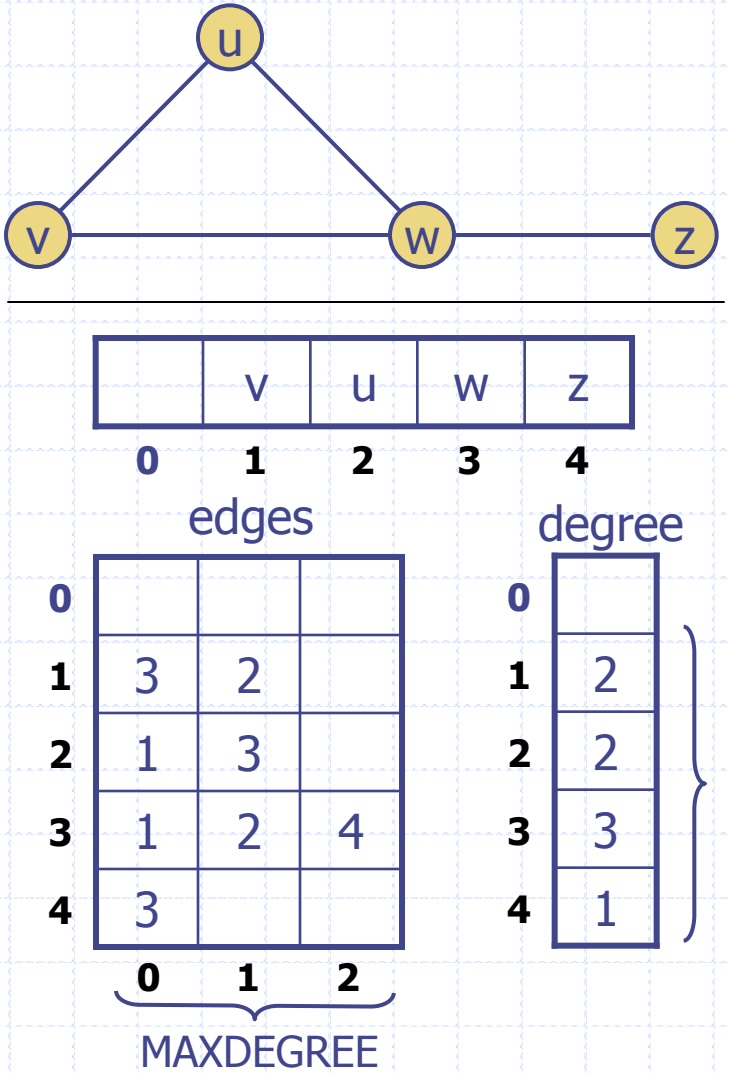
SCC-210
Algoritmos Avançados

Capítulo 9
Grafos

Adaptado por João Luís G. Rosa

Representação (Skiena & Revilla, 2003)

- Vértices rotulados:
  - Chaves (índices) são associadas aos vértices
- Arestas sem elementos.
- Matrizes e vetores dimensionados para:
  - No. de vértices (MAXV)
  - Grau (MAXDEGREE)
- Permite representar multi-grafos e ter o desempenho da lista de adjacências sem alocação dinâmica.



Representação (Skiena & Revilla, 2003)

```
/*  graph.h */

#define MAXV          100      /* maximum number of vertices */
#define MAXDEGREE     50       /* maximum outdegree of a vertex */

typedef struct {
    int edges[MAXV+1][MAXDEGREE]; /* adjacency info */
    int degree[MAXV+1];
    int nvertices;
    int nedges;
} graph;
```

```
#include "graph.h"
initialize_graph(graph *g) {
    int i,j;  /* counters */

    g -> nvertices = 0;
    g -> nedges = 0;

    for (i=1; i<=MAXV; i++) {
        g->degree[i] = 0;
        for (j=1; j<=MAXDEGREE; j++)
            g->edges[i][j] = 0;
    }
}
```

