Grafos

introdução, estruturas de dados e buscas

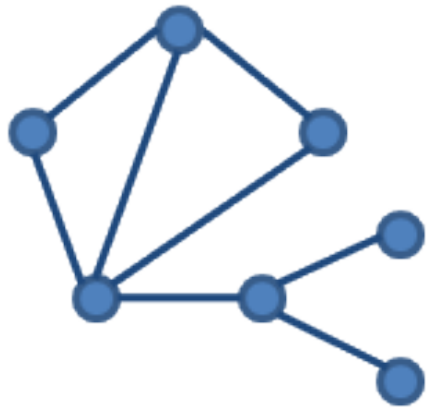
Profs. Andre Gustavo, Salles Magalhaes

Departamento de Informática
Universidade Federal de Viçosa

INF 333 - 2024/1

Grafos

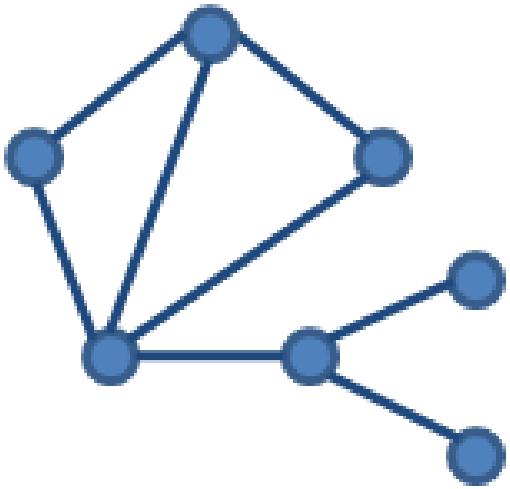
- Conceito muito utilizado em matemática e ciência da computação
- Representa um conjunto de entidades e seus relacionamentos



- O mais importante é saber se existe um relacionamento entre duas entidades
- A forma desse relacionamento é, muitas vezes, irrelevante

Exemplos

- Pode representar um mapa rodoviário



Cidade

Estradas

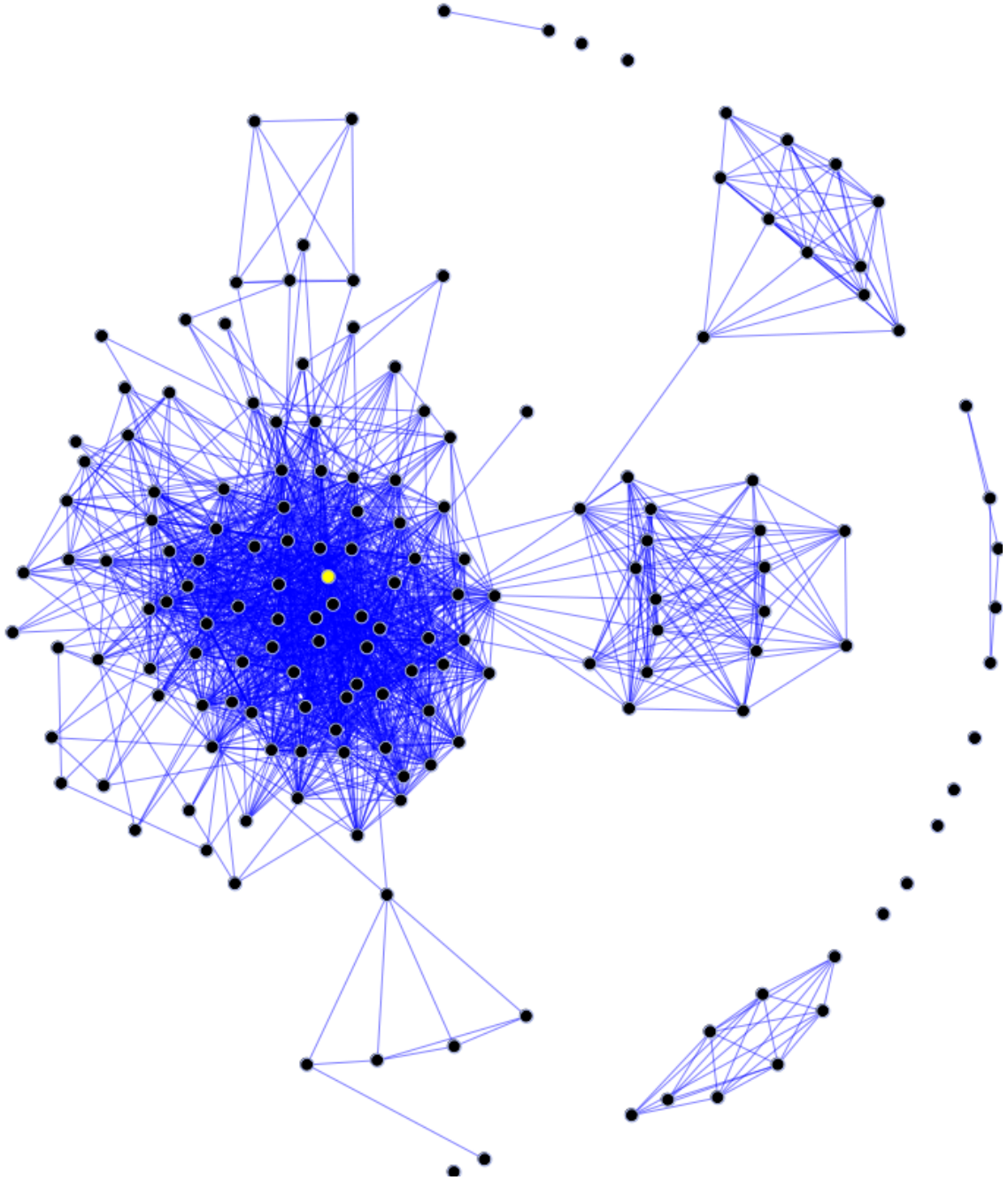
- Também pode representar, entre outros

Rede	Rede social	Campeonato de futebol
Computadores	Pessoas	Times
Cabos	Amizades	Partidas disputadas

