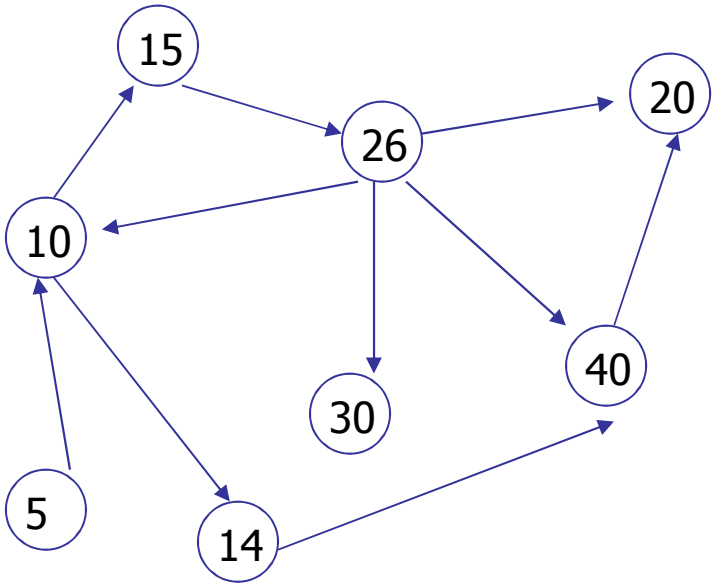
Grafos

Universidade Federal do Amazonas
Departamento de Eletrônica e Computação

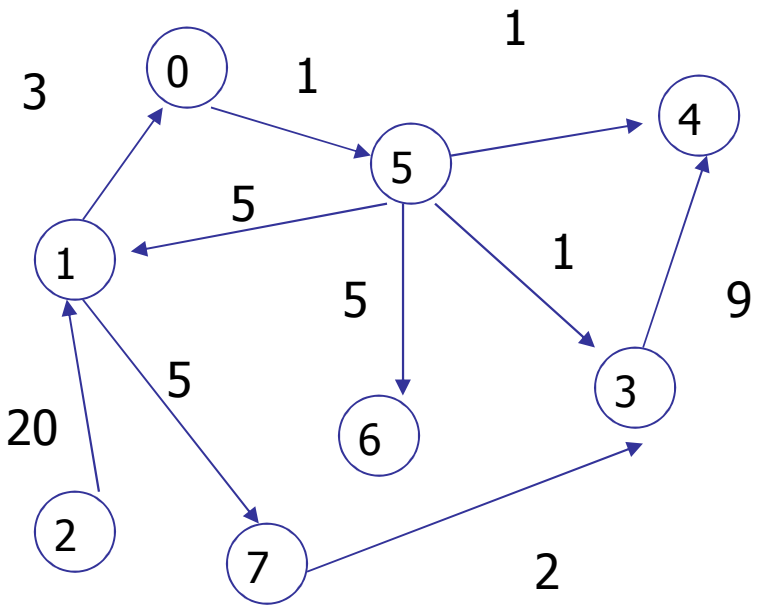
Grafos (1)

- Um grafo é composto por um conjunto de **vértices** e um conjunto de **arestas**
- Cada aresta liga dois vértices do grafo



Grafos (2)

- Arestas e vértices podem ter atributos de acordo com a aplicação:

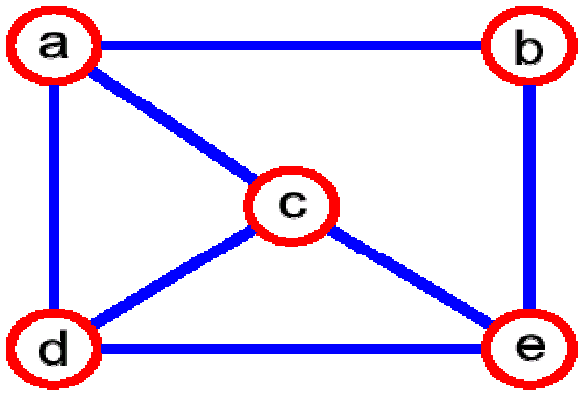


Definição Formal

- **Um grafo G = (V,E)** é composto por:
  - V: um conjunto de **vértices** ou **nodos**
  - $E \subseteq V \times V$: conjunto de **arcos** ou **arestas** conectando os vertices
- Um **arco e** = (u,v) é um par de vértices
- Se (u,v) é ordenado, **G** é um grafo **dirigido**

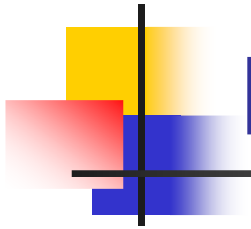
(u,v) é ordenado se *u* é designado como primeiro elemento e *v* como segundo elemento

Dois pares ordenados *(a,b)* e *(d,e)* são iguais se, e somente se, *a=d* e *b=e*



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

E= {(a,b),(a,c),(a,d), (b,e),(c,d),(c,e), (d,e)}

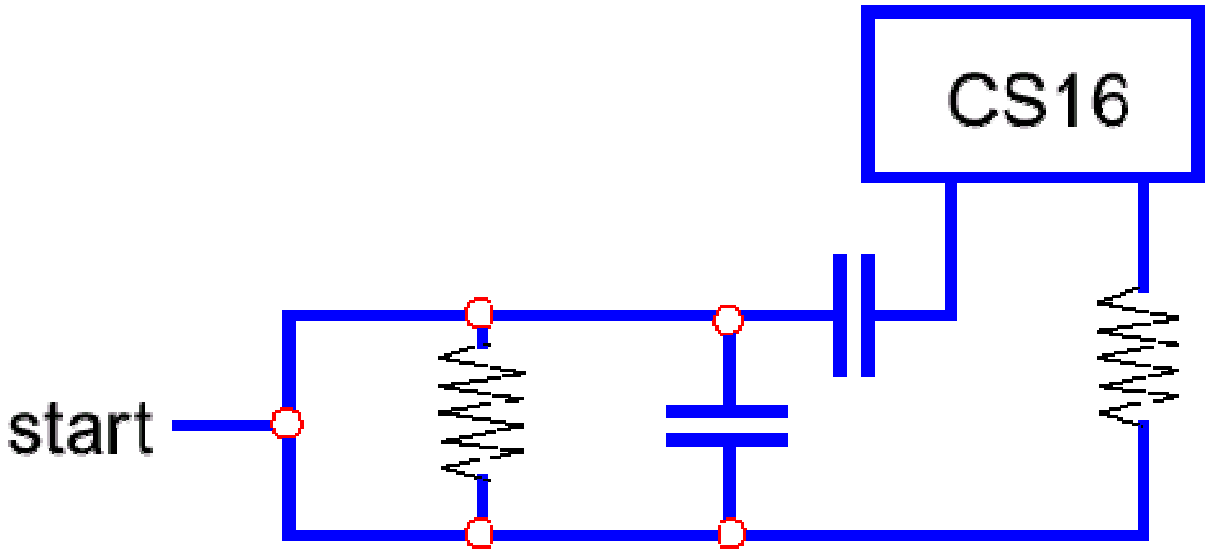


Exemplos de aplicações

- Representação de conjuntos de pontos geográficos (tais como ruas, cidades, estados e circuitos) e das ligações entre estes conjuntos
- Representação de redes de computadores
- Representação de programas computacionais
- Representação da Web

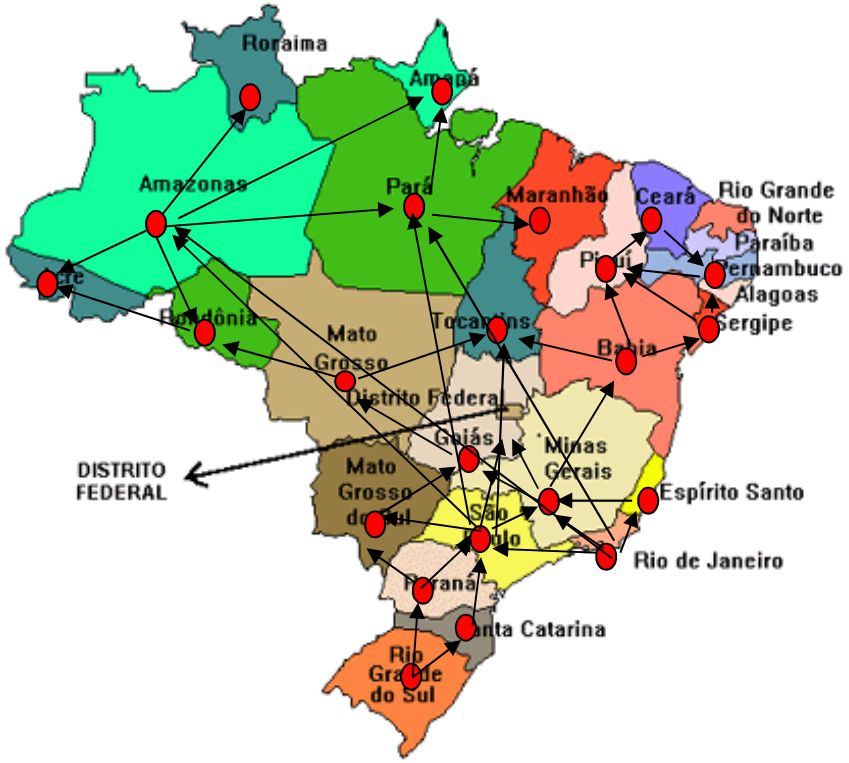
Aplicações (1)

- Circuitos eletrônicos:
  - Qual o caminho com menor resistência para CS16 ?



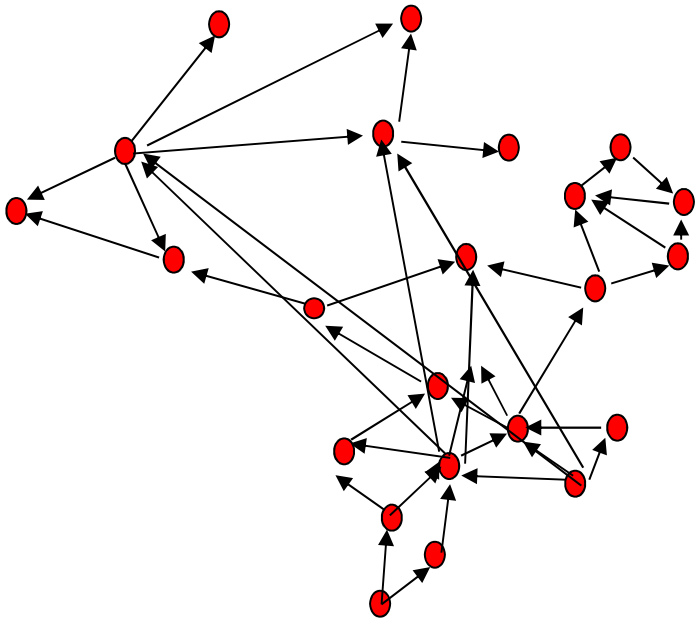
Aplicações (2)

- Transportes
  - Qual a rota de POA até MAO que maximiza custo/benefício



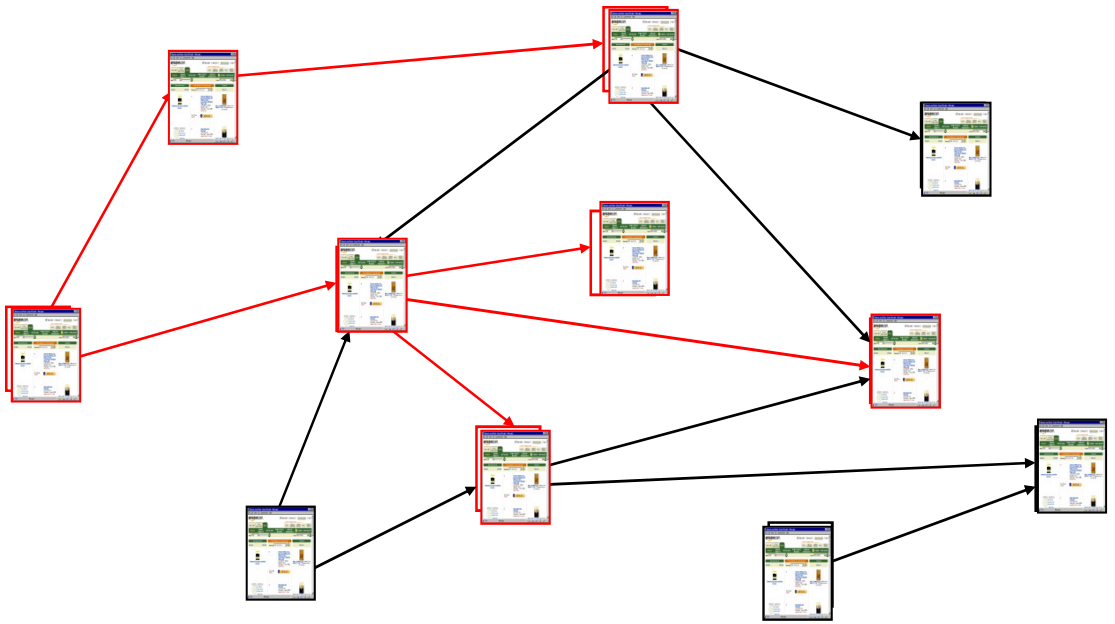
Aplicações (3)

- Transportes
  - Qual a rota de POA até MAO que maximiza custo/benefício



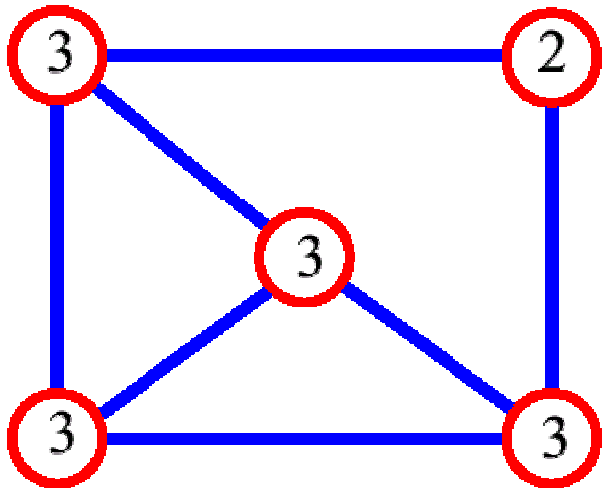
Aplicações (4)

- A Word-Wide Web
  - Um robô deve coletar as páginas de interesse para os usuários de uma máquina de busca no menor tempo possível



Grafos – Terminologia (1)

