

ALGORITMOS E ESTRUTURAS DE DADOS II

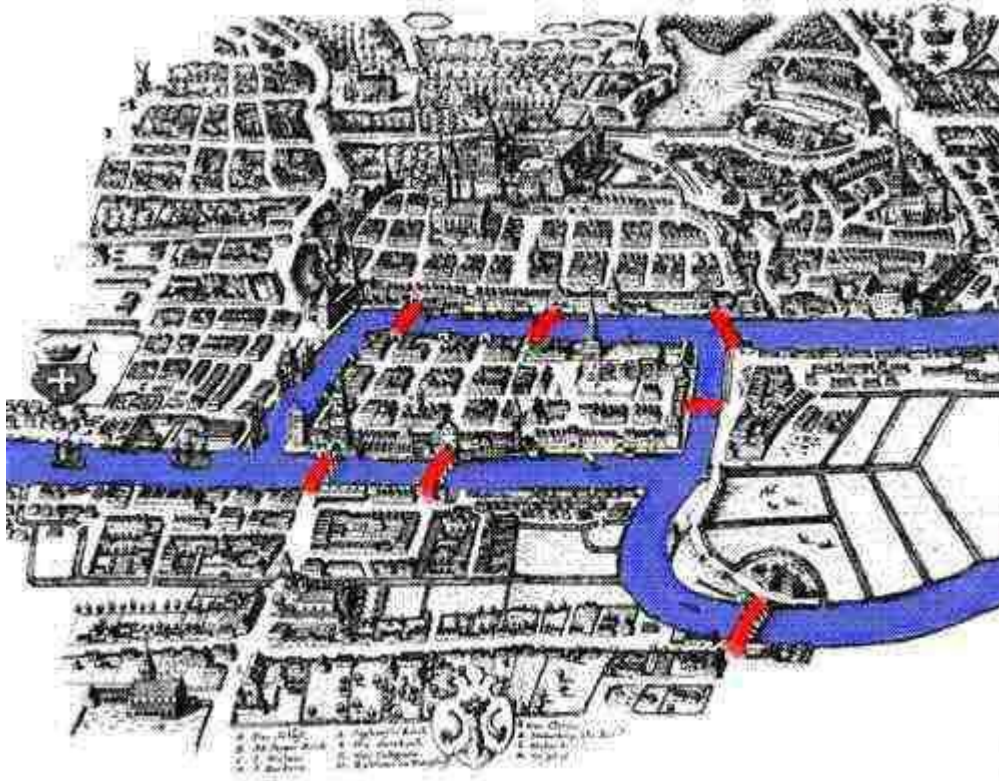
Grafos

Prof. Rafael Fernandes Lopes

<http://www.dai.ifma.edu.br/~rafaelf>

rafaelf@ifma.edu.br

História



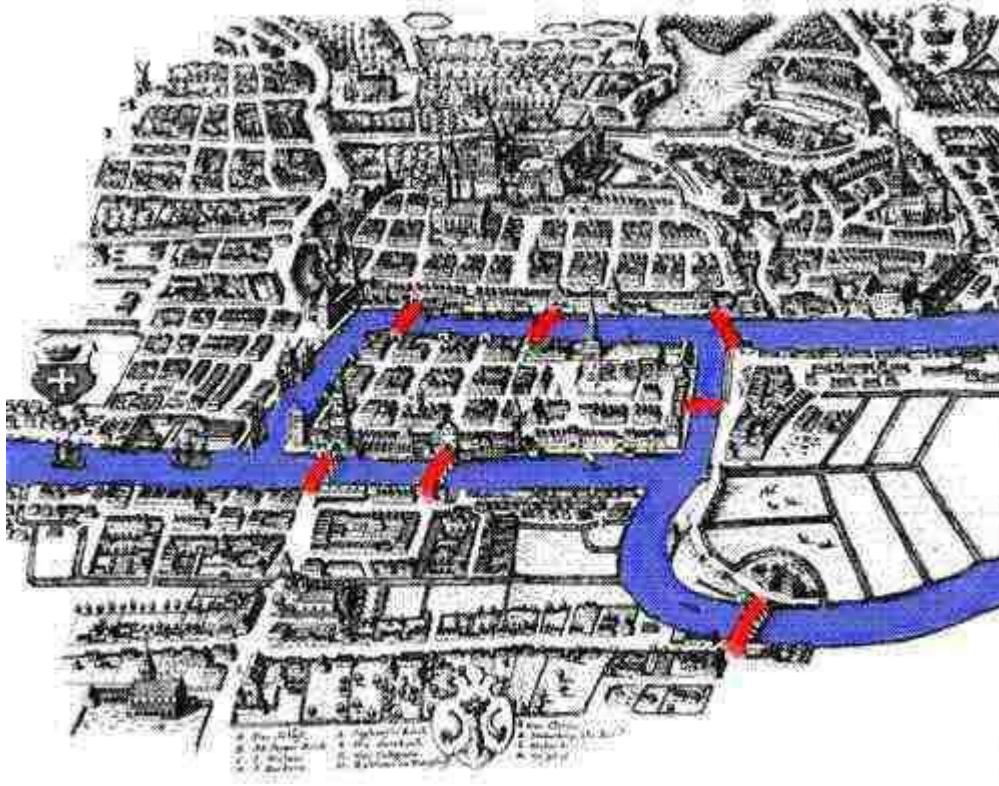
Königsberg/Kaliningrado



Leonh. Euler

1707 - 1783

História

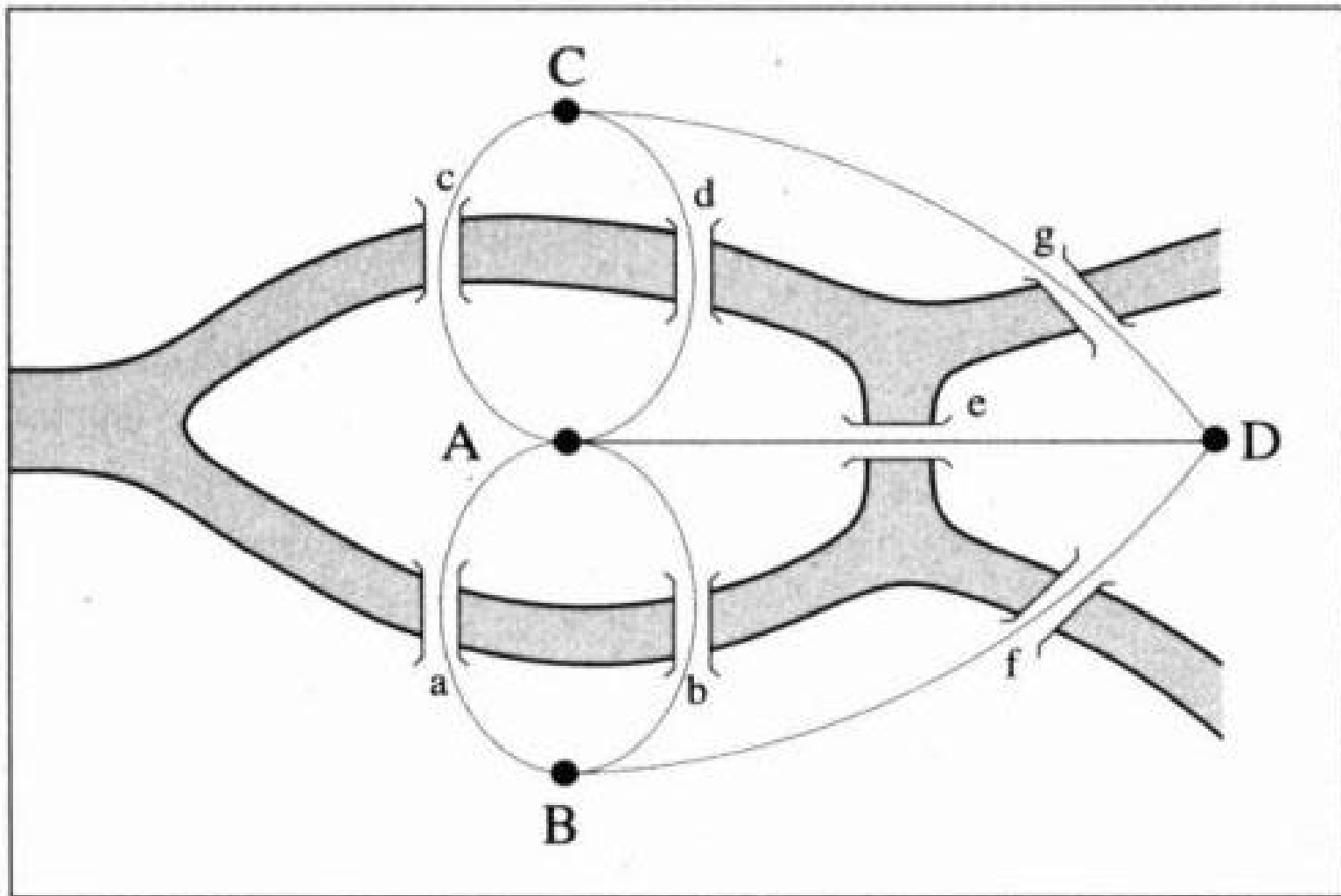


Königsberg/Kaliningrado



Como passar por todas as pontes, sem passar duas vezes por nenhuma delas?

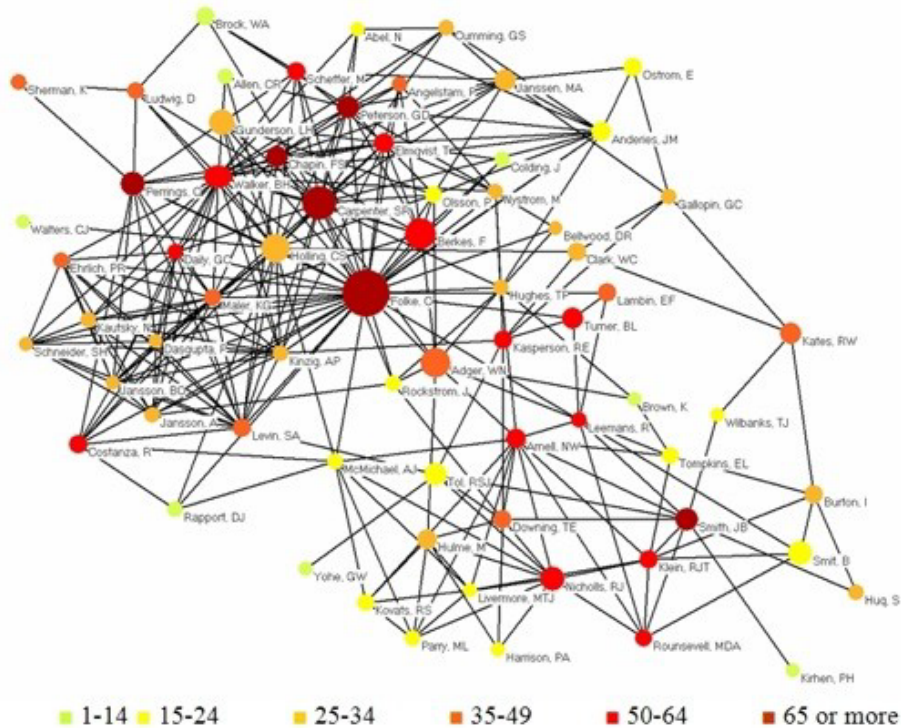
História



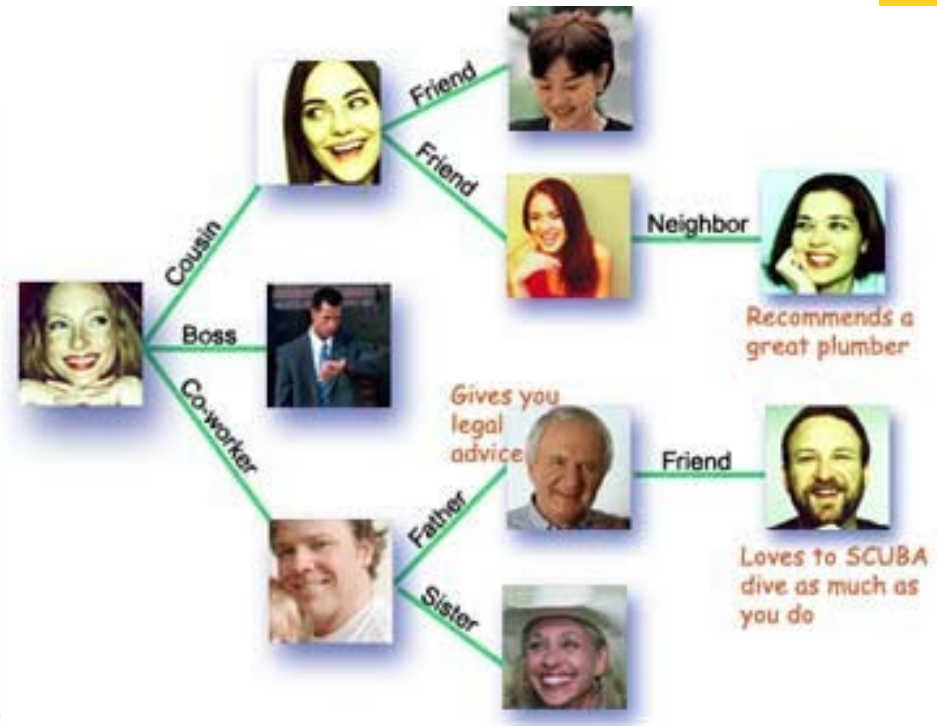
Aplicações

- Vários problemas podem ser representados como problemas de grafos
 - Rede de Transporte
 - Roteamento de pacotes em rede de computadores
 - Planejamento de tarefas (workflow)