Representação (Skiena & Revilla, 2003)

```
insert_edge(graph *g, int x, int y, bool directed) {
    if (g->degree[x] > MAXDEGREE)
        printf("Warning: insertion(%d,%d) exceeds max degree\n",x,y);

    g->edges[x][g->degree[x]] = y;
    g->degree[x] ++;

    if (directed == FALSE) insert_edge(g,y,x,TRUE);
    else g->nedges ++;
}
```

Duas arestas direcionadas, (x,y) e (y,x), representando uma aresta não-direcionada

```
delete_edge(graph *g, int x, int y, bool directed) {
    int i;                  /* counter */
    for (i=0; i<g->degree[x]; i++)
        if (g->edges[x][i] == y) {
            g->degree[x] --;
            g->edges[x][i] = g->edges[x][g->degree[x]];
            if (directed == FALSE)
                delete_edge(g,y,x,TRUE);
            return;
        }
    printf("Warning: deletion(%d,%d) not found in g.\n",x,y);
}
```

Busca em Profundidade (DFS)

```
dfs(graph *g, int v) {

    int i;                  /* counter */
    int y;                  /* successor vertex */

    discovered[v] = TRUE;
    process_vertex(v);

    for (i=0; i<g->degree[v]; i++) {
        y = g->edges[v][i];
        if (discovered[y] == FALSE) {
            parent[y] = v;
            dfs(g,y);
        } else if (processed[y] == FALSE) process_edge(v,y);
        if (finished) return;  /* allow for search termination */
    }
    processed[v] = TRUE;
}
```

Busca em Profundidade - Complexidade

- Quando uma **matriz de adjacências** é utilizada, o procedimento DFS (*Depth-First Search*) requer $\mathcal{O}(|V|^2)$.
- Quando uma **lista de adjacências** é utilizada, a busca profundidade requer $\mathcal{O}(|V|+|A|)$.
- Repare que DFS é ótima para listas de adjacências (por que?).

Algoritmos baseados na Busca em Profundidade

- Teste de existência de ciclos (linear);
- Teste de conectividade fraca (linear);
- Encontrar componentes fracamente conexos (linear);
- Teste de conectividade forte (linear);
- Encontrar componentes fortemente conexos (linear);
- Fechamento transitivo ($\mathcal{O}(|V|(|V|+|A|))$);
- Ordenação topológica (linear);
- Identificação de grafos bi-coloridos, bipartidos ou com ciclos de tamanho ímpar (linear);
- Identificação de pontes (linear);
- Identificação de vértices de articulação (linear).

Teste de Existência de Ciclos

- A busca em profundidade pode ser usada para verificar se um grafo é acíclico ou se contém um ou mais ciclos.
- Se uma aresta de retorno é encontrada durante a busca em profundidade em *G*, então o grafo é cíclico.

Teste de Existência de Ciclos

- Isso sugere as seguintes especializações de DFS para buscar ciclos:

```
process_vertex(int v) {
}

process_edge(int x, int y) {
    if (parent[x] != y) {    /* found back edge! */
        printf("Cycle from %d to %d:",y,x);
        find_path(y,x,parent);
        printf("\n\n");
        finished = TRUE;
    }
}
```

Encontrar Componentes Fracamente Conexos

- Componentes conexos de um grafo é um conjunto máximo de vértices tal que existe um caminho entre todos os pares de vértices.
  - Existem problemas aparentemente complexos, como testar se 15-puzzle ou cubo mágico podem ser solucionados a partir de qualquer posição, que se resumem a um teste de conectividade.
  - Componentes fracamente conexos podem ser facilmente encontrados a partir de uma busca em largura ou profundidade.
  - Basta iniciar a busca e verificar se existem vértices não descobertos. Reiniciar a busca a partir de um vértice não descoberto até que todos sejam descobertos.

