Buscas em Grafos

Notas de aula da disciplina IME 04-11312 (Otimização em Grafos)

Paulo Eustáquio Duarte Pinto
(pauloedp at ime.uerj.br)

março/2021

Grafos - Buscas em Grafos

Idéia: Visitar, a cada passo, algum vizinho não visitado, de vértices já visitados

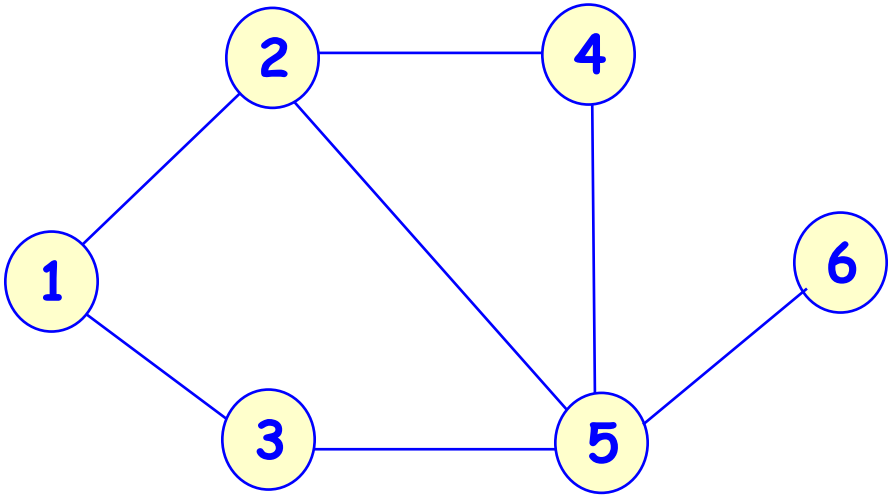
Busca em Profundidade (DFS - Depth First Search)

Busca em Largura (BFS - Breadth First Search)

Busca Irrestrita em Profundidade

Grafos - Busca em Profundidade (DFS)

Idéia: Visitar, a cada passo, algum vizinho não visitado, do vértice mais recentemente visitado



Ex: A sequência 2 - 1 - 3 - 5 - 6 -4 é uma DFS.

Ex: A sequência 2 - 1 - 3 - 4 - 5 -6 NÃO é uma DFS.

Grafos - Busca em Profundidade (DFS)

Idéia: Visitar, a cada passo, algum vizinho não visitado, do vértice mais recentemente visitado

```
BP(u,v):
    marcar v
    para vizinhos w de v:
        se w não marcado:
            BP(v,w)
```

```
desmarcar vértices()
para i ← 1..n incl.:
    se i não marcado:
        BP(i,i)
```

Cada vértice visitado é marcado. Normalmente usa-se um vetor para isso.

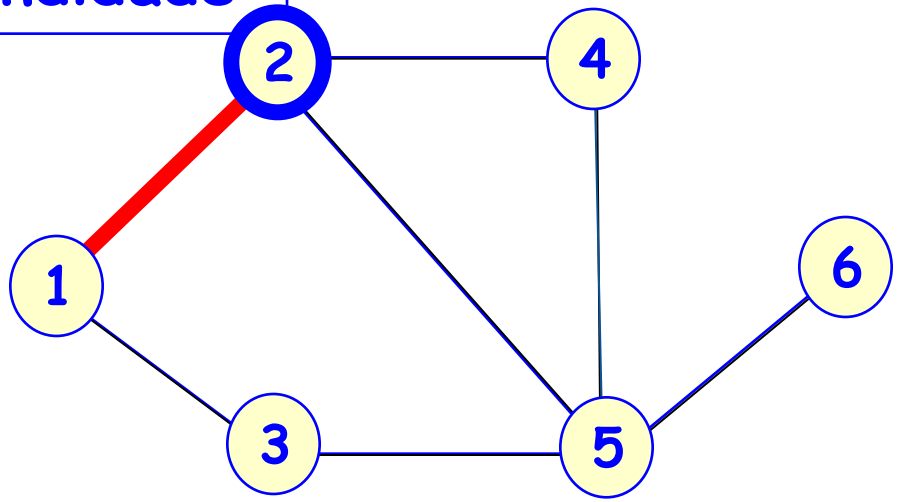
Podem ser usadas várias convenções para marcação:

a) 0/1 (desmarcado/marcado).
b) 0/número do componente (desmarcado/marcado).
c) 0/ordem da visita (desmarcado/marcado).
d) outras opções.

Apenas o parâmetro v de BP(u, v) foi usado nesta versão da busca. Muitas vezes é necessário usar os dois.

Grafos - Busca em Profundidade

Árvores (Floresta) de Profundidade: são as árvores da recursão da busca.

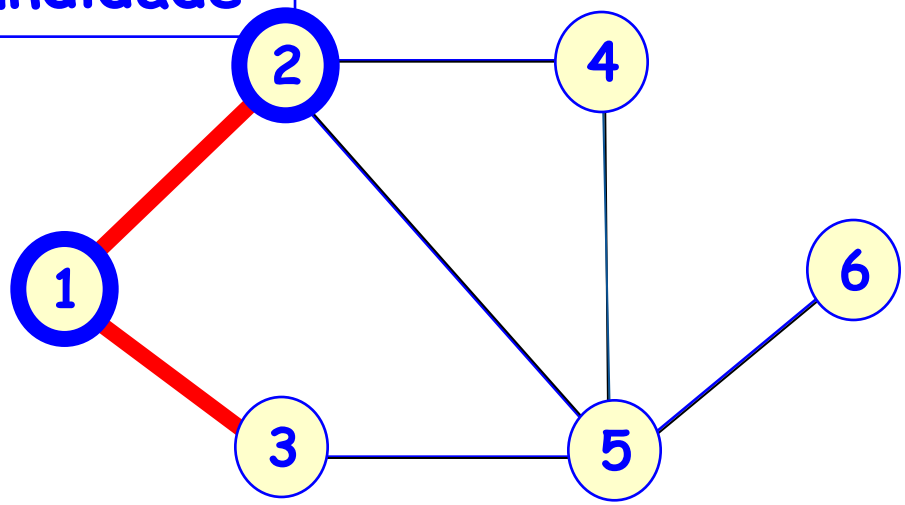


Após BP(2,2):

Árvore de Profundidade:

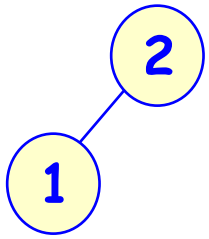
Grafos - Busca em Profundidade

Árvores (Floresta) de Profundidade: são as árvores da recursão da busca.



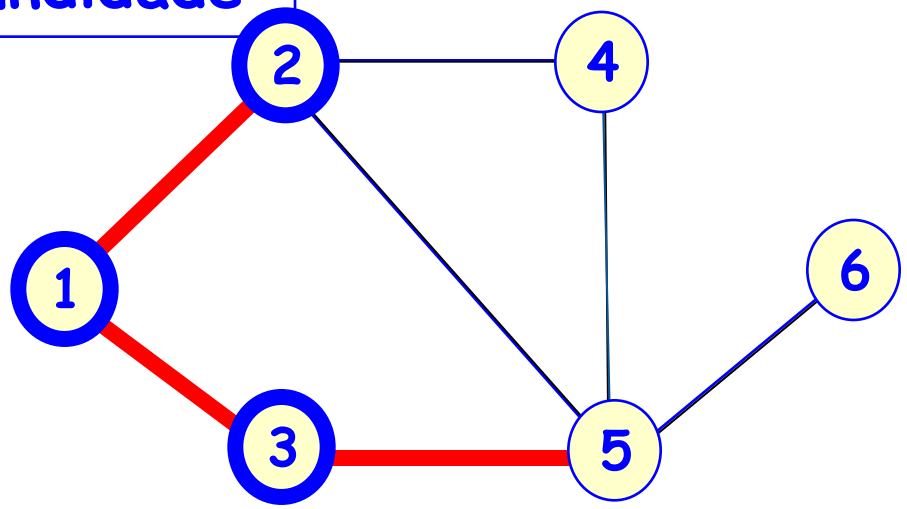
Após BP(2,1):

Árvores de Profundidade:



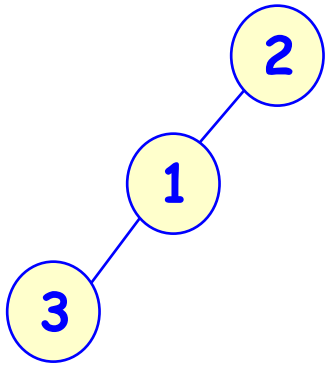
Grafos - Busca em Profundidade

Árvores (Floresta) de Profundidade: são as árvores da recursão da busca.



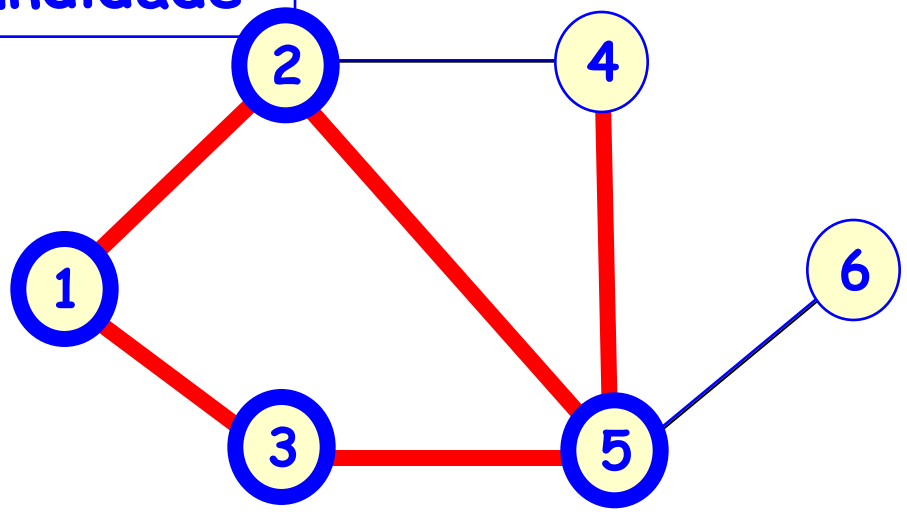
Após BP(1,3):

Árvores de Profundidade:



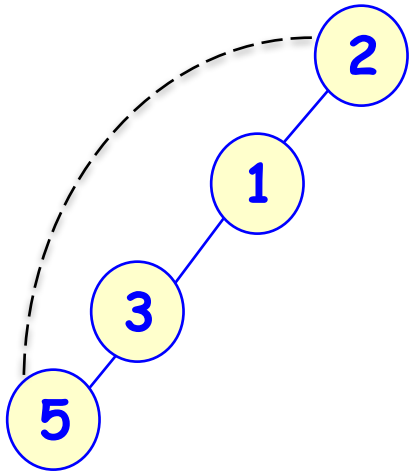
Grafos - Busca em Profundidade

Árvores (Floresta) de Profundidade: são as árvores da recursão da busca.



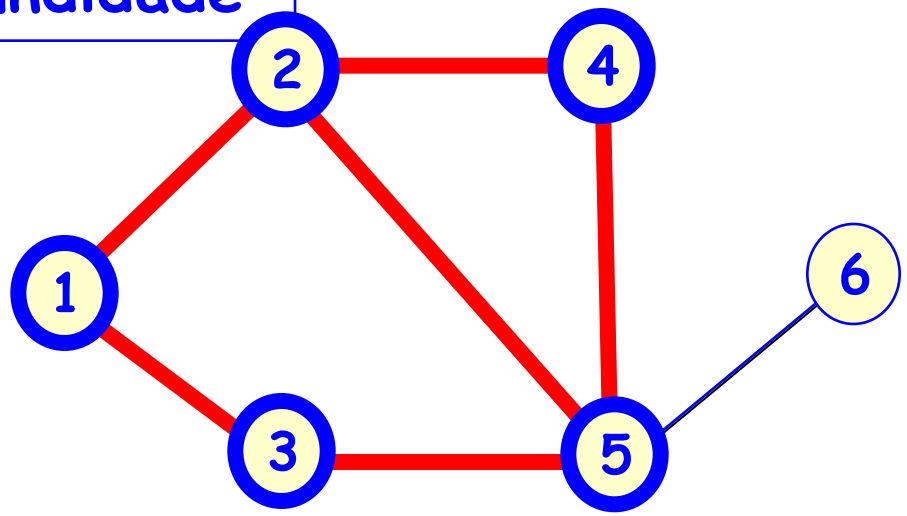
Após BP(3,5):

Árvores de Profundidade:



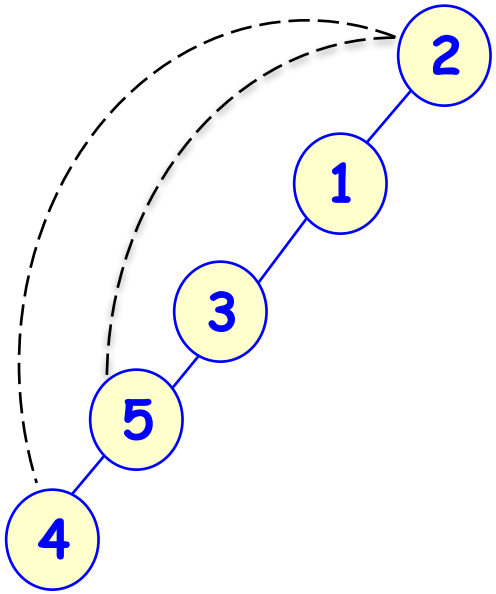
Grafos - Busca em Profundidade

Árvores (Floresta) de Profundidade: são as árvores da recursão da busca.



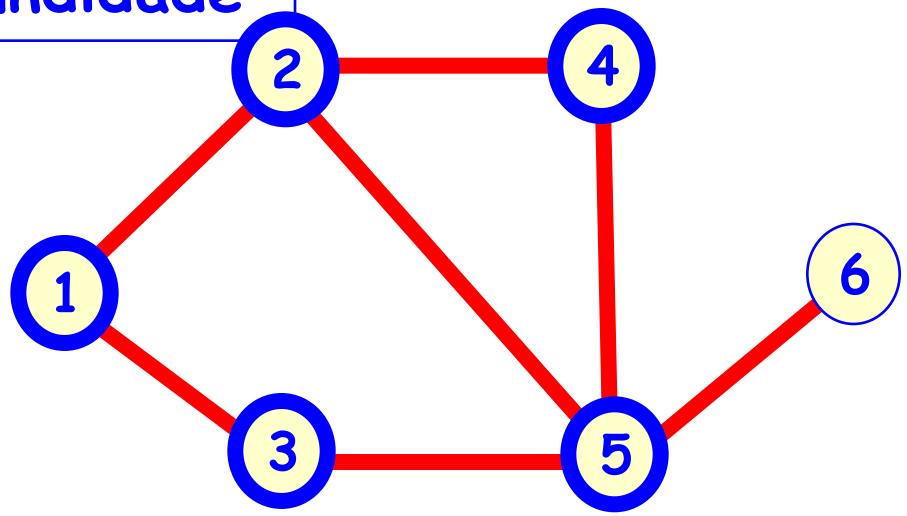
Após BP(5,4):

Árvores de Profundidade:



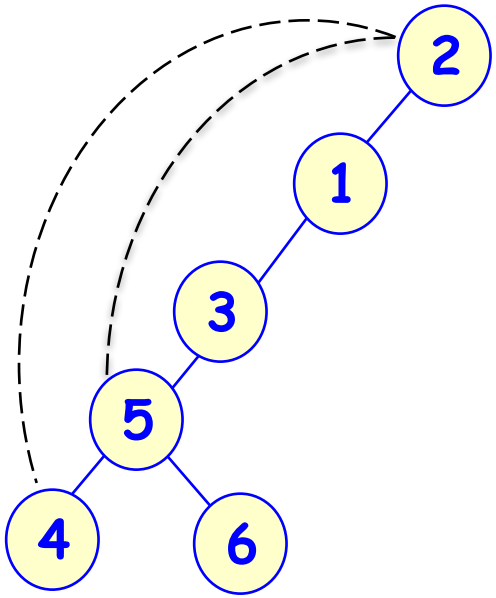
Grafos - Busca em Profundidade

Árvores (Floresta) de Profundidade: são as árvores da recursão da busca.



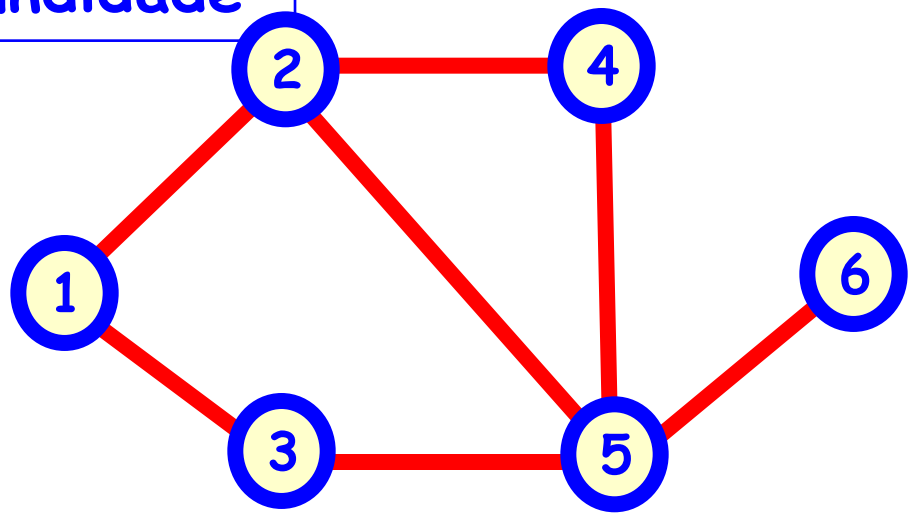
Após o retorno de BP(5,6):

Árvores de Profundidade:



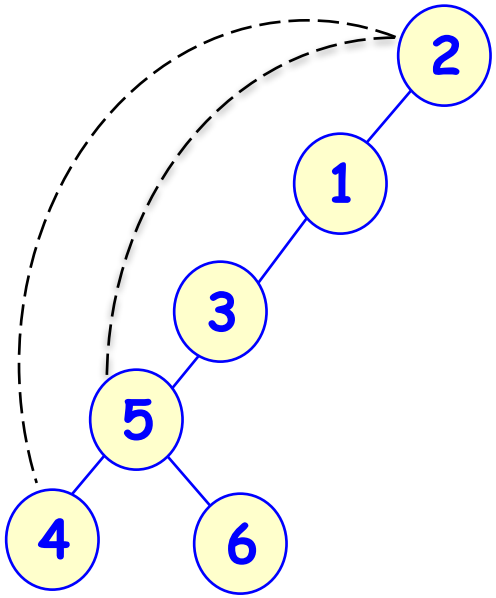
Grafos - Busca em Profundidade

Árvores (Floresta) de Profundidade: são as árvores da recursão da busca.