Fonte: http://www.metro.sp.gov.br

Mapa de rotas

Fonte: http://www.voeazul.com.br



Fonte: http://en.wikipedia.org/wiki/Social_graph

Grafo

Um grafo é um par ordenado $G = (V, A)$, sendo V um conjunto de vértices e A um conjunto de arestas, cada uma conectando dois vértices $\in V$

Exemplo de Grafo

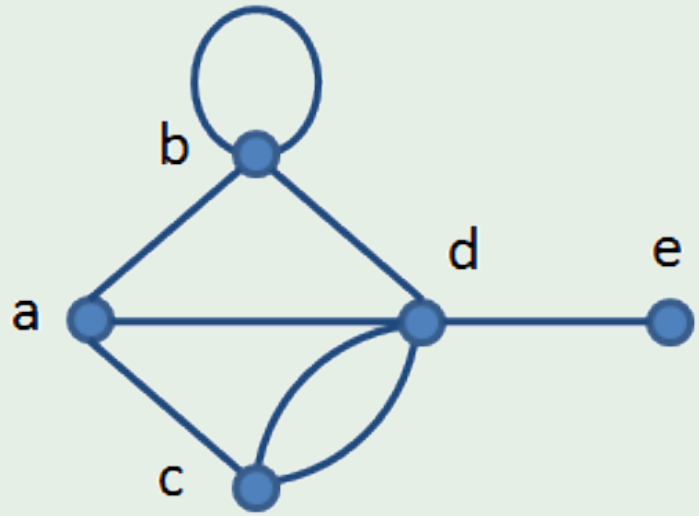
$G = (V, A)$
$V = \{a, b, c, d, e\}$
$A = \{\{a, b\}, \{a, c\}, \{b, b\}, \{a, d\}, \{b, d\}, \{c, d\}, \{c, d\}, \{d, e\}\}$

- Vértices são também chamados nós ou nodos
- Arestas são também chamadas links
- Grafos são também representados por $G = (V, E)$, pois *edges* é o termo mais comum em inglês

- Grafos são geralmente representados graficamente (desenhados)
- Muitas de suas propriedades são mais facilmente entendidas com esse desenho
- Cada vértice é representado por um ponto (círculo) e cada aresta por uma linha

Exemplo de Grafo

$G = (V, A)$
$V = \{a, b, c, d, e\}$
$A = \{\{a, b\}, \{a, c\}, \{b, b\},$
$\{a, d\}, \{b, d\}, \{c, d\},$
$\{c, d\}, \{d, e\}\}$



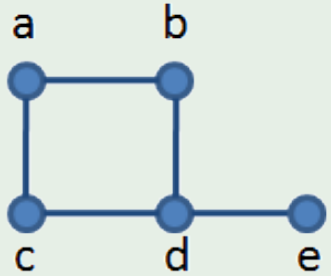
- O desenho de um grafo meramente descreve a relação entre vértices e arestas
- Entretanto, o desenho de um grafo geralmente é chamado de grafo, como se ele mesmo fosse o grafo. E os pontos e linhas chamados de vértices e arestas.
- Não há uma forma única de desenhar um grafo; as posições relativas dos pontos e linhas não tem significado

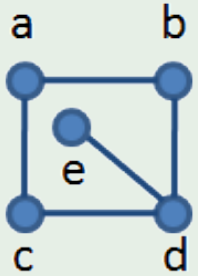
Exemplos de representação gráfica

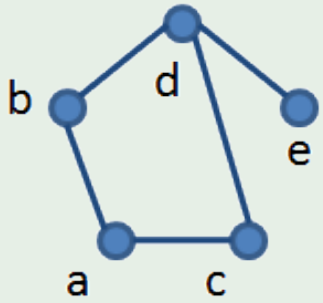
$G = (V, A)$

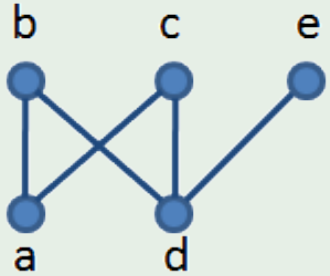
$V = \{a, b, c, d, e\}$

$A = \{\{a, b\}, \{a, c\}, \{b, d\}, \{c, d\}, \{d, e\}\}$





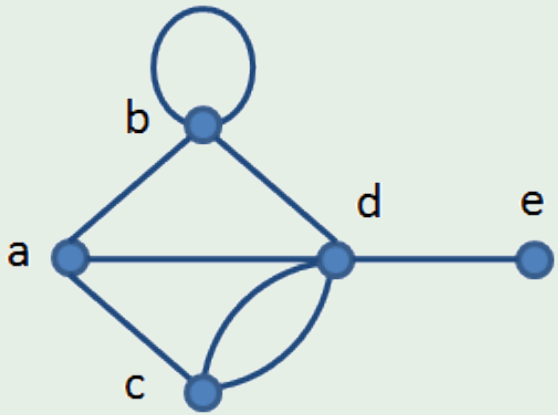




- O número de vértices de um grafo é representado por $|V|$ ou n
- O número de arestas de um grafoé representado por $|A|$ ou m

- Os vértices de uma aresta são ditos incidentes à aresta, e vice-versa
- Dois vértices incidentes a uma aresta em comum são ditos adjacentes
- Uma aresta com vértices incidentes idênticos é chamada loop ou laço
- Múltiplas arestas podem incidir em um mesmo par de vértices