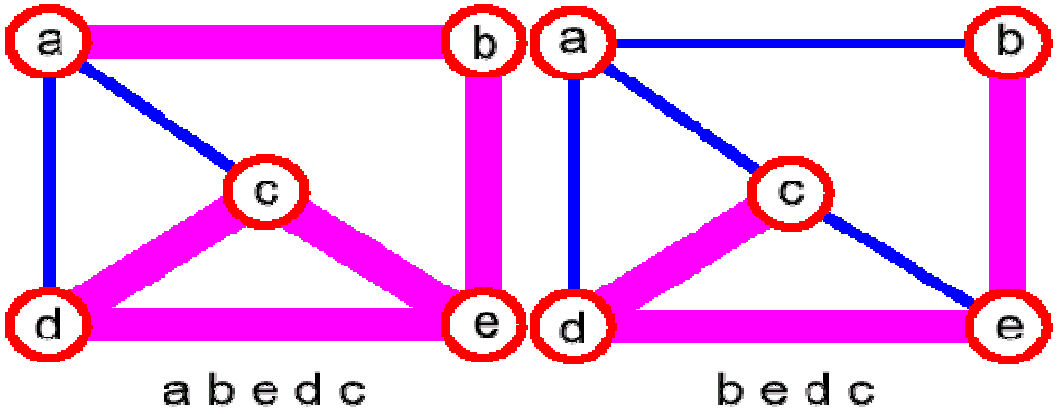
- Vértices adjacentes: conectados por um arco
- grau de um vértice: número de vertices adjacentes a ele



- caminho: sequência de vértices v_1 ,..., v_k tais que todo par de vértices consecutivos v_i e v_{i+1} são adjacentes

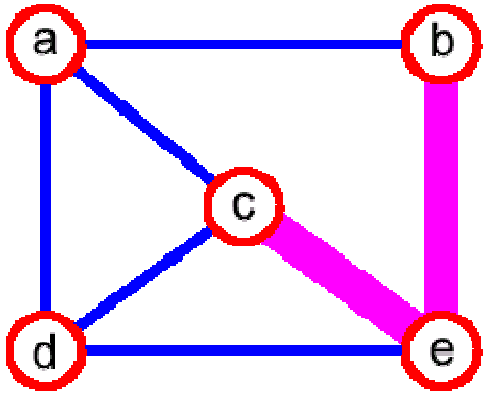
Grafos – Terminologia (2)

- Caminho simples: sem vértices repetidos



a b e d c

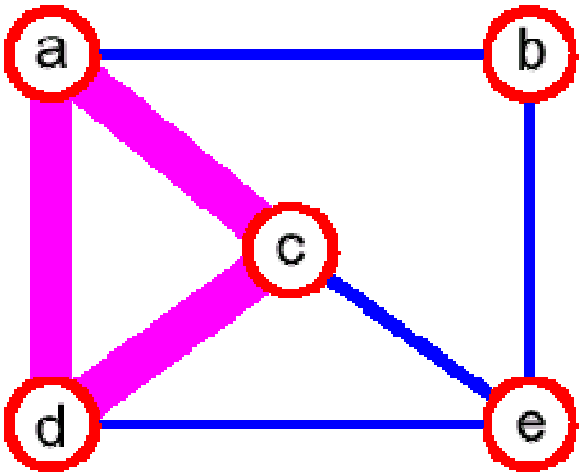
b e d c



b e c

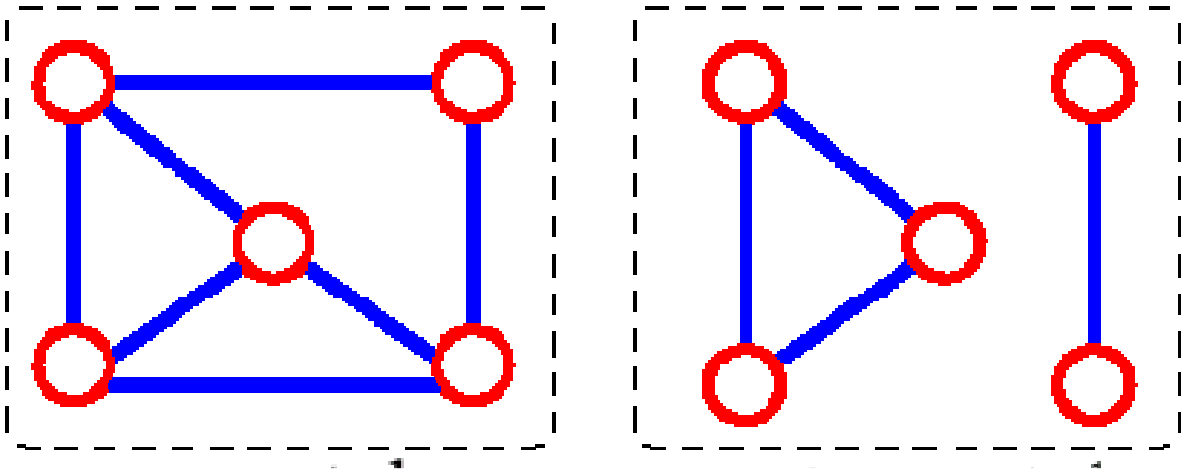
Grafos – Terminologia (3)

- Ciclo: caminho simples, exceto pelo fato de que o primeiro vértice se repete como adjacente ao último vértice



Grafos – Terminologia (4)

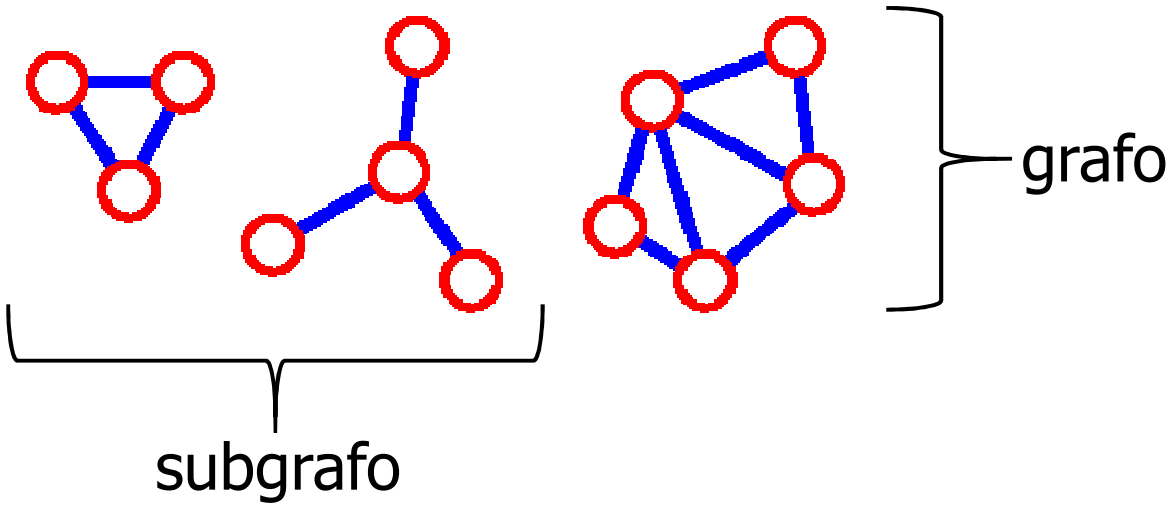
- Grafo conexo: existe um caminho entre qualquer par de vértices no grafo



Conexo Não-conexo

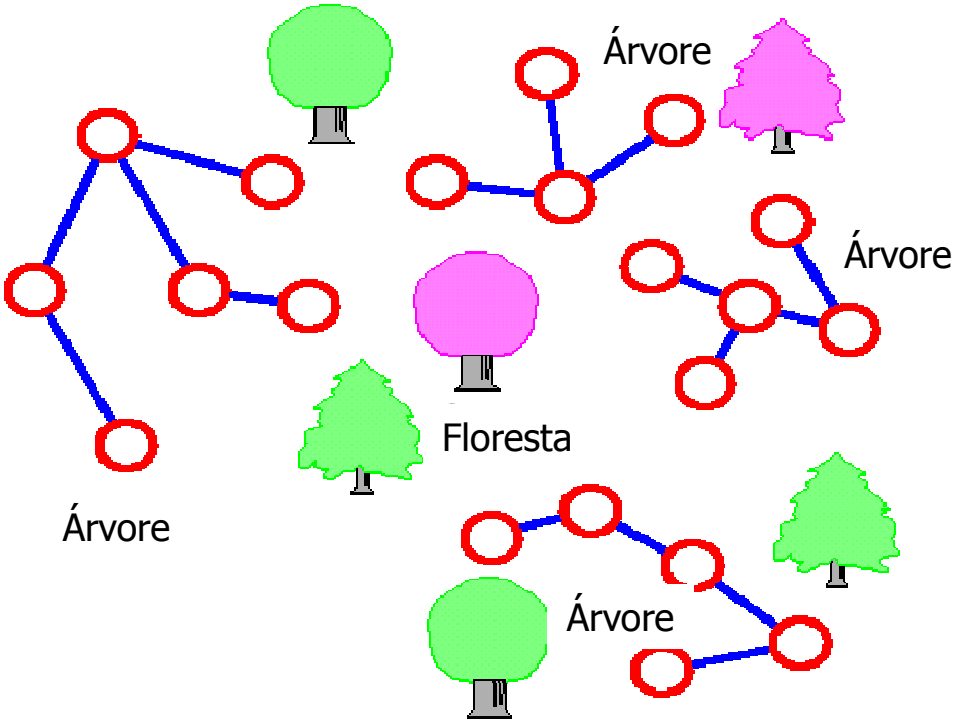
Grafos – Terminologia (5)

- Subgrafo: subconjunto dos vértices e arcos que formam um grafo
- Componentes conexos: subgrafos que são conexos e que deixam de ser conexos se acrescentarmos qualquer vértice novo



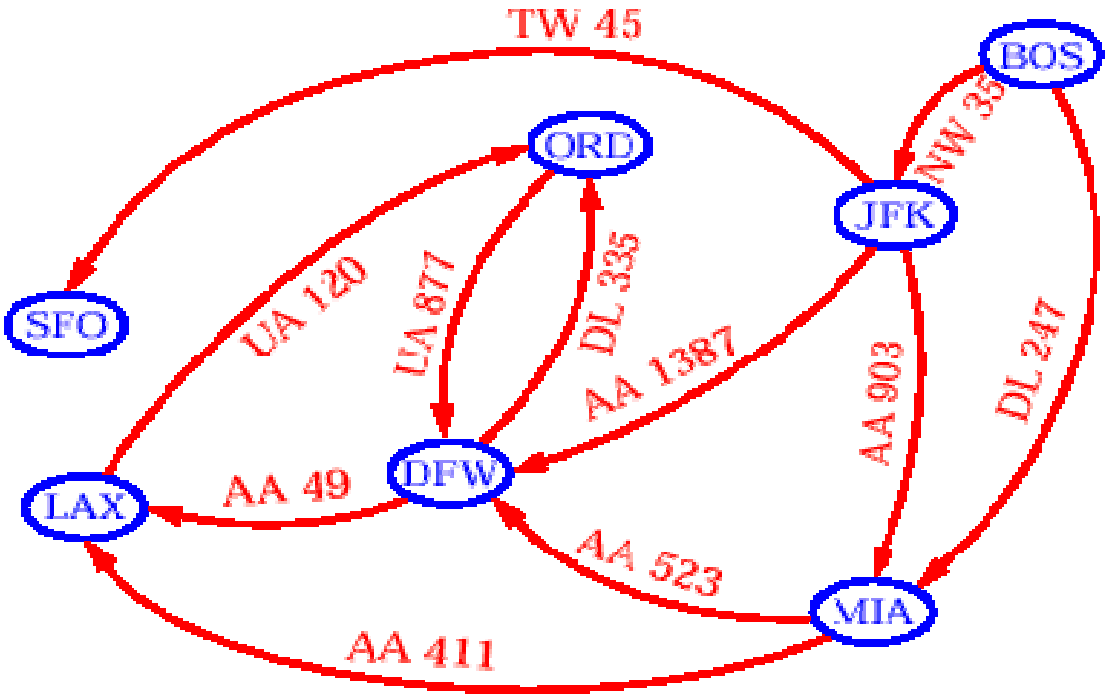
Grafos – Terminologia (6)

- Árvore – grafo conexo sem ciclos
- Floresta – coleção de árvores



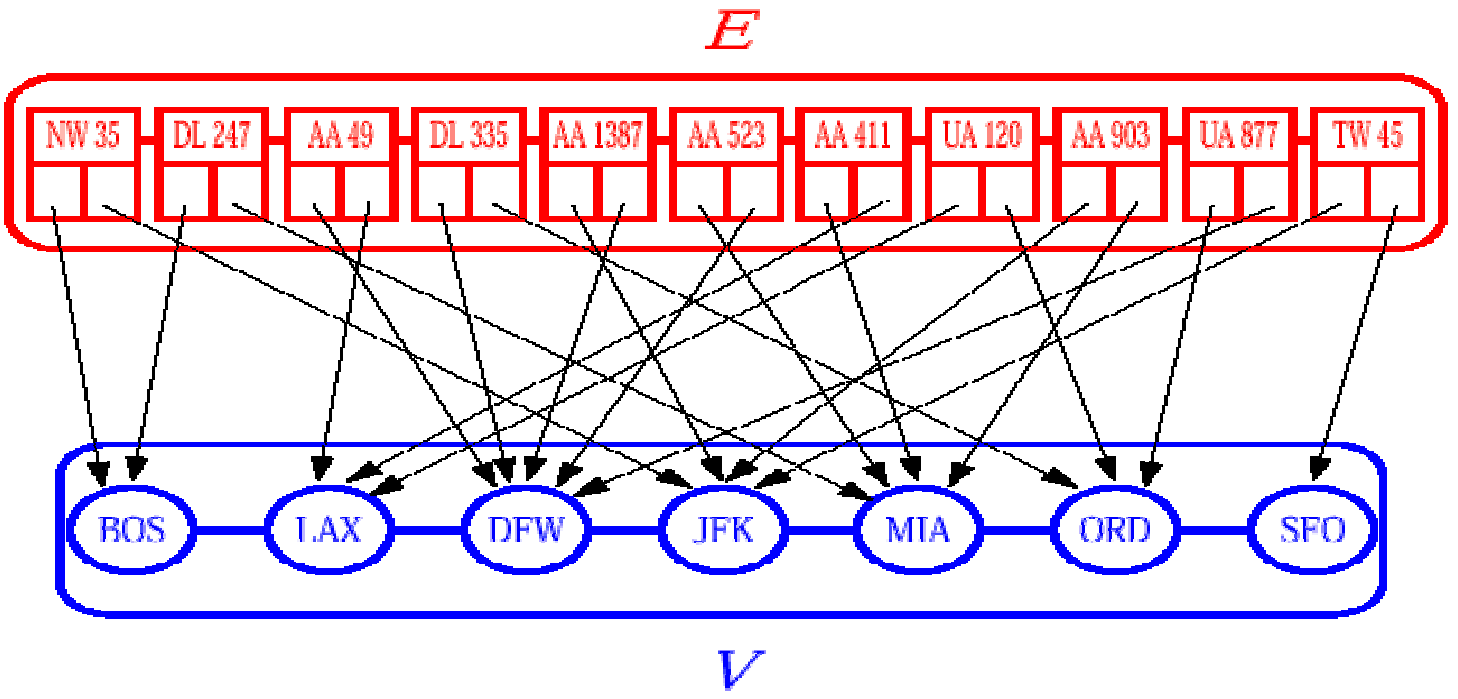
Representação de Grafos

- Ideia Básica
  - Dois tipos de objetos: **vértices e arcos**
  - Aos vértices (e arcos) associam-se rótulos



