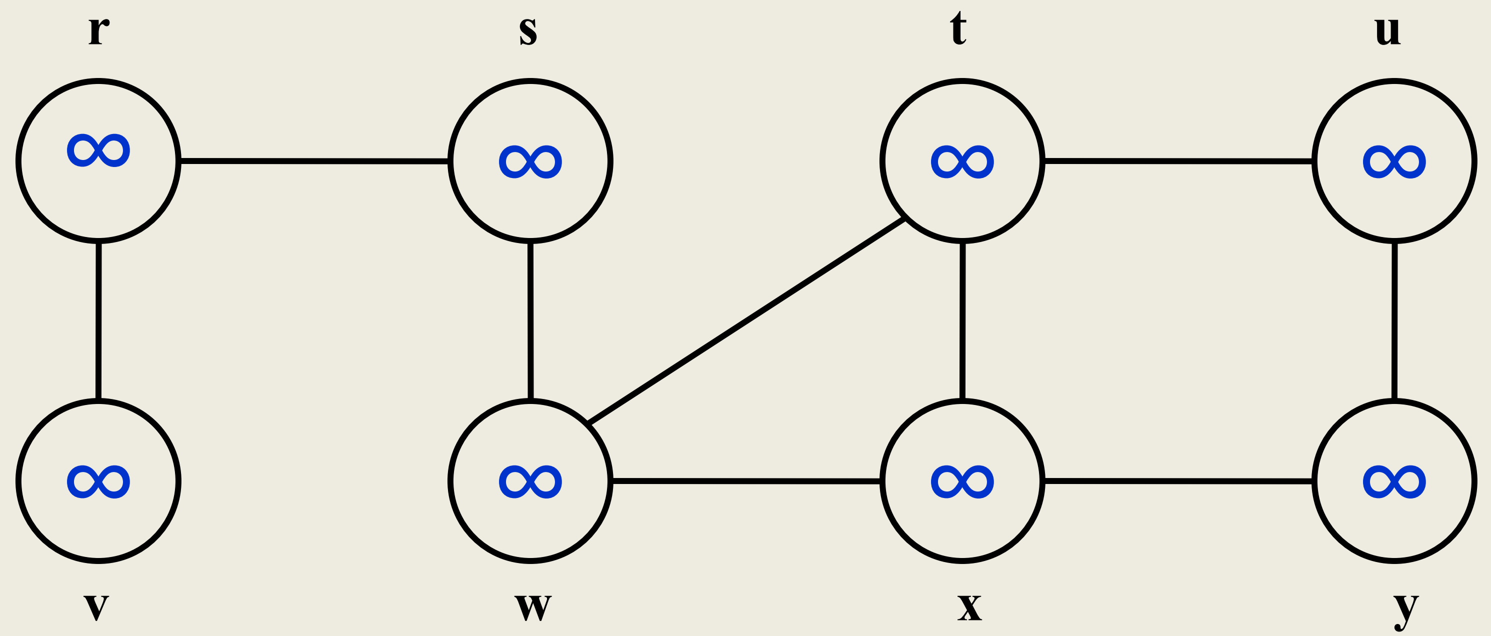
Para organizar o processo de busca os vértices são pintados:

- **branco**: não foram descobertos ainda
- **cinza:** são a fronteira. O vértice já foi descoberto mas ainda não examinamos os seus vizinhos.
- **preto:** são os vértices já descobertos e seus vizinhos já foram examinados.

É utilizada uma fila para manter os vértices cinzas.

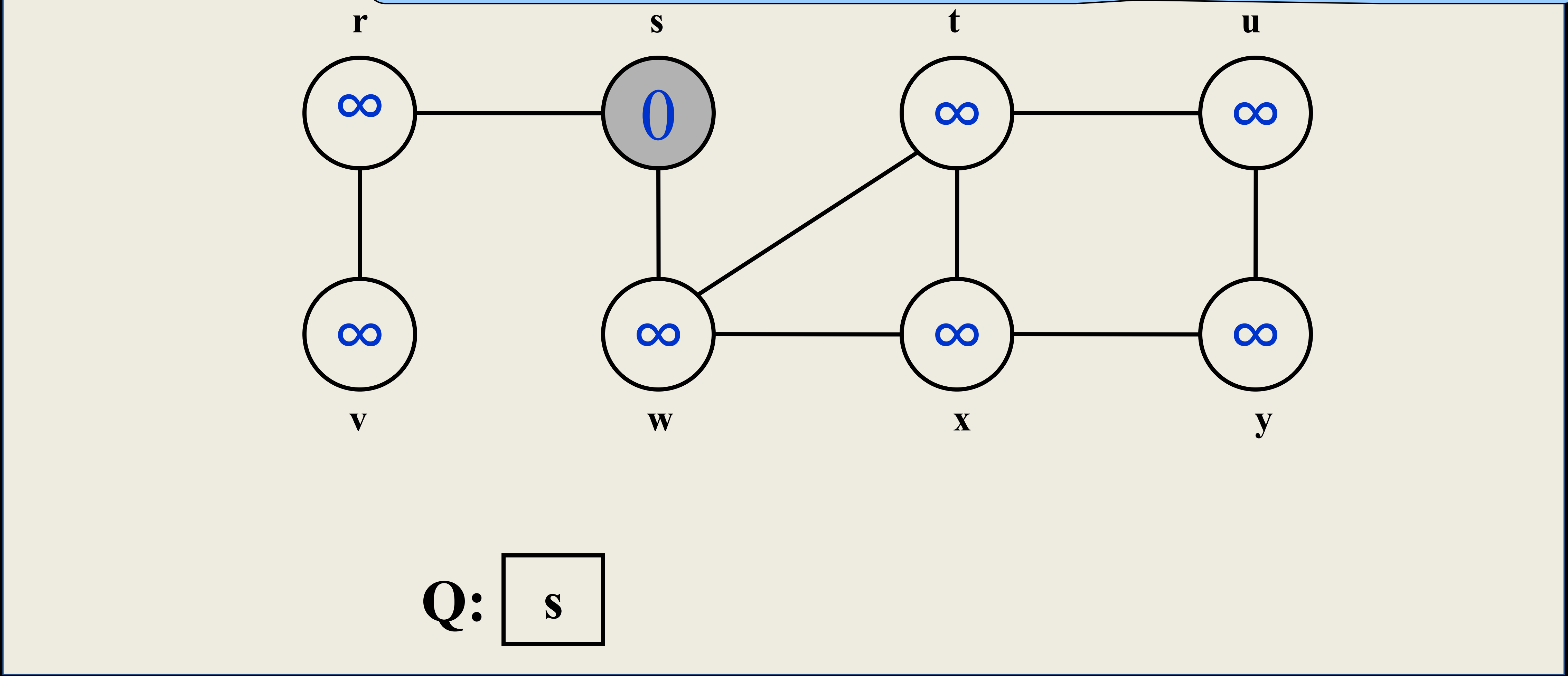
Busca em largura

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



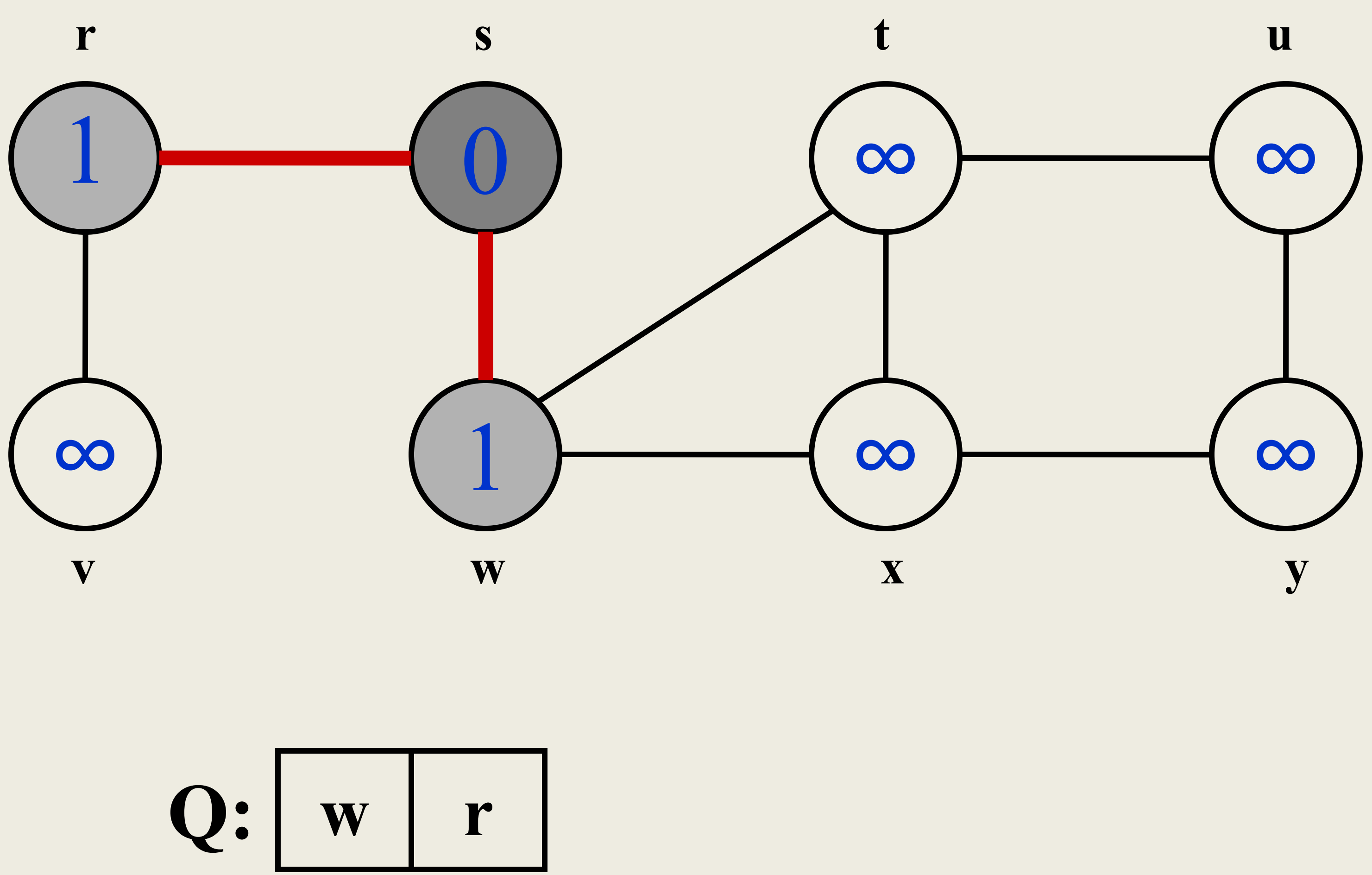
Busca em largura

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



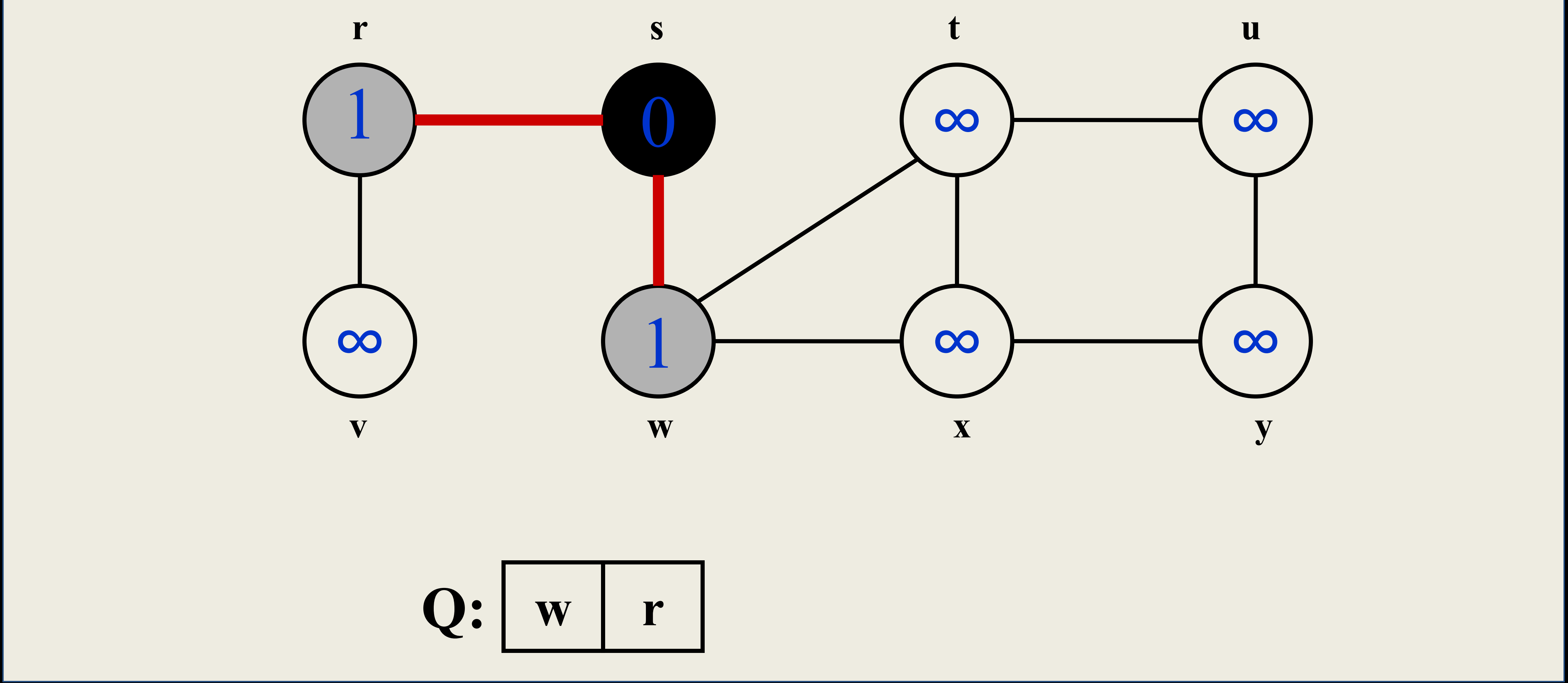
Busca

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

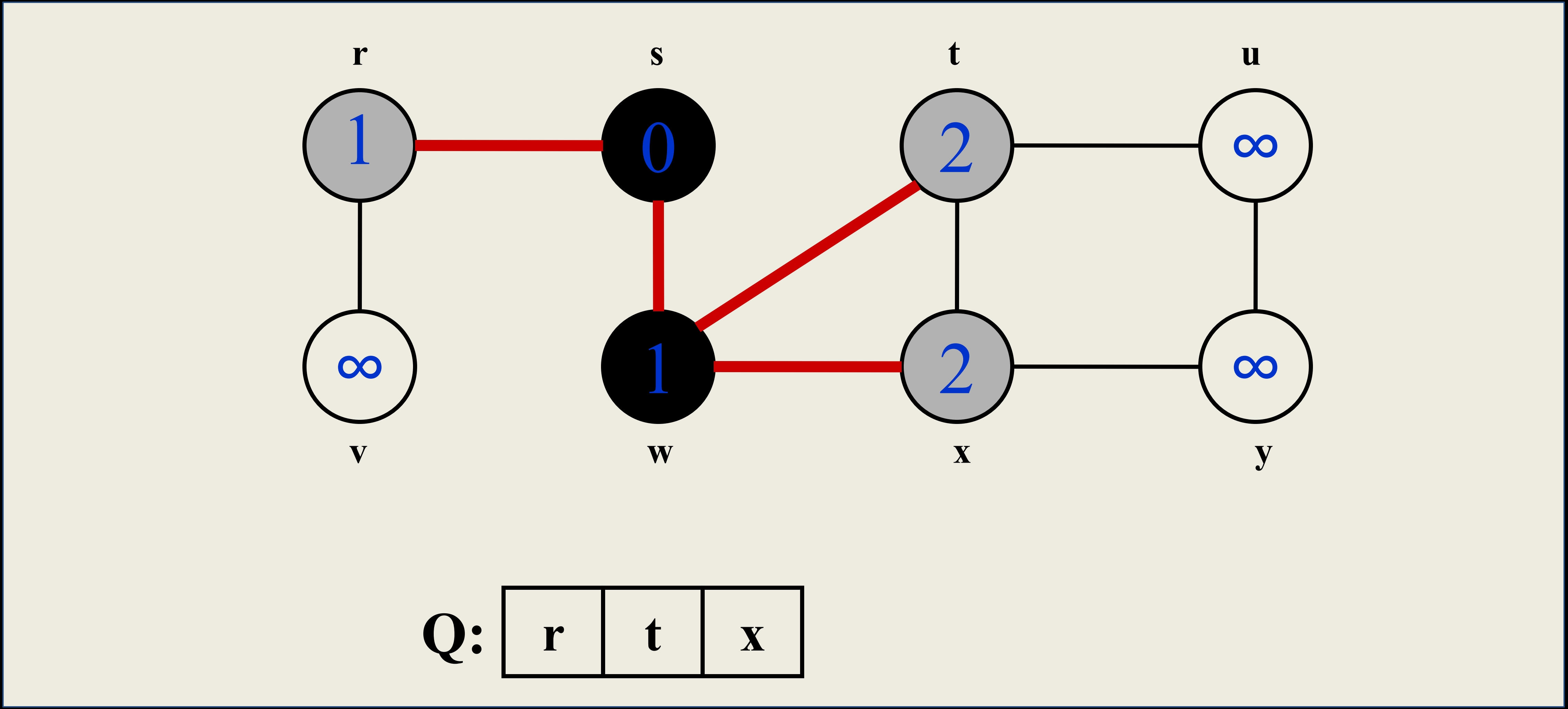


Busca e…

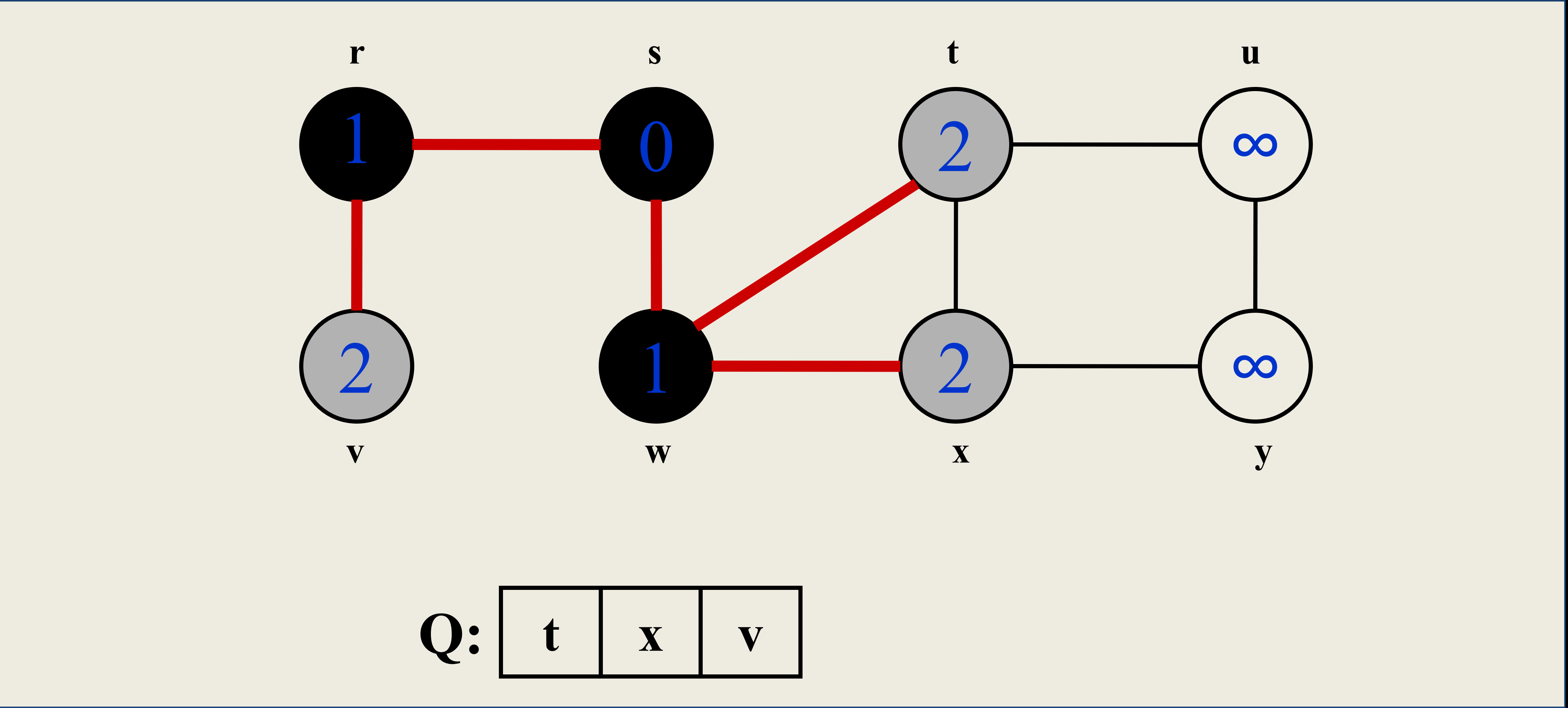
colorimos o vértice com preto (os seus vizinhos já foram descobertos)



Busca em largura

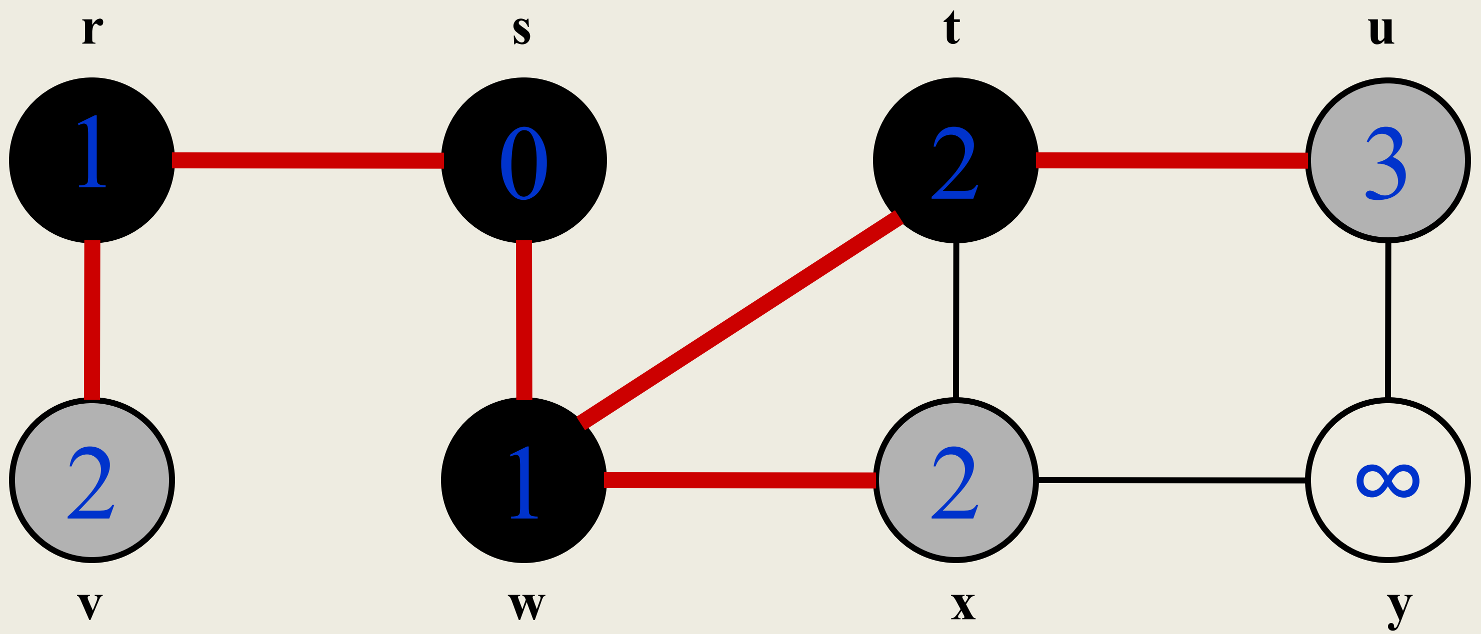


Busca em largura



Busca em

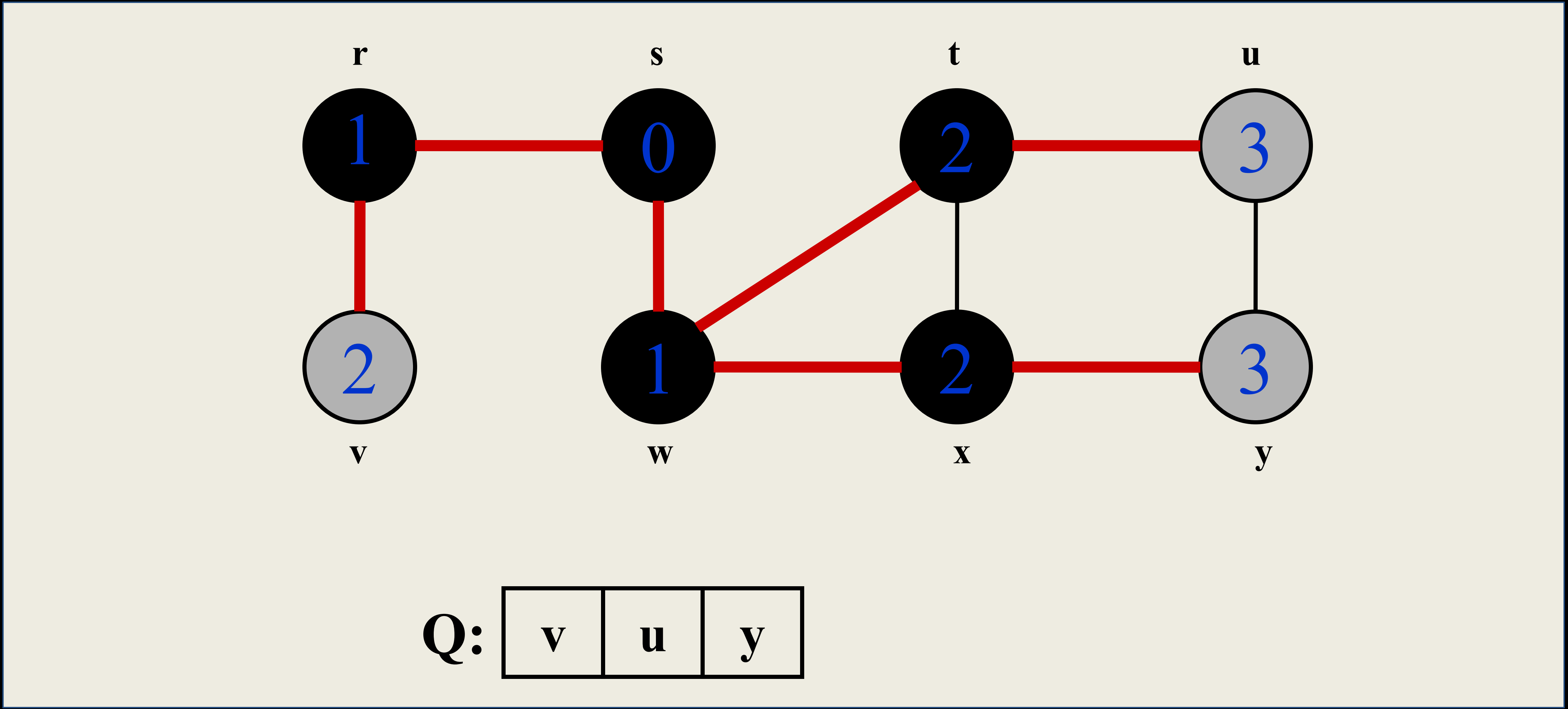
Observe que somente colocamos na fila vértices brancos, que são imediatamente coloridos de cinza ao entrar na fila