Aplicações – Redes Sociais



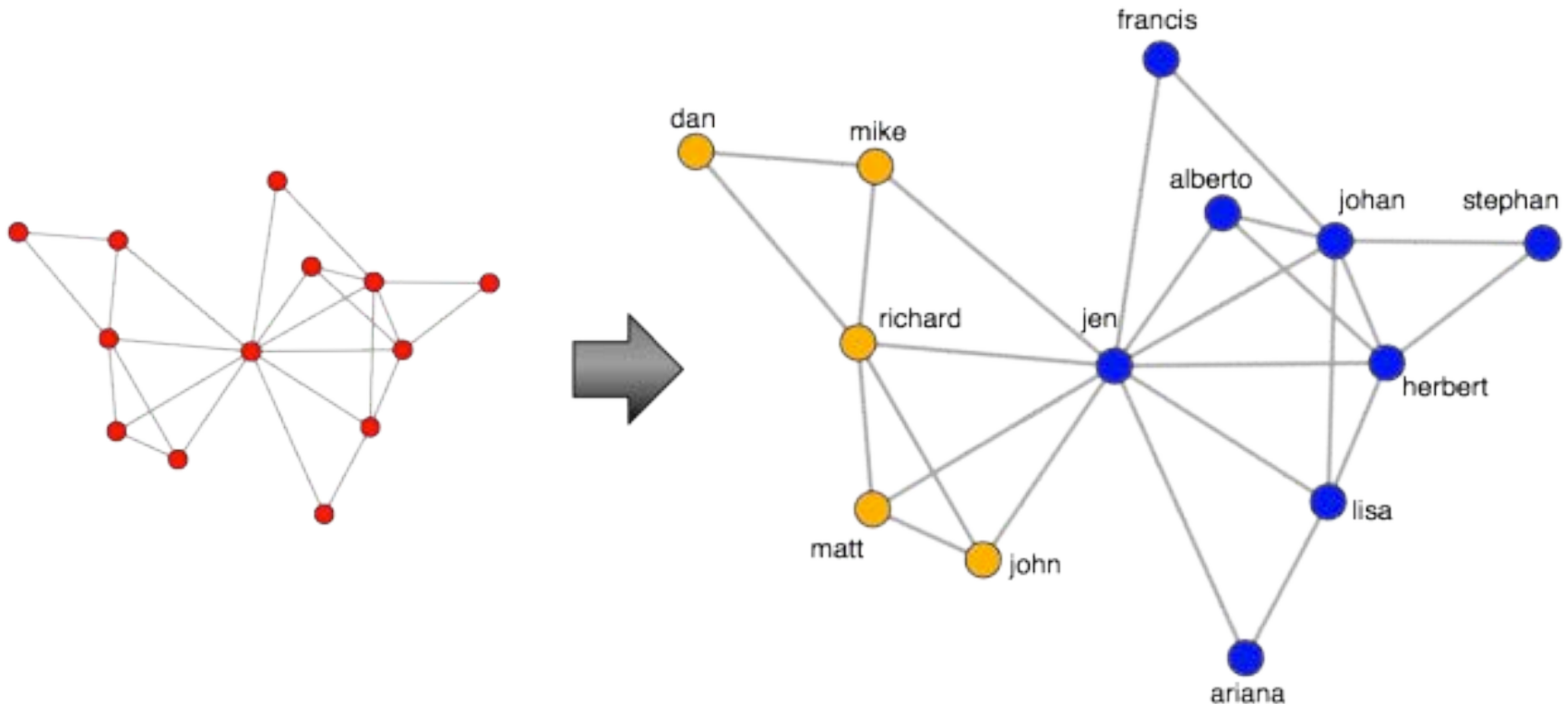
Co-autores



Pessoas conhecidas

Aplicações – Redes Sociais

- Encontrar comunidades em grafos de relacionamentos



Aplicações – Redes Sociais

- Experimento “*small-world*” (Milgram, 1969)
- Análise do “número de saltos” em uma rede de pessoas conhecidas
- Idéia: pessoas deveriam selecionar um conhecido que poderia encaminhar uma carta ao destino final
- Muitas cartas se perderam
 - 25% chegaram ao destino, após passar, em média, por 6 pessoas - *Six Degree of Separation*





Stanley Milgram
1933-1984



Aplicações – SIG

Get directions

My places



A

Liberty Grove, New South Wales

B

48 Pirrama Rd, Pyrmont NSW

Add Destination - Show options

GET DIRECTIONS

Bicycling directions are in beta.

Use caution and please report unmapped bike routes, streets that aren't suited for cycling, and other problems [here](#).

Suggested routes

Lilyfield Rd	14.9 km, 1 hour 1 min
Lyons Rd West and Lilyfield Rd	16.4 km, 1 hour 6 mins

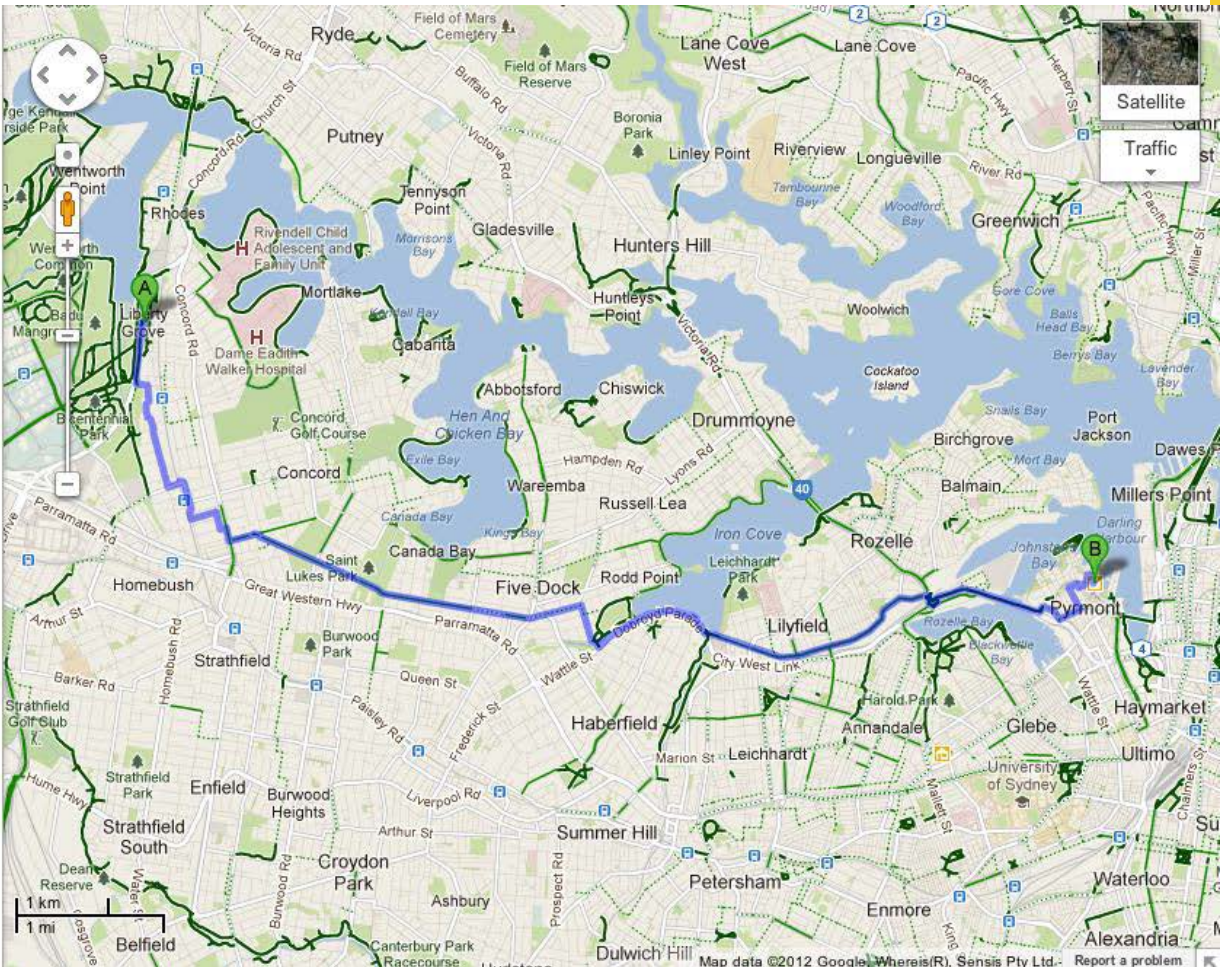
Or take [Public Transport](#) (Bus, one transfer) 1 hour 10 mins