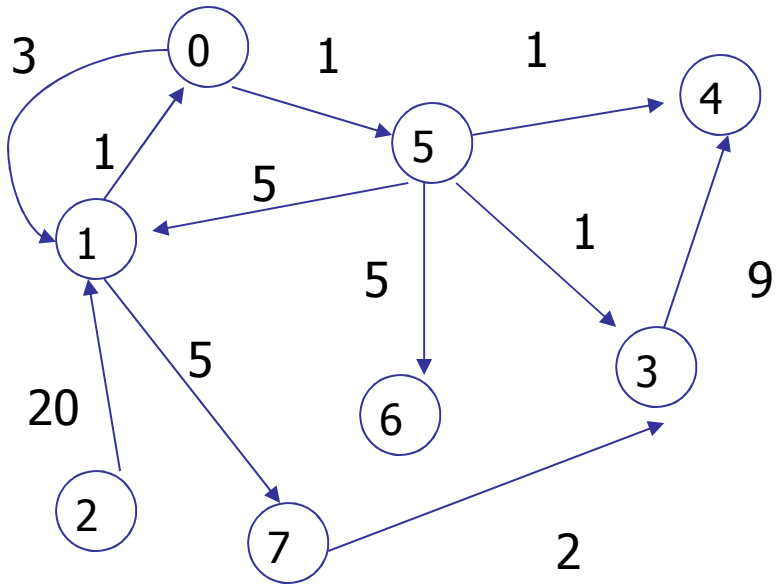
Lista de Arcos

- Fácil de implementar
- Encontrar os arcos que incidem em um vértice exige percorrer toda a lista



Matriz de Adjacências (1)

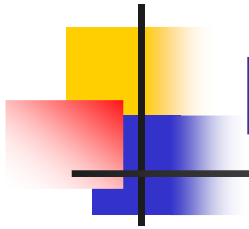
- A representação de um grafo G=(V,E) consiste em uma matriz |V| × |V|



Vértices de destino

Vértices de origem

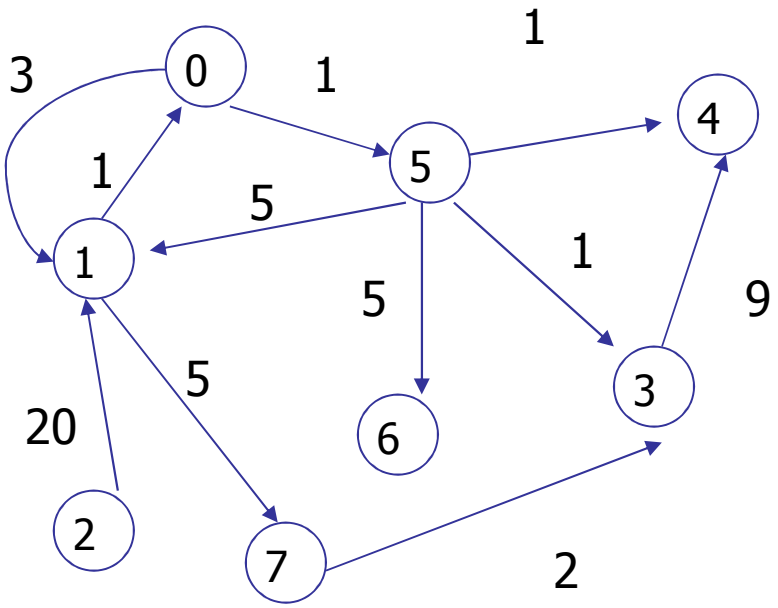
	0	1	2	3	4	5	6	7
0		3				1		
1	1							5
2		20						
3					9			
4								
5		5		1	1		5	
6								
7				2				

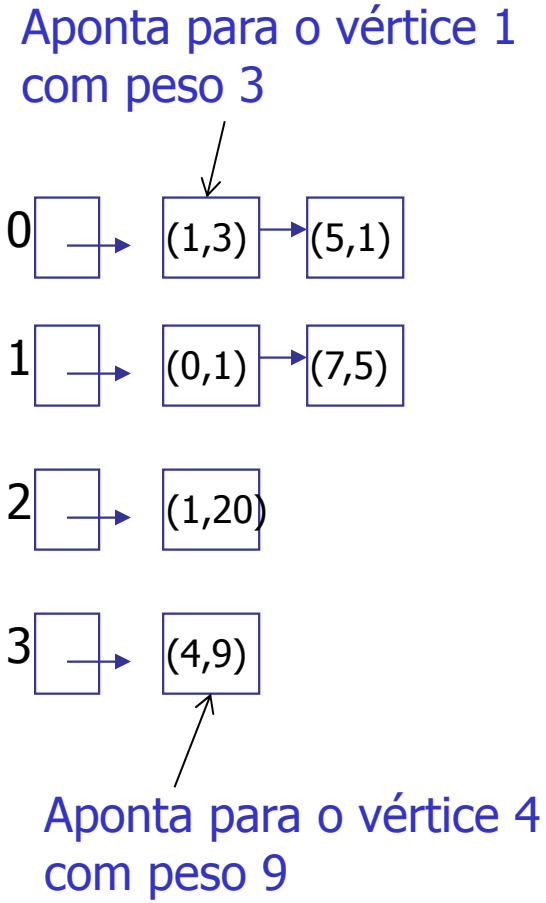


Matriz de Adjacências (2)

- Matrizes podem ocupar muito espaço sem necessidade
- O espaço total é $O(V^2)$
- A solução nestes casos pode ser trocar a forma de representação

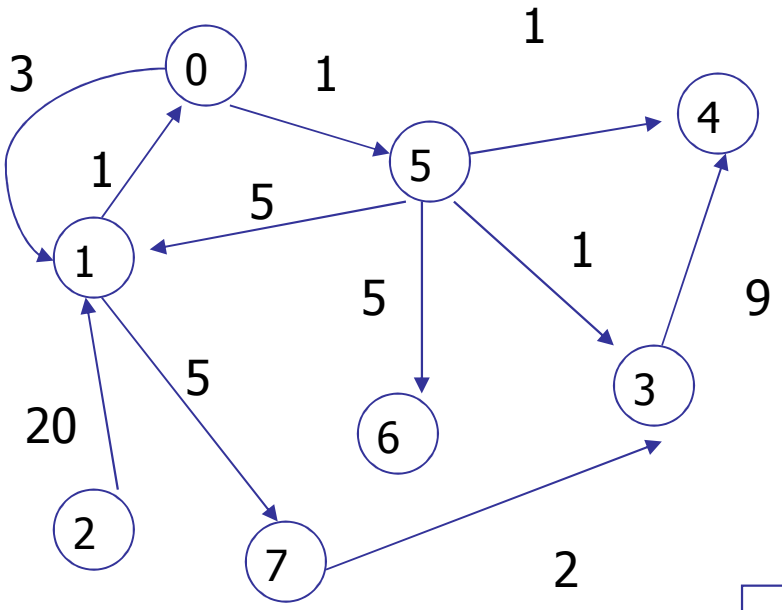
Listas de Adjacências



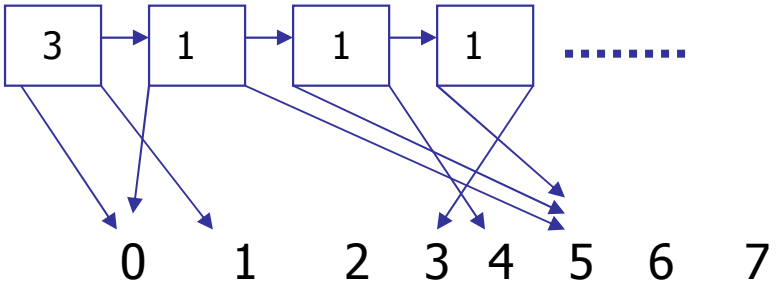


Espaço é proporcional ao número de arestas

Listas de Arestas



Os nodos da lista representam as arestas do grafo



Representação da Lista em C

- Os campos **key** *e* **next** representam o conteúdo a ser armazenado e o ponteiro para o próximo nodo da lista

```
typedef struct node {
  int key;
  struct node *next;
} NODE;
```

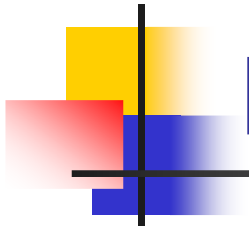
A estrutura da lista como um todo é representada por um ponteiro para o início (*myelem*head*)

Inserção na Lista em C

- Qual é o tempo de execução do *insert_node* sobre uma lista de *n* elementos?

  O(1)

```
void insert_node(NODE *l){
  if (head==NULL)
    l->next = NULL;
  else
    l->next = head;
  head = l;
}
```

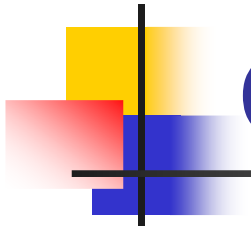


Remoção da Lista em C

- Qual é o tempo de execução do *delete_node* no pior caso?
  O(n)

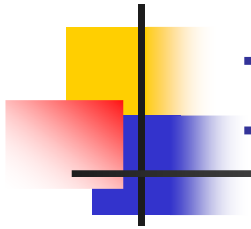
```
void delete_node(NODE *l){
  NODE *tmp; tmp = head;
  if (head != l) {
    while(tmp->next!=l)
      tmp = tmp->next;
    tmp->next = l->next;
  } else { head = l->next; }
  free(l);
}
```

Caminhamento em Grafos



Caminhamento em grafos

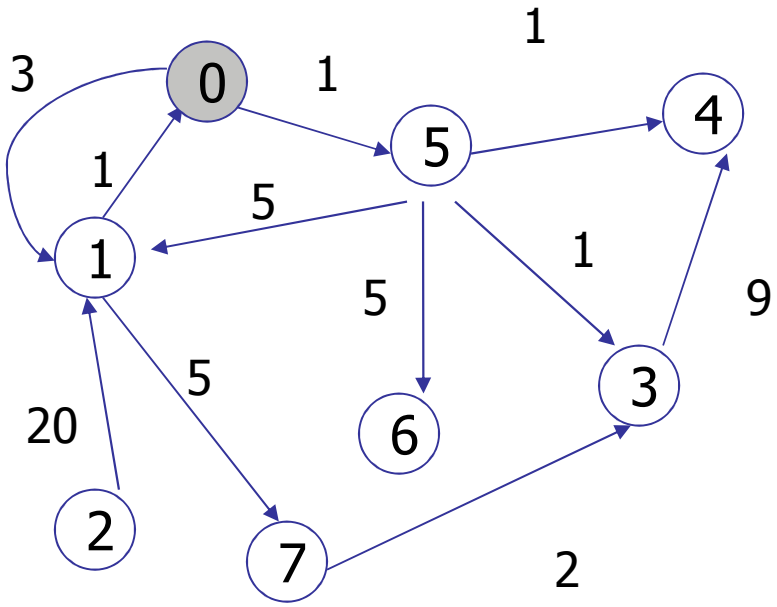
- Algoritmos de caminhamento determinam a ordem com que os vértices de um grafo serão visitados
- Pode ser feito em **largura** ou em **profundidade**
  - *Breadth-first search (BFS)*
  - *Depth-first search (DFS)*



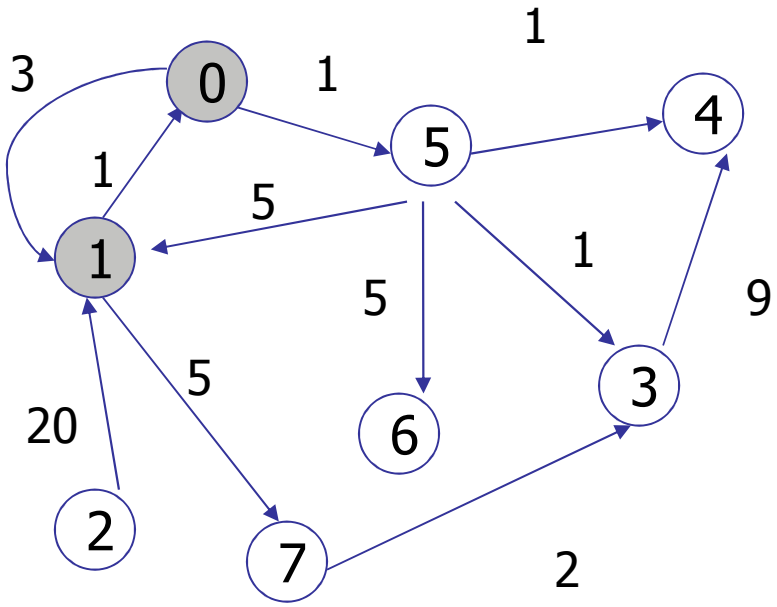
Ideia Básica do BFS (1)

- Caminha(v)
- Marcar v como visitado
- Visitar todos os adjacentes, guardando seus filhos em uma fila
- Para cada elemento da fila, marcar elemento como visitado e guardar todos os seus adjacentes em uma fila
- Proseguir operação até que não haja mais elementos a visitar

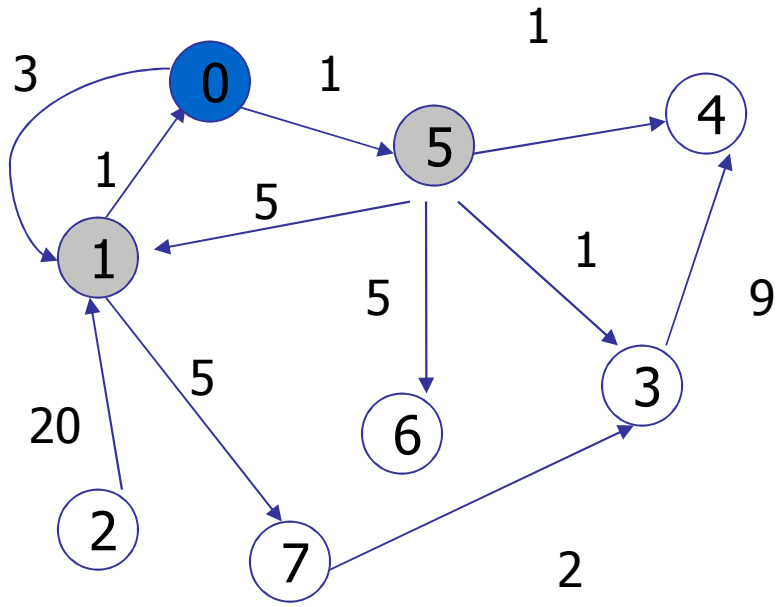
Ideia Básica do BFS (2)