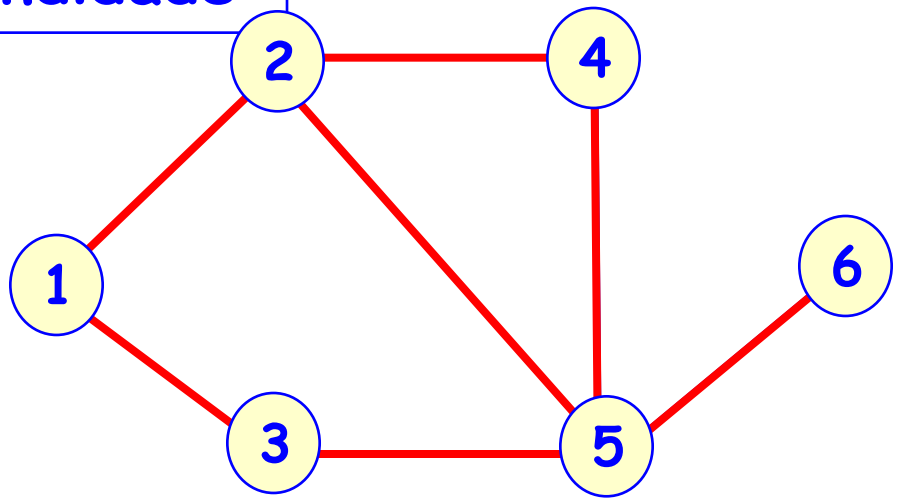
Após BP(5,6):

Árvores de Profundidade:

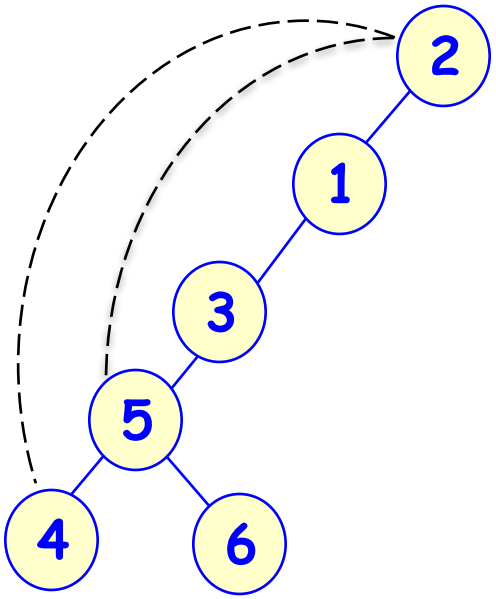


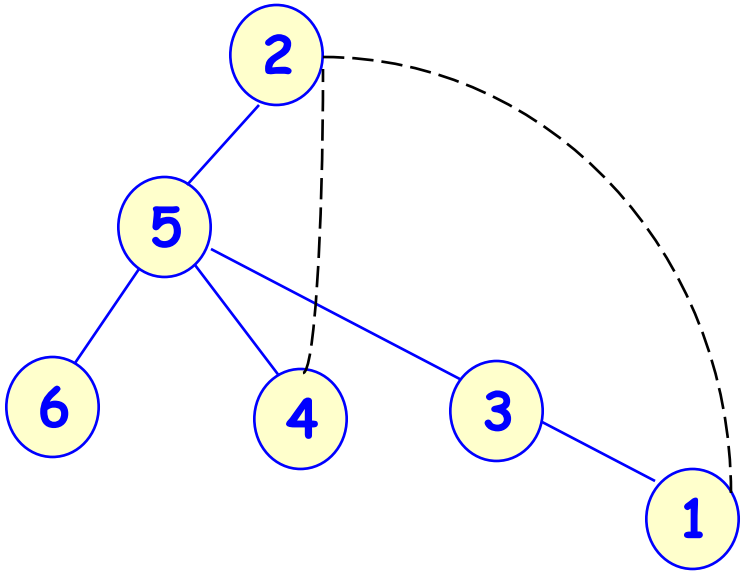
Grafos - Busca em Profundidade

Árvores (Floresta) de Profundidade: são as árvores da recursão da busca.



Árvores de Profundidade:





Grafos - Busca em Profundidade (DFS) - MA

```
BP(p,v):
    cpre ← cpre+1
    pre[v] ← cpre
    para w ← 1..n incl.:
        se  E[v,w] = 1   e   pre[w] = 0:
                BP(v,w)
```

```
para i ← 1..n incl.:
    pre[i] ← 0
cpre ← 0
para  i  ← 1..n incl:
    se  pre[i] = 0:
        BP(i,i)
```

Complexidade: $O(n^2)$

A marcação feita é guardando a ordem de visita no vetor pre.

Grafos - Busca em Profundidade (DFS) - LA

```
BP(p,v):
    cpre ← cpre+1
    pre[v] ← cpre
    w ←A[v]
    enquanto  w ≠ nulo:
        se   pre[w.v] = 0:
            BP(v, w.v)
        w ← w.prox

para i ← 1..n incl.:
    pre[i] ← 0
cpre ← 0
para  i  ← 1..n incl.:
    se  pre[i] = 0:
        BP(i,i)
```

Complexidade: O(n+m)

A marcação feita é guardando a ordem de visita no vetor pre.

Grafos - Busca em Profundidade (DFS) - LA
Usando vector C++

```
BP(p,v);
    cpre ← cpre+1
    pre[v] ← cpre
    para j ← 0..A[v].size():
        se   pre[A[v][j]] = 0:
            BP(v,A[v][j])


para i ← 1..n incl.:
    pre[i] ← 0
cpre ← 0
para i ← 1..n incl.:
    se  pre[i] = 0:
        BP(i,i)
```

Delcaração do grafo em C++

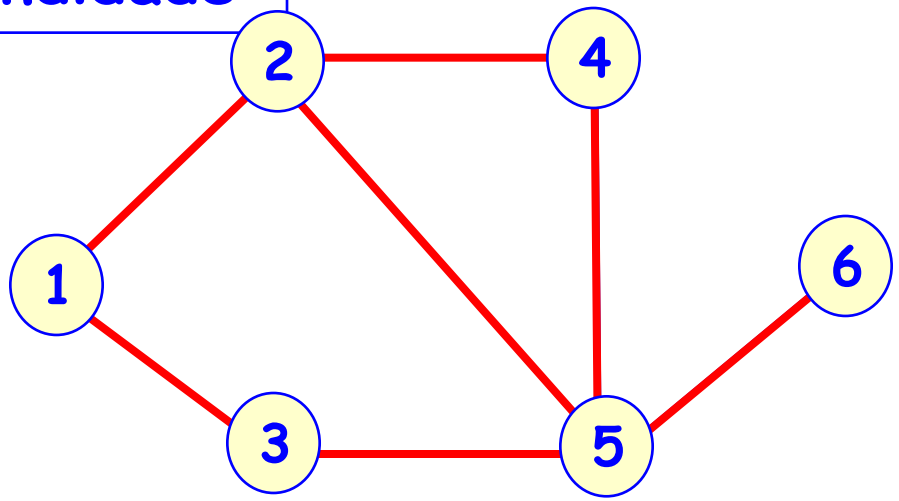
```
    vector<int> A[NMAX];
```

Complexidade: O(n+m)

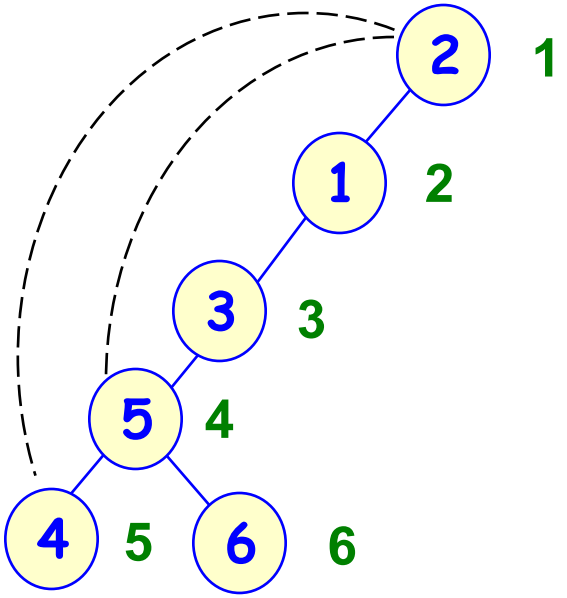
A marcação feita é guardando a ordem de visita no vetor pre.

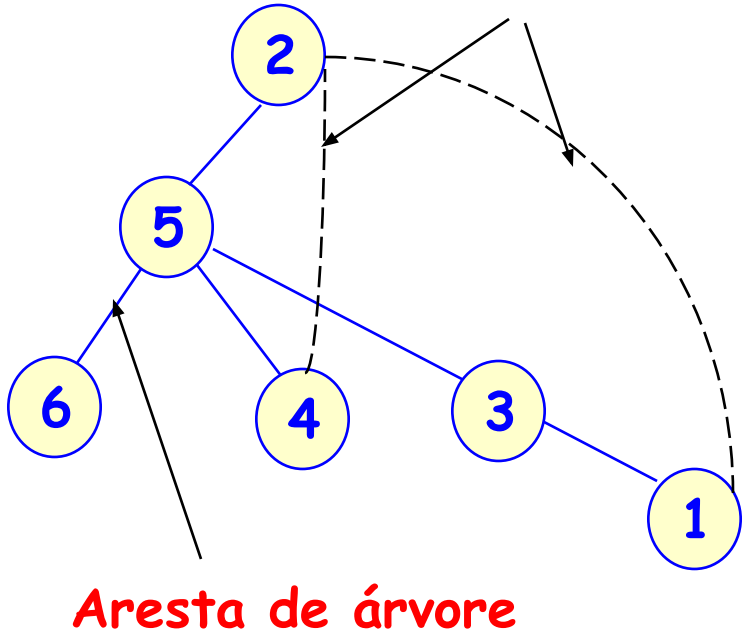
Grafos - Busca em Profundidade

Árvores (Floresta) de Profundidade:
Mostram a sequência das visitas. Pode haver mais de uma árvore de profundidade.



Árvores de Profundidade:





Grafos - Busca em Profundidade (DFS)

Propriedade I da BP:
Todos os vértices e todas as arestas são visitados na busca.

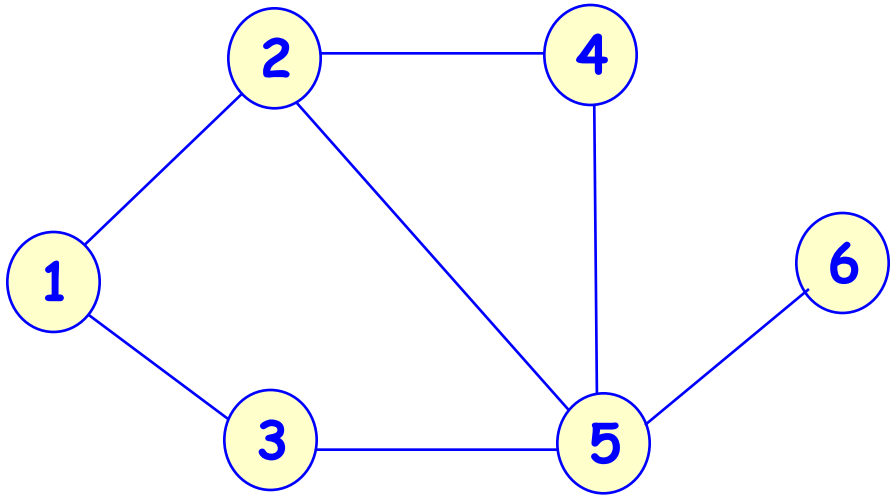
Propriedade II da BP:
A complexidade da busca é $O(n^2)$ se for usada a representação por matriz de adjacências.

Propriedade III da BP:
A complexidade da busca é $O(n+m)$ se for usada a representação por listas de adjacências.

Propriedade IV da BP:
Dado qualquer caminho C em G, existe um vértice v desse caminho tal que todos os demais vértices do caminho estão em nós descendentes de v, na árvore de profundidade de G.

Grafos - Busca em Profundidade

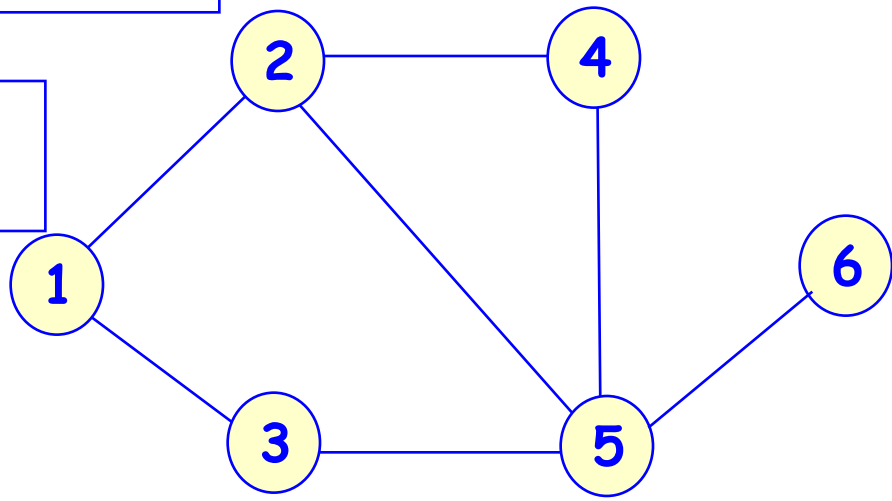
Possibilidades da DFS:
Pequenas modificações no algoritmo permitem descobrir propriedades do grafo



- Visita a todas as arestas
- Obtenção dos graus dos vértices
- Determinação se o grafo é conexo
- Determinação dos componentes conexos
- Determinação de existência de ciclos
- Descoberta de elementos estruturais (pontes, blocos etc)
- Descoberta se o grafo é bipartido, ciclo, árvore, completo, regular

Grafos - Busca em Profundidade

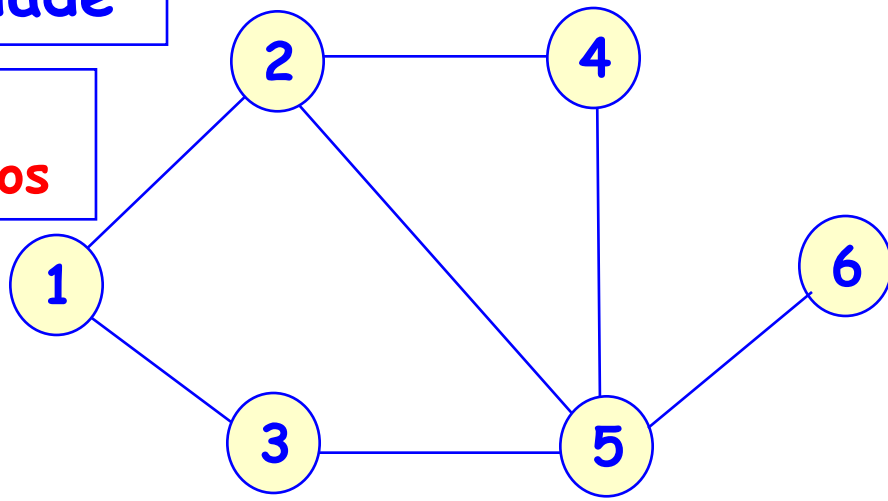
Possibilidades da DFS:
Determinar se o grafo é conexo




```
desmarcar vértices()
c ← 0
para  i  ← 1..n incl.:
    se  i não marcado:
        c ← c+1
        BP(i,i)
se c > 1:
    escrever ('Desconexo')
senão:
    escrever ('Conexo')
```