Bicycling directions to 48 Pirrama Rd, Pyrmont NSW 2009

A

Liberty Grove NSW

1. Head east

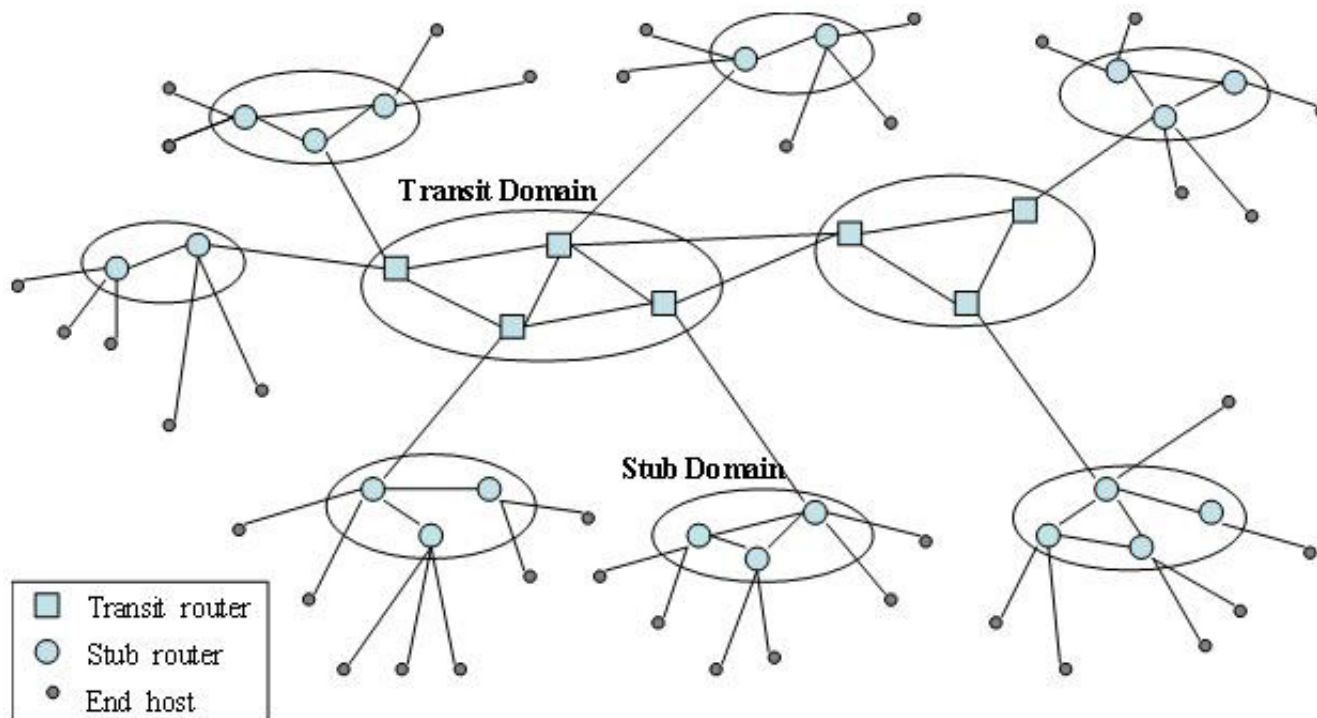


Satellite

Traffic

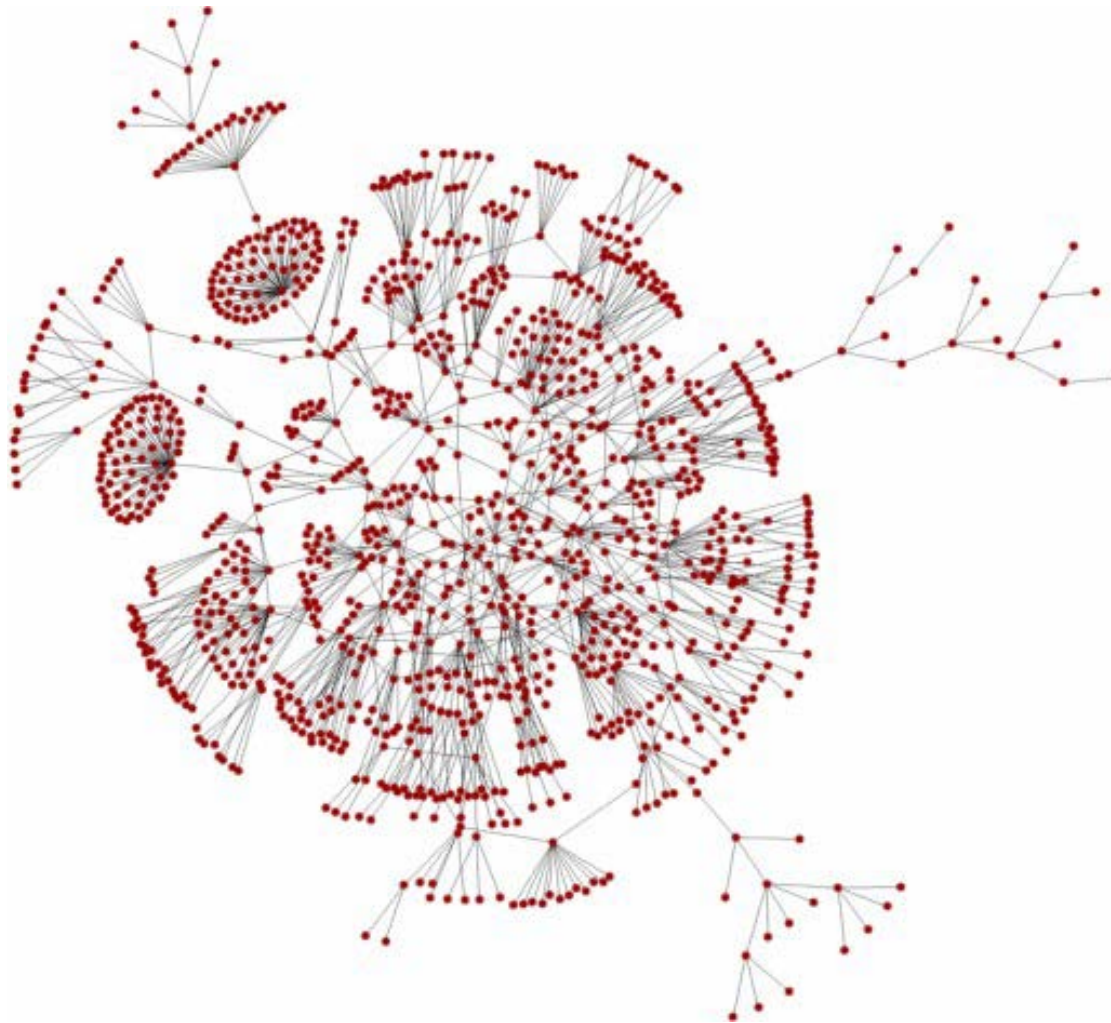
Map data ©2012 Google, Whereis(R), Sensis Pty Ltd. Report a problem

Aplicações – Redes de Computadores



Roteamento

Aplicações – Redes de Computadores



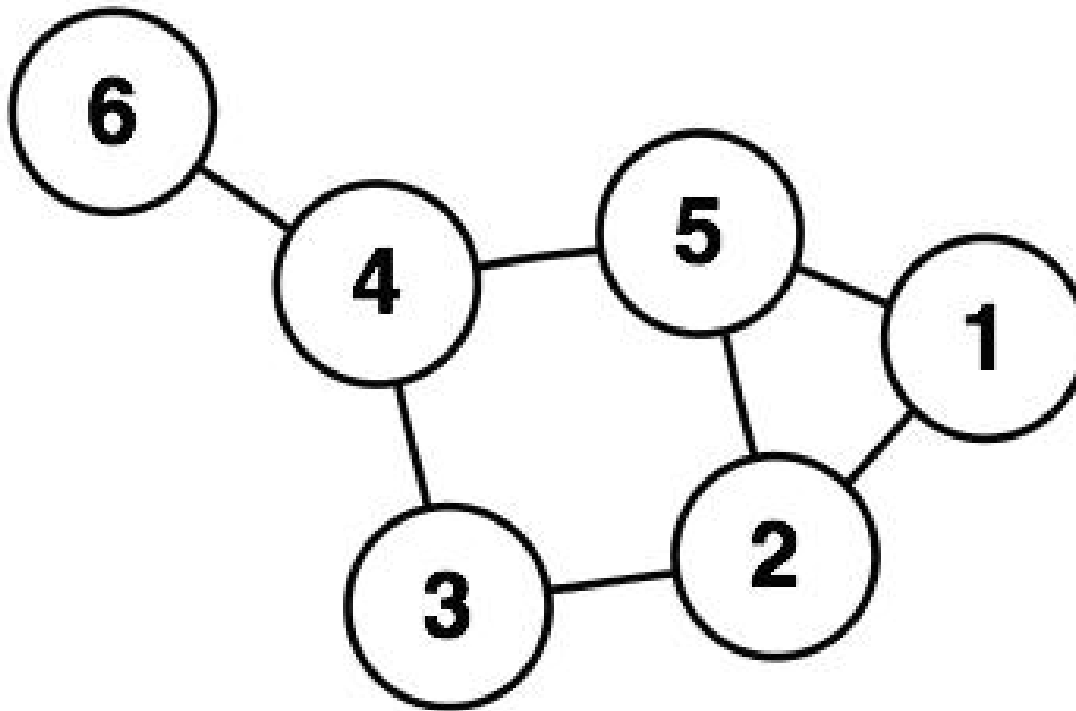
Redes Peer-to-Peer

Definição

- Um **grafo** consiste em um conjunto de vértices (V) e um conjunto de arestas (E), na forma $G=(V,E)$
 - Grafos direcionados (ou digrafos): E representa relações binárias em V
 - Grafos não-direcionados: E consiste em pares não ordenados de vértices, logo cada elemento de E é um conjunto (u,v) , em que u e $v \in V$, $(u,v) = (v,u)$ e $u \neq v$
- Muitos problemas são simplificados pela utilização de grafos, especialmente os problemas envolvendo caminhos
- A busca em grafos se dividem basicamente em busca **em profundidade** e busca **em largura**

Representação gráfica

- Vértices são representados por círculos
- Arestas ligam dois vértices



Representação gráfica

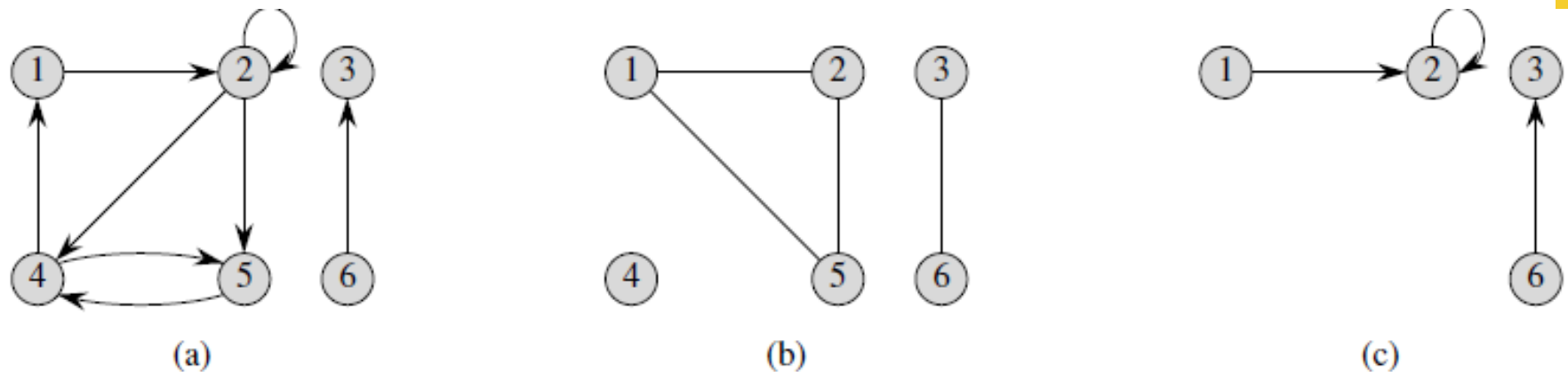


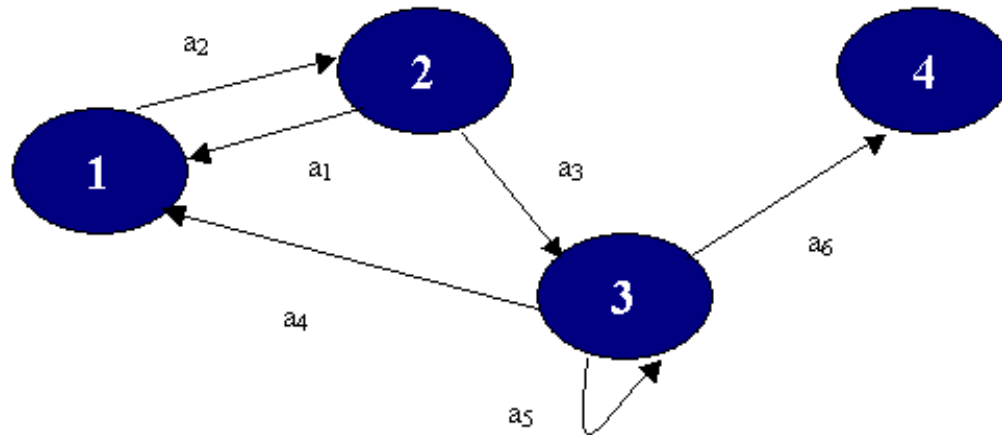
Figure B.2 Directed and undirected graphs. (a) A directed graph $G = (V, E)$, where $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)\}$. The edge $(2, 2)$ is a self-loop. (b) An undirected graph $G = (V, E)$, where $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{(1, 2), (1, 5), (2, 5), (3, 6)\}$. The vertex 4 is isolated. (c) The subgraph of the graph in part (a) induced by the vertex set $\{1, 2, 3, 6\}$.

Alguns Conceitos

- Se (u, v) é uma aresta de um digrafo, dizemos que (u, v) é **incidente de** ou **deixa** o vértice u e é **incidente para** ou **entra** no vértice v
- Caso o grafo seja não-direcionado dizemos que (u, v) é **incidente** nos vértices u e v
- Se (u, v) é uma aresta de um grafo G , então dizemos que v é adjacente ao vértice u
- Quando o grafo é não-direcionado, a relação de adjacência é **simétrica**

Alguns Conceitos

■ Grafos direcionados



- No exemplo o vértice 4 é chamado de sumidouro
- O grau de um vértice indica a quantidade de arestas que estão ligadas ao vértice
 - Em grafos direcionados, vértices têm grau de entrada e saída
- Um circuito é um caminho partindo da origem e voltando a ela mesma

Alguns Conceitos

- Uma aresta pode ter um custo associado a ela w , onde $w(u,v)$ representa o peso da aresta
- O peso de uma aresta é armazenado juntamente com o vértice v na aresta u
- Um caminho de tamanho k de um vértice u para um vértice u' é uma sequência $\langle v_0, v_1, v_2, \dots, v_k \rangle$ de vértices dado que $u = v_0$, $u' = v_k$ e $(v_{i-1}, v_i) \in E$, para i de 1 a k
 - O caminho contém os vértices $v_0, v_1, v_2, \dots, v_k$ e as arestas $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$

Ciclos

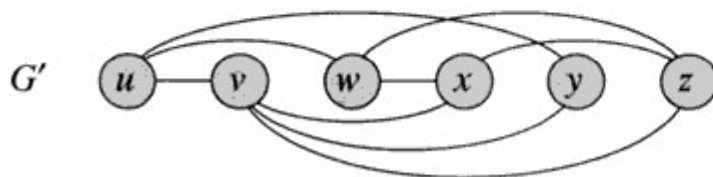
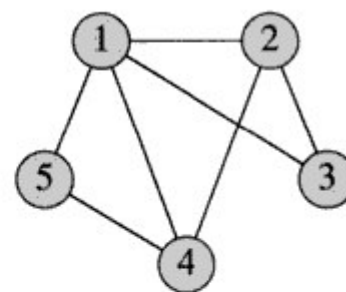
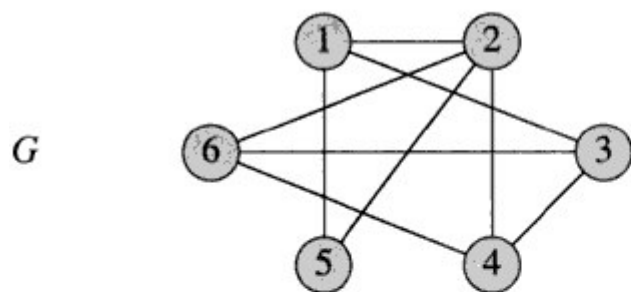
- Em um grafo direcionado, um caminho $\langle v_0, v_1, v_2, \dots, v_k \rangle$ forma um **ciclo** se $v_0 = v_k$
 - Um ciclo é **simples** se, adicionalmente, v_1, v_2, \dots, v_k são distintos
- Em um grafo não-direcionado, um caminho $\langle v_0, v_1, v_2, \dots, v_k \rangle$ forma um **ciclo** se $v_0 = v_k$ e v_1, v_2, \dots, v_k são distintos
 - Ex: Figura B.2 (b) o caminho $\langle 1, 2, 5, 1 \rangle$ é um ciclo
- Um grafo simples e sem ciclos é dito **acíclico**