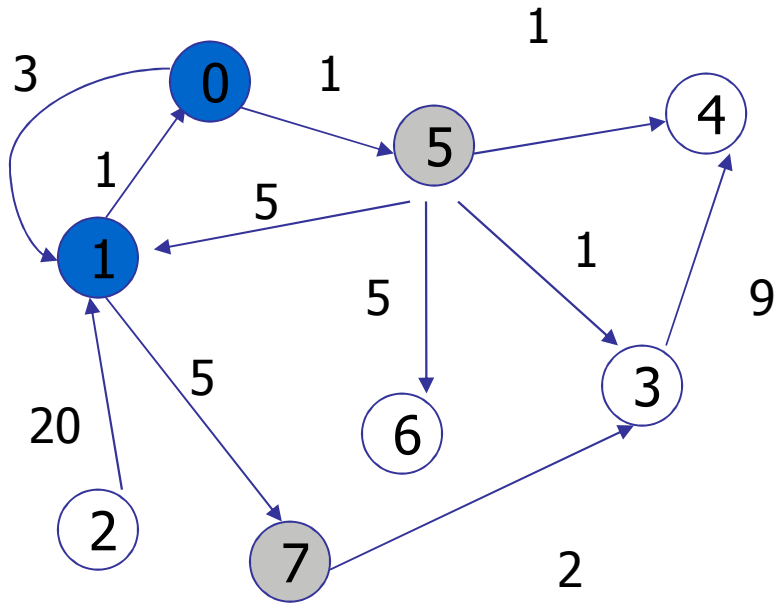
Ideia Básica do BFS (3)



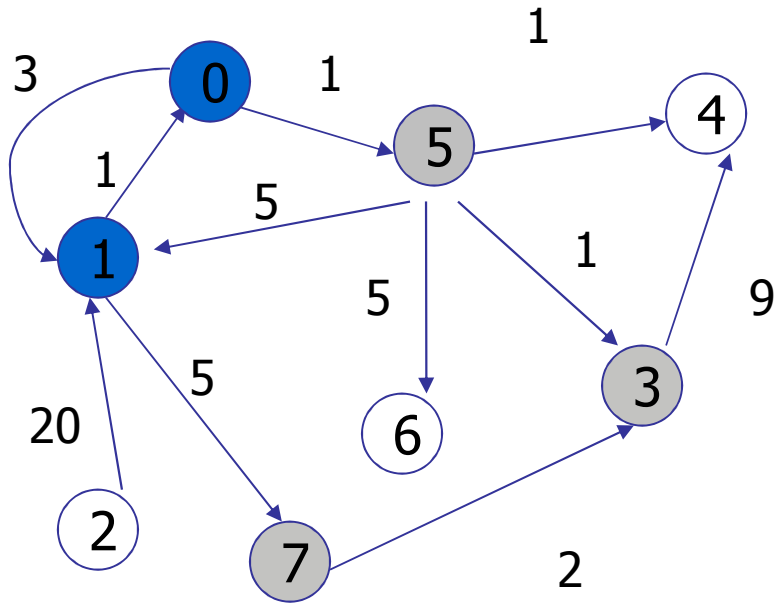
Ideia Básica do BFS (4)



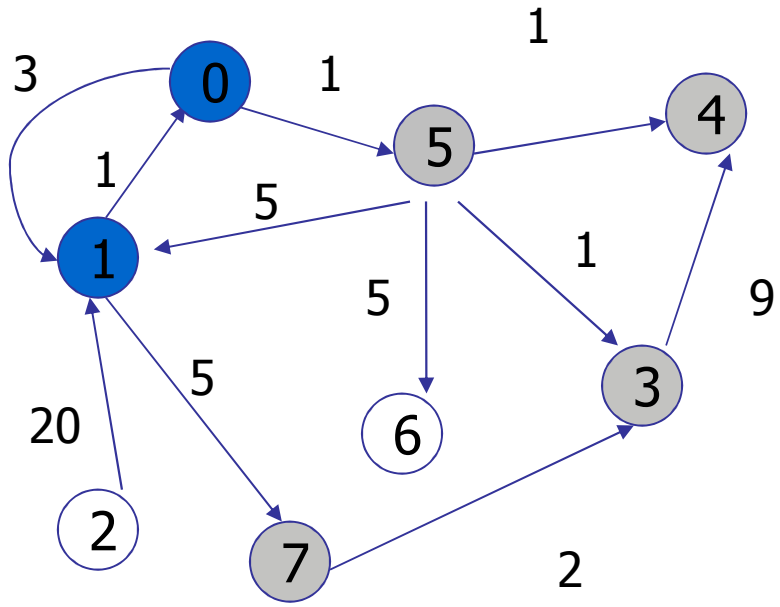
Ideia Básica do BFS (5)



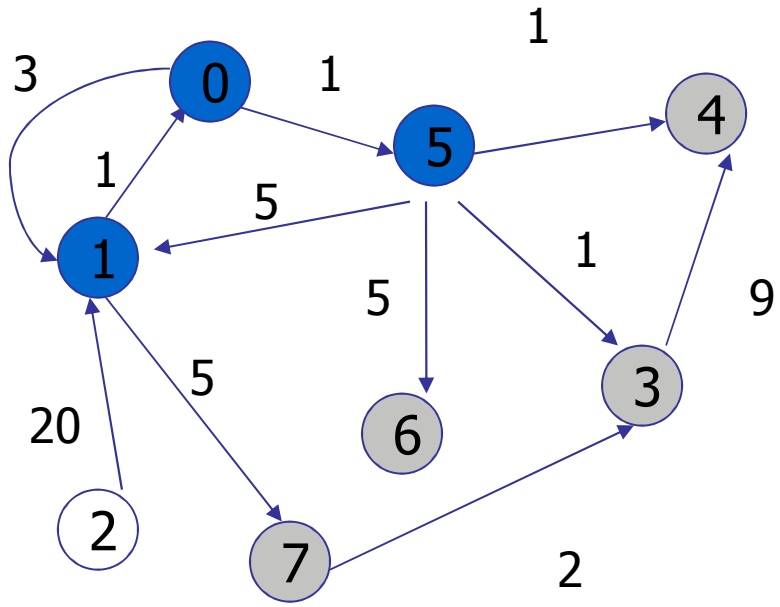
Ideia Básica do BFS (6)



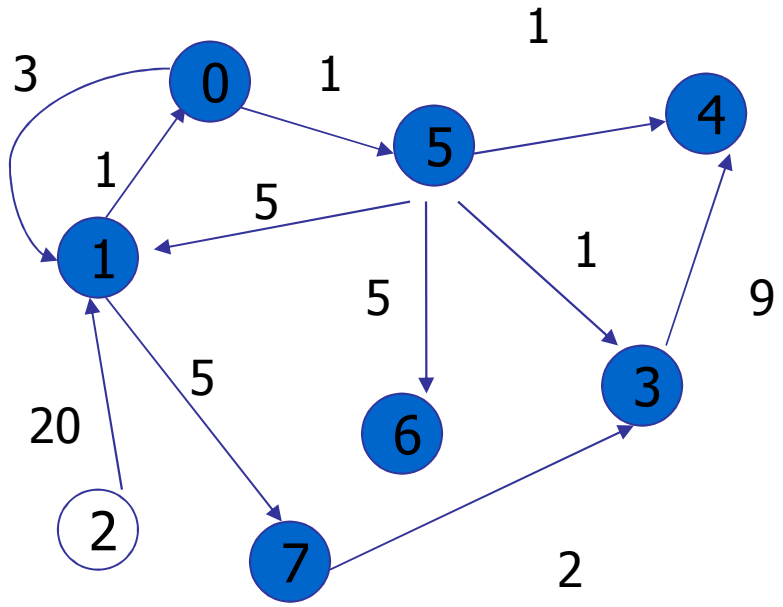
Ideia Básica do BFS (7)

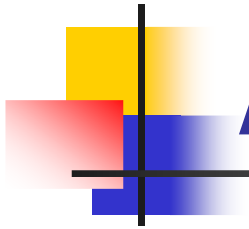


Ideia Básica do BFS (8)



Ideia Básica do BFS (9)

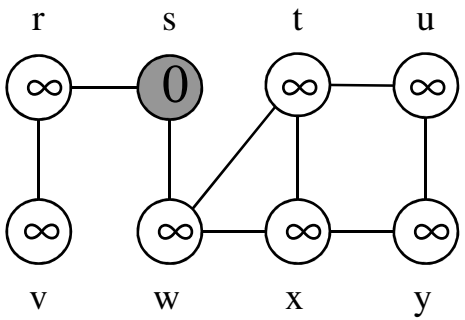




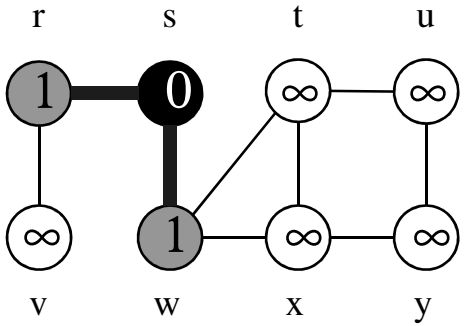
Algoritmo do BFS

- A partir de um vértice inicial *s (ancôra)*, visita-se todos os seus vértices adjacentes (**componentes conexos**)
- Para o vértice inicial *s* é assinado uma distância=0
- No primeiro passo, todos os vértices conectados a *s* são visitados e é assinalada uma distância=1 a estes vértices
- No segundo passo, todos os vértices alcançados a partir dos vértices de distância=1 são visitados
  - a estes são assinalados a distância=2, e assim sucessivamente até que todos tenham uma distância assinalada

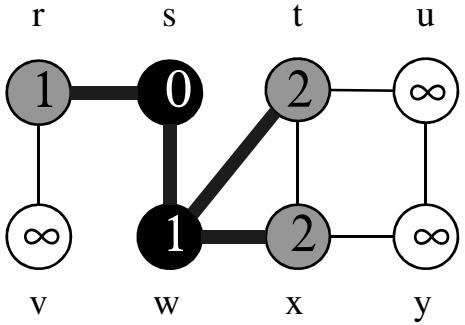
Exemplo BFS (1)



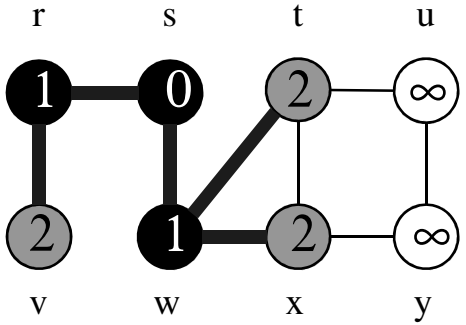
Q | s
0



Q | w | r
1 1

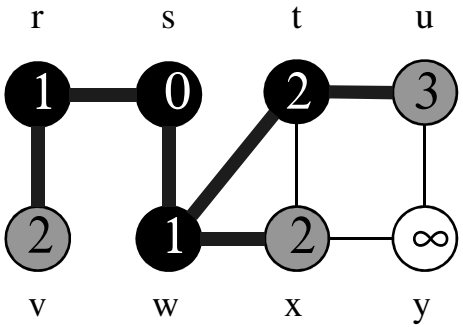


Q | r | t | x
1 2 2

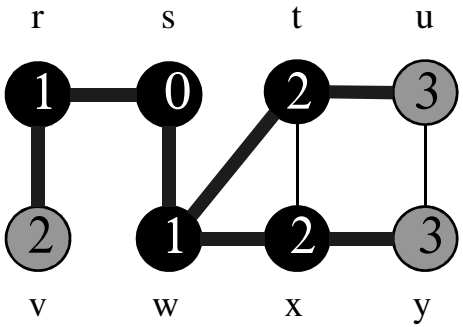


Q | t | x | v
2 2 2

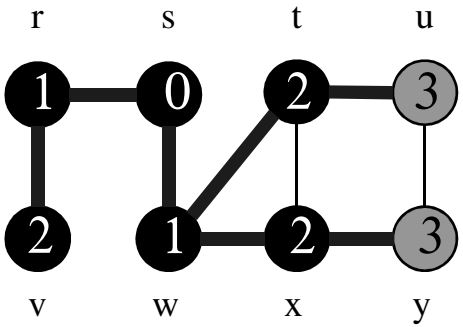
Exemplo BFS (2)



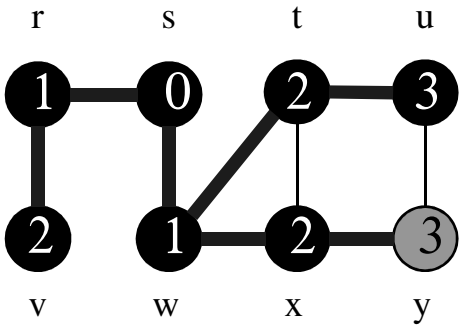
Q | x | v | u
2 2 3



Q | v | u | y
2 3 3

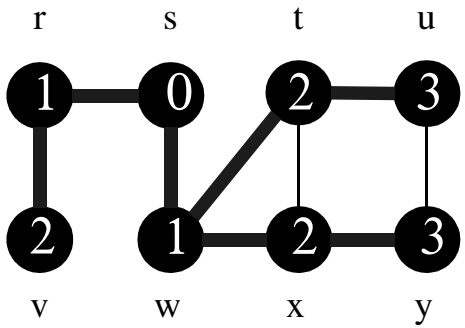


Q | u | y
3 3

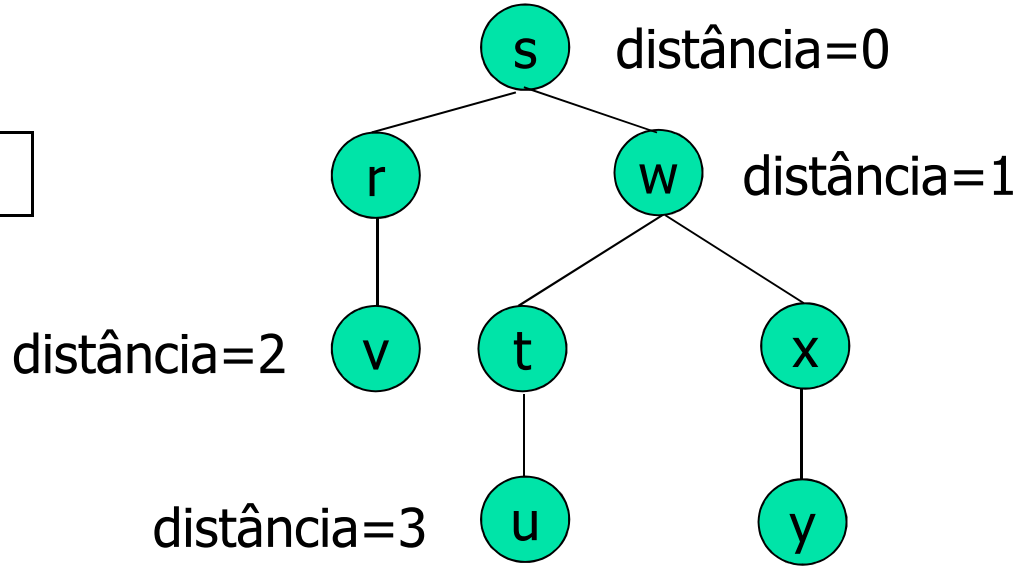


Q | y
3

Exemplo BFS (3)



