

Estrutura de Dados – 1º semestre de 2021

Professor Mestre Fabio Pereira da Silva

Grafos

- 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

Transporte aéreo

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



Páginas Web

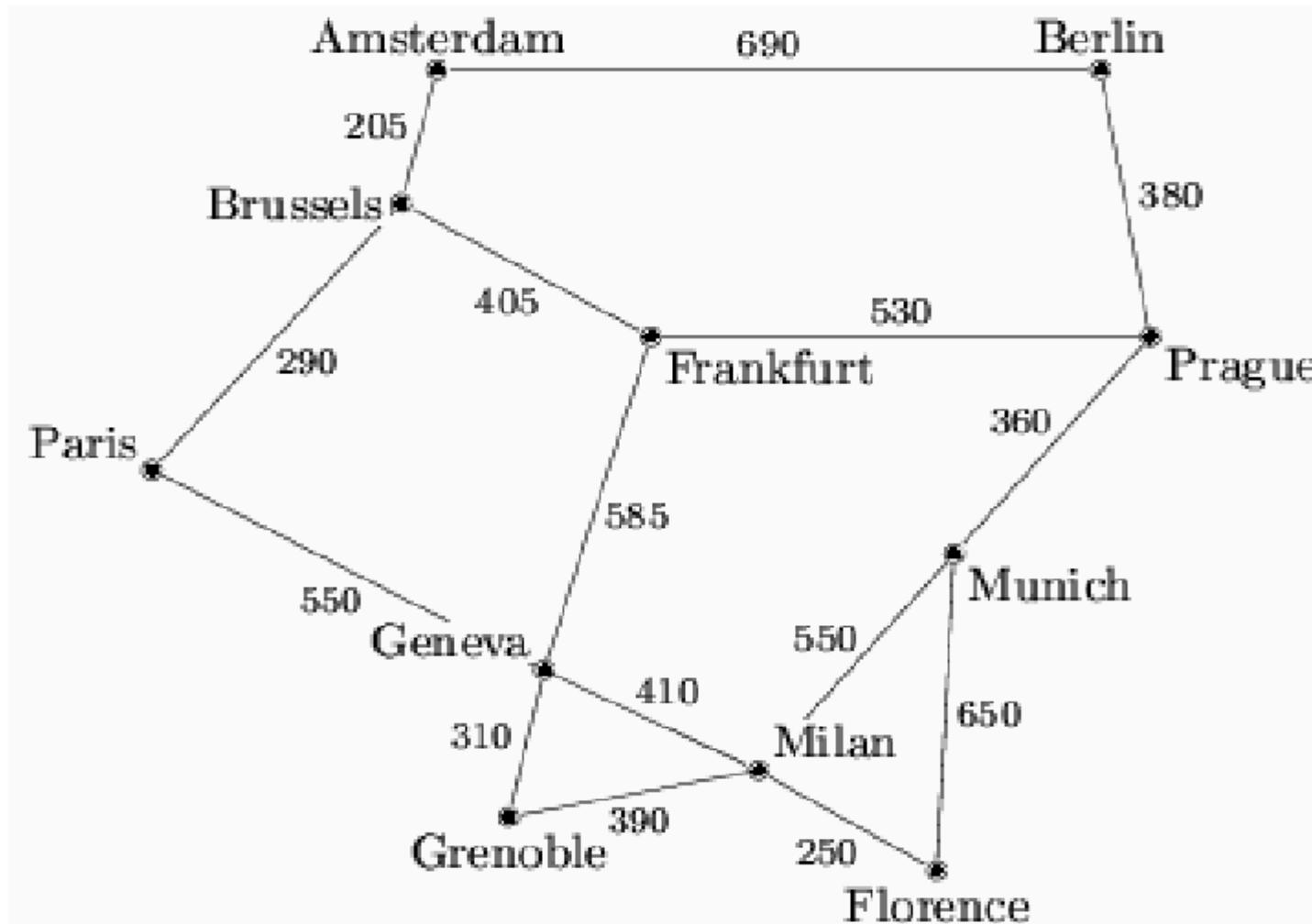
- objeto: páginas web
- relacionamento: link de uma página para outra



Aplicação

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

Viagem entre duas cidades



Cidades Europeias



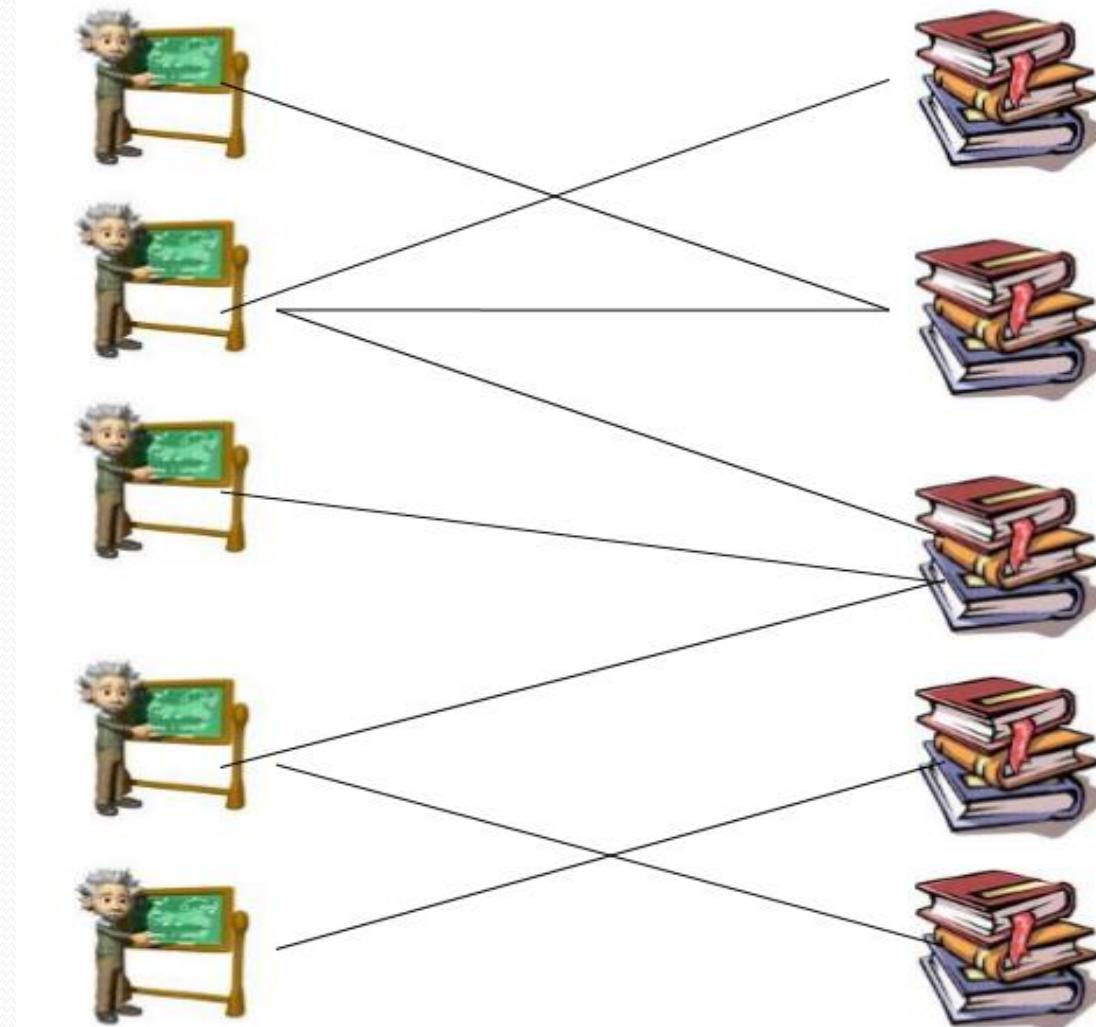
Mapa de estradas



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?

Alocação de professores

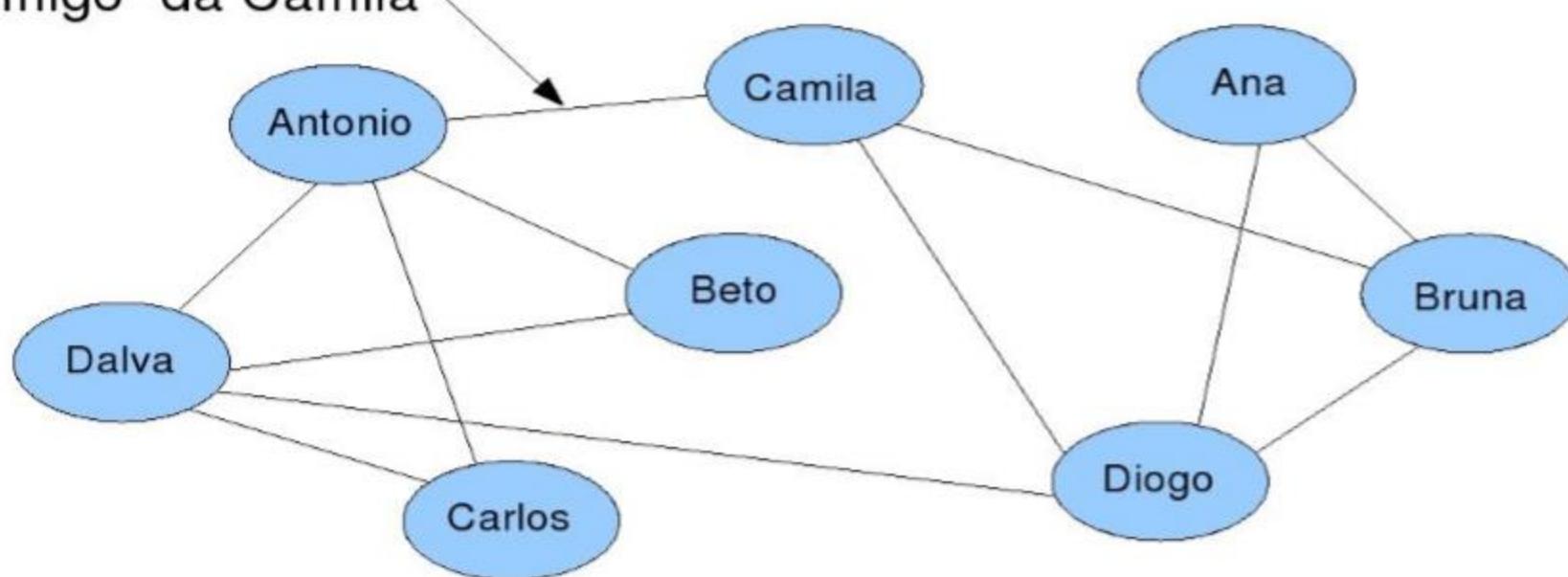


Pesquisa de contatos no Facebook

- 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?

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

Antonio é
“amigo” da Camila

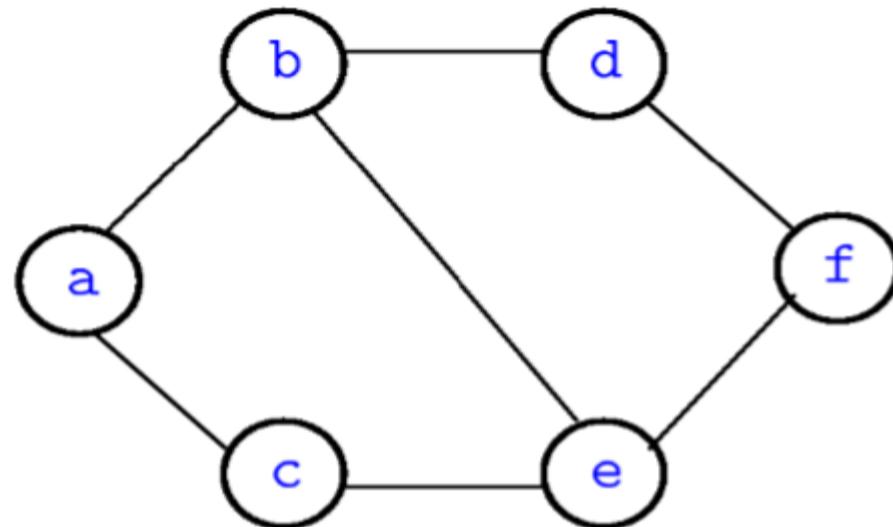


Modelos de utilização de Grafos

Grafo	Vértice	Aresta
Comunicação	Centrais telefônicas, Computadores, Satélites	Cabos, Fibra óptica, Enlaces de microondas
Circuitos	Portas lógicas, registradores, processadores	Filamentos
Hidráulico	Reservatórios, estações de bombeamento	Tubulações
Financeiro	Ações, moeda	Transações
Transporte	Cidades, Aeroportos	Rodovias, Vias aéreas
Escalonamento	Tarefas	Restrições de precedência
Arquitetura funcional de um software	Módulos	Interações entre os módulos
Internet	Páginas Web	<i>Links</i>
Jogos de tabuleiro	Posições no tabuleiro	Movimentos permitidos
Relações sociais	Pessoas, Atores	Amizades, Trabalho conjunto em filmes

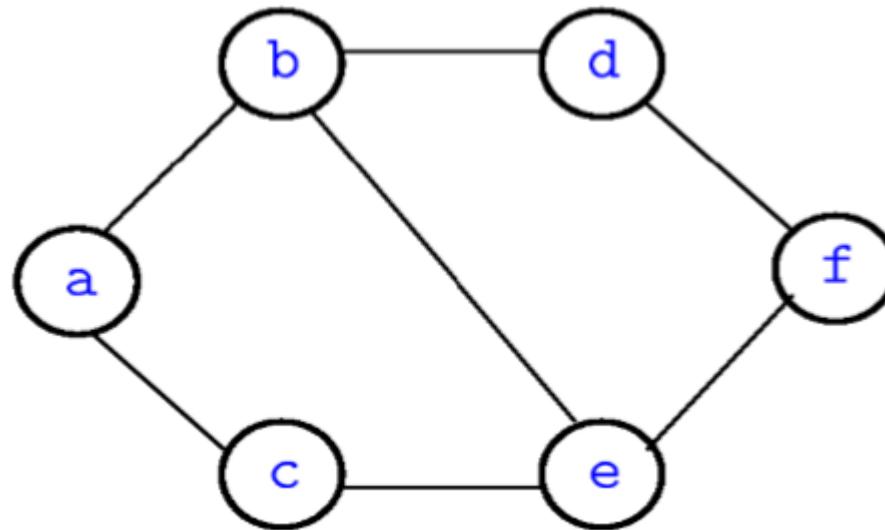
Conceitos Básicos

- Grafo: Um conjunto de vértices e arestas
- Vértice: Objeto simples que pode ter nome e outros atributos.
- Aresta: Conexão entre dois vértices.
- Representação:



Conceitos Básicos

- Notação: $G = (V, A)$.
- Em que:
- G : Grafo
- V : conjunto de vértices
- A : conjunto de arestas

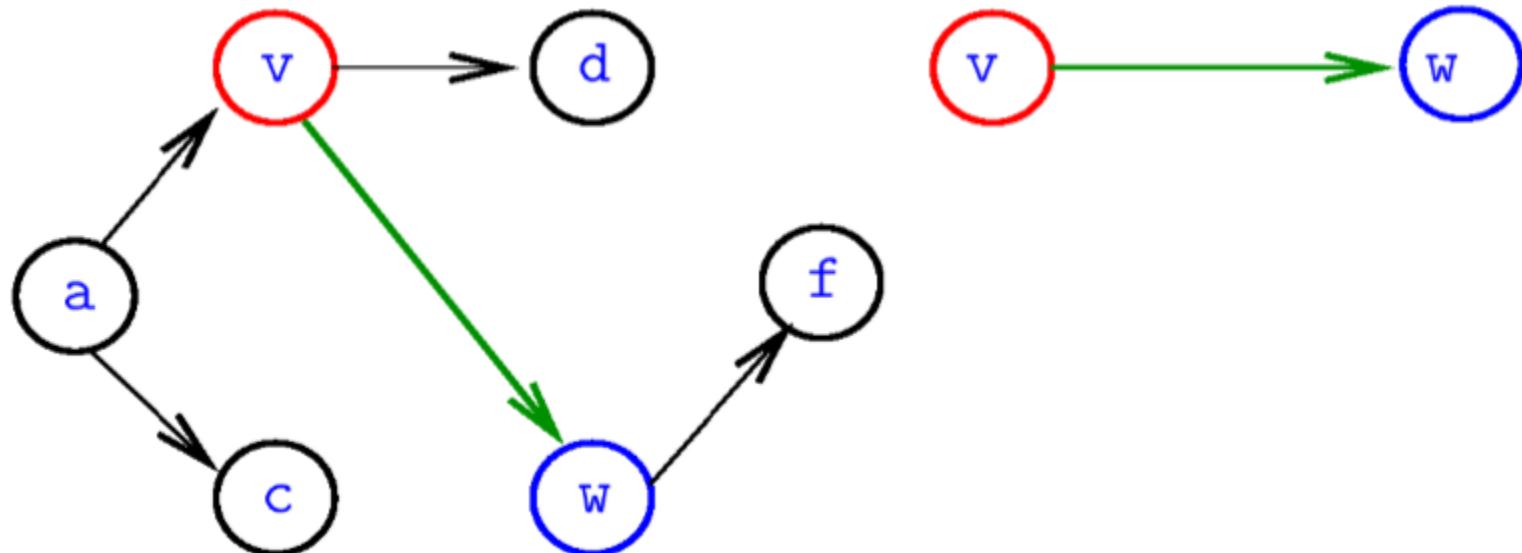


Conceitos Básicos

- **Extremidade** de uma aresta: vértice da aresta.
- **Arestas paralelas**: arestas associadas ao mesmo conjunto de vértices.
- **Dois vértices que são conectados por uma aresta são chamados de adjacentes.**
- Um vértice que é nó terminal de um laço é dito ser **adjacente a si próprio**.
- Duas arestas incidentes ao mesmo vértice são chamadas de **adjacentes**.
- Um vértice que não possui nenhuma aresta incidente é chamado de **isolado**.
- Um grafo com nenhum vértice é chamado de **vazio**.

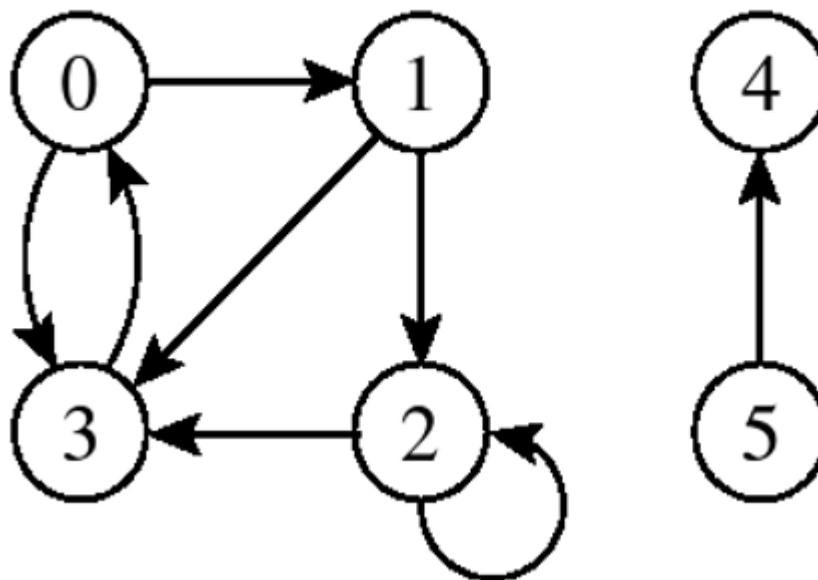
Grafos Direcionados

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



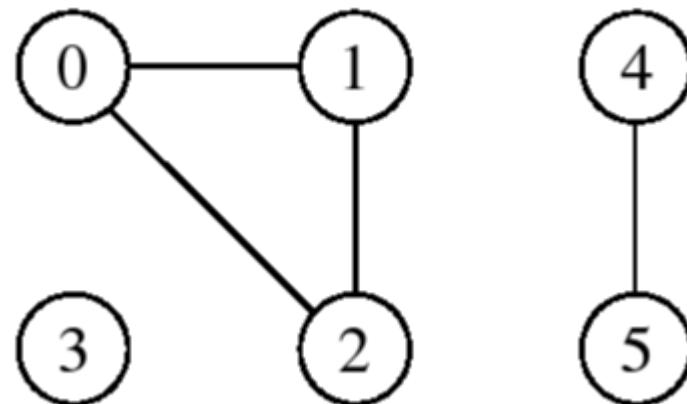
Grafos Direcionados

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



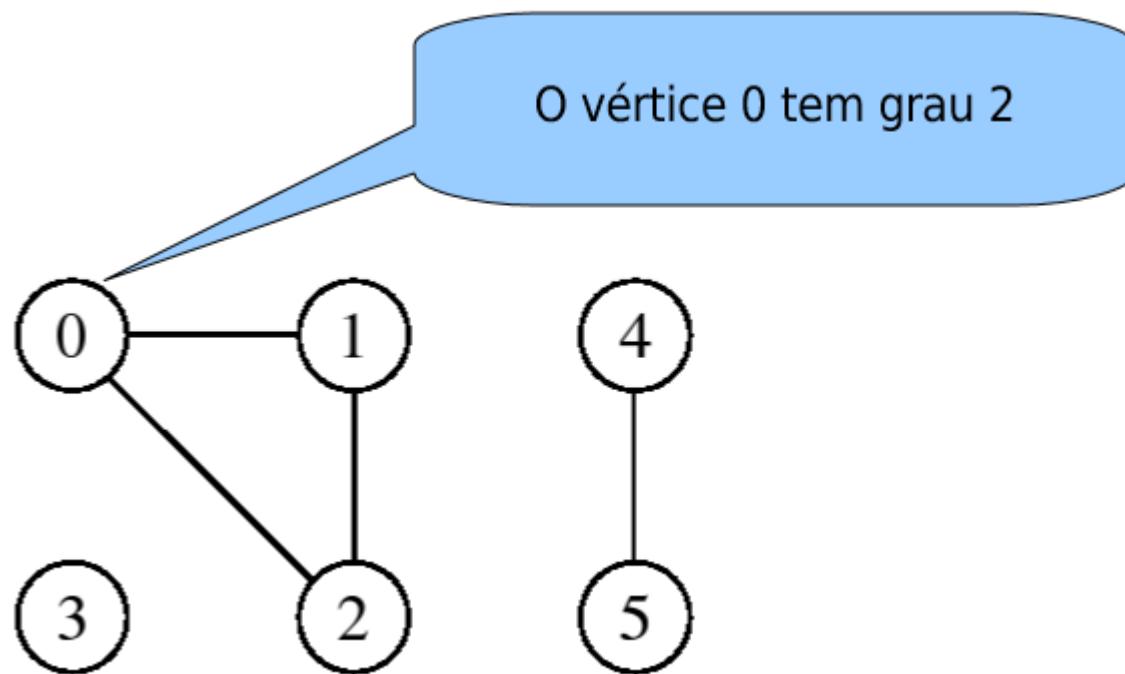
Grafos não Direcionados

- 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.



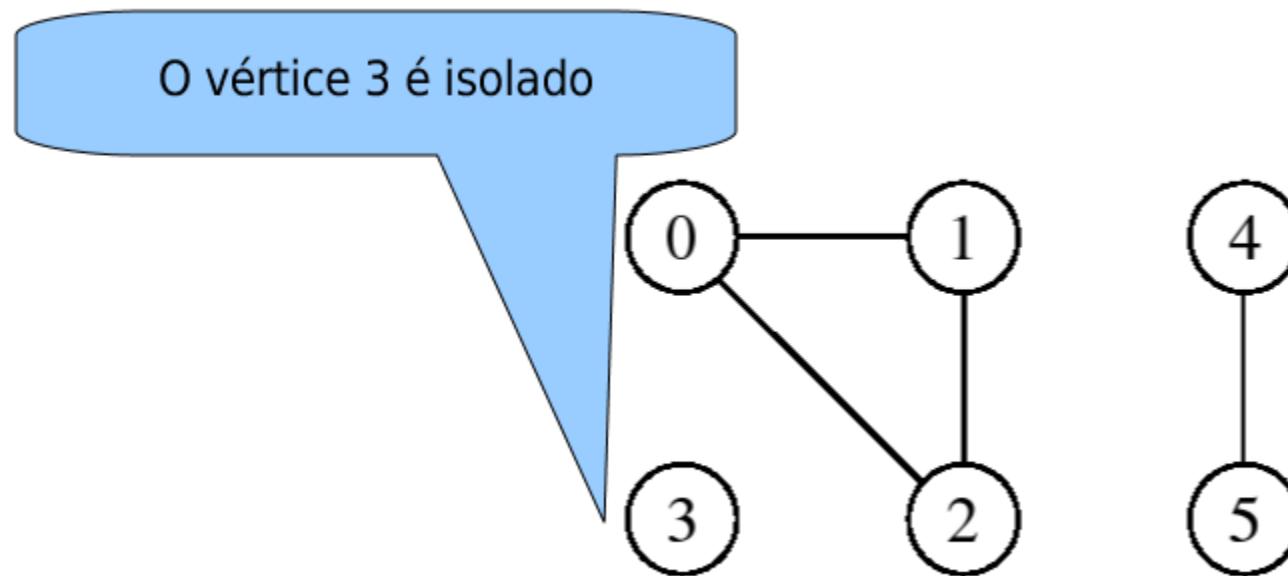
Grau de um vértice em um grafo não direcionado

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



Grau de um vértice em um grafo não direcionado

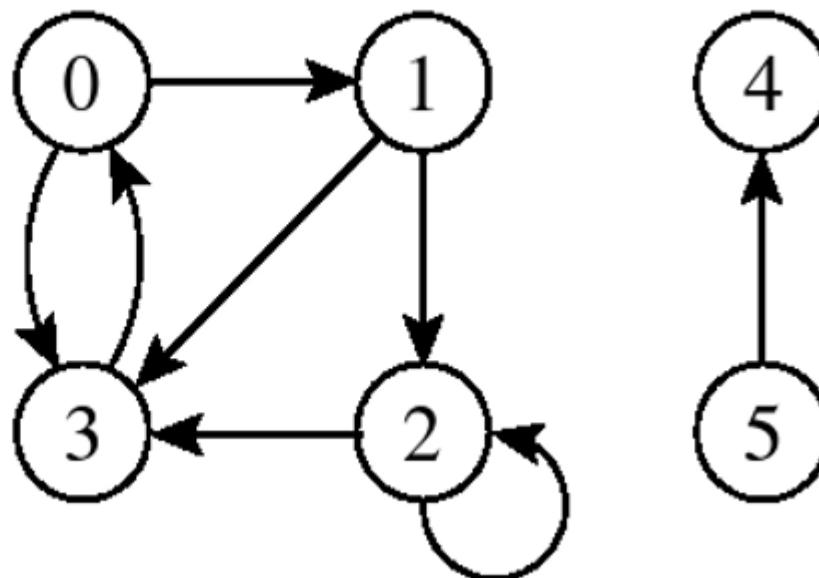
- O grau de um vértice é o número de arestas que incidem nele.
- Um vértice de grau zero é dito isolado ou não conectado.