Grafos Conectados e Fortemente Conectados

- Um grafo não-direcionado é conectado se todos os pares de vértices são conectados por um caminho
 - **Componentes conectados** seguem a relação “é alcançável a partir de”
 - Ex: Figura B.2(b) tem três componentes conectados: $\{1,2,5\}$, $\{3,6\}$ e $\{4\}$
- Um grafo direcionado é fortemente conectado se todos os pares de vértices são alcançáveis um a partir do outro
 - **Componentes fortemente conectados** seguem a relação “são mutuamente alcançáveis”
 - Um grafo direcionado só é fortemente conectado se ele tiver somente um componente fortemente conectado
 - Ex: Figura B.2(a) tem três componentes fortemente conectados: $\{1,2,4,5\}$, $\{3\}$ e $\{6\}$

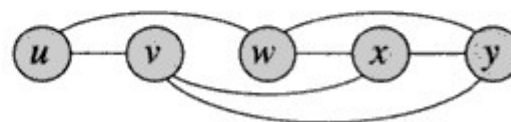
Grafos Isomórficos

- Dois grafos G e G' são isomórficos se e somente se seus vértices podem ser rotulados de tal forma que as suas correspondentes adjacências sejam iguais



(a)

Isomórficos



(b)

Não isomórficos

Tipos especiais de grafos

- Se for possível estabelecer um caminho de qualquer vértice para qualquer outro vértice de um grafo, diz-se que o grafo é **conexo**
 - *i.e.*, ele é conectado ou fortemente conectado
- Um grafo completo é um grafo em que, para cada vértice do grafo, existe uma aresta conectando este vértice a cada um dos demais
- Uma árvore é um grafo simples, acíclico, não-direcionado e conexo
 - Às vezes, um vértice da árvore é distinto e chamado de raiz
- *Directed Acyclic Graph* = DAG

Algumas considerações

- Um grafo $G' = (V', E')$ é um subgrafo de $G = (V, E)$ se $V' \subseteq V$ e $E' \subseteq E$
- Um grafo não-direcionado pode ser convertido em uma versão direcionada ((u, v) é substituído por (u, v) e (v, u))
- Um grafo direcionado pode ser convertido em sua versão não-direcionada (toda aresta (u, v) só pode ser aresta não-direcionada caso $u \neq v$, ou seja, devem ser removidas as direções e os *self-loops*)
 - Um vizinho de um vértice u é qualquer vértice adjacente a u na versão não-direcionada, ou seja, v é vizinho de u caso (u, v) $\in E$ ou (v, u) $\in E$

Algumas considerações

- Técnicas básicas para representação de grafos:
 - Matriz de adjacências
 - Adequada para grafos **densos** (*i.e.*, $|E|$ é próximo de $|V|$)
 - Lista de adjacências
 - Adequada para grafos esparsos (*i.e.*, $|E|$ é muito menor que $|V|$)

Estruturas de Dados – Matriz de adjacências

- Matriz $|V| \times |V|$ com $E = (e_{ij})$, dado que:

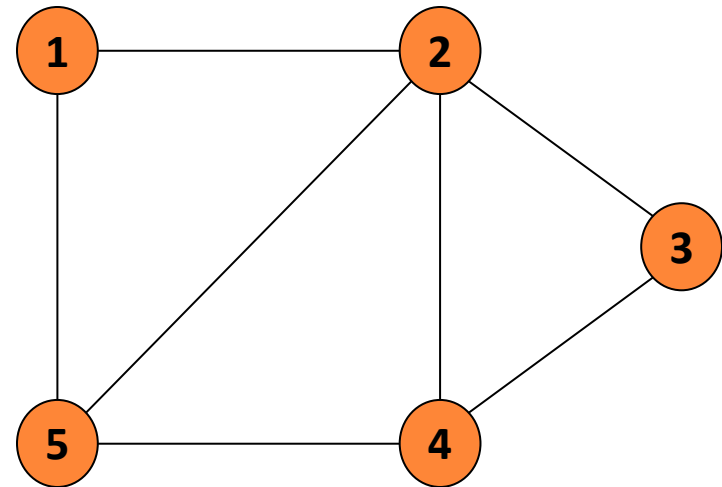
$$e_{ij} = \begin{cases} 1, & \text{se } (i,j) \in E, \\ 0, & \text{caso contrário} \end{cases}$$

- A matriz de adjacências de um grafo requer memória $\Theta(|V|^2)$, independente do número de arestas
- Dado que um em um grafo não-direcionado (u,v) e (v,u) representam a mesma aresta, a matriz de adjacências de um grafo não direcionado é igual à sua transposta: $A = A^T$

Estruturas de Dados

■ Matriz de Adjacências

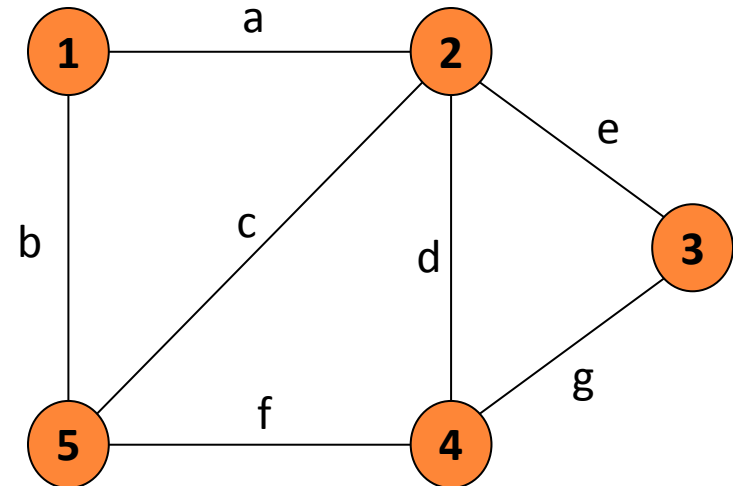
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0



Estrutura de Dados

■ Matriz de Adjacências com Pesos

	1	2	3	4	5
1	0	a	0	0	b
2	a	0	e	d	c
3	0	e	0	g	0
4	0	d	g	0	f
5	b	c	0	f	0



Arestas que não existem podem ser representadas com o valor 0 ou ∞

Estruturas de Dados

■ Matriz de Adjacências

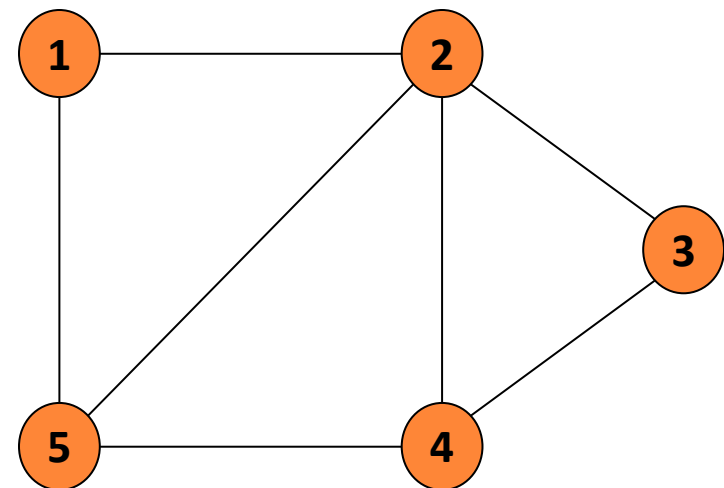
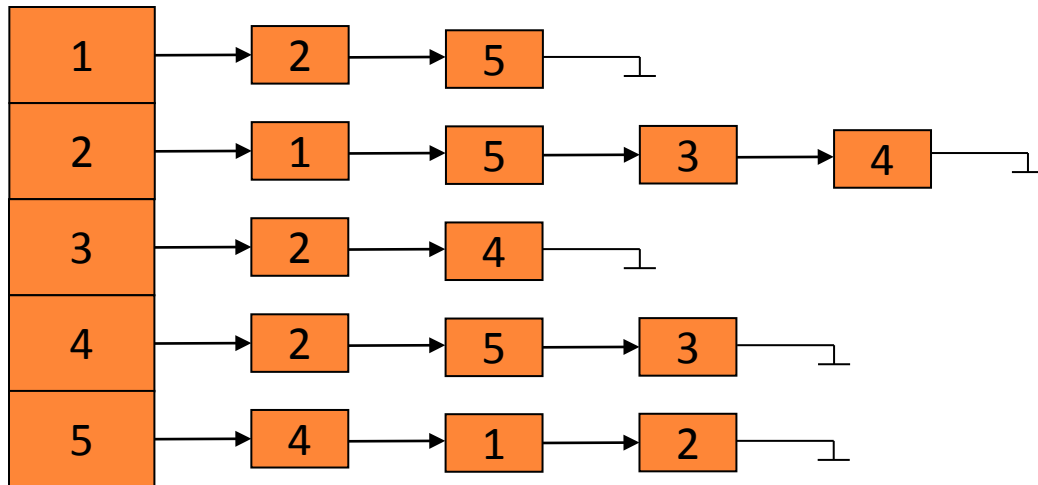
- Exige bastante memória $O(|V|^2)$, pois geralmente os grafos são esparsos
 - Grafos esparsos levam a matrizes esparsas
- O espaço pode ser reduzido armazenando-se apenas a matriz triangular
- Outra forma de redução no armazenamento é a utilização do algoritmo de Zollenkopf

Estruturas de Dados – Lista de adjacências

- Representação da lista de adjacências de um grafo $G = (V, E)$ consiste de um vetor de $|V|$ listas, uma para cada vértice em V
- Para cada $u \in V$, a lista de adjacências $Adj[u]$ contém ponteiros para todos os vértices v em que exista uma aresta $(u, v) \in E$
 - Se G é um grafo direcionado, a soma dos tamanhos de todas as listas de adjacências é $|E|$
 - Se G for um grafo não-direcionado, a soma dos tamanhos de todas as listas de adjacências é $2|E|$
- Quantidade de memória consumida: $O(\max(V, E)) = O(|V| + |E|)$

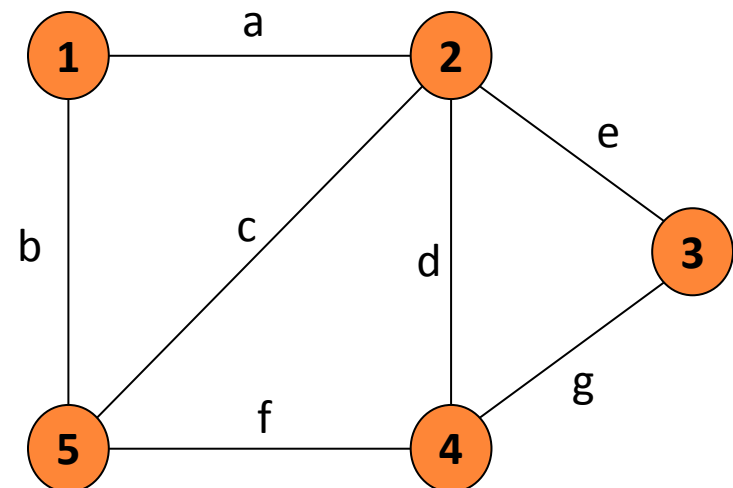
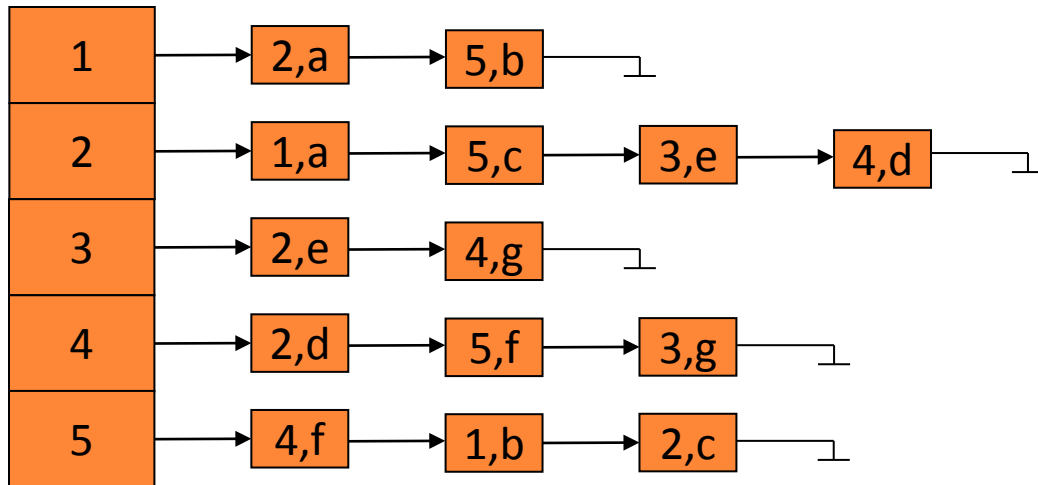
Estruturas de Dados

■ Lista de Adjacências



Estrutura de Dados

■ Lista de Adjacências com Pesos



Estrutura de Dados

■ Lista de Adjacências

- Consome menos memória
- É a forma mais simples
- Se o grafo for orientado a lista de adjacências é menor ainda
- No pior caso a busca por uma adjacência é $\Theta(n)$

Estrutura de Dados

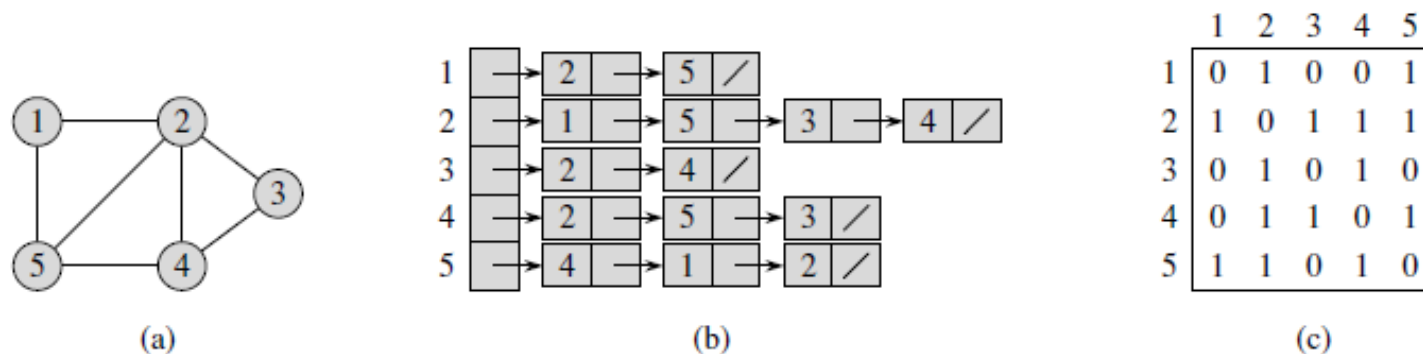


Figure 22.1 Two representations of an undirected graph. (a) An undirected graph G having five vertices and seven edges. (b) An adjacency-list representation of G . (c) The adjacency-matrix representation of G .