Q -



Algoritmo BFS

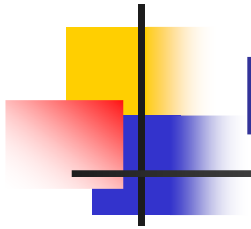
BFS(G,s)

```
01 for each vertex u ∈ V[G]-{s} // ancôra (s)
02    cor[u] ← branco  // cor de u
03    d[u] ← ∞         // distância de s
04    π[u] ← NIL       // predecessor de u
05 cor[s] ← cinza
06 d[s] ← 0
07 π[s] ← NIL
08 enqueue(Q,s)
09 while Q ≠ ∅ do
10    u ← dequeue(Q)
11    for each v ∈ Adj[u] do
12       if cor[v] = branco then
13          cor[v] ← cinza
14          d[v] ← d[u] + 1
15          π[v] ← u
16          enqueue(Q,v)
17    cor[u] ← preto
```

Inicialização dos vértices do grafo

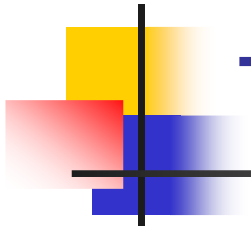
Inicializa o vértice ancôra *(s)*

Visita cada vértice adjacente de u



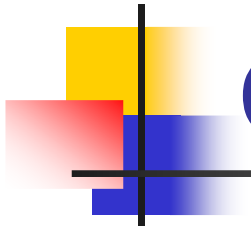
Propriedades do BFS

- Dado um grafo *G = (V,E)*, o BFS descobre todos os vértices alcançáveis a partir da fonte ***s***
- Computa a menor distância para todos os vértices alcançáveis
- O subgrafo contendo os caminhos percorridos é chamado de ***breadth-first tree***
- Para cada vértice *v* alcançável de *s*, o caminho na *breadth-first tree* de *s* a *v*, corresponde ao **caminho mínimo** ente *s* e *v* em *G*



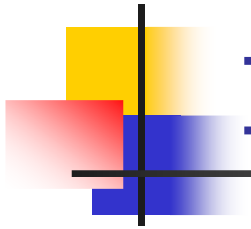
Tempo de Execução do BFS

- Seja um grafo *G = (V,E)*
  - Os vértices são enfileirados quando sua cor é branca
  - Assumindo que o *enqueue* e *dequeue* são *O(1)*, esta operação custa *O(|V|)*
  - As listas de adjacência são percorridas somente quando um vértice é desenfileirado
  - A soma de todos os tamanhos da lista é *Θ(|E|)*. O tempo *O(|E|)* é gasto no caminhamento
- Portanto o algoritmo é O(|V|+|E|)
  - ou seja, linear com a lista de adjacência que representa G



Caminhamento em Profundidade

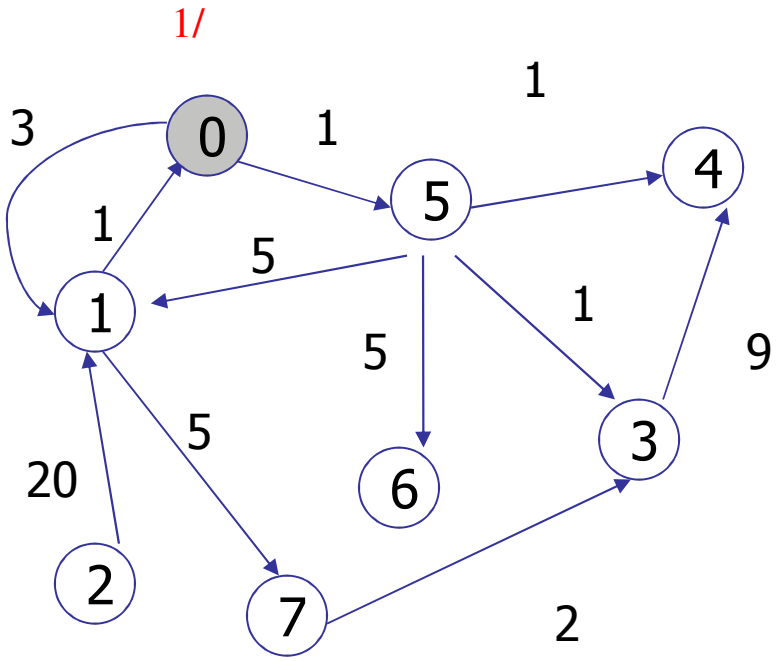
- Nós que são visitados pela primeira vez são marcados em cinza
  - O momento da primeira visita é definido como **tempo de descoberta do nó**
  - A cor cinza em *v* indica que ainda há vértices adjacentes a *v* que não foram visitados
- Quando não houver mais adjacentes a visitar a partir de um nó, ele é marcado em preto (azul nos exemplos)
  - Este momento é denominado tempo de término



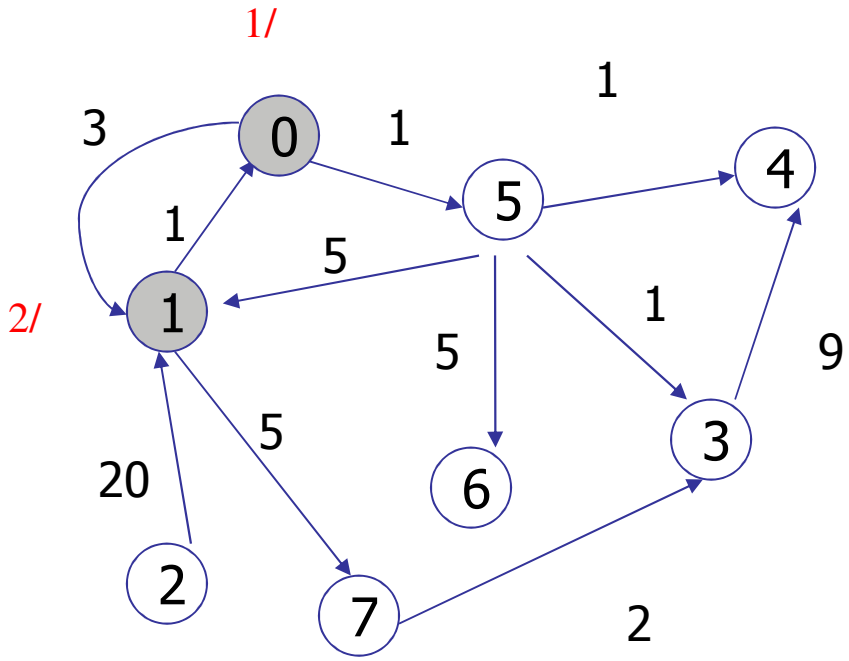
Ideia Básica do DFS (1)

- *Caminha(v)*
- Marcar v como visitado
- Para cada adjacente(va) executamos *caminha(va)*
- *Caminha* é executado novamente para todo v que não tenha sido alcançado

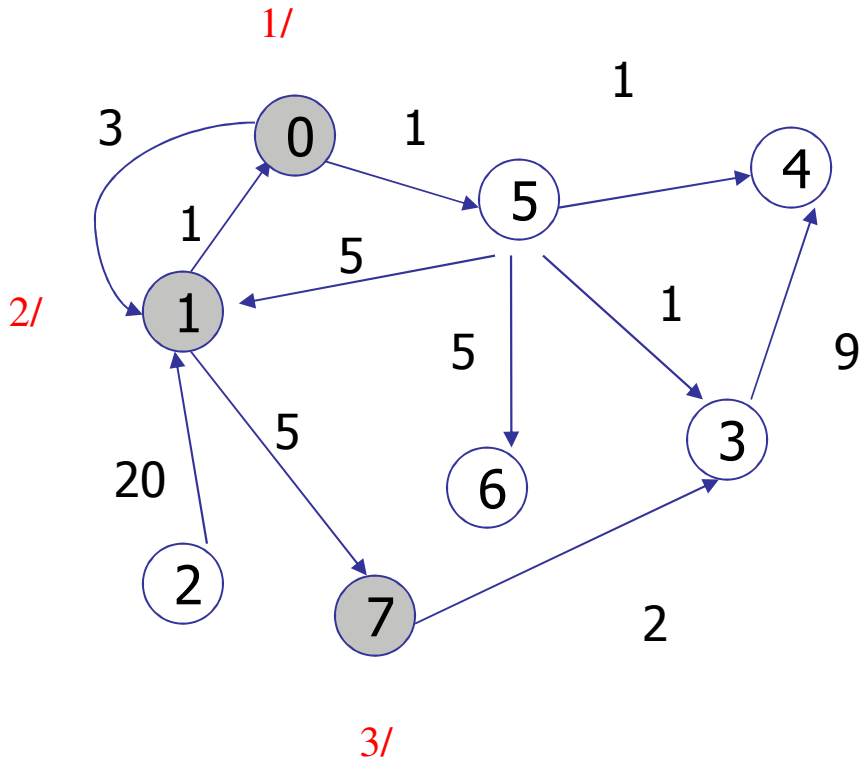
Ideia Básica do DFS (2)



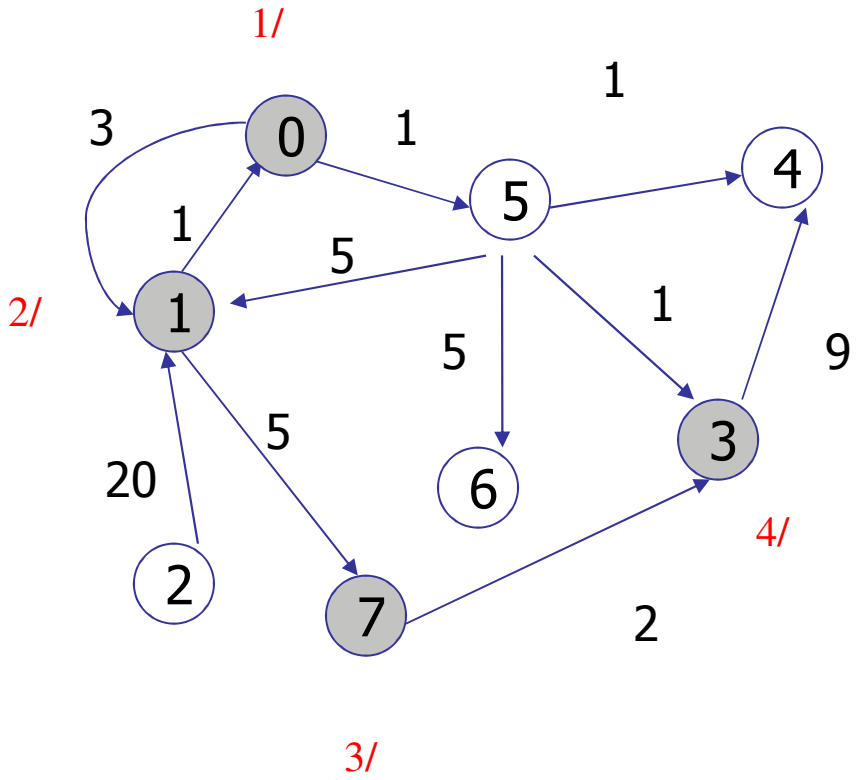
Ideia Básica do DFS (3)



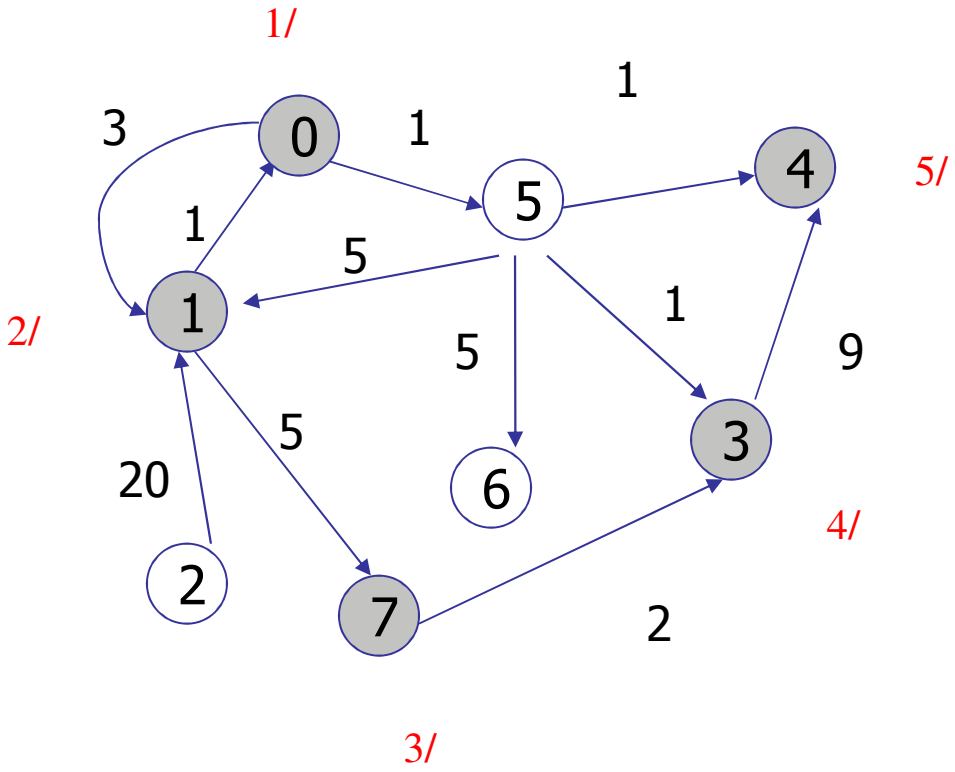
Ideia Básica do DFS (4)