Grafos - Busca em Profundidade

Possibilidades da DFS:
Determinação dos componentes conexos

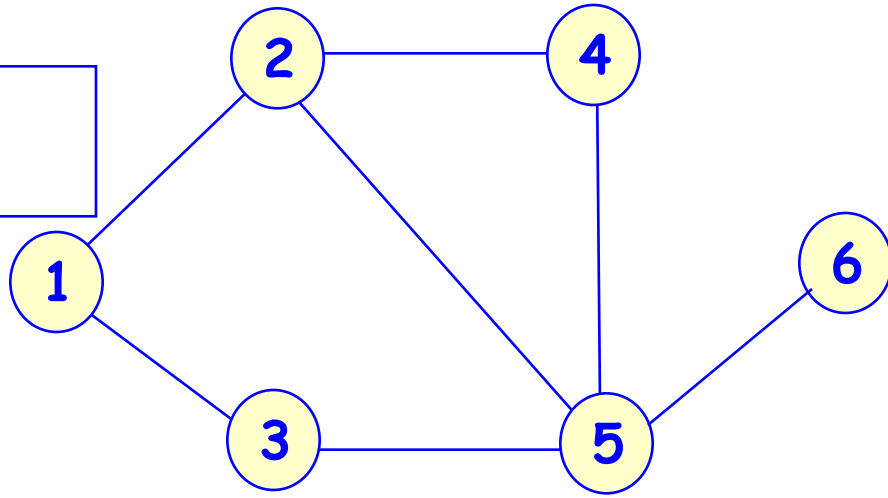



```
co[*] ← 0;    c ← 0
para  i  ← 1..n incl.:
    se co[i] = 0:
        c ← c+1
        BP(i,i)

para i ← 1..c incl.:
    escrever('Componente',i)
    para j ← 1..n incl.:
        se co[j] = i:
            escrever (j)
```

```
BP(p,v):
    co[v] ← c;
    para vizinhos w  de   v:
        se   co[w] = 0:
            BP(v,w)
```

Grafos - Busca em Profundidade

Possibilidades da DFS:
Determinação de existência de ciclos

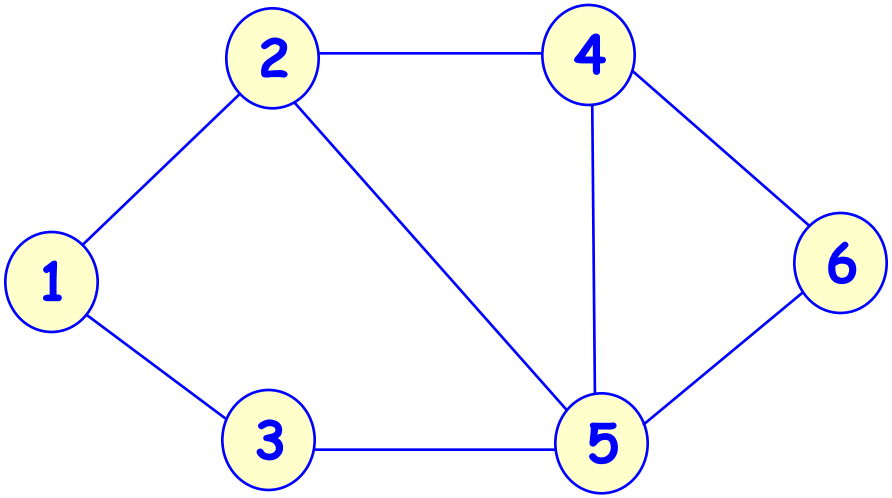



```
pre[*] ← 0
tc ← F
para  i  ← 1..n incl.:
    se pre[i] = 0:
        tc ← tc ou BP(i,i)

se tc:
    escrever ('Cíclico')
senão:
    escrever ('Acíclico')
```

```
lógico BP(p,v):
    pre[v] ← 1
    para vizinhos w  de   v:
        se   pre[w] = 0:
            se BP(v,w):
                retornar V
        senão se  w ≠ p:
            retornar V
    retornar F
```

Grafos - Busca em Profundidade

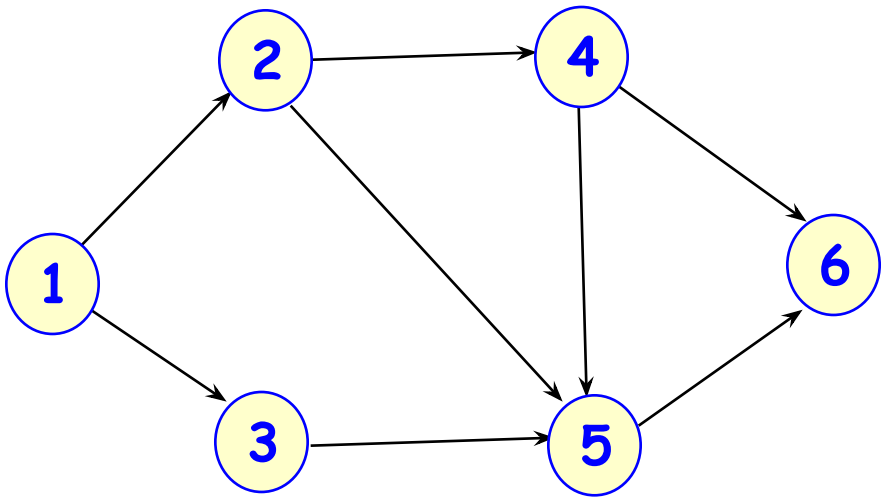
Problema ORIENTAÇÃO ACÍCLICA:
Dado um grafo simples, orientar cada aresta do grafo (transformá-lo em um digrafo), tal que não hajam ciclos no digrafo criado.



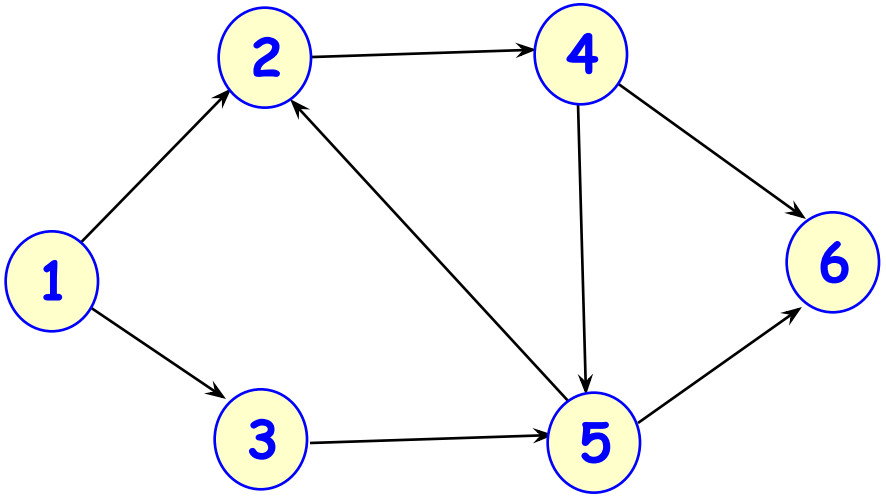
Grafos - Busca em Profundidade

Problema ORIENTAÇÃO PARA DAG:
Dado um grafo simples, orientar cada aresta do grafo (transformá-lo em um digrafo), tal que não hajam ciclos no digrafo criado.

OK



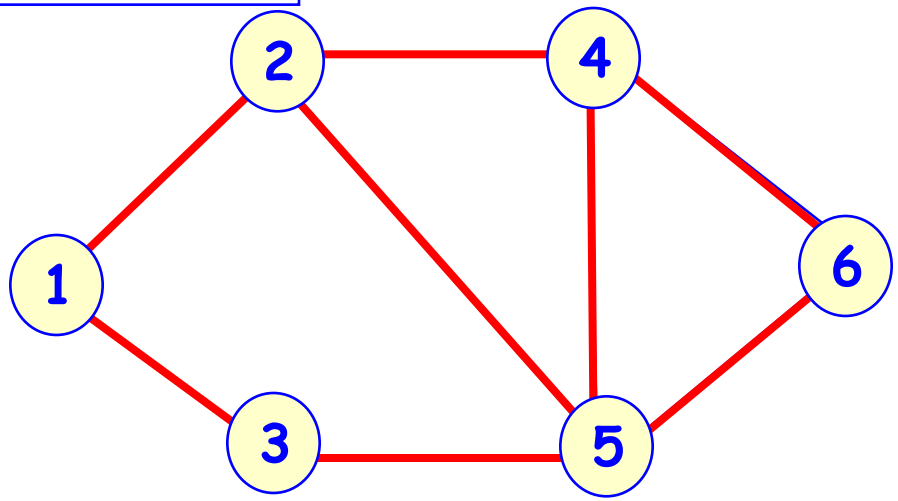
NOK

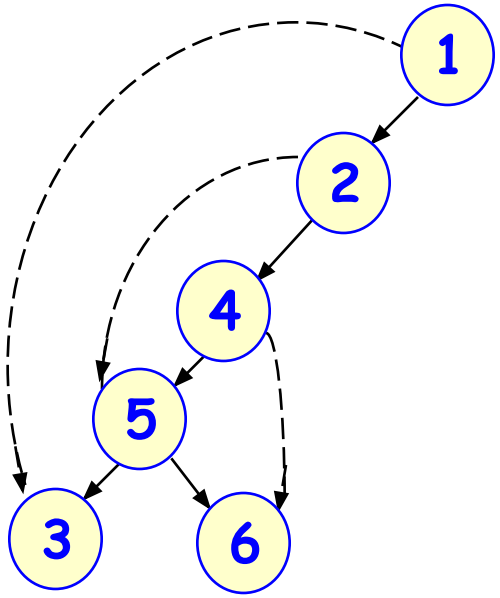


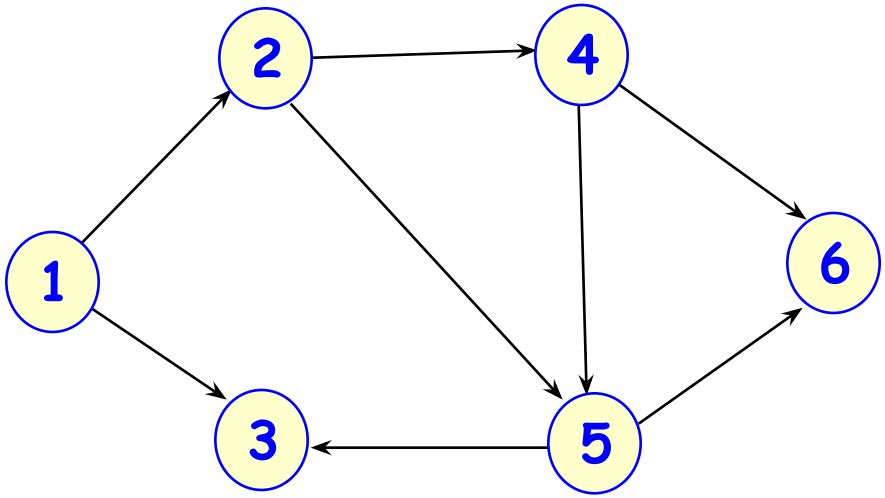
Grafos - Busca em Profundidade

Solução ORIENTAÇÃO PARA DAG

Criar a árvore de profundidade. Arestas de árvore tornam-se arestas diretas num digrafo. Arestas de retorno tornam-se arestas inversas.





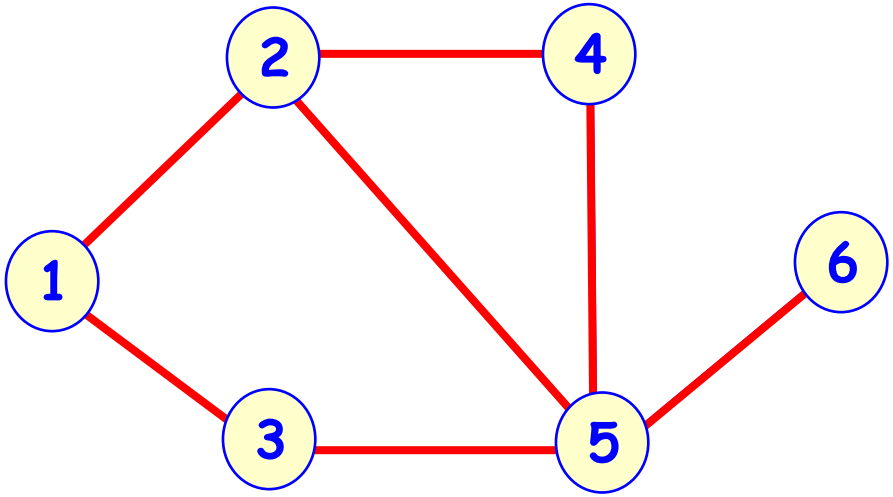


Grafos - Busca em Profundidade II (DFS) - MA

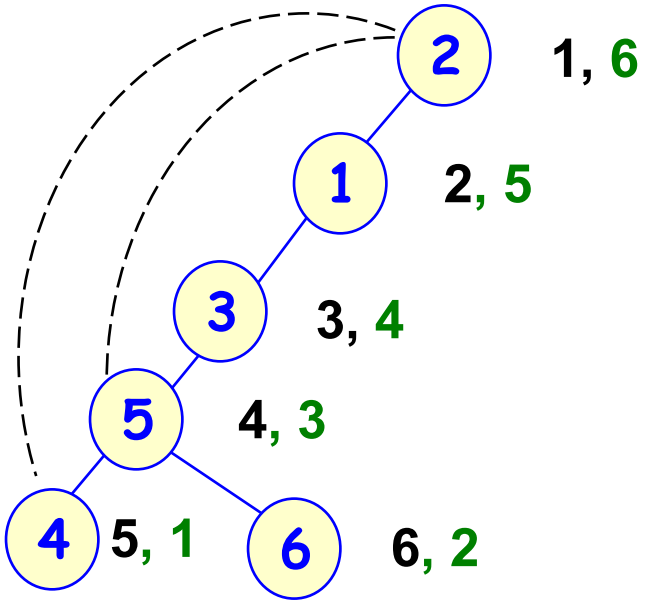
```
BP(u,v):
    cpre ← cpre+1
    pre[v] ← cpre
    para w ← 1 até n incl.:
        se   E[v,w] = 1  e  pre[w] = 0:
            BP(v,w)
    cpos ← cpos+1
    pos[v] ← cpos


para i ← 1 até n incl:
    pre[i], pos[i] ← 0, 0
cpre, cpos  ← 0, 0
para  i ← 1 até  n incl.:
    se  pre[i] = 0:
        BP(i,i)
```

Complexidade: $O(n^2)$

Grafos - Busca em Profundidade



Árvores de Profundidade:

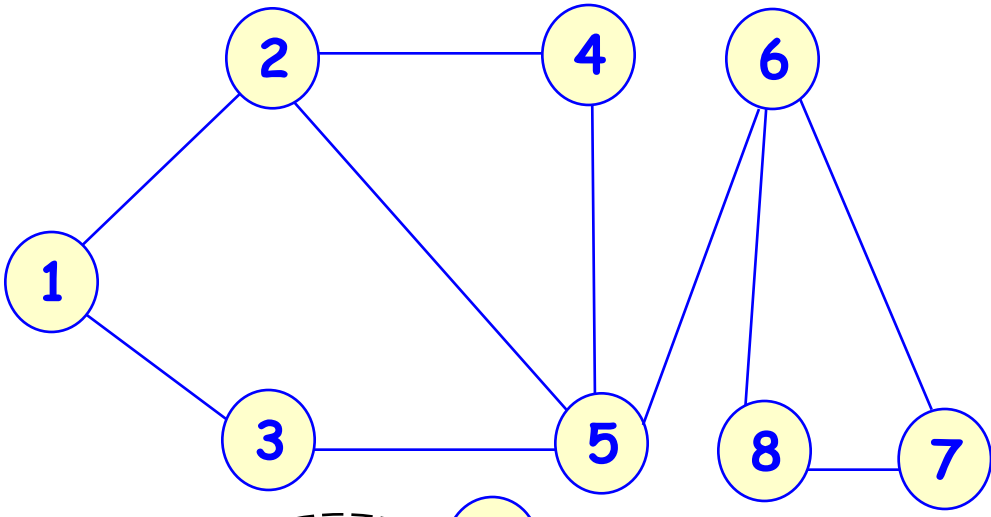


Em preto, a ordem de entrada na busca.

Em verde, a ordem de saída da busca.

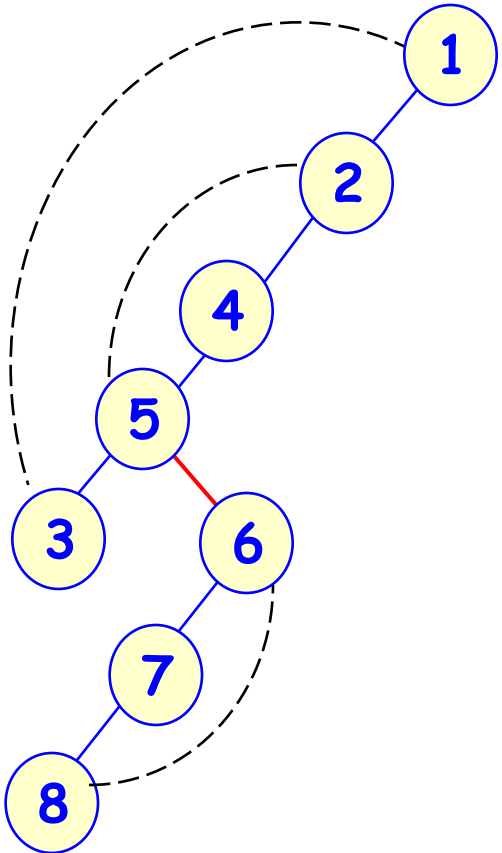
Grafos – Pontes

Ponte: Aresta que, se removida, desconecta o grafo: Ex (5,6)



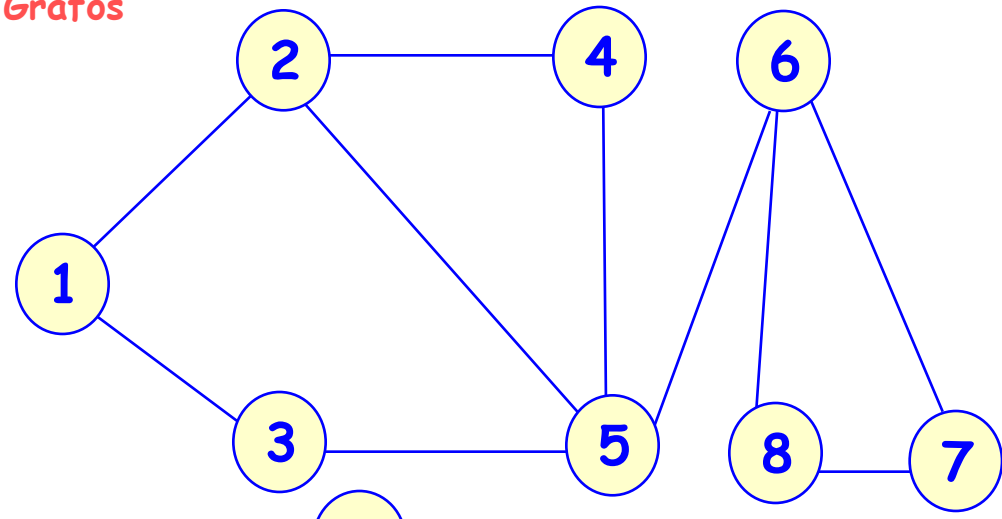
Idéia de um algoritmo:

Dada uma árvore de profundidade, a aresta (v,w) será ponte quando w é filho de v e não há descendentes de w (incluindo w) com arestas de retorno para ancestrais de w.