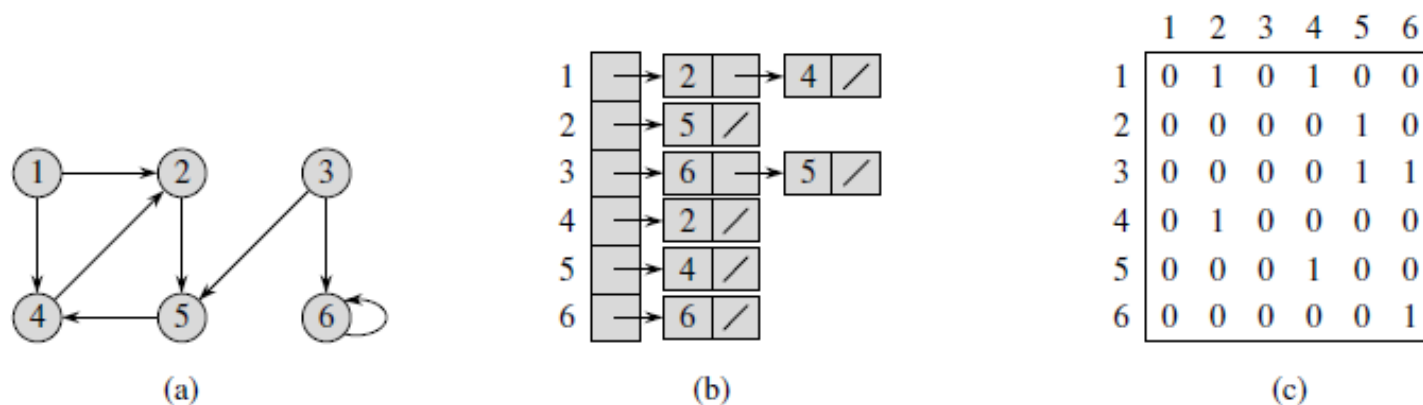
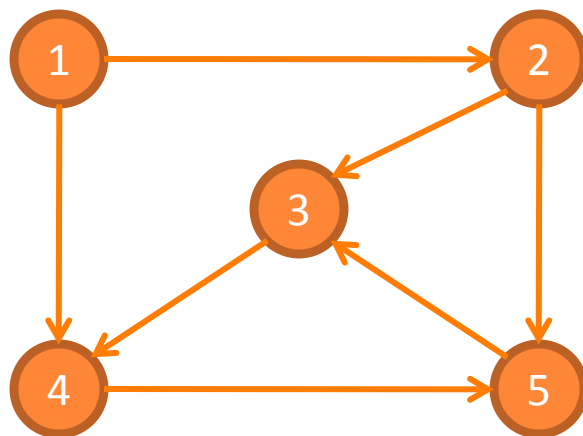


Figure 22.2 Two representations of a directed graph. (a) A directed graph G having six vertices and eight edges. (b) An adjacency-list representation of G . (c) The adjacency-matrix representation of G .

Percurso em Grafos

- Um caminho ou percurso será fechado se a última ligação for adjacente ao vértice inicial
- A notação de um caminho é feita usando pares de vértices
- Um percurso no grafo abaixo: $P(1,2,3,4,5) = ((1,2),(2,3),(3,4),(4,5),(5,3))$



Percurso em um Grafo

- Validação de cada vértice ou aresta
- Cópia de um grafo ou conversão de um tipo em outro
- Contagem do número de vértices ou arestas
- Determinação de componentes conexas
- Determinação de caminhos entre dois vértices, se existirem
- Eficiência – um vértice não pode ser visitado repetidamente
- Corretude – o percurso deve ser feito de modo que não se perca nada

Percurso em Grafos

- Os vértices devem ser marcados quando visitados pela primeira vez
- Cada vértice apresenta três estados
 - Não visitado
 - Visitado
 - Completamente explorado
- Mantém-se uma estrutura de dados com todos os vértices já visitados mas não completamente explorados (Fila ou Pilha)
- Arestas não orientadas podem ser consideradas duas vezes

Exercícios

- Construa o grafo orientado para o percurso $P(1,2,3,4,5,6)=((1,3),(3,2),(2,4),(4,6),(6,5),(5,1))$
- Implemente um programa em C que receba um grafo na forma de matriz e o converta em lista de adjacências.

Busca em Largura

- É um dos algoritmos mais simples
- Dado um grafo $G(V,E)$ e uma origem s , a busca em largura explora sistematicamente as arestas de G até descobrir cada vértice acessível a partir de s
- O algoritmo calcula a distância (menor número de arestas) para todos os vértices acessíveis a partir de s
- *Breadth-first search = BFS*

Busca em Largura

■ Passos

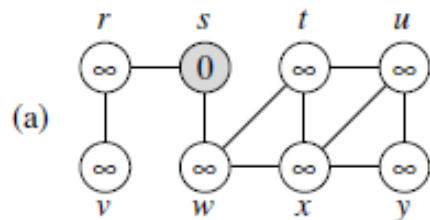
- Escolhe um vértice para o início do caminhamento
- Visita os vértices adjacentes marcando-os como visitados
- Coloca cada um dos vértices em uma fila
- Após visitados os vértices adjacentes, o primeiro da fila se torna o próximo vértice inicial
- Termina quando todos os vértices tenham sido visitados ou o vértice procurado seja encontrado

Algoritmo

BFS(G, s)

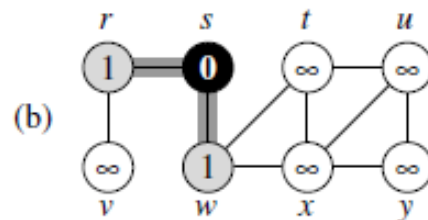
```
1  for each vertex  $u \in V[G] - \{s\}$ 
2      do  $color[u] \leftarrow \text{WHITE}$ 
3           $d[u] \leftarrow \infty$ 
4           $\pi[u] \leftarrow \text{NIL}$ 
5   $color[s] \leftarrow \text{GRAY}$ 
6   $d[s] \leftarrow 0$ 
7   $\pi[s] \leftarrow \text{NIL}$ 
8   $Q \leftarrow \emptyset$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11     do  $u \leftarrow \text{DEQUEUE}(Q)$ 
12         for each  $v \in Adj[u]$ 
13             do if  $color[v] = \text{WHITE}$ 
14                 then  $color[v] \leftarrow \text{GRAY}$ 
15                      $d[v] \leftarrow d[u] + 1$ 
16                      $\pi[v] \leftarrow u$ 
17                     ENQUEUE( $Q, v$ )
18      $color[u] \leftarrow \text{BLACK}$ 
```