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.

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

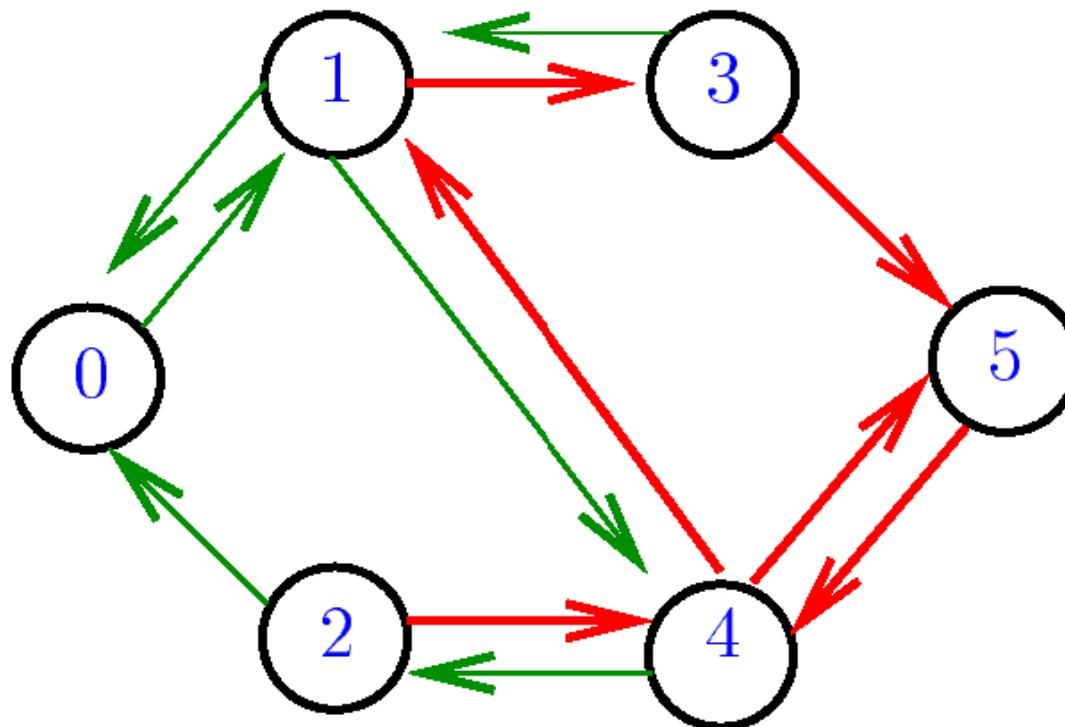


Caminho entre vértices

- Um caminho de comprimento k de um vértice x a um vértice y em um grafo $G = (V, A)$ é uma sequência de vértices $(v_0, v_1, v_2, \dots, v_k)$ tal que $x = v_0$ e $y = v_k$, e $(v_{i-1}, v_i) \in A$ para $i = 1, 2, \dots, k$.
- **O comprimento de um caminho é o número de arestas nele**

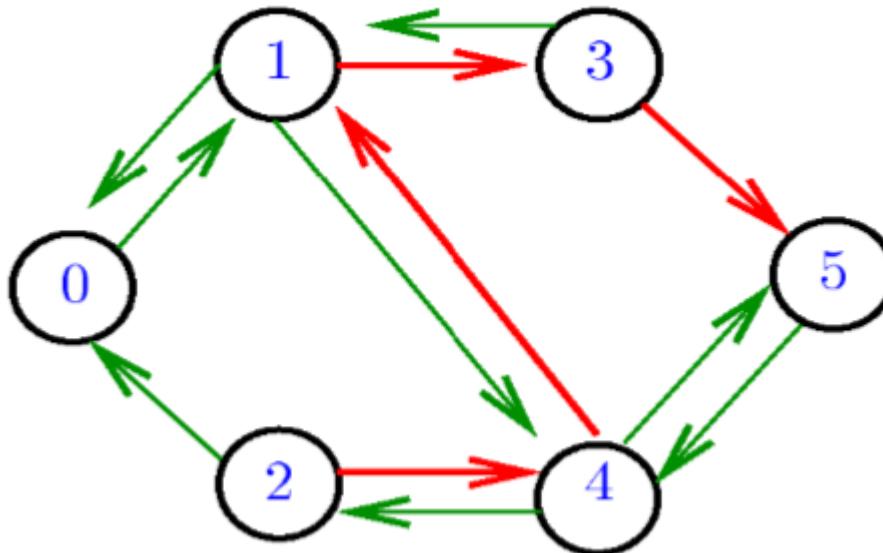
Caminho entre vértices

- $(2,4,1,3,5,4,5)$ 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: $(2,4,1,3,5)$

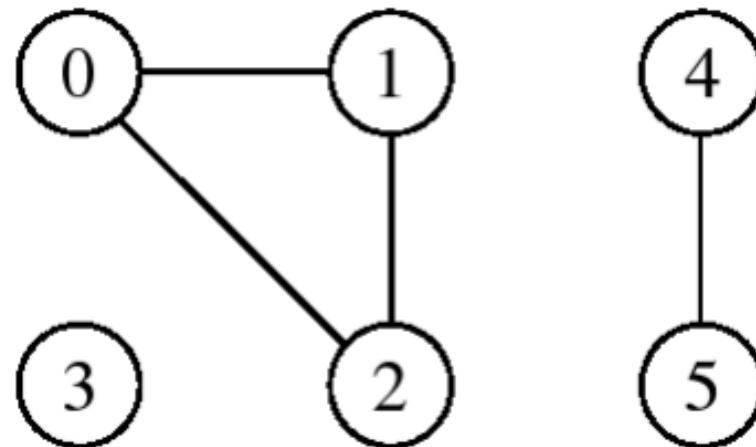


Componentes conectados

- 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.

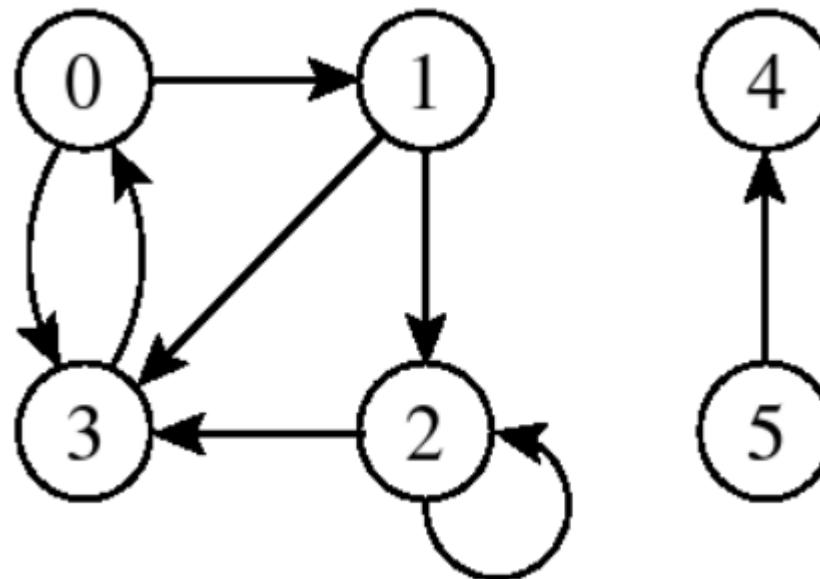


Componentes fortemente conectados

- 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.

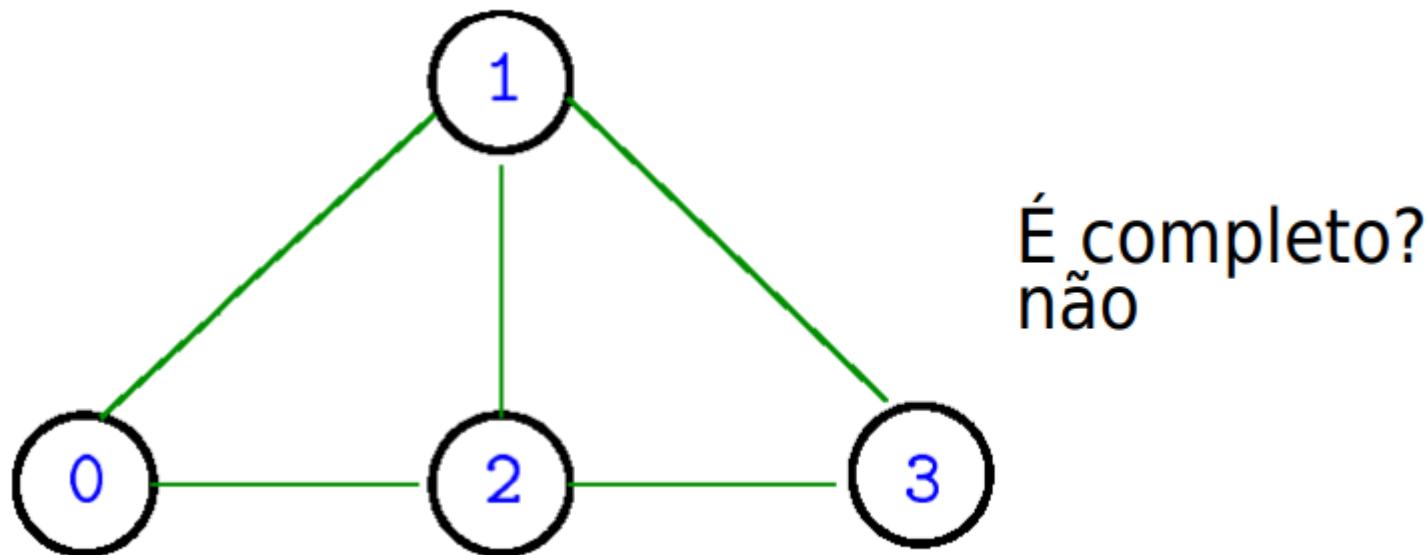
Componentes fortemente conectados

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



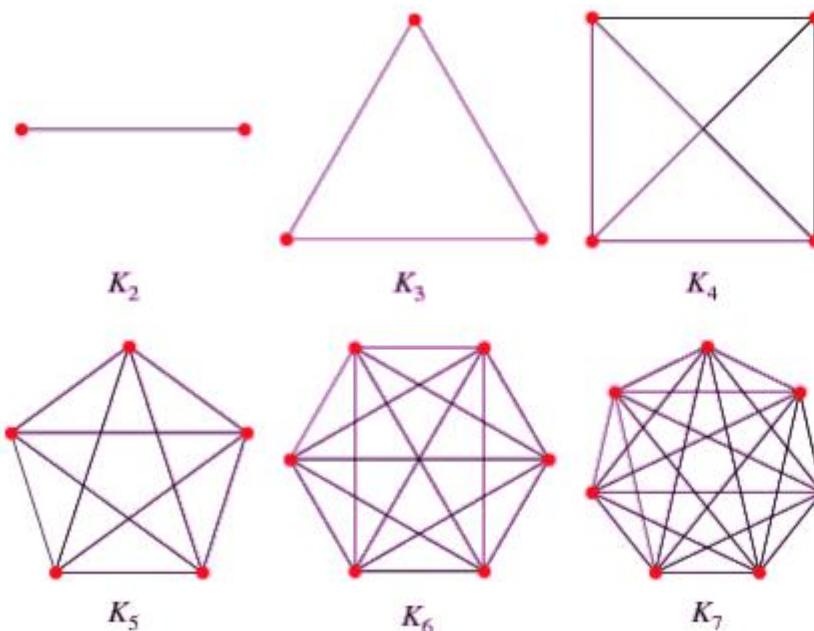
Grafo Completo

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



Grafo Completo

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



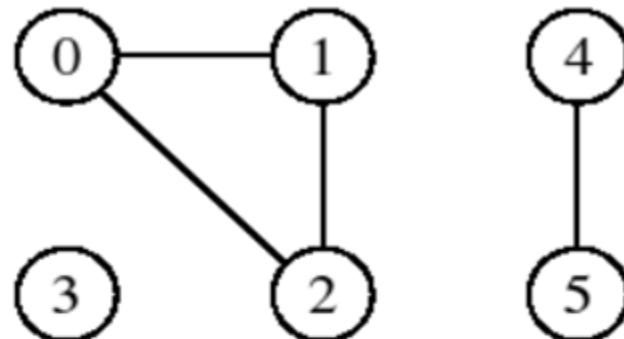
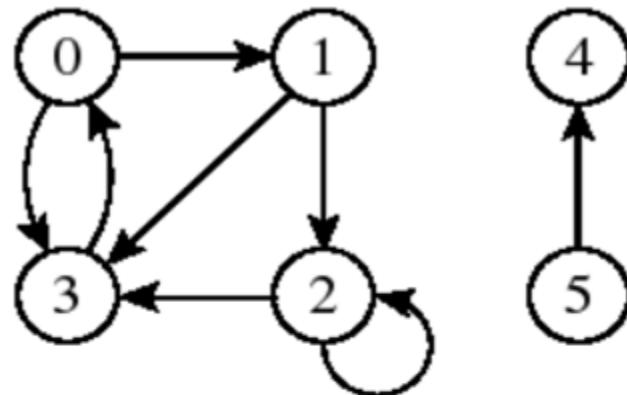
Operações

- Criar um grafo vazio
- Inserir uma aresta n grafo
- Verificar se existe determinada aresta no grafo
- **Obter a lista de vértices adjacentes a determinado vértice**
- Eliminar uma aresta do grafo
- Imprimir o 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 em Matrizes de Adjacências

- Associar vértices à linhas e colunas da matriz e o elemento da matriz indica se há aresta
- 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.
- Considere grafos grandes e esparços:
 - Grande: muitos vértices
 - Esparço: relativamente poucas arestas
- **Matriz formada principalmente de zeros! – Grande consumo de memória (desnecessário)!**

Representação de Grafos em Matrizes de Adjacências



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

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

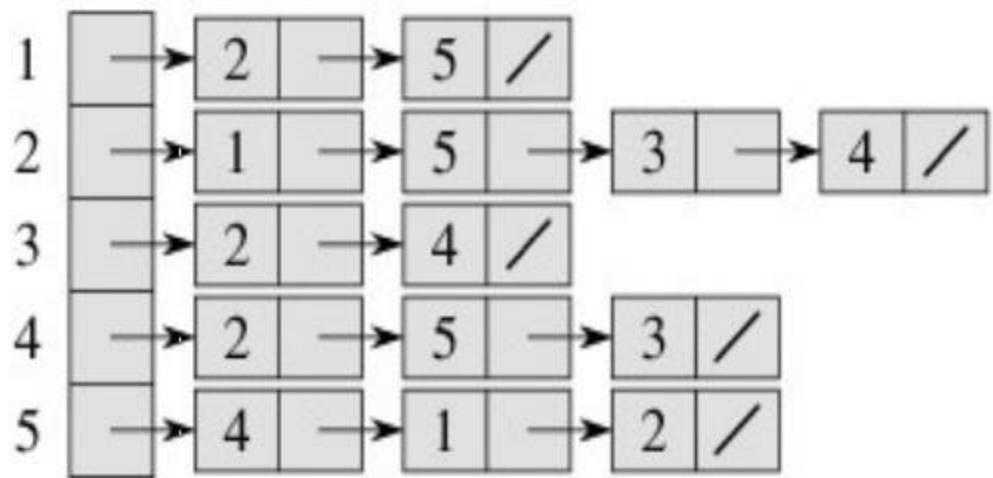
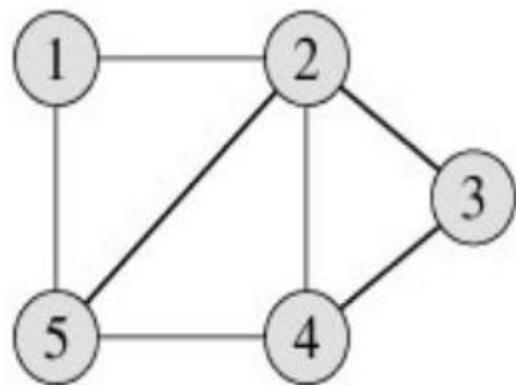
Representação de Grafos em Matrizes de Adjacências

- 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.

Representação de Grafos em Listas de Adjacências

- **Como representar grafos grandes e esparços?**
- **Associa a cada vértice uma lista de vértices adjacentes.**
- 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 $\text{Adj}[u]$ consiste de todos os vértices adjacentes a u , i.e., todos os vértices v tais que existe uma aresta $(u,v) \in A$

Representação de Grafos em Listas de Adjacências

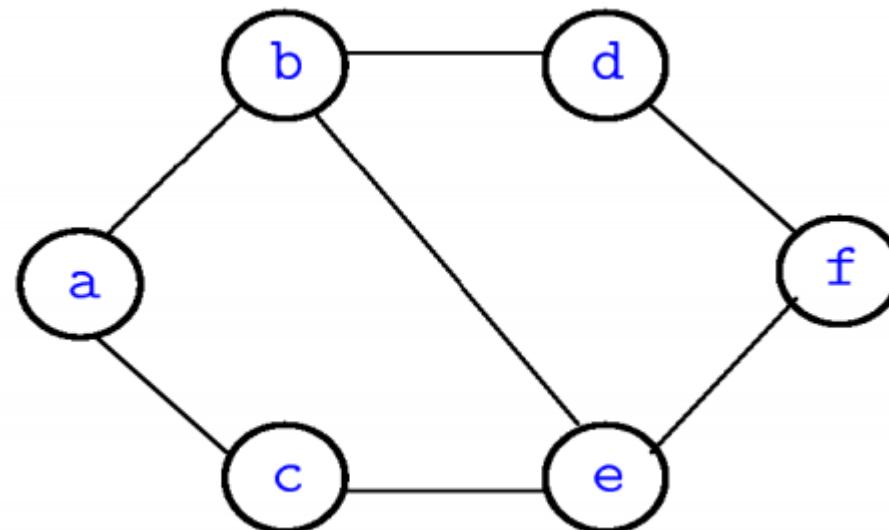


Busca em Grafos

- 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!**

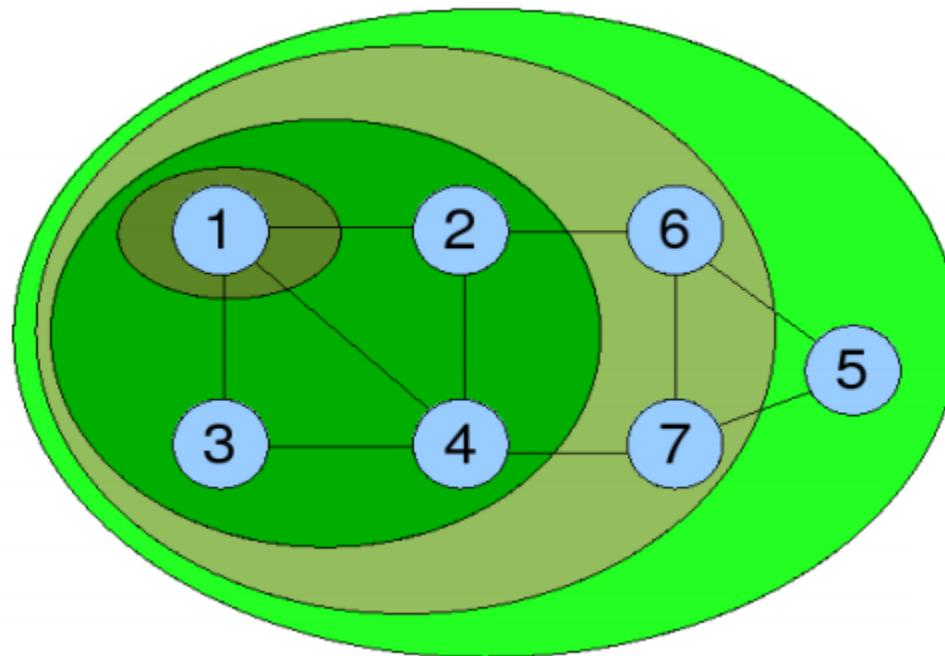


Busca em Largura

- 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

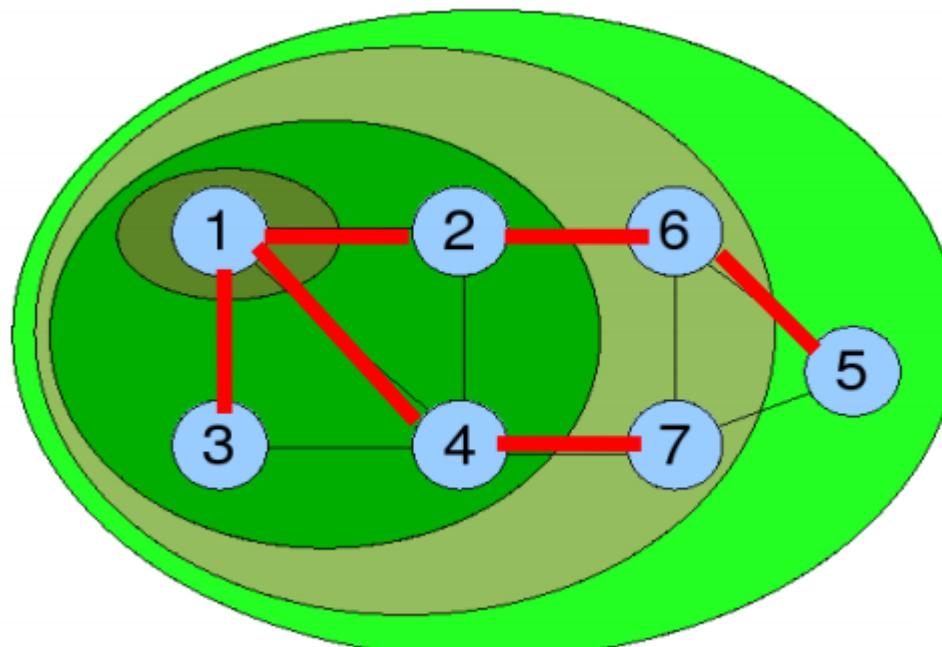
- Antes de encontrar um vértice à distância $k+1$ de s , todos os vértices à distância k são encontrados.