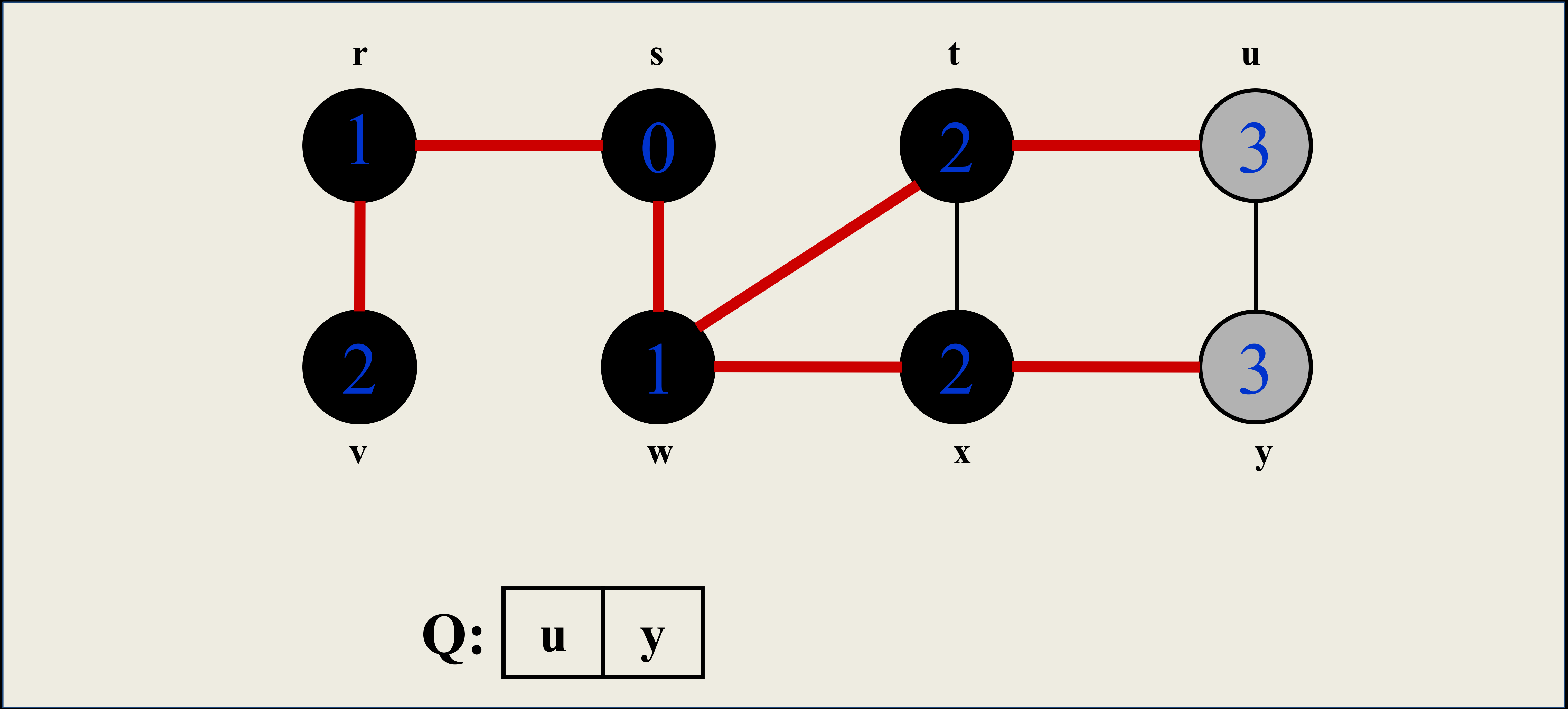
Q:

x	v	u

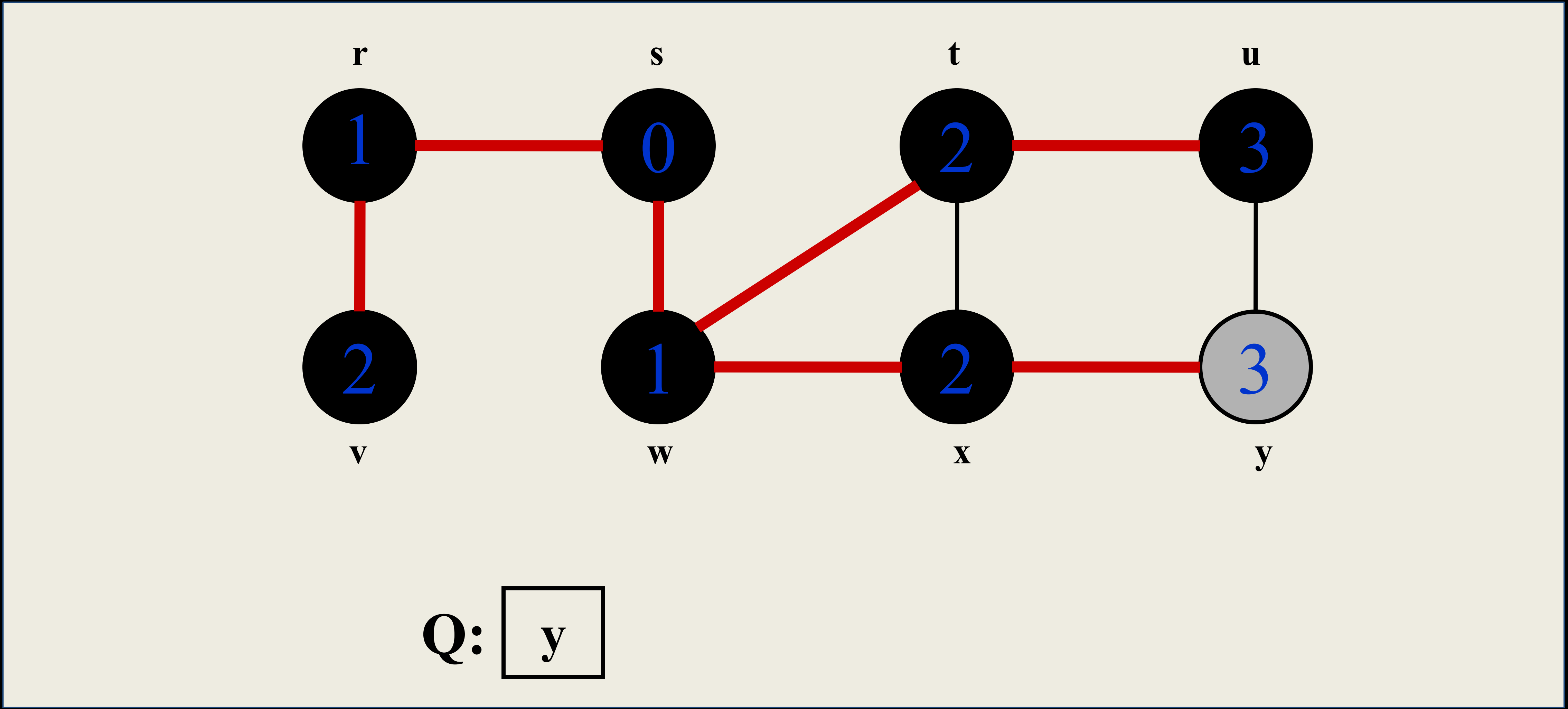
Busca em largura



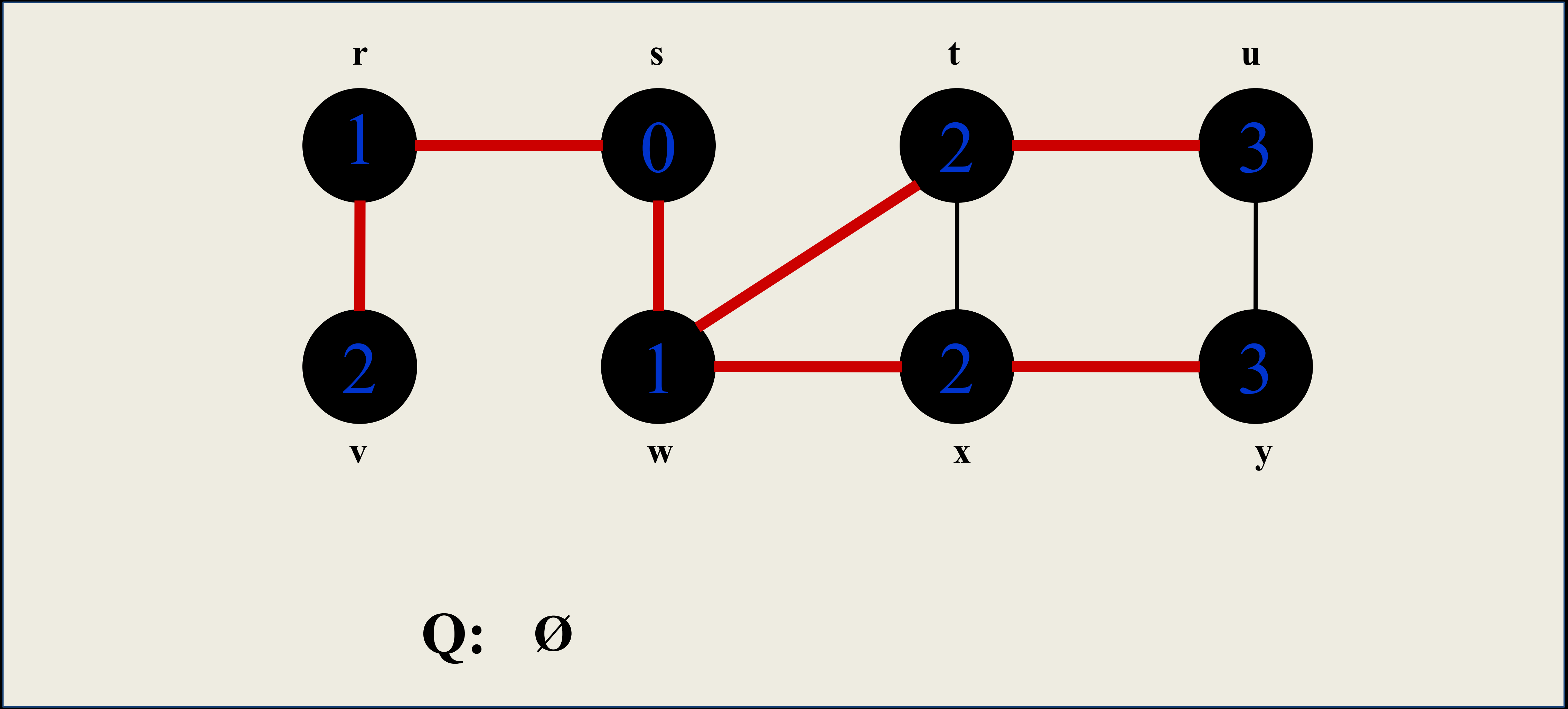
Busca em largura



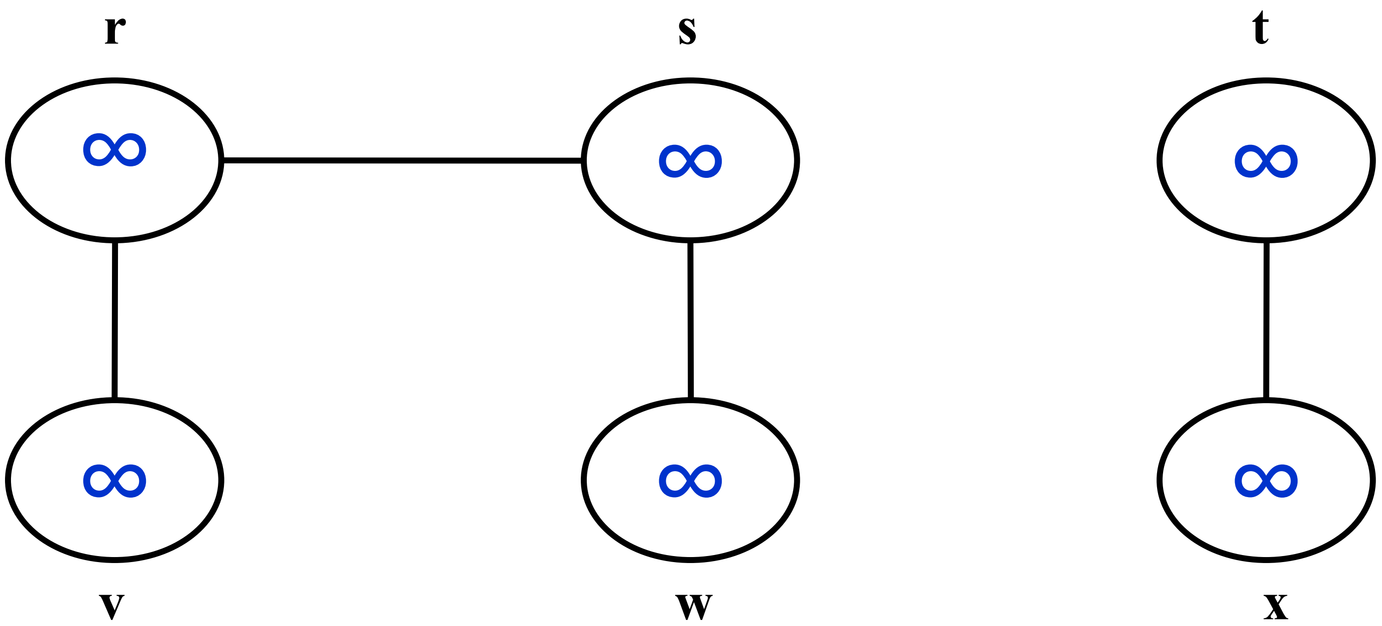
Busca em largura



Busca em largura



Busca em largura (BFS breadth-first-search)



Aplicar o algoritmo BFS,
considerar s como vértice origem

Busca em largura

```
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(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(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(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(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(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 à descoberta de v
                     ENQUEUE(Q, v)
        color[u] ← BLACK               ▷ os vizinhos de u já foram examinados
```

Busca em largura

```
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)                     ▷
        for each v adjacent to u           ▷ p
            if color[v] = WHITE            ▷ s
                then  color[v] ← GRAY
                      d[v] ← d[u] + 1
                      π[v] ← u             ▷ p
                      ENQUEUE(Q, v)
        color[u] ← BLACK
```

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

Busca em largura

```
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)               ▷
        for each v adjacent to u     ▷ p
            if color[v] = WHITE      ▷ s
                then  color[v] ← GRAY
                      d[v] ← d[u] + 1
                      π[v] ← u       ▷ p
                      ENQUEUE(Q, v)
        color[u] ← BLACK
```

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

Busca em largura

```
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    ▷ p
            if color[v] = WHITE     ▷ s
                then  color[v] ← GRAY
                      d[v] ← d[u] + 1
                      π[v] ← u      ▷ p
                      ENQUEUE(Q, v)
        color[u] ← BLACK
```

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(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             ▷ Enquant
        u ← DEQUEUE(Q)               ▷
        for each v adjacent to u     ▷ pa
            if color[v] = WHITE      ▷ se
                then  color[v] ← GRAY
                      d[v] ← d[u] + 1
                      π[v] ← u       ▷ pa
                      ENQUEUE(Q, v)
        color[u] ← BLACK
```

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

Busca em largura

```
BFS(V, A, s)
1.    for each u in V − {s}          ▷ para cada
2.        color[u] ← WHITE            ▷ no iníci
3.        d[u] ← infinity
4.        π[u] ← NIL
5.    color[s] ← GRAY                 ▷ Vértice
6.    d[s] ← 0
7.    π[s] ← NIL
8.    Q ← {}
9.    ENQUEUE(Q, s)                   ▷ Colocar
10    while Q is non-empty            ▷ Enquant
11.       u ← DEQUEUE(Q)                  ▷
12.       for each v adjacent to u      ▷ pa
13.           if color[v] = WHITE       ▷ se
14.               then  color[v] ← GRAY
15.                     d[v] ← d[u] + 1
16.                     π[v] ← u          ▷ pa
17.                     ENQUEUE(Q, v)
18.       color[u] ← BLACK
```

A parte de inicialização é O(V)

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

Busca em largura

```
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                  ▷ Vértic
d[s] ← 0
π[s] ← NIL
Q ← {}
ENQUEUE(Q, s)                    ▷ Coloc
while Q is non-empty             ▷ Enqua
    u ← DEQUEUE(Q)
    for each v adjacent to 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
```

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

Busca em largura

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

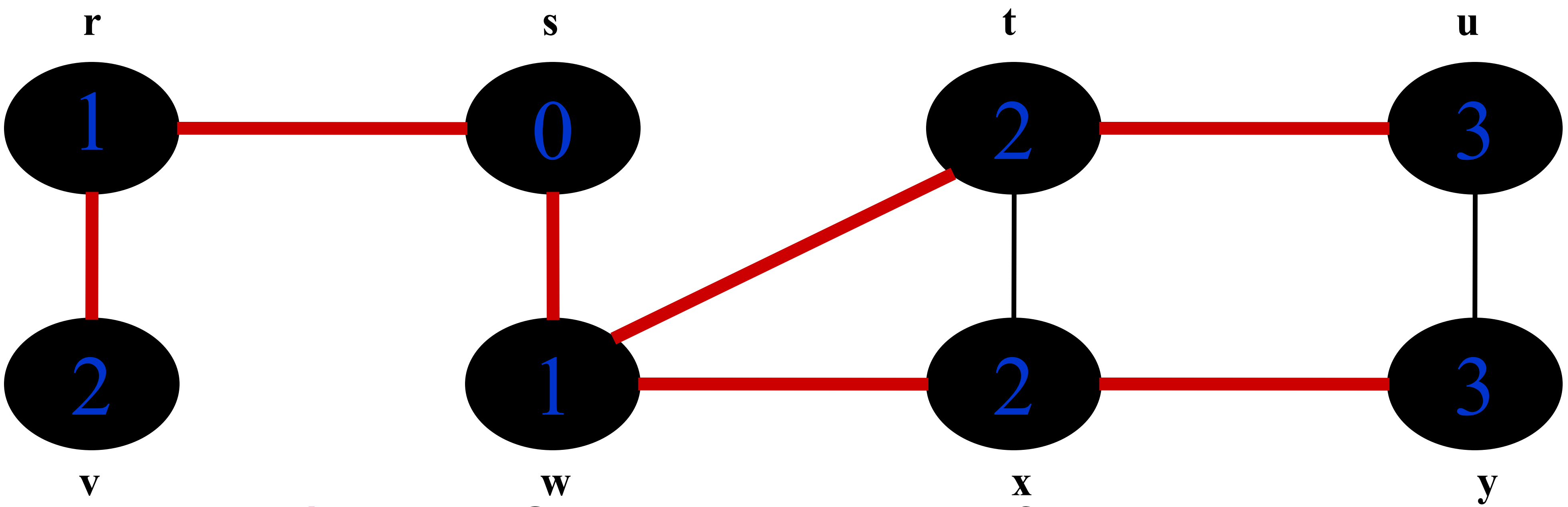
É necessário realizar o BFS completo?

Caminho entre dois vértices

```
BFS(V, A, s)
    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 vértic
    while Q is non-empty            ▷ Enquanto existam
        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
```

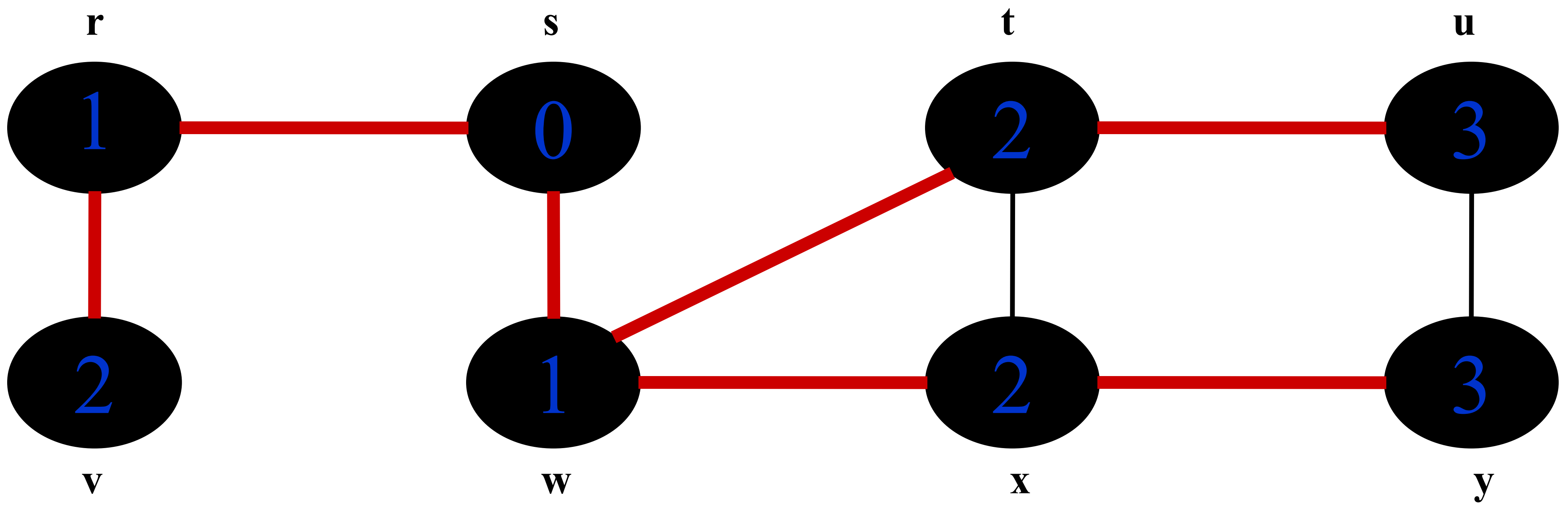
Que linhas mudar?

Caminhos mais curto



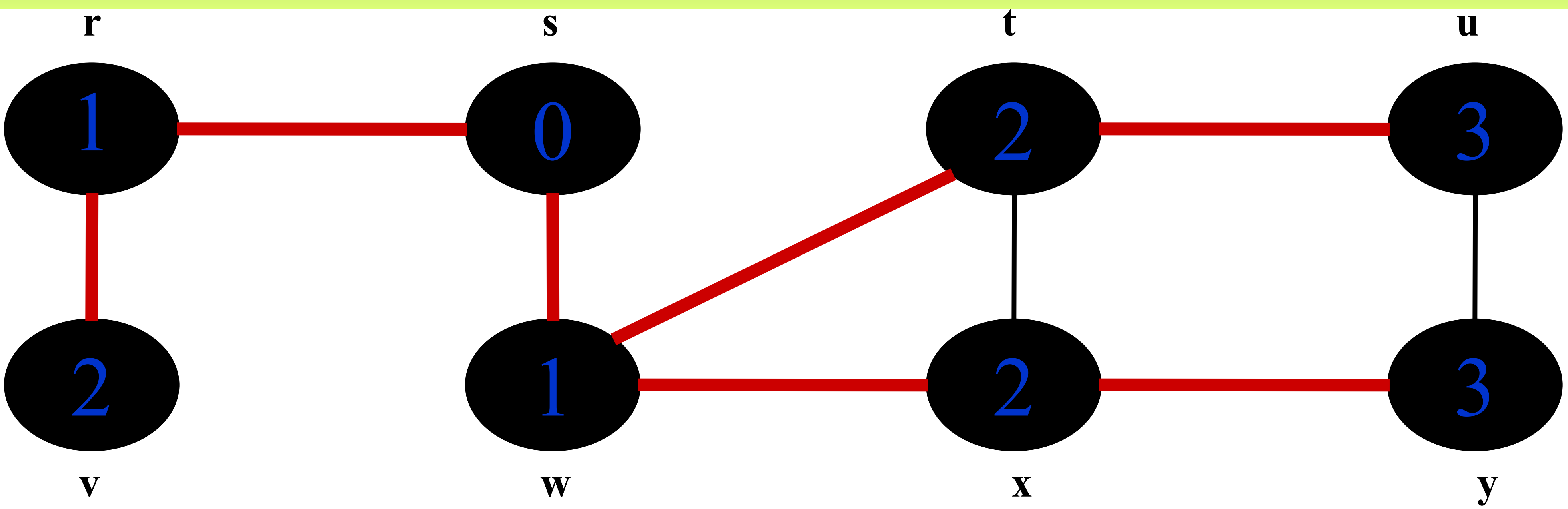
– BFS produz uma árvore BFS com raiz em s e define o caminho mais curto
– Qual é o caminho mais curto entre u e s?
– Qual é o caminho mais curto entre u e y?

Caminhos mais curto



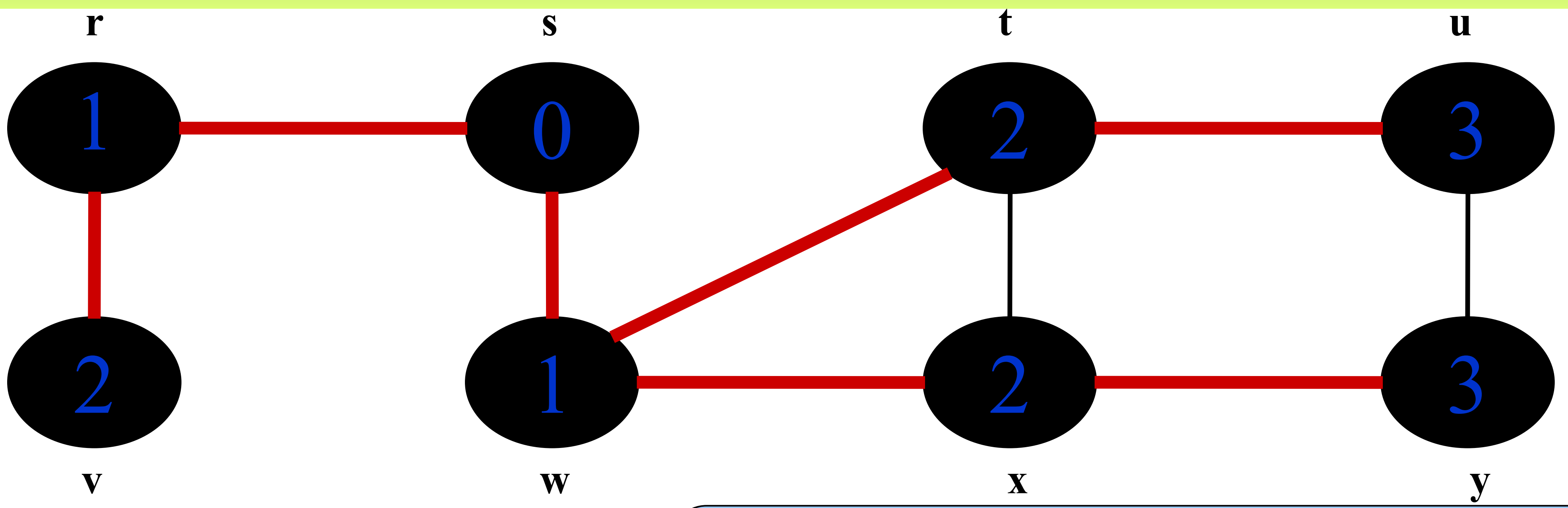
- BFS produz uma árvore BFS com raiz em s e define o caminho mais curto
- Qual é o caminho mais curto entre u e s?
- Qual é o caminho mais curto entre u e y?
- A árvore define o caminho mais curto para a raiz!!!!

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 executamos a bus[...]
entre s e os vértice acessíveis a pa[...]

– Elabore um algoritmo para imprimi[...] v
qualquer

Podemos usar recursão.

Print-Path(G, s, v)

Caso Base: ?

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, mesmo fazendo conexão, 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, mesmo fazendo conexão, 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, mesmo fazendo conexão, de qualquer cidade para qualquer cidade.

- Solução: usar BFS
  - Escolher um vértice v qualquer de G
  - Executar B
  - Verificar se

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, mesmo fazendo conexão, de qualquer cidade para qualquer cidade.

- Solução: usar BFS
  - Escolher um vértice v qualquer de G
  - Executar B
  - Verificar se

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

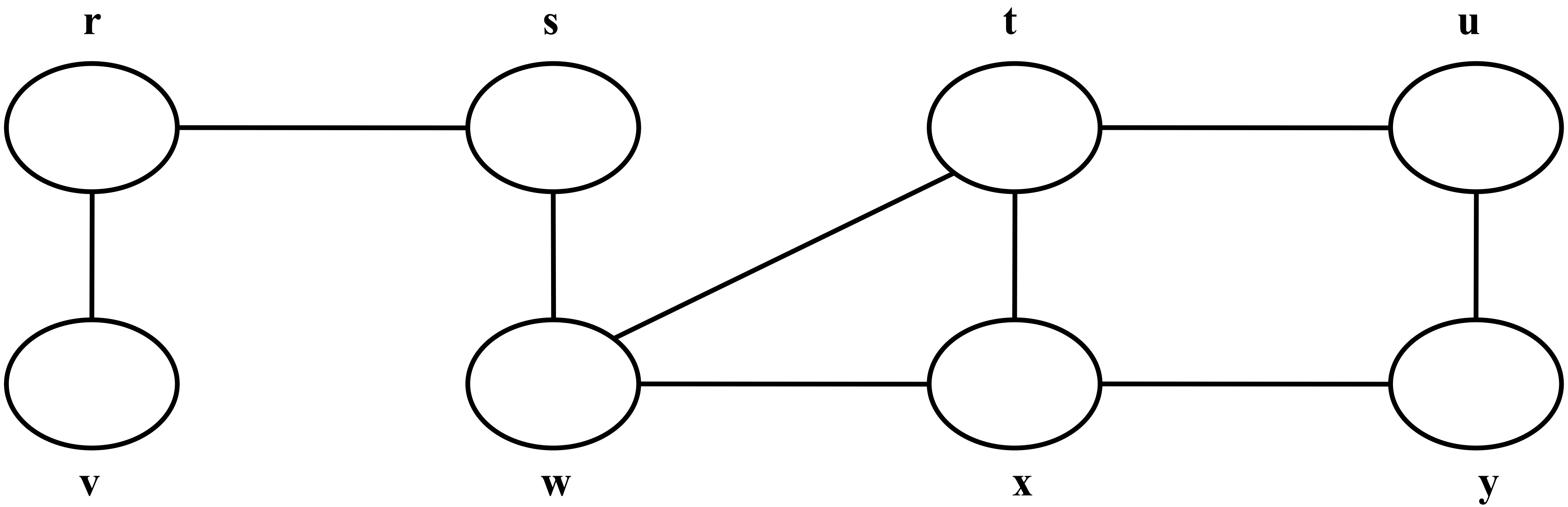
(i.e., se cada par de vértices está conectado por um caminho)?

Exemplo: gostaria de saber se posso voar, mesmo fazendo conexão, 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}
```

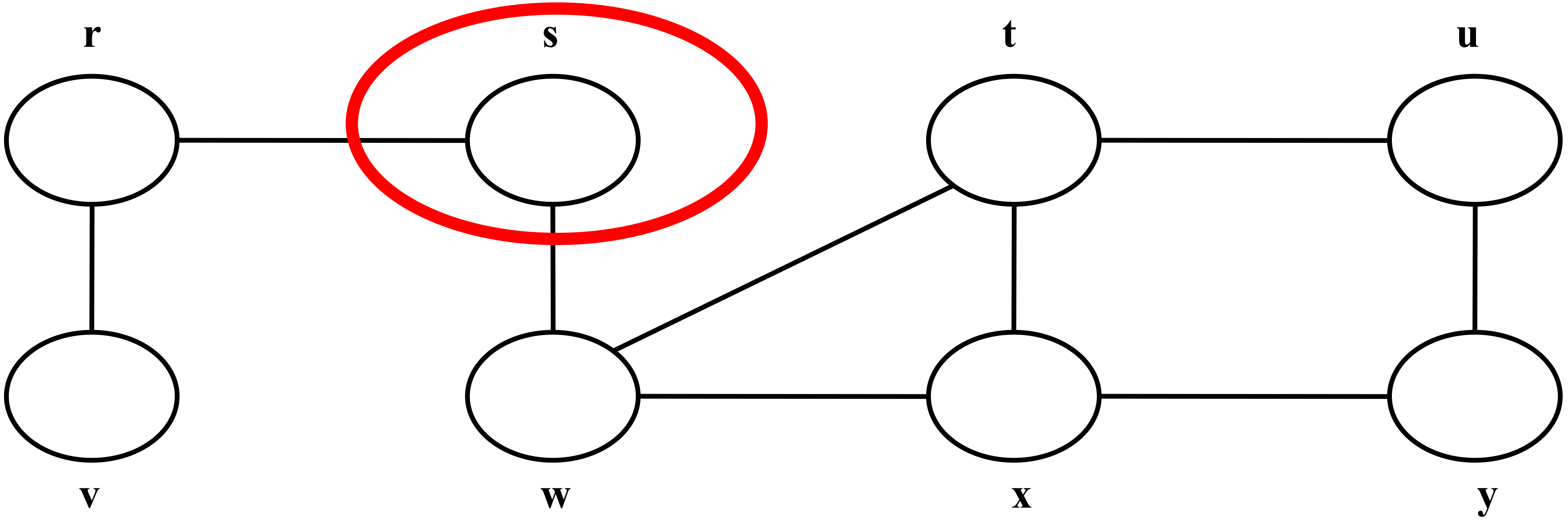
Grafo conectado



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?