Algoritmo



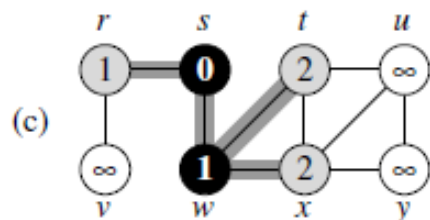
Q

s
0



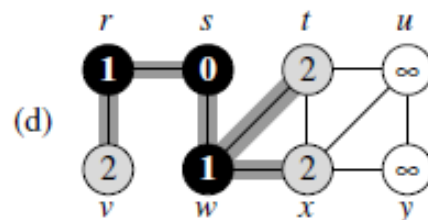
Q

w	r
1	1



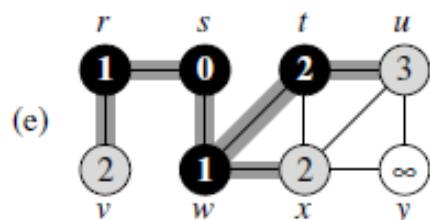
Q

r	t	x
1	2	2



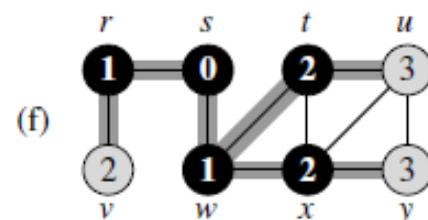
Q

t	x	v
2	2	2



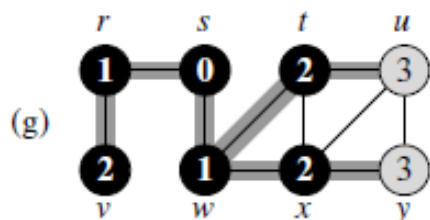
Q

x	v	u
2	2	3



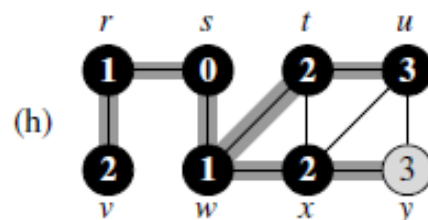
Q

v	u	y
2	3	3



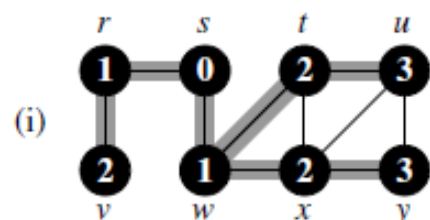
Q

u	y
3	3



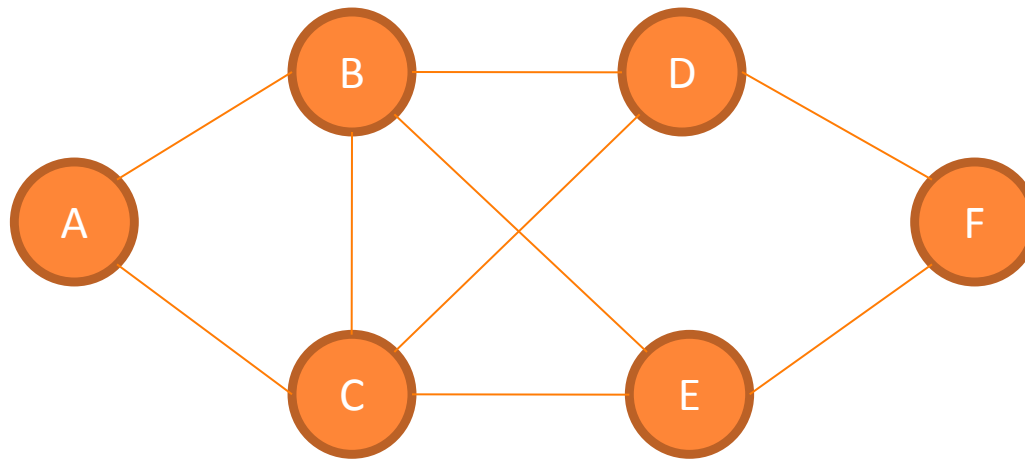
Q

y
3



Q \emptyset

Exemplo



Q	A	A	Q	E	D
Q	B	C	Q	E	F
Q	C	B	Q	F	E
Q	C	D	E	Q	F
Q	D	E	C		

Menor caminho sobre BFS

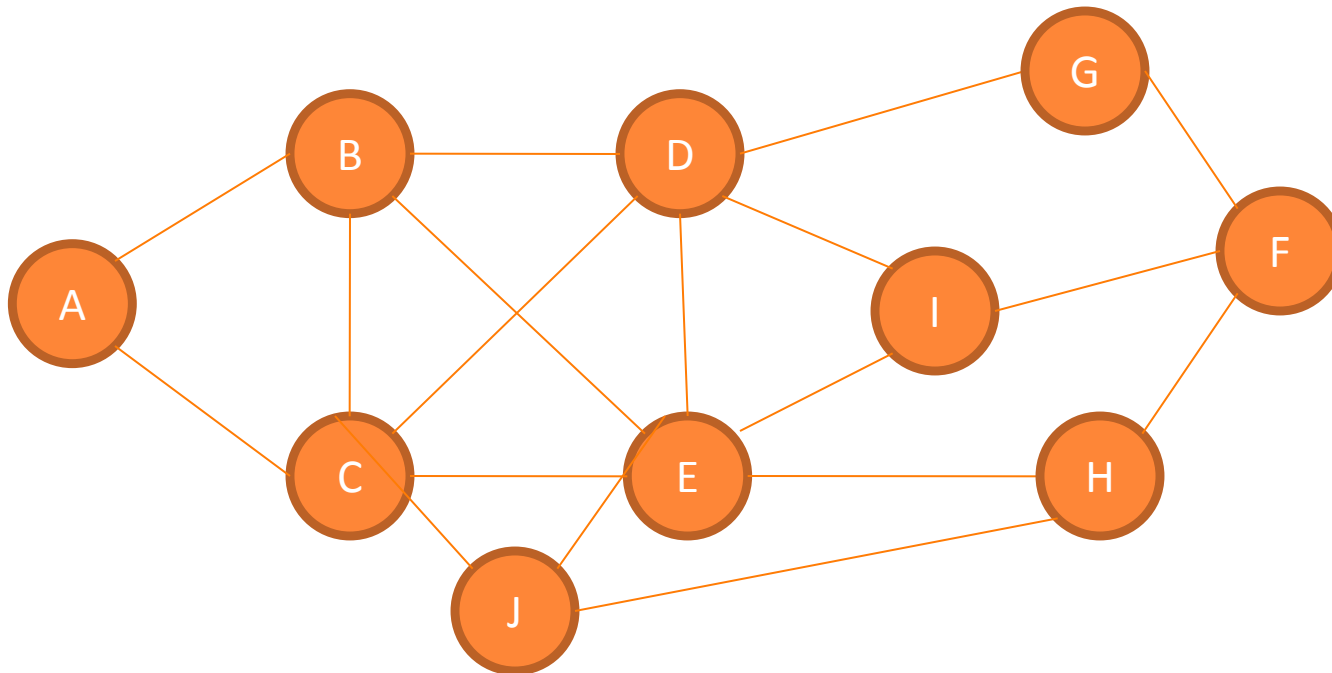
- O seguinte procedimento imprime os vértices do caminho mais curto de s à v , assumindo que o BFS já executou previamente (tempo linear):

PRINT-PATH(G, s, v)

```
1  if  $v = s$ 
2      then print  $s$ 
3      else if  $\pi[v] = \text{NIL}$ 
4          then print “no path from”  $s$  “to”  $v$  “exists”
5          else PRINT-PATH( $G, s, \pi[v]$ )
6          print  $v$ 
```

Exercício

- Dado o grafo a seguir qual o maior comprimento que a fila alcança e quais vértices fazem parte da fila...
 - Partindo de A até F
 - Partindo de A até H



Implementação

- Implementar em C um programa que leia um grafo qualquer de um arquivo e verifique se todos os vértices são acessíveis a partir de um vértice de origem qualquer (fornecido).

Busca em Profundidade

- A estratégia seguida pela busca em profundidade é a de procurar o mais profundamente o possível no grafo
- *Depth-first search = DFS*
- Arestas são exploradas a partir do vértice v descoberto mais recentemente
 - Quando todas as arestas de v foram exploradas, a busca “retroage” para explorar as outras arestas que deixam o vértice a partir do qual v foi descoberto
 - Processo continua até que tenham sido descobertos todos os nós que são alcançáveis a partir da origem

Busca em Profundidade

- Enquanto o sub-grafo de predecessores (vetor π) produzido pelo BFS gera uma árvore, o sub-grafo de predecessores produzido pelo DFS pode ser composto de várias árvores
 - Este sub-grafo é chamado de “depth-first forest”, composta por várias “depth-first trees”
- Como no BFS, os vértices são coloridos durante a busca para indicar seu estado
 - Cada vértice é inicialmente branco, é pintado de cinza quando é descoberto na busca e é pintado de preto quando a busca é finalizada
- Além de criar uma depth-first forest, o DFS também pode atribuir *timestamps* a cada vértice
 - Cada vértice tem 2 timestamps:
 - O primeiro $d[v]$ armazenado quando v é descoberto (e pintado de cinza)
 - O segundo $f[v]$ armazenado quando a busca termina (e é pintado de preto)

Busca em Profundidade

DFS(G)

```
1  for each vertex  $u \in V[G]$ 
2      do  $color[u] \leftarrow \text{WHITE}$ 
3           $\pi[u] \leftarrow \text{NIL}$ 
4   $time \leftarrow 0$ 
5  for each vertex  $u \in V[G]$ 
6      do if  $color[u] = \text{WHITE}$ 
7          then DFS-VISIT( $u$ )
```

DFS-VISIT(u)

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

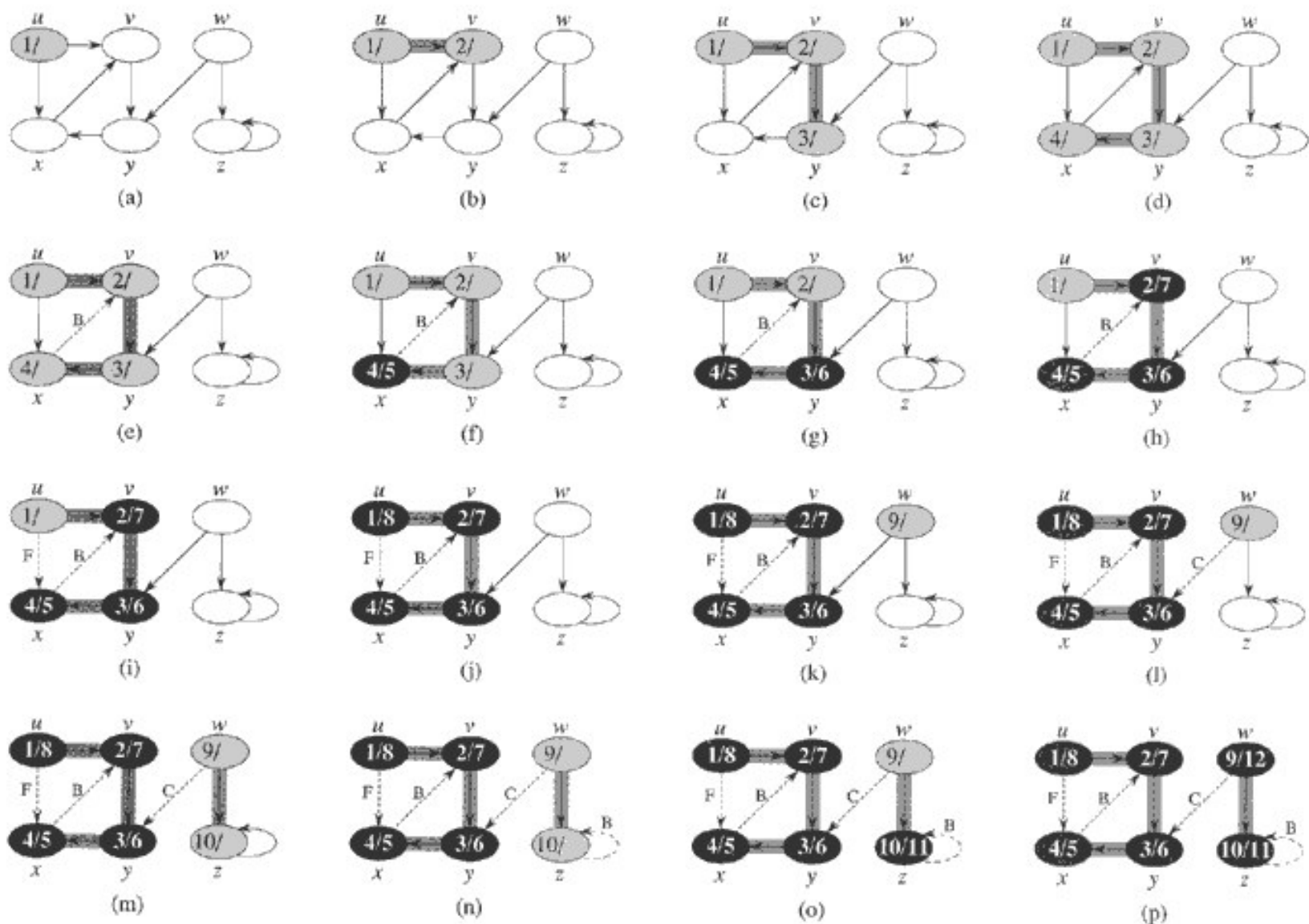
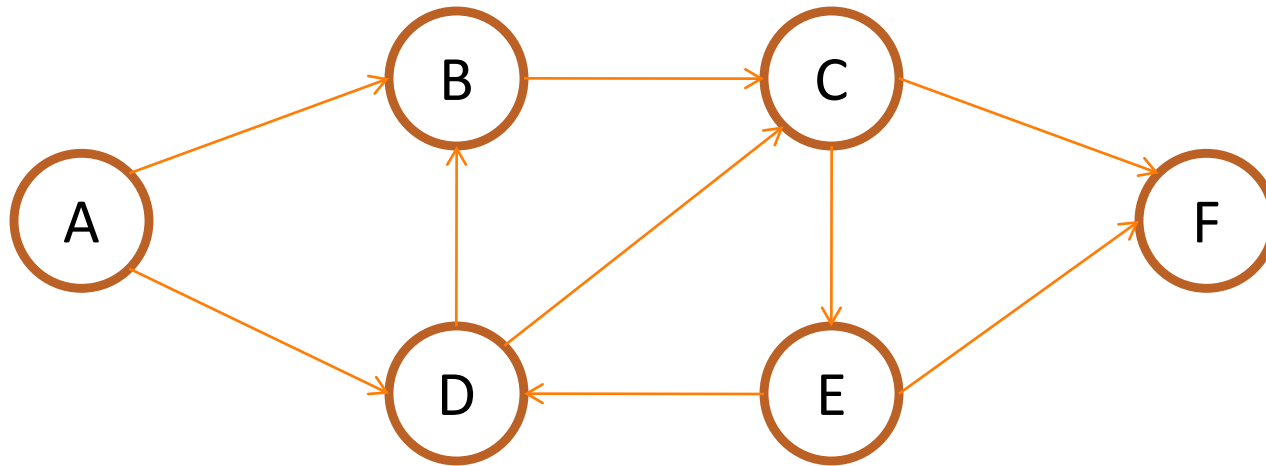



Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time.