Exemplo de Grafo



- a e b são incidentes a $\{a, b\}$
- a e b são adjacentes
- b e c não são adjacentes
- $\{b, b\}$ é um loop
- há múltiplas arestas $\{c, d\}$

Grafo simples

Um grafo é simples se não tem loops nem arestas múltiplas

- A teoria de grafos estuda principalmente as propriedades de grafos simples

- Para todo grafo G existe uma matriz $|V| \times |A|$ chamada matriz de incidência
- Seja $V = \{v_1, v_2, ..., v_n\}$ e $A = \{a_1, a_2, ..., a_m\}$
- A matriz de incidência de G é a matriz $\mathbf{M}(G) = [m_{ij}]$, sendo m_{ij} o número de vezes (0, 1 ou 2) que o vértice v_i e a aresta a_j são incidentes

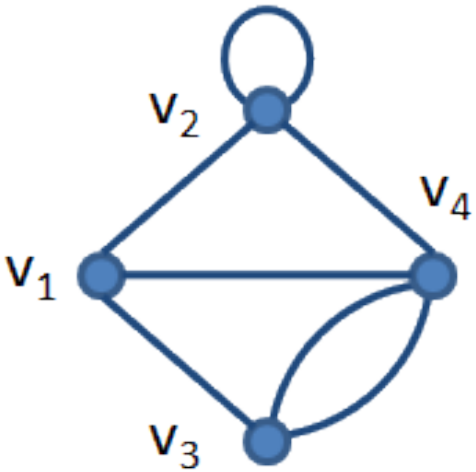


Table: $\mathbf{M}(G)$

	a_1	a_2	a_3	a_4	a_5	a_6	a_7
v_1	1	1	1	0	0	0	0
v_2	1	0	0	1	2	0	0
v_3	0	1	0	0	0	1	1
v_4	0	0	1	1	0	1	1

- Para todo grafo G existe uma matriz $|V| \times |V|$ chamada matriz de adjacência
- Seja $V = \{v_1, v_2, ..., v_n\}$
- A matriz de adjacência de G é a matriz $\mathbf{A}(G) = [a_{ij}]$, sendo a_{ij} o número de arestas unindo os vértices v_i e v_j

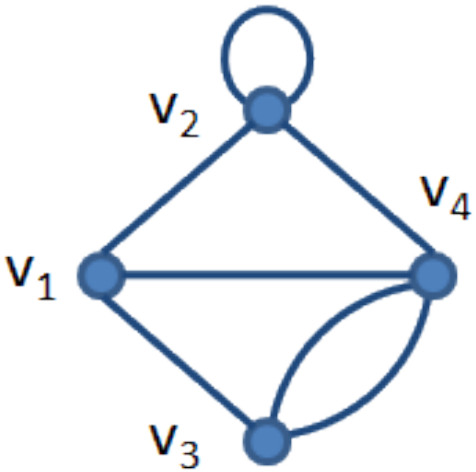


Table: $\mathbf{A}(G)$

	v_1	v_2	v_3	v_4
v_1	0	1	1	1
v_2	1	1	0	1
v_3	1	0	0	2
v_4	1	1	2	0

- A matriz de adjacência é geralmente consideravelmente menor que a matriz de incidência (por que?)
- Por isso a de adjacência é mais usada que a de incidência para armazenar grafos em computadores

Table: **M**(G)

	a_1	a_2	a_3	a_4	a_5	a_6	a_7
v_1	1	1	1	0	0	0	0
v_2	1	0	0	1	2	0	0
v_3	0	1	0	0	0	1	1
v_4	0	0	1	1	0	1	1

Table: **A**(G)

	v_1	v_2	v_3	v_4
v_1	0	1	1	1
v_2	1	1	0	1
v_3	1	0	0	2
v_4	1	1	2	0

- Grafos esparsos são grafos com poucas arestas em relação à quantidade de vértices
- São bastante comuns em situações reais, por exemplo, o grafo das ruas e esquinas de uma cidade
- Para esses grafos, mesmo a matriz de adjacência é um desperdício de memória!
  - A grande maioria das células vale 0
- A estrutura de dados mais adequada é a lista de adjacência

- A lista de adjacência de G é uma lista de listas lineares, uma para cada vértice, contendo seus vértices adjacentes

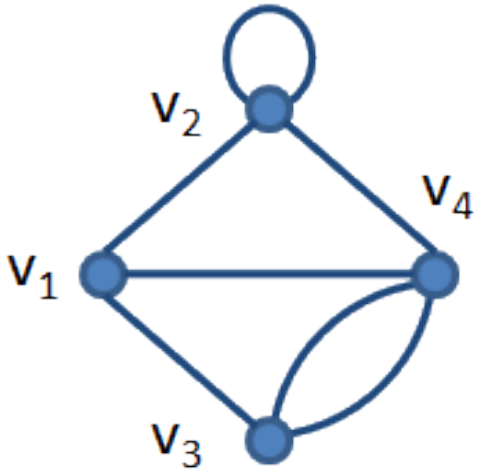


Table: **L**(G)

v_1	v_2	v_3	v_4	
v_2	v_1	v_2	v_4	
v_3	v_1	v_4	v_4	
v_4	v_1	v_2	v_3	v_3

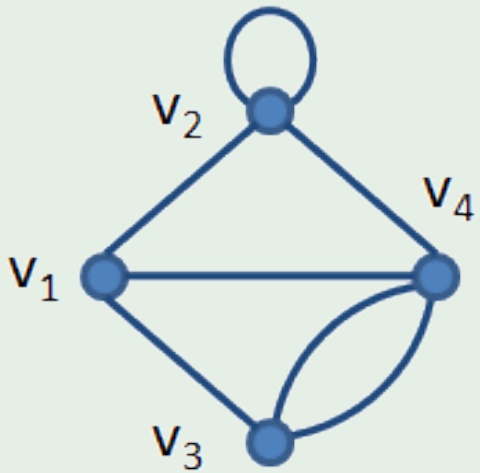
- A lista de adjacêcia de cada vértice pode estar ordenada ou não, pode ser contígua ou encadeada...
- A melhor forma depende da aplicação, do tamanho das listas, da frequência de consulta e de inserção/remoção