Intuição: uma onda é propagada a partir da raiz

Busca em Largura

- 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).**

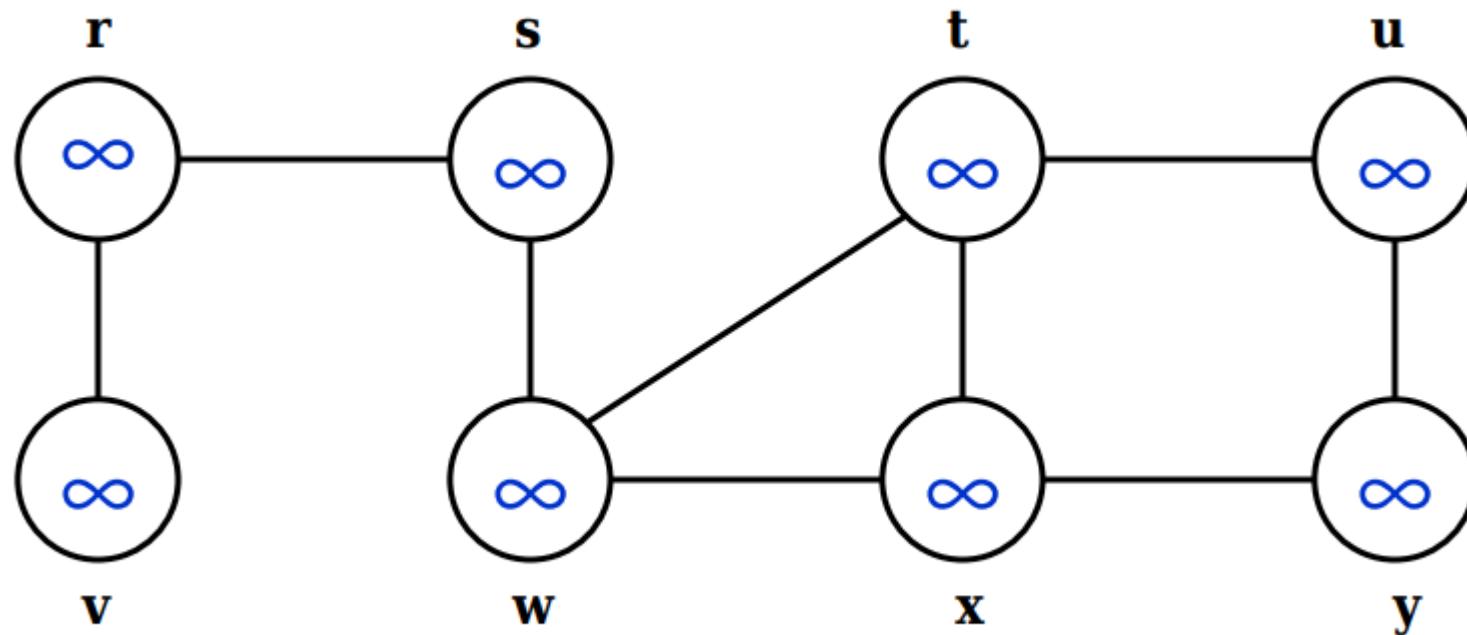


Busca em Largura

- 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

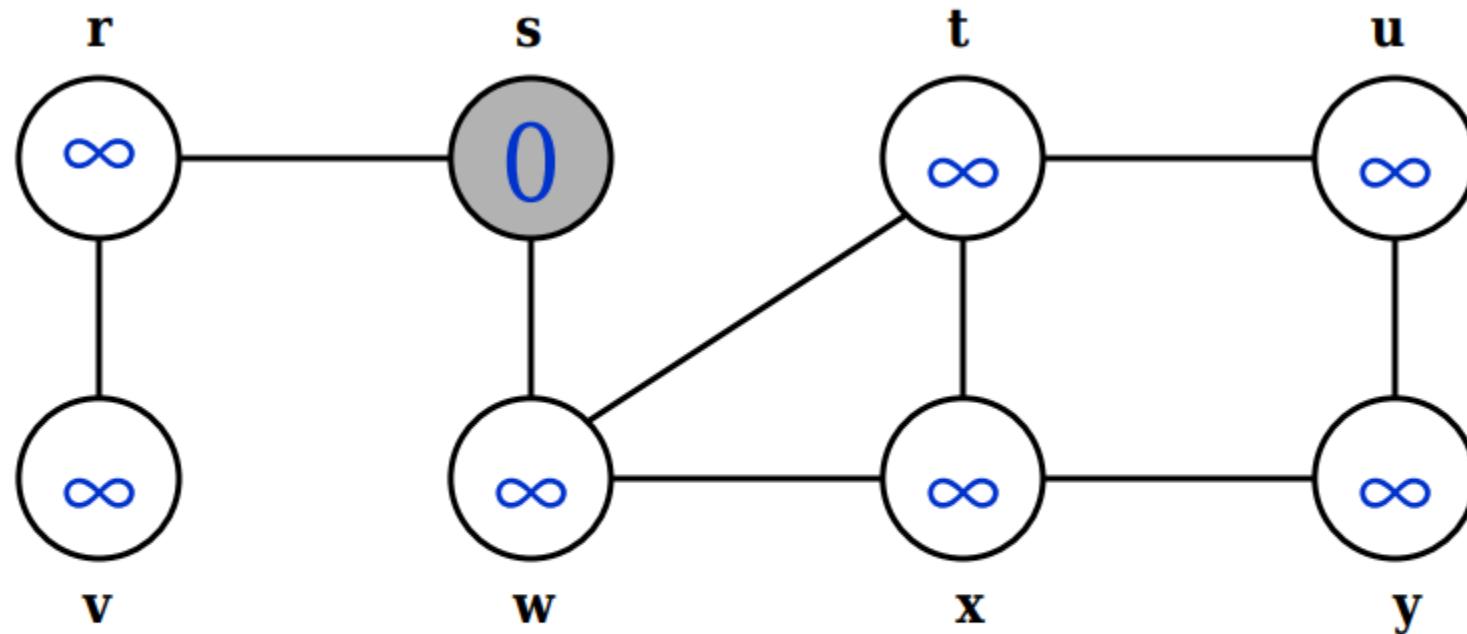
Busca em Largura

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



Busca em Largura

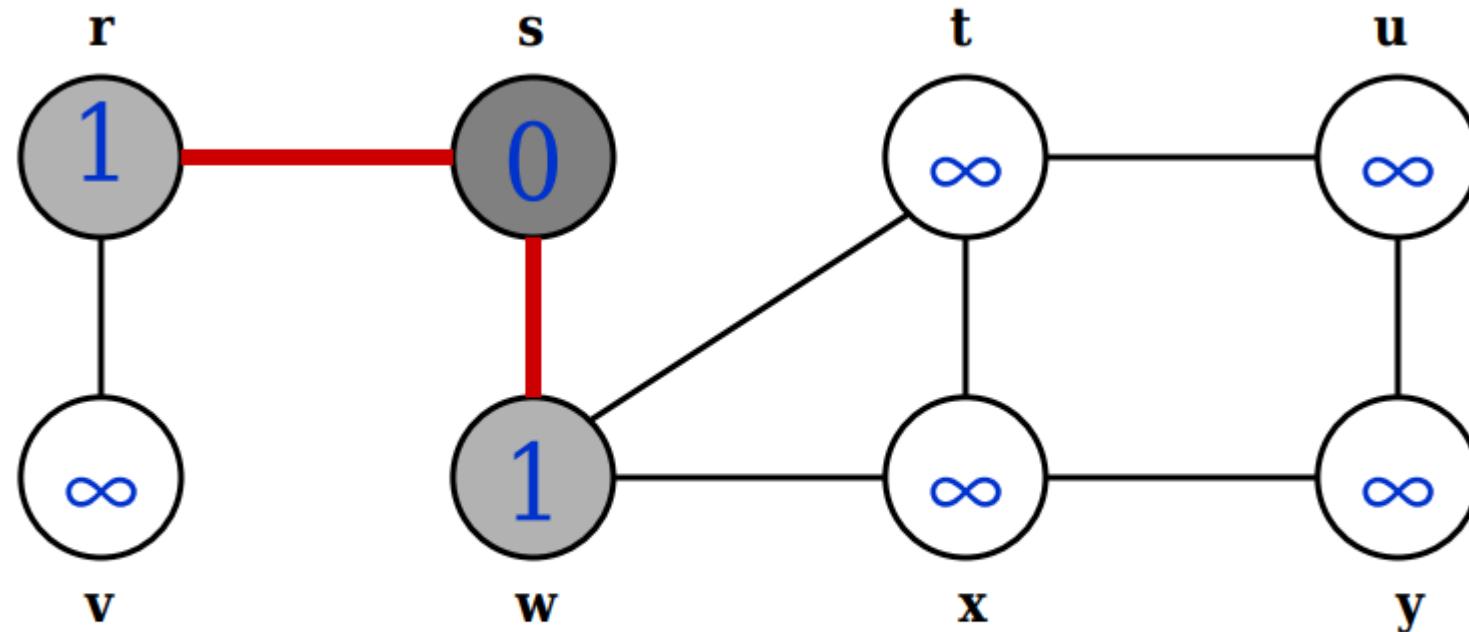
O vértice origem é pintado de cinza (ele é considerado descoberto) e é colocado na fila



Q: s

Busca em Largura

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

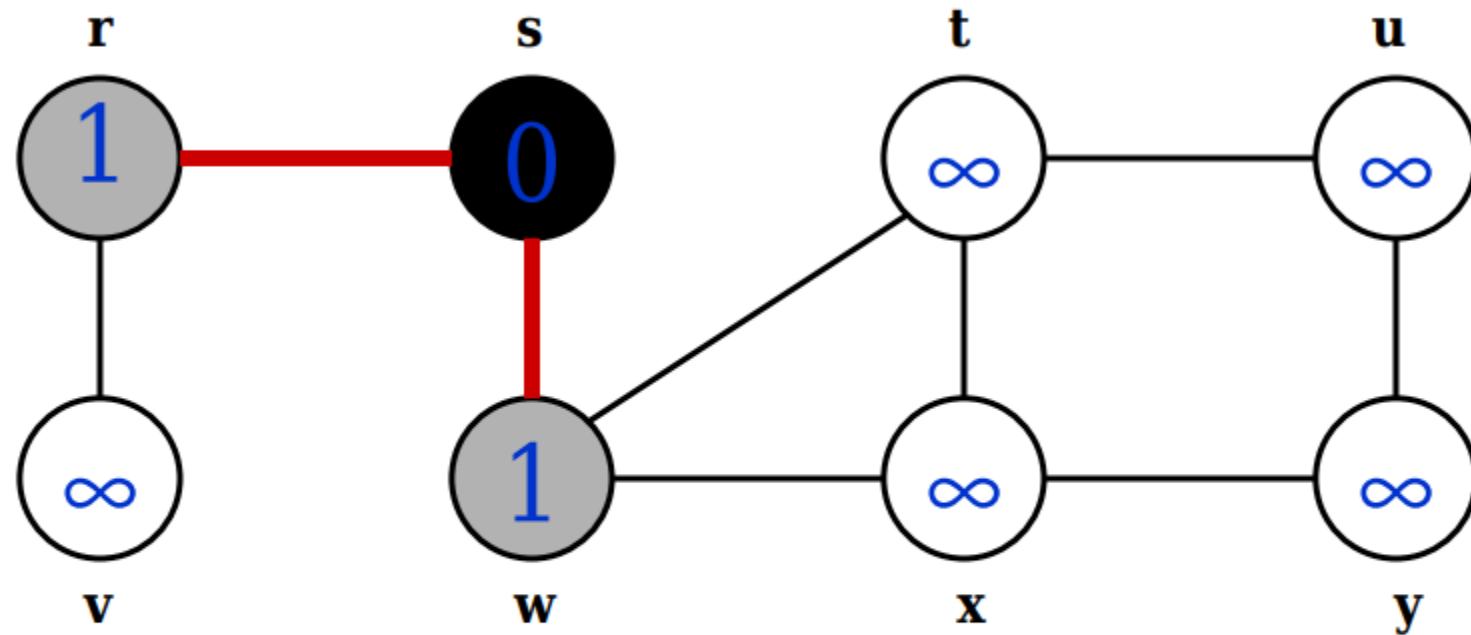


$Q:$

w	r
-----	-----

Busca em Largura

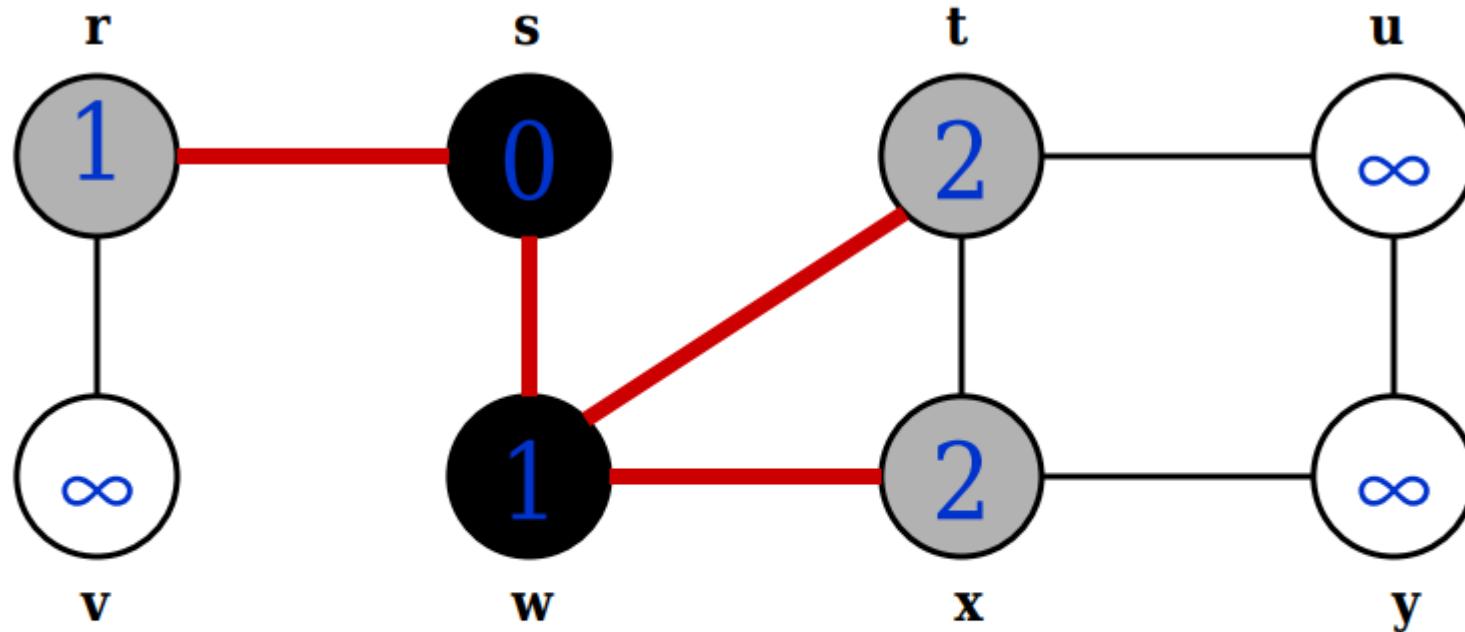
colorimos o vértice com preto (os seus vizinhos já foram descobertos)



Q:

w	r
---	---

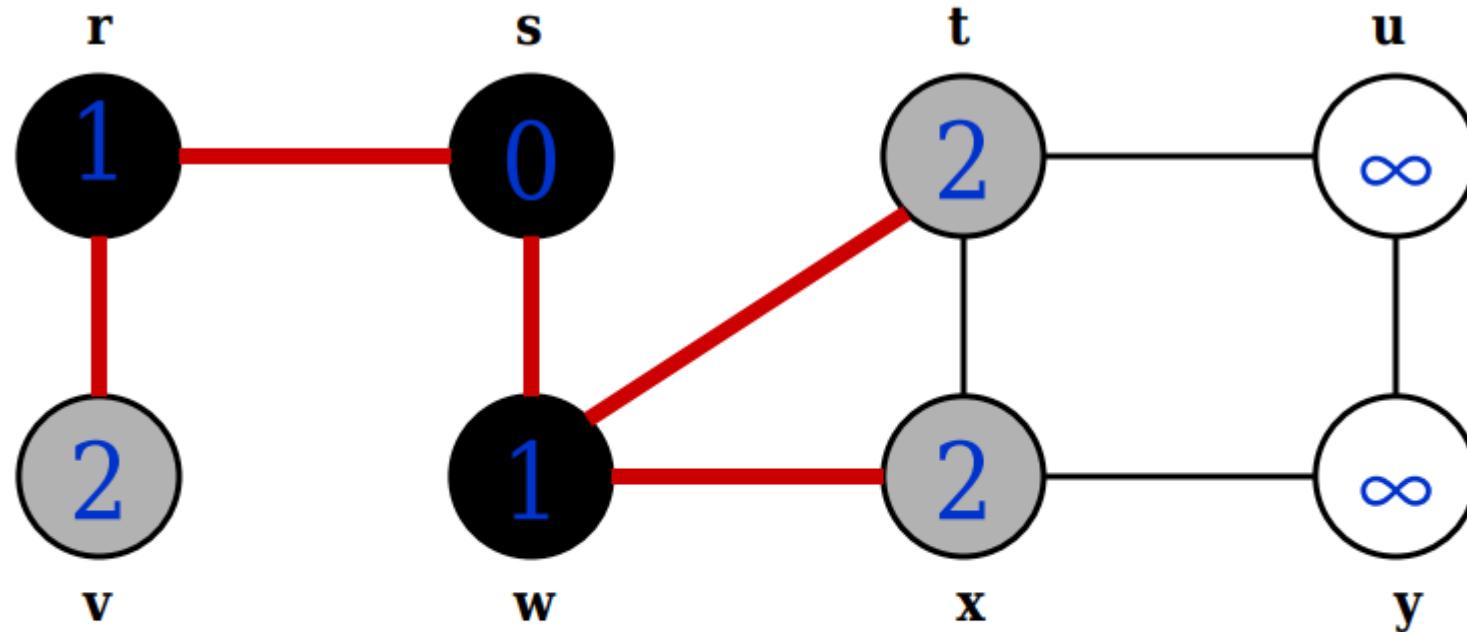
Busca em Largura



Q:

r	t	x
---	---	---

Busca em Largura

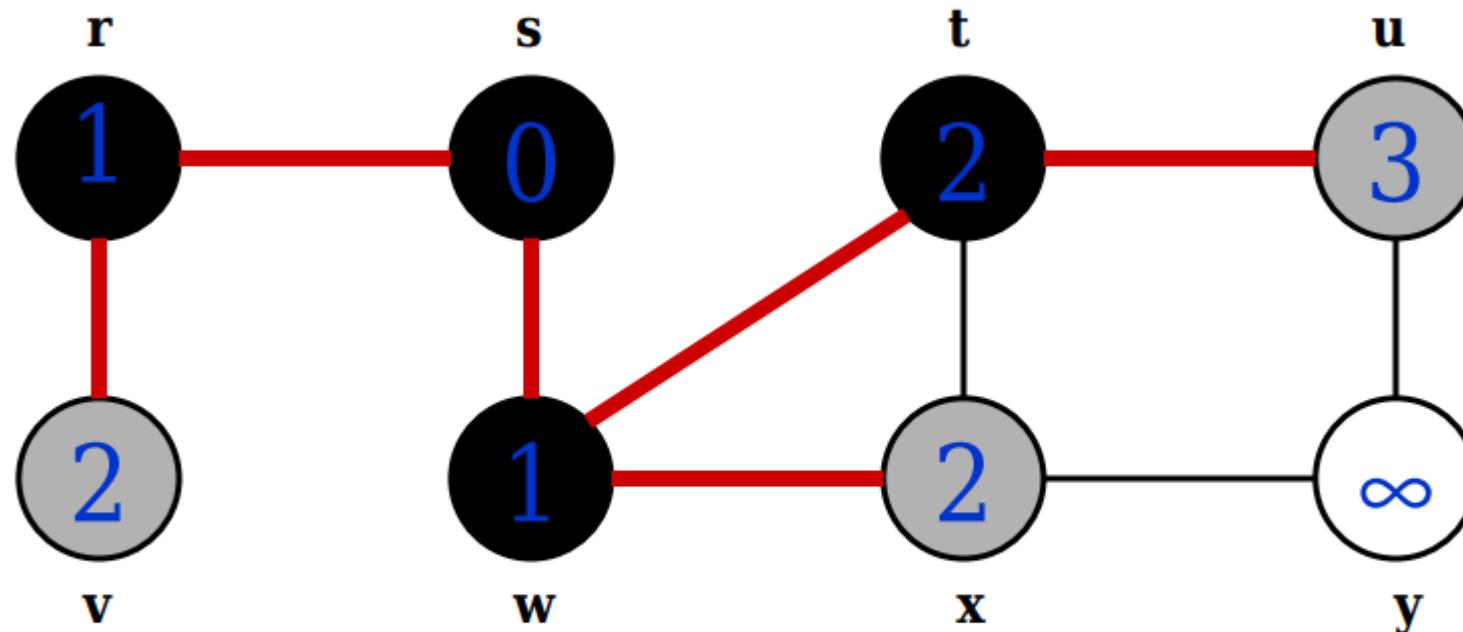


Q:

t	x	v
---	---	---

Busca em Largura

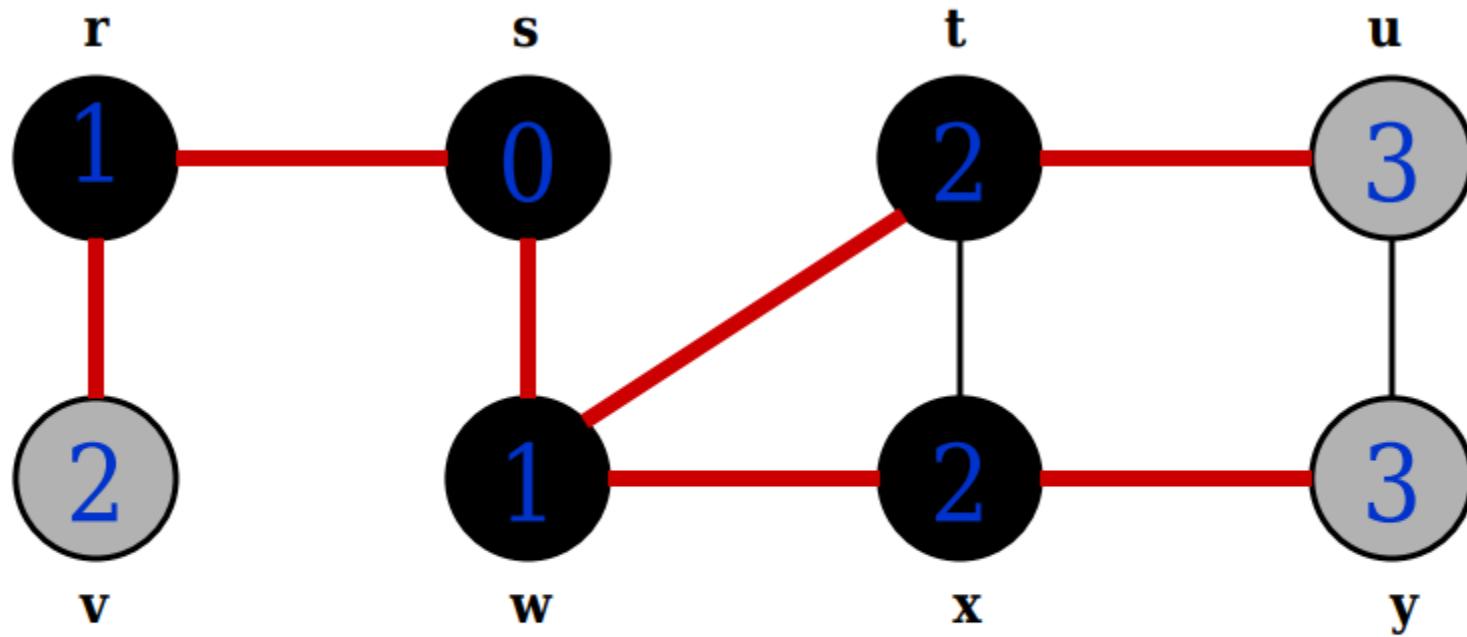
Observe que somente enfileramos vértices brancos, que são imediatamente coloridos de cinza ao entrar na fila



Q:

x	v	u
---	---	---

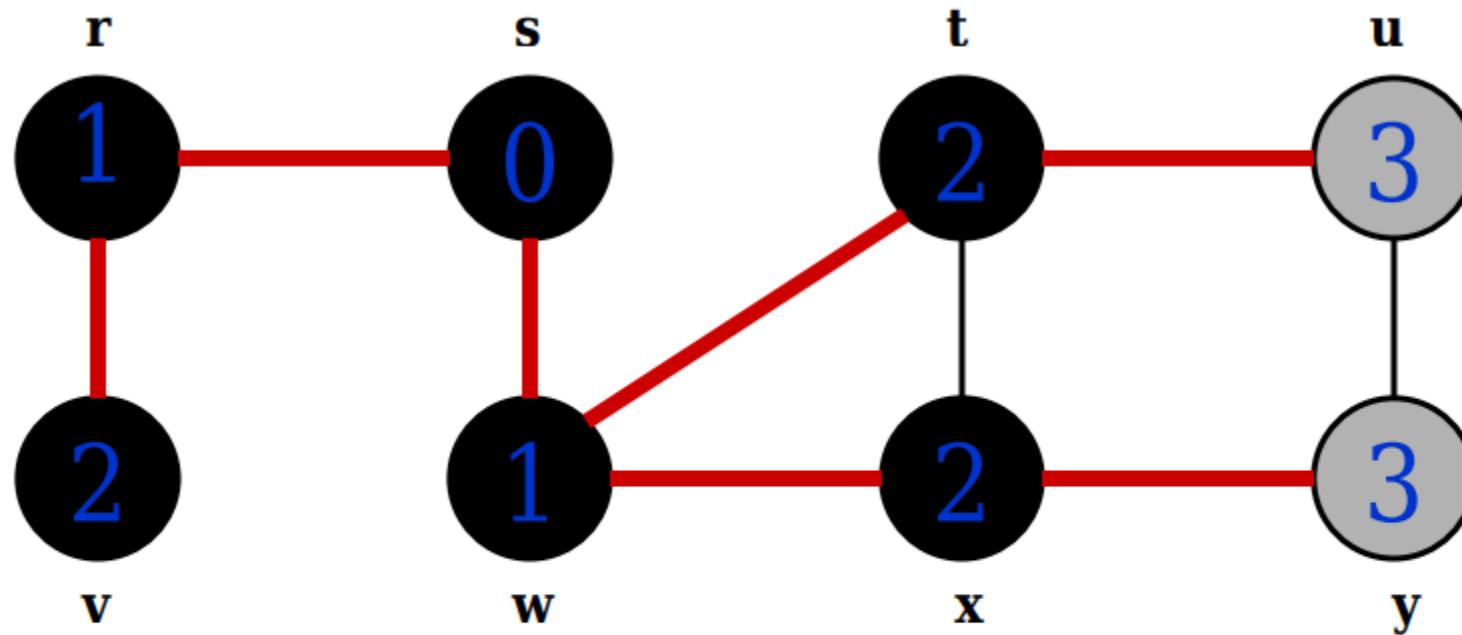
Busca em Largura



Q:

v	u	y
---	---	---

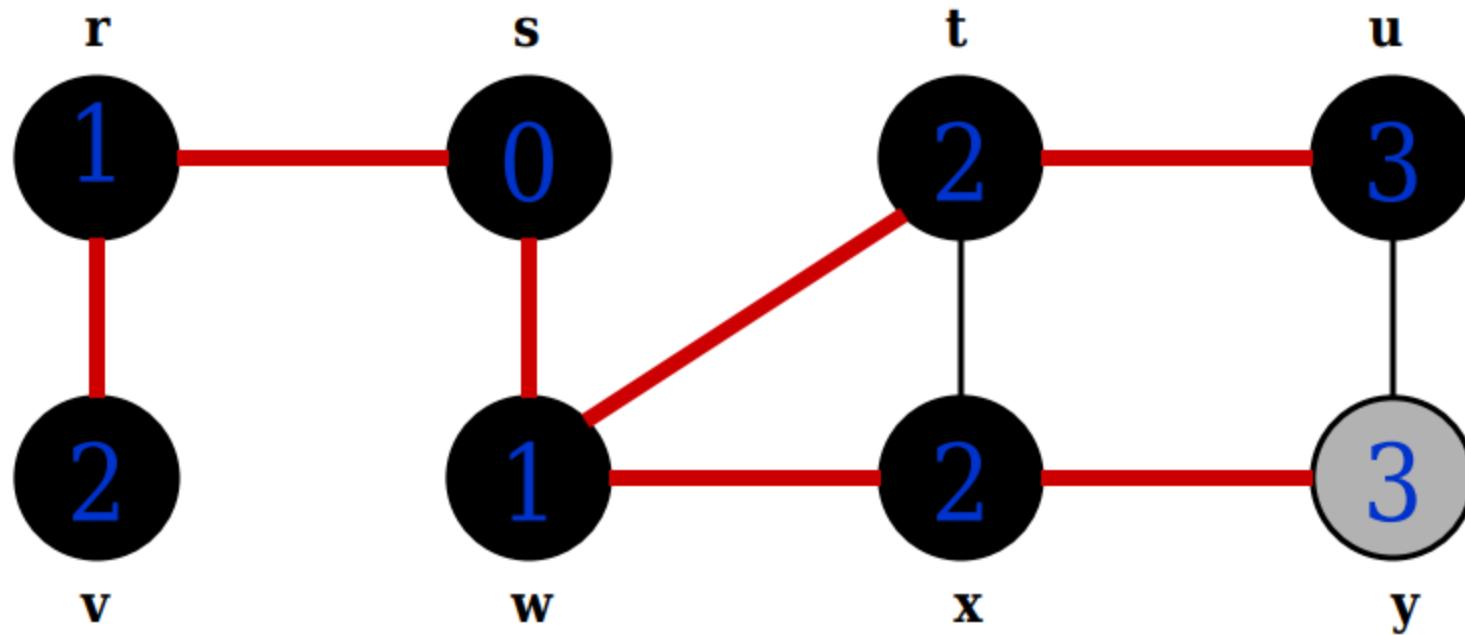
Busca em Largura



Q:

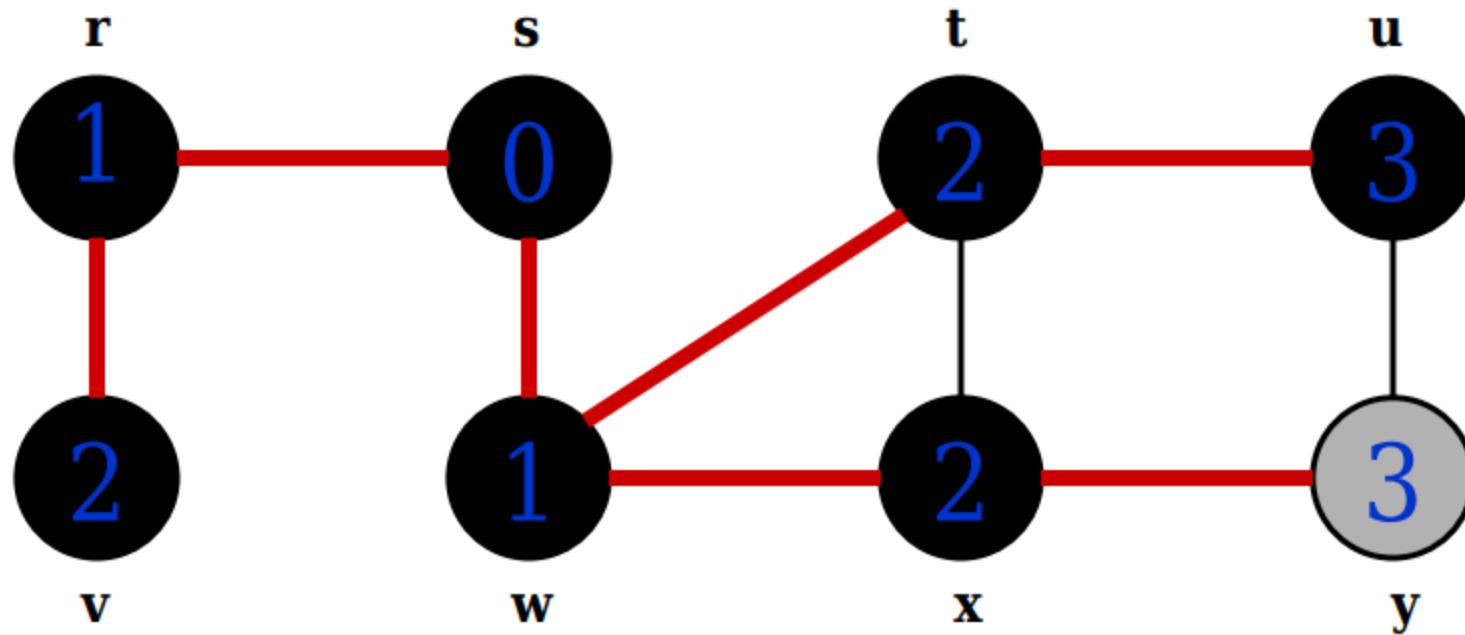
u	y
----------	----------

Busca em Largura



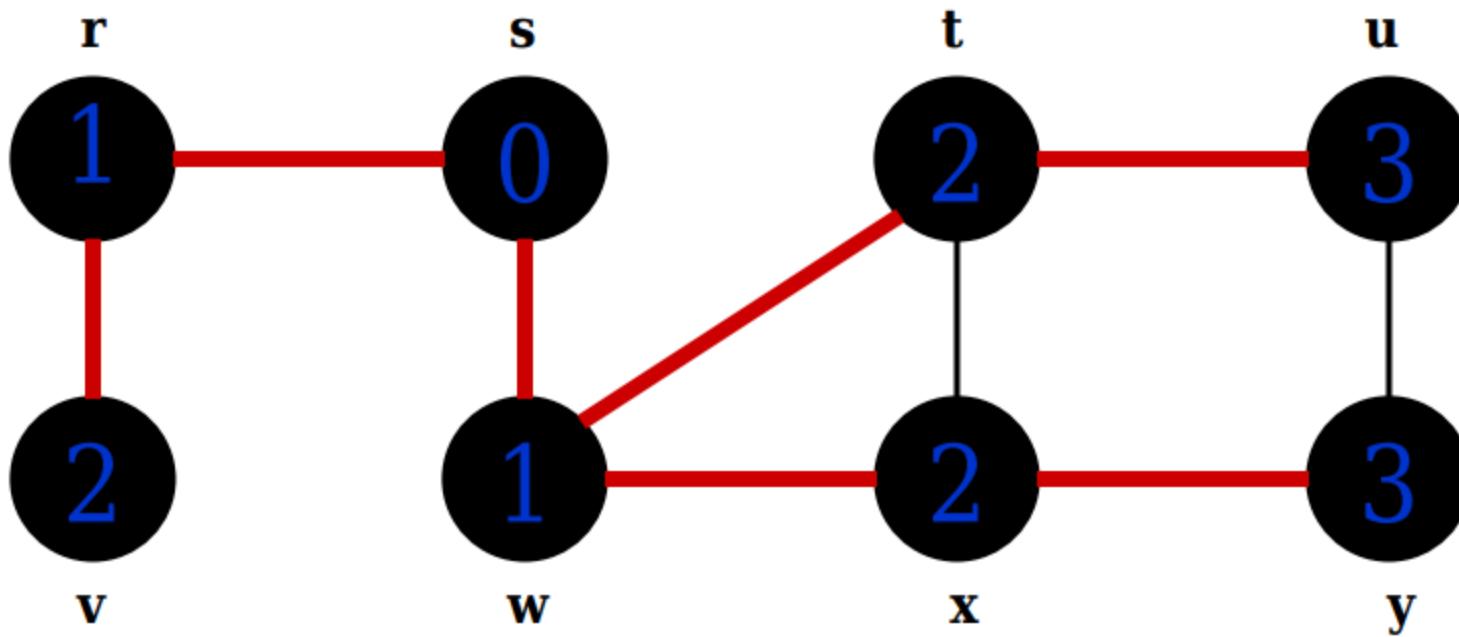
Q:

Busca em Largura



Q:

Busca em Largura



Q: \emptyset

Busca em Largura

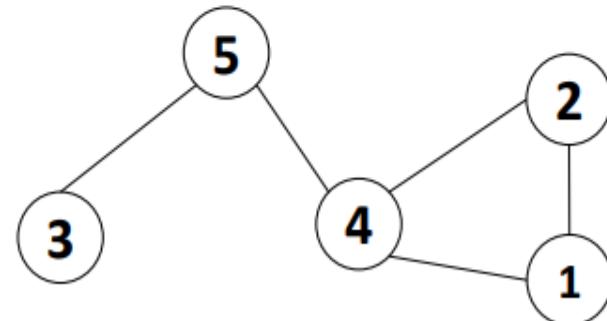
- Problema: Como saber se existe caminho entre dois vértices? Exemplo: existe um caminho entre São Luis e Limeira?
- Solução: usar BFS
- Marcar São Luis como raiz
- Realizar BFS
- Ao terminar BFS se Limeira tiver distância diferente de infinito ou se o vértice está pintado de preto, então há caminho, caso contrário, não há.

Busca em Largura

- 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

Busca em Largura - Implementação

```
private static int buscaEmLargura(No inicio, No fim){
    if (inicio == fim) return 0;
    LinkedList<No> ll = new LinkedList<No>();
    inicio.distancia = 0; inicio.visitado = true;
    ll.add(inicio);
    No atual;
    while (!ll.isEmpty()){
        atual = ll.remove();
        for (No temp: atual.vizinhos){
            if (!temp.visitado){
                if (temp == fim) return atual.distancia + 1;
                temp.visitado = true;
                temp.distancia = atual.distancia + 1;
                ll.add(temp);
            }
        }
    }
    return -1;
}
```



```

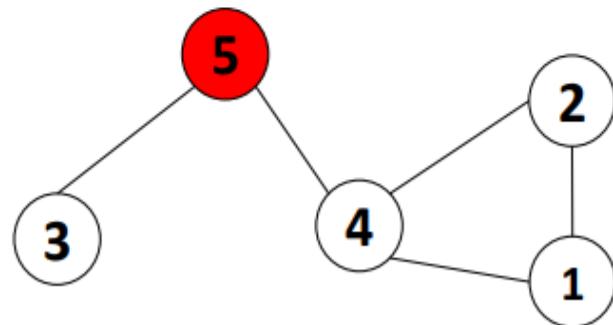
private static int buscaEmLargura(No inicio, No fim){
    if (inicio == fim) return 0;
    LinkedList<No> ll = new LinkedList<No>();
    inicio.distancia = 0; inicio.visitado = true;
    ll.add(inicio);
    No atual;
    while (!ll.isEmpty()){
        atual = ll.remove();
        for (No temp: atual.vizinhos){
            if (!temp.visitado){
                if (temp == fim) return atual.distancia + 1;
                temp.visitado = true;
                temp.distancia = atual.distancia + 1;
                ll.add(temp);
            }
        }
    }
    return -1;
}

```

buscando o nó 2

atual=5 ; dist=0

Fila: <vazia>



```

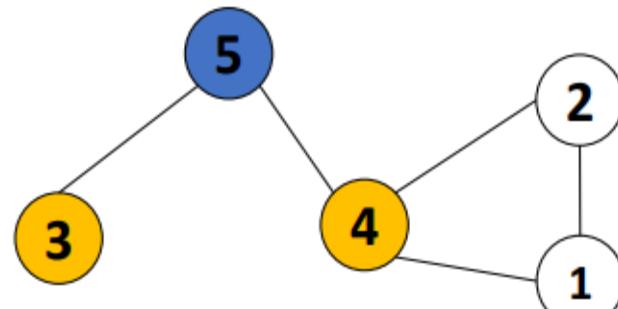
private static int buscaEmLargura(No inicio, No fim){
    if (inicio == fim) return 0;
    LinkedList<No> ll = new LinkedList<No>();
    inicio.distancia = 0; inicio.visitado = true;
    ll.add(inicio);
    No atual;
    while (!ll.isEmpty()){
        atual = ll.remove();
        for (No temp: atual.vizinhos){
            if (!temp.visitado){
                if (temp == fim) return atual.distancia + 1;
                temp.visitado = true;
                temp.distancia = atual.distancia + 1;
                ll.add(temp);
            }
        }
    }
    return -1;
}

```

buscando o nó 2

atual=5 ; dist=0

Fila: 3, 4



```

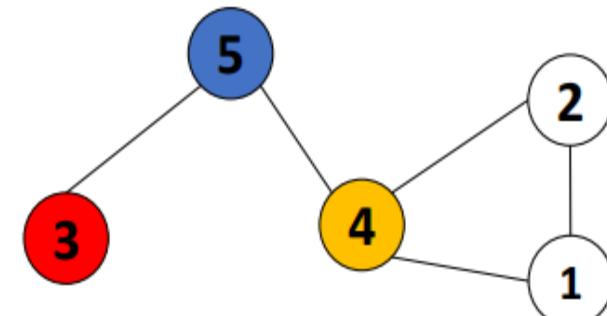
private static int buscaEmLargura(No inicio, No fim){
    if (inicio == fim) return 0;
    LinkedList<No> ll = new LinkedList<No>();
    inicio.distancia = 0; inicio.visitado = true;
    ll.add(inicio);
    No atual;
    while (!ll.isEmpty()){
        atual = ll.remove();
        for (No temp: atual.vizinhos){
            if (!temp.visitado){
                if (temp == fim) return atual.distancia + 1;
                temp.visitado = true;
                temp.distancia = atual.distancia + 1;
                ll.add(temp);
            }
        }
    }
    return -1;
}

```

buscando o nó 2

atual=3 ; dist=1

Fila: 4



```

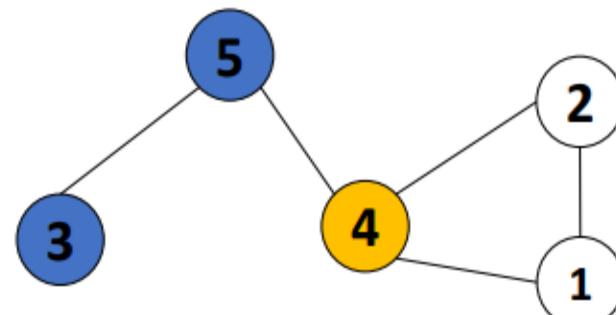
private static int buscaEmLargura(No inicio, No fim){
    if (inicio == fim) return 0;
    LinkedList<No> ll = new LinkedList<No>();
    inicio.distancia = 0; inicio.visitado = true;
    ll.add(inicio);
    No atual;
    while (!ll.isEmpty()){
        atual = ll.remove();
        for (No temp: atual.vizinhos){
            if (!temp.visitado){
                if (temp == fim) return atual.distancia + 1;
                temp.visitado = true;
                temp.distancia = atual.distancia + 1;
                ll.add(temp);
            }
        }
    }
    return -1;
}

```

buscando o nó 2

atual=3 ; dist=0

Fila: 4



```

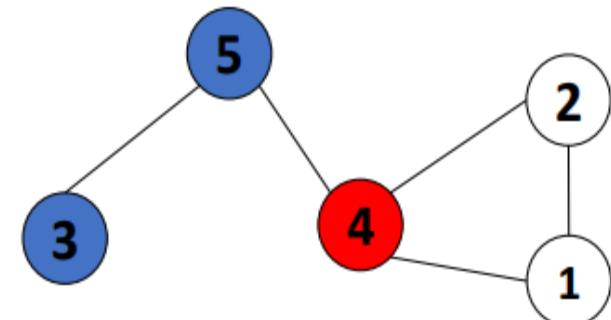
private static int buscaEmLargura(No inicio, No fim){
    if (inicio == fim) return 0;
    LinkedList<No> ll = new LinkedList<No>();
    inicio.distancia = 0; inicio.visitado = true;
    ll.add(inicio);
    No atual;
    while (!ll.isEmpty()){
        atual = ll.remove();
        for (No temp: atual.vizinhos){
            if (!temp.visitado){
                if (temp == fim) return atual.distancia + 1;
                temp.visitado = true;
                temp.distancia = atual.distancia + 1;
                ll.add(temp);
            }
        }
    }
    return -1;
}

```

buscando o nó 2

atual=4 ; dist=1

Fila:



```

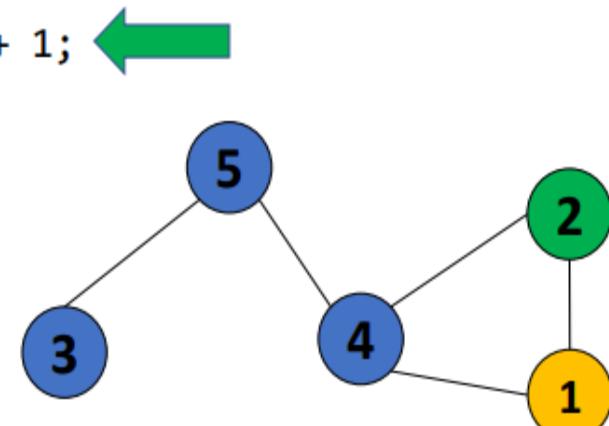
private static int buscaEmLargura(No inicio, No fim){
    if (inicio == fim) return 0;
    LinkedList<No> ll = new LinkedList<No>();
    inicio.distancia = 0; inicio.visitado = true;
    ll.add(inicio);
    No atual;
    while (!ll.isEmpty()){
        atual = ll.remove();
        for (No temp: atual.vizinhos){
            if (!temp.visitado){
                if (temp == fim) return atual.distancia + 1;
                temp.visitado = true;
                temp.distancia = atual.distancia + 1;
                ll.add(temp);
            }
        }
    }
    return -1;
}

```

buscando o nó 2

atual=4 ; dist=1

Fila: 1



```

private static int buscaEmLargura(No inicio, No fim){
    if (inicio == fim) return 0;
    LinkedList<No> ll = new LinkedList<No>();
    inicio.distancia = 0; inicio.visitado = true;
    ll.add(inicio);
    No atual;
    while (!ll.isEmpty()){
        atual = ll.remove();
        for (No temp: atual.vizinhos){
            if (!temp.visitado){
                if (temp == fim) return atual.distancia + 1;
                temp.visitado = true;
                temp.distancia = atual.distancia + 1;
                ll.add(temp);
            }
        }
    }
    return -1;
}

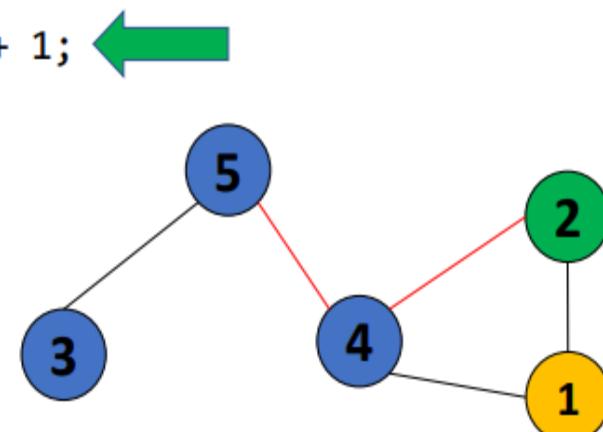
```

buscando o nó 2

atual=4 ; dist=1

Fila: 1

Distância = 2



Busca em Largura

- Considerada também como uma busca por nível.
- Característica: visita primeiro todos os nós próximos da raiz da busca (por níveis), antes de visitar os mais distantes.
- **A árvore geradora é uma árvore com menor profundidade (mais larga) e com muitos filhos para cada nó.**
- **Poderá ser implementada utilizando-se uma fila.**

Busca em Profundidade

- É outro método de busca
- 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

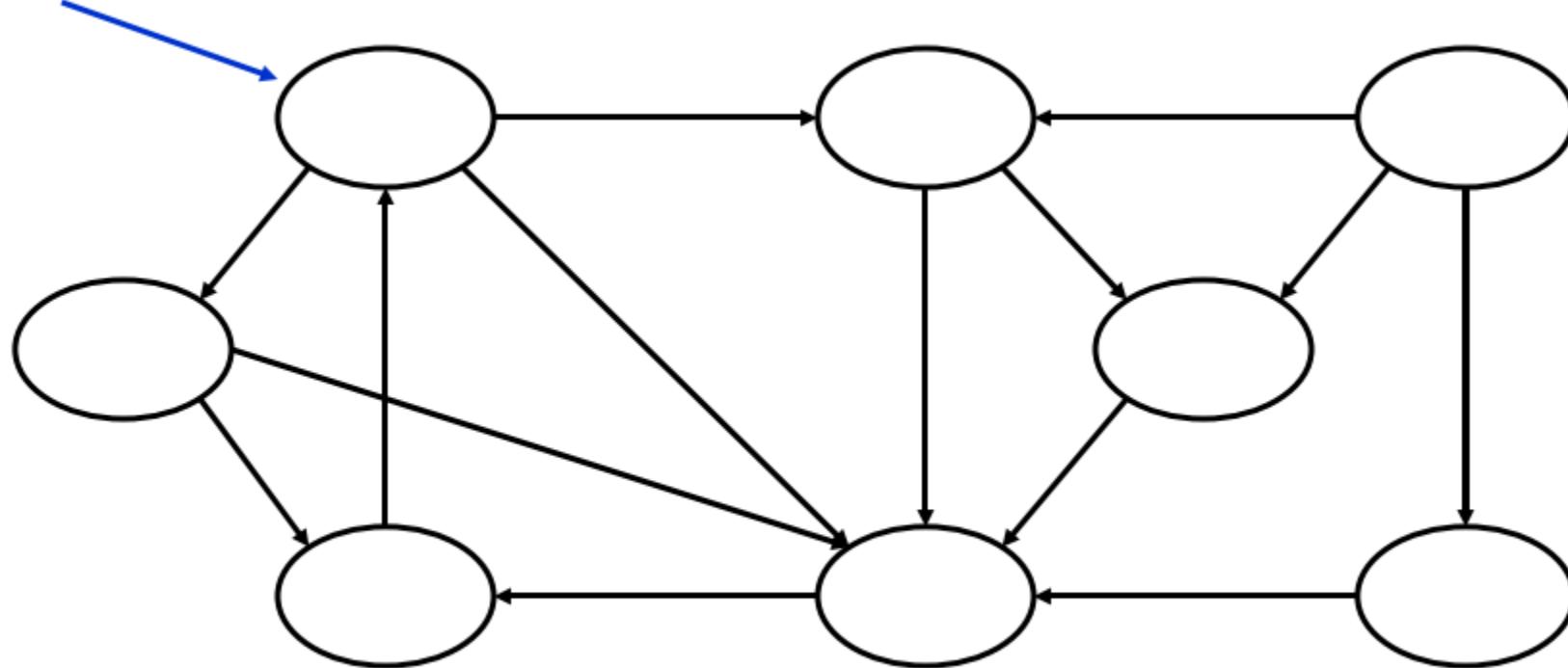
- **Intuição:** Procurar uma saída de um labirinto
- Vai fundo atrás da saída (tomando decisões a cada encruzilhada)
- Volta a última encruzilhada quando encontrar um beco sem saída ou encontrar um lugar já visitado.

Busca em Profundidade

- 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 .