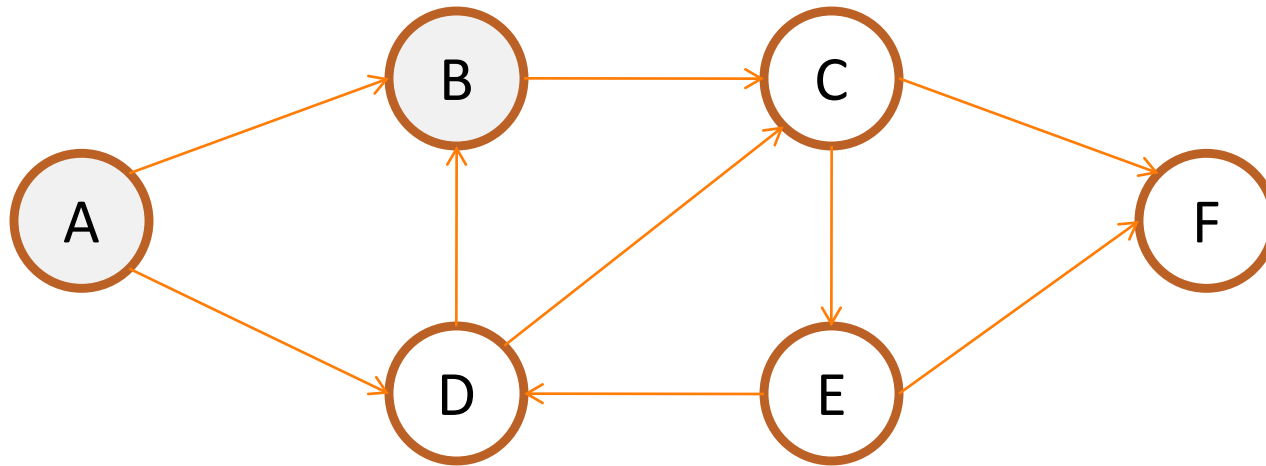
Exemplo



P



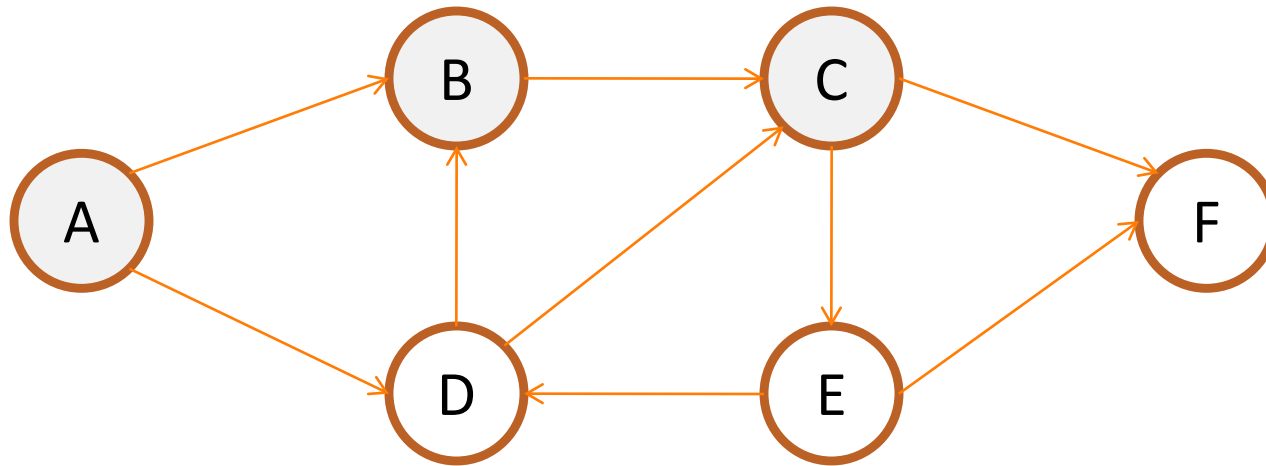
Exemplo



P



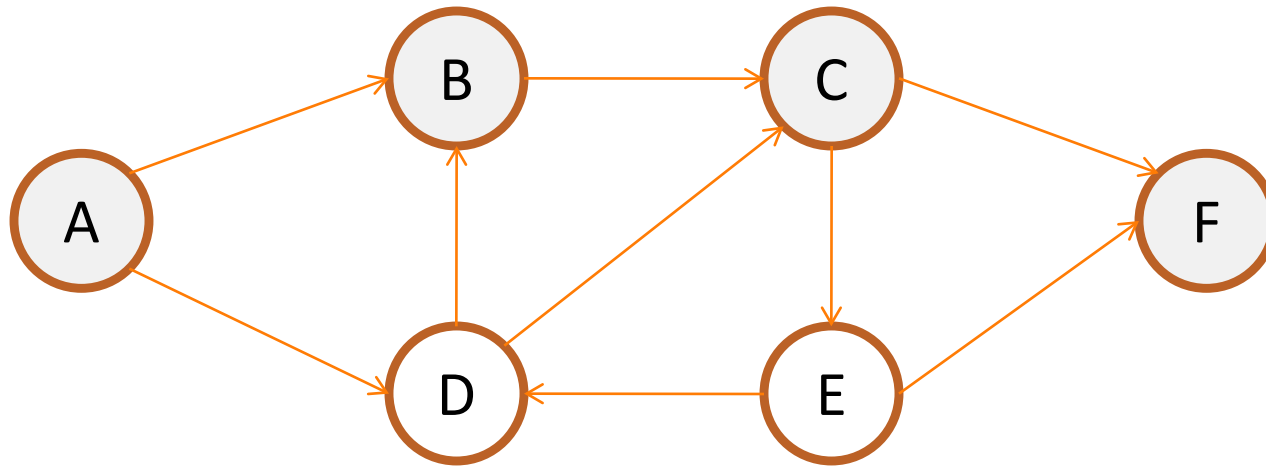
Exemplo



P



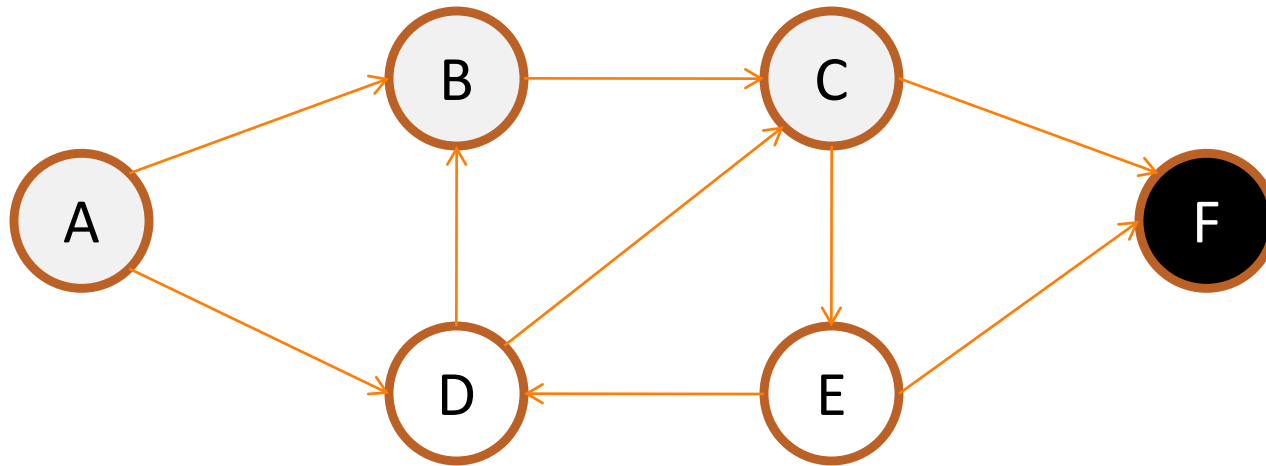
Exemplo



P



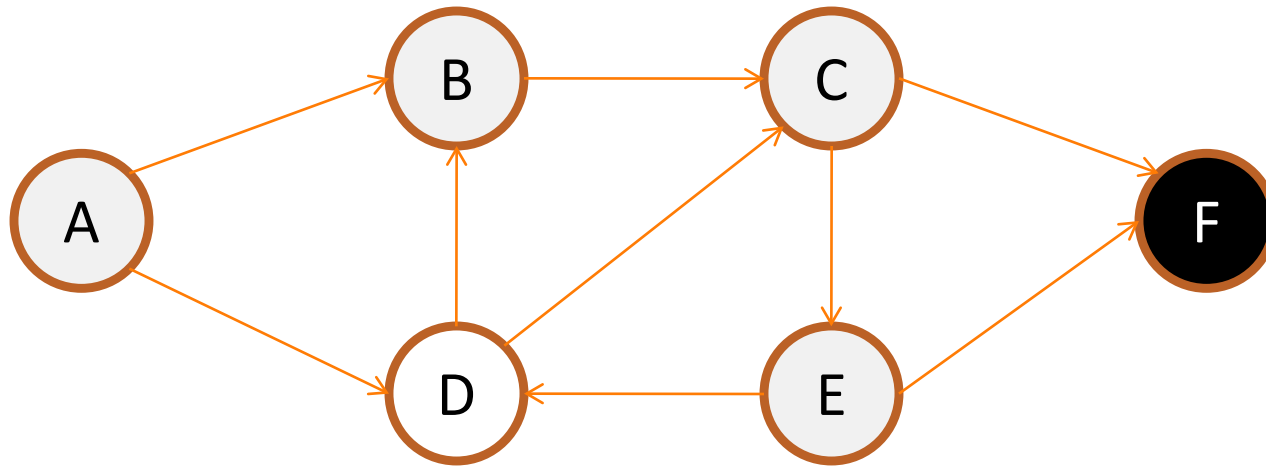
Exemplo



P



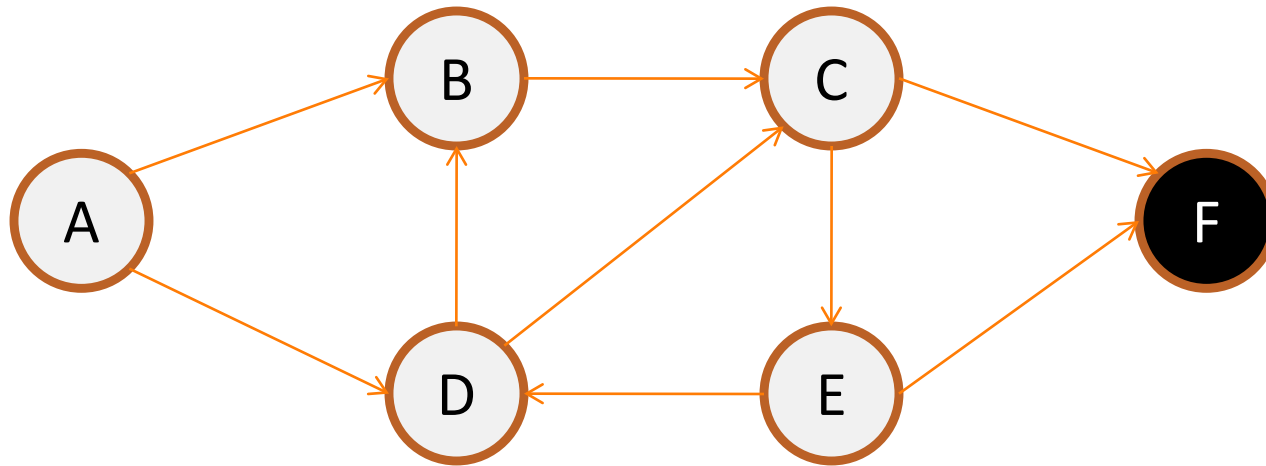
Exemplo



P



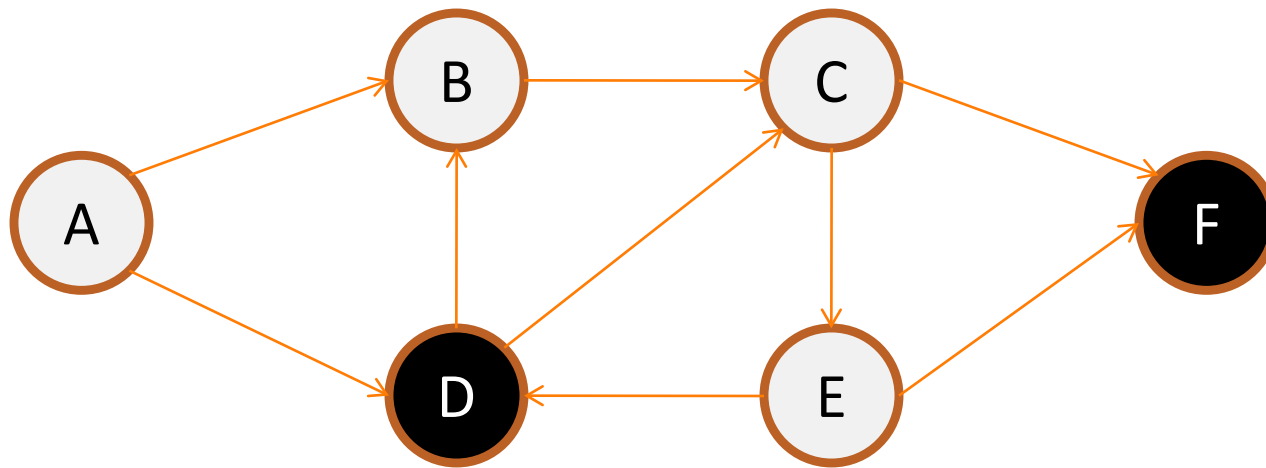
Exemplo



P



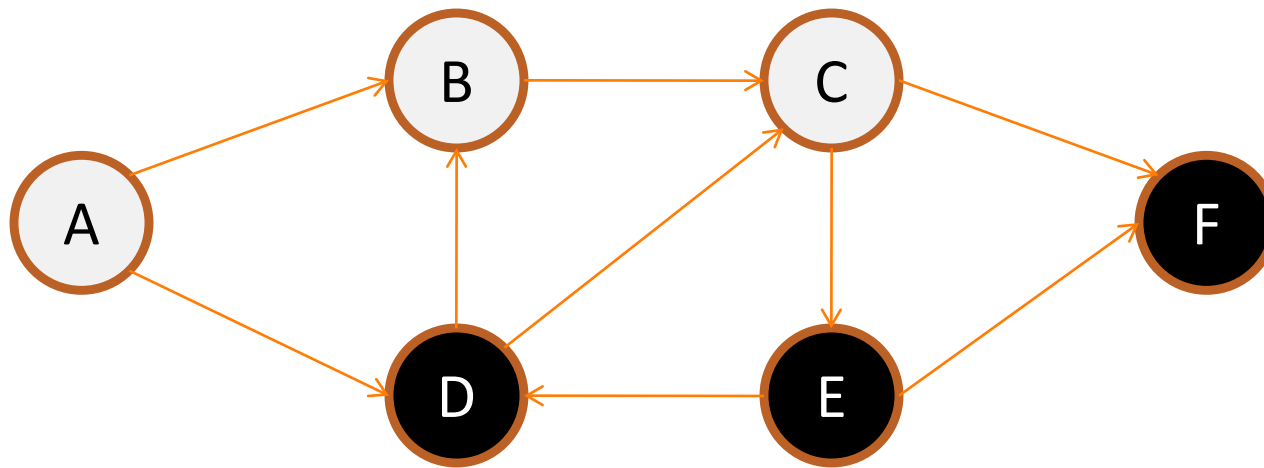
Exemplo



P



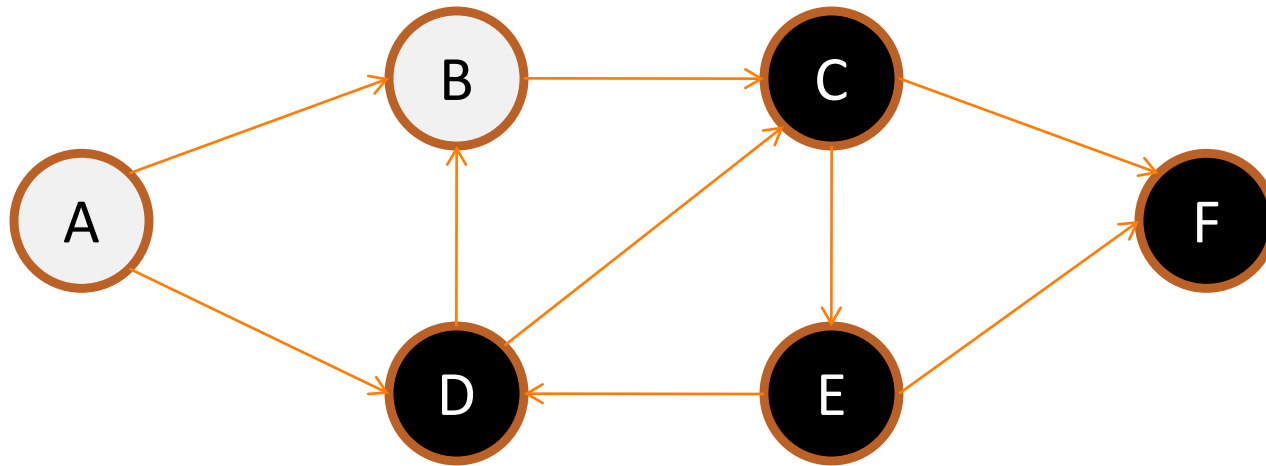
Exemplo



P



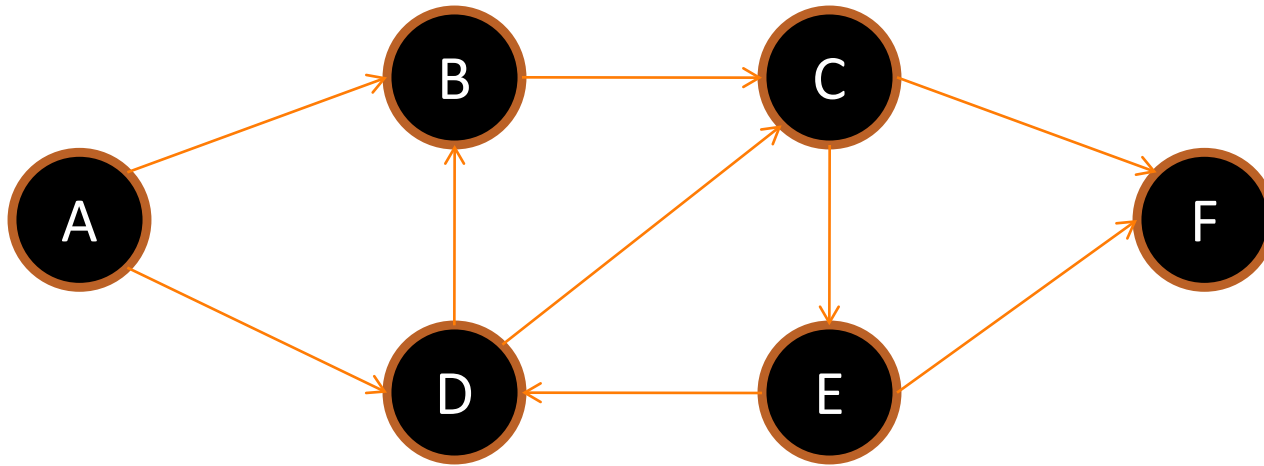
Exemplo



P



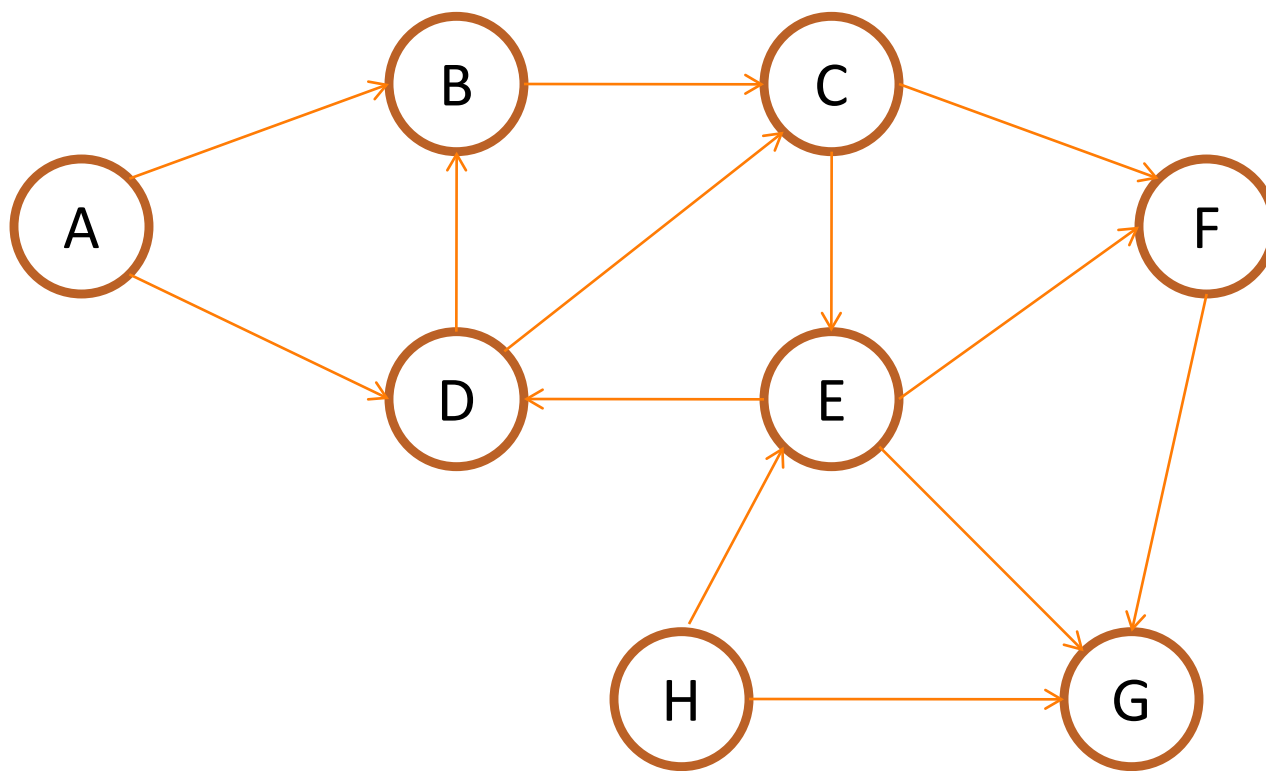
Exemplo



P

Exercícios

- Verifique se os nós H, F e G podem ser visitados a partir do vértice A. Mostre a pilha para provar que o nó pode ser alcançado.



Exercícios

- Implemente um programa em C que leia um grafo de um arquivo e verifique se um nó pode ser alcançado a partir de um nó aleatório. Utilize o mesmo arquivo utilizado no algoritmo de busca em largura.

Topological Sort

- Algoritmo que gera a “ordem topológica” de DAGs
- A ordem topológica de um DAG $G = (V, E)$ é uma ordenação linear de todos os seus vértices dado que se G contém uma aresta (u, v) , então u deve aparecer antes de v na ordenação
- A ordem topológica de um grafo pode ser vista como uma ordenação de seus vértices ao longo de uma linha horizontal dado que todas as arestas direcionadas vão da esquerda para a direita
- DAGs são comumente utilizados para indicar a precedência entre eventos

Topological Sort

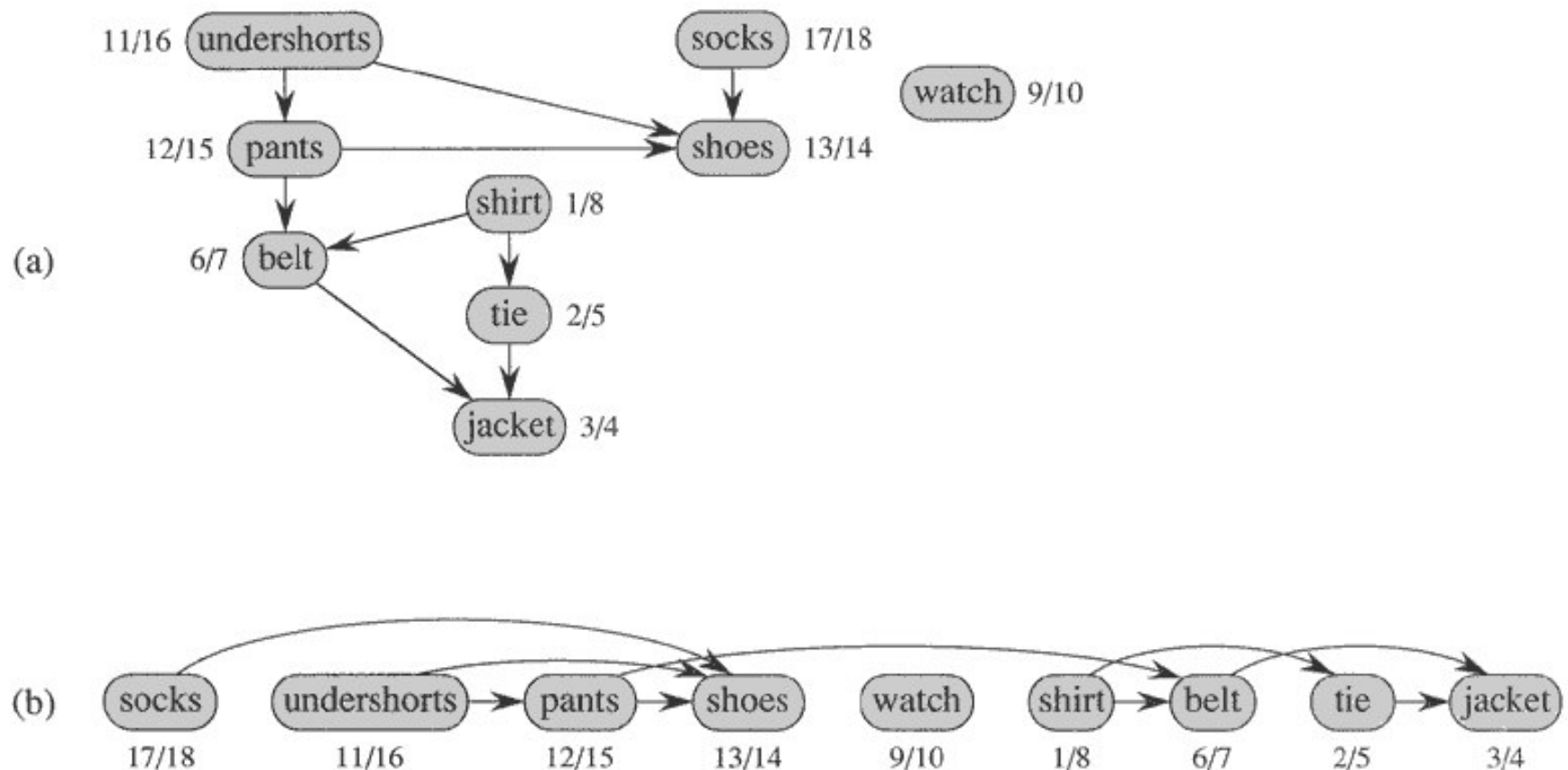