Grafos – Pontes

Ponte: Aresta que, se removida, desconecta o grafo: Ex (5,6)

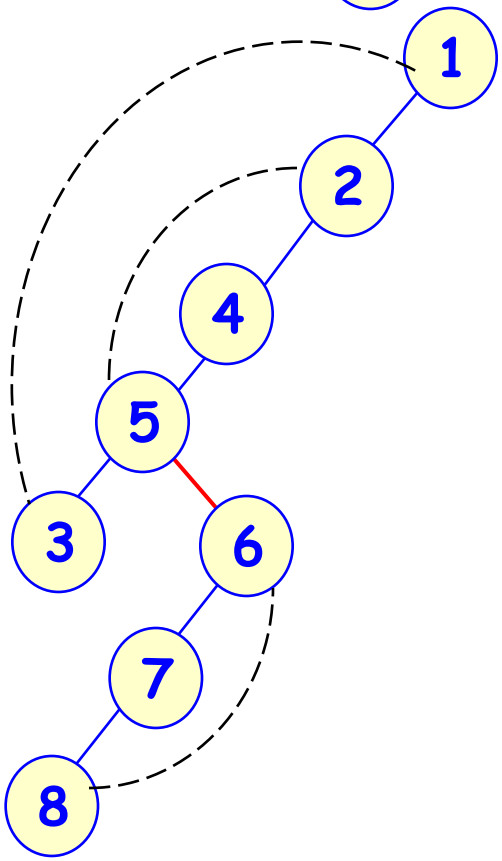


Idéia de um algoritmo:

Funções p/ os vértices:

pre(v) = ordem de entrada de v na BP.

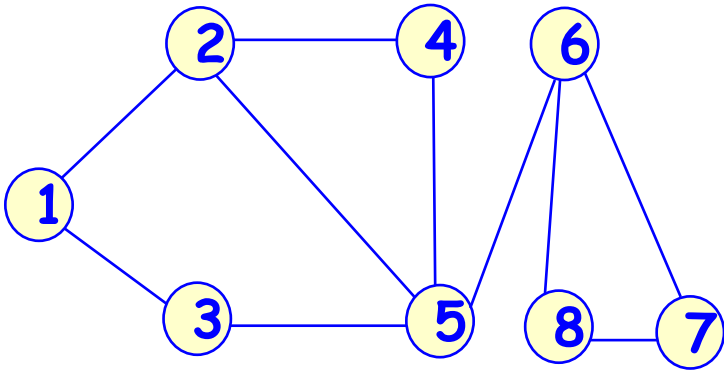
low(v) = menor ordem de entrada de um vértice alcançado diretamente por v ou por descendente de v na árvore de profund.



v	Pre	Low
1	1	1
2	2	1
4	3	1
5	4	1
3	5	1
6	6	6
7	7	6
8	8	6

Conclusão: Aresta (5,6) é ponte

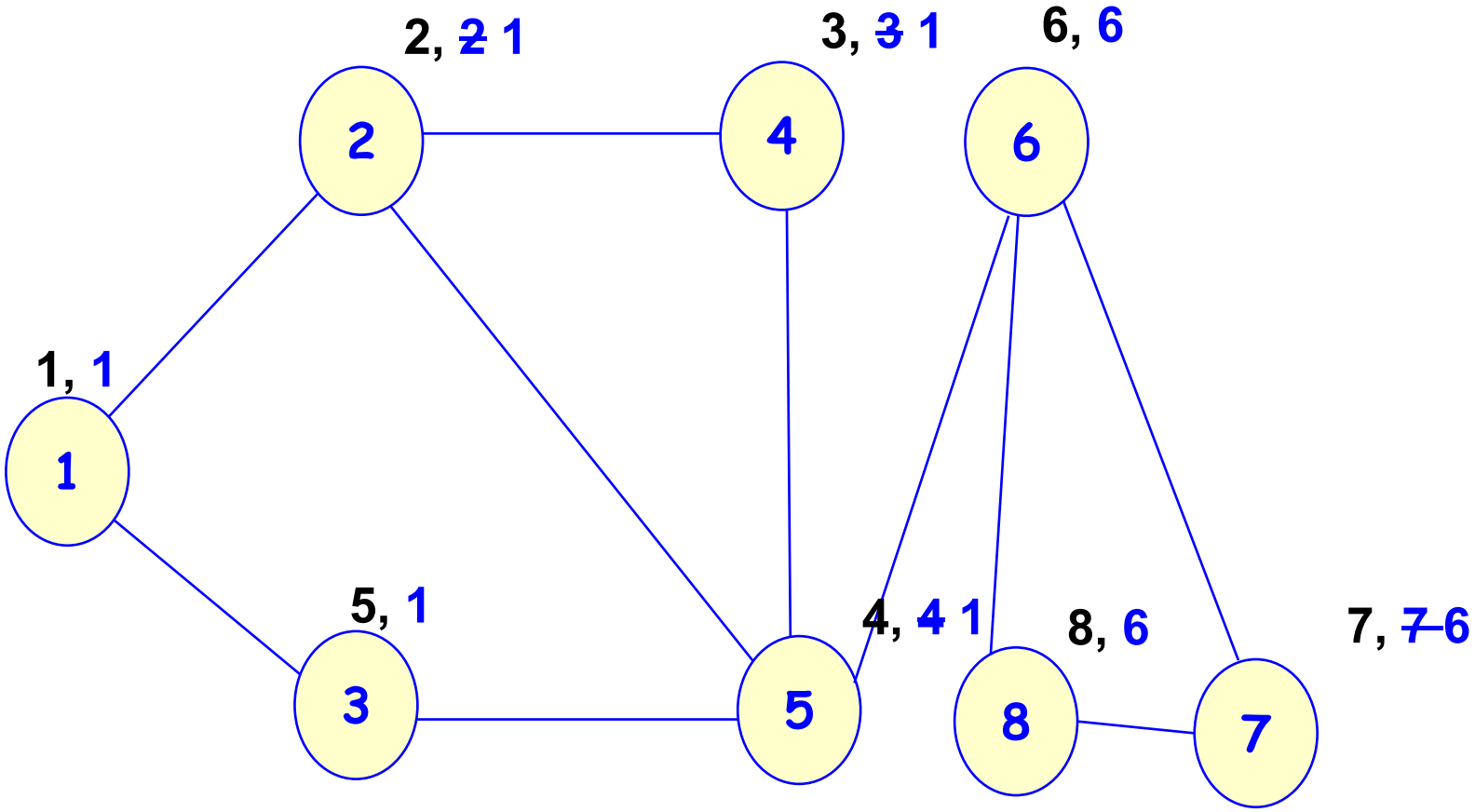
Grafos - Pontes




```
Pontes (p, v):
    cpre ← cpre+1
    pre[v], low[v] ← cpre, cpre
    para vizinhos w de v:
        se  pre[w] = 0:
            Pontes(v, w)
            se  low[w] = pre[w]:
                escrever( v , w, ' é Ponte')
            low[v] ← min (low[v], low[w])  #w é descendente de v
        senão se  w ≠ p:
            low[v] ← min (low[v], pre[w])  #v descende de w e w não é pai de v

pre[*] ← 0;
cpre ← 0;
para i ← 1..n:
    se pre[i]=0:
        Pontes(i,i)
```

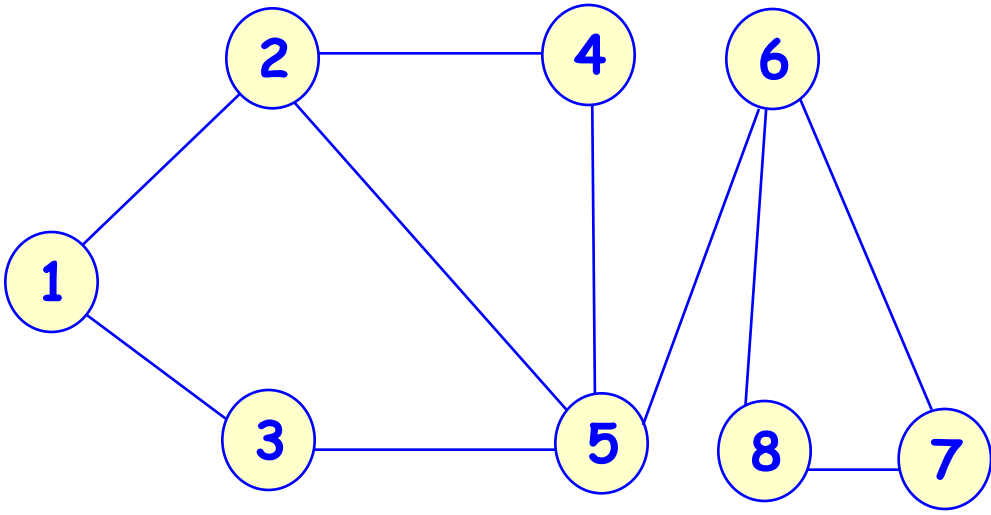
Grafos - Pontes

Exemplo com a Busca começando no vértice 1.



Grafos - Pontes

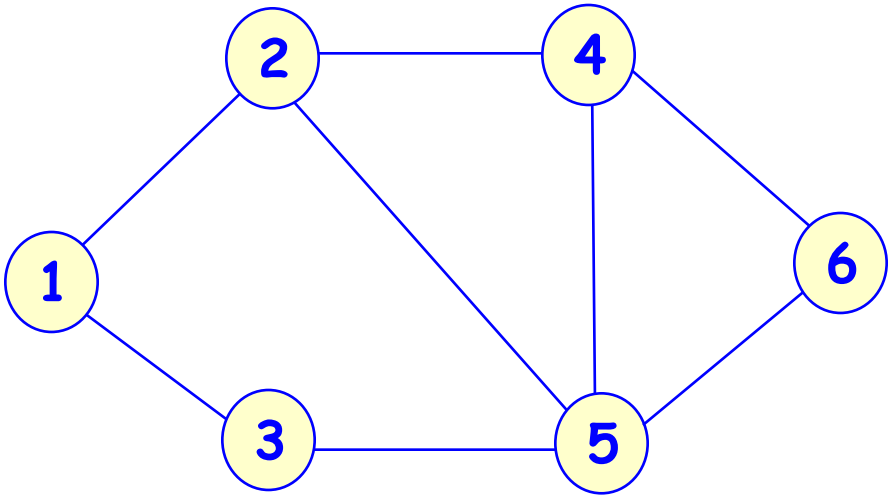
Exemplo com a Busca começando no vértice 1.



v	1		2		3		4		5		6		7		8	
	P	L	P	L	P	L	P	L	P	L	P	L	P	L	P	L
1	1	1														
2	1	1	2	2												
4	1	1	2	2			3	3								
5	1	1	2	2			3	3	4	4						
3	1	1	2	2	5	1	3	3	4	1						
6	1	1	2	2	5	1	3	3	4	1	6	6				
7	1	1	2	2	5	1	3	3	4	1	6	6	7	7		
8	1	1	2	1	5	1	3	1	4	1	6	6	7	6	8	6

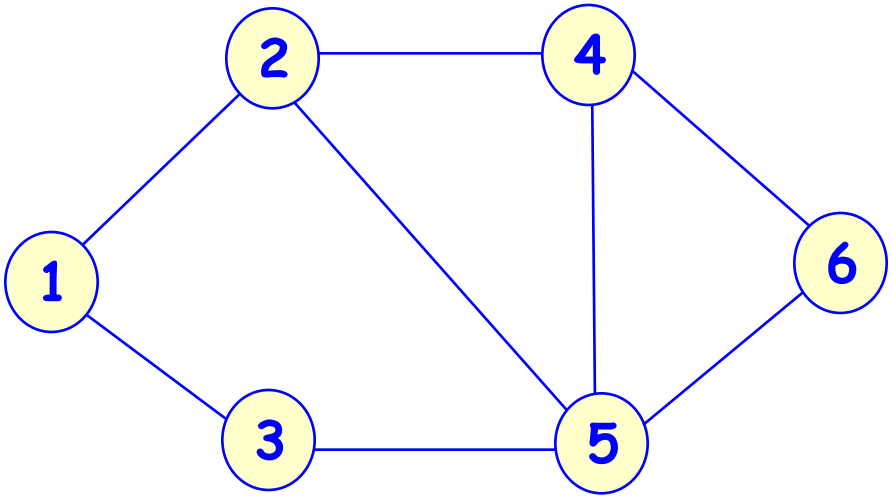
Grafos - Pontes

Problema MÃO ÚNICA (I):
Dado o mapa de uma cidade, representado como um grafo simples, onde os vértices são as esquinas e as arestas são os trechos de rua entre esquinas, dizer se é possível atribuir mão única a todos os trechos de ruas, tal que toda esquina seja atingível a partir de qualquer outra.



Grafos - Pontes

Problema MÃO ÚNICA (I):
Dado o mapa de uma cidade, representado como um grafo simples, onde os vértices são as esquinas e as arestas são os trechos de rua entre esquinas, dizer se é possível atribuir mão única a todos os trechos de ruas, tal que toda esquina seja atingível a partir de qualquer outra.

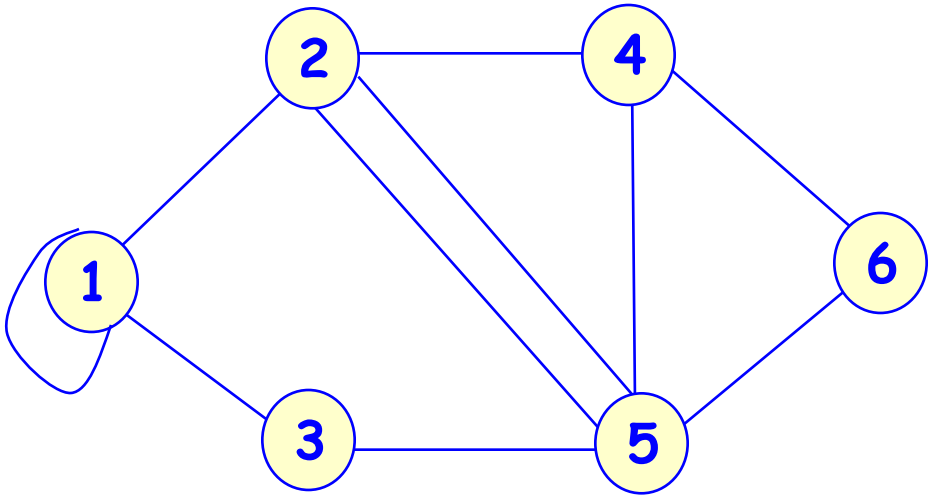


Solução MÃO ÚNICA (I):

Basta verificar se existe ou não ponte no grafo dado.

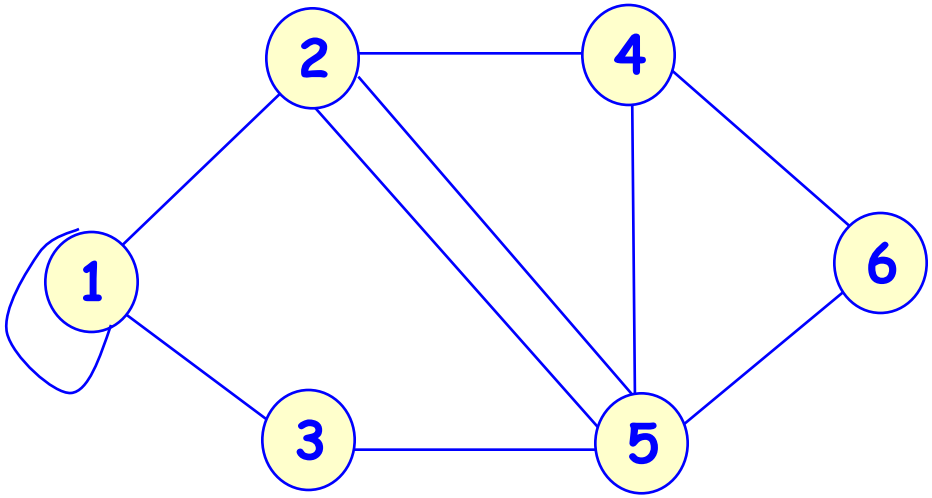
Grafos - Pontes

Problema MÃO ÚNICA (II):
Dado o mapa de uma cidade, representado como um hipergrafo (grafo com multi-arestas e loops), onde os vértices são as esquinas e as arestas são os trechos de rua entre esquinas, dizer se é possível atribuir mão única a todos os trechos de ruas, tal que toda esquina seja atingível a partir de qualquer outra.