Encontrar Componentes Fracamente Conexos

```
connected_components(graph *g) {
    int c, i;

    initialize_search(g);

    c = 0;
    for (i = 1; i <= g->vertices; i++) {
        if (discovered[i] == false) {
            c++;
            printf("Component %d:", c);
            dfs(g, i);
            printf("\n");
        }
    }
}
```

Encontrar Componentes Fracamente Conexos

```
connected_components(graph *g) {
    int c, i;

    initialize_search(g);

    c = 0;
    for (i = 1; i <= g->vertices; i++) {
        if (discovered[i] == false) {
            c++;
            printf("Component %d:", c);
            dfs(g, i);
            prin
        }
    }
}
```

```
process_vertex(int v) {
    printf(" %d", v);
}

process_edge(int x, int y) {
}
```

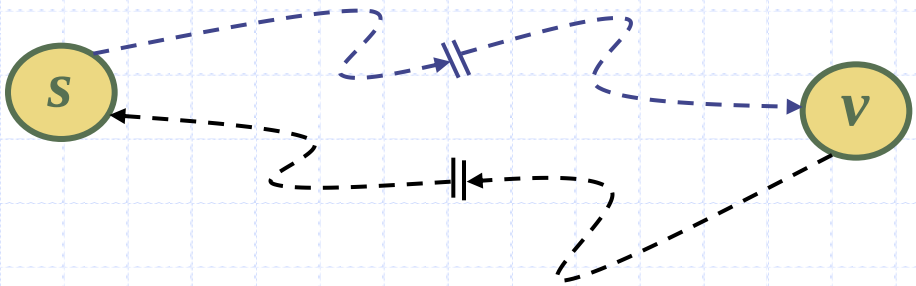
Teste de Conexão Forte

- Podemos executar DFS ou BFS (*breadth-first search*) múltiplas vezes e verificar se o grafo é fortemente conexo.

- Verifica-se se a partir de cada vértice tomado como origem todos os demais vértices são alcançáveis ou não: $\mathcal{O}(|V| (|V| + |A|))$ no pior caso.

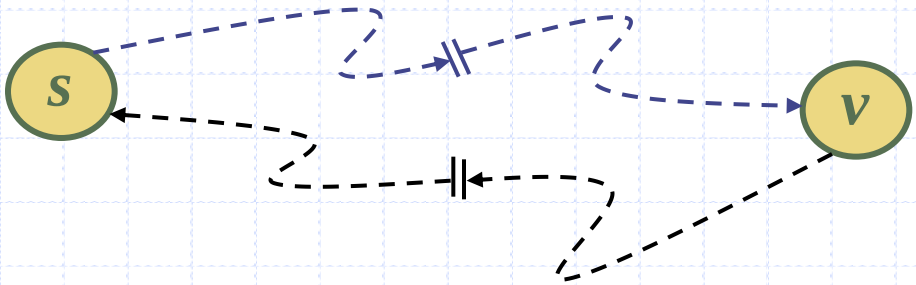
Teste de Conexão Forte Eficiente

- Basta que exista um único vértice de um dígrafo G que alcance qualquer outro e que seja alcançável por qualquer outro para que todos os vértices de G possuam essa mesma propriedade através dele $\Rightarrow$ G fortemente conexo.
  - Note que um vértice s alcança qualquer outro v e é alcançável por v se e somente se s alcança v em ambos os dígrafos G e G^T (transposto), pois o ciclo direcionado entre s e v se mantém (invertido) em G^T.
  - Logo, basta tomar qualquer vértice e executar DFS ou BFS duas vezes, uma sobre o dígrafo original G e a outra sobre G^T: $\mathcal{O}(|V| + |A|)$.



Encontrar Componentes Fortemente Conexos

- Podemos utilizar a propriedade anterior também para calcular os componentes fortemente conexos de G:
  - Toma-se um vértice v e calcula-se o componente fortemente conexo que inclui v como todos aqueles vértices (e as respectivas arestas) que são alcançados por v em ambos os dígrafos G e G^T.
  - Faz-se isso sucessivas vezes, sempre a partir de um vértice não presente no componente fortemente conexo anterior.