Figure 22.7 (a) Professor Bumstead topologically sorts his clothing when getting dressed. Each directed edge (u, v) means that garment u must be put on before garment v . The discovery and finishing times from a depth-first search are shown next to each vertex. (b) The same graph shown topologically sorted. Its vertices are arranged from left to right in order of decreasing finishing time. Note that all directed edges go from left to right.

Topological Sort

TOPOLOGICAL-SORT(G)

- 1 call DFS(G) to compute finishing times $f[v]$ for each vertex v
- 2 as each vertex is finished, insert it onto the front of a linked list
- 3 **return** the linked list of vertices

Componentes fortemente conectados

- Clássica aplicação do DFS: decompor um grafo direcionado em seus componentes fortemente conectados
- Muitos algoritmos que trabalham com grafos direcionados iniciam com esta decomposição
 - Esta abordagem muitas vezes permite que o problema original seja dividido em sub-problemas, um para cada componente fortemente conectado
- O algoritmo apresentado utiliza a transposta $G^T = (V, E^T)$, em que $E^T = (u, v) : (v, u) \in E$. Isto é, E^T consiste em um conjunto de arestas de G com as direções invertidas

Componentes fuertemente conectados

STRONGLY-CONNECTED-COMPONENTS(G)

- 1 call DFS(G) to compute finishing times $f[u]$ for each vertex u
- 2 compute G^T
- 3 call DFS(G^T), but in the main loop of DFS, consider the vertices
in order of decreasing $f[u]$ (as computed in line 1)
- 4 output the vertices of each tree in the depth-first forest formed in line 3 as a
separate strongly connected component

Componentes fuertemente conectados