Grafos - Pontes

Problema MÃO ÚNICA (II):
Dado o mapa de uma cidade, representado como um hipergrafo (grafo com multi-arestas e loops), onde os vértices são as esquinas e as arestas são os trechos de rua entre esquinas, dizer se é possível atribuir mão única a todos os trechos de ruas, tal que toda esquina seja atingível a partir de qualquer outra.

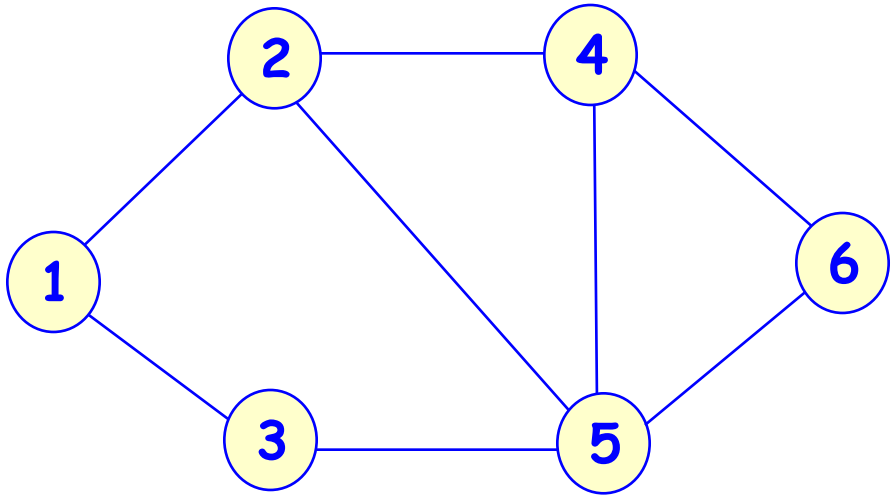


Solução MÃO ÚNICA (II):

Basta transformar o hipergrafo em grafo simples e verificar se existe ou não ponte no grafo transformado e se essa ponte é a única aresta entre os dois vértices, no hipergrafo.

Grafos - Pontes

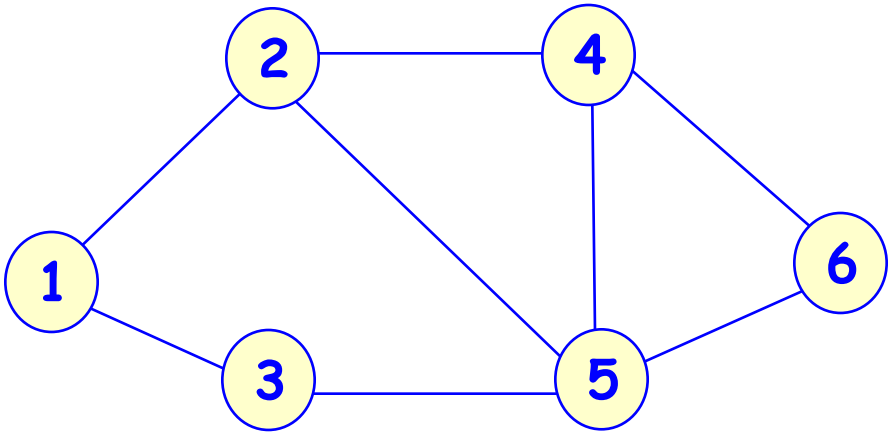
Problema MÃO ÚNICA (III):
Dado o mapa de uma cidade, representado como um grafo simples, onde os vértices são as esquinas e as arestas são os trechos de rua entre esquinas, dizer se é possível atribuir mão única a todos os trechos de ruas, tal que toda esquina seja atingível a partir de qualquer outra. Quando for possível, apresentar uma atribuição válida.

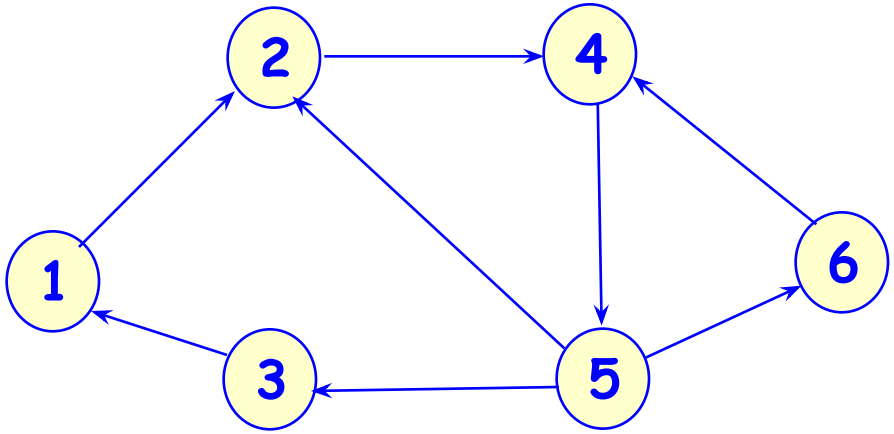


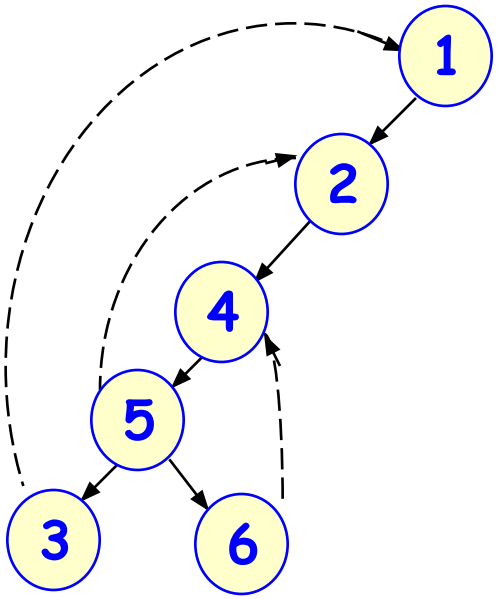
Grafos - Pontes

Solução MÃO ÚNICA (III):

Quando não existir ponte, usar a árvore de profundidade, atribuindo o sentido de cima para baixo para arestas da árvore e o sentido de baixo para cima para arestas de retorno.

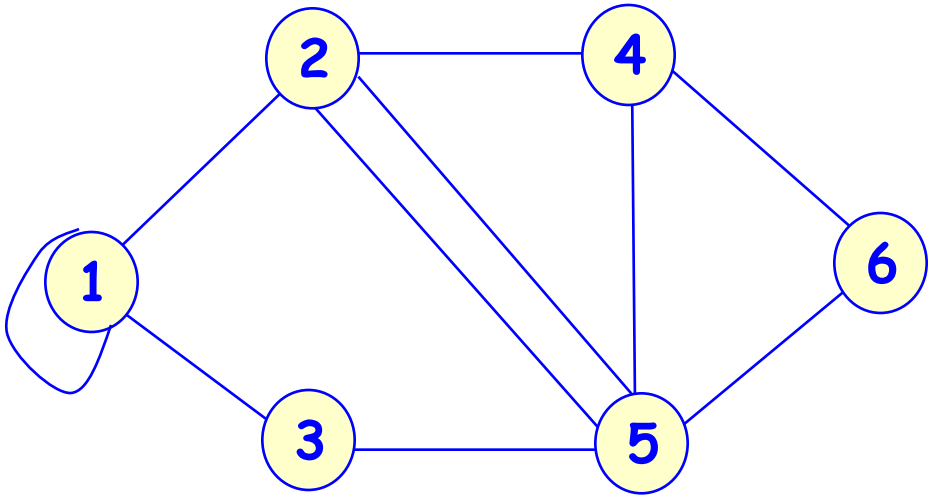






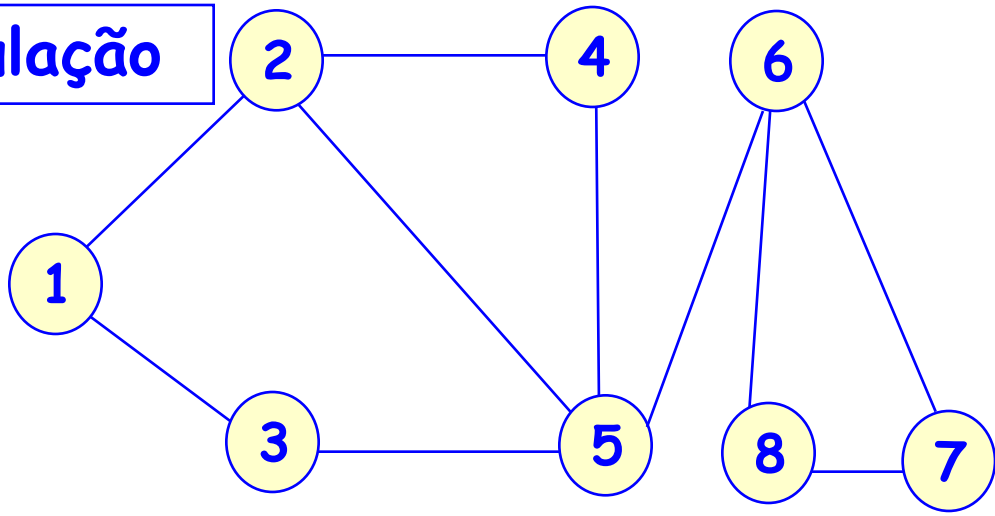
Grafos - Pontes

Problema MÃO ÚNICA (IV):
Dado o mapa de uma cidade, representado como um hiper-grafo, onde os vértices são as esquinas e as arestas são os trechos de rua entre esquinas, dizer se é possível atribuir mão única a todos os trechos de ruas, tal que toda esquina seja atingível a partir de qualquer outra. Quando for possível, apresentar uma atribuição válida.



Grafos – Pontos de Articulação

Ponto de Articulação:
Vértice que, se removido, desconecta o grafo: Ex 5



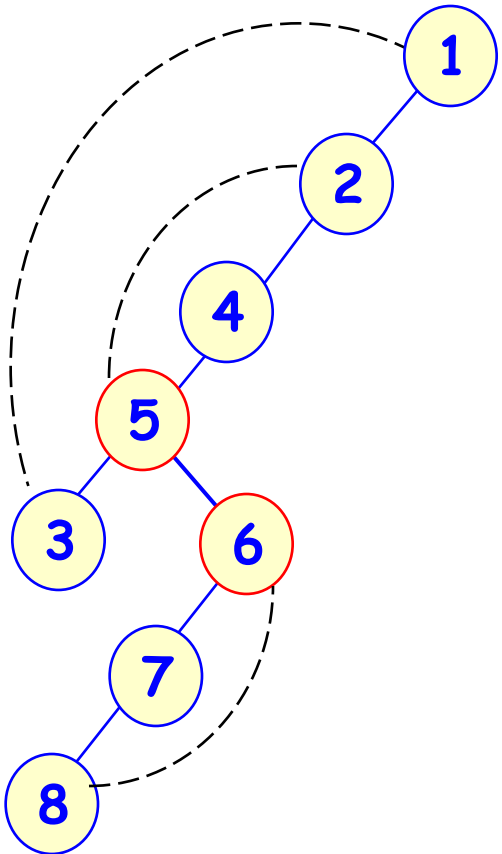
Idéia de um algoritmo:

Dada uma árvore de profund., o vértice v é ponto de articulação, se:

a) v é raiz da árv.prof. e tem mais de um filho.

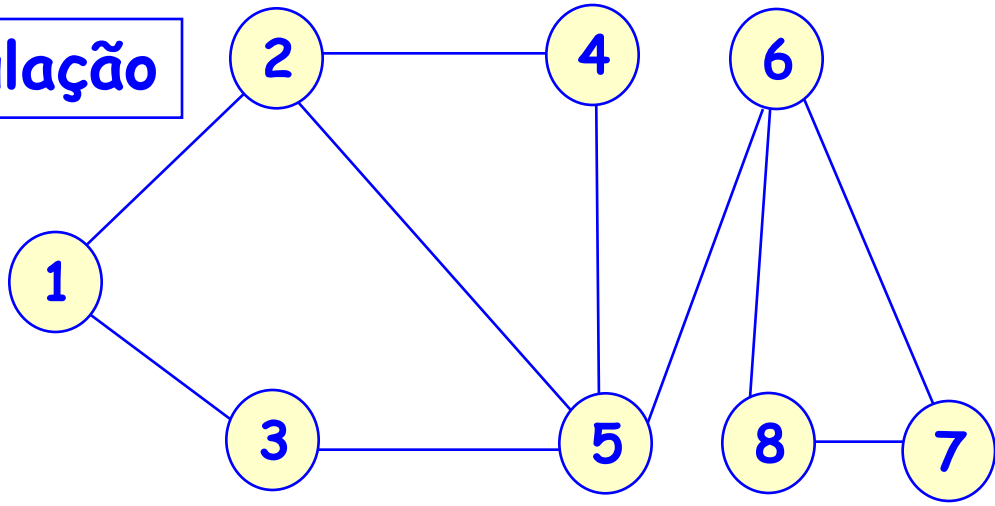
ou

b) v não é raiz, tem um filho w e não há descendentes de w (incluindo o próprio) com arestas de retorno para ancestrais de v.



Grafos – Pontos de Articulação

Ponto de articulação:
Vértice que, se removido, desconecta o grafo: Ex 5

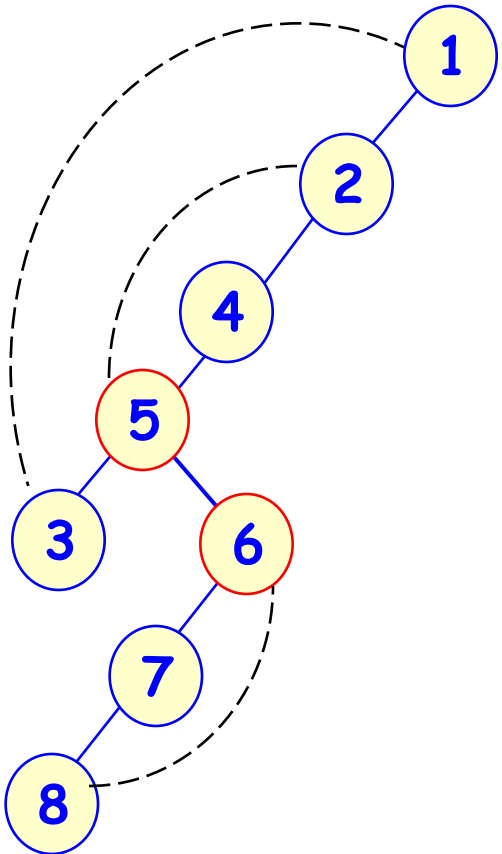


Idéia de um algoritmo:

Funções p/ vértices:

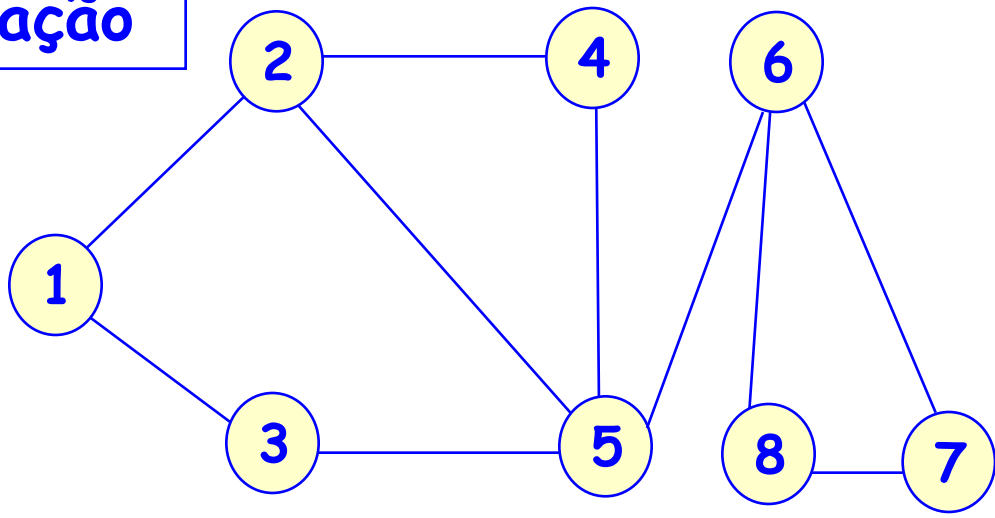
pre(v) e low(v), como para pontes.

pa(v) = número de filhos de v que não alcançam ancestrais de v na árv. prof.