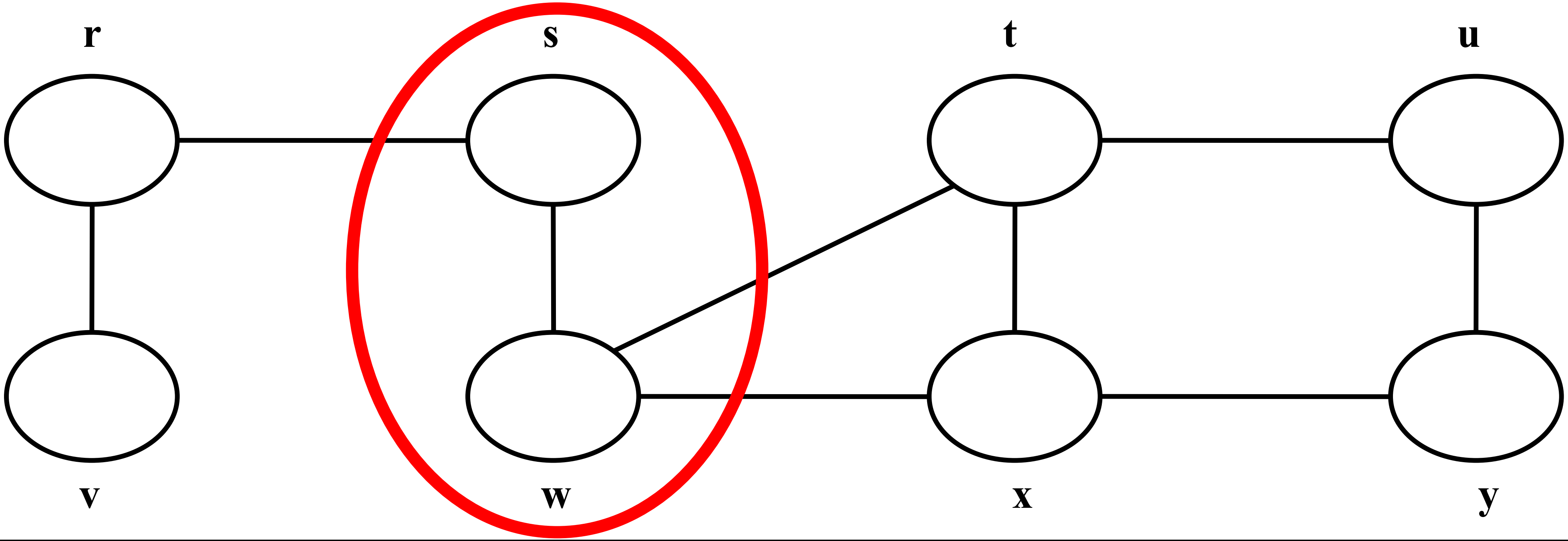
Grafo conectado



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?

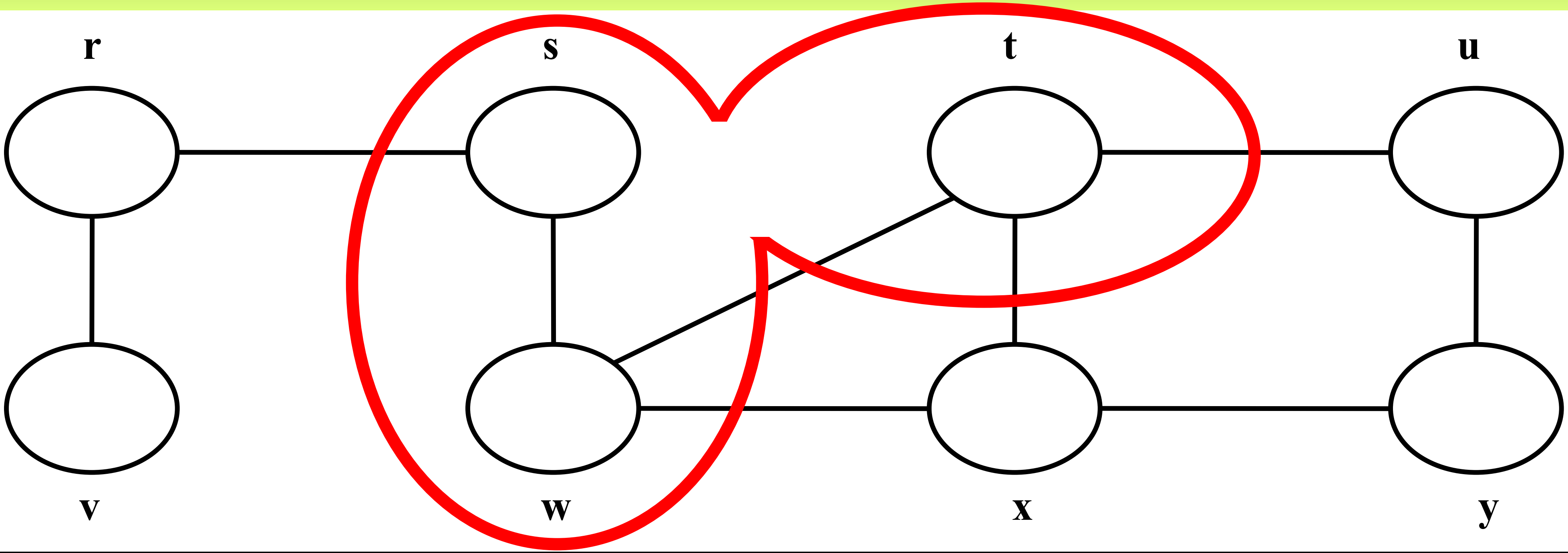
Grafo conectado



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?

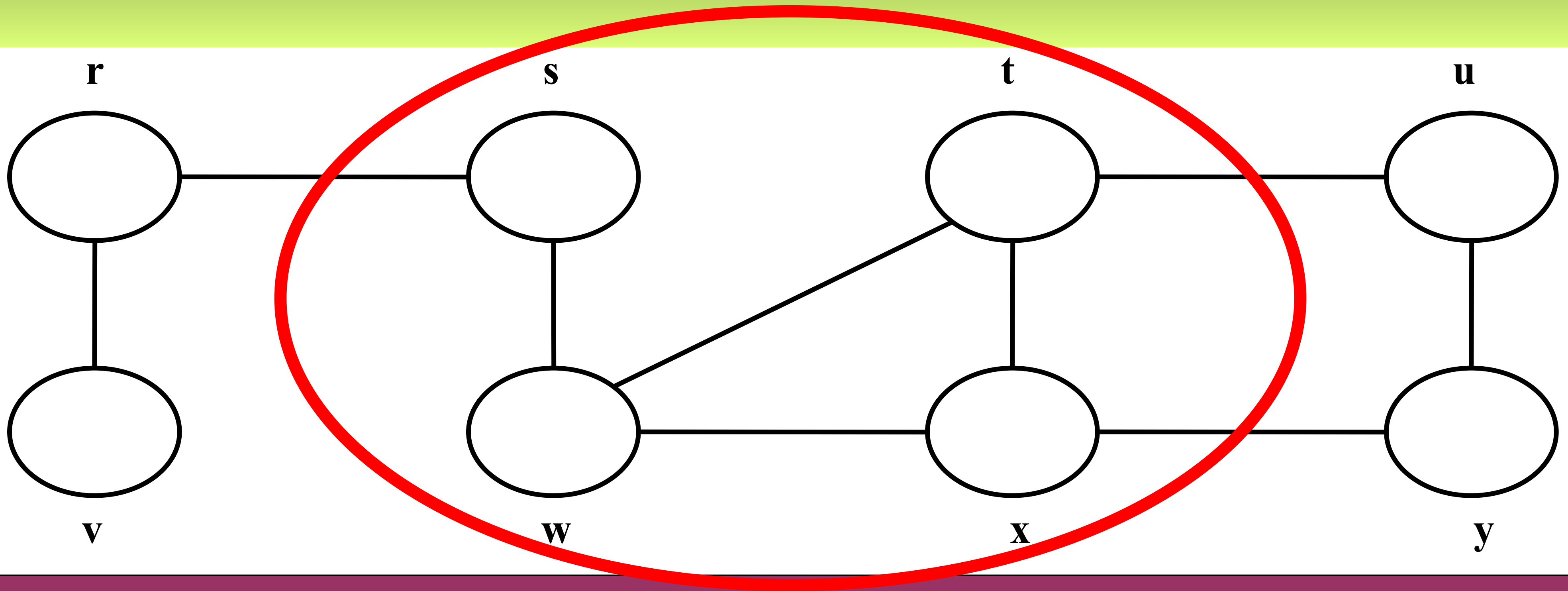
Grafo conectado



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?

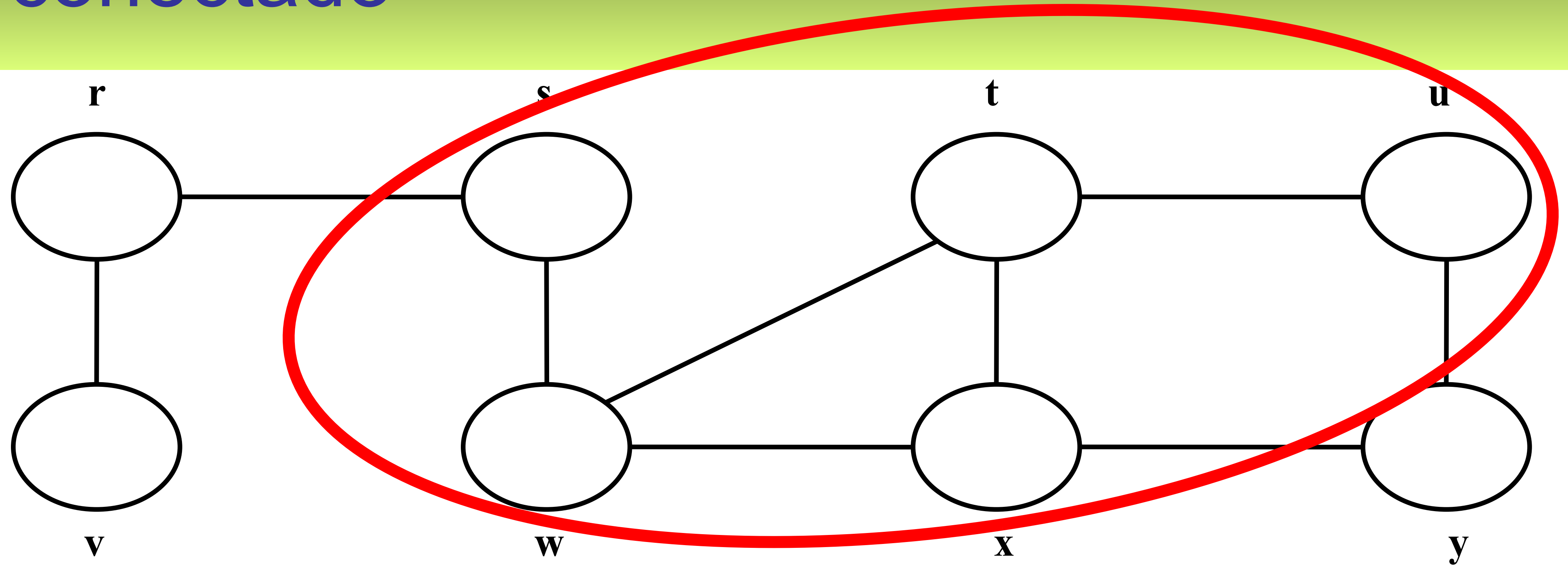
Grafo conectado



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?

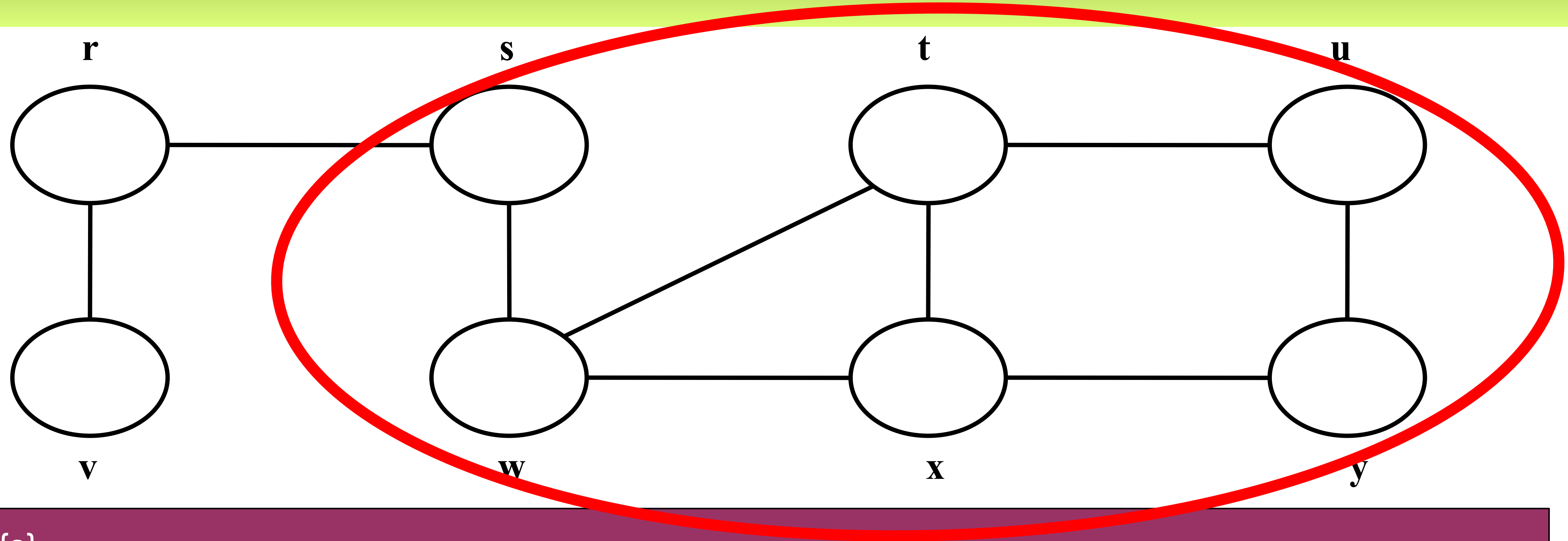
Grafo conectado



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?

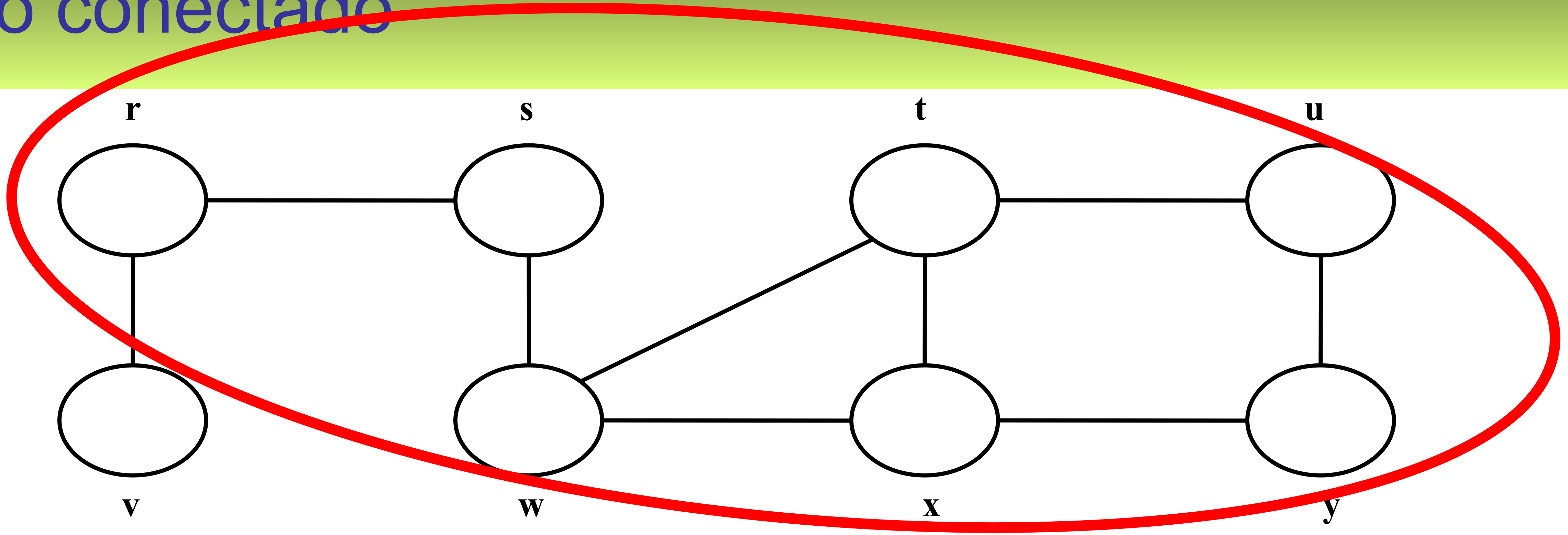
Grafo conectado



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?

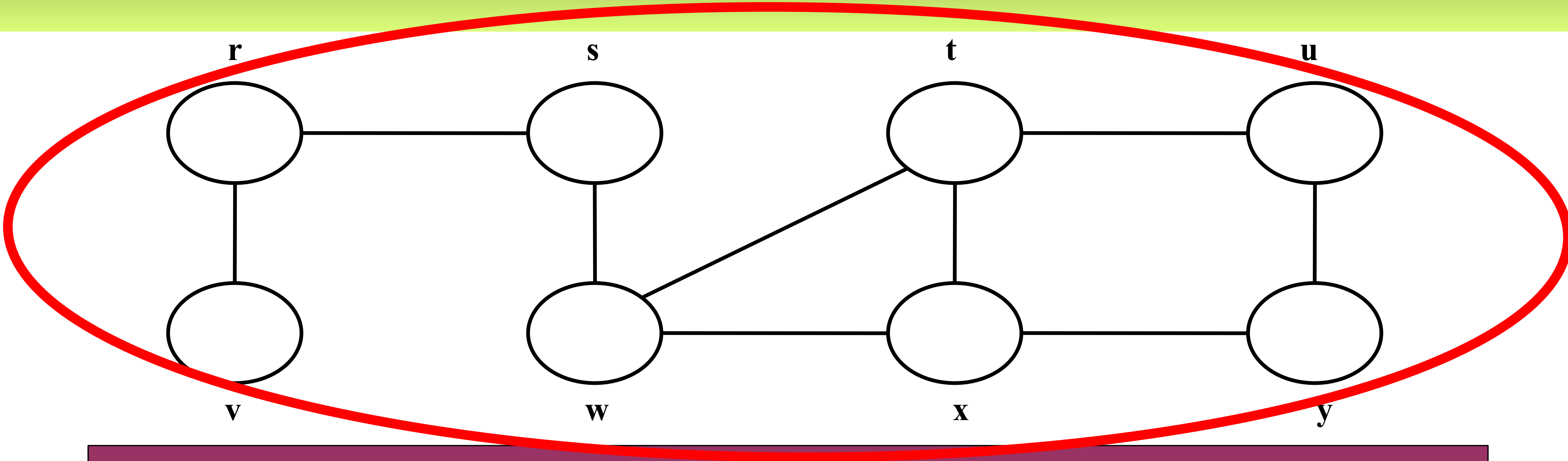
Grafo conectado



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}
```

- 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}
```

- 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

Depth-first-search (DFS)

A ideia é prosseguir 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**.

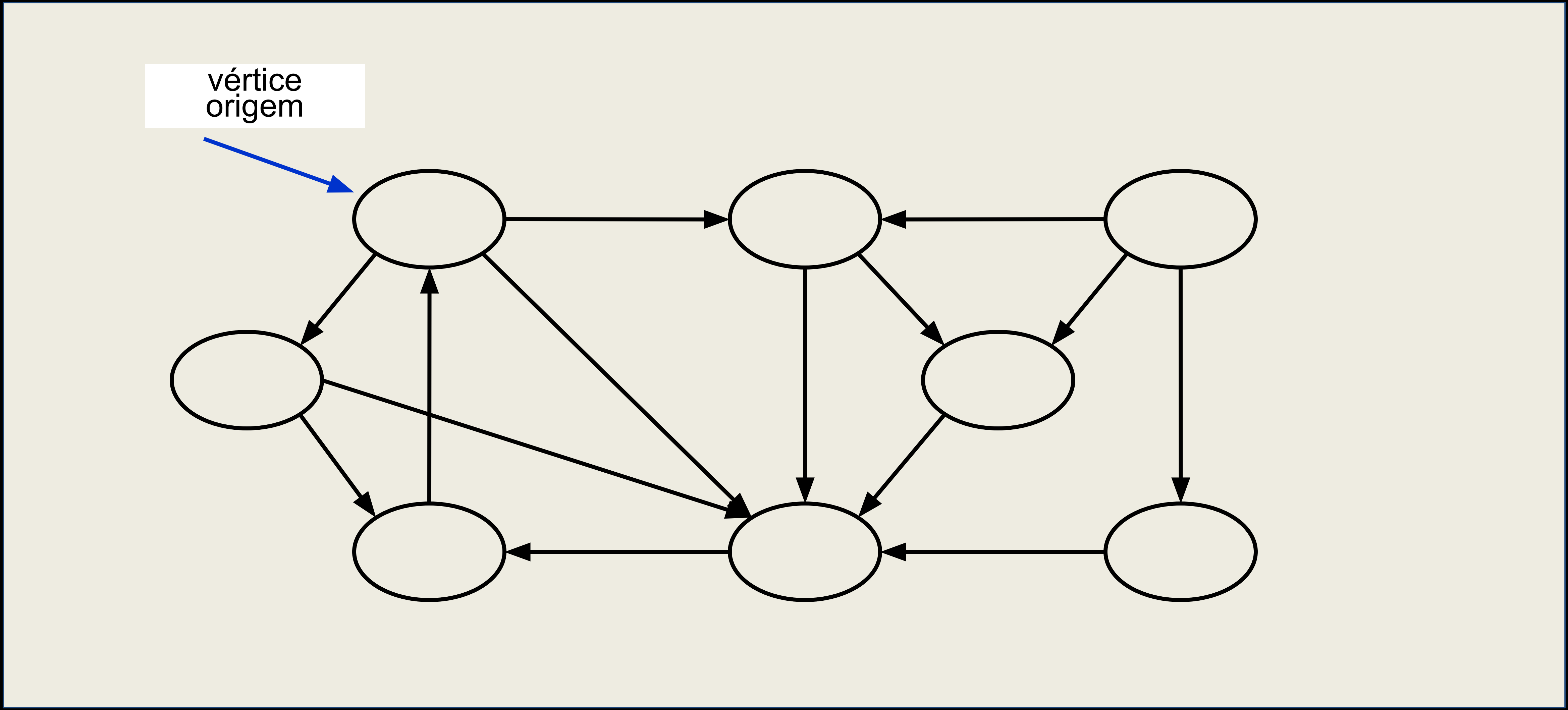
Busca em profundidade

Os vértices recebem 2 rótulos:

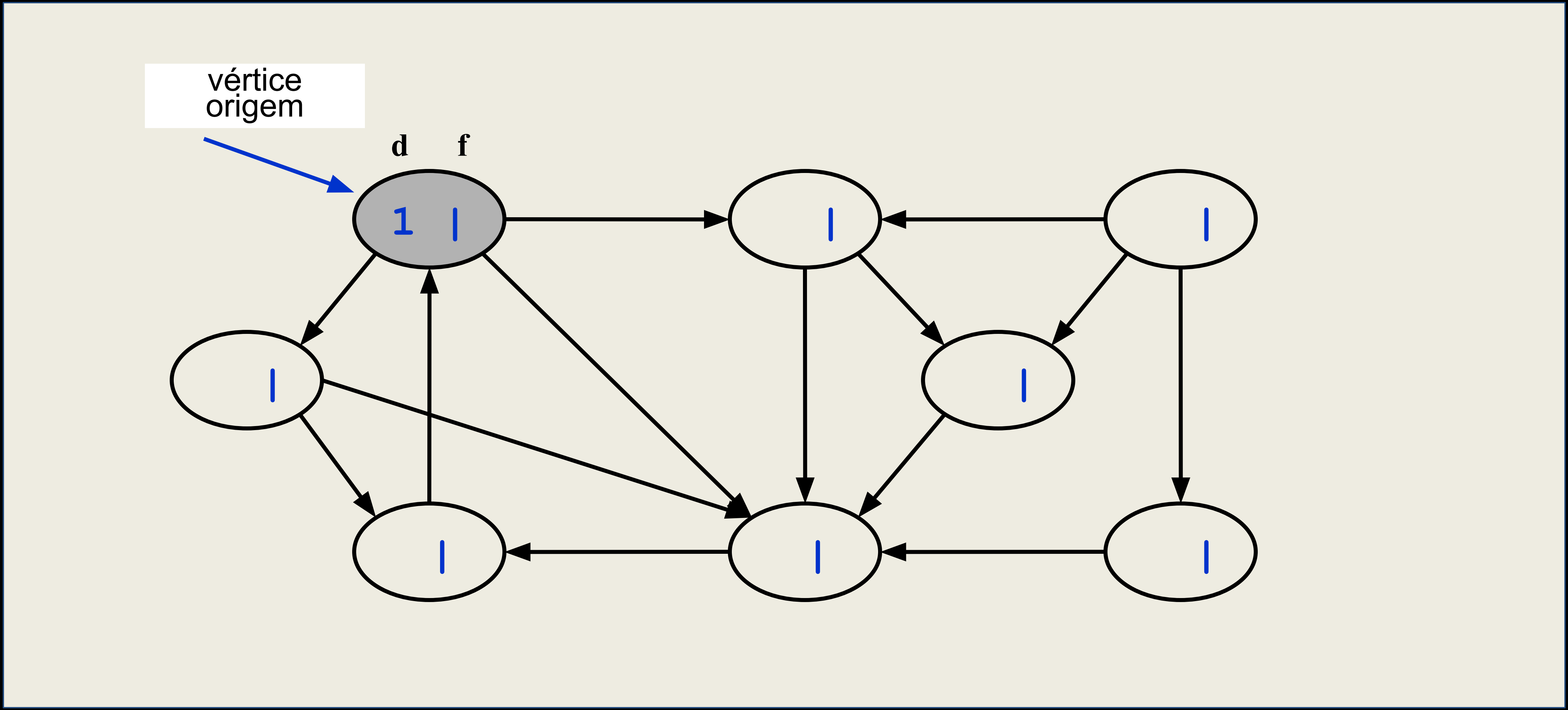
- d[.]: o momento em que o vértice foi descoberto (tornou-se cinza).
- f[.]: o momento em que examinamos os seus vizinhos (tornou-se preto).