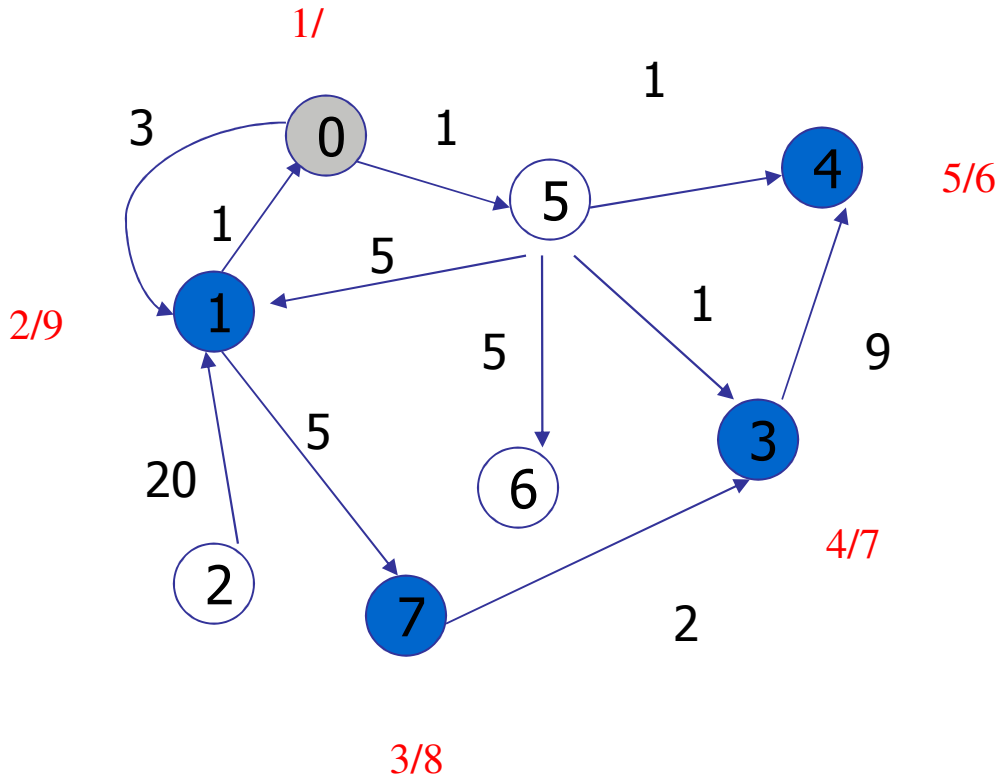
Ideia Básica do DFS (5)



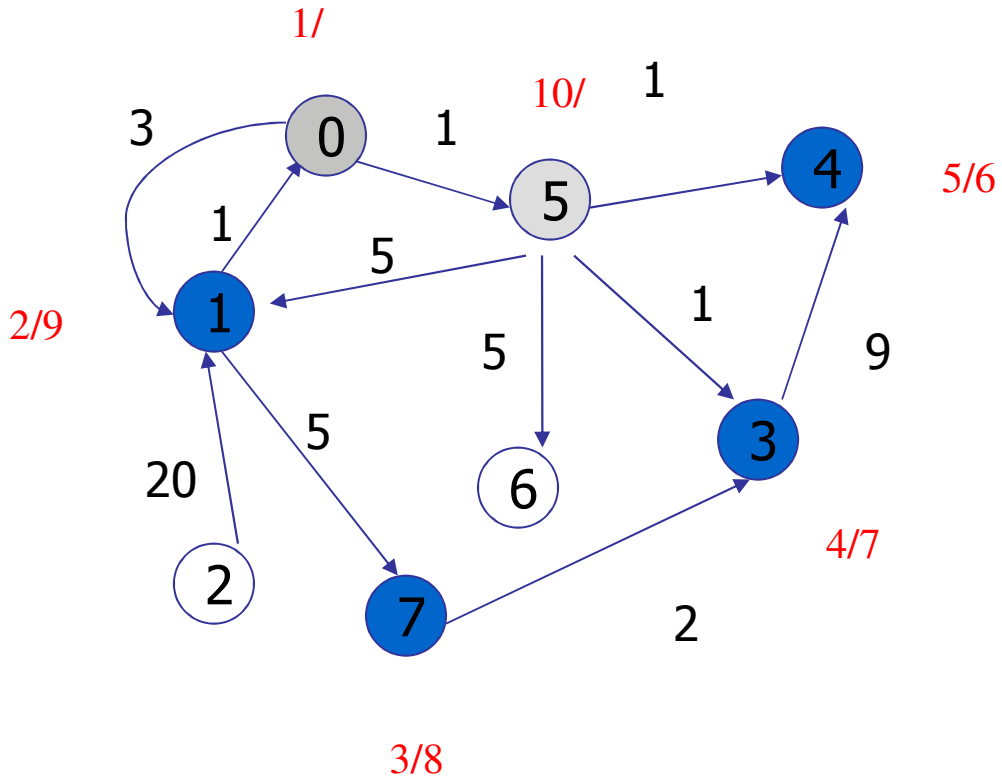
Ideia Básica do DFS (6)



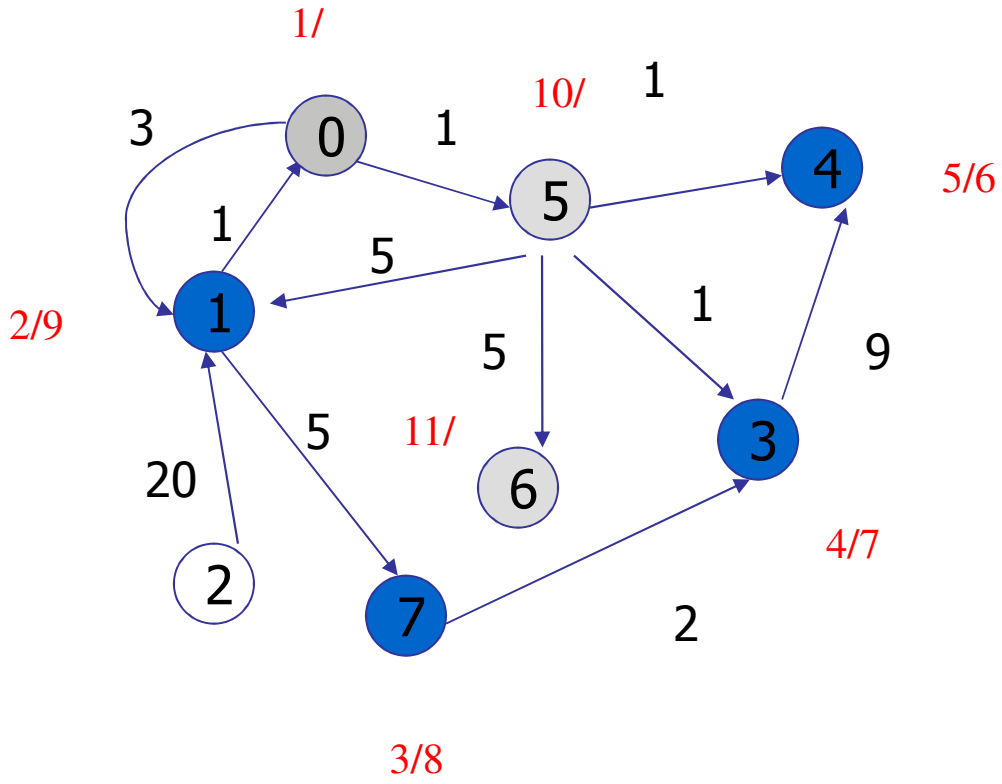
Ideia Básica do DFS (7)



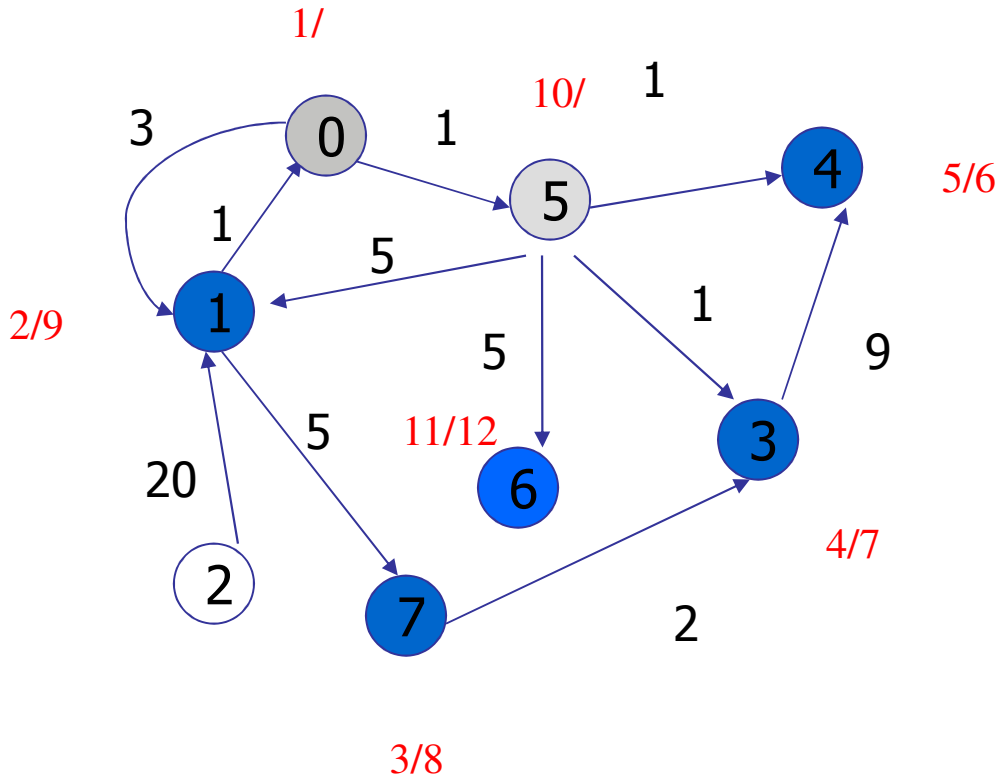
Ideia Básica do DFS (8)



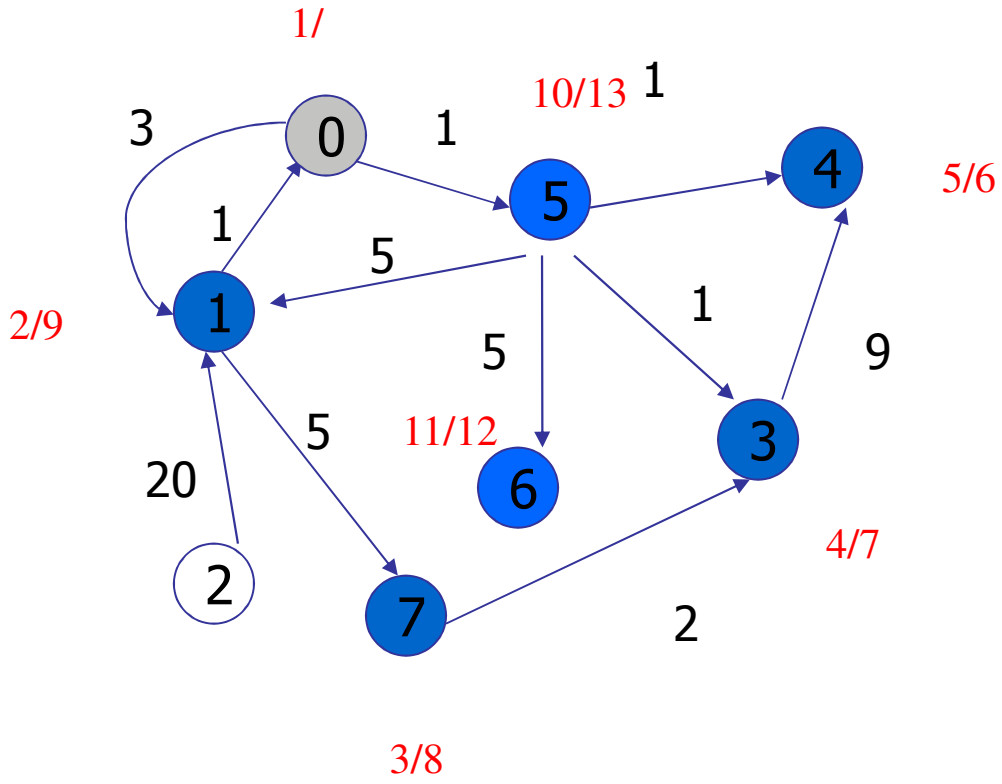
Ideia Básica do DFS (9)



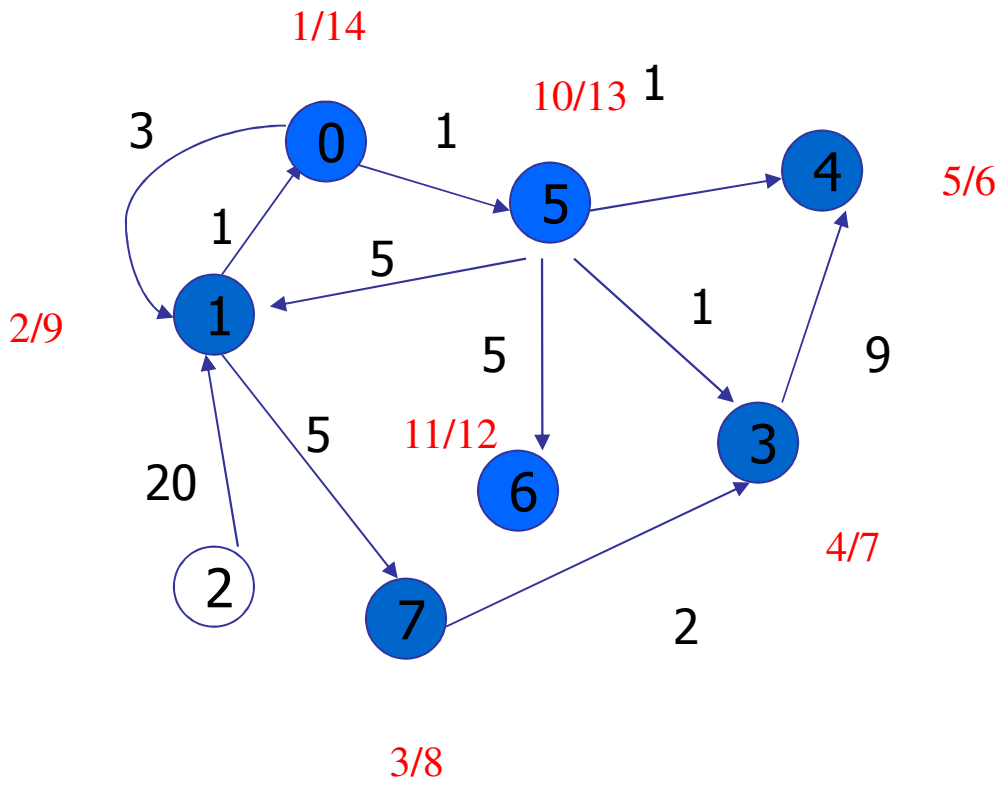
Ideia Básica do DFS (10)