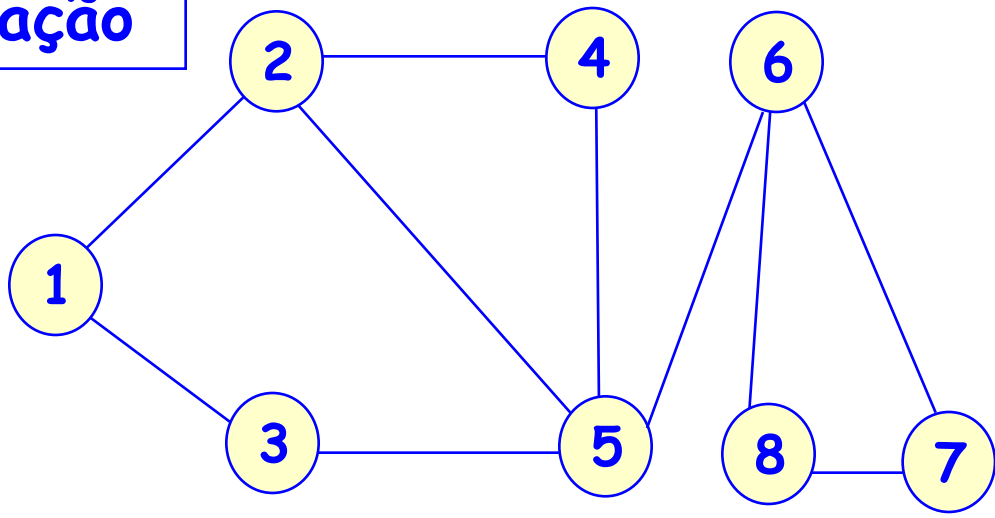
v	Pre	Low	Pa
1	1	1	1
2	2	1	0
4	3	1	0
5	4	1	1
3	5	1	0
6	6	6	1
7	7	6	0
8	8	6	0

Grafos – Pontos de Articulação




```
Pontos (p, v):
    cpre ← cpre+1;      pre[v] ← cpre;   low[v] ← cpre;   pa[v] ← 0;
    para vizinhos w de v:
        se  pre[w] = 0:
            Pontos(v, w)
            se  pre(v) ≤ low[w]:
                pa[v] ← pa[v]+1
            low[v] ← min (low[v], low[w])
        senão se  w ≠ p:
            low[v] ← min (low[v], pre[w])
```

Grafos – Pontos de Articulação

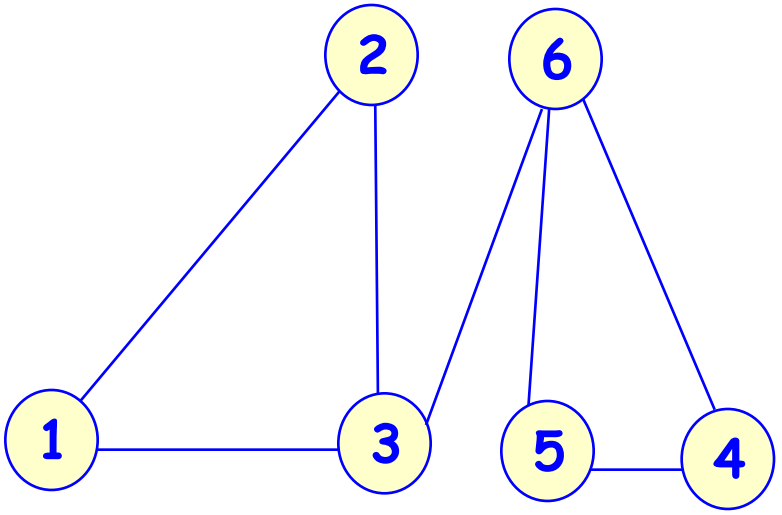


Externamente:

```
pre[*] ← 0;    cpre ← 0;
low[*] ← 0;    pa[*] ← 0;
para i ← 1 até n incl.:
    se  pre[i]=0:
        Pontos (i, i)
para i ← 1 até n incl.:
    se   i=1  e   pa[1] > 1:
        escrever ('vertice 1 é ponto  de articulação')
    senão se   i ≠ 1  e  pa[i] > 0:
        escrever ('vertice'+ i + ' é ponto  de articulação')
```

Grafos – Pontos de Articulação

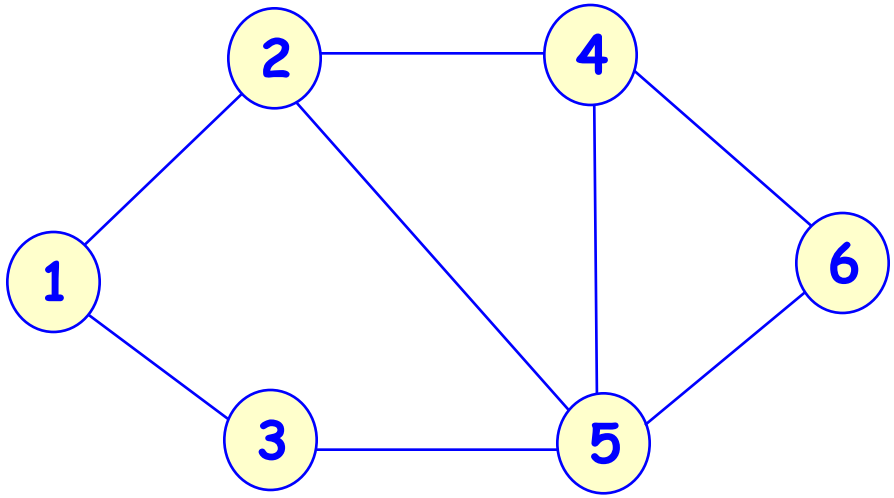
Exemplo com a Busca começando no vértice 1.



	P	L	Pa	P	L	Pa	P	L	Pa	P	L	Pa	P	L	Pa	P	L	Pa
v	1			2			3			4			5			6		
1	1	1	0															
2	1	1	0	2	2	0												
3	1	1	0	2	2	0	3	1	0									
6	1	1	0	2	2	0	3	1	0							4	4	0
4	1	1	0	2	2	0	3	1	0	5	5	0				4	4	0
5	1	1	1	2	1	0	3	1	1	5	4	0	6	4	0	4	4	1

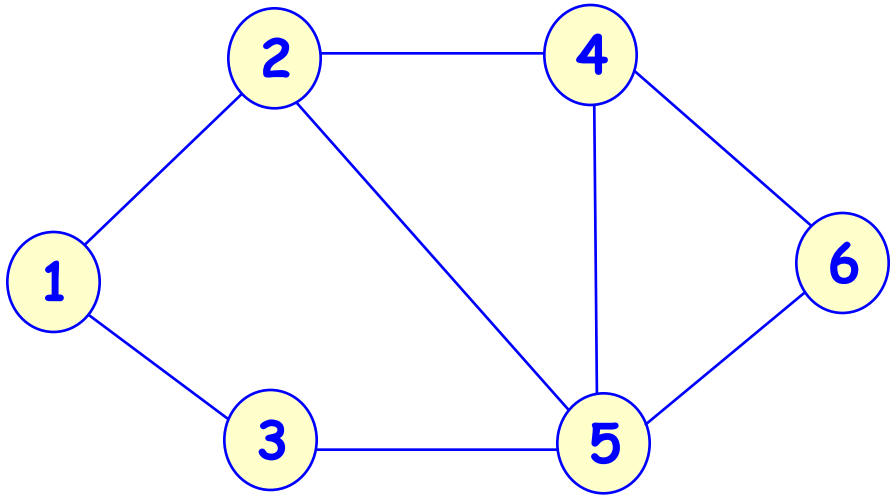
Grafos – Pontos de Articulação

Problema MÃO ÚNICA (I):
Dado o mapa de uma cidade, representado como um grafo simples, onde os vértices são as esquinas e as arestas são os trechos de rua entre esquinas, dizer se é possível atribuir mão única a todos os trechos de ruas, tal que toda esquina seja atingível a partir de qualquer outra.



Grafos – Pontos de Articulação

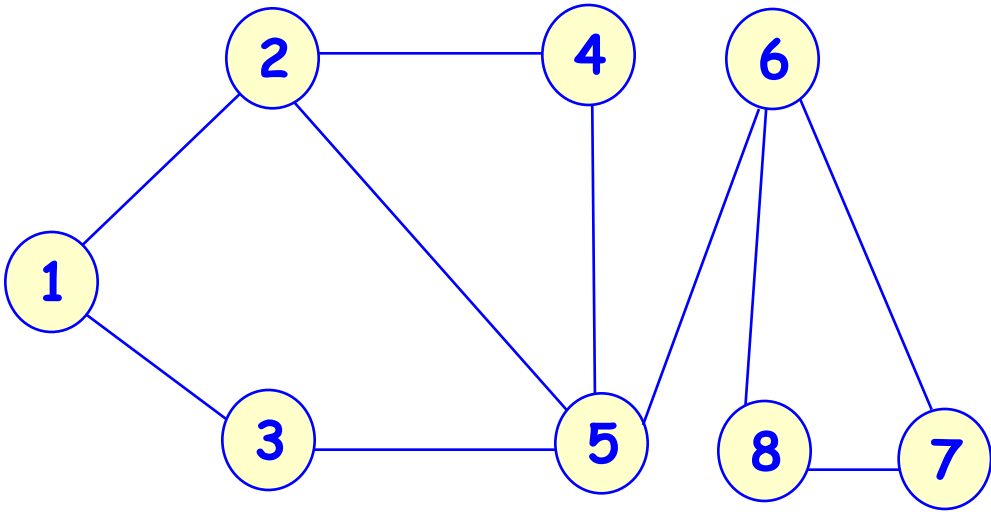
Problema MÃO ÚNICA (I):
Dado o mapa de uma cidade, representado como um grafo simples, onde os vértices são as esquinas e as arestas são os trechos de rua entre esquinas, dizer se é possível atribuir mão única a todos os trechos de ruas, tal que toda esquina seja atingível a partir de qualquer outra.



Solução MÃO ÚNICA (I):

Pontos de articulação interferem nas soluções???.

Grafos – Componentes biconexas



Componente biconexa:

subgrafo maximal tal que ou seja uma aresta ou haja, pelo menos, dois caminhos distintos entre cada par de vértices do subgrafo.

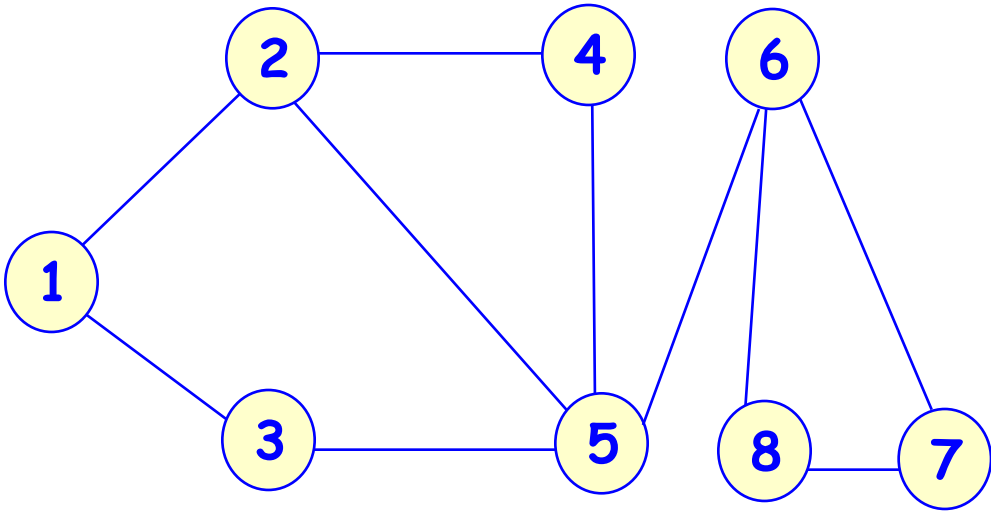
É equivalente a dizer que cada par de vértices da componente biconexa (quando há mais de 2 vértices na componente) faz parte de um ciclo.

Componentes biconexas do exemplo:

{1,2,3,4,5} {5,6} {6,7,8}

Grafos – Componentes biconexas

Marcador: Vértice w que, tem low(w) ≥ pre(pai(w))



Idéia de um algoritmo:

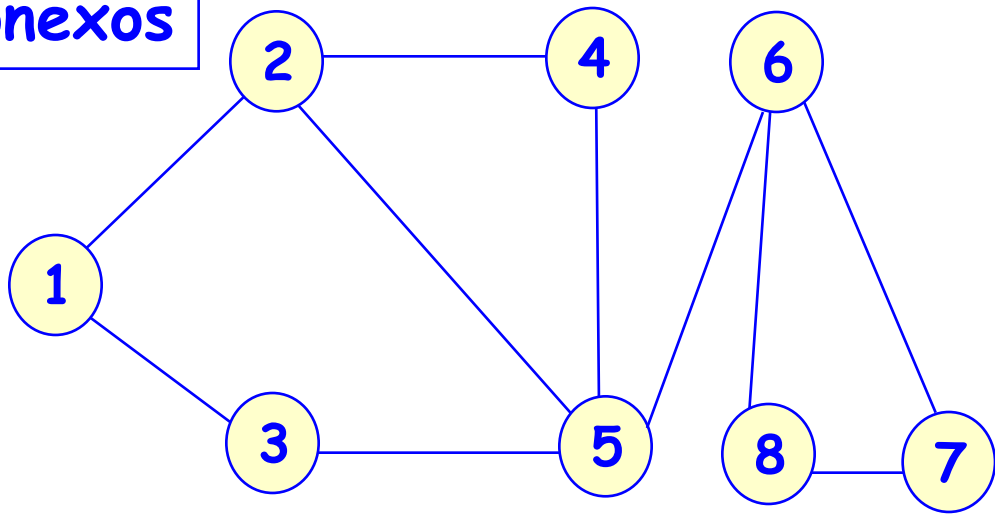
o marcador w mais "em baixo" da árvore de profundidade determina um componente = {elementos da subárvore com raiz em w} + pai(w). Removido esse componente, aplica-se recursão no grafo restante.

a) Fazer BP identificando marcadores e empilhando as arestas visitadas:

b) Quando um marcador w é encontrado, desempilham-se as arestas, até a aresta (w, pai(w)), inclusive.

Grafos – Componentes biconexos

Externamente:
Desmarcar vértices/arestas
Esvaziar pilha
cpre ← 0
Blocos (1, 1)

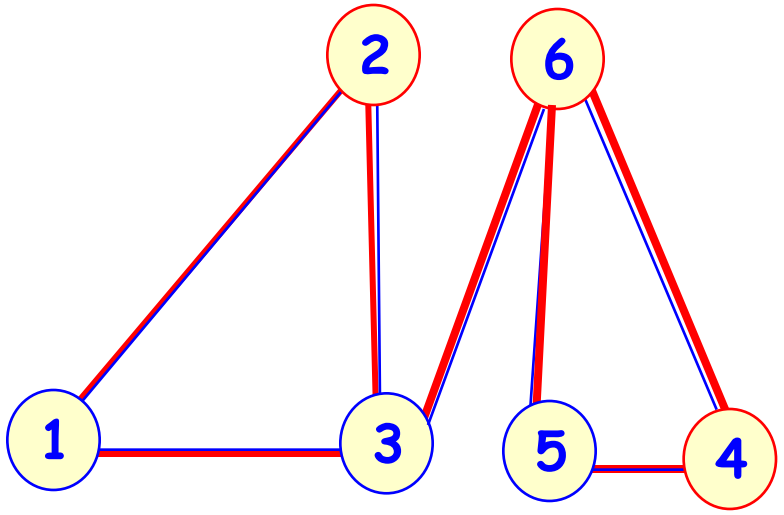


Blocos (p, v):

```
    cpre ← cpre+1;   pre[v] ← cpre;    low[v] ← cpre;
    para vizinhos w de v:
        se  aresta (w, v) não marcada:
            Empilhar(w, v)
            Marcar (w, v)
        se  pre[w] = 0:
            Blocos(v, w)
            se  pre[v] ≤ low[w]:
                Desempilhar até (w, v)
            low[v] ← min (low[v], low[w])
        senão se  w ≠ p:
            low[v] ← min (low[v], pre[w])
```

Grafos – Determinação de Blocos

Exemplo com a Busca começando no vértice 1.

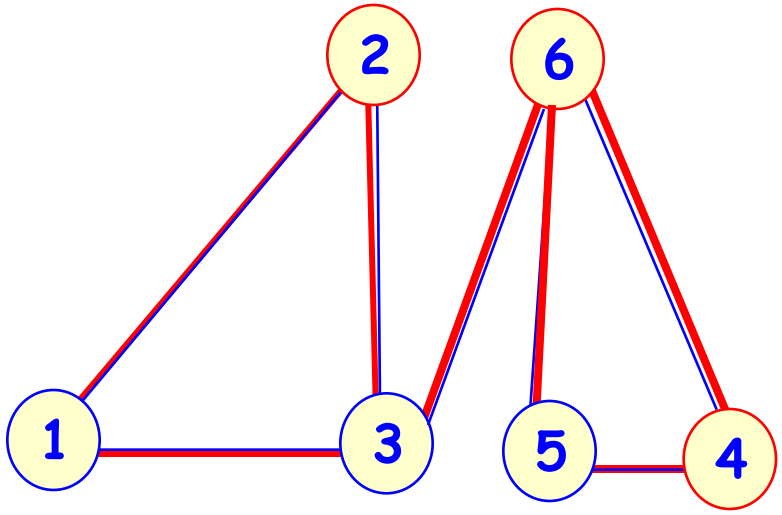


	P	L	P	L	P	L	P	L	P	L	P	L
v	1		2		3		4		5		6	
1	1	1										
2	1	1	2	2								
3	1	1	2	2	3	1						
6	1	1	2	2	3	1					4	4
4	1	1	2	2	3	1	5	5			4	4
5	1	1	2	1	3	1	5	4	6	4	4	4

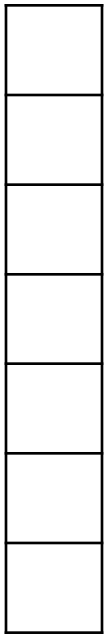
5/6
4/5
4/6
3/6
1/3
2/3
1/2

Grafos – Determinação de Blocos

Exemplo com a Busca começando no vértice 1.