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

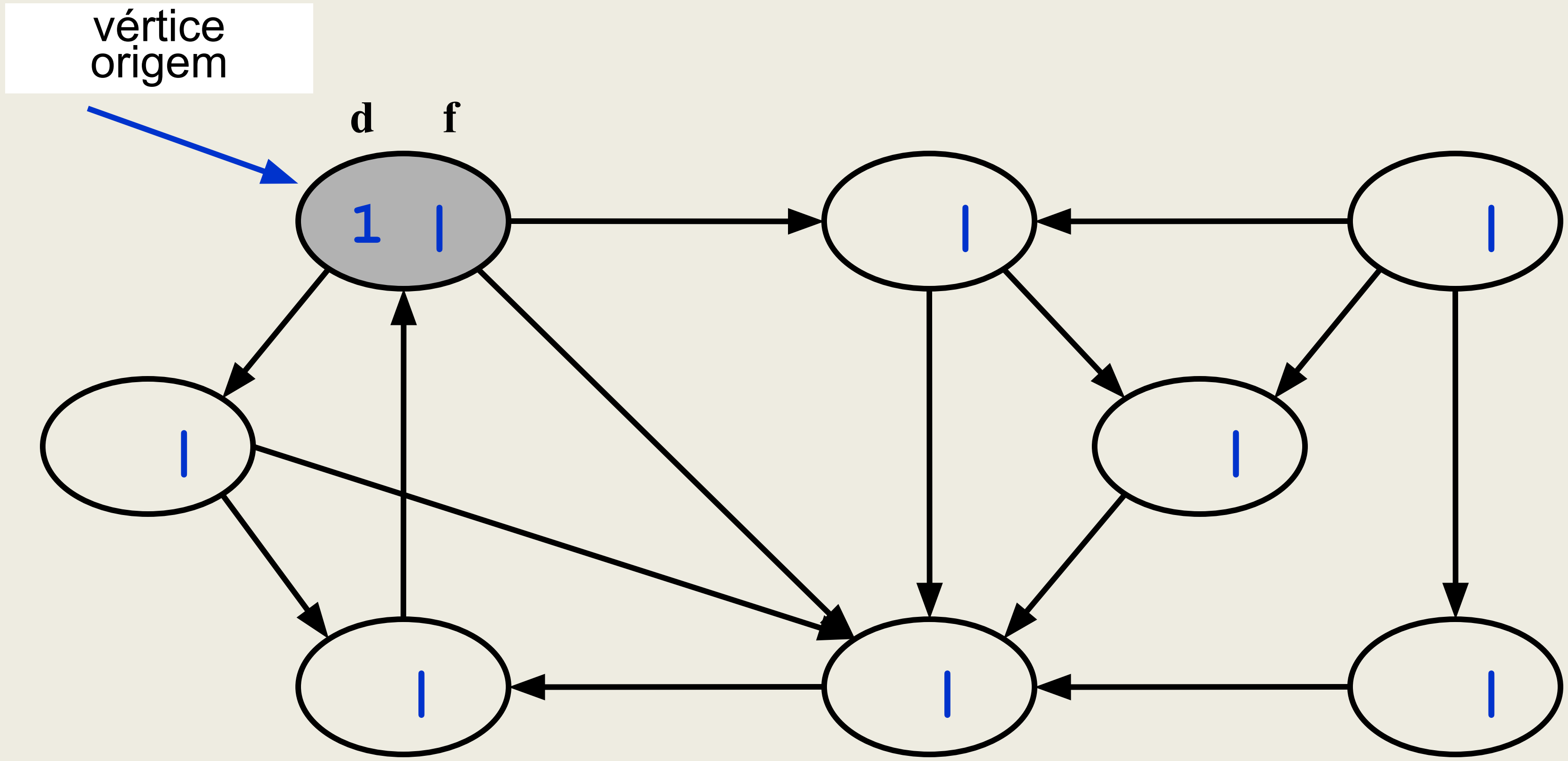
Busca em profundidade



Busca em profundidade

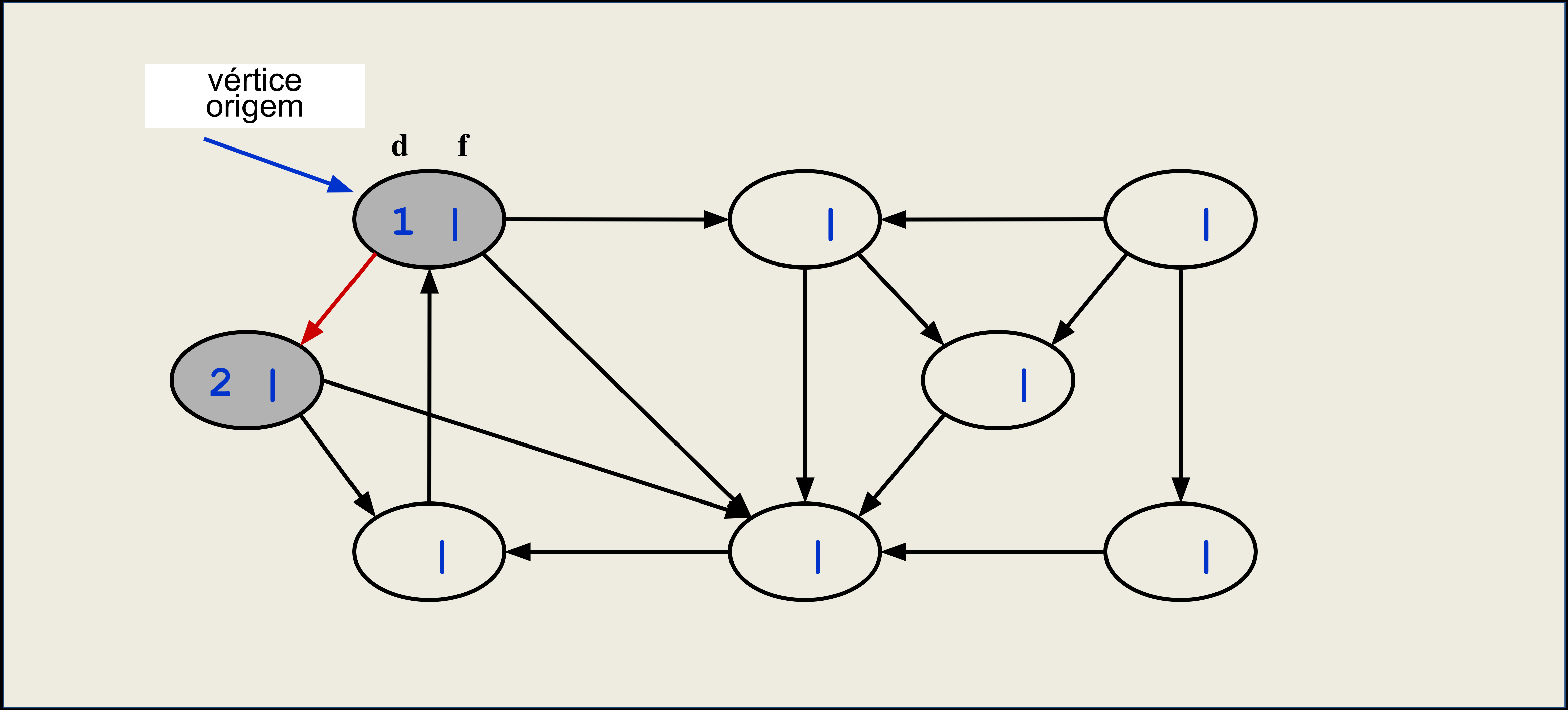


Busca em profundidade

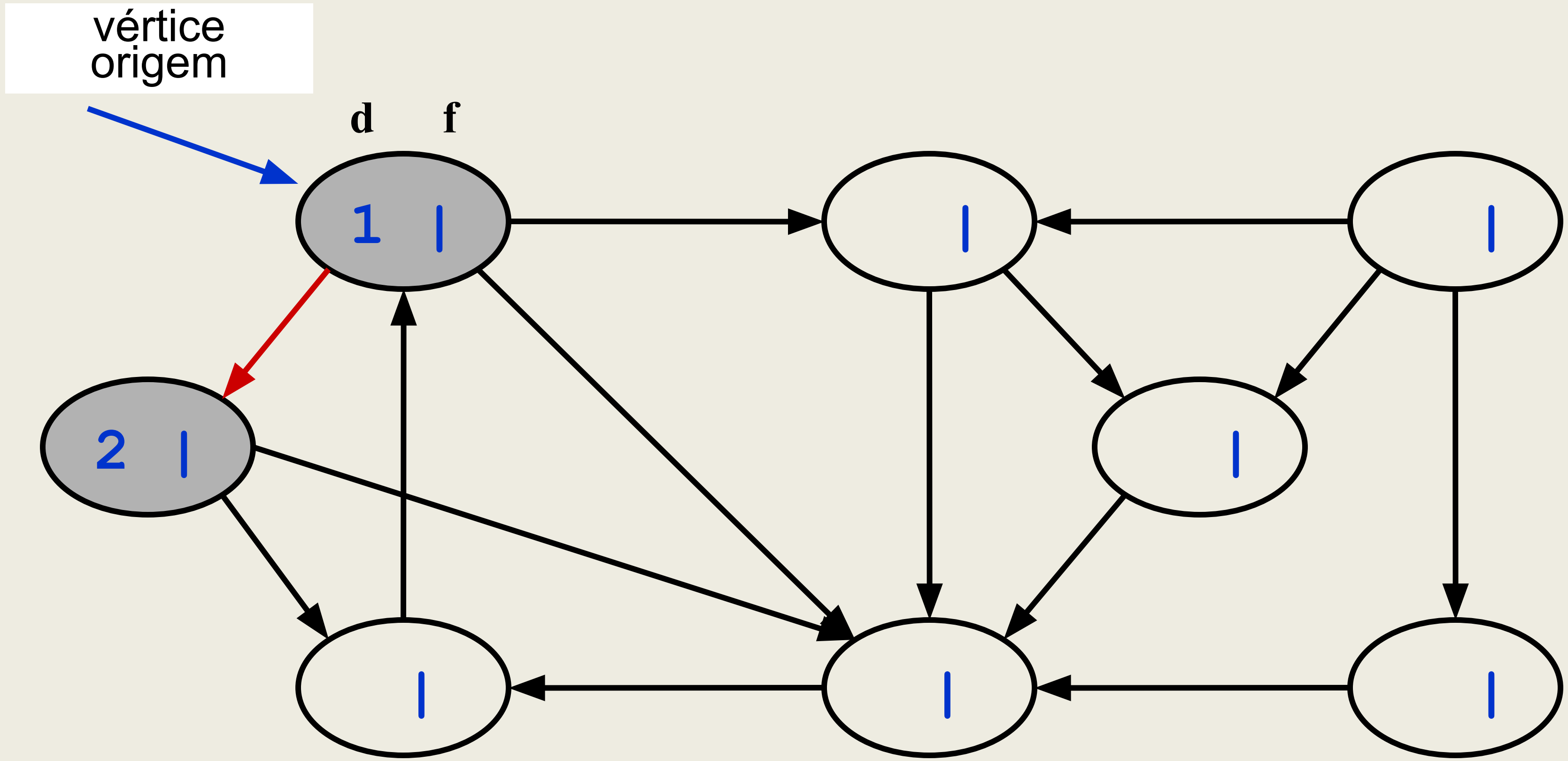


Existe algum vértice adjacente ao vértice que não tenha sido descoberto?

Busca em profundidade

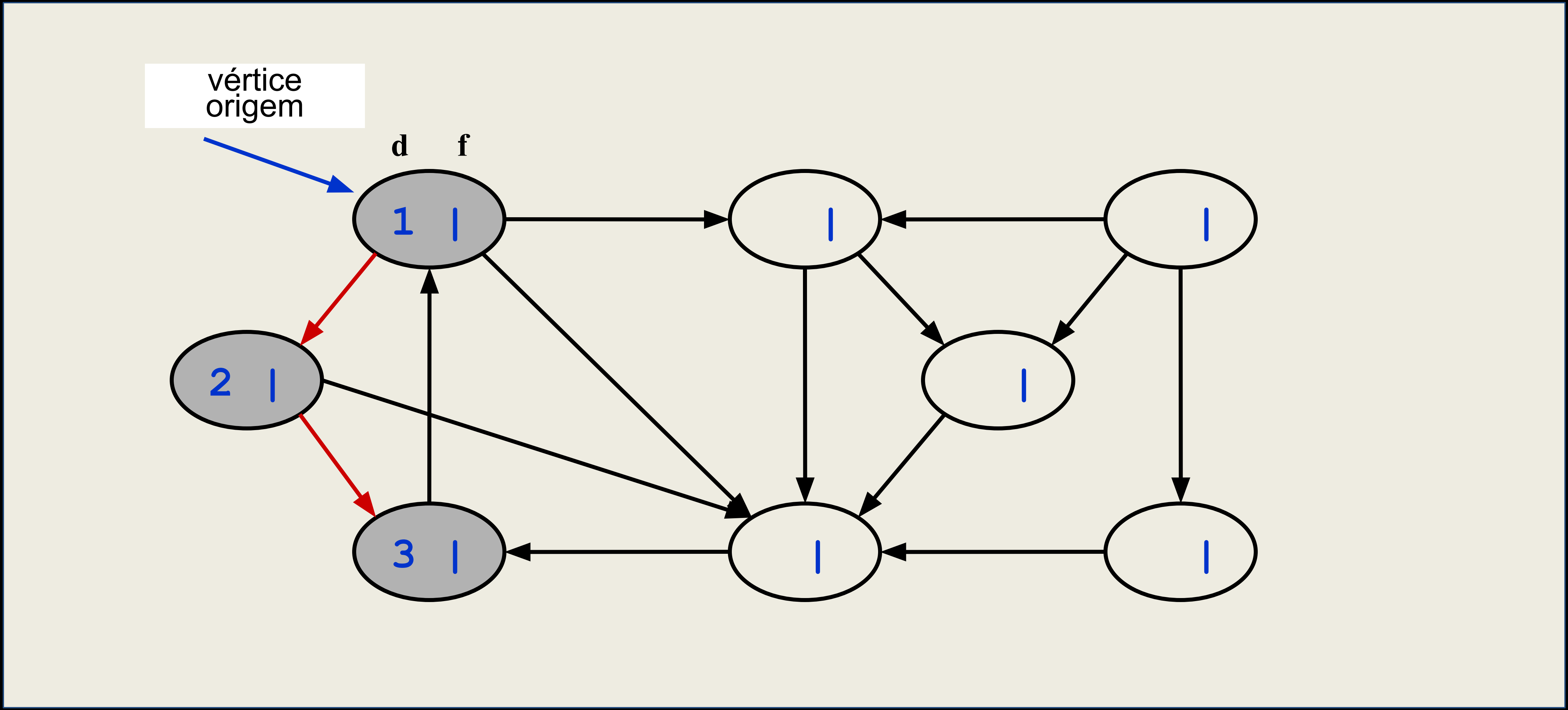


Busca em profundidade

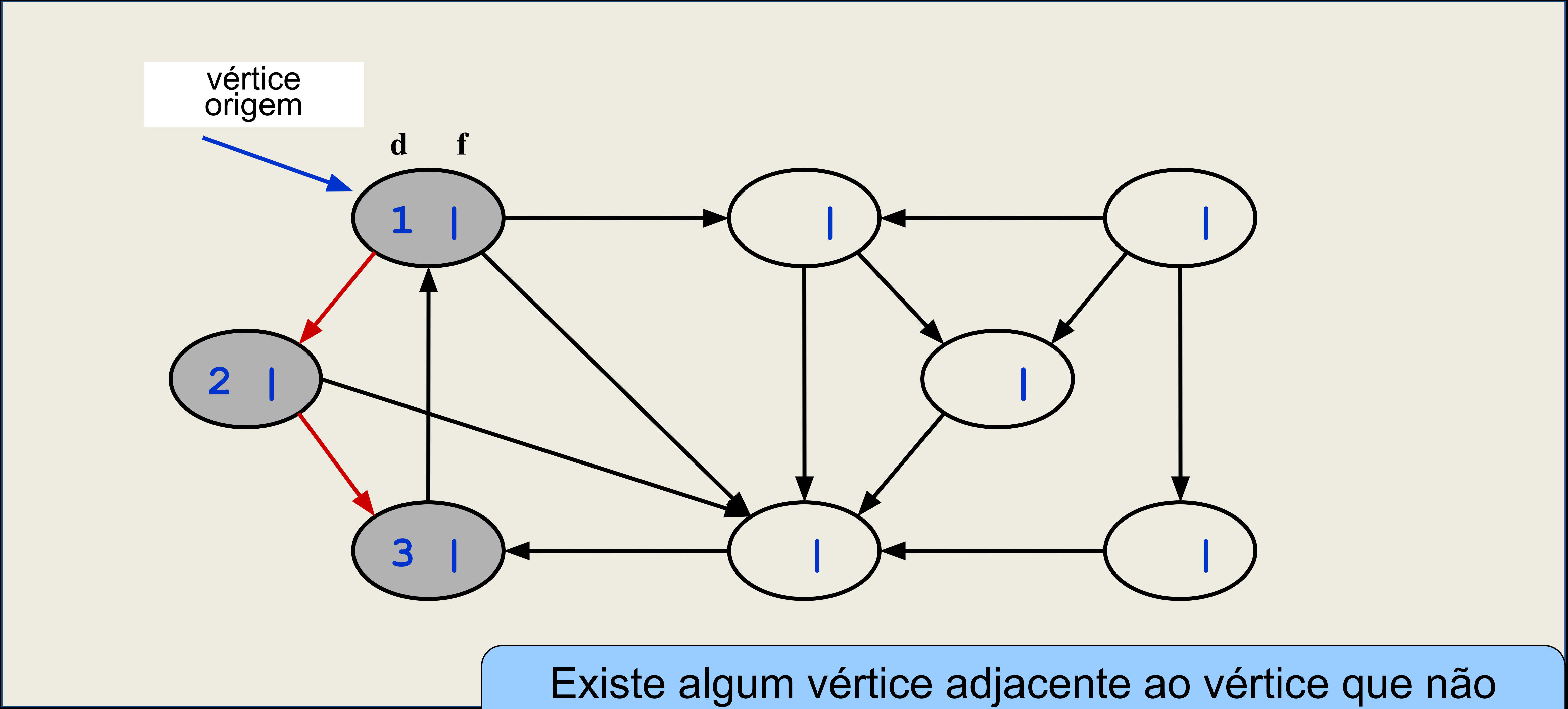


Existe algum vértice adjacente ao vértice que não tenha sido descoberto?

Busca em profundidade

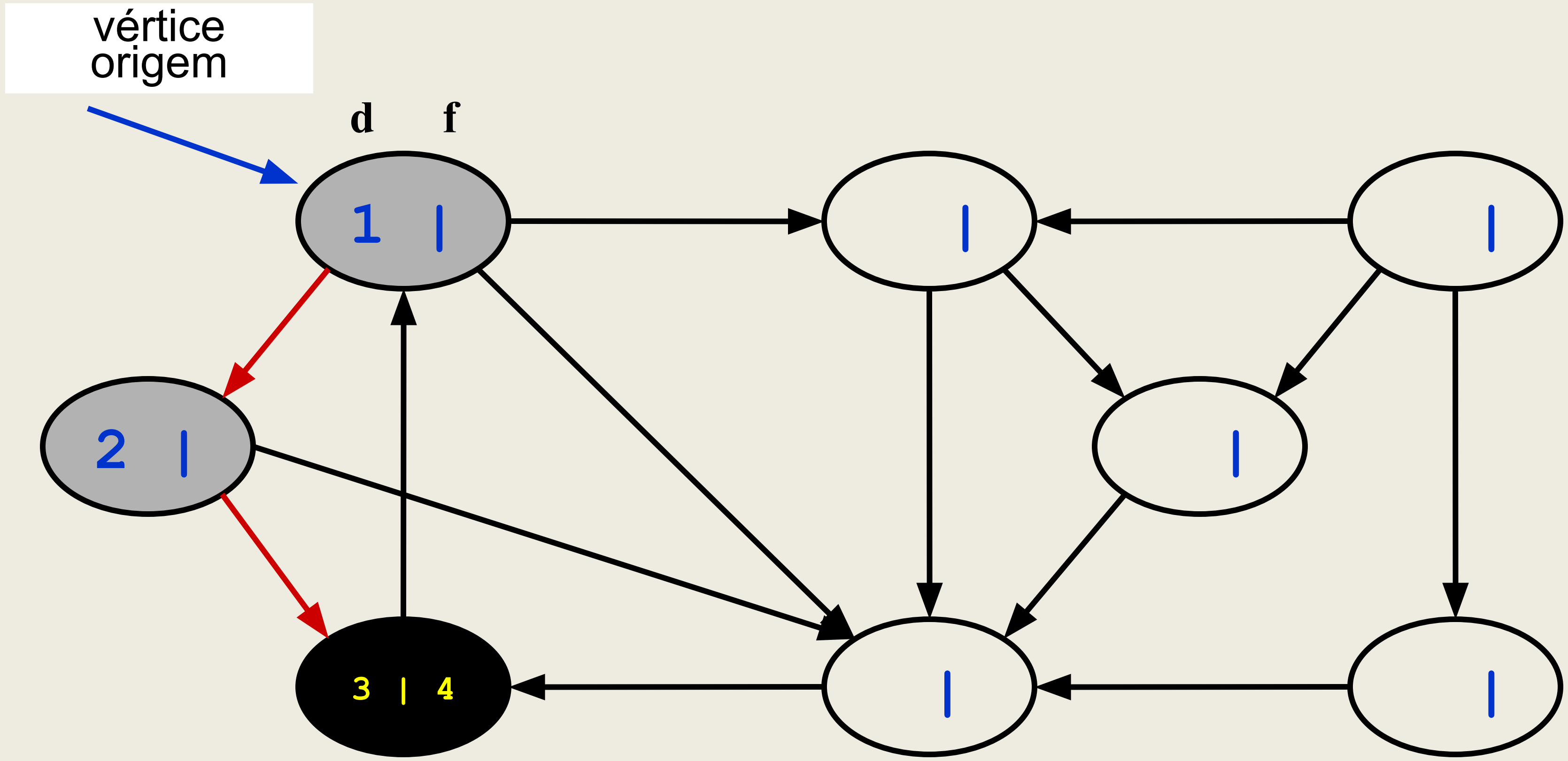


Busca em profundidade



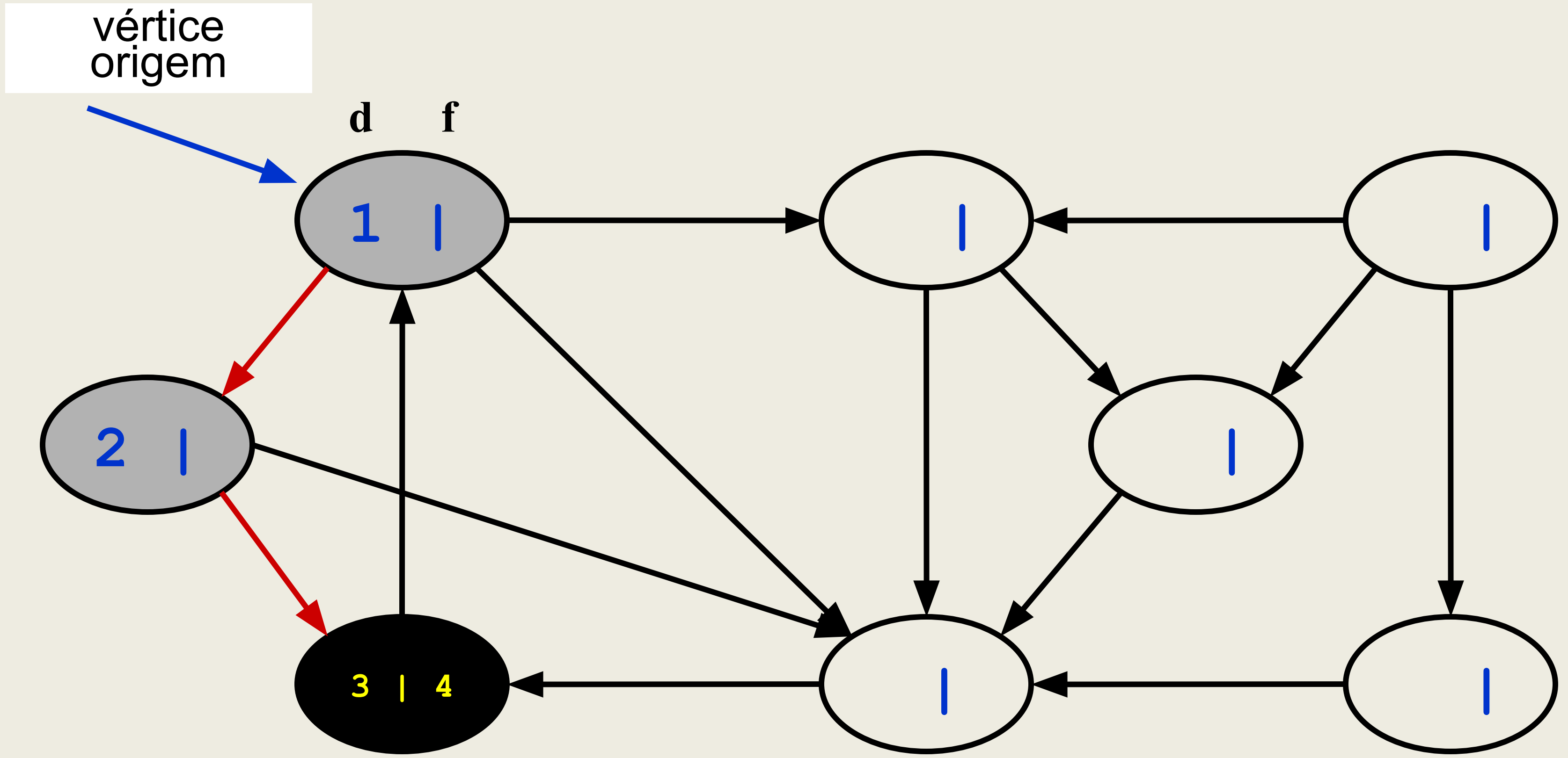
Existe algum vértice adjacente ao vértice que não tenha sido descoberto?