Fechamento Transitivo

- Um fechamento transitivo F é um grafo construído a partir de $G = (V, A)$, tal que se existe um caminho entre 2 vértices em G, então existe uma aresta em F.
- É simples calcular o fechamento transitivo de um dígrafo G via DFS ou BFS executando o percurso a partir de cada vértice s de G e inserindo uma aresta direcionada adicional ligando a origem s a cada vértice alcançável a partir de s (se esta aresta já não existir).
  - Tempo = $|V|$ percursos $\Rightarrow$ $\mathcal{O}(|V|(|V| + |A|))$ no pior caso.
  - Superior a algoritmo de Floyd-Warshall ($\mathcal{O}(|V|^3)$) se G não for denso.

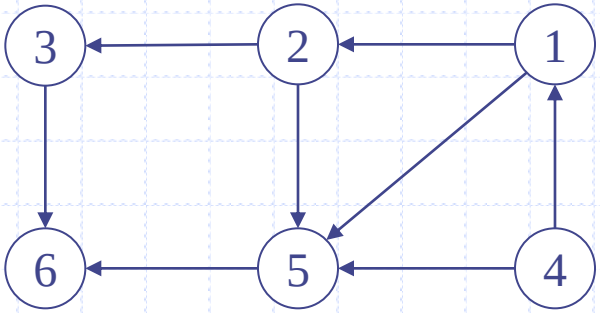
Ordenação Topológica

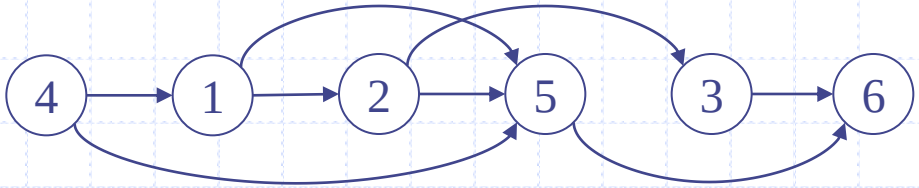
- Um grafo direcionado acíclico é também chamado de **DAG** (*directed acyclic graph*).
- Um DAG é diferente de uma árvore, uma vez que as árvores são não direcionadas.
- DAGs podem ser utilizados, por exemplo, para indicar precedências entre eventos.

Ordenação Topológica

- A ordenação topológica é uma ordenação linear de todos os vértices, tal que se *G* contém uma aresta *(u,v)* então *u* aparece antes de *v*.
- Pode ser vista como uma ordenação de seus vértices ao longo de uma linha horizontal de tal forma que todas as arestas estão direcionadas da esquerda para a direita.

Ordenação Topológica

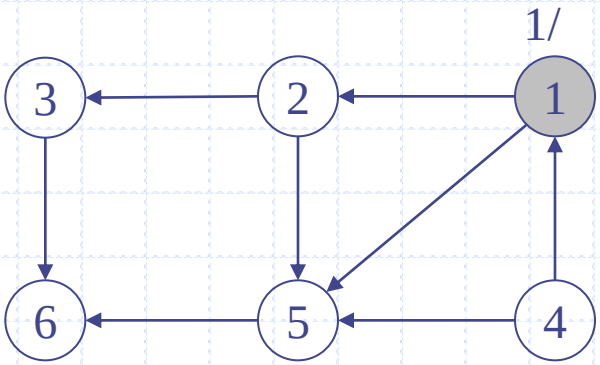




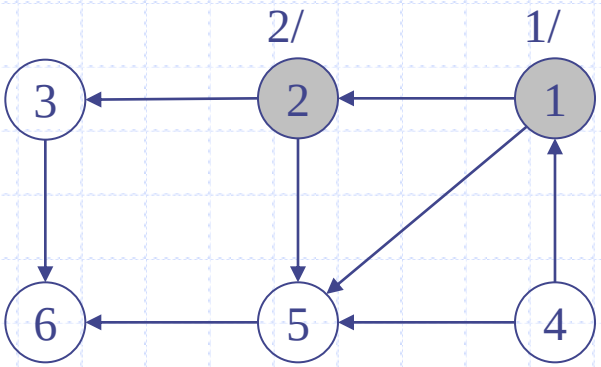
Ordenação Topológica

- A ordenação topológica de um DAG pode ser obtida utilizando-se uma busca em profundidade.
- Para isso deve-se fazer o seguinte algoritmo:
  1. Faça uma busca em profundidade;
  2. Quando um vértice é processado, insira-o numa fila de vértices;
  3. Retorne a fila de vértices.