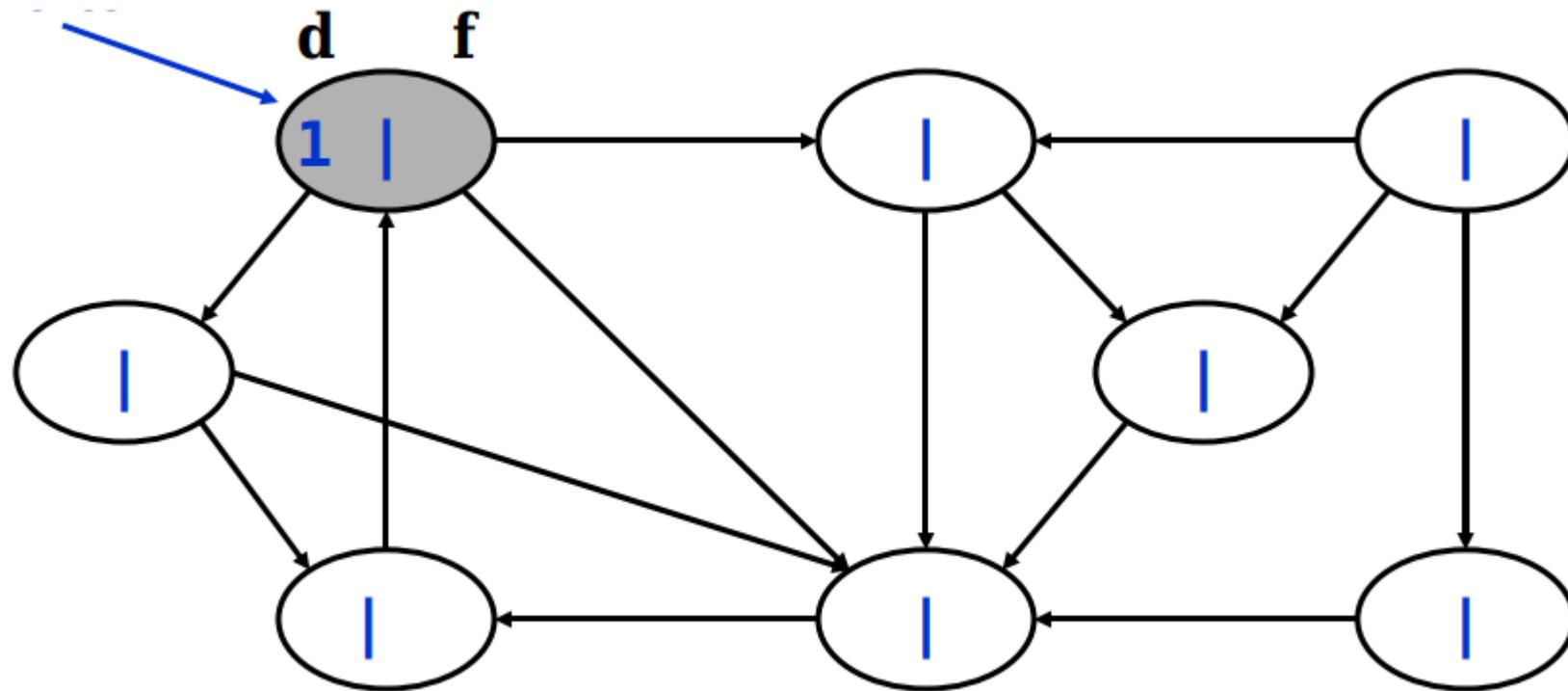
Busca em Profundidade

vértice origem



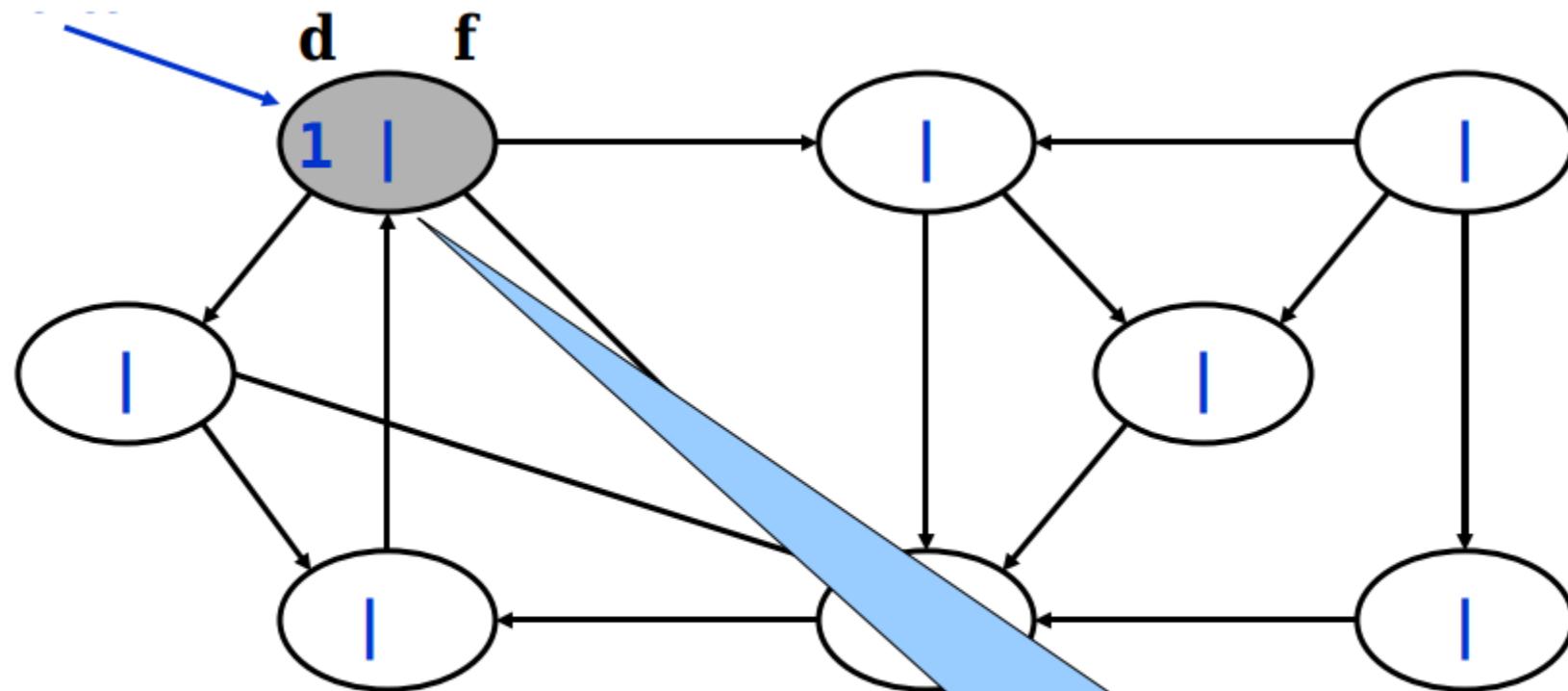
Busca em Profundidade

vértice origem



Busca em Profundidade

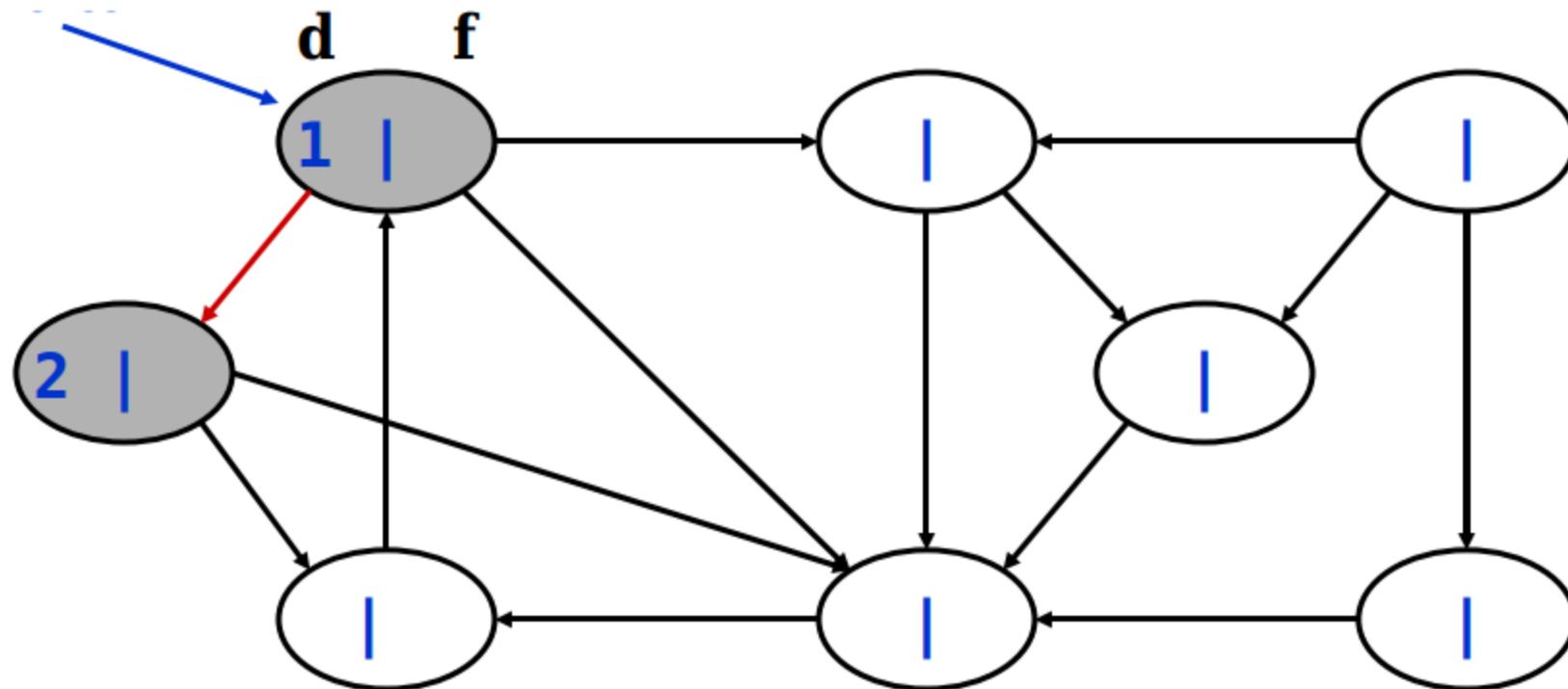
vértice origem



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

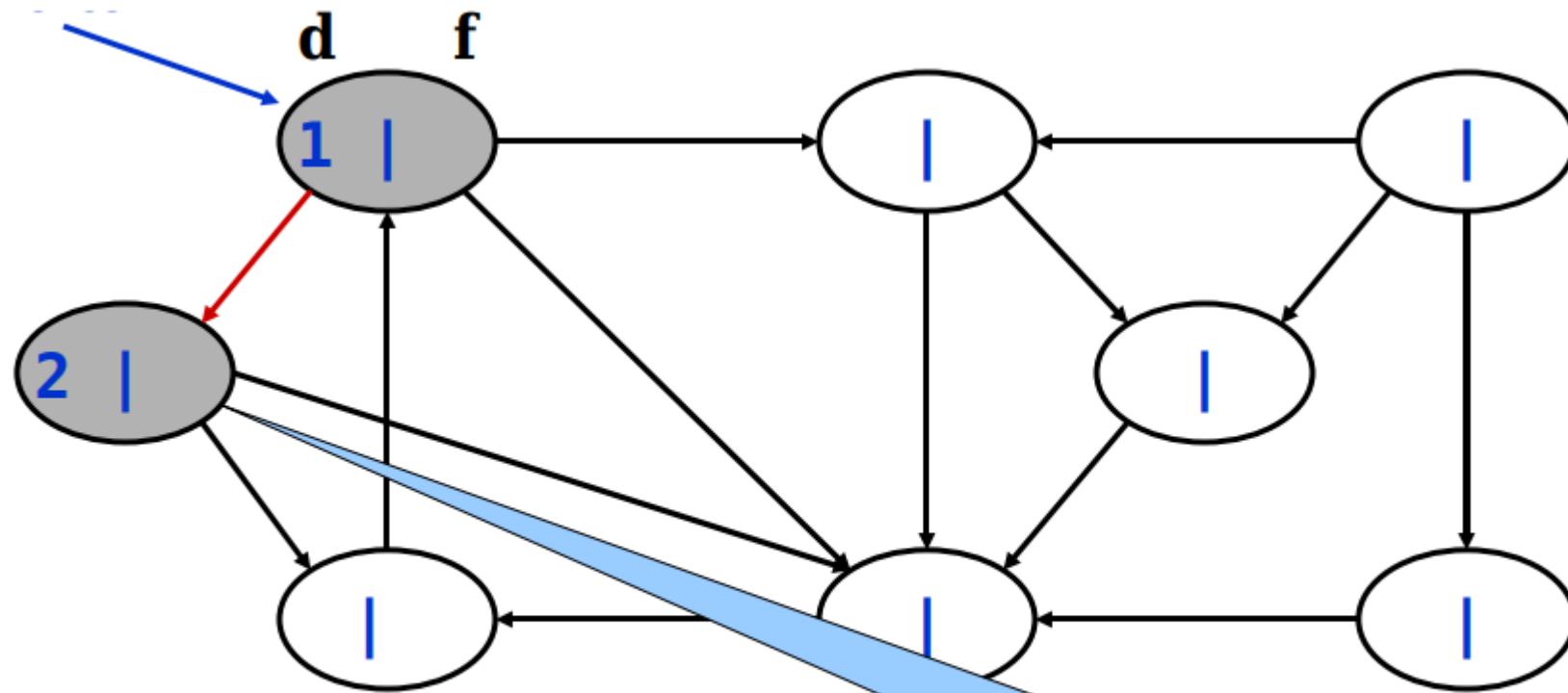
Busca em Profundidade

vértice origem



Busca em Profundidade

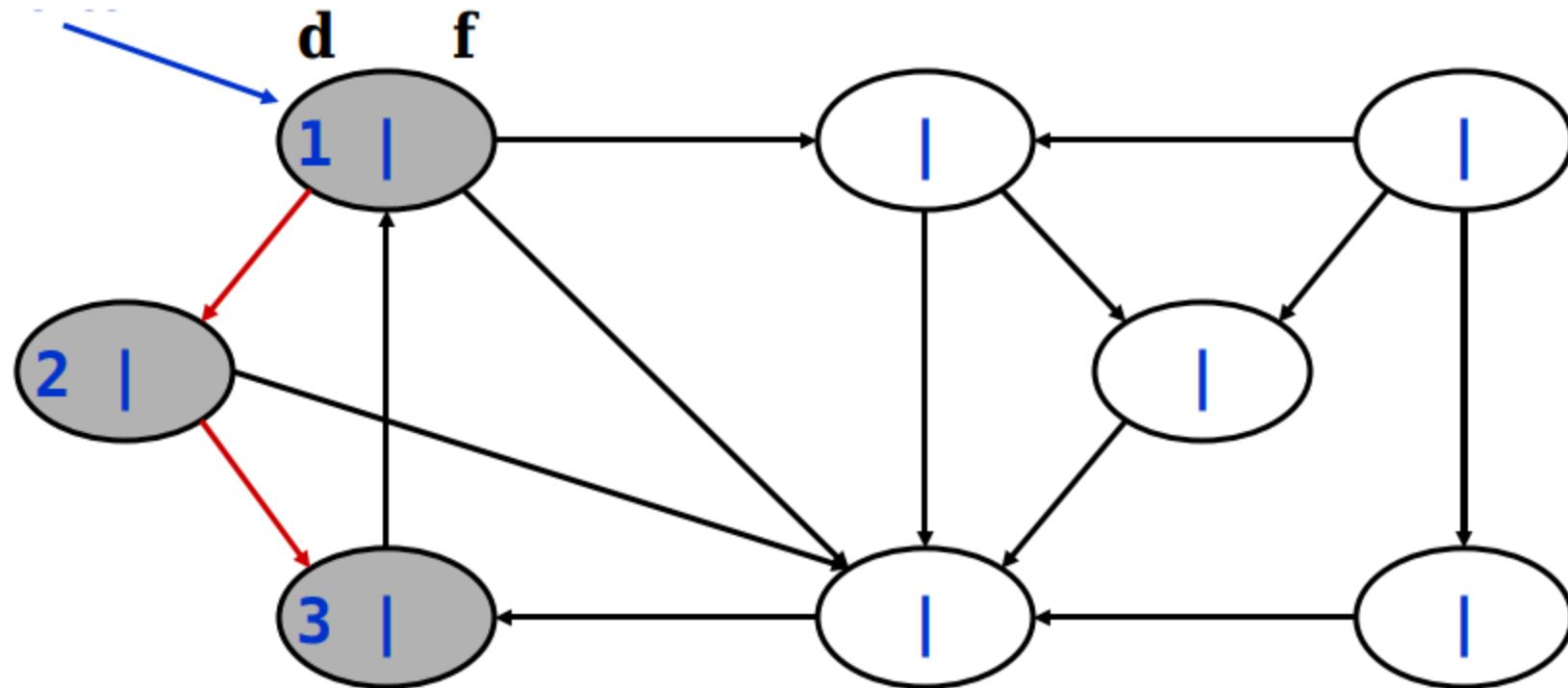
vértice origem



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

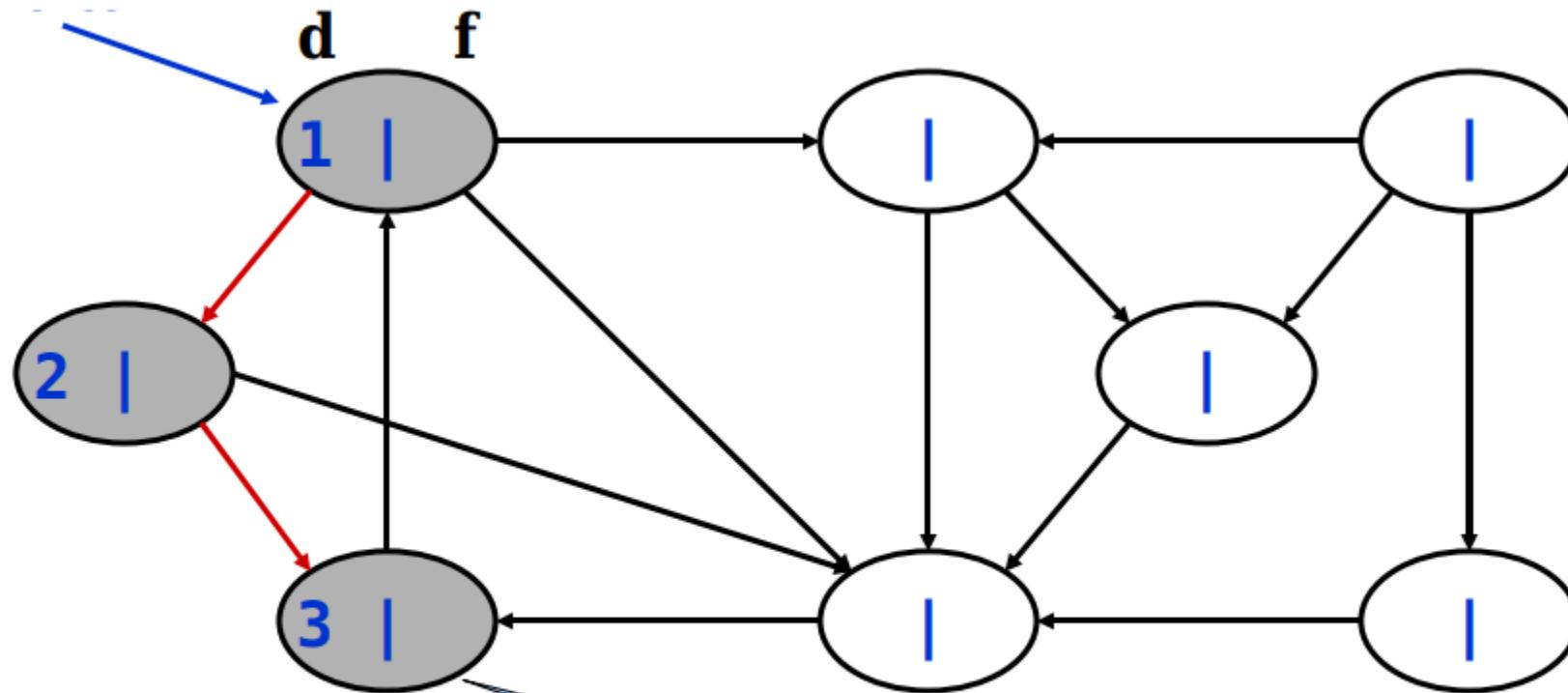
Busca em Profundidade

vértice origem



Busca em Profundidade

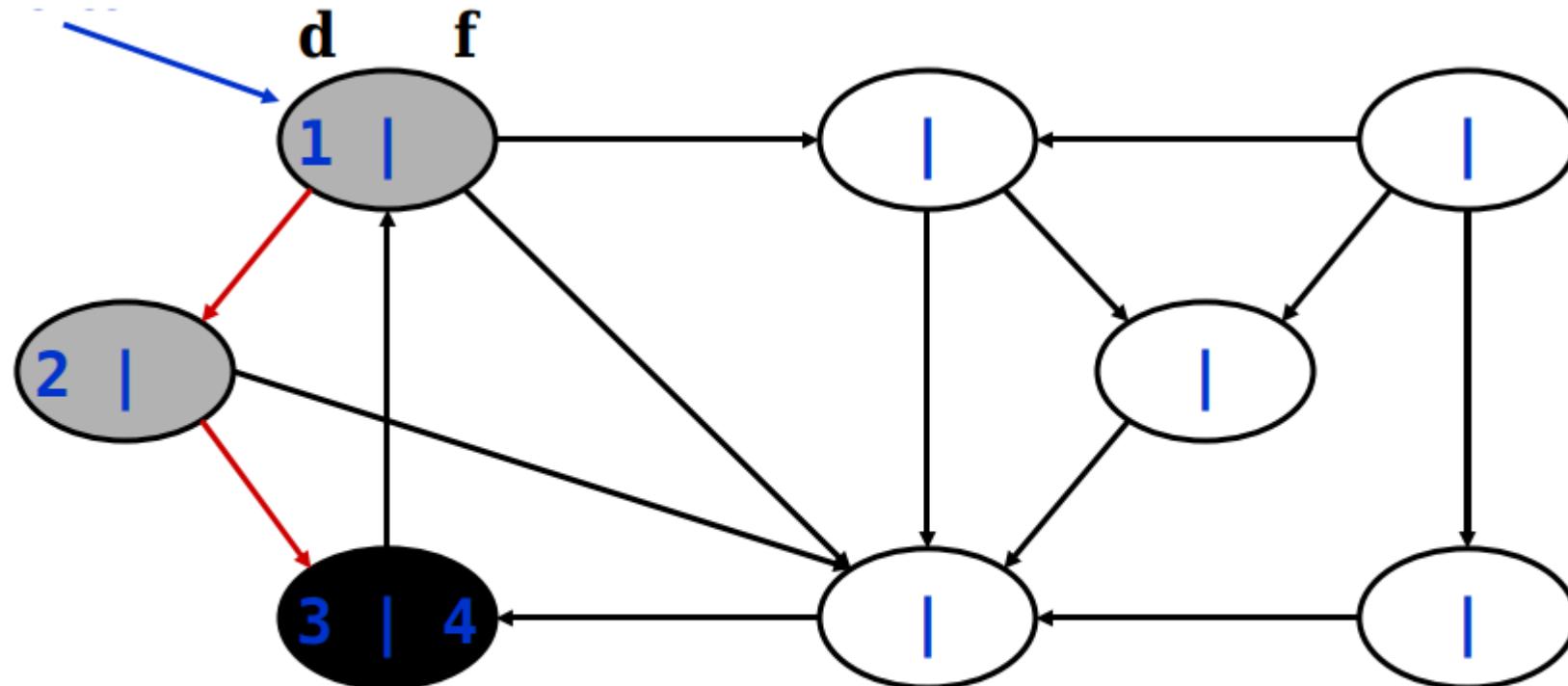
vértice origem



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

Busca em Profundidade

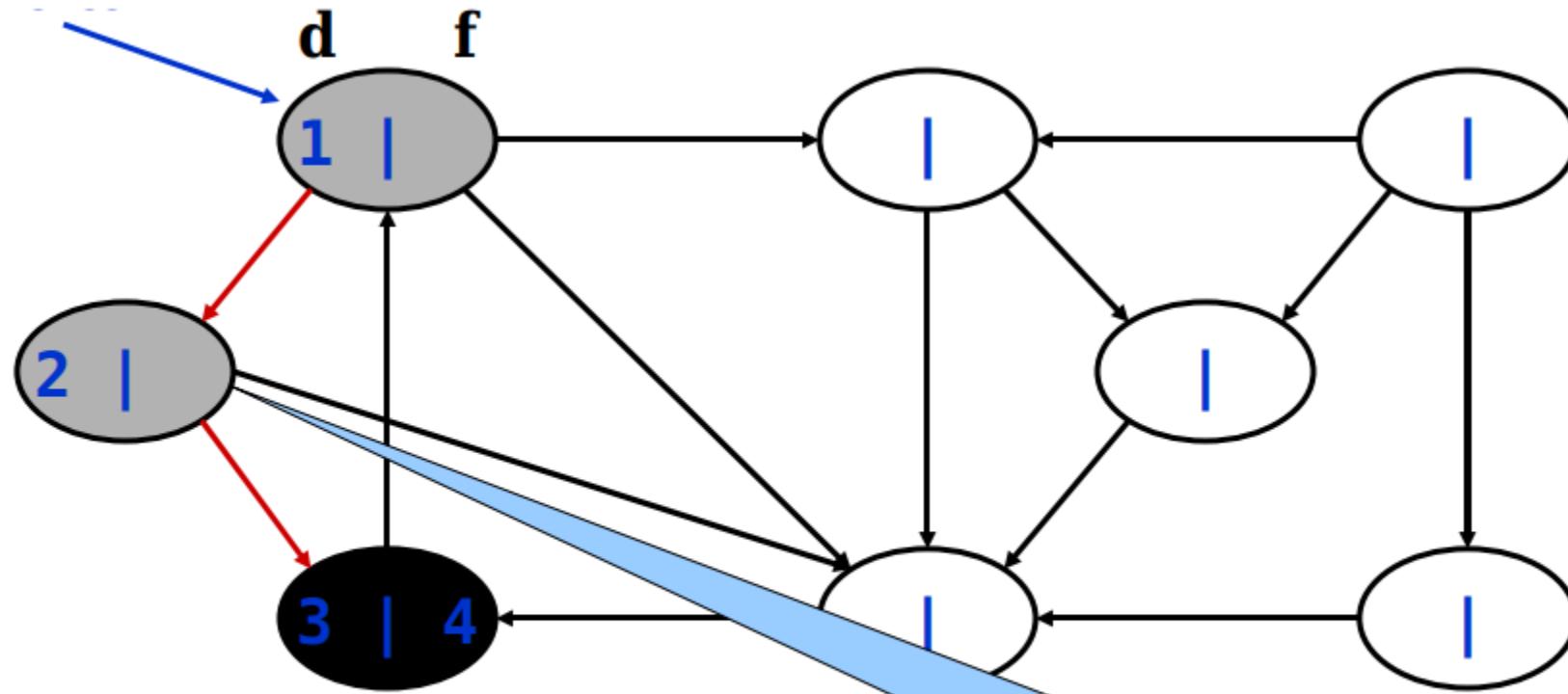
vértice origem