Busca em profundidade



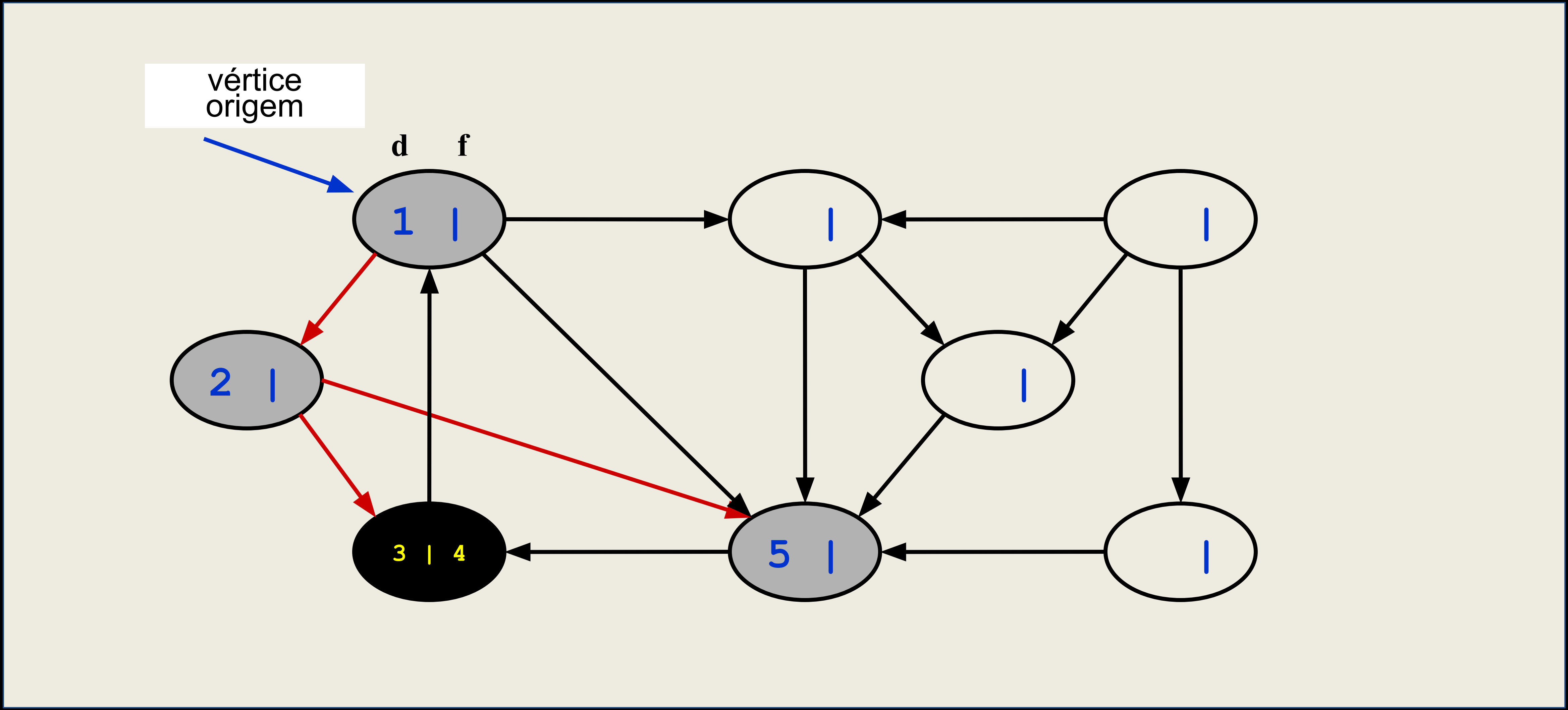
Não existe. Então, terminei com o vértice (pinta ele de preto)
Volta-se na busca para o precursor desse vértice.

Busca em profundidade

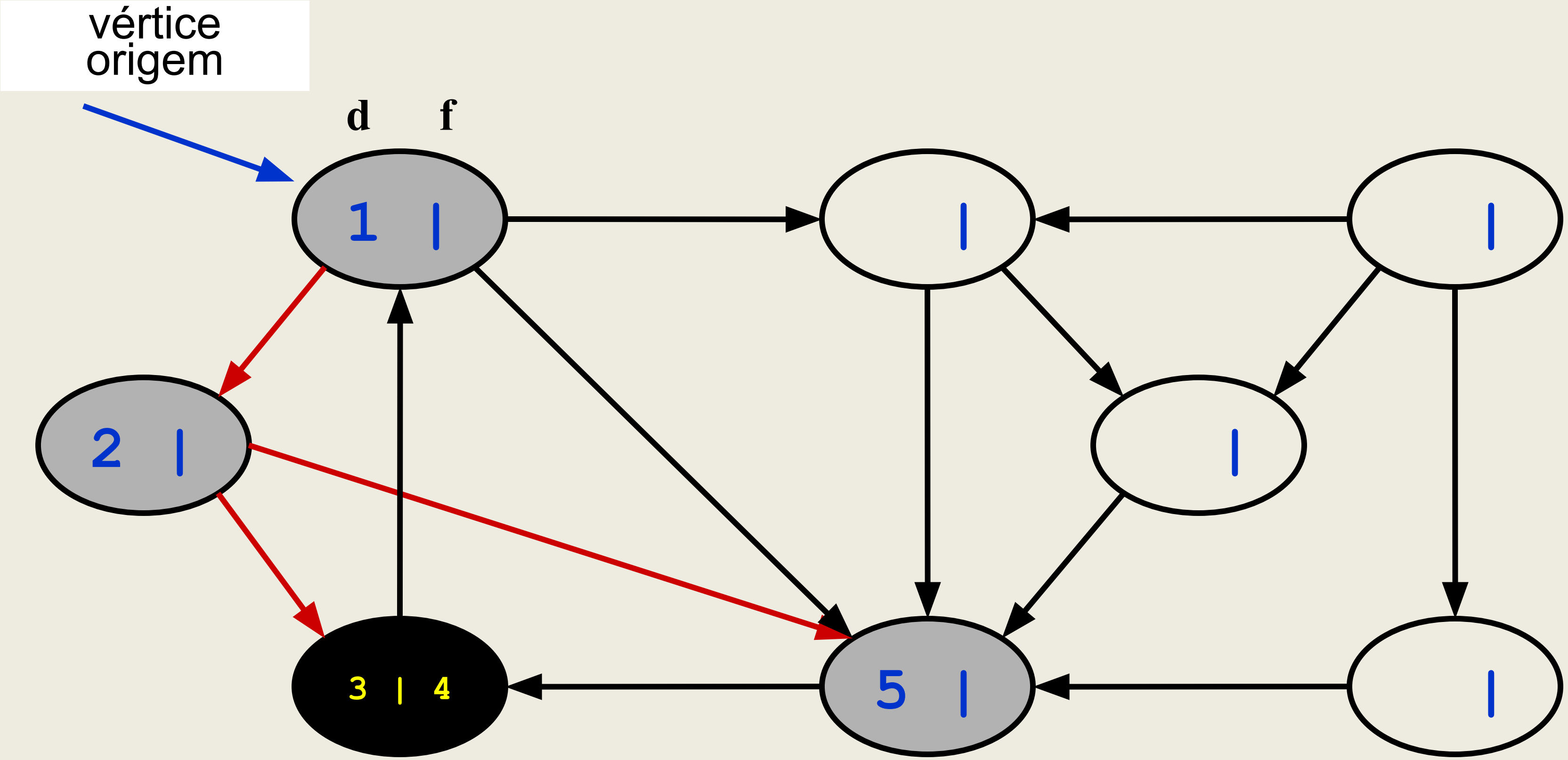


Existe algum vértice adjacente ao vértice que não tenha sido descoberto?

Busca em profundidade

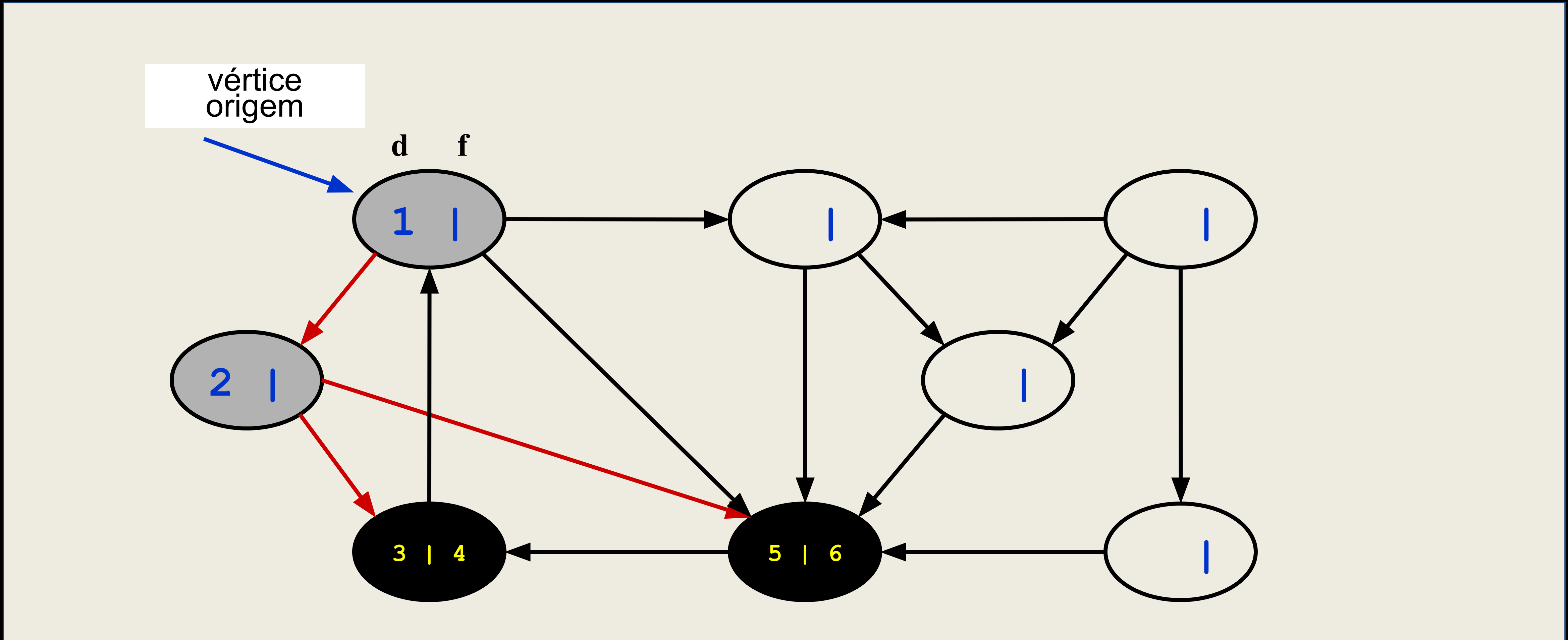


Busca em profundidade



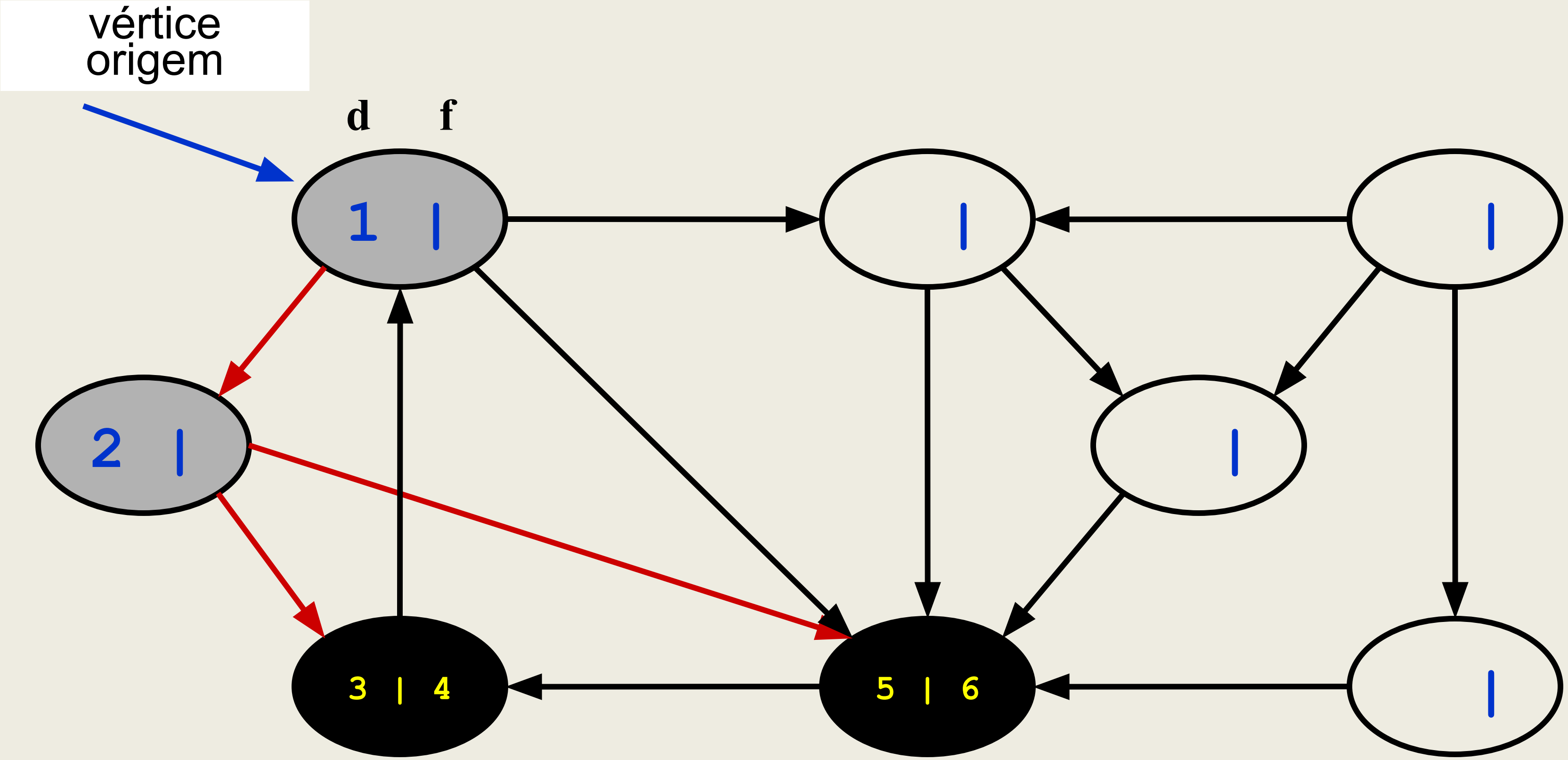
Existe algum vértice adjacente ao vértice que não tenha sido descoberto?

Busca em profundidade



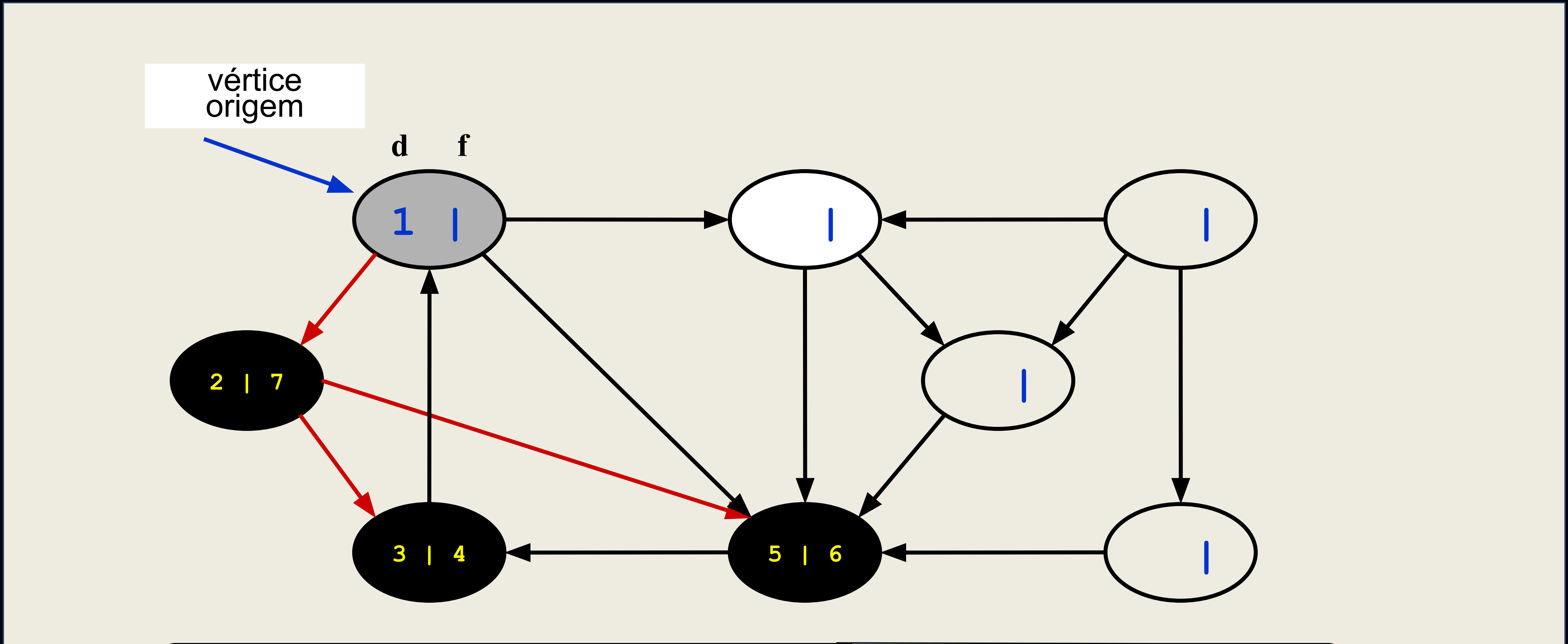
Não existe. Terminei!
Volta-se na busca para o precursor desse vértice.

Busca em profundidade



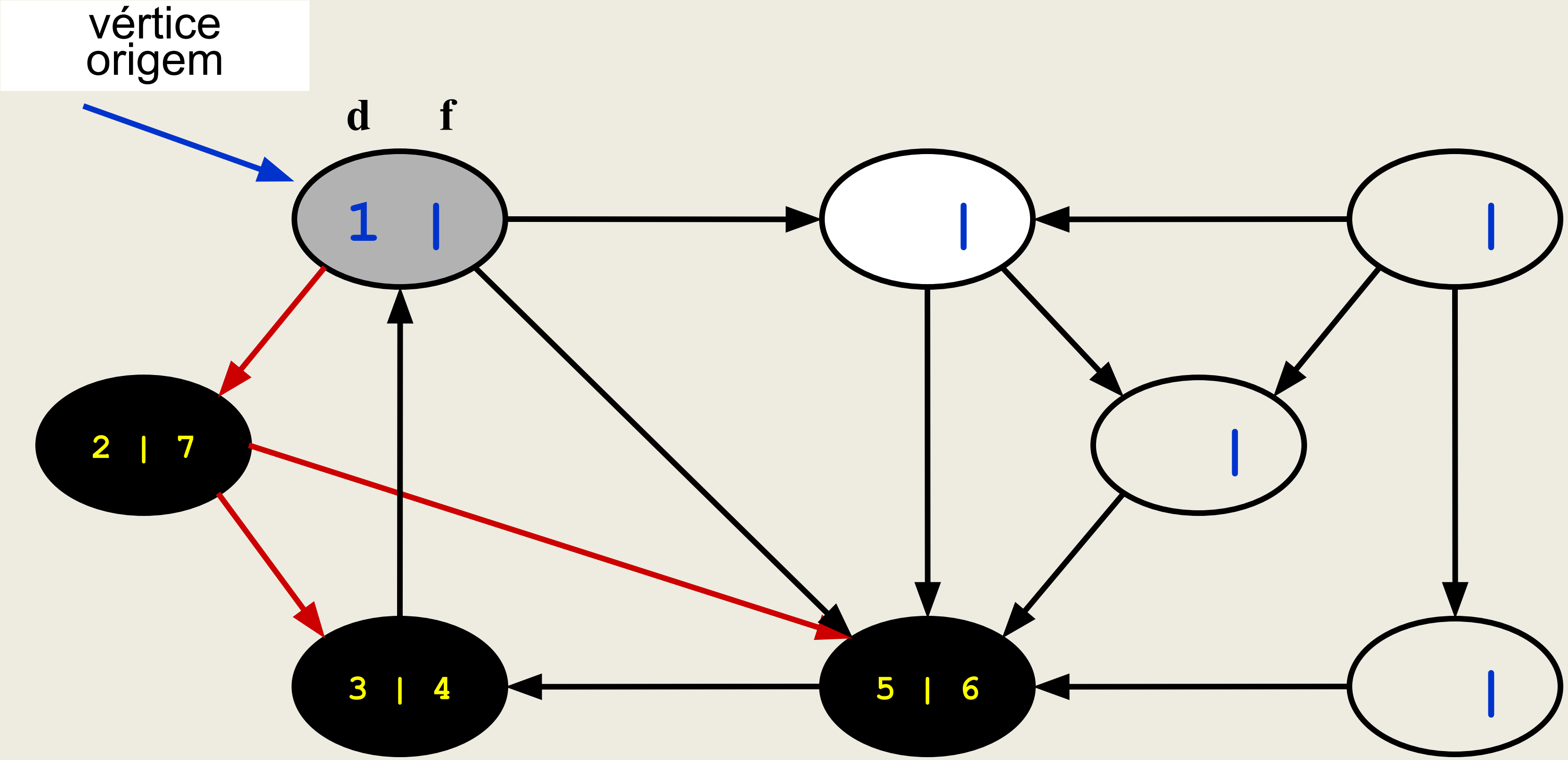
Existe algum vértice adjacente ao vértice que não tenha sido descoberto?

Busca em profundidade



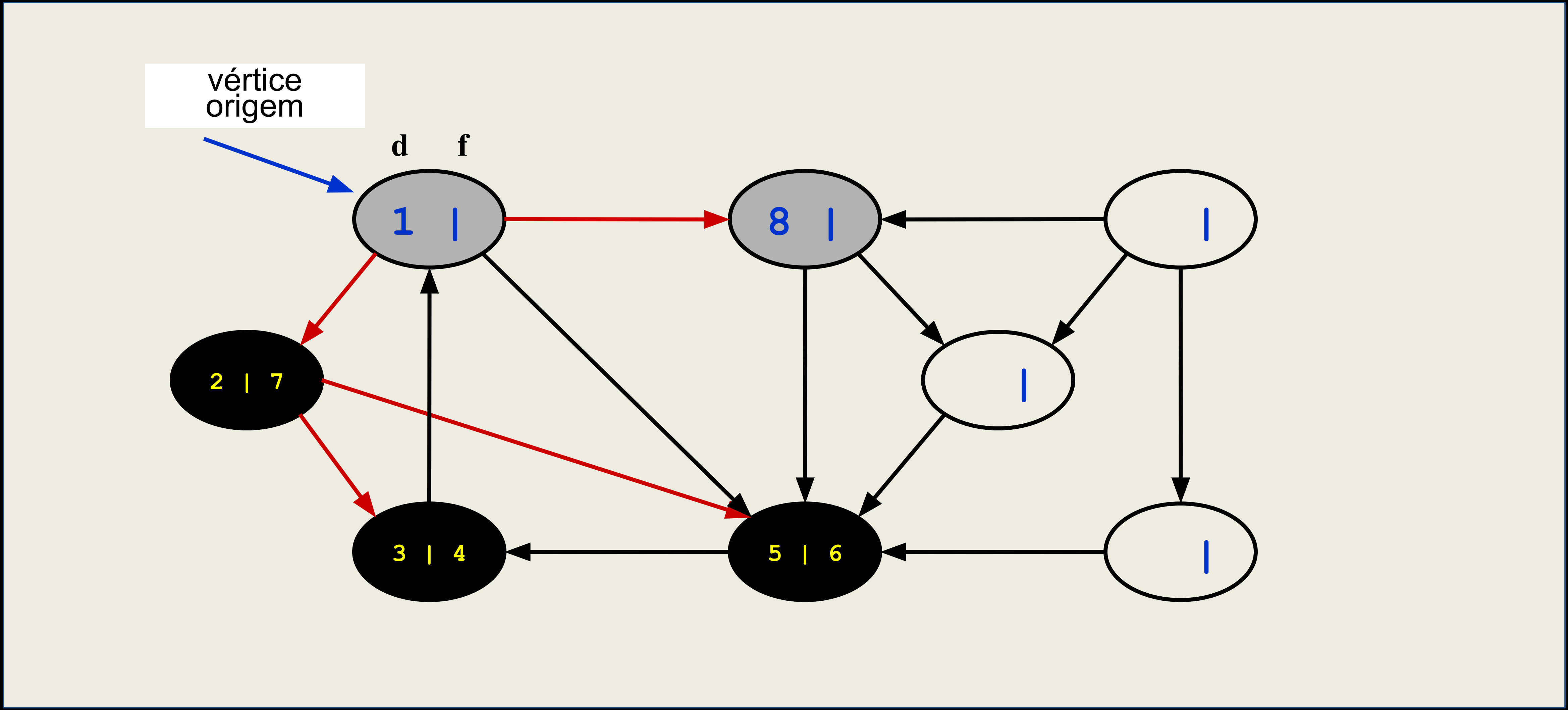
Não existe. Terminei!
Volta-se na busca para o precursor desse vértice.

Busca em profundidade

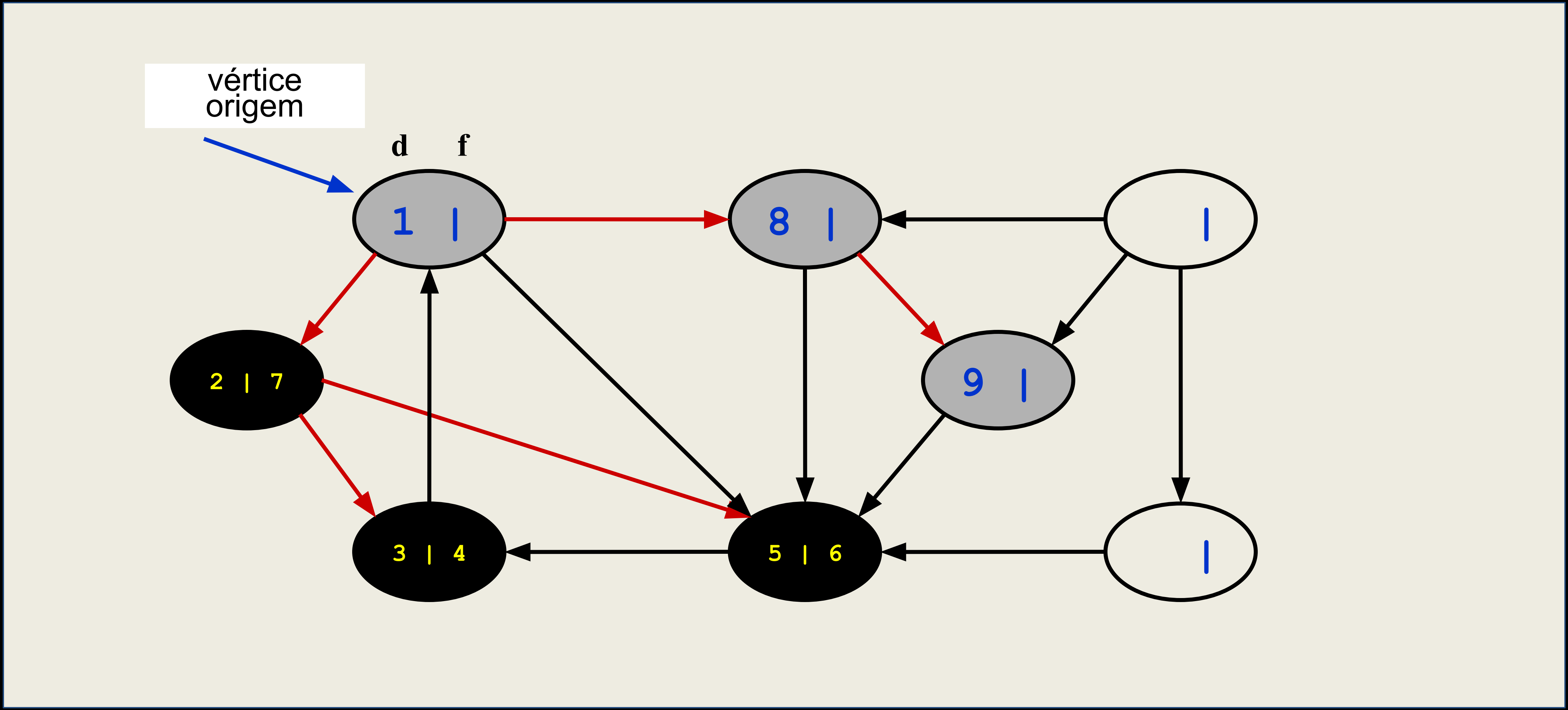


Existe mais algum vértice adjacente ao vértice que não tenha sido descoberto?