Definição

O grau $d_G(v)$ de um vértice v em G é o número de arestas de G incidentes a v, cada loop contando como duas

- Denota-se por $\delta(G)$ e $\Delta(G)$ respectivamente o mínimo e o máximo grau dos vértices de G

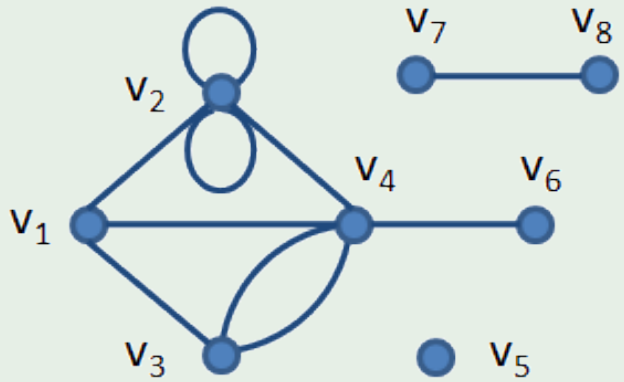
Exemplo



- $d(v_1) = 3$
- $d(v_2) = 4$
- $d(v_3) = 3$
- $d(v_4) = 4$

- $\delta = 3$
- $\Delta = 4$

Outro exemplo



- $d(v_1) = 3$
- $d(v_2) = 6$
- $d(v_3) = 3$
- $d(v_4) = 5$
- $d(v_5) = 0$
- $d(v_6) = 1$
- $d(v_7) = 1$
- $d(v_8) = 1$

- Um vértice de grau 0 é chamado vértice isolado

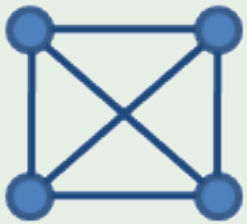
Grafo completo

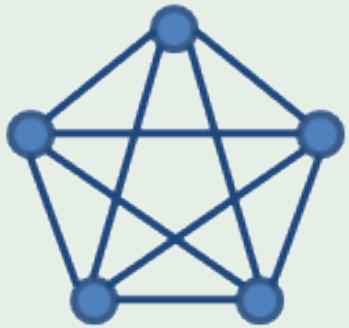
Um grafo completo é um grafo simples em que cada par de vértices é unido por uma aresta

- Sem contar isomorfismo, há apenas um grafo completo com n vértices. Tal grafo é denotado por K_n

Exemplos de grafos completos

K_3 

K_4 

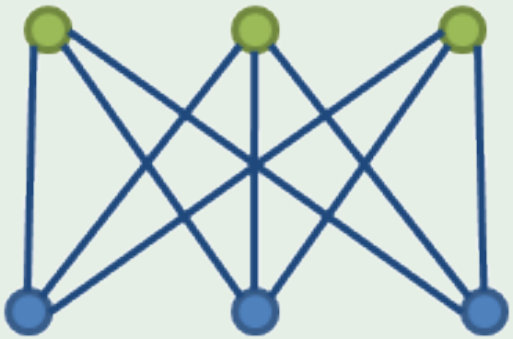
K_5 

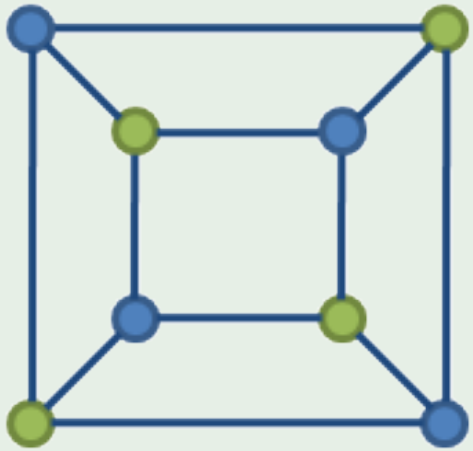
Grafo bipartido

Um grafo bipartido é um cujo conjunto de vértices pode ser dividido em dois subconjuntos X e Y tal que toda aresta seja incidente a vértice em X e a um em Y.

- Ou seja, não há arestas incidentes a vértices do mesmo subconjunto

Exemplos de grafos bipartidos



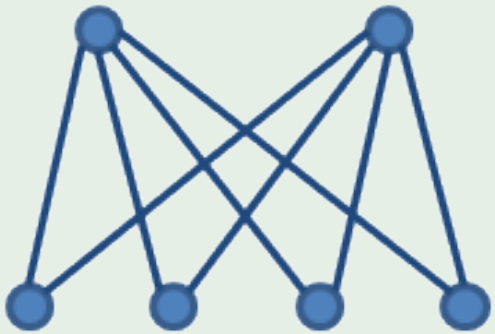


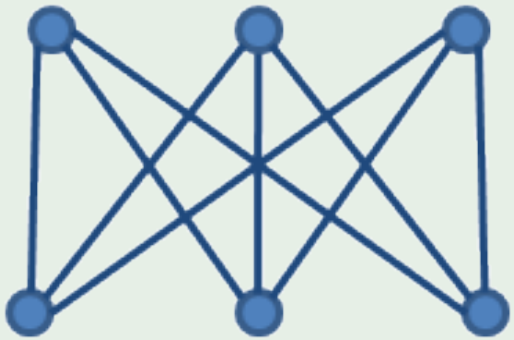
Grafo bipartido completo

Um grafo bipartido completo é um grafo simples, bipartido, com uma bipartição (X, Y) tal que todo vértice de X é adjacente a todo vértice de Y