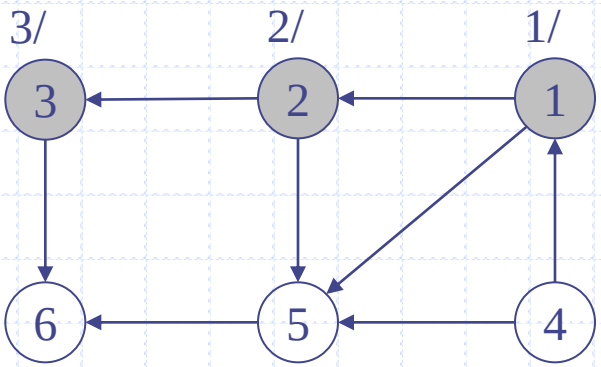
Ordenação Topológica



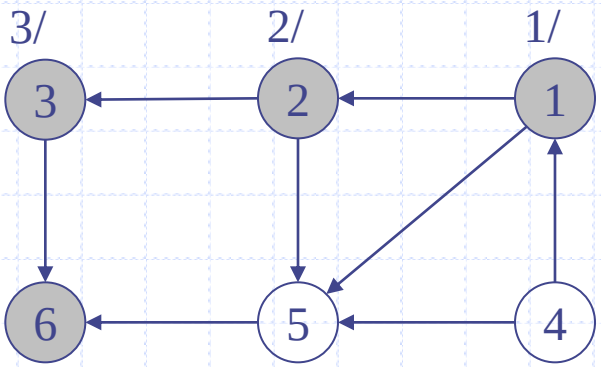
Ordenação Topológica



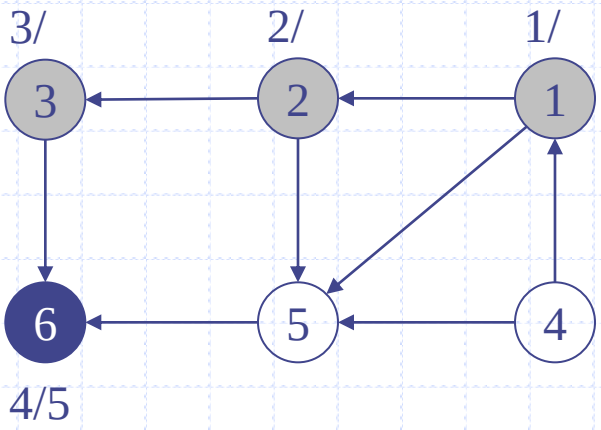
Ordenação Topológica



Ordenação Topológica

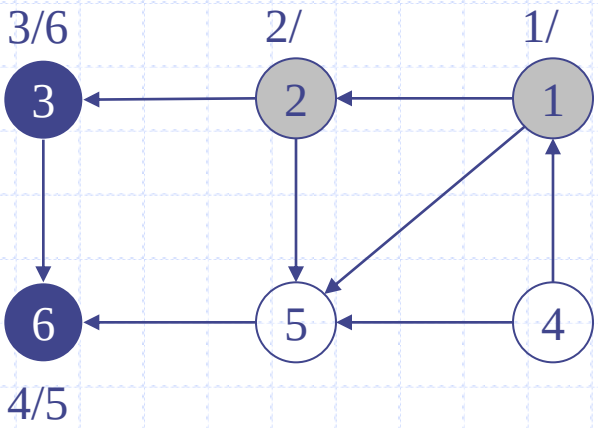


Ordenação Topológica