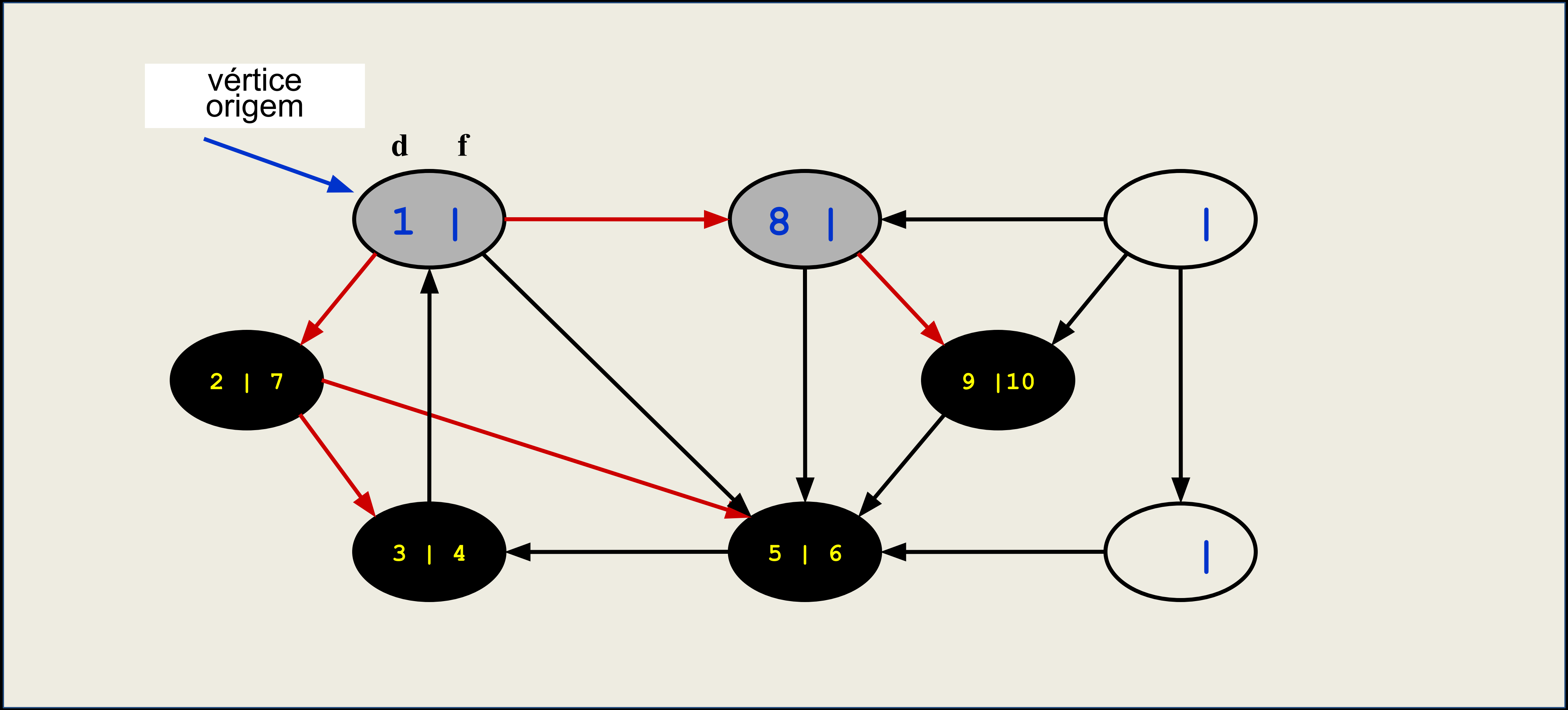
Busca em profundidade



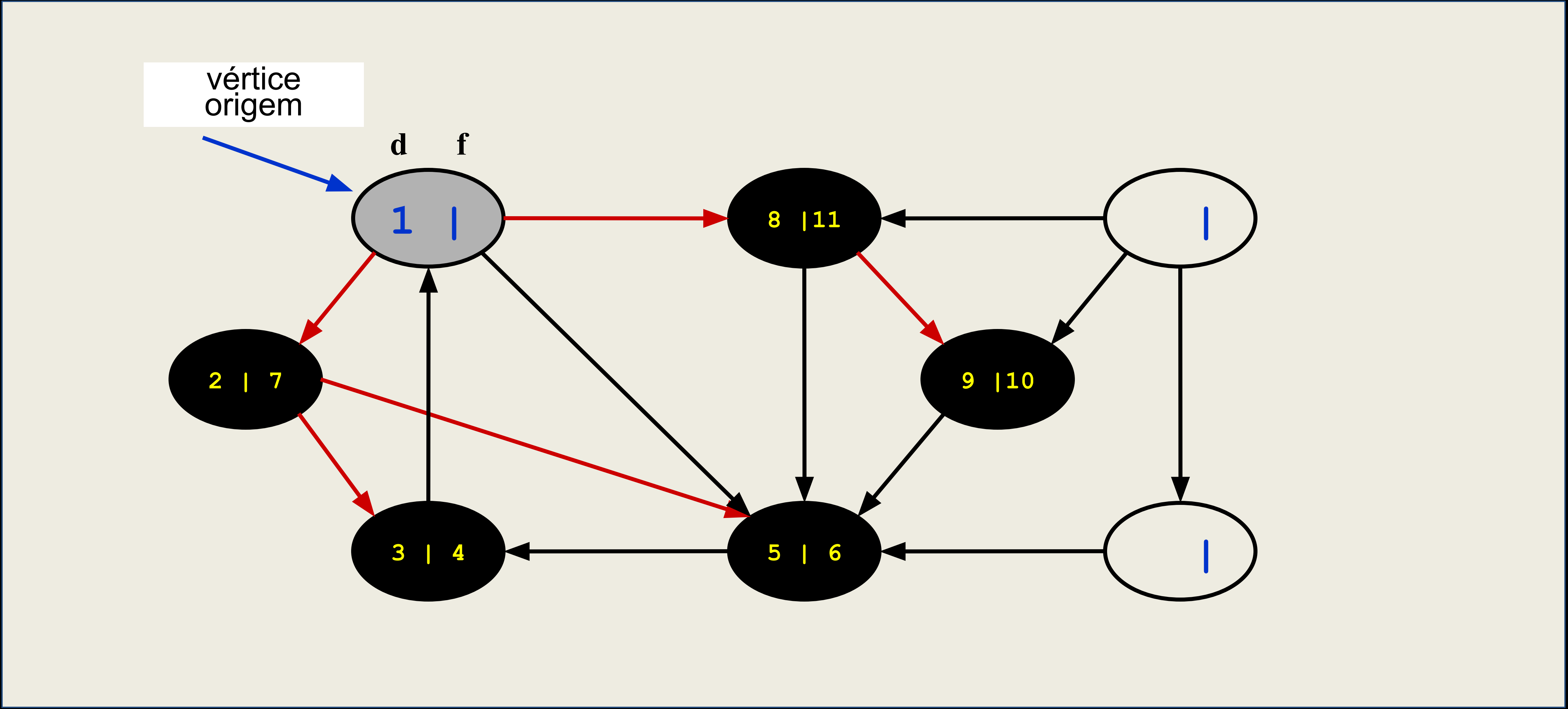
Busca em profundidade



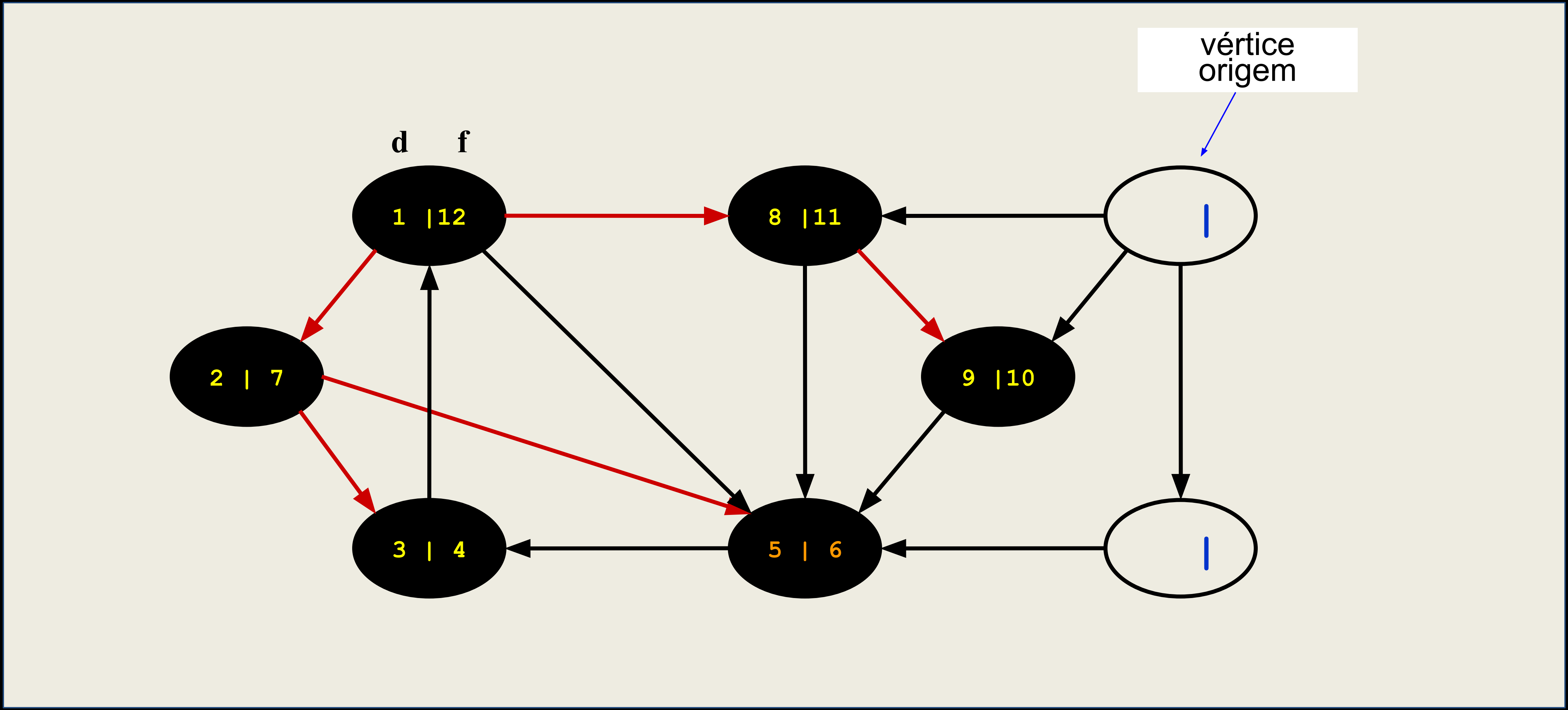
Busca em profundidade



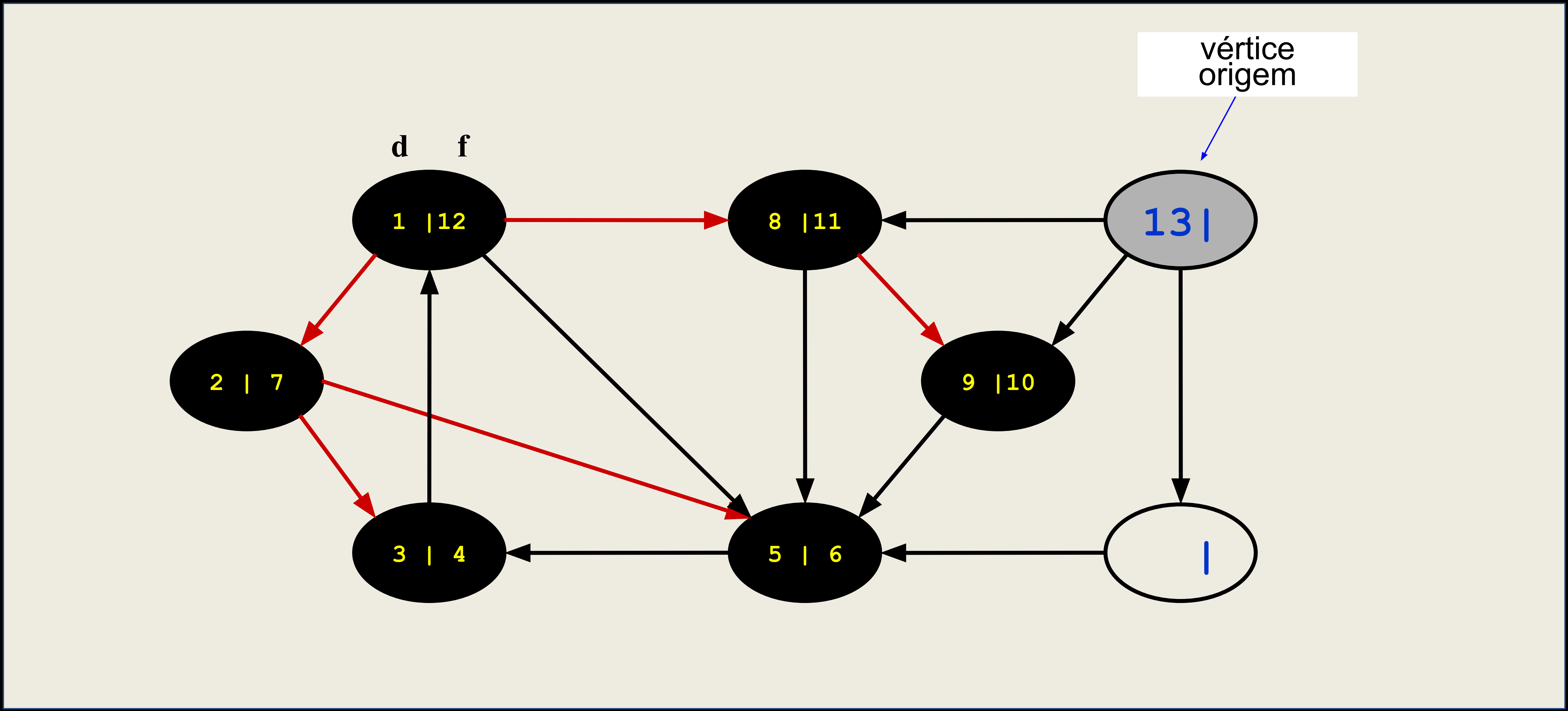
Busca em profundidade



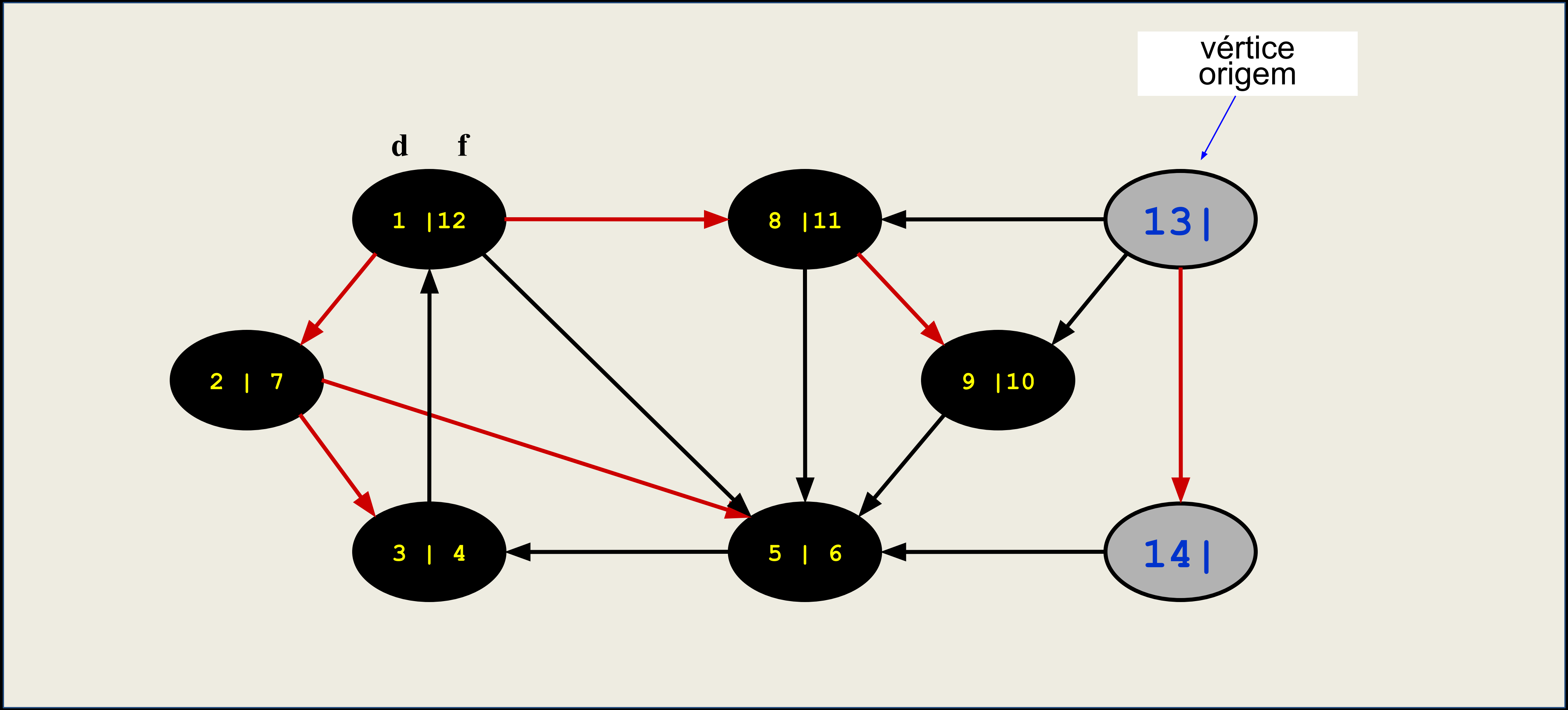
Busca em profundidade



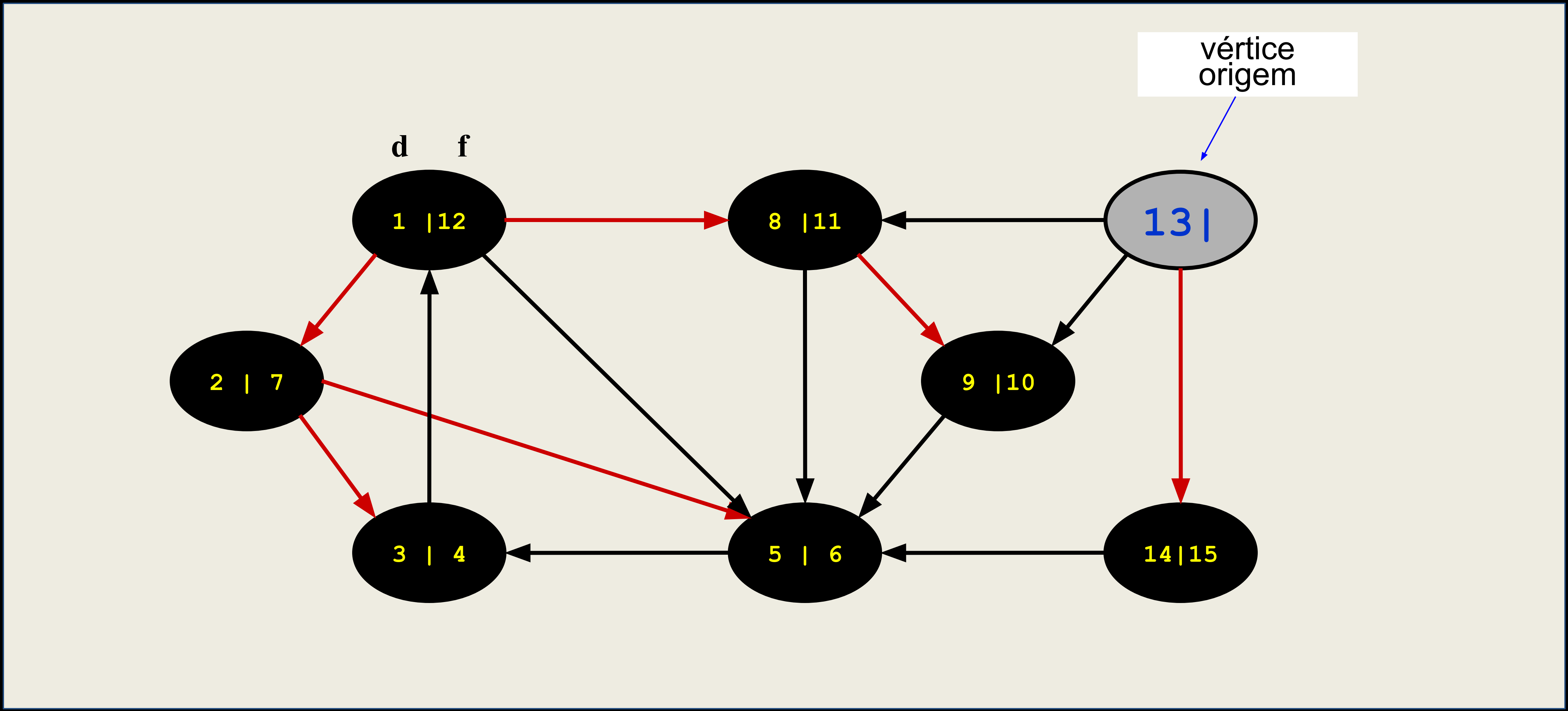
Busca em profundidade



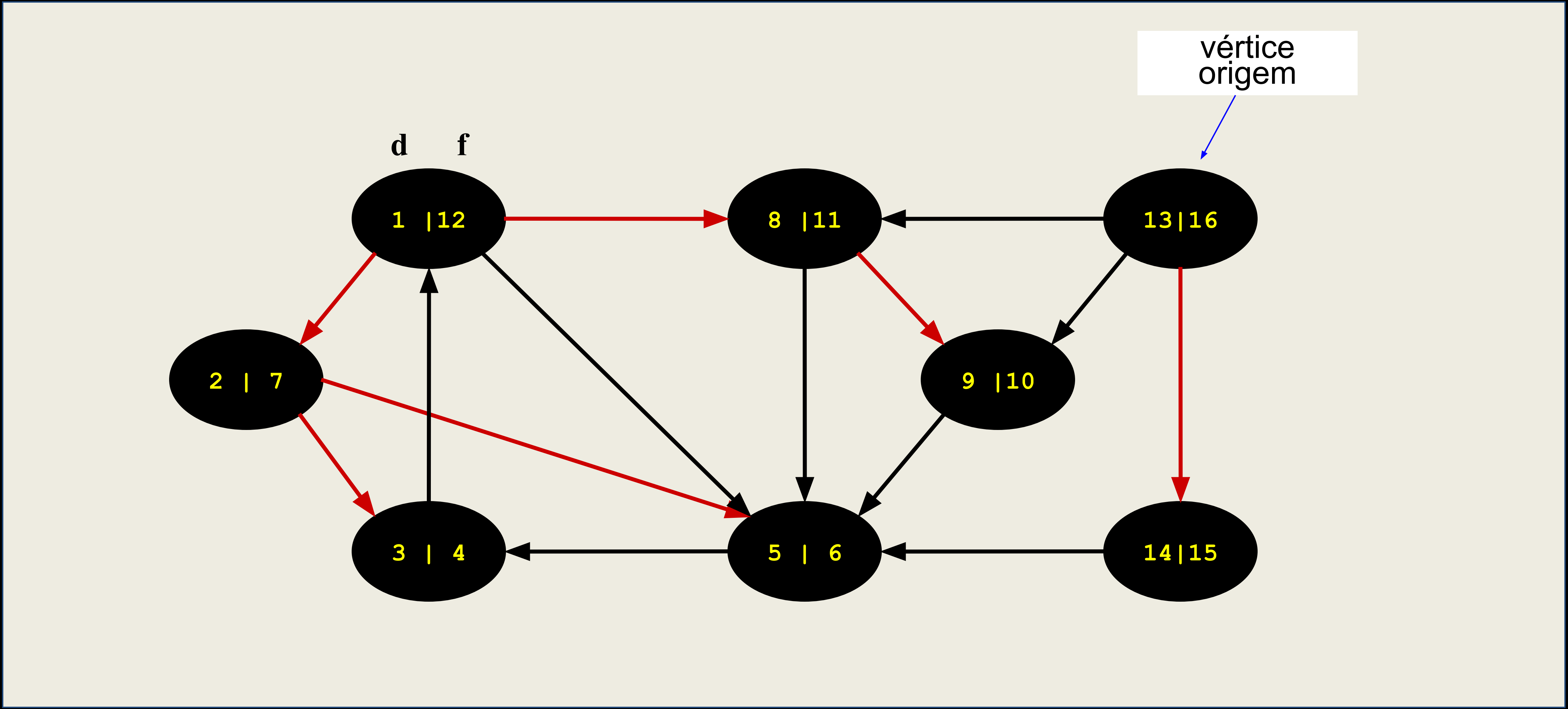
Busca em profundidade



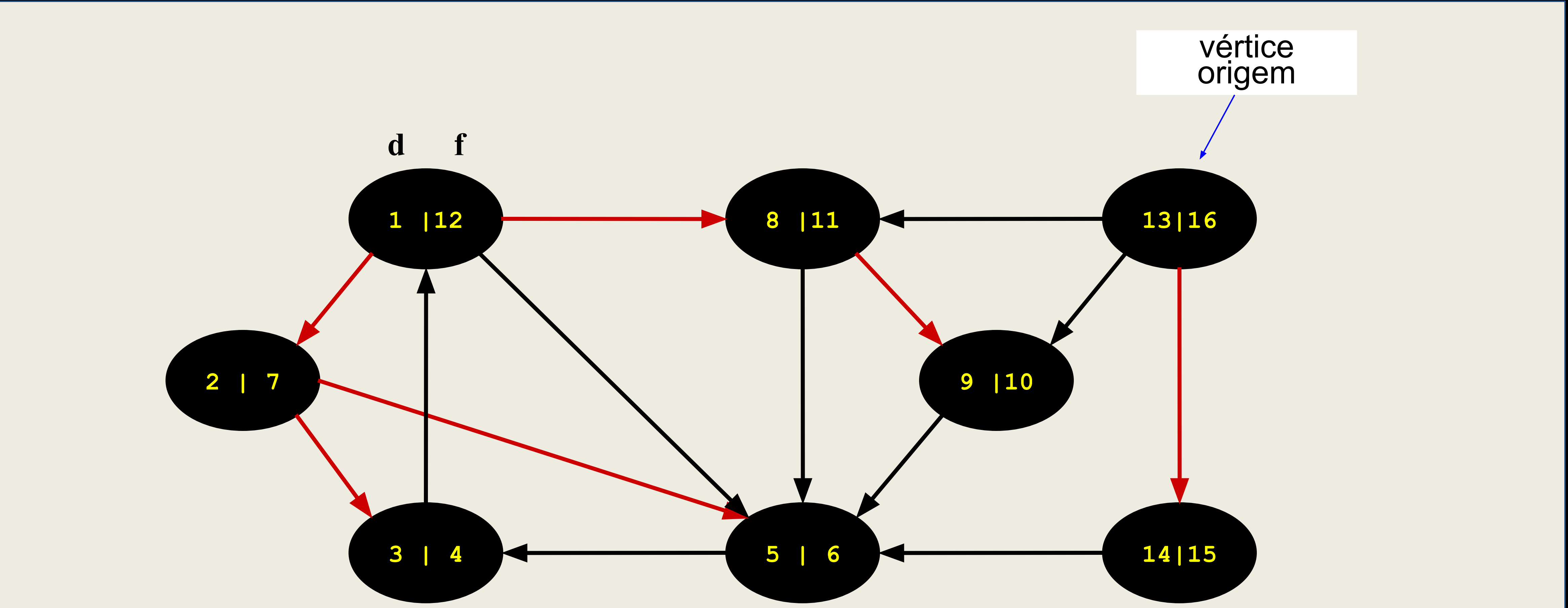
Busca em profundidade



Busca em profundidade

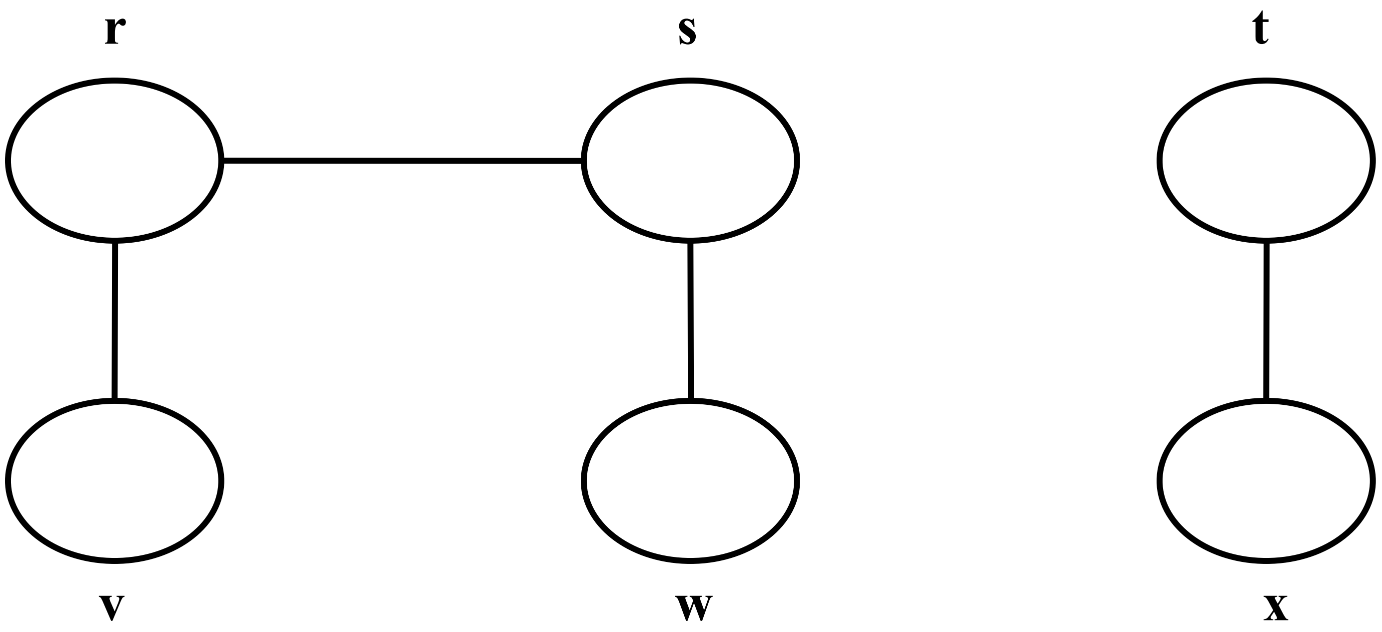


Busca em profundidade



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-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
```

Busca em profundidade