- Um grafo bipartido completo com $|X| = m$ e $|Y| = n$ é denotado por $K_{m,n}$

Exemplos de grafos bipartidos completos

$K_{2,4}$ 

$K_{3,3}$ 

Busca em grafo

- Percorrer sistemática e completamente um grafo
- Visitar todo vértice e toda aresta numa ordem bem definida e sem repetições desnecessárias

- Percorrer/visitar nesse caso não é propriamente achar um caminho no grafo, mas percorrer sua estrutura, processar cada vértice e aresta

Tipos de busca

- Há dois algoritmos básicos de busca em grafos
  - Busca em profundidade (*DFS - Depth-First Search*)
  - Busca em largura (*BFS - Breadth-First Search*)
- Em alguns problemas não faz diferença qual busca usar
- Em outros, a escolha da busca certa é muito importante

Tipos de busca

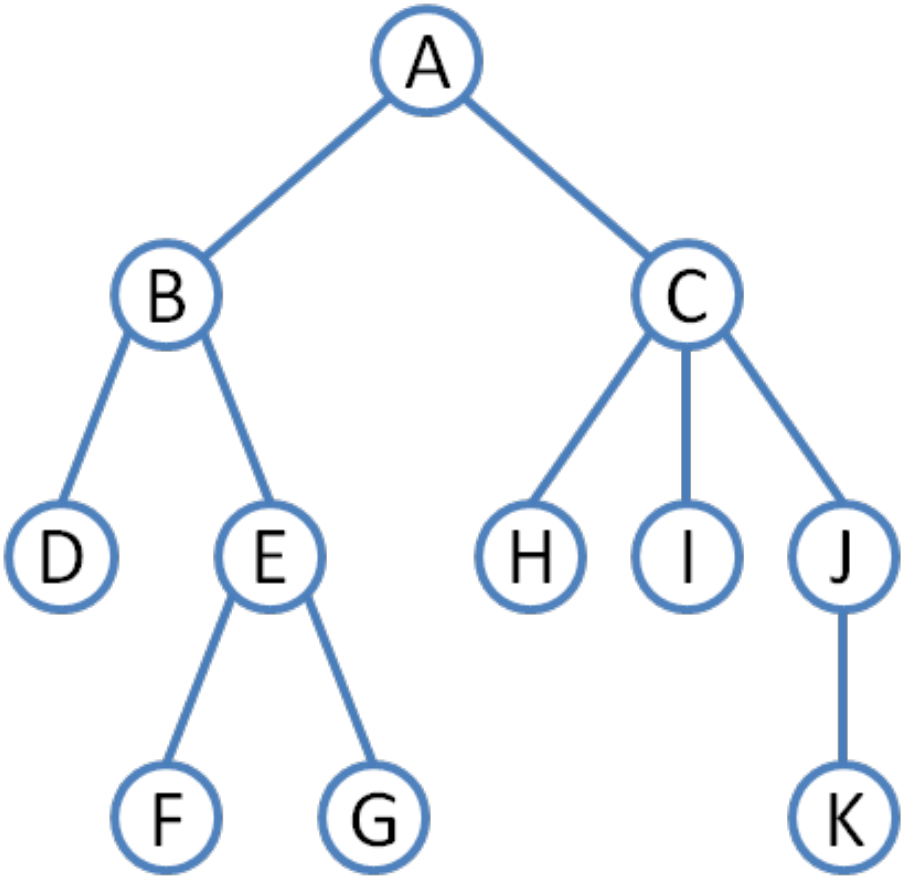
- As duas buscas compartilham uma mesma idéia fundamental
  - Marcar vértices (arestas) já vistos para não explorá-los novamente
  - Do contrário podem se perder num labirinto sem achar saída
- Elas diferem basicamente na ordem de visita aos vértices

Busca em profundidade a partir de um vértice

```
BP (Grafo G, vertice v, bool visitado[]) {
    visitado[v] = true
    escrever(v), processar(v), ...
    para cada w adjacente a v
        se !visitado[w]
            BP(G,w,visitado)
}
```

- *G*: grafo a ser percorrido
- *v*: vértice inicial da busca
- *visitado*[]: array de booleanos, inicialmente todos *false*