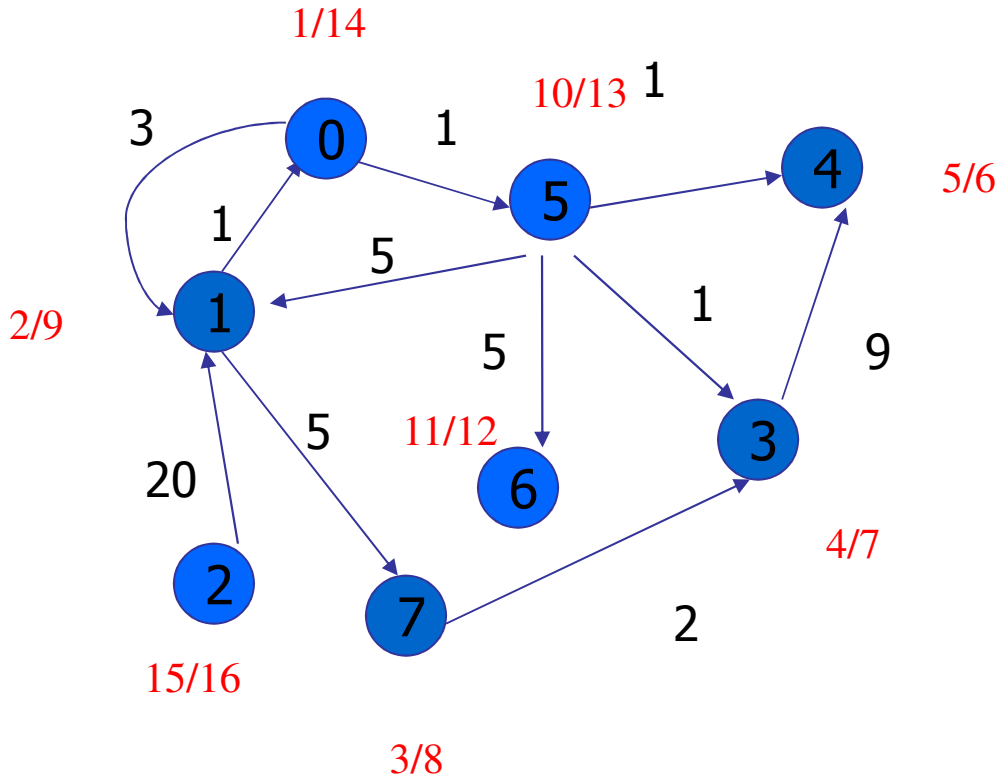
Ideia Básica do DFS (11)

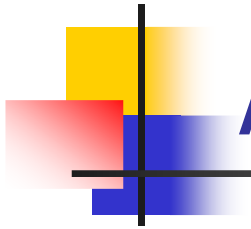


Ideia Básica do DFS (12)



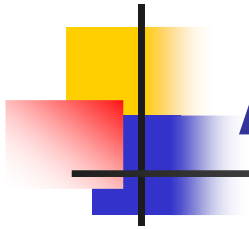
Ideia Básica do DFS (13)





Algoritmo do DFS (1)

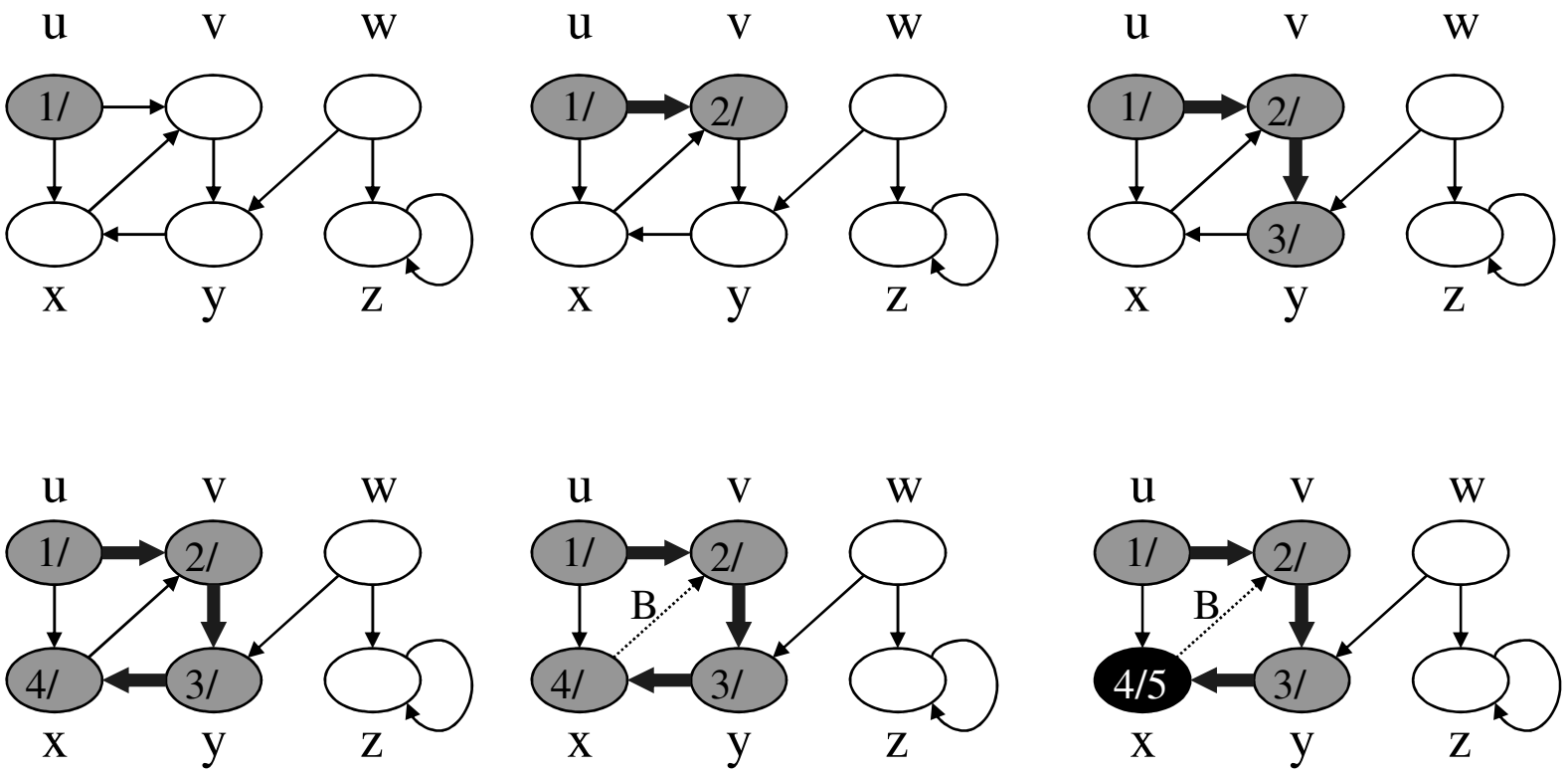
- Iniciando em um vértice *s*, escolhe-se um adjacente *t não visitado*
- Visita-se *t* e escolhe-se um adjacente *u* não visitado, e assim por diante
- Quando todos os adjacentes a *t* tiverem sido visitados, toma-se um próximo adjacente a *s* não visitado e prossegue-se o caminhamento
- Se restarem quaisquer vértices não descobertos, então um deles será selecionado como uma nova origem



Algoritmo do DFS (2)

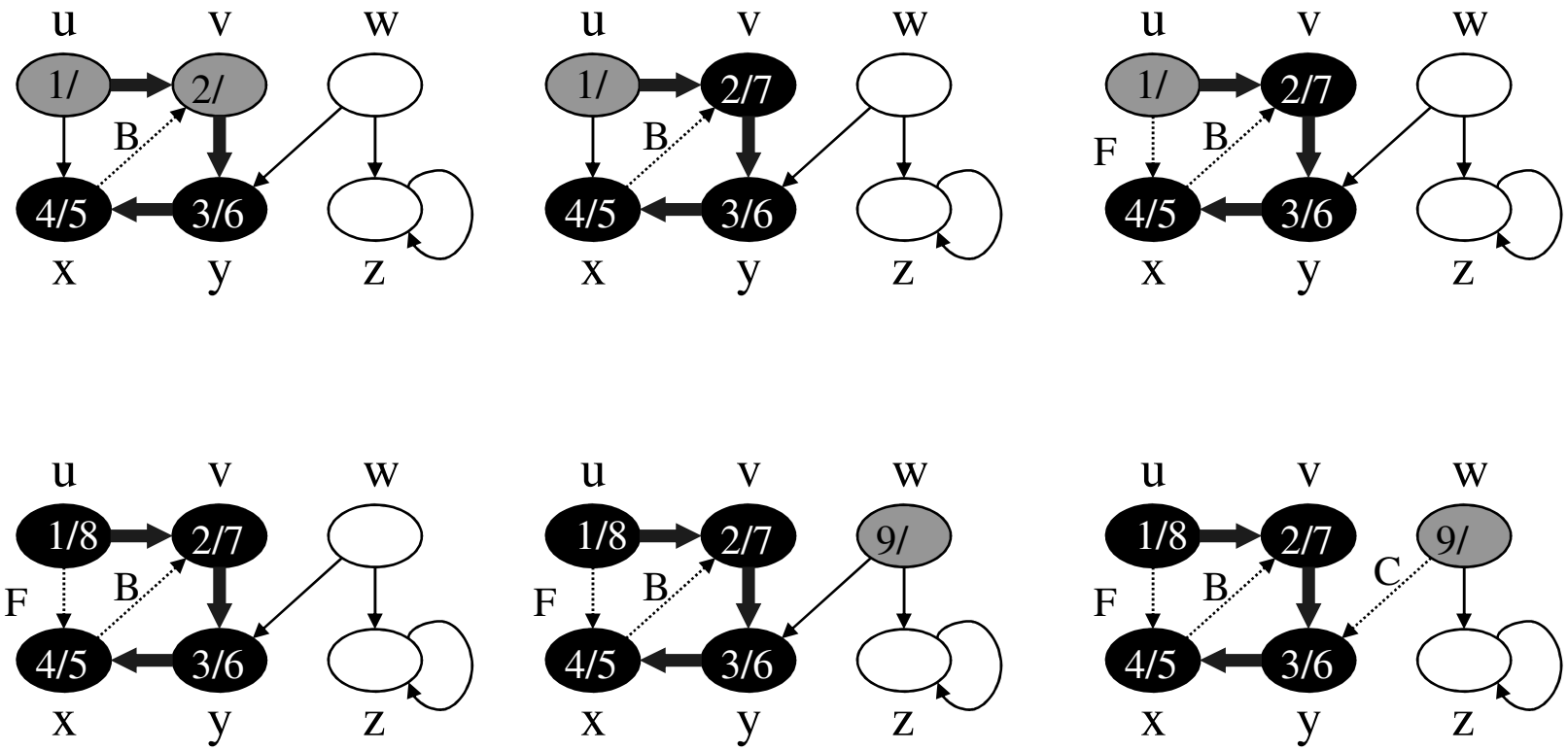
- Inicializa todos os vértices de branco
- Visita cada um dos vértives brancos
- Um vértice é **branco** se ainda não foi visitado
- Um vértice é **cinza** se foi visitado mas se os seus adjancentes ainda não foram todos visitados
- Um vértice é **preto** se ele e todos os seus adjcentes foram visitados

Exemplo DFS (1)



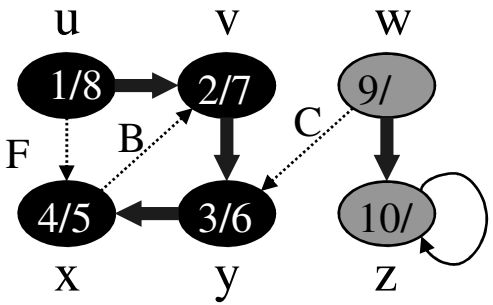
B = Aresta Reversa (cinza-cinza) – descendente para o ancestral

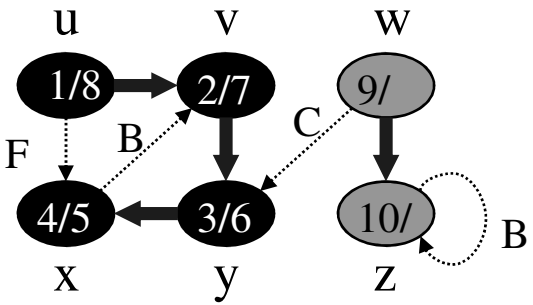
Exemplo DFS (2)

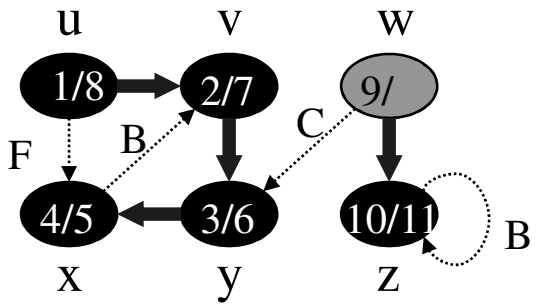


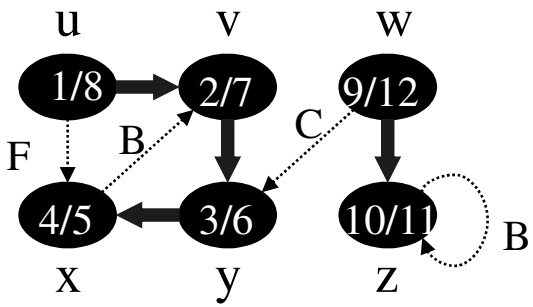
F = Aresta Direta (cinza-preto) – ancestral para o descendente
C = Aresta Cruzamento (cinza-preto) – entre sub-árvores

Exemplo DFS (3)







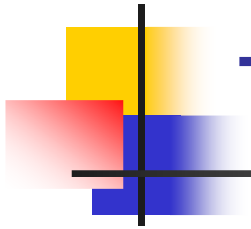


Algoritmo DFS

```
DFS(G)
1  for each vertex u ∈ V[G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V[G]
6       do if color[u] = WHITE
7             then DFS-VISIT(u)
```

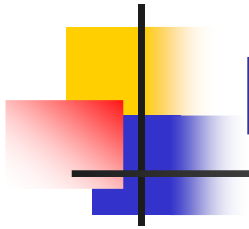
Pintam todos os vértices de branco e inicializam os campos π com NIL onde $\pi[u]$ representa o predecessor de u

```
DFS-VISIT(u)
1  color[u] ← GRAY        ▷ White vertex u has just been discovered.
2  time ← time + 1
3  d[u] ← time
4  for each v ∈ Adj[u]    ▷ Explore edge (u, v).
5       do if color[v] = WHITE
6             then π[v] ← u
7                  DFS-VISIT(v)
8  color[u] ← BLACK       ▷ Blacken u; it is finished.
9  f[u] ← time ← time + 1
```



Tempo de Execução do DFS

- Os loops do DFS tomam tempo $\Theta(|V|)$ cada, excluindo o tempo de execução de *DFS-Visit*
- *DFS-Visit* é chamado uma vez para cada vértice branco e pinta o vértice de cinza imediatamente
- Para cada chamada *DFS-visit* o loop interage sobre os adjacentes do vértice
- Assim o custo somado de todas as chamadas de *DFS-Visit* é $\Theta(|E|)$
- **Portanto, o DFS é $\Theta(|V|+|E|)$**