```
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                            ▹
9.    time ← time + 1
10.   f[u] ← time
```

DFS-Visit cria uma árvore DFS com raiz em u

Busca em profundidade

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

```
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
```

Busca em profundidade

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 ▷ criar a floresta
6. if color[u] = WHITE
7. then DFS-Visit(u) ▷ criar uma nova árvore a partir de u

O(V)

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 ▷ explora a aresta (u, v)
5. if color[v] = WHITE
6. then π[v] ← u
7. DFS-Visit(v)
8. color[u] ← BLACK ▷ os vizinhos de u já foram examinados
9. time ← time + 1
10. f[u] ← time

Busca em profundidade

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 ▹ criar a flores
6. if color[u] = WHITE
7. then DFS-Visit(u) ▹ criar uma nova árvore a partir de u

Quantas vezes é chamado DFS-Visit para cada vértice?

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 ▹ explora a aresta (u, v)
5. if color[v] = WHITE
6. then π[v] ← u
7. DFS-Visit(v)
8. color[u] ← BLACK ▹ os vizinhos de u já foram examinados
9. time ← time + 1
10. f[u] ← time

Busca em profundidade

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

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

```
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
```

Busca em profundidade

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

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
```

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

Busca em profundidade

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 ▹ criar a floresta
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 ▹ vértice u acabou de ser descoberto
2. time ← time + 1
3. d[u] ← time
4. for each vertex v adjacent to u ▹ explora a aresta (u, v)
5. if color[v] = WHITE
6. then π[v] ← u
7. DFS-Visit(v)
8. color[u] ← BLACK ▹ os vizinhos de u já foram examinados
9. time ← time + 1
10. f[u] ← time

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

Busca em profundidade

```
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                  ▷ vértice u ac
    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
```

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

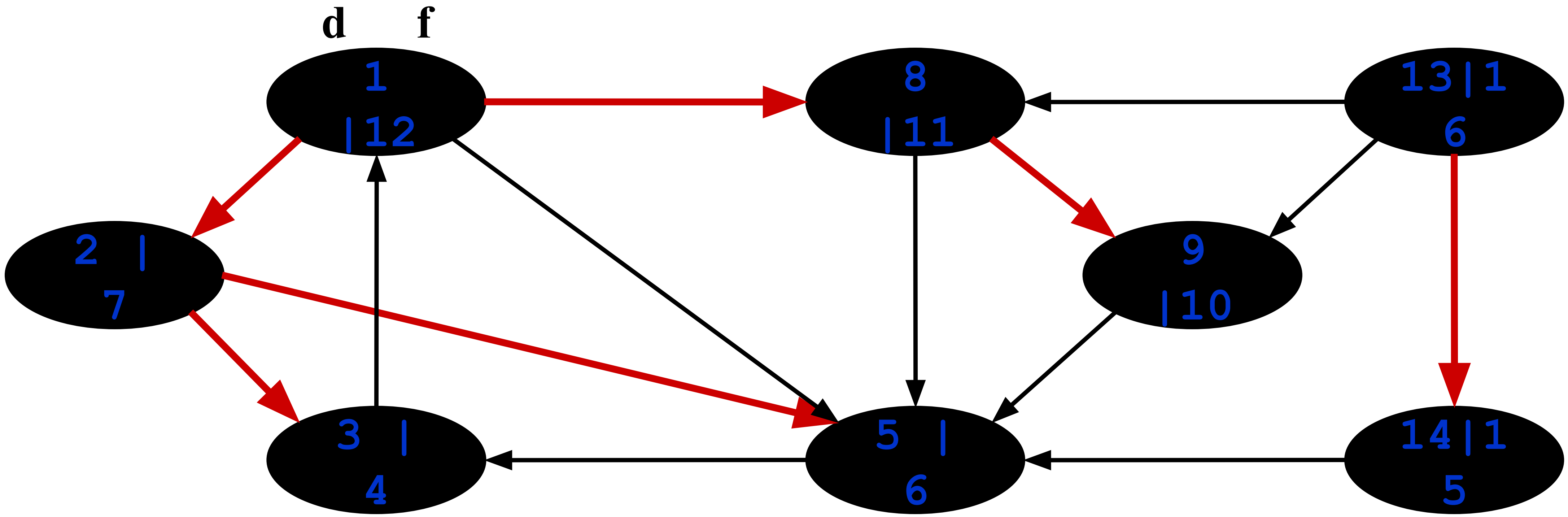
Busca em profundidade – Classificação das arestas

- 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

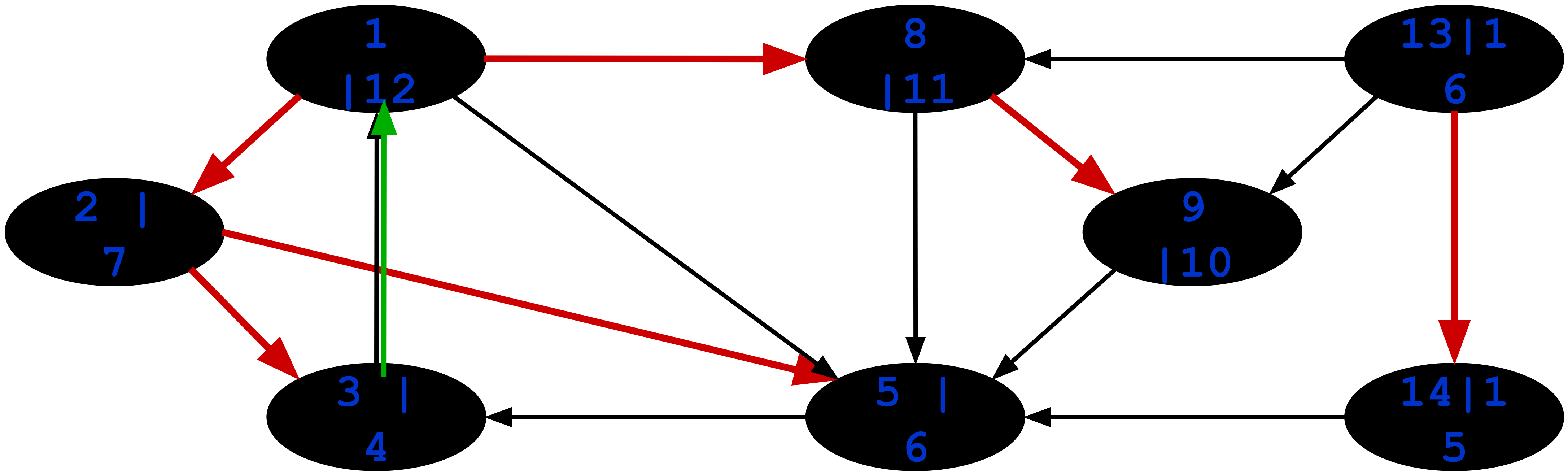


Arestas da árvore

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

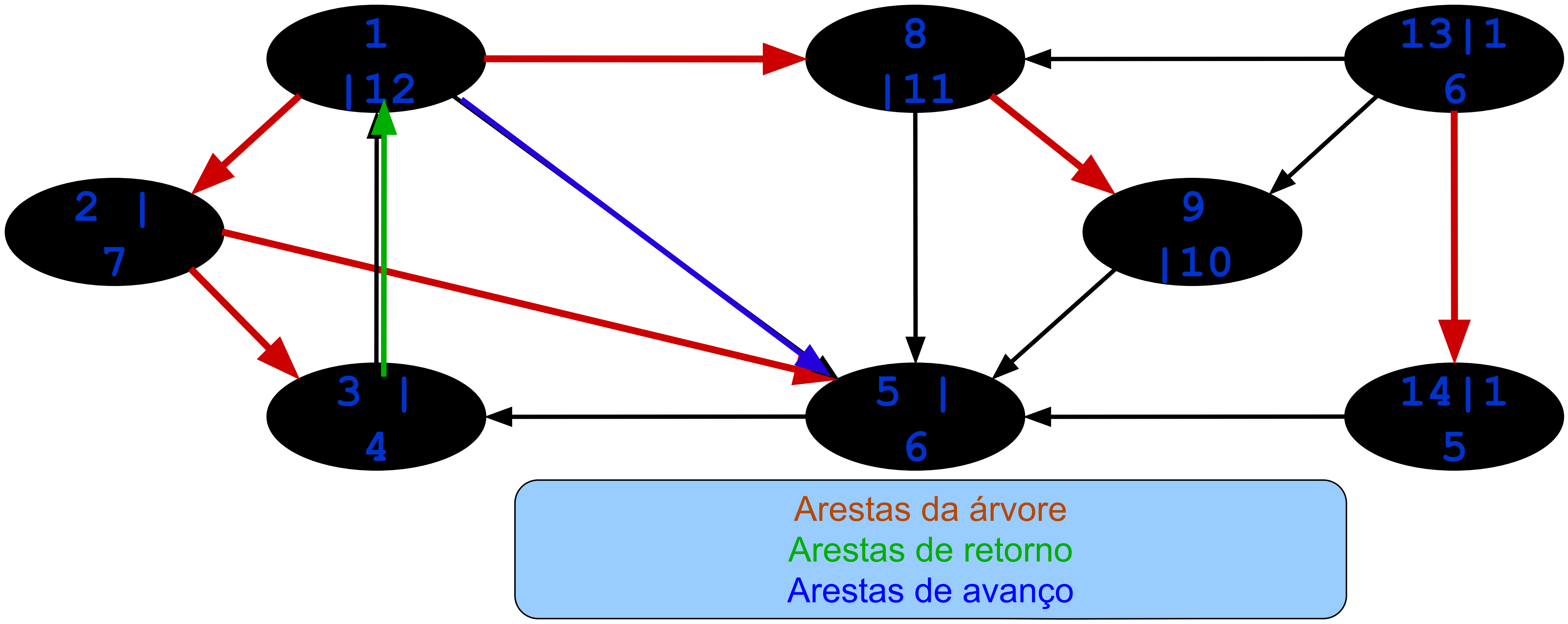


Arestas da árvore
Arestas de retorno

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

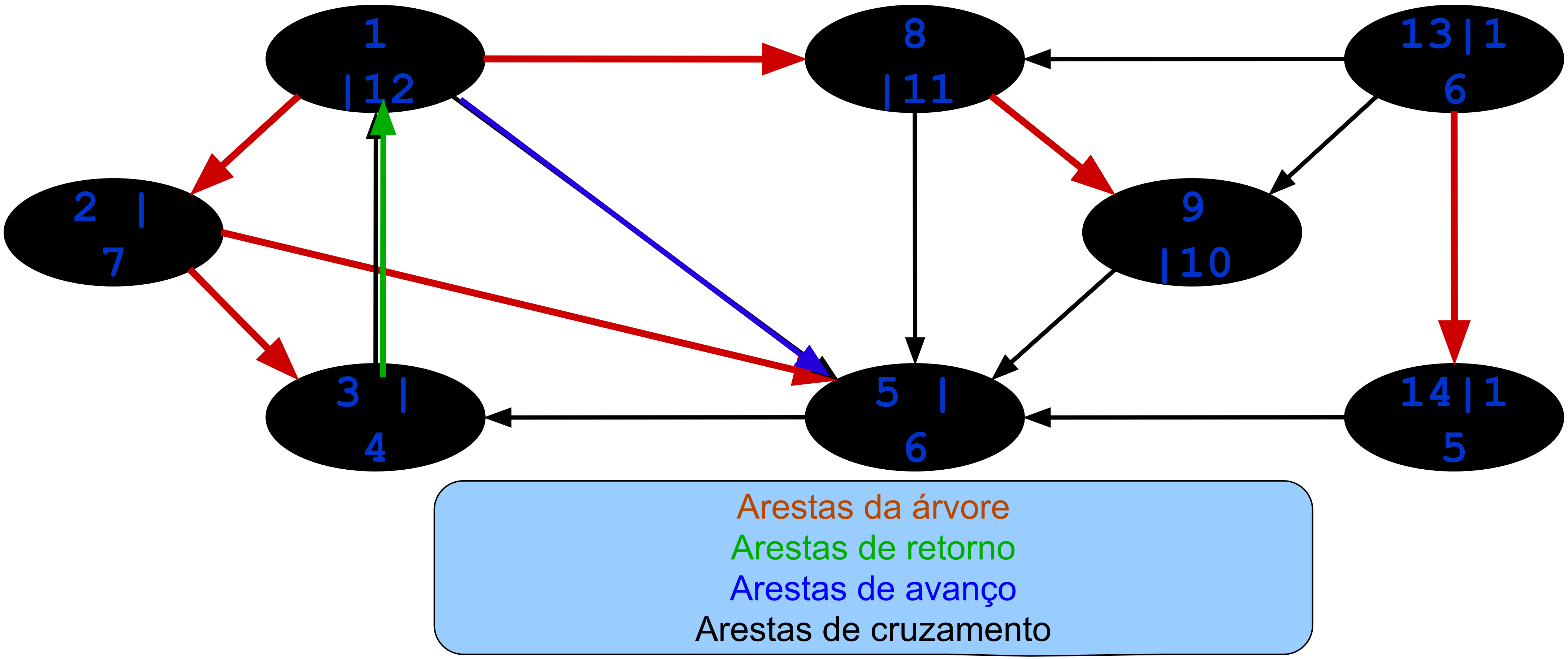
Busca em profundidade – Classificação das arestas



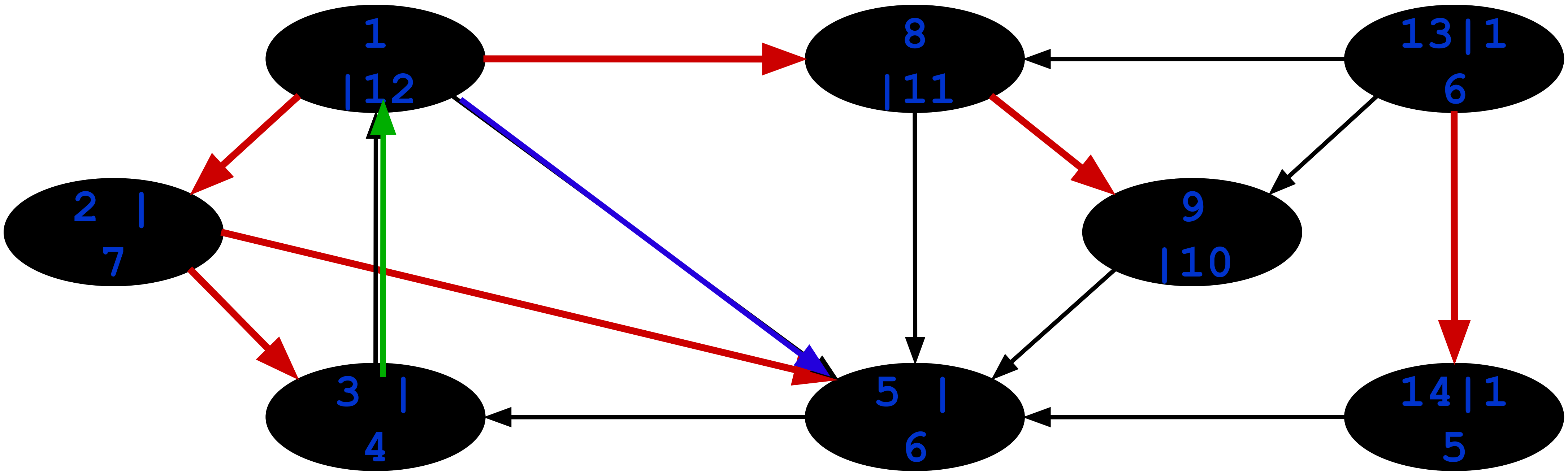
Busca em profundidade – Classificação das arestas

- 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



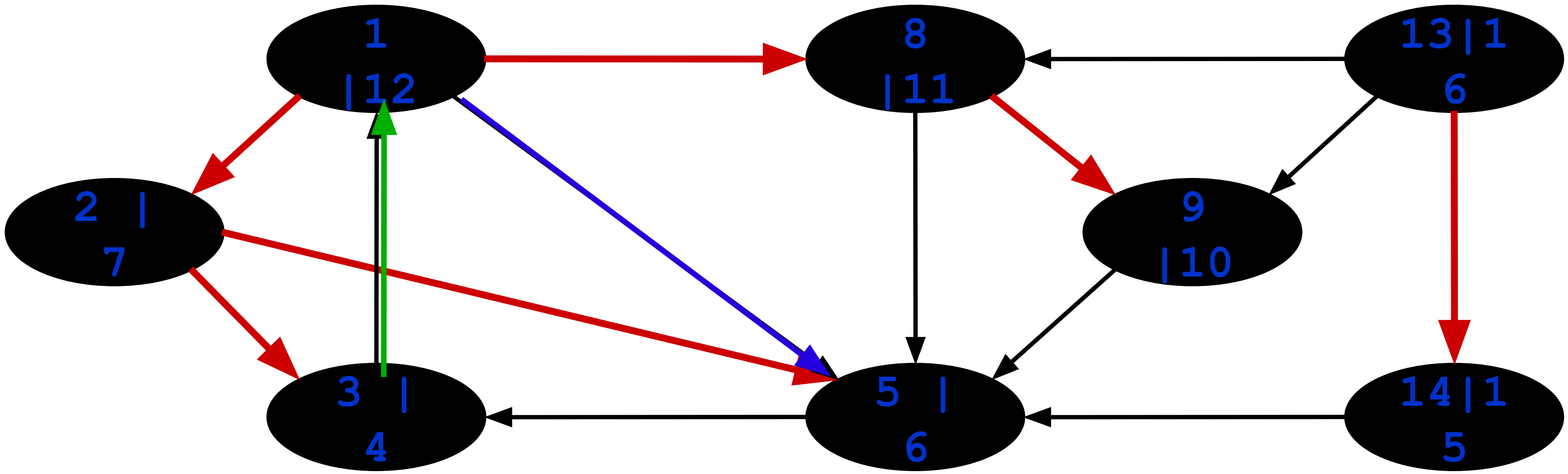
Busca em profundidade – Classificação das arestas



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

Podem as arestas de uma árvore formar ciclos?

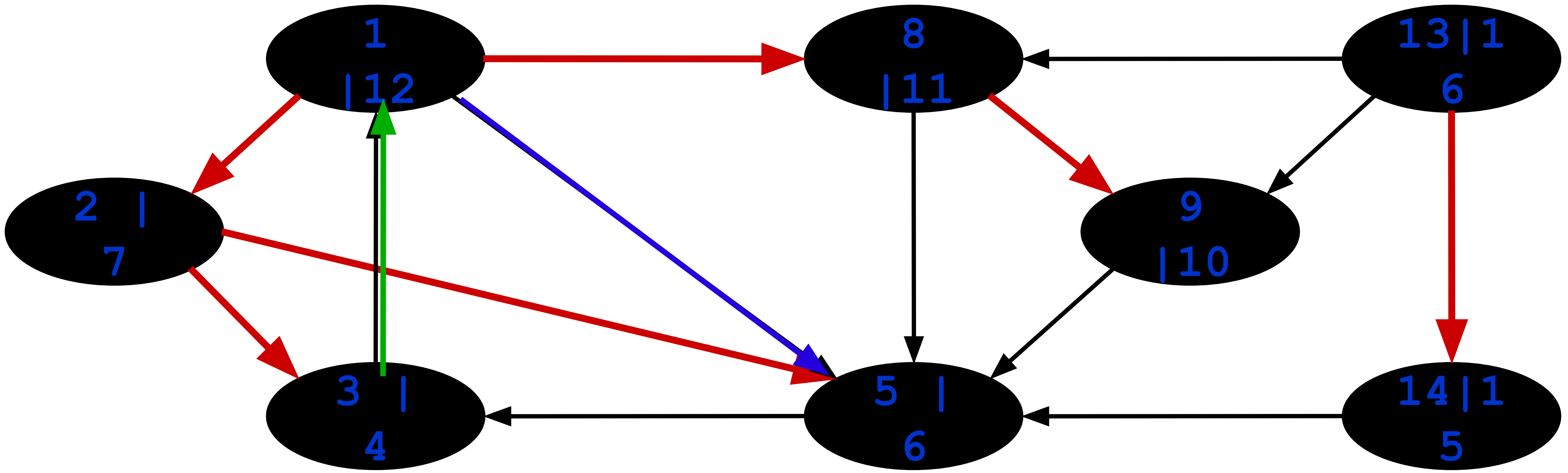
Busca em profundidade – Classificação das arestas



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

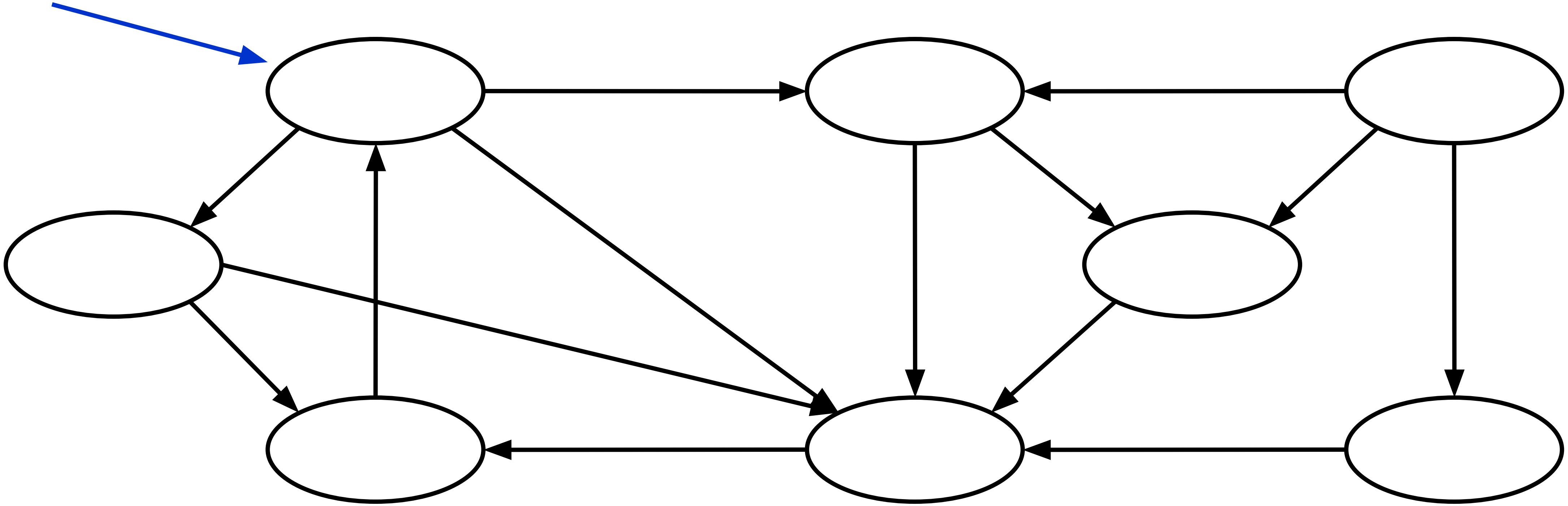
O que significa que existam aresta de retorno?