Exemplo #1



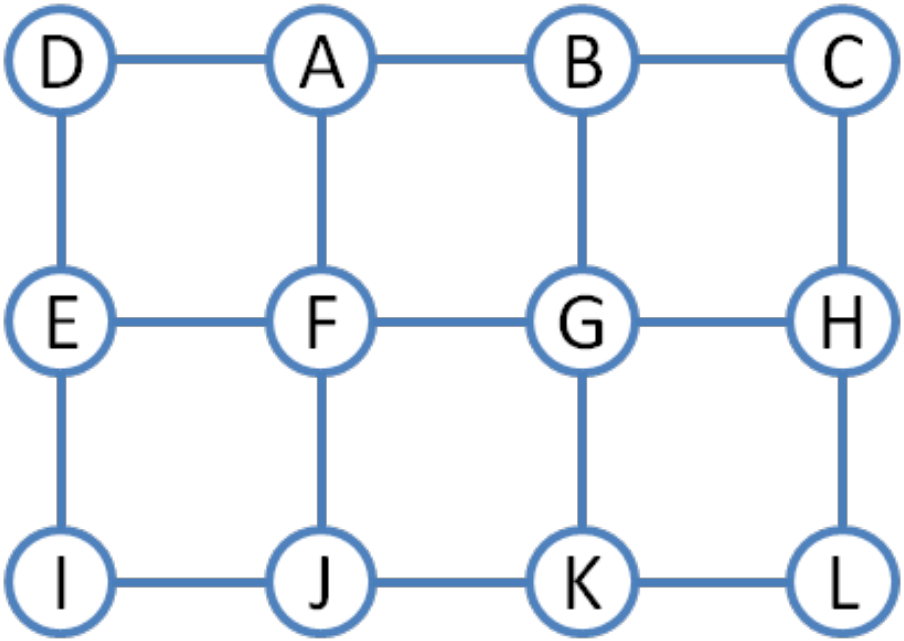
Exemplo #1

v	Adjacentes			
A	B	C		
B	A	D	E	
C	A	H	I	J
D	B			
E	B	F	G	
F	E			
G	E			
H	C			
I	C			
J	C	K		
K	J			

```
BP (Grafo G, vertice v, bool visitado[]) {
    visitado[v] = true
    escrever(v), processar(v), ...
    para cada w adjacente a v
        se !visitado[w]
            BP(G,w,visitado)
}
```

- A BP visita os vértices na ordem: A B D E F G C H I J K

Exemplo #2



Exemplo #2

v	Adjacentes			
A	B	D	F	
B	A	C	G	
C	B	H		
D	A	E		
E	D	F	I	
F	A	E	G	J
G	B	F	H	K
H	C	G	L	
I	E	J		
J	F	I	K	
K	G	J	L	
L	H	K		

```
BP (Grafo G, vertice v, bool visitado[]) {
    visitado[v] = true
    escrever(v), processar(v), ...
    para cada w adjacente a v
        se !visitado[w]
            BP(G,w,visitado)
}
```

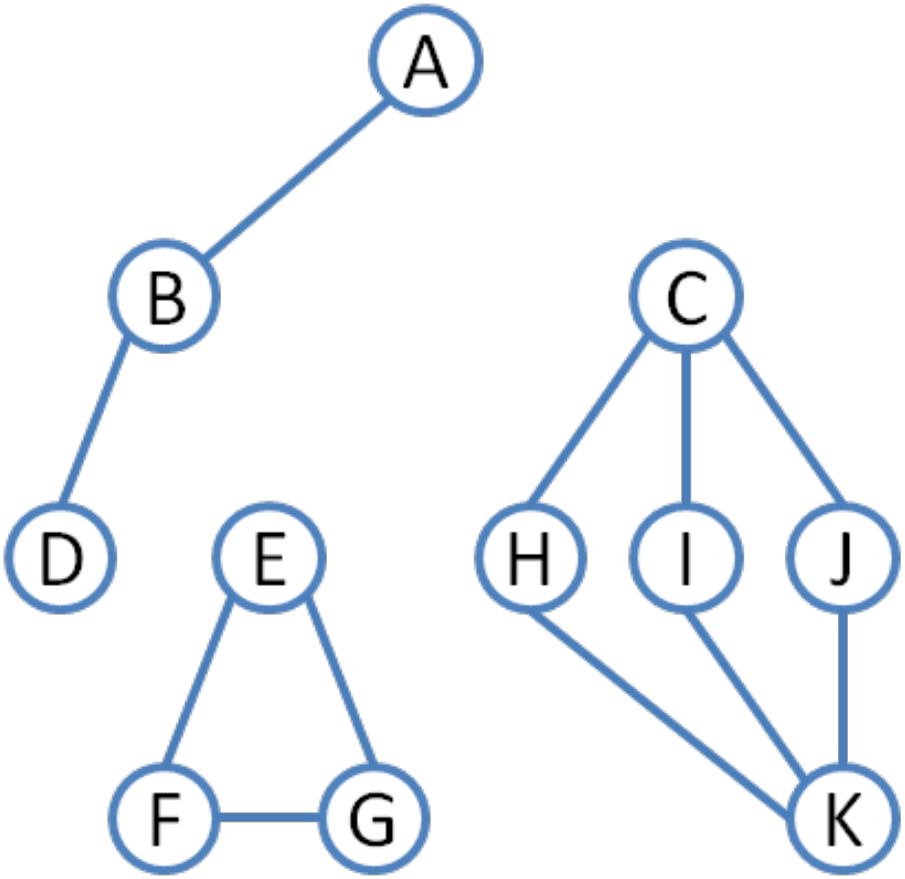
- A BP visita os vértices na ordem: A B C H G F E D I J K L

Busca em profundidade em todo o grafo

```
BuscaProfundidade (Grafo G) {
    para todo vértice v de G
        visitado[v] = false
    para todo vértice v de G
        se !visitado[v]
            BP(G,v,visitado)
}
```

- *G*: grafo a ser percorrido
- enquanto não visitar todos os vértices, faz nova busca a partir de um vértice ainda não visitado (outro componente)