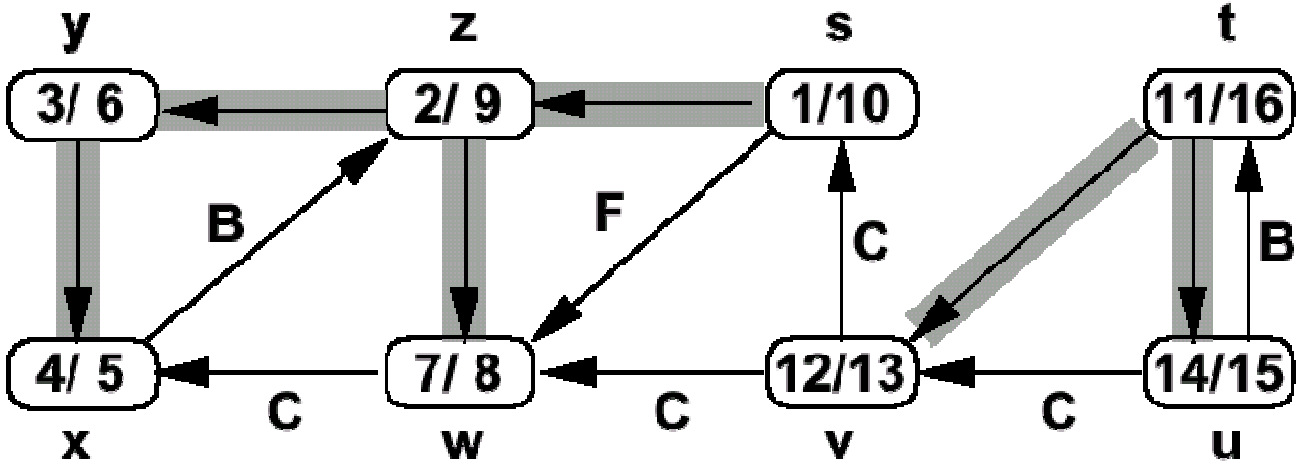


Propriedades do DFS

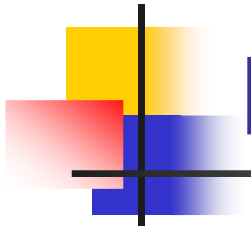
- O resultado da DFS é um **grafo de precedência**
- Pode-se identificar **ciclos** no grafo (usando as arestas de retorno (B))
- Tempos de descoberta e término podem ser usados para diversas finalidades
  - DFS *Timestamping*

DFS *Timestamping*

- O DFS gera uma ordenação de tempo monotonicamente crescente, ou seja um "relógio global" entre os vértices
- Tempo de descoberta/tempo de término
  - *d[u]/f[u]*

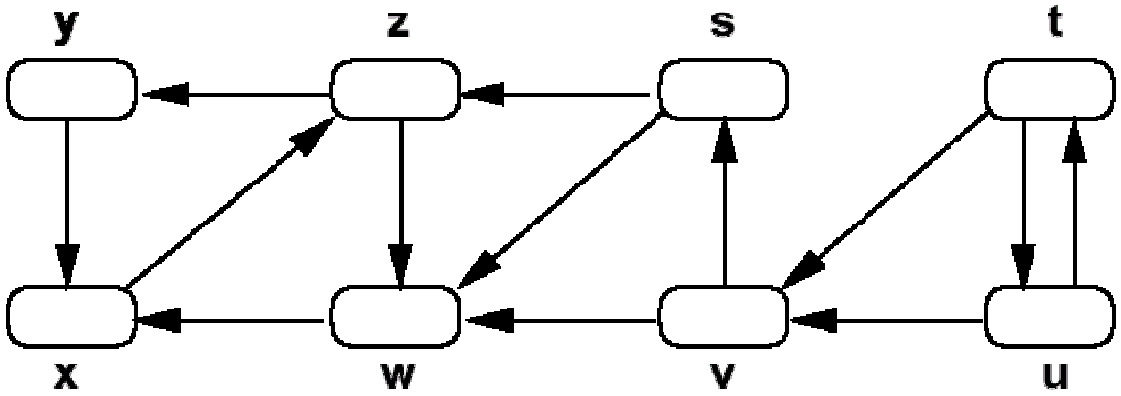


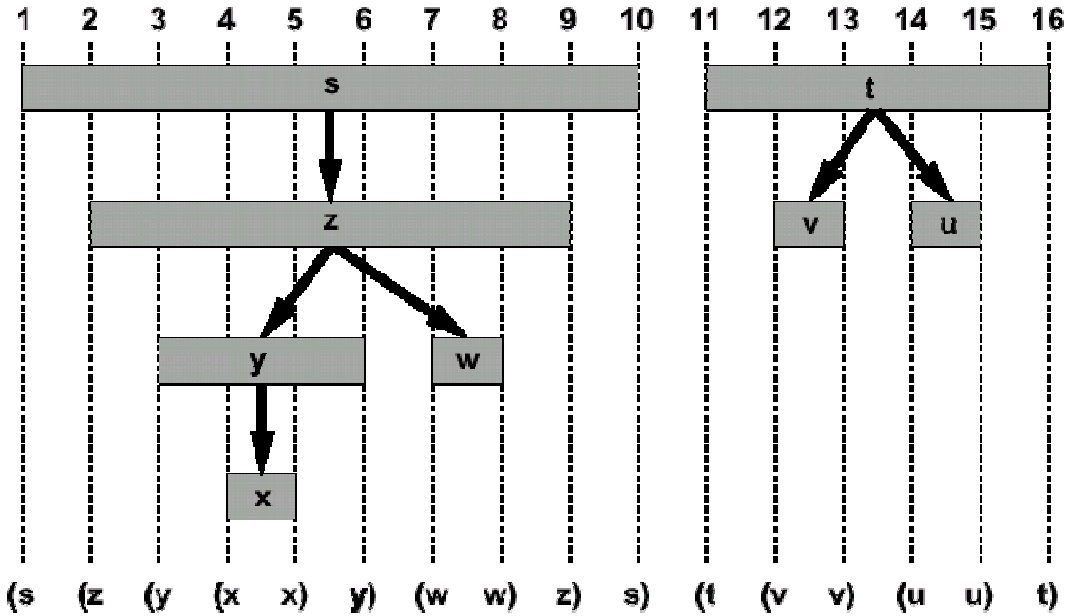
Teorema dos Parêntesis: Propriedades (1)



- Tempos de descoberta e fim podem ser associados a uma estrutura de parêntesis
  - A descoberta de u é representada com "(u"
  - O fim do processamento de u é representado com "u)"
  - A história das descobertas e fins gera uma expressão bem formada (parêntesis corretamente aninhados)

Teorema dos Parêntesis: Propriedades (2)

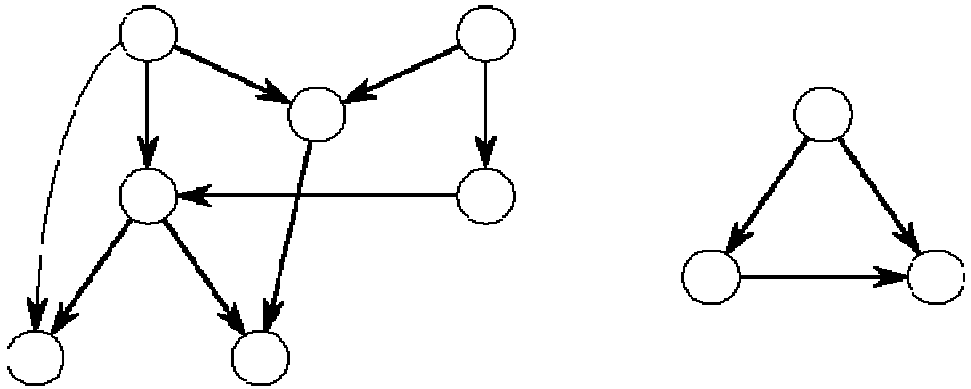




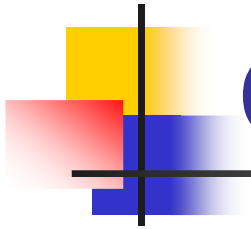
Se dois intervalos se superpõem, então um deles é aninhado no outro, e o vértice correspondente ao menor intervalo é um descendente do vértice que corresponde ao maior

DAGs

- Grafos Dirigidos Acíclicos ou *Direct Aciclic Graphs* (DAGS)



- Usados geralmente para indicar a precedência entre eventos
- Uma ordenação total para os eventos pode ser gerada usando o algoritmo de **Ordenação Topológica**



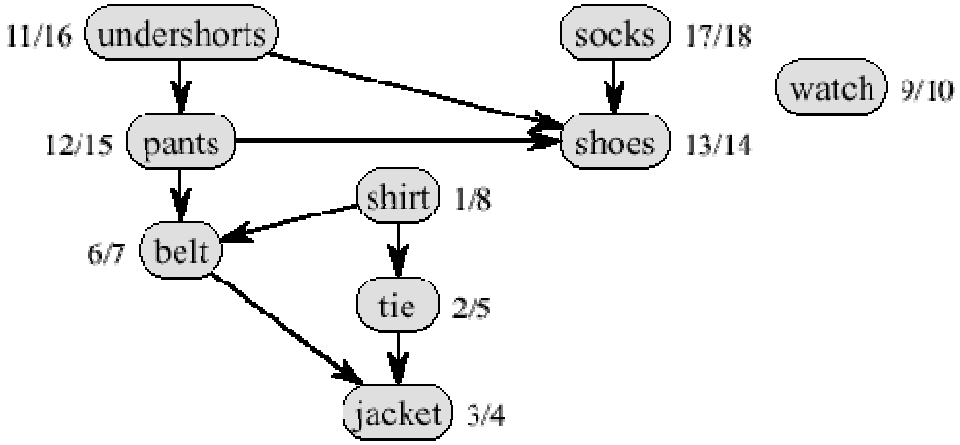
Ordenação Topológica

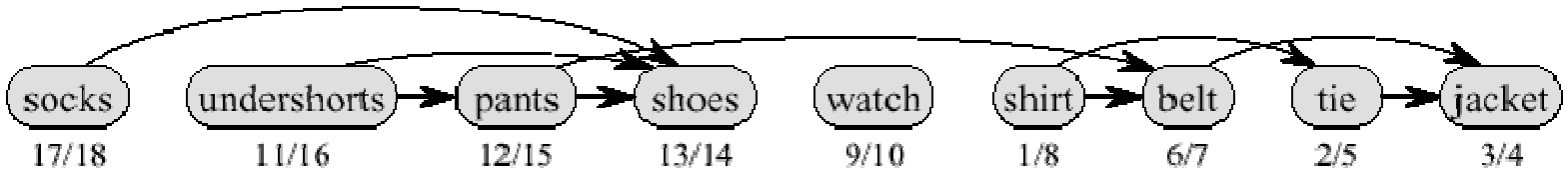
- Ordenação de um DAG
- Ordenação linear de todos os vértices tal que, para toda aresta *(u,v)* no DAG, *u* aparece antes de *v* na ordenação

```
TopSort(G)

1) Execute DFS(G) para computar os tempos de
   término f[v] de cada vértice v

2) Gere uma lista ordenada de acordo com f[v]
```

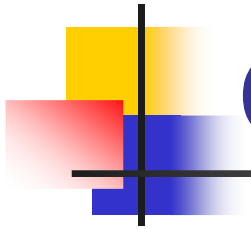
Ordenação Topológica: Exemplo

Qual seria a ordenação topológica (relações de precedência)?



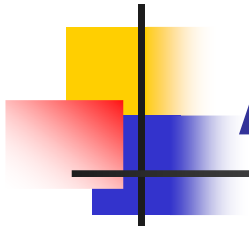


Ordenação Topológica: Observações

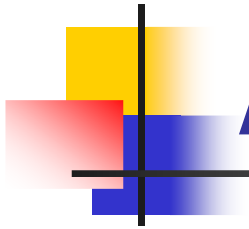


- A ordenação pode iniciar por qualquer vértice
- É possível encontrar diferentes ordenações topológicas corretas para um mesmo grafo orientado acíclico

Ordenação Topológica: Aplicações (1)

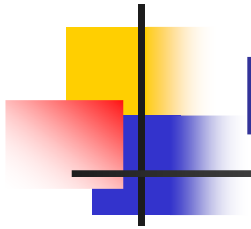


- É utilizado sempre se necessita uma ordem para execução de tarefas onde há pré-requisitos (dependências) entre tarefas
- Matérias para se cursar uma graduação: o algoritmo garante que nenhum pré-requisito será quebrado
- Construir um prédio, onde há várias tarefas: colocar telhado, montar parede, montar porta, parede, acabamento
  - Não se coloca o teto antes de subir as paredes



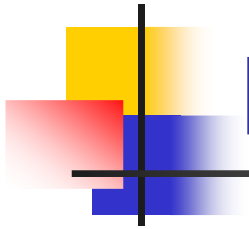
Ordenação Topológica: Aplicações (2)

- Uma outra aplicação são programas de planilha eletrônica
  - Há várias células e às vezes uma célula tem uma fórmula que depende de outras células. Se existe uma célula A1, uma célula B1 com fórmula que depende de A1 e uma célula C1 que depende de B1, não se pode atualizar A1, depois C1 e depois B1



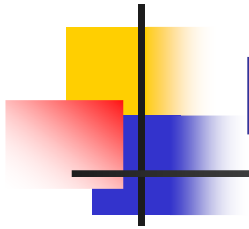
Exercício 1

- Desenhe o grafo *G=(V,E)* onde
  - *V={0,1,2,3,4,5,6}*
  - *E={ (0,1,4), (2,5,5), (1,3,2), (4,6,1) }*
  - O terceiro elemento de cada aresta é um atributo da aresta
- Escreva um programa para representar este grafo usando lista e matriz de adjacência
  - Use as funções **fopen** (abri arquivo) e **feof** (checa fim do arquivo) e **fscanf** (extrai caracter)
  - O arquivo deve conter pares de arestas do grafo



Exercício 2

- Apresente o caminhamento em largura para o seguinte grafo *G=(V,E)* onde
  - *V={0,1,2,3,4,5,6,7}*
  - *E={ (0,1), (0,2), (1,3), (1,4), (2,3), (3,4), (3,5), (5,6), (5,7), (6,7)}*



Exercício 3

- Apresente o caminhamento em profundidade para o seguinte grafo *G=(V,E)* onde
  - *V={0,1,2,3,4,5,6,7}*
  - *E={ (0,1), (0,2), (1,3), (1,4), (2,3), (3,4), (3,5), (5,6), (5,7), (6,7)}*

Exercício 4

- Mostre a ordenação de vértices produzida por TOPOLOGICAL-SORT quando ele é executado sobre o grafo abaixo