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

Busca em Profundidade

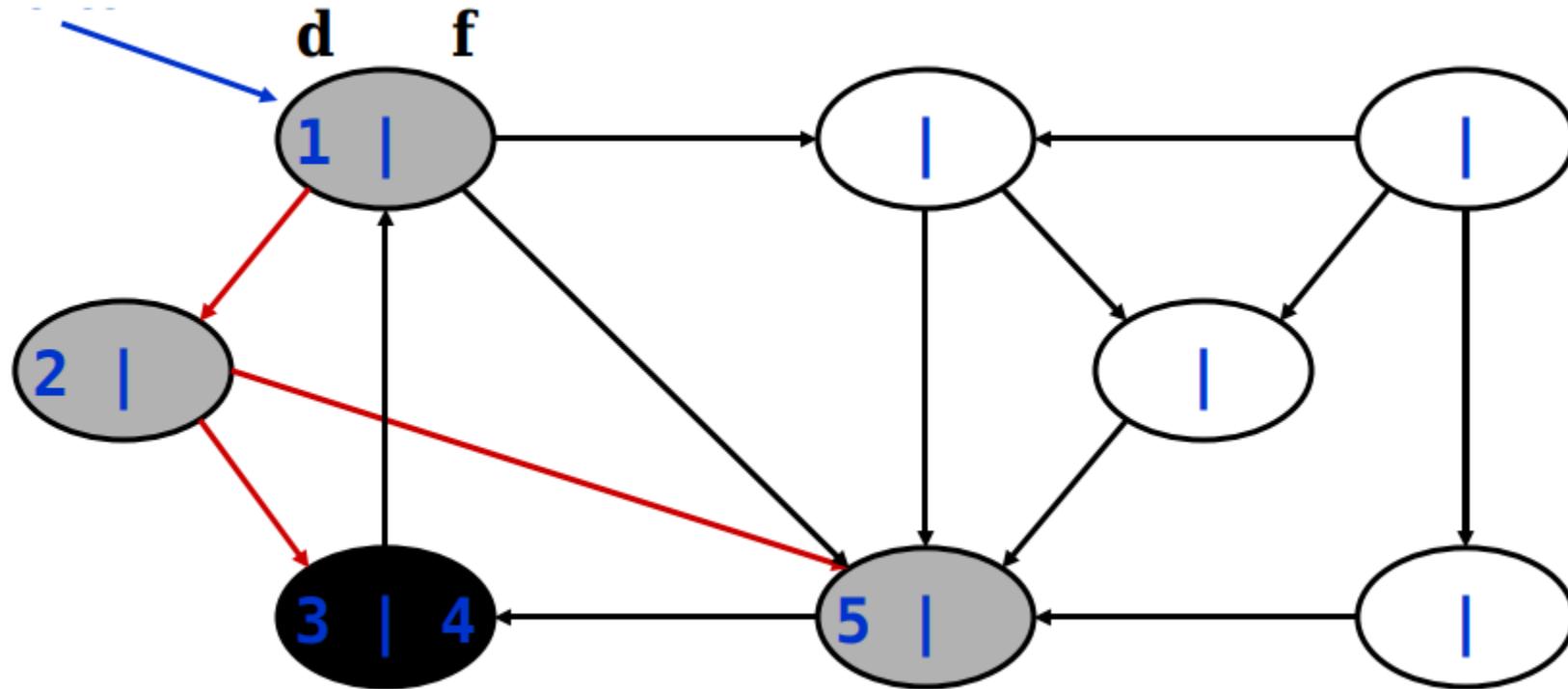
vértice origem



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

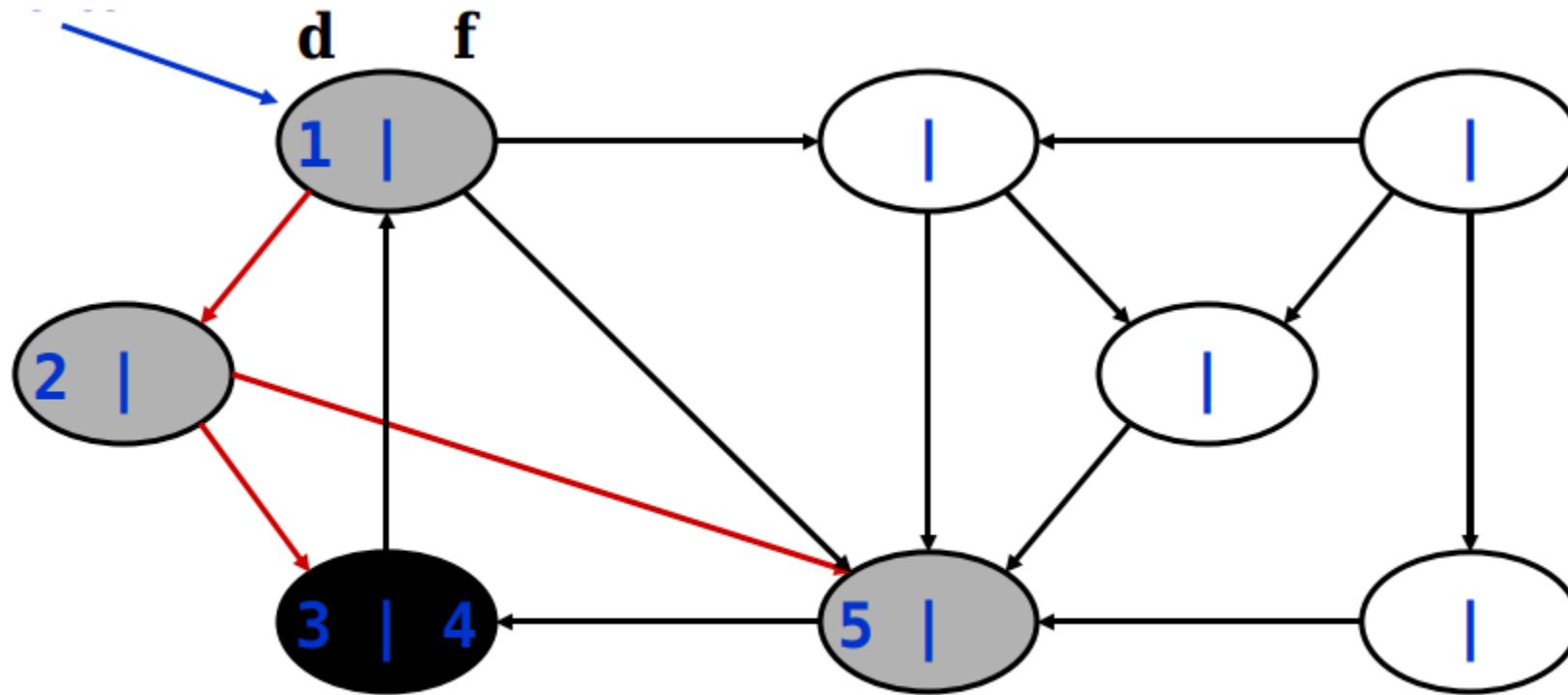
Busca em Profundidade

vértice origem



Busca em Profundidade

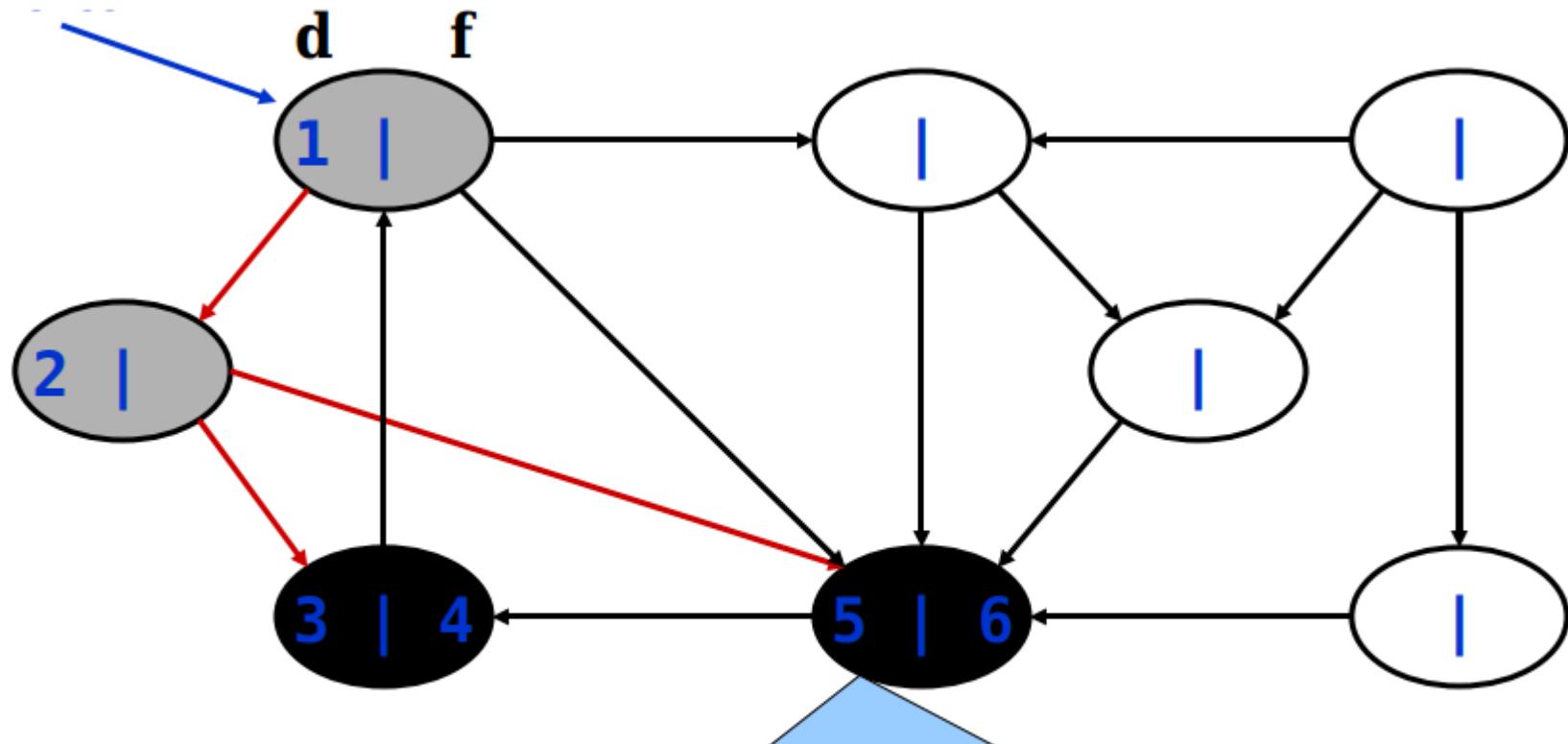
vértice origem



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

Busca em Profundidade

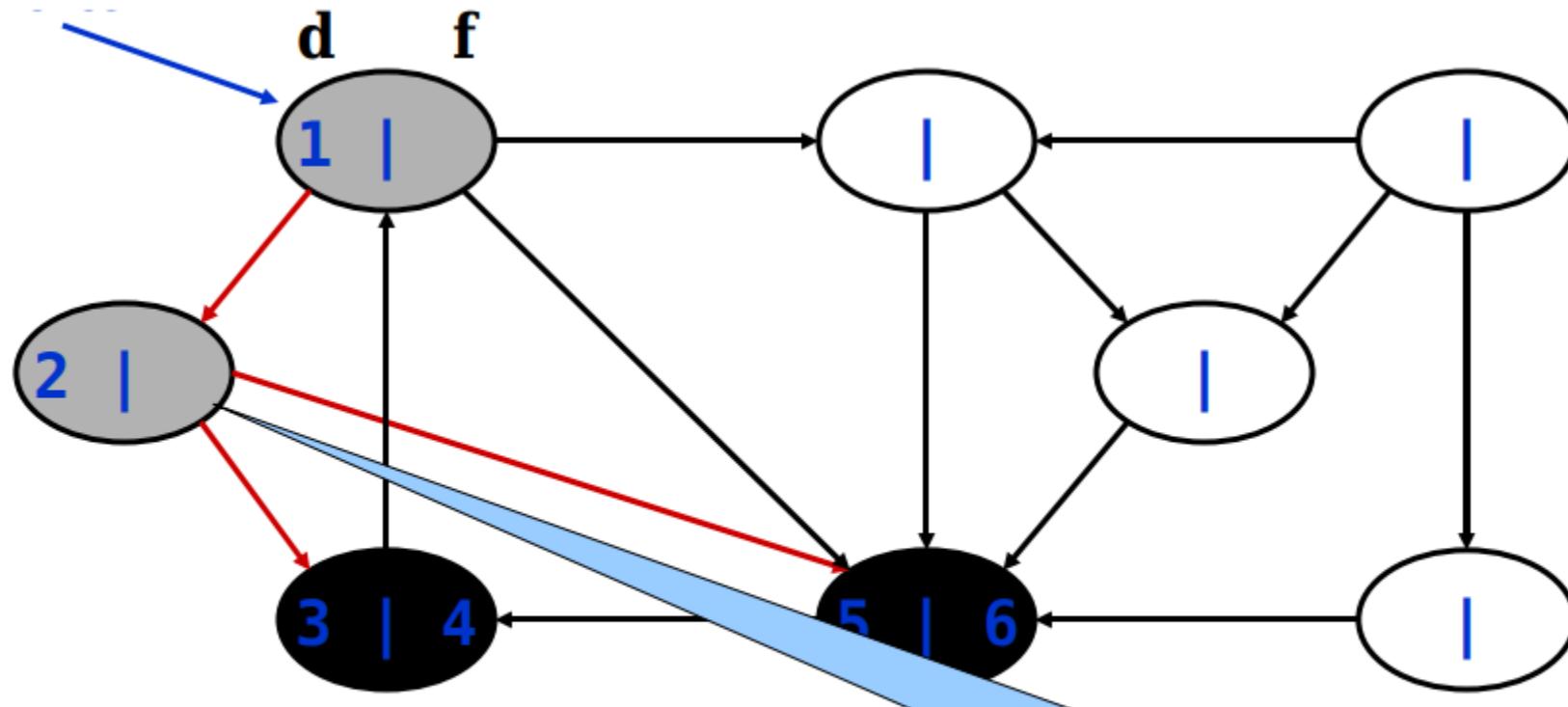
vértice origem



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

Busca em Profundidade

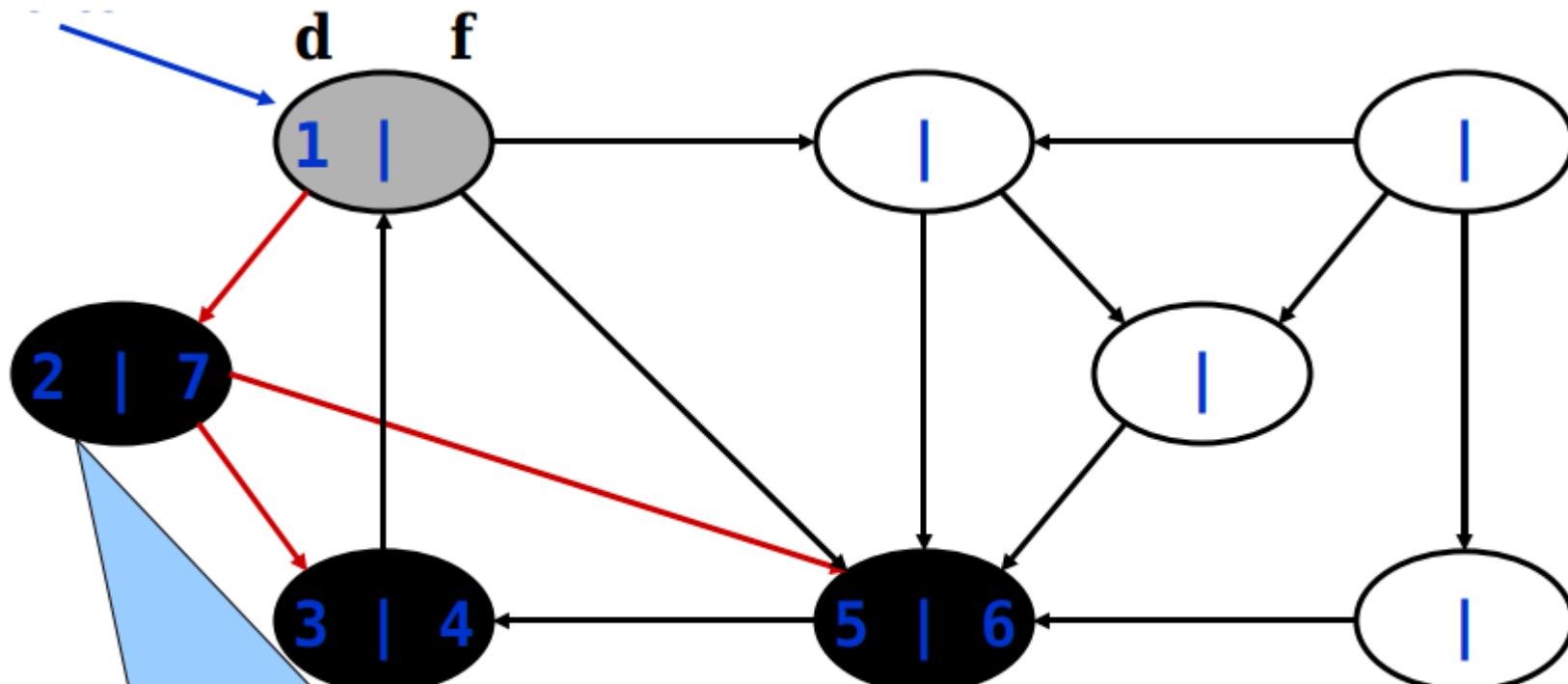
vértice origem



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

Busca em Profundidade

vértice origem

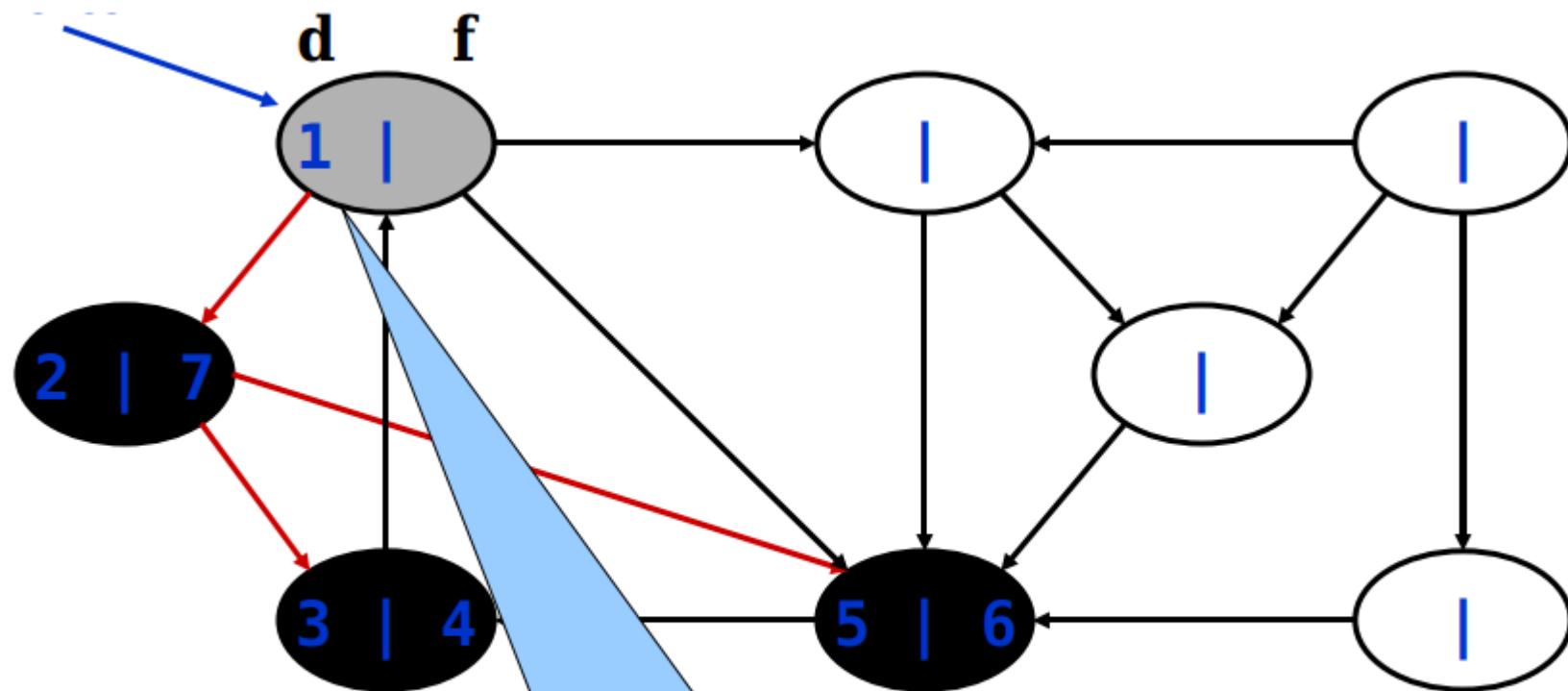


Não existe. Terminei!

Volta-se na busca para o precursor
desse vértice.