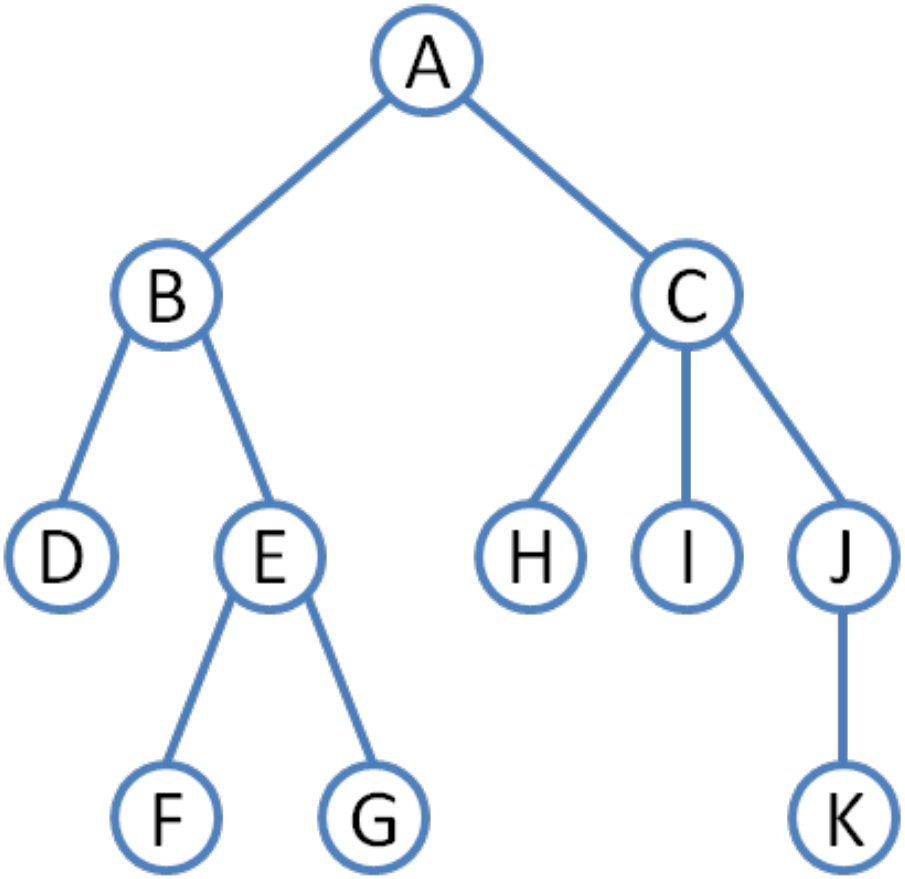
Exemplo #3



Busca em largura

```
BL (Grafo G, vertice v, bool visitado[]) {
    Fila F
    F.insere(v)
    visitado[v] = true
    enquanto !F.vazia()
        v ← F.remove()
        escrever(v), processar(v), ...
        para cada w adjacente a v
            se !visitado[w]
                F.insere(w)
                visitado[w] = true
}
```

Exemplo #1



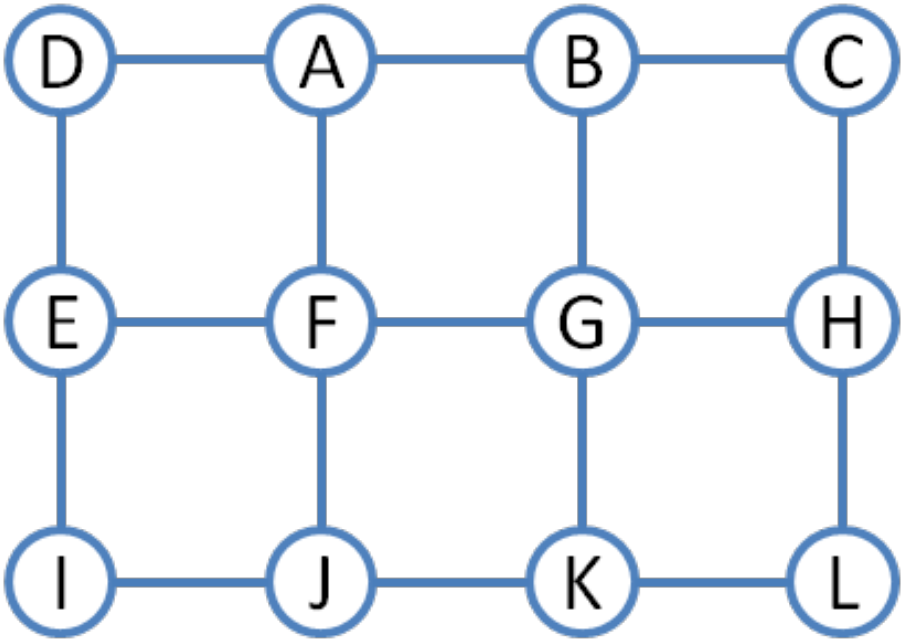
Exemplo #1

v	Adjacentes			
A	B	C		
B	A	D	E	
C	A	H	I	J
D	B			
E	B	F	G	
F	E			
G	E			
H	C			
I	C			
J	C	K		
K	J			

```
BL (Grafo G, vertice v, bool visitado[]) {
    Fila F
    F.insere(v)
    visitado[v] = true
    enquanto !F.vazia()
        v ← F.remove()
        escrever(v), processar(v), ...
        para cada w adjacente a v
            se !visitado[w]
                F.insere(w)
                visitado[w] = true
}
```

- A BL visita os vértices na ordem: A B C D E H I J F G K

Exemplo #2



Exemplo #2

v	Adjacentes			
A	B	D	F	
B	A	C	G	
C	B	H		
D	A	E		
E	D	F	I	
F	A	E	G	J
G	B	F	H	K
H	C	G	L	
I	E	J		
J	F	I	K	
K	G	J	L	
L	H	K		

```
BL (Grafo G, vertice v, bool visitado[]) {
    Fila F
    F.insere(v)
    visitado[v] = true
    enquanto !F.vazia()
        v ← F.remove()
        escrever(v), processar(v), ...
        para cada w adjacente a v
            se !visitado[w]
                F.insere(w)
                visitado[w] = true
}
```