	P	L	P	L	P	L	P	L	P	L	P	L
v	1		2		3		4		5		6	
1	1	1										
2	1	1	2	2								
3	1	1	2	2	3	1						
6	1	1	2	2	3	1					4	4
4	1	1	2	2	3	1	5	5			4	4
5	1	1	2	1	3	1	5	4	6	4	4	4



6: 5/6 4/5 4/6

3: 3/6

1: 1/3 2/3 1/2

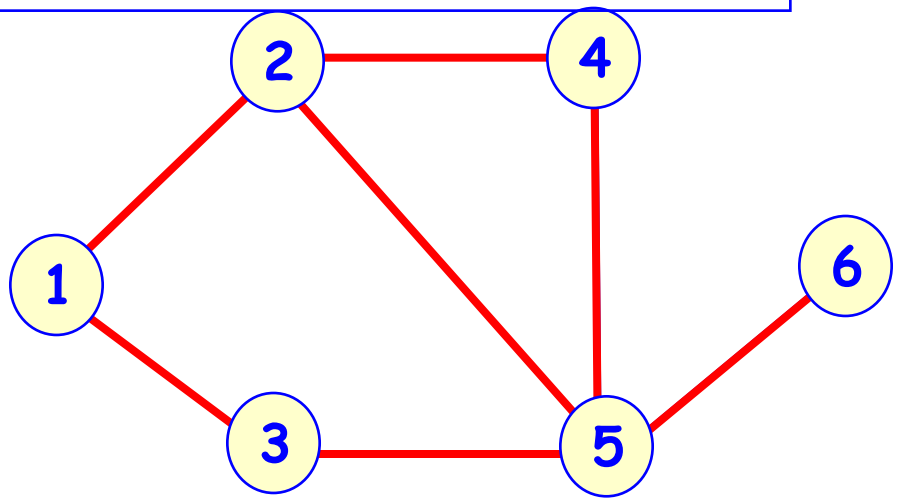
Grafos - Busca em Profundidade II (DFS) - MA

```
BP(u,v);
    cpre ← cpre+1;  pre[v] ← cpre;
    para w ← 1 até n incl.:
        se   E[v,w] = 1  e  pre[w] = 0:
            BP(v,w)
    cpos ← cpos+1;   pos[v] ← cpos;


para i ← 1 até n incl:
    pre[i] ← 0;    pos[i] ← 0;
cpre ← 0;          cpos ← 0;
para  i ← 1 até  n incl.:
    se  pre[i] = 0:
        BP(i,i)
```

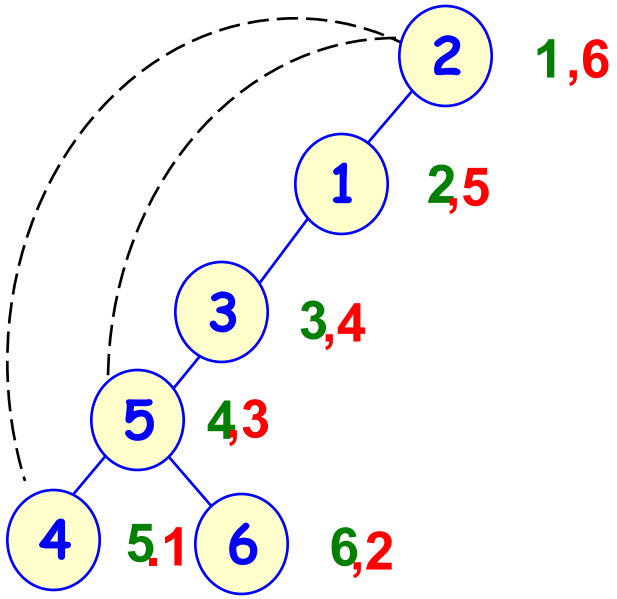
Complexidade: $O(n^2)$

Grafos - Busca em Profundidade II (DFS) - MA

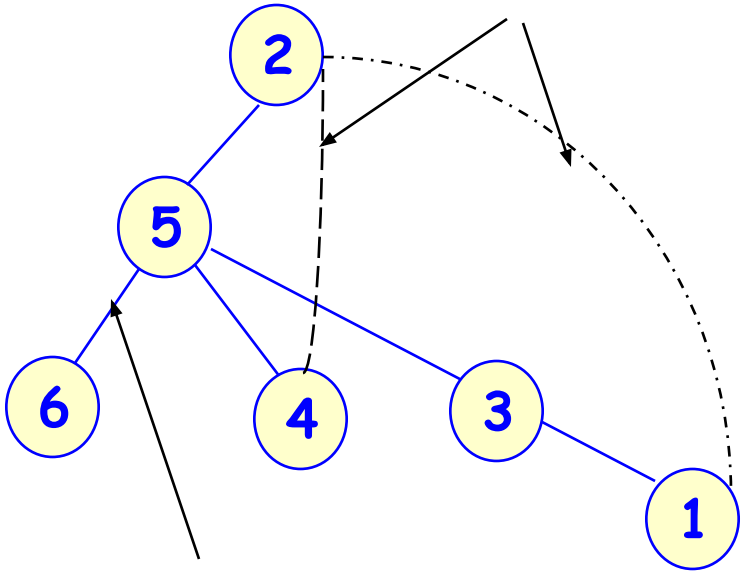
Árvore (Floresta) de Profundidade:



Árvore de Profundidade:



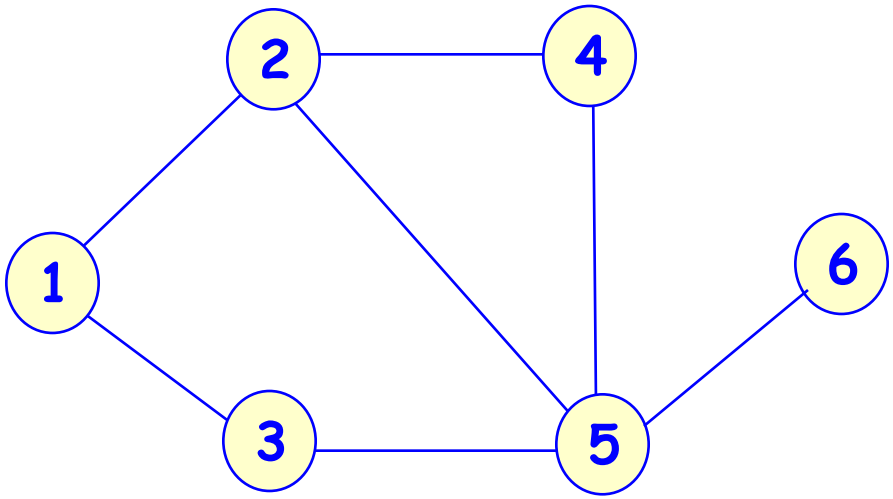
Arestas de retorno



Aresta de árvore

Grafos - Busca em Largura (BFS)

Idéia: Visitar, a cada passo, algum vizinho não visitado, do vértice mais antigamente visitado



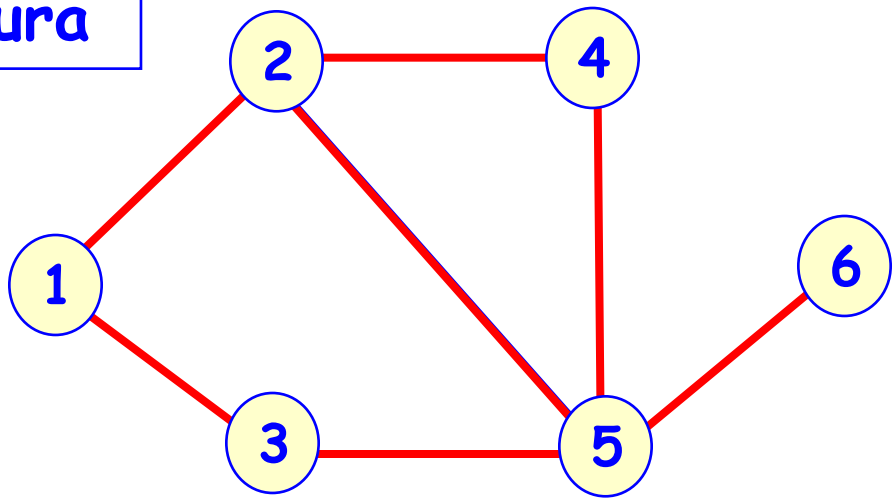
Grafos - Busca em Largura (BFS)

Idéia: Visitar, a cada passo, algum vizinho não visitado, do vértice mais antigamente visitado

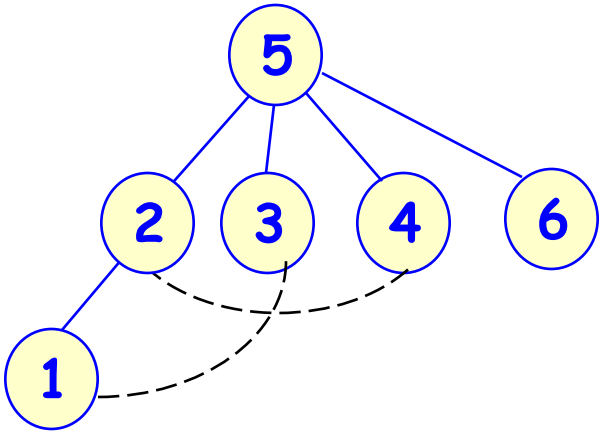
```
BL(p,v):
    Esvaziar fila Q;  Enfilar (p,v);  Marcar v;
    enquanto Q não vazia:
        (p,t) ← elemento inicial de Q
        para vizinhos w de t:
            se w não marcado:
                Enfilar (t,w)
                Marcar w
        Desenfilar (p,t)


Desmarcar vértices
para i ← 1 até n incl.:
    se i não marcado:
        BL(i,i)
```

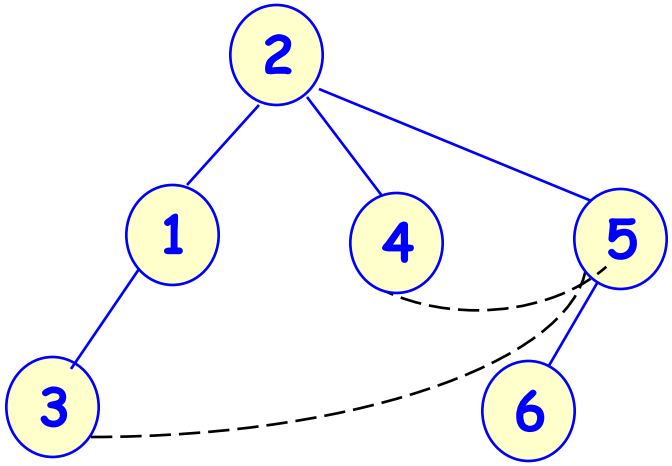
Grafos - Busca em Largura

Árvores de Largura:
Mostram a sequência das visitas. Pode haver mais de uma árvore de largura.



Árvores de Largura:





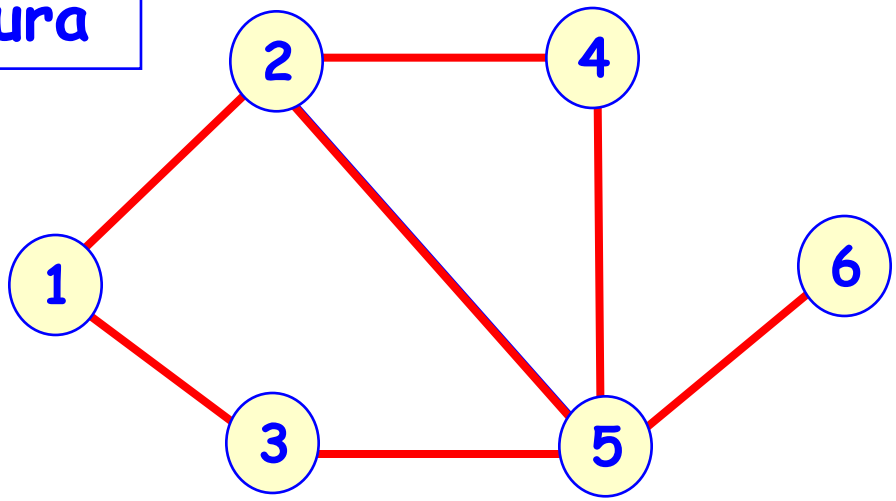
Grafos - Busca em Largura (BFS) - MA

```
BL(p,v):
    Esvaziar fila Q;  Enfilar (p,v);  pre[v] ← ++cpre;
    enquanto  Q não vazia:
        (p,t) ← elemento inicial de Q
        para w ← 1 até n incl.:
            se   E[t,w]=1  e pre[w] = 0:
                Enfilar (v,w)
                pre[w] ← ++cpre
        Desenfilar (p,t)

pre[*] ← 0
cpre ← 0
para  i  ← 1 até  n incl.:
    se   pre[i] = 0:
        BL(i,i)
```

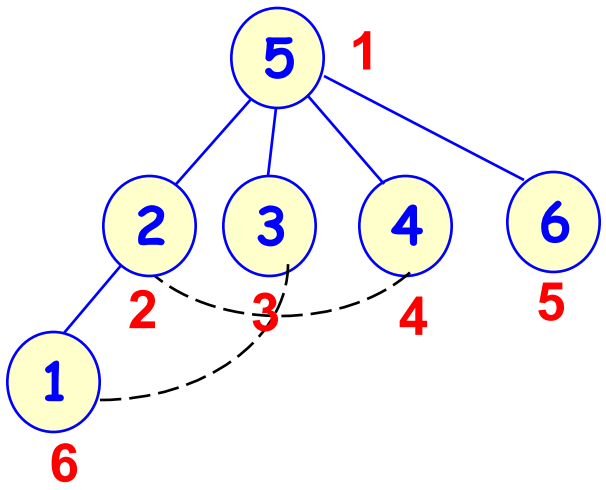
Complexidade: $O(n^2)$

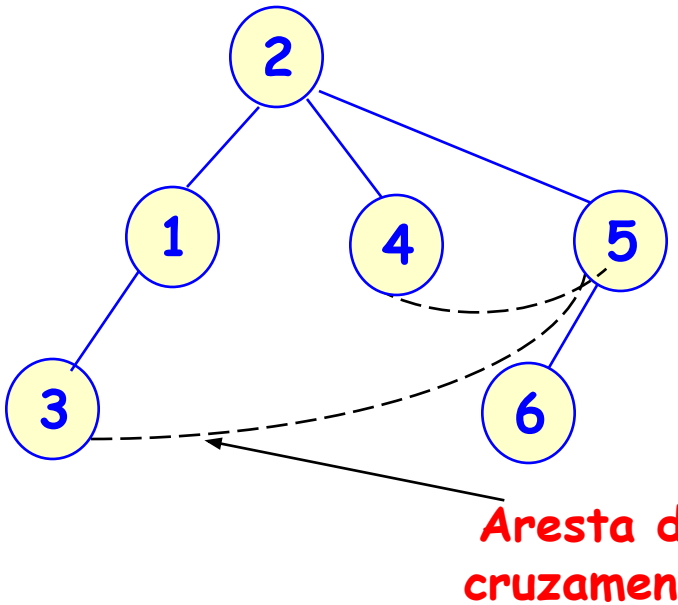
Grafos - Busca em Largura

Árvores de Largura:
Mostram a sequência das visitas. Pode haver mais de uma árvore de largura.



Árvores de Largura:





Grafos - Busca em Largura (BFS) - LA

```
BL(p,v):
    Esvaziar fila Q;  Enfilar (p,v);  pre[v] ← ++cpre;
    enquanto  Q não vazia:
        (p,v) ← elemento inicial de Q;  w ← A[v];
        enquanto  w ≠ nulo:
            se   pre[w.v] = 0:
                Enfilar (w, w.v)
                pre[w.v ] ← ++cpre
            w ← w.prox
        Desenfilar (p, t);

para  i  ← 1  até  n incl.:
    pre[i] ← 0
cpre ← 0
para  i  ← 1 até  n incl.:
    se  pre[i] = 0:
        BL(A[i],i)
```

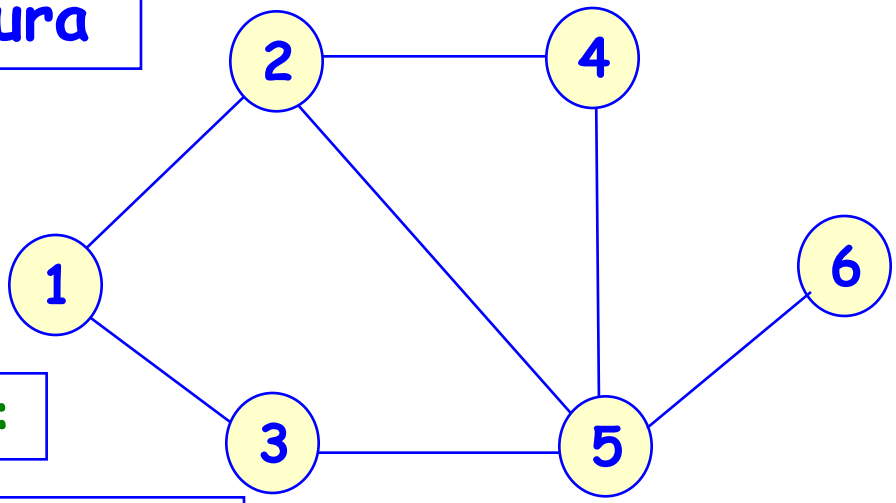
Complexidade: O(n+m)

Grafos - Busca em Largura (BFS) - LA (com vector)

```
BL(p,v):
    Esvaziar fila Q;  Enfilar (p,v);  pre[v] ← ++cpre;
    enquanto  Q não vazia:
        (p,t) ← elemento inicial de Q
        para j ← 0 até A[t].size():
            w ← A[t][j]
            se   pre[w] = 0:
                Enfilar (t,w)
                pre[w] ← ++cpre
        Desenfilar (p,t)

para  i  ← 1  até  n incl.:
    pre[i] ← 0
cpre ← 0
para  i  ← 1 até  n incl.:
    se  pre[i] = 0:
        BL(i,i)
```

Complexidade: O(n+m)

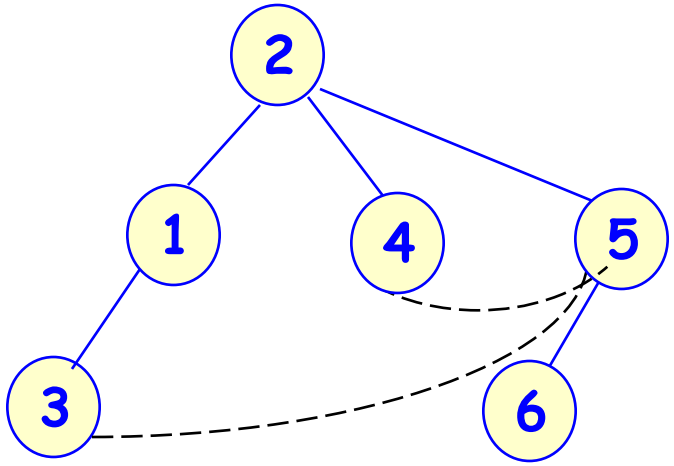
Grafos - Busca em Largura



Propriedades importantes da BL:

a) Durante a BL, os vértices são enfilados por ordem de distância ao vértice da raiz da busca.

b) Para qualquer nó da árvore de largura, o caminho da raiz até esse nó é um caminho mínimo entre os dois nós.