Busca em Profundidade

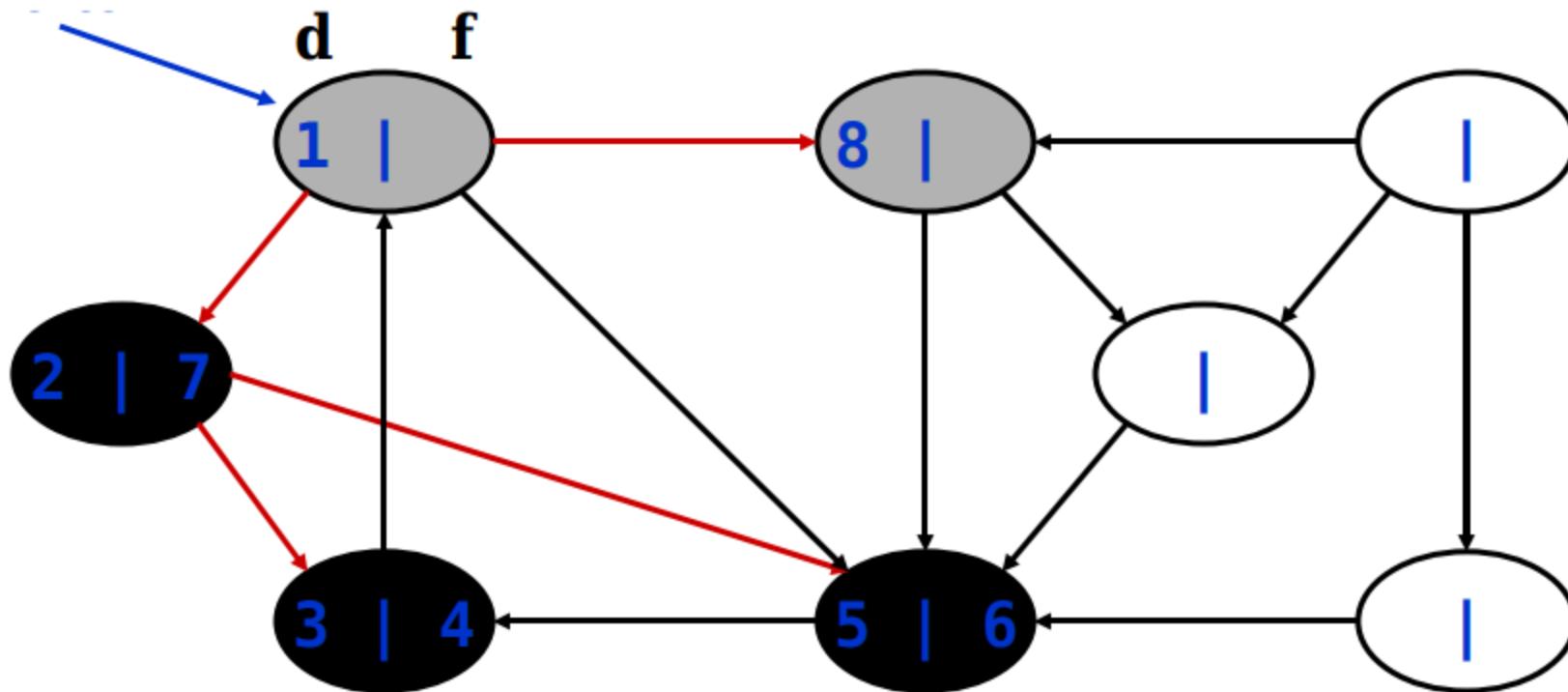
vértice origem



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

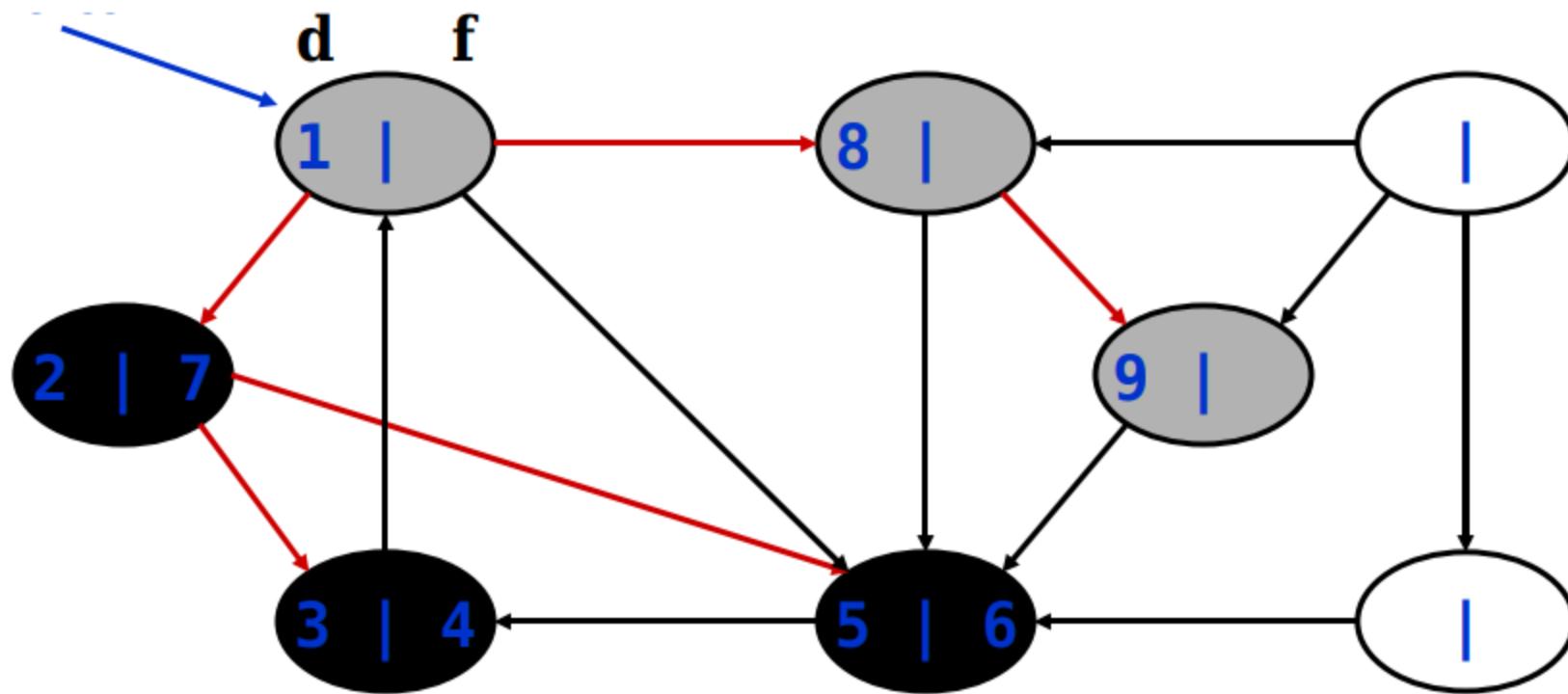
Busca em Profundidade

vértice origem



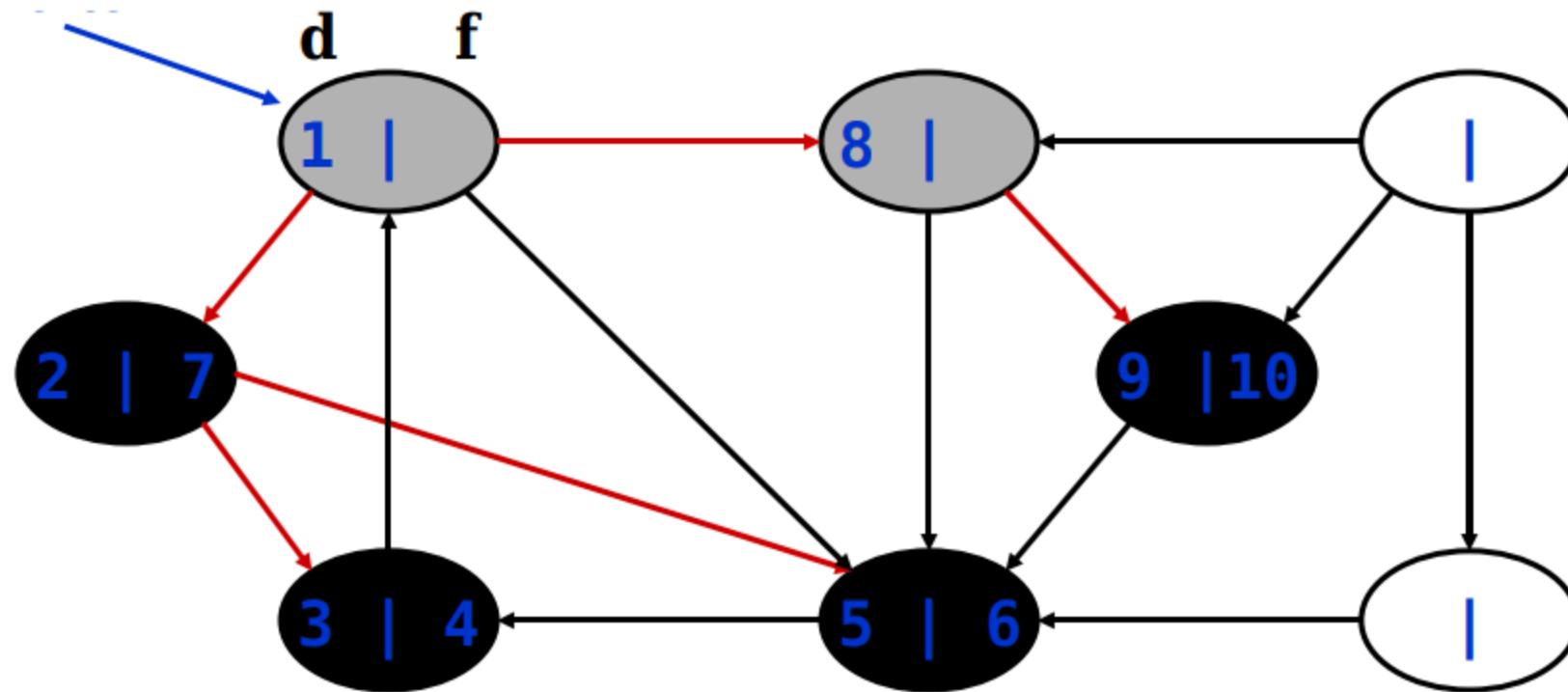
Busca em Profundidade

vértice origem



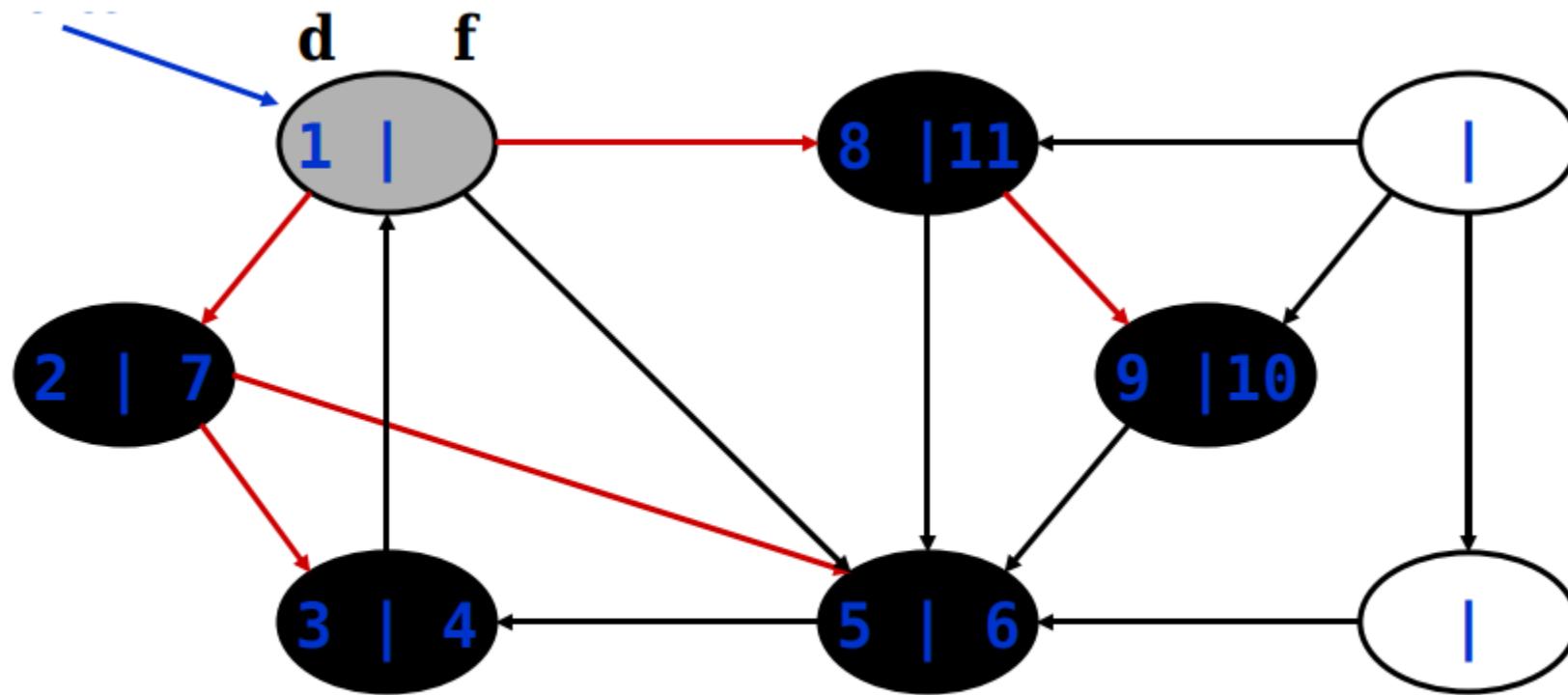
Busca em Profundidade

vértice origem

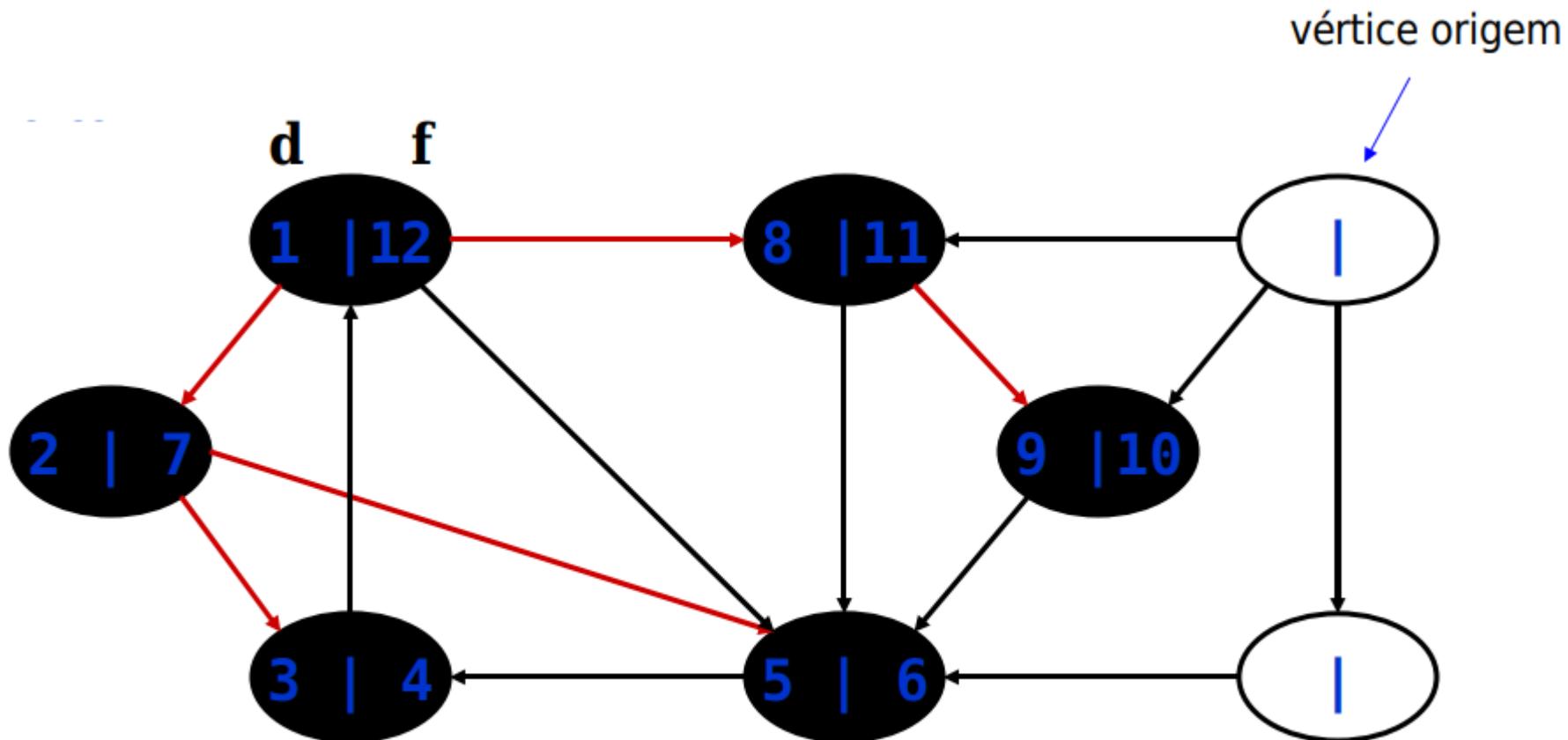


Busca em Profundidade

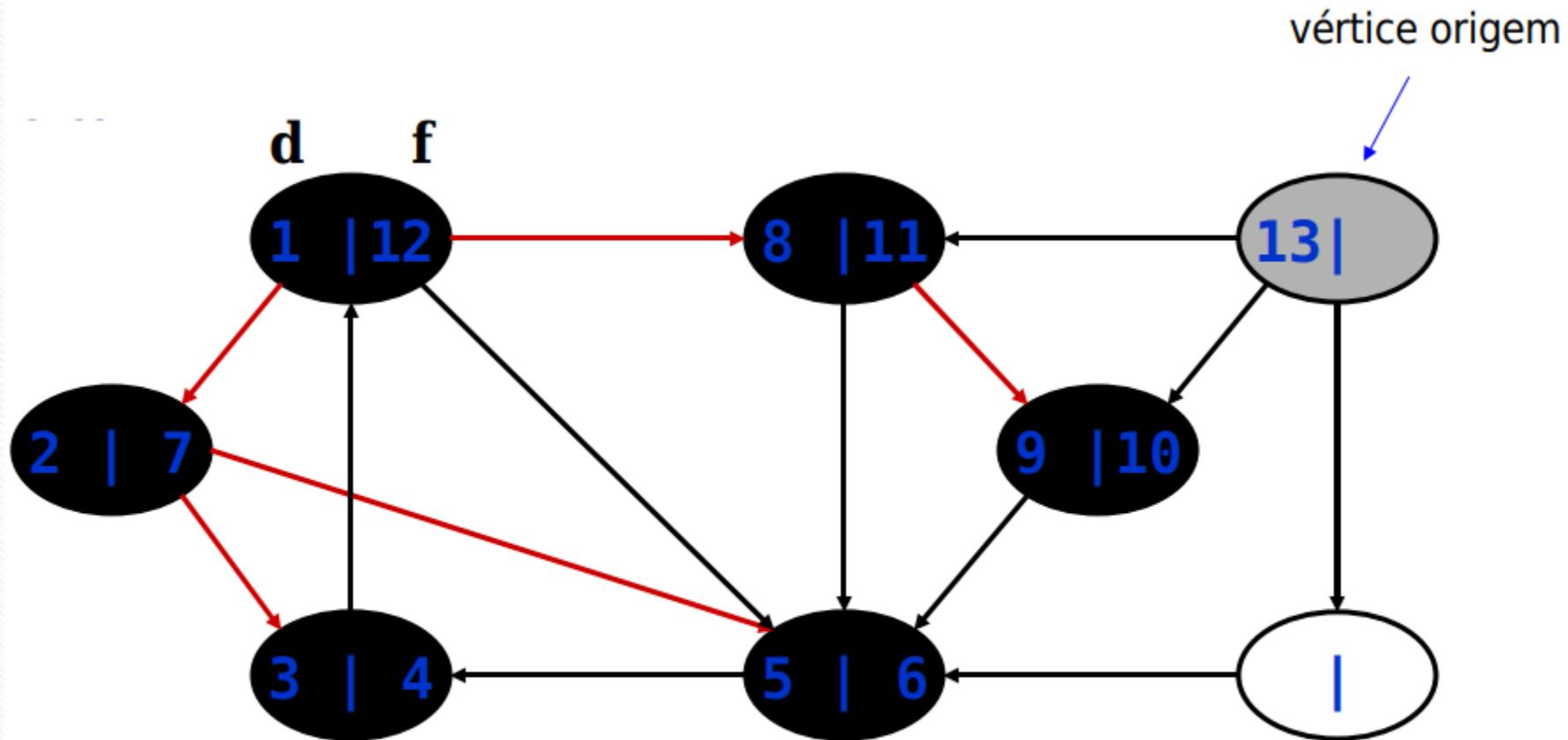
vértice origem



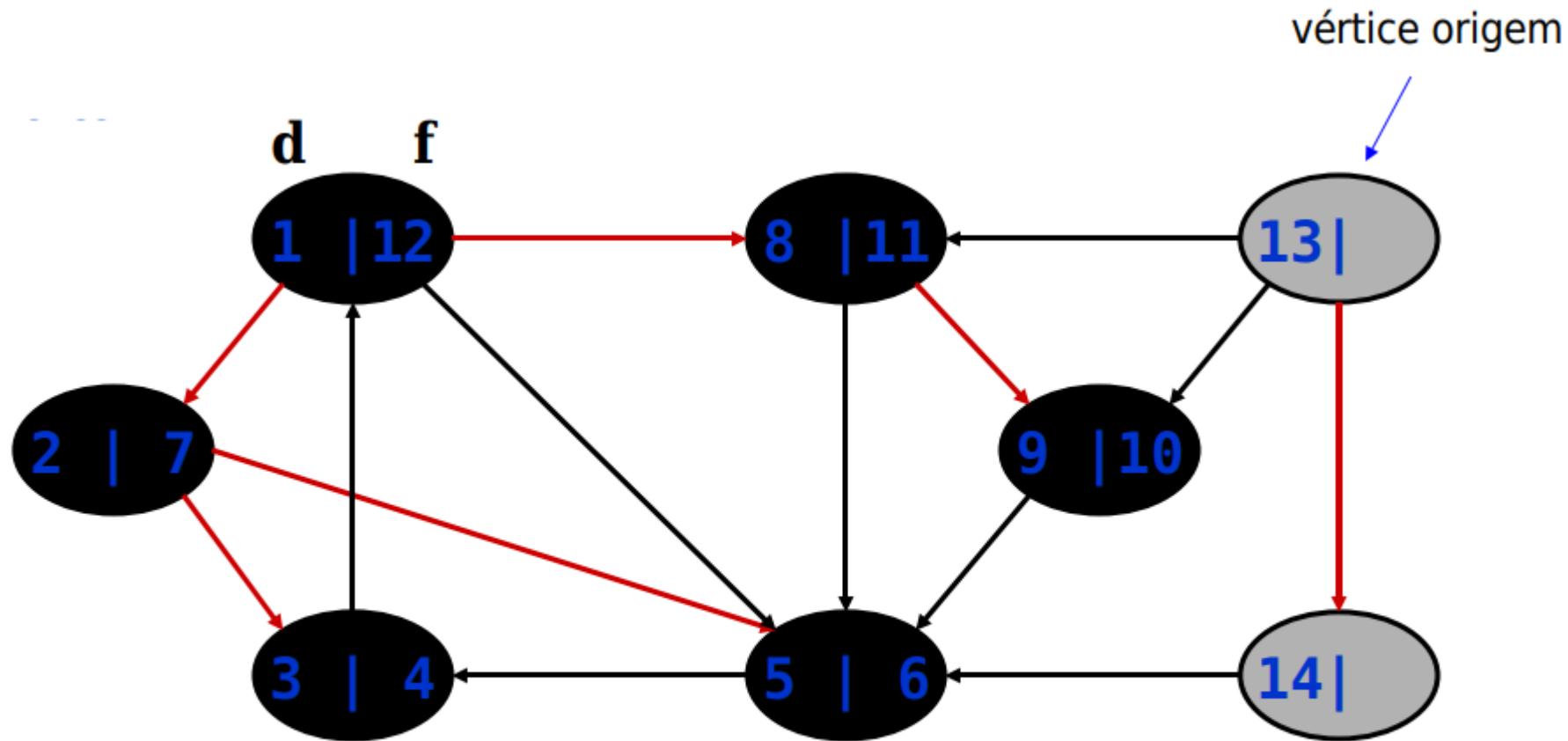
Busca em Profundidade



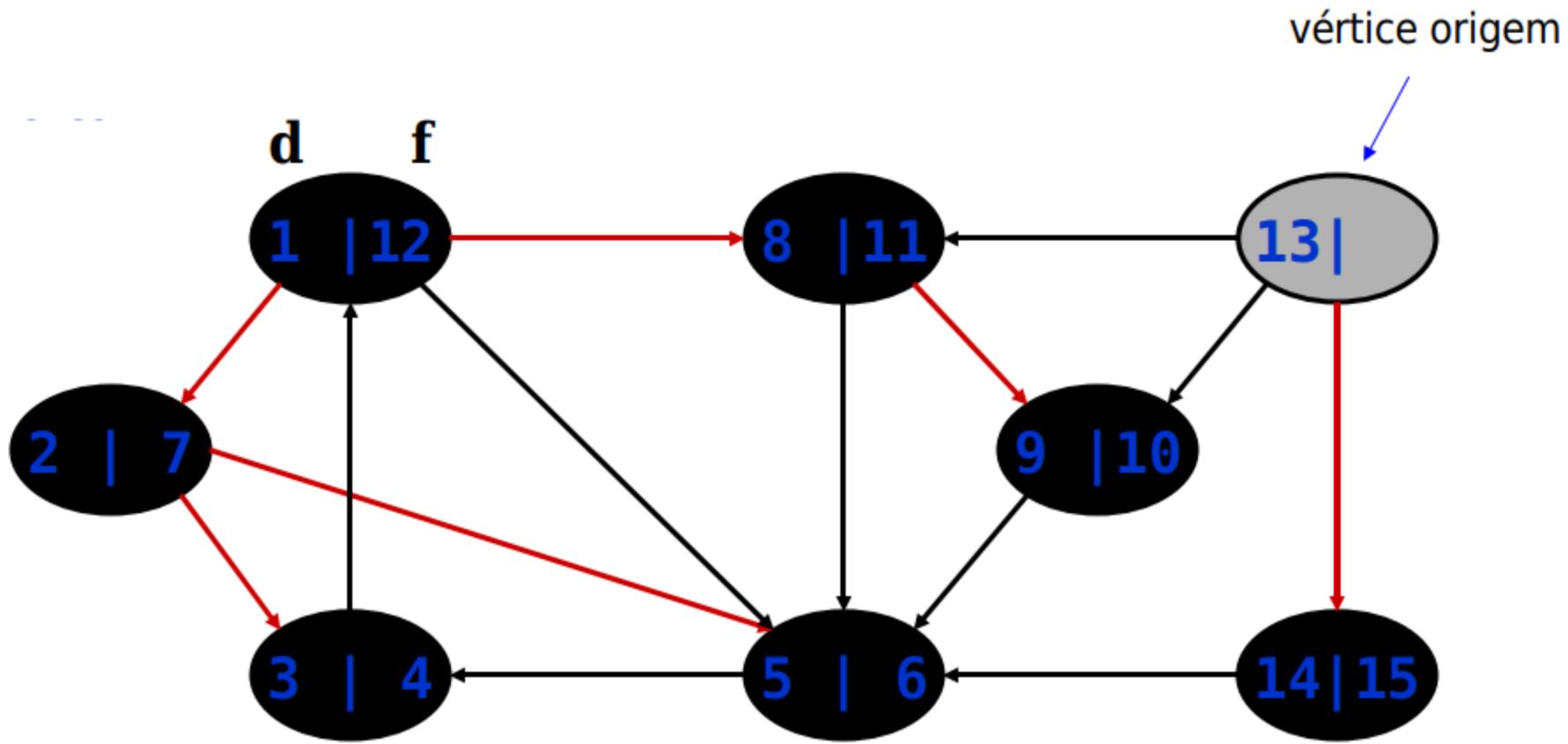
Busca em Profundidade



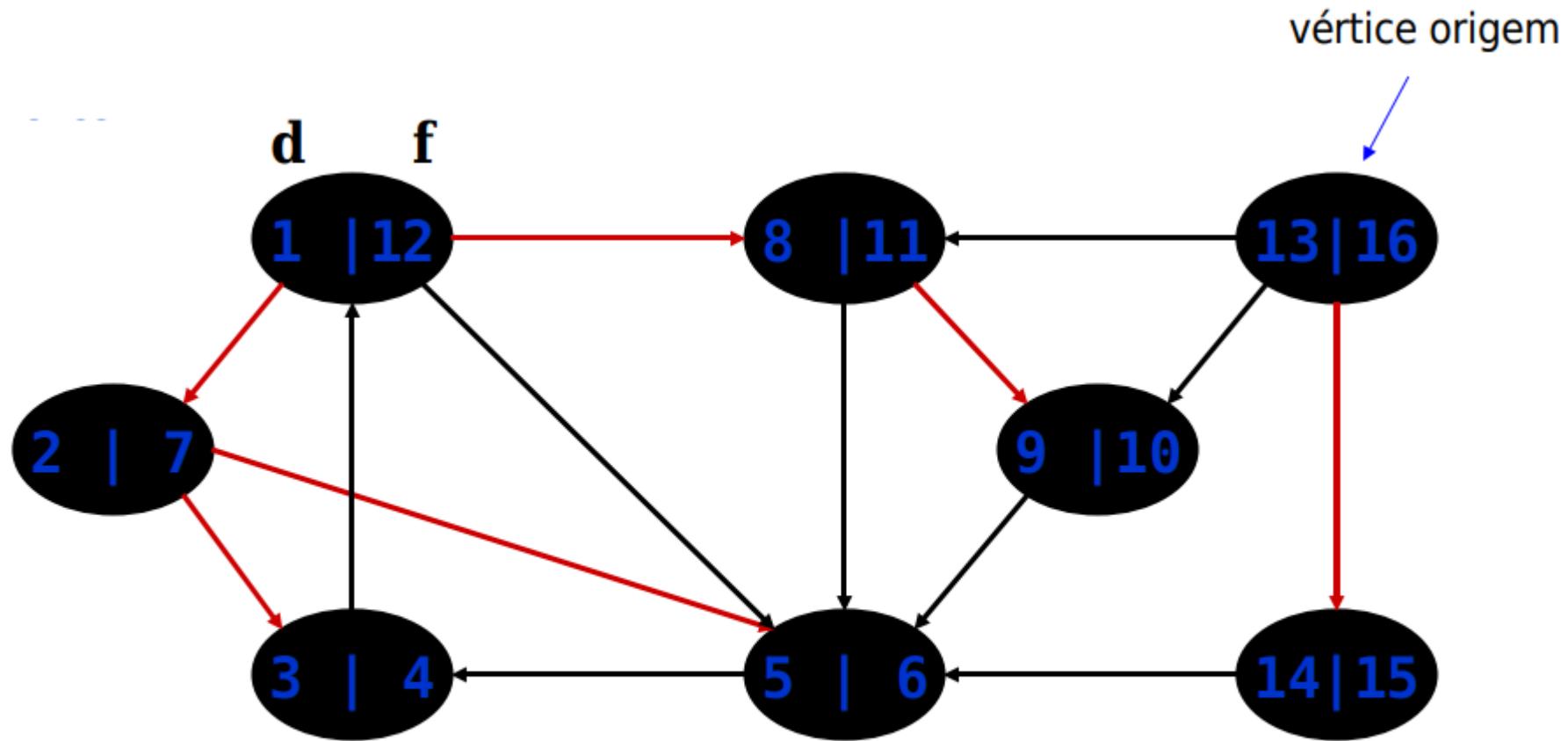
Busca em Profundidade



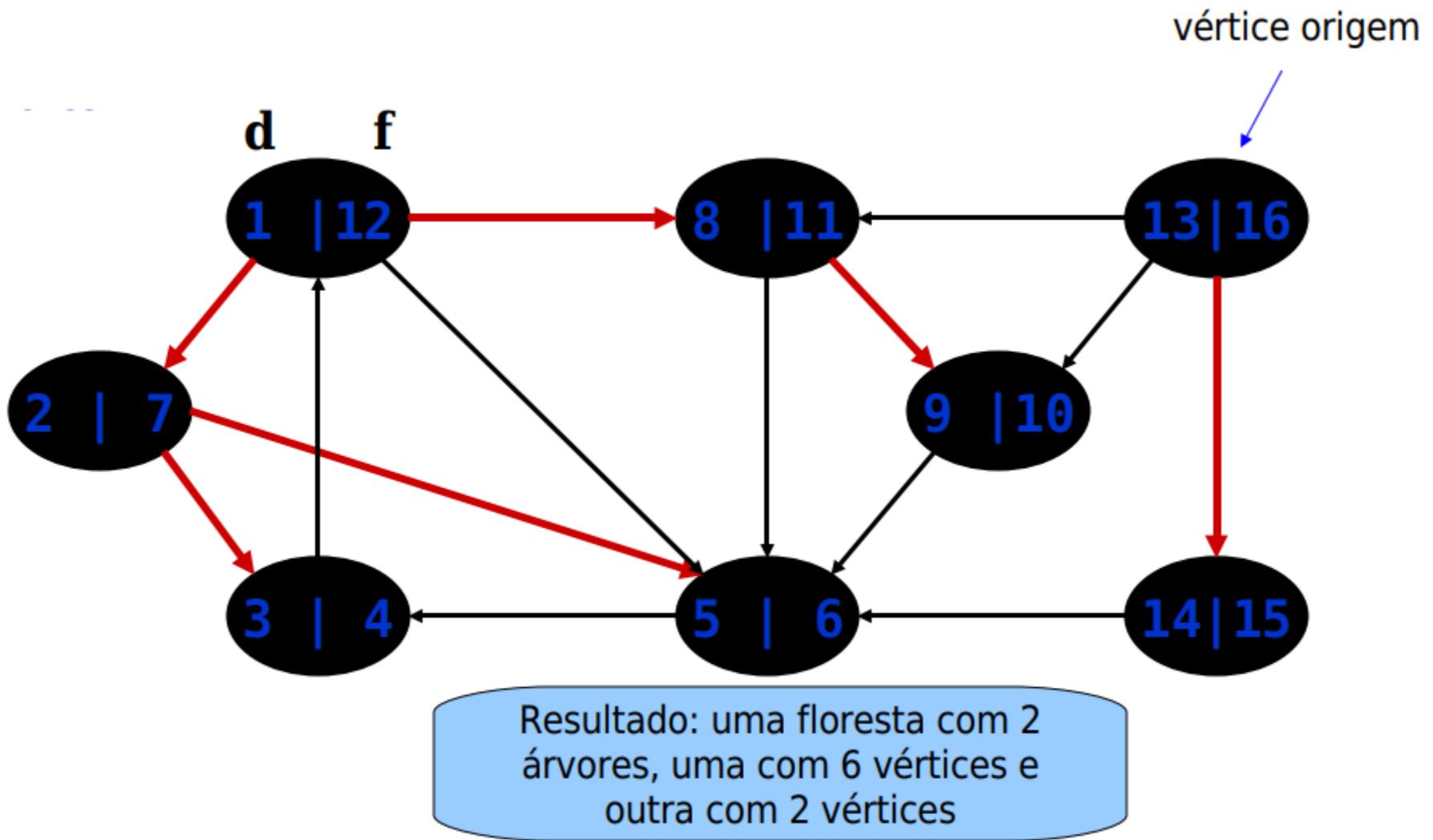
Busca em Profundidade



Busca em Profundidade

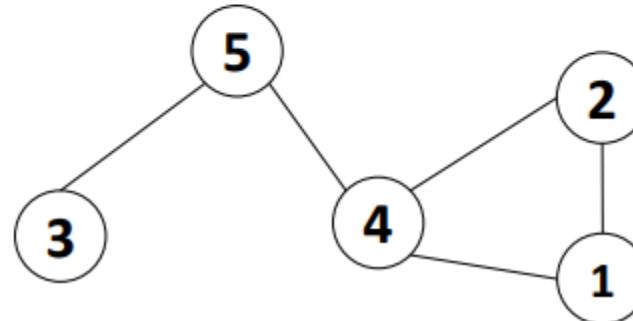


Busca em Profundidade



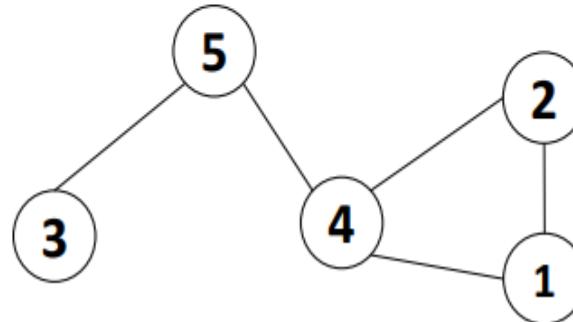
Busca em Profundidade - Implementação

```
private static int buscaEmProfundidade(No atual, No fim){  
    atual.visitado = true;  
    if (atual == fim) return 0;  
    for (No temp: atual.vizinhos){  
        if (!temp.visitado){  
            int res = buscaEmProfundidade(temp, fim) + 1;  
            if (res > 0) return res;  
        }  
    }  
    return -1;  
}
```