- A BL visita os vértices na ordem: A B D F C G E J H K I L

Semelhanças

- Ambas visitam todos os vértices e todas as arestas
  - BL geralmente é feita para apenas um componente, mas pode ser reiniciada para os demais componentes
- Ambas marcam os vértices visitados para não entrar em loop
- Ambas tem tempo de execução de acordo com o número de vértices e arestas
- Ambas funcionam para grafos construídos *on the fly* (gerado dinamicamente à medida que é percorrido)

Semelhanças (e diferença)

```
BL (Grafo G, vertice v, bool visitado[]) {
    Fila F
    F.insere(v)
    visitado[v] = true
    enquanto !F.vazia()
        v ← F.remove()
        escrever(v), processar(v), ...
        para cada w adjacente a v
            se !visitado[w]
                F.insere(w)
                visitado[w] = true
}
```

Semelhanças (e diferença)

```
BP (Grafo G, vertice v, bool visitado[]) {
    Pilha F
    F.insere(v)
    visitado[v] = true
    enquanto !F.vazia()
        v ← F.remove()
        escrever(v), processar(v), ...
        para cada w adjacente a v
            se !visitado[w]
                F.insere(w)
                visitado[w] = true
}
```

Diferenças

- A ordem de visita dos vértices é diferente
  - Numa árvore, a busca em largura faz a visita por nível
  - Já a busca em profundidade primeiramente vai da raiz à folha
- Uso de memória
  - a busca em largura usa memória adicional para a fila
  - a busca em profundidade usa memória adicional para a pilha (estrutura ou chamadas recursivas)

Diferenças

- A busca em profundidade é essencialmente um *backtracking*
  - Caminhamentos *in-order*, *pre-order* e *pos-order* são exemplos de BP, diferindo somente no local do código em que o vértice é processado
- A busca em largura prioriza a distância do caminho
  - Os vértices são visitados em ordem de distância (em número de arestas) até o vértice inicial
  - Logo, é a busca apropriada para caminho mais curto em grafo sem pesos

Exemplos de aplicação

- As buscas podem ser adaptadas para resolver vários problemas em grafos
- Em alguns casos as mudanças são mínimas, basicamente o que deve ser feito ao visitar um vértice
- Em outros algumas são acrescentados outras funcionalidades e estruturas adicionais

Contar o número de componentes conexos

```
BP (Grafo G, vertice v, bool visitado[]) {
    visitado[v] = true
    escrever(v), processar(v), ...
    para cada w adjacente a v
        se !visitado[w]
            BP(G,w,visitado)
}

BuscaProfundidade (Grafo G) { cont = 0
    para todo vértice v de G
        visitado[v] = false
    para todo vértice v de G
        se !visitado[v]
            BP(G,v,visitado) cont++
}
```

Verificar se há ciclo no grafo

```
BP (Grafo G, vertice v, bool visitado[], vertice pai[]) {
    visitado[v] = true
    escrever(v), processar(v), ...
    para cada w adjacente a v
        se !visitado[w]
            pai[w] = v; BP(G,w,visitado,pai)
        senão se w != pai[v] CICLO!! //vertice já visitado antes
}

BuscaProfundidade (Grafo G) {
    para todo vértice v de G
        visitado[v] = false ; pai[v] = '-'
    para todo vértice v de G
        se !visitado[v]
            BP(G,v,visitado,pai)
}
```

Caminho com menos arestas

- Encontrar caminho com menor número de arestas de v a x
- É também o caminho com menos vértices intermediários

Caminho com menos arestas

```
BL (Grafo G, vertice v, bool visitado[], vertice x) {
    Fila F
    F.insere(v)
    visitado[v] = true
    enquanto !F.vazia()
        v ← F.remove()
        escrever(v), processar(v), ...
        para cada w adjacente a v
            se !visitado[w]
                F.insere(w); pai[w] = v
                visitado[w] = true
                se w == x
                    EscreveCaminho(x,pai); return
}
```

Caminho com menos arestas

- Escreve recursivamente o caminho guardado no vetor ’pai’

```
EscreveCaminho (vertice x, vertice pai[]) {
    se x!=’-’
        EscreveCaminho(pai[x],pai)
        escreve x
}
```

Verificar se é bipartido

- Verificar se um grafo é bipartido
- É o mesmo que verificar se é *2-colorable*

Verificar se é bipartido

```
BP (Grafo G, vertice v, bool visitado[], int cor[]) {
    visitado[v] = true
    escrever(v), processar(v), ...
    para cada w adjacente a v
        se cor[v] == cor[w] NÃO É BIPARTIDO!
        se !visitado[w]
            se cor[v]==1 cor[w]=2 senão cor[w]=1
            BP(G,w,visitado,cor)
}

BuscaProfundidade (Grafo G) {
    para todo vértice v de G
        visitado[v] = false ; cor[v] = 0 //sem cor
    para todo vértice v de G
        se !visitado[v]
            cor[v] = 1; BP(G,v,visitado,cor)
}
```

Exemplos

Exemplo: UVa 10067 - Playing with Wheels (link)

- Um vértice para cada valor (0000 a 9999)
- Aresta se um obtido do outro por 1 movimento
- Apagar vértices proibidos (ou suas arestas)
- Busca em largura do valor inicial ao final

Obs.: não é necessário construir o grafo, pode ser usado implicitamente

Exemplos

Exemplo: UVa 11902 - Dominator (link)

- Fazer uma busca a partir do inicial e marcar os atingidos
- “Apagar” um vértice v e fazer a busca novamente
- Os atingidos anteriormente que não forem atingidos na nova busca são dominados por v

Exemplos

UVa 469 - Wetlands of Florida (link)

- Fazer uma busca a partir da posição visitando `W` adjacentes
- Técnica recursiva conhecida como *flood fill* (inundação)
  - inunda como o “balde” de tinta num editor de imagens