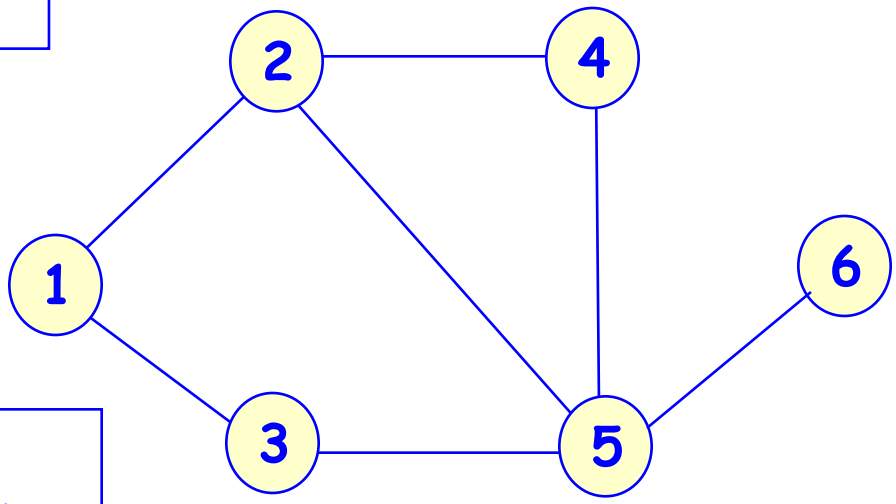


Distâncias para o vértice 2: 1, 4 e 5 estão à distância 1, 3 e 6, à distância 2.

Menor caminho de 2 a 6: 2 5 6

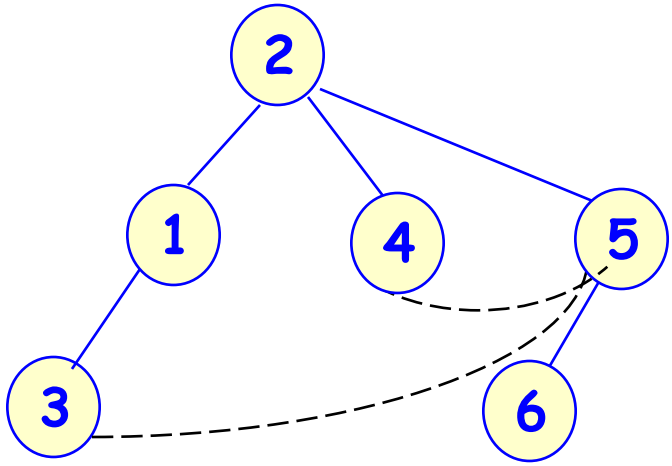
Grafos - Busca em Largura

Determinação do caminho mínimo entre a raiz e outro vértice:



a) Pode ser feito a partir da fila da BL, desde que se guarde, junto com cada vértice, a posição na fila do vértice pai na busca e a distância mínima.

b) Pode tb ser feito empilhando-se os caminhos mínimos durante a BL.



1	2	3	4	5	6
2/0/0	1/1 /1	4/1/1	5/1/1	3/2/2	6/4/2

Grafos - Busca em Largura

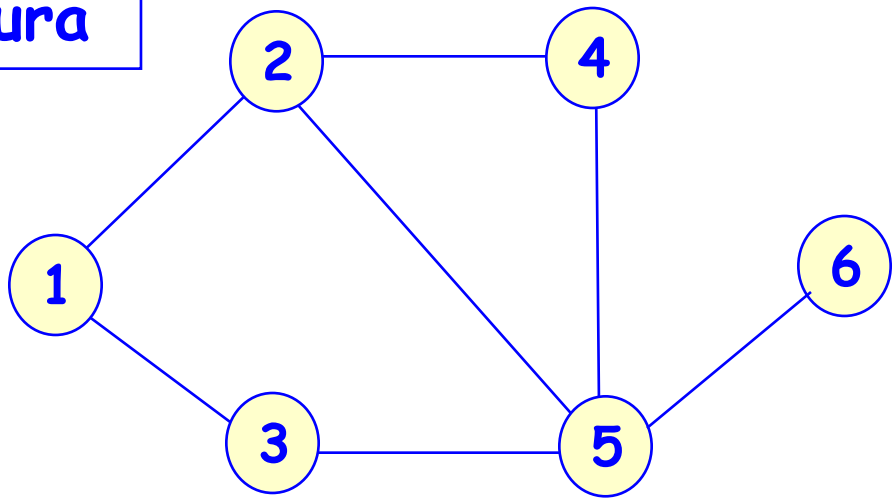
Determinação do caminho mínimo entre a raiz e outro vértice:

1	2	3	4	5	6
2/0	1/1	4/1	5/1	3/2	6/4

```
Caminho(p):   #p = a posição do vértice na Fila:
    j ← p
    enquanto  j ≠ 0:
          guardar Fil[j].v no vetor C
          j ← Fil[j].pospai
    escrever C invertido
```

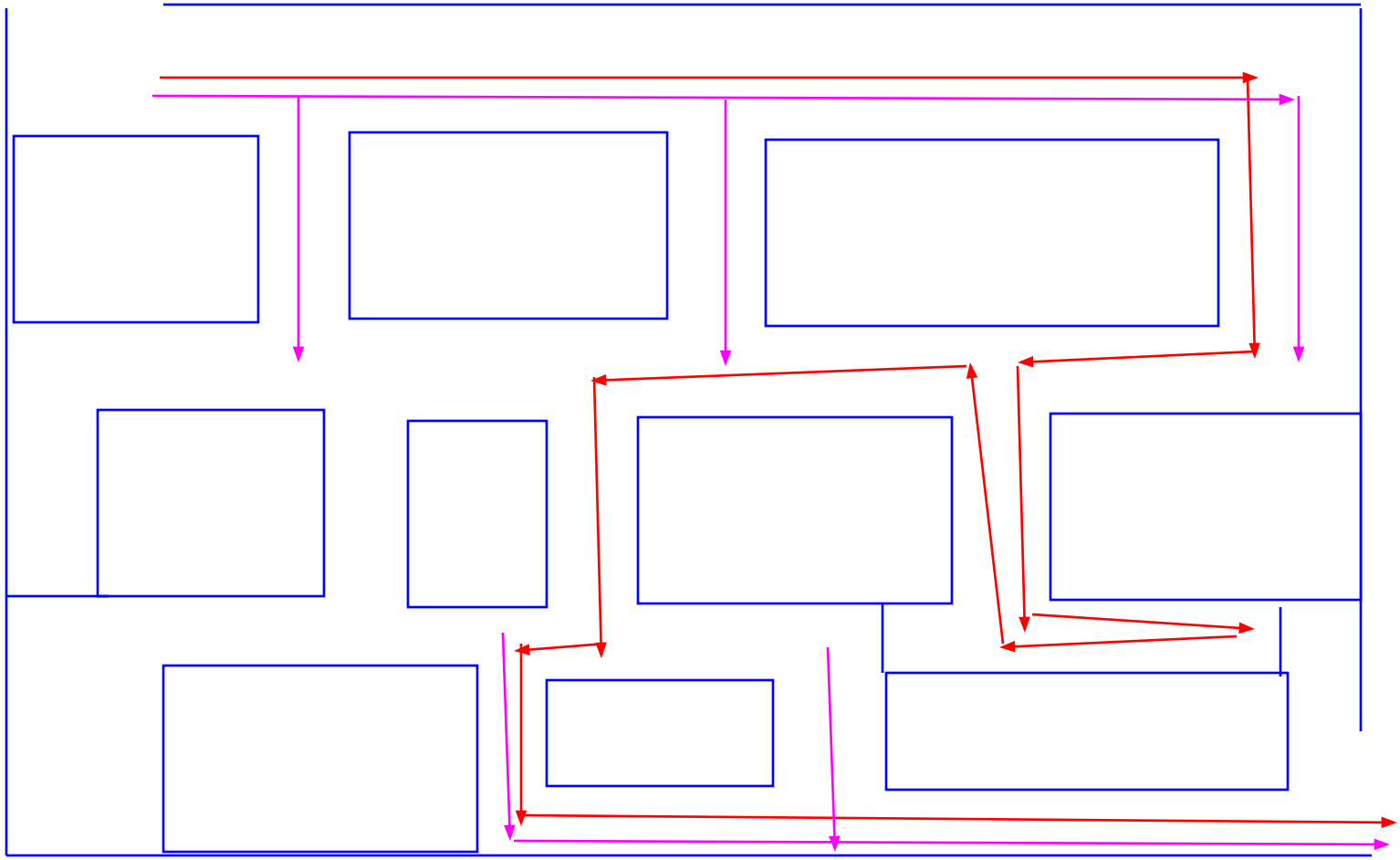
Grafos - Busca em Largura

Outras possibilidades da BL:
Pequenas modificações no algoritmo permitem descobrir propriedades do grafo

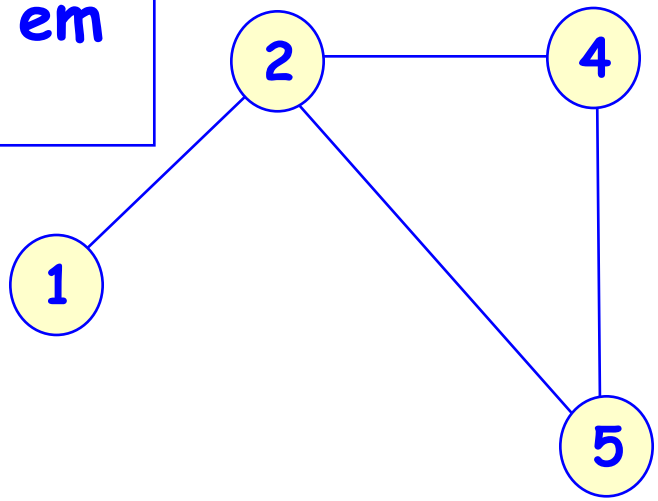


- Visita a todas as arestas
- Obtenção dos graus dos vértices
- Determinação se o grafo é conexo
- Determinação dos componentes conexos
- Determinação de existência de ciclos
- Descoberta de elementos estruturais (pontes, blocos etc)
- Descoberta se o grafo é bipartido, ciclo, árvore, completo, regular

Grafos - Comparação Profundidade x Largura



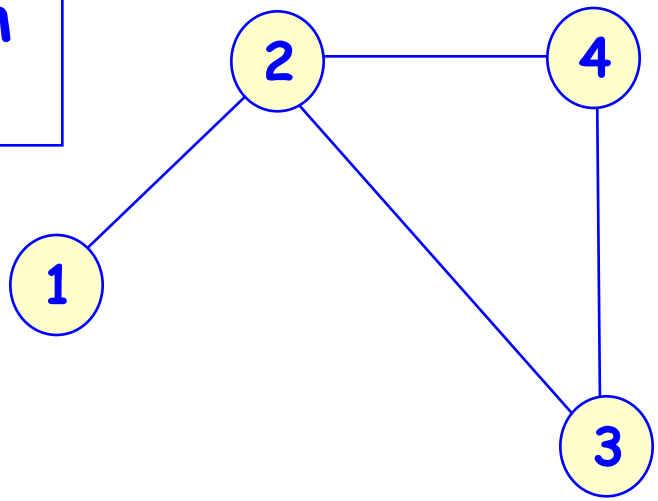
Grafos - Busca Irrestrita em Profundidade



Idéia:
Visitar, a cada passo, algum vizinho do vértice mais recentemente visitado

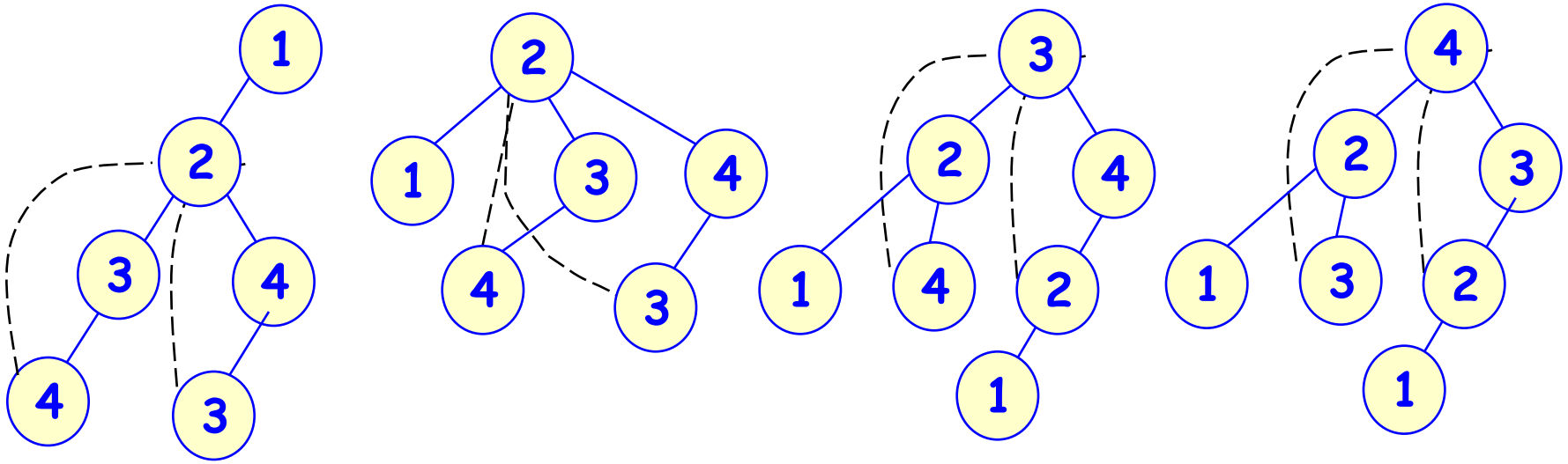
```
BPI(v):
    Marcar v
    para vizinhos w de v:
        se w não marcado:
            BPI(w)
    Desmarcar v

Desmarcar vértices
para i ← 1 até n incl.:
    se i não marcado:
        BPI(i)
```

Grafos - Busca Irrestrita em Profundidade



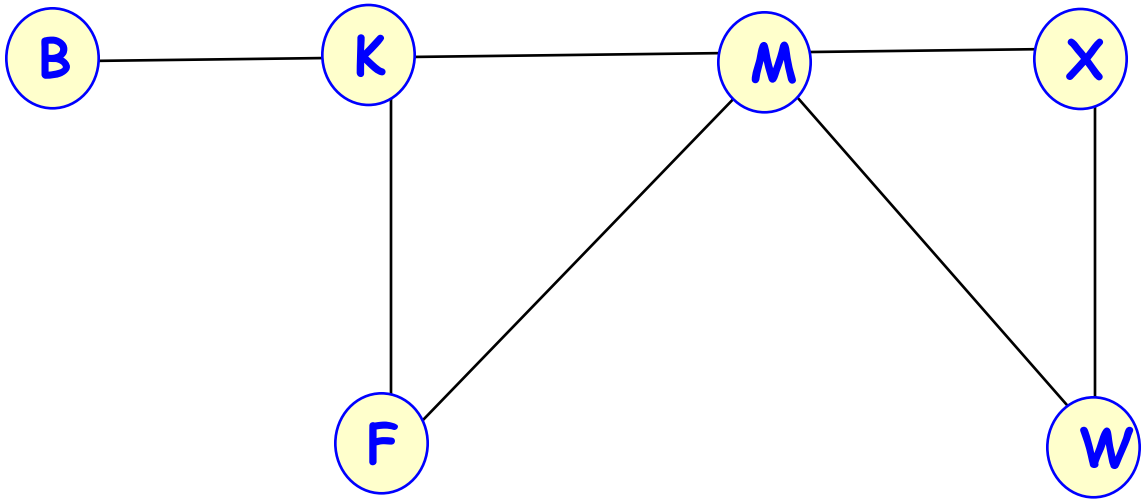
Árvores de Busca Irrestrita: Mostram a sequência das visitas. Pode haver mais de uma árvore.

Floresta de Busca Irrestrita:



Grafos - Busca Irrestrita em Profundidade

Exercício recomendado : Mostrar uma árvore de profundidade irrestrita iniciando no vértice de letra mais próxima de um dos nomes do grupo.



FIM