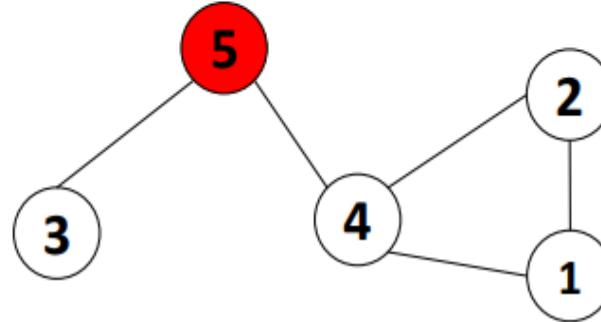


```
private static int buscaEmProfundidade(No atual, No fim){  
    atual.visitado = true;  
    if (atual == fim) return 0;  
    for (No temp: atual.vizinhos){  
        if (!temp.visitado){  
            int res = buscaEmProfundidade(temp, fim) + 1;  
            if (res > 0) return res;  
        }  
    }  
    return -1;  
}
```

buscando o nó 2



```
private static int buscaEmProfundidade(No atual, No fim){  
    atual.visitado = true;  
    if (atual == fim) return 0;  
    for (No temp: atual.vizinhos){  
        if (!temp.visitado){  
            int res = buscaEmProfundidade(temp, fim) + 1;  
            if (res > 0) return res;  
        }  
    }  
    return -1;  
}  
  
buscando o nó 2  
1. atual=5
```



```
private static int buscaEmProfundidade(No atual, No fim){  
    atual.visitado = true;  
    if (atual == fim) return 0;  
    for (No temp: atual.vizinhos){  
        if (!temp.visitado){  
            int res = buscaEmProfundidade(temp, fim) + 1; 1. atual=5  
            if (res > 0) return res;  
        }  
    }  
    return -1;  
}
```

buscando o nó 2

1. atual=5
1.1. atual=3

```
graph TD; 5((5)) --- 3((3)); 5 --- 4((4)); 4 --- 2((2)); 4 --- 1((1));
```

```

private static int buscaEmProfundidade(No atual, No fim){
    atual.visitado = true;
    if (atual == fim) return 0;
    for (No temp: atual.vizinhos){
        if (!temp.visitado){
            int res = buscaEmProfundidade(temp, fim) + 1; 1.1. atual=3
            if (res > 0) return res;
        }
    }
    return -1;
}

```

buscando o nó 2

1. atual=5
~~1.1. atual=3~~
 1.2. atual=4

```

private static int buscaEmProfundidade(No atual, No fim){
    atual.visitado = true;
    if (atual == fim) return 0;
    for (No temp: atual.vizinhos){
        if (!temp.visitado){
            int res = buscaEmProfundidade(temp, fim) + 1; 1.1. atual=3
            if (res > 0) return res;
        }
    }
    return -1;
}

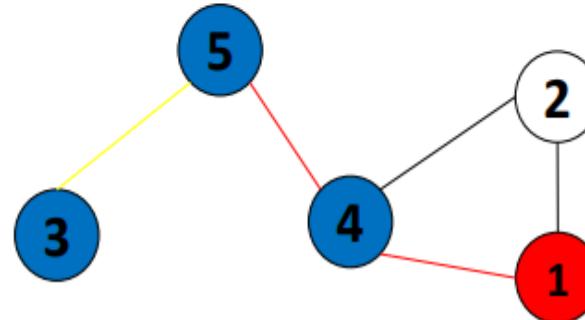
```

buscando o nó 2

1. atual=5

1.2. atual=4

1.2.1 atual=1



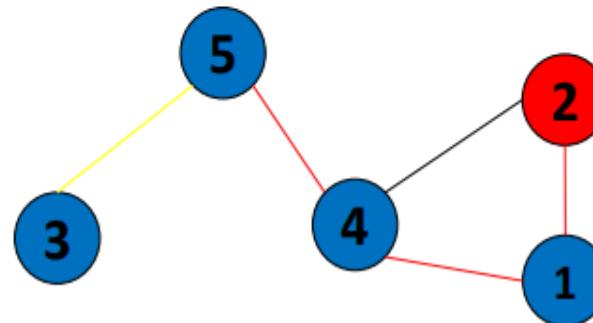
```

private static int buscaEmProfundidade(No atual, No fim){
    atual.visitado = true;
    if (atual == fim) return 0;
    for (No temp: atual.vizinhos){
        if (!temp.visitado){
            int res = buscaEmProfundidade(temp, fim) + 1;
            if (res > 0) return res;
        }
    }
    return -1;
}

```

buscando o nó 2

1. atual=5
- 1.1. atual=3
- 1.2. atual=4
- 1.2.1 atual=1
- 1.2.1.1 atual=2



```

private static int buscaEmProfundidade(No atual, No fim){
    atual.visitado = true;
    if (atual == fim) return 0;
    for (No temp: atual.vizinhos){
        if (!temp.visitado){
            int res = buscaEmProfundidade(temp, fim) + 1;
            if (res > 0) return res;
        }
    }
    return -1;
}

```

buscando o nó 2

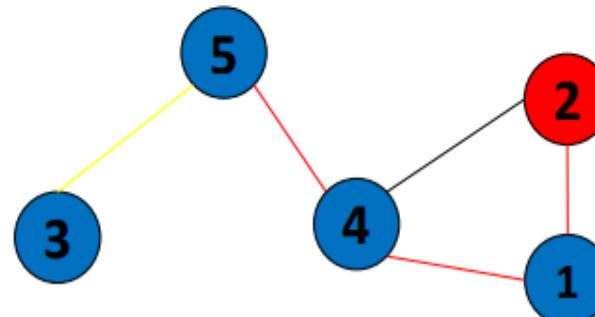
1. atual=5

1.1. ~~atual=3~~

1.2. atual=4

1.2.1 atual=1

1.2.1.1 atual=2



Distância = 3

Busca em Profundidade

- Característica: inicia-se a partir de um nó raiz e percorre cada caminho de forma a ir o mais longe possível antes de passar para outro caminho.
- A árvore Geradora é uma árvore com maior profundidade.
- **Poderá ser implementada com uma pilha.**

Problemas combinatórios

- São problemas em que uma solução é a combinação de um subconjunto de elementos;
- O espaço de busca de um problema combinatório é o conjunto de todas as soluções possíveis, podendo ser restrito as soluções viáveis ou não.

Problemas combinatórios

- As soluções de um problema combinatório são avaliadas de acordo com o objetivo a ser alcançado
- Que pode ser representado através de uma expressão matemática
- Tais expressões possuem como variáveis os elementos a serem combinados e que formam o espaço de busca;
- **Podemos desejar maximizar ou minimizar a função objetivo.**

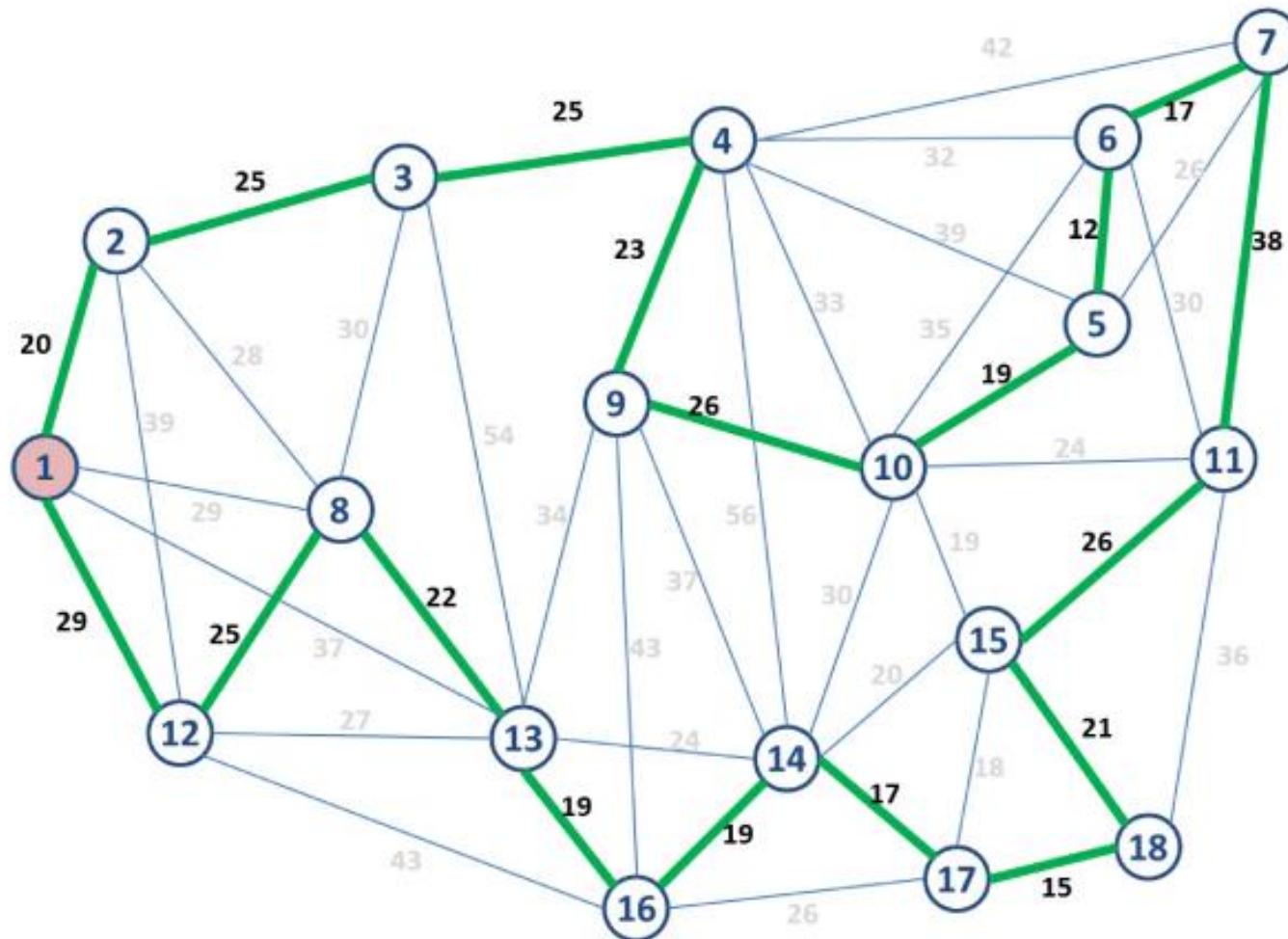
Exemplo – Problema do Caixeiro Viajante

- O Problema do Caixeiro Viajante (PCV) é um problema combinatório fundamentalmente matemático que tem perdurado por longo tempo. Em meados de 1800, problemas relacionados ao PCV começaram a ser desenvolvidos, mas somente em 1950 ficou conhecido globalmente em sua forma geral (APPLEGATE et al., 2007).

Exemplo – Problema do Caixeiro Viajante

- Define-se o Problema do Caixeiro Viajante como a busca de um circuito em um grafo, em que se partindo de um vértice inicial visita-se cada um dos outros vértices uma única vez, voltando ao ponto de partida, de forma que a distância do circuito percorrido seja mínimo (NILSSON, 1982).
- Denomina-se este circuito como ciclo ou circuito hamiltoniano.

Exemplo – Problema do Caixeiro Viajante



Exemplo – Problema do Caixeiro Viajante

- Considere o circuito a ser formado a partir de um grafo com n vértices. Partindo-se de determinado vértice, o próximo vértice a ser escolhido deve ser um dos $n-1$ vértices restantes do conjunto, dado que não haja repetição de vértice, exceto o primeiro e o último vértice caracterizando o circuito hamiltoniano.
- O seguinte a ser escolhido será um dos $n-2$ vértices ainda não escolhidos. Dessa forma, por um raciocínio indutivo, conclui-se que o conjunto de todas as rotas possíveis é descrita pela relação
- $(n - 1)(n - 2) \dots (2)(1) = (n - 1)!$

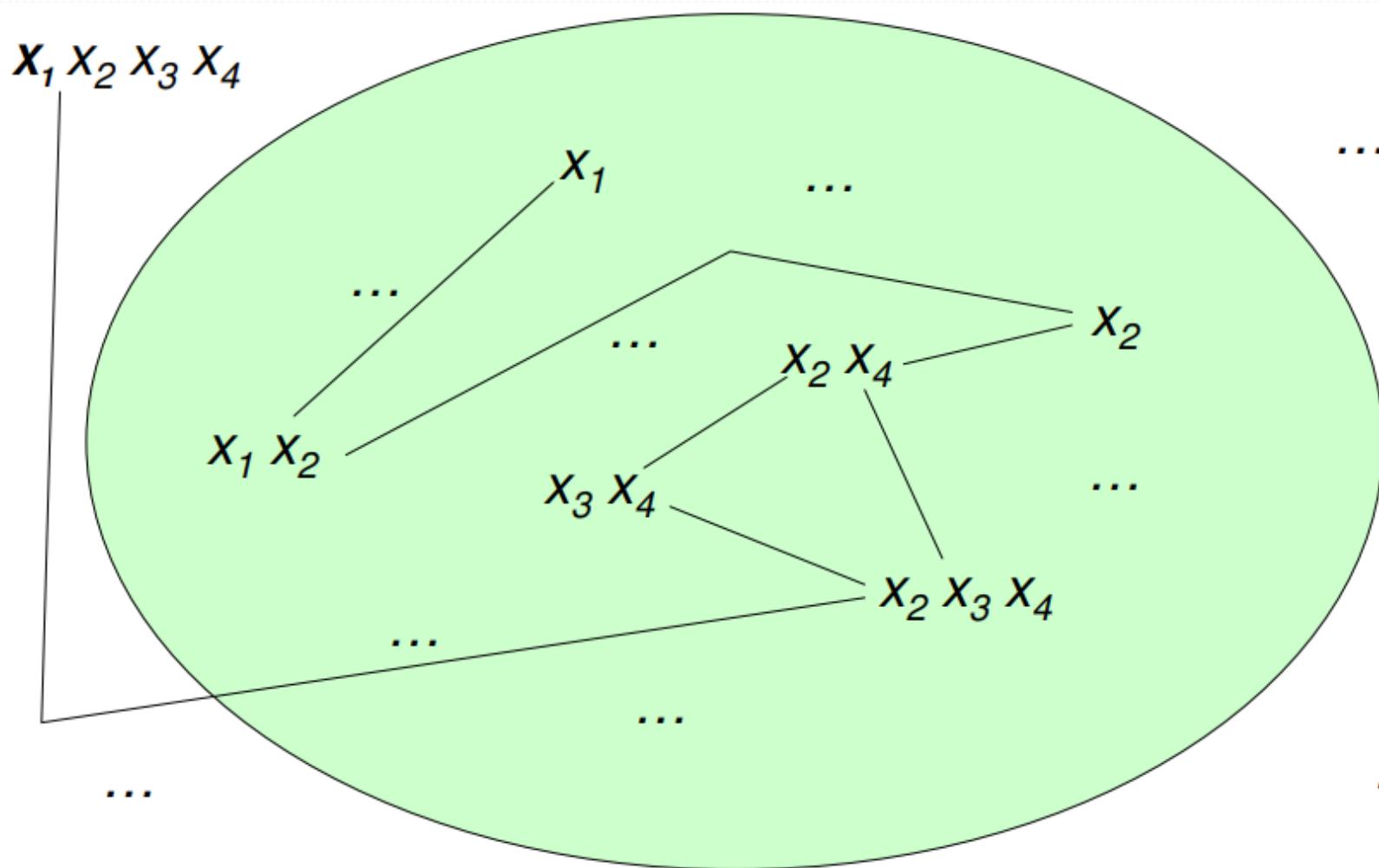
Soluções

- Uma solução ótima é uma solução viável que optimiza (maximiza ou minimiza) a função objetiva de um problema combinatório
- Ou seja, atinge o melhor valor para um problema.
- Podemos ter uma ou múltiplas soluções ótimas para um modelo, todas com o mesmo valor de avaliação da função objetivo

Soluções

- Ao explorarmos o espaço de busca utilizando alguma técnica, podemos realizar movimentos entre soluções
- Ou seja, a partir de uma solução atual, a alteramos de uma determinada maneira e chegamos a outra solução.
- Duas soluções que se diferem entre si por um movimento são ditas vizinhas.

Soluções



Ótimo local e Ótimo global

- Um ótimo global é uma solução considerada ótima **entre todas as soluções possíveis**
 - Ou seja, considerando-se todo o espaço de busca
- Um ótimo local é uma solução considerada ótima entre **todas as soluções de sua vizinhança**
 - Ou seja, não necessariamente um ótimo global, podendo ser considerada uma solução subótima.

Métodos Exatos e Heurísticos

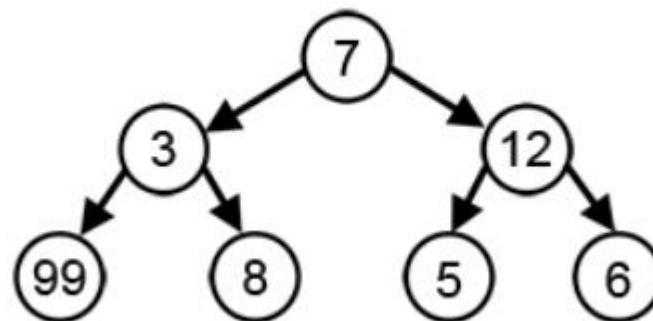
- Um **método exato (ou método ótimo)** para um problema combinatório é aquele que sempre gera uma **solução ótima global**
- Em casos de modelagem complexa, determinar uma solução ótima em tempo razoável pode ser até impossível com as ferramentas e técnicas disponíveis atualmente.
- Podemos utilizar também **métodos heurísticos (ou não exatos) para gerar boas soluções (geralmente subótimas)**

Algoritmos Gulosos

- Algoritmos Gulosos são algoritmos que funcionam em uma sequência de passos e se baseiam na melhor escolha a cada passo, de acordo com um conjunto de opções restrito
- Em outras palavras, escolhe o ótimo local a cada passo.
- Em algumas situações esta estratégia leva à soluções ótimas ou ótimos globais;
- No entanto, não existe garantia de bom desempenho em geral
- Em casos de problemas difíceis, algoritmos gulosos bem projetados produzem soluções boas, aceitáveis diante da dificuldade dos problemas.

Algoritmos Gulosos

- Algoritmos gulosos não reconsideram escolhas uma vez feitas
- Não realizam uma busca exaustiva;
- O que diferencia este método de backtracking, por exemplo
- Uma escolha errada e a solução ótima pode escapar
- Considere o problema do caminho de maior soma.



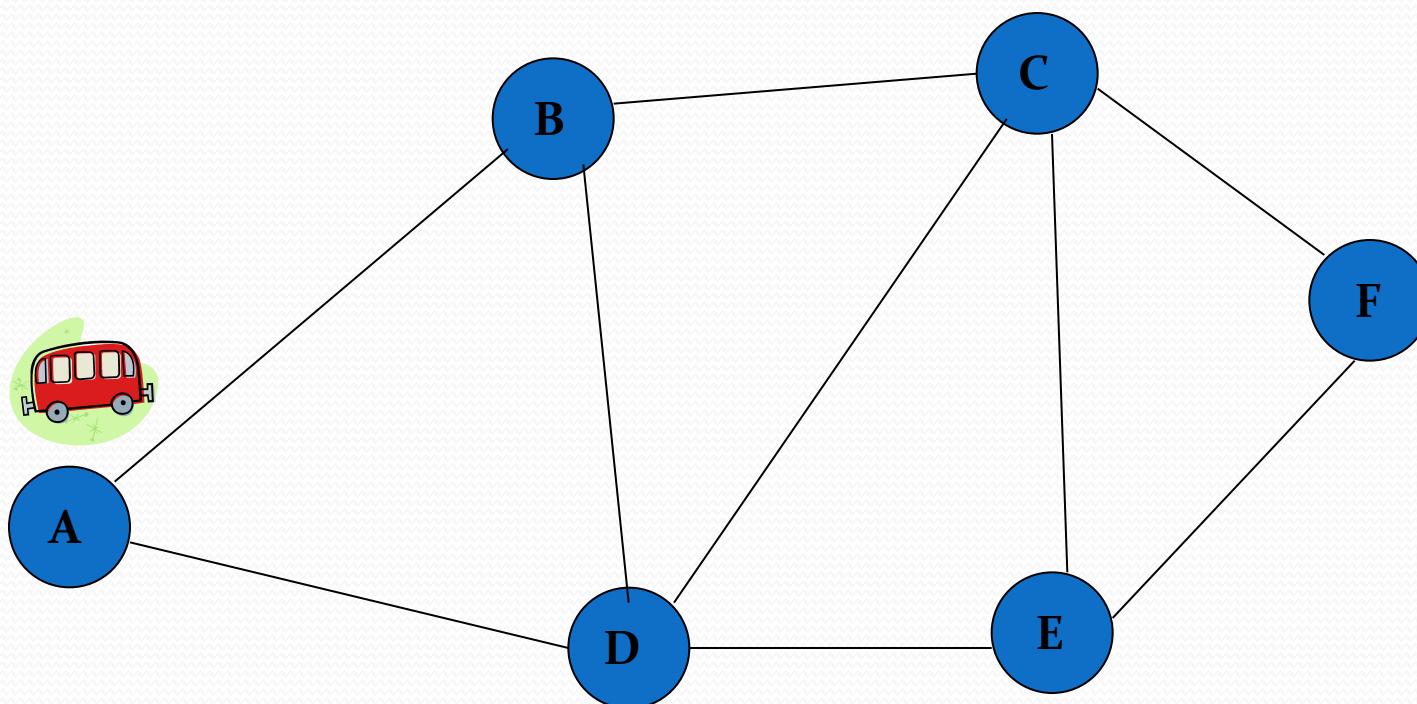
Algoritmos Gulosos

- Boa parte do projeto de um algoritmo guloso é devido à criatividade do projetista
- Novamente, é necessário enxergar o problema pela perspectiva correta, ter insights.
- Em geral, algoritmos gulosos possuem:
 - Fácil definição;
 - Fácil implementação;
 - Bom desempenho computacional
- Provar a corretude de um algoritmo guloso pode requerer provas matemáticas rigorosas e geralmente difíceis

Algoritmos Gulosos

- **Composição**
- Conjunto de opções : Do qual é escolhido o próximo passo.
- **Critério guloso de escolha**
- Como será realizada a escolha. É independente de escolhas passadas.
- **Função de viabilidade**
- Avalia quais escolhas são de fato viáveis.
- **Função objetivo**
- Determina o valor de cada escolha.
- **Função de solução**
- Determina se a solução corrente é completa

Imagine que uma transportadora decidiu alavancar as suas entregas e fez uma promoção de 50% em todas as entregas da cidade A até a cidade F. Para que ela não tenha prejuízo ela terá sempre que percorrer o menor caminho entre as cidades. **Como fazer isso?**



PROBLEMA DO MENOR CAMINHO

- **Problema:**
 - Obter *Caminhos* interligando *Vértices* de um *Grafo*, cujo comprimento (*Custo*) seja *Mínimo*.
- **Implementações:**
 - Algoritmo de Dijkstra
- **Aplicação:**
 - Redes de Computadores (*Percorso entre Roteadores*)
 - Tráfego Urbano.
 - Sistemas Rodoviários, Ferroviários e Aéreos,
 - Importante para vários outros problemas

Algoritmo de Dijkstra

- Problema dos caminhos mais curtos de origem (fonte) única: para um dado vértice, chamado fonte, em um grafo ponderado conectado, encontrar os caminhos mais curtos a todos os outros vértices
- Algoritmo de Dijkstra: melhor algoritmo conhecido aplicado a grafos com pesos não negativos

Algoritmo de Dijikstra

- Encontra os caminhos mais curtos de acordo com a distância a uma dada fonte
- Primeiro, encontra o caminho mais curto da fonte ao vértice mais próximo e assim por diante.
- Em geral, antes da i -ésima iteração iniciar, o algoritmo já encontrou os menores caminhos para os outros $i-1$ vértices próximos da fonte

Algoritmo de Dijkstra

- Estes vértices, a fonte, e as arestas dos caminhos mais curtos formam uma subárvore T_i .
- Como todos os pesos das arestas são não negativos, o vértice seguinte mais próximo da fonte pode ser encontrado dentre os vértices adjacentes aos vértices de T_i .
- Para identificar o i -ésimo vértice mais próximo, o algoritmo calcula, para cada vértice candidato u , a soma da distância ao vértice da árvore v mais próximo e o comprimento dv do caminho mais curto da fonte à v e então seleciona o vértice cuja soma seja a menor.

Algoritmo de Dijkstra

- Cada vértice possui duas etiquetas:
- Um valor numérico d que indica o comprimento do caminho mais curto da fonte ao vértice encontrado pelo algoritmo até o momento. Quando um vértice for adicionado a árvore, d indica o comprimento do caminho mais curto da fonte até o vértice.
- A outra etiqueta indica o nome do próximo ao último vértice neste caminho, i.e., o pai do vértice na árvore sendo construída.

Algoritmo de Dijkstra

- Com estas etiquetas, encontrar o vértice mais próximo u^* seguinte torna-se uma tarefa simples que consiste em encontrar o vértice candidato com o menor valor de d .
- Após identificado o vértice u^* a ser adicionado à árvore, as seguintes operações devem ser realizadas:
 - Mover o vértice u^* do vértice candidato para o conjunto dos vértices da árvore

Algoritmo de Dijkstra

- Para cada vértice candidato remanescente u que estiver conectado a u^* por uma aresta de pesos(u^*, u) tal que $du^* + w(u^*, u) < dw$, atualize as etiquetas de u por u^* e $du^* + w(u^*, u)$ respectivamente.
- Dijkstra compara comprimento de caminhos

Algoritmo de Dijkstra

- topologia da rede, custos dos enlaces conhecidos por todos os nós
 - realizado através de “difusão do estado dos enlaces”
 - todos os nós têm mesma info.
- calcula caminhos de menor custo de um nó (“origem”) para todos os demais
 - gera **tabela de rotas** para aquele nó
- iterativo: depois de k iterações, sabemos menor custo p/ k destinos

Notação:

- $c(i,j)$: custo do enlace do nó i ao nó j. custo é infinito se não forem vizinhos diretos
- $D(V)$: valor corrente do custo do caminho da origem ao destino V
- $p(V)$: nó antecessor no caminho da origem ao nó V
- N : conjunto de nós cujo caminho de menor custo já foi determinado

O algoritmo de Dijkstra

1 *Inicialização:*

2 $N = \{A\}$

3 para todos os nós V

4 se V for adjacente ao nó A

5 então $D(V) = c(A,V)$

6 senão $D(V) = \text{infinito}$

7

8 *Repete*

9 determina W não contido em N tal que $D(W)$ é o mínimo

10 adiciona W ao conjunto N

11 atualiza $D(V)$ para todo V adjacente ao nó W e ainda não em N :

12 $D(V) = \min(D(V), D(W) + c(W,V))$

13 /* novo custo ao nó V ou é o custo velho a V ou o custo do

14 menor caminho ao nó W , mais o custo de W a V */

15 *até que todos nós estejam em N*

Áreas de aplicação

- Resolução de problemas complexos
- Redes de computadores
- Inteligência Artificial (Algoritmos Genéticos)
- Mineração de Dados

Contatos

- Email: fabio.silva321@fatec.sp.gov.br
- Linkedin: <https://br.linkedin.com/in/b41a5269>
- Facebook: <https://www.facebook.com/fabio.silva.56211>