Busca em profundidade – Classificação das arestas

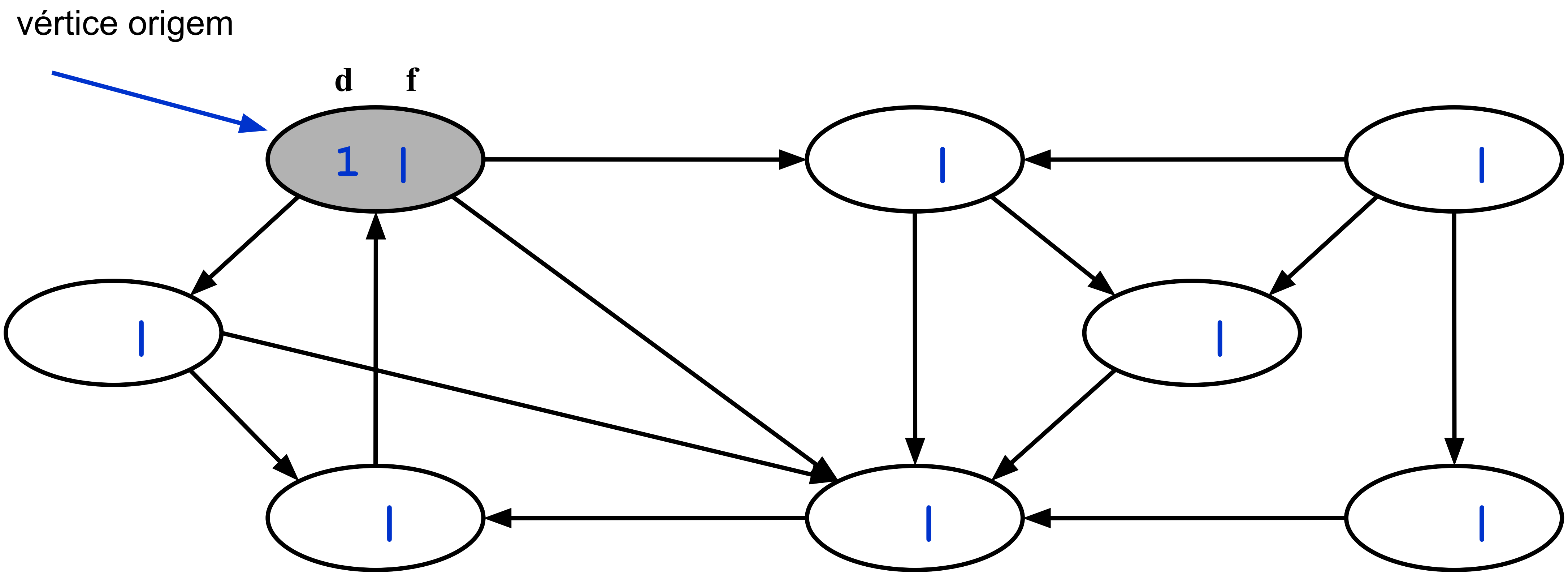
- O algoritmo DFS pode ser modificado para classificar arestas à medida que as encontra.
- Cada aresta (u,v) pode ser classificada pela cor do vértice v que é alcançado quando a aresta é explorada. Sabemos que:
  - Se v é **branco**, (u,v) é uma aresta da árvore

Busca em profundidade (DFS depth-first-search)

vértice origem

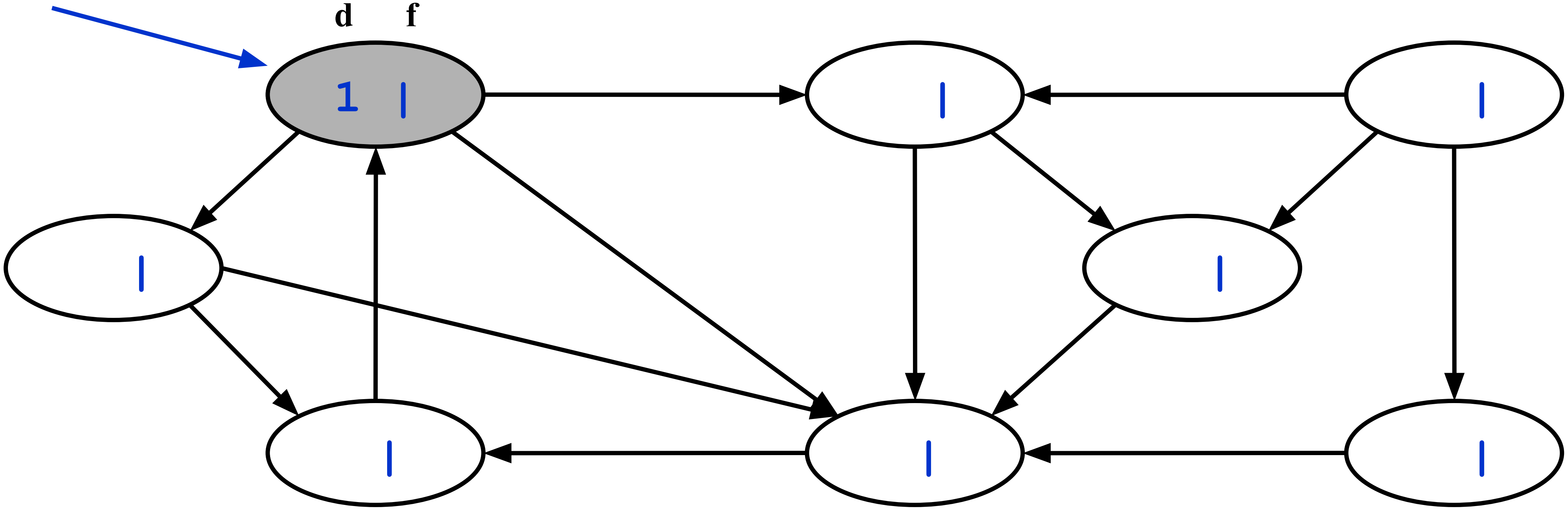


Busca em profundidade (DFS depth-first-search)



Busca em profundidade (DFS depth-first-search)

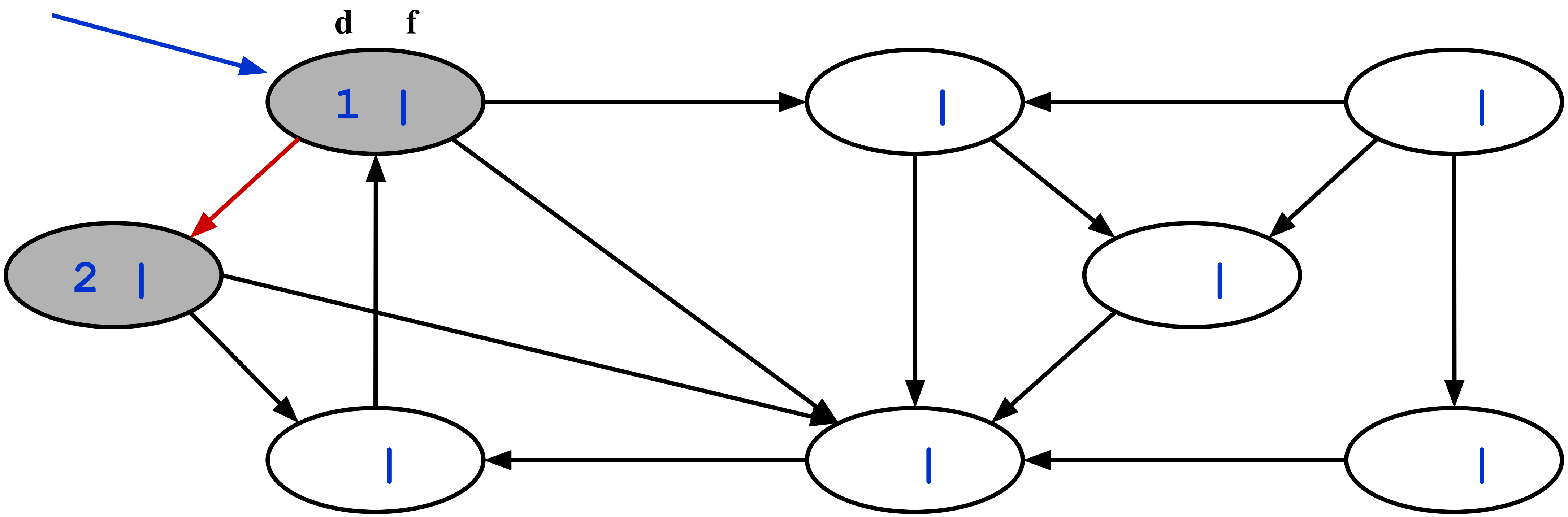
vértice origem



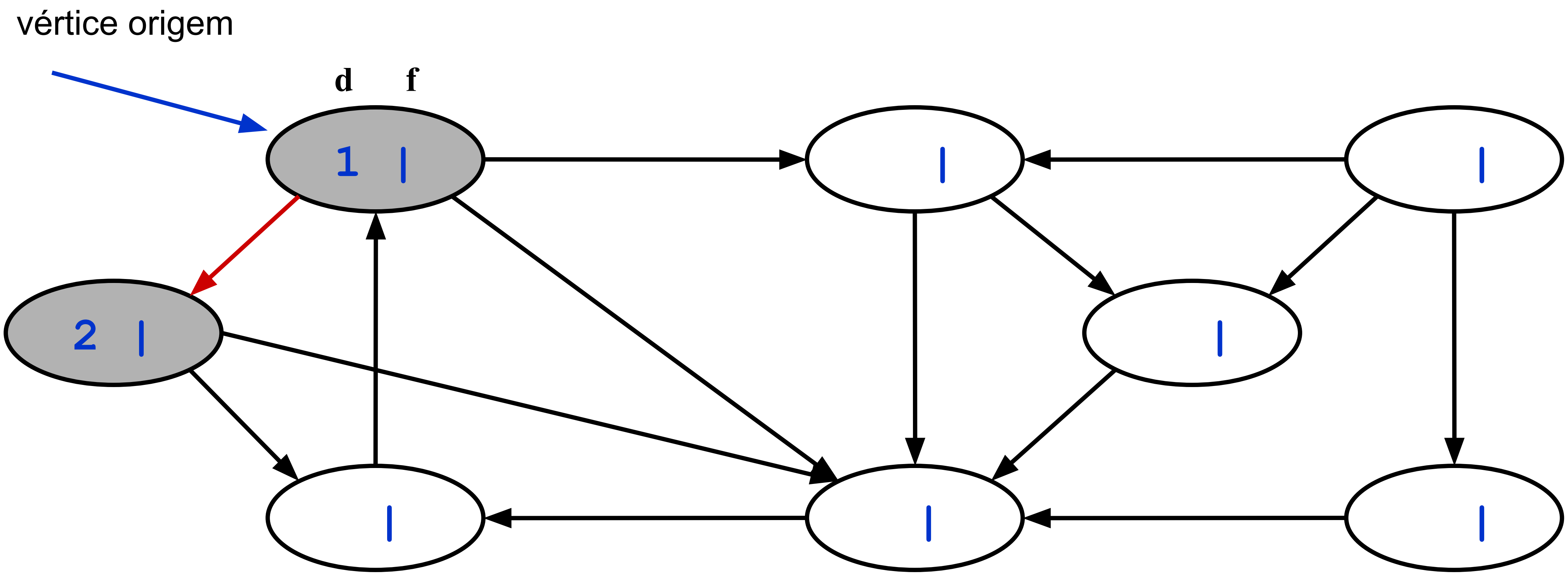
Existe algum vértice adjacente ao vértice que não tenha sido descoberto?

Busca em profundidade (DFS depth-first-search)

vértice origem

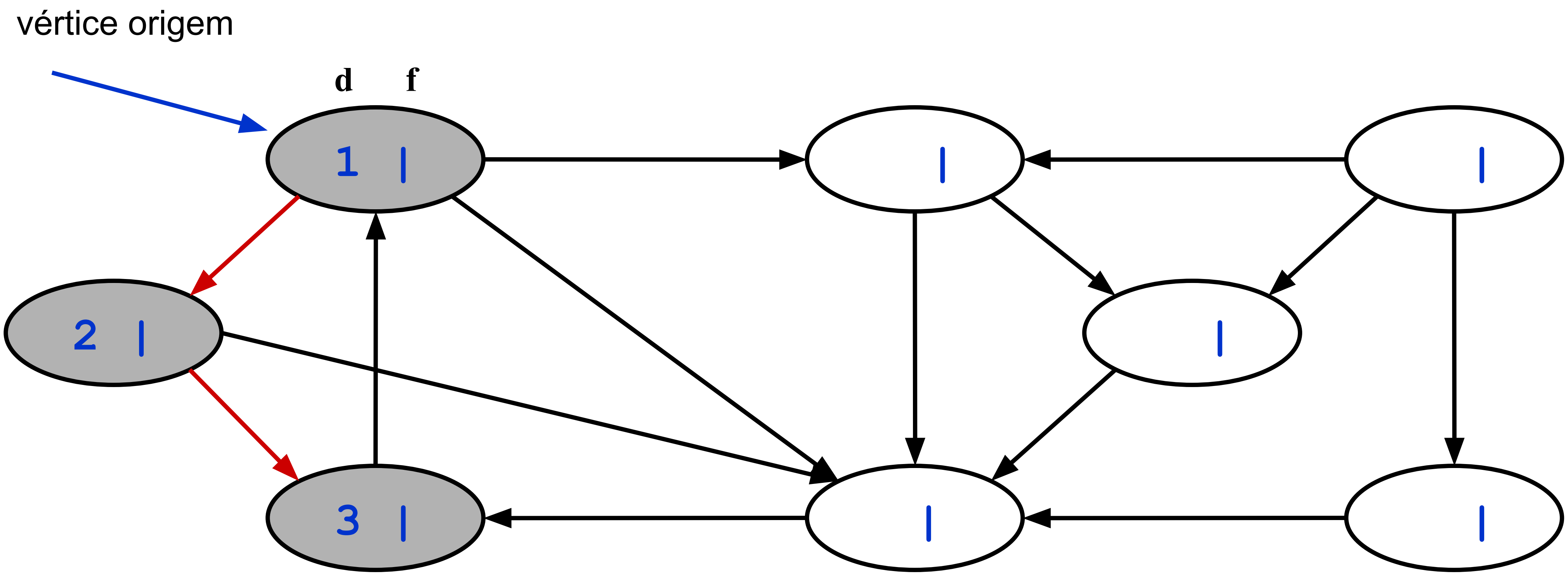


Busca em profundidade (DFS depth-first-search)

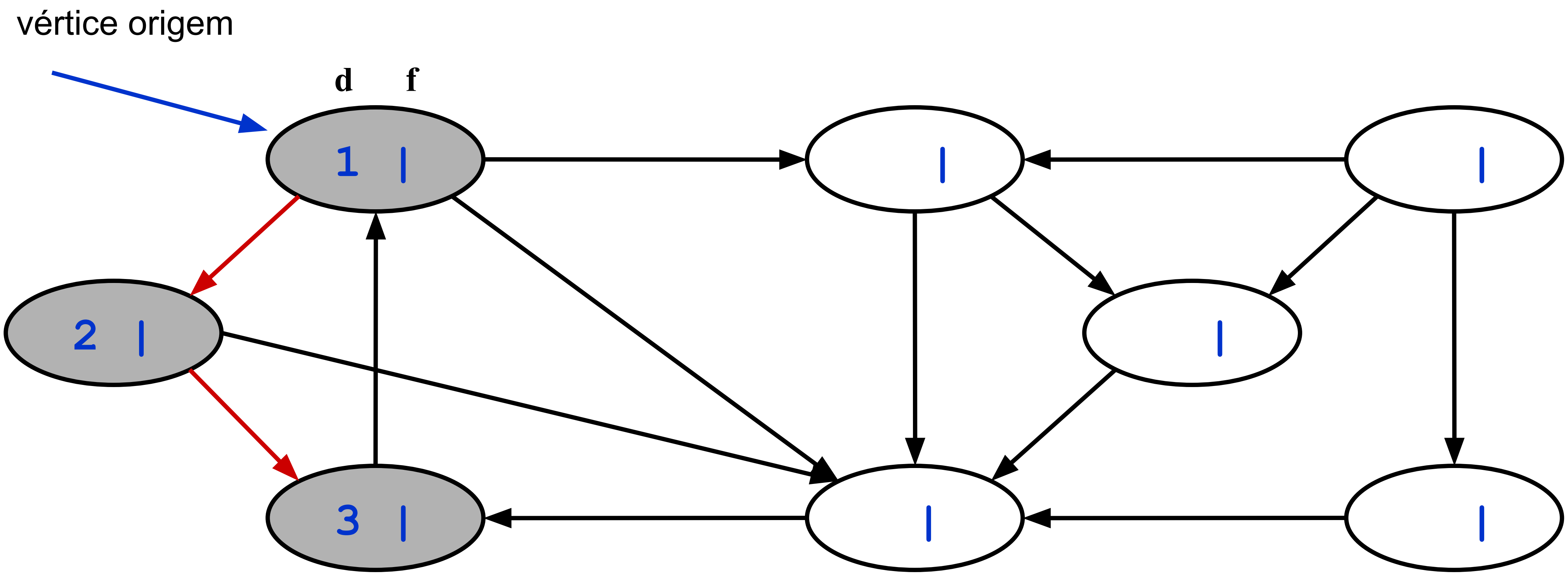


Existe algum vértice adjacente ao vértice que não tenha sido descoberto?

Busca em profundidade (DFS depth-first-search)

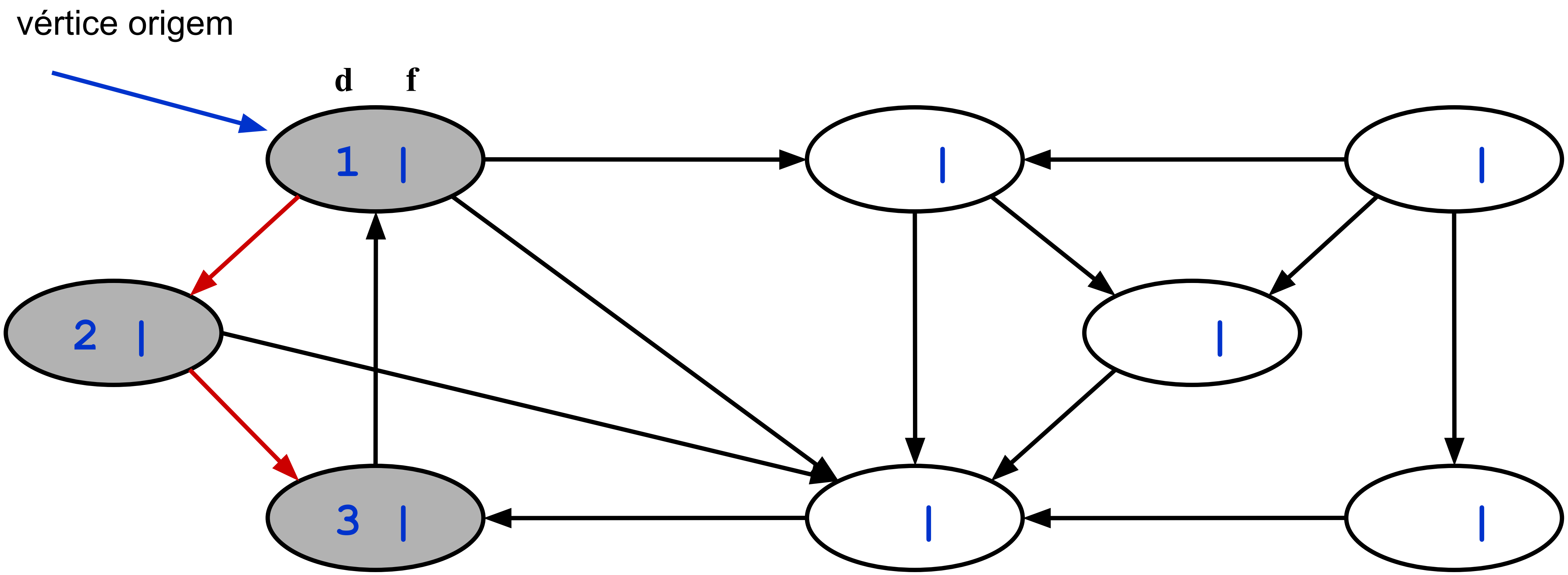


Busca em profundidade (DFS depth-first-search)



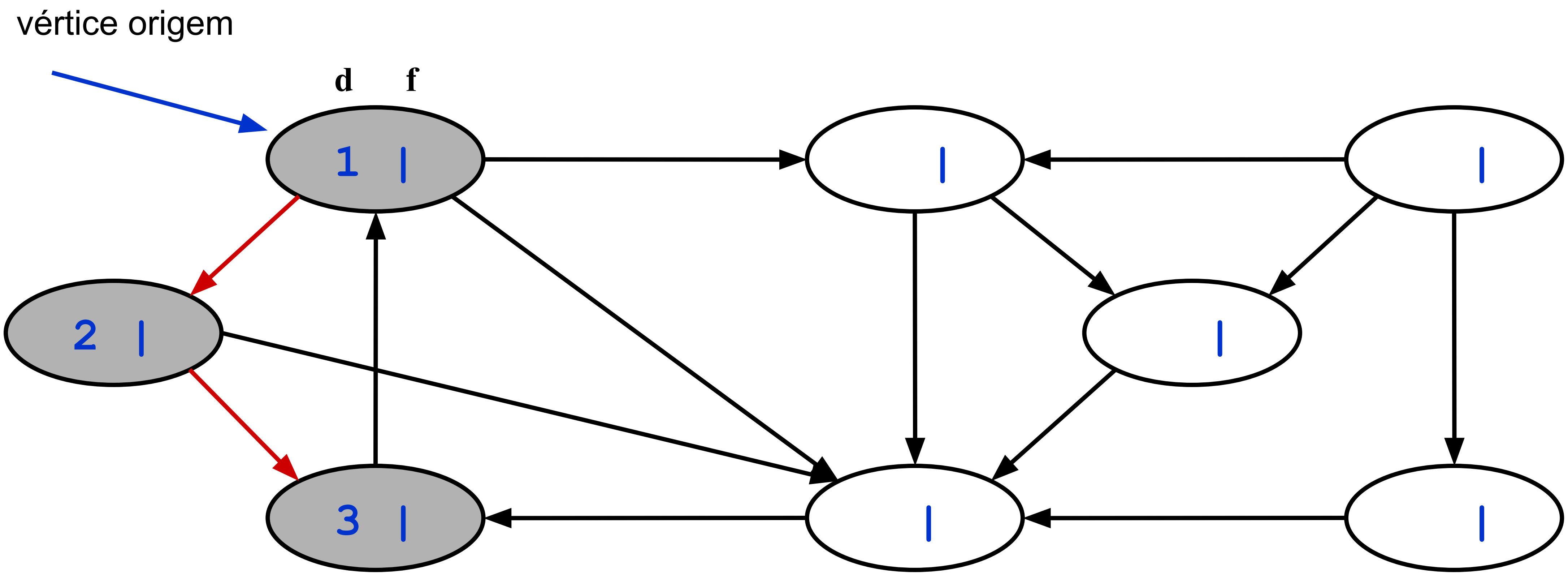
Existe algum vértice adjacente ao vértice?
Sim, mas é cinza

Busca em profundidade (DFS depth-first-search)



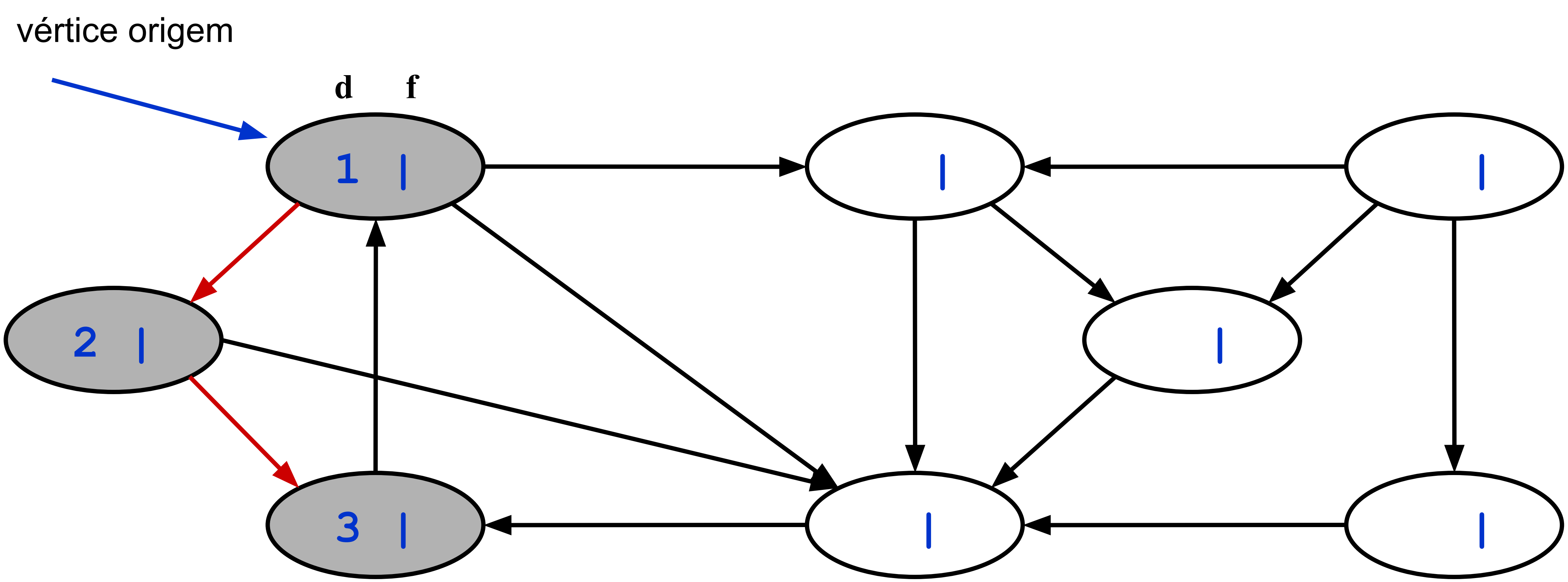
O que significa isso?

Busca em profundidade (DFS depth-first-search)



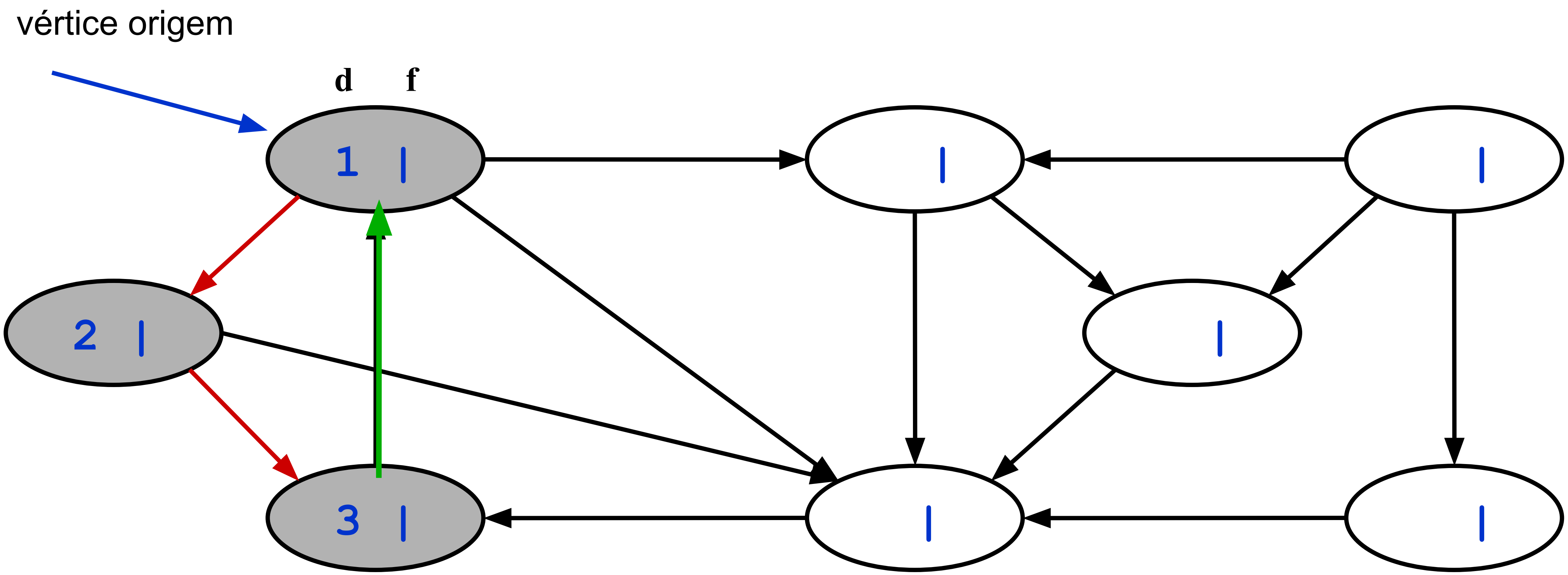
Observe que os vértices cinzas sempre formam uma cadeia linear de descendentes

Busca em profundidade (DFS depth-first-search)



Assim, uma aresta que alcança outro vértice cinza, alcança um ancestral

Busca em profundidade (DFS depth-first-search)



Cinza-cinza indica uma 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
  - 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.

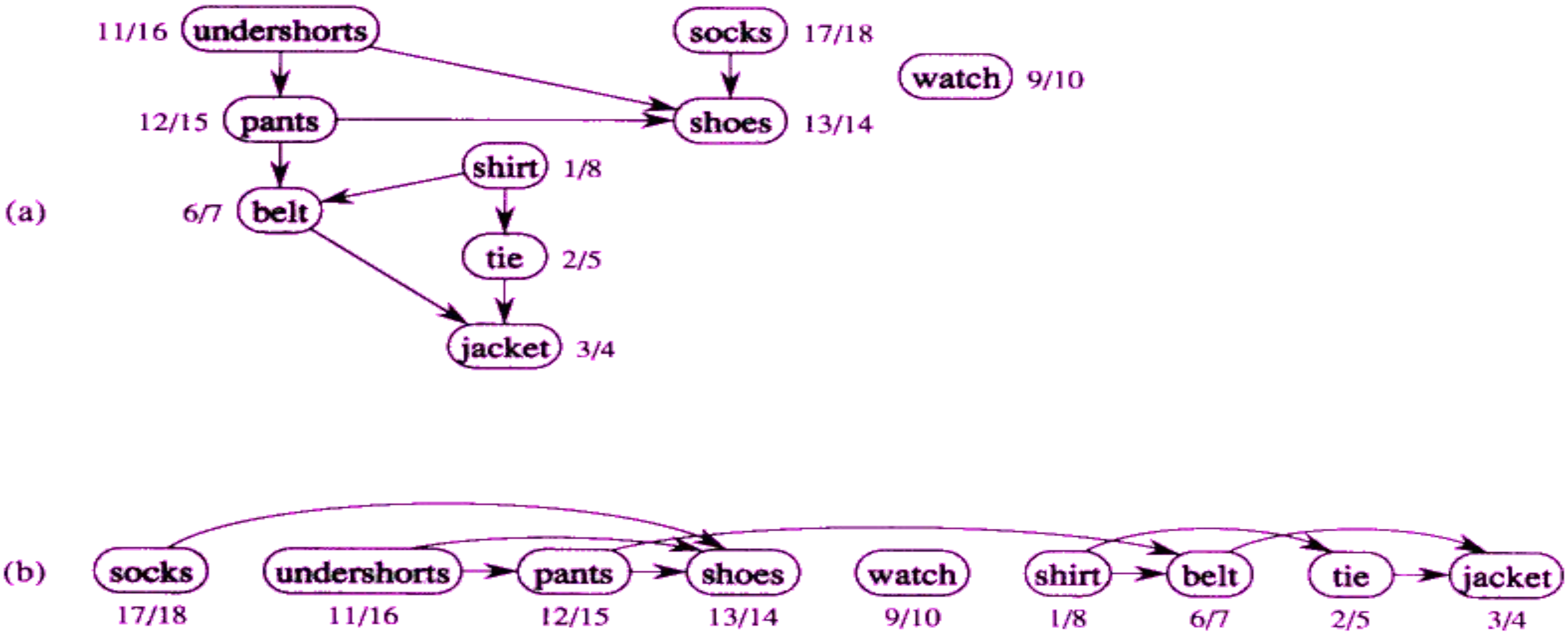
Problema: Determinar quantos componentes conectados tem um grafo não orientado.

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.

AULA 02

Algoritmos de busca em largura e profundidade em grafos

Karina Valdivia Delgado