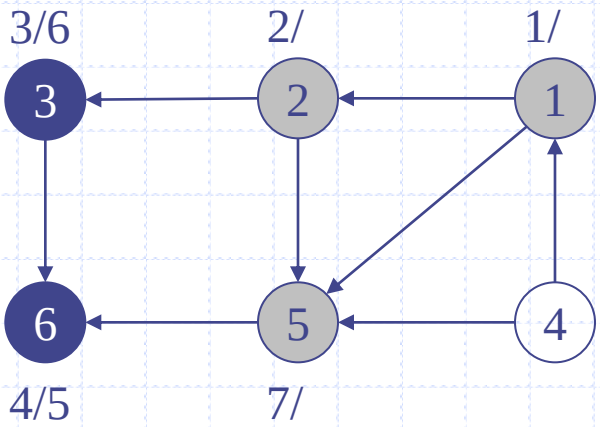
6

Ordenação Topológica



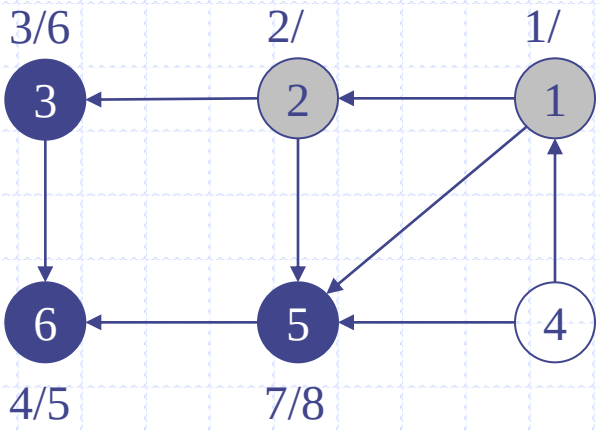
3 6

Ordenação Topológica



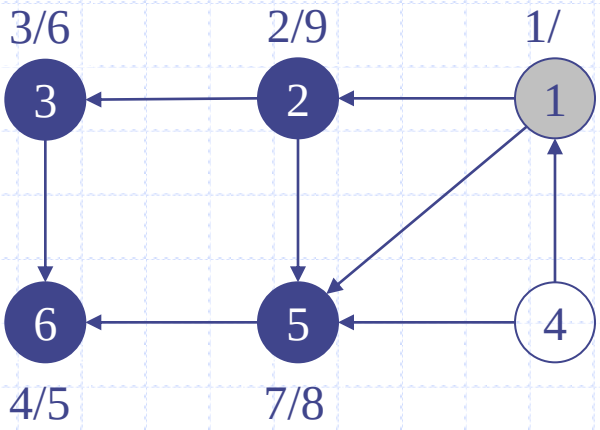
3 6

Ordenação Topológica



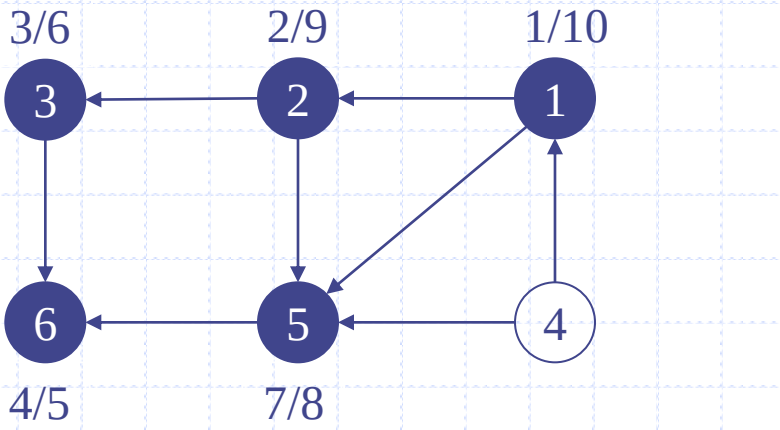
5 3 6

Ordenação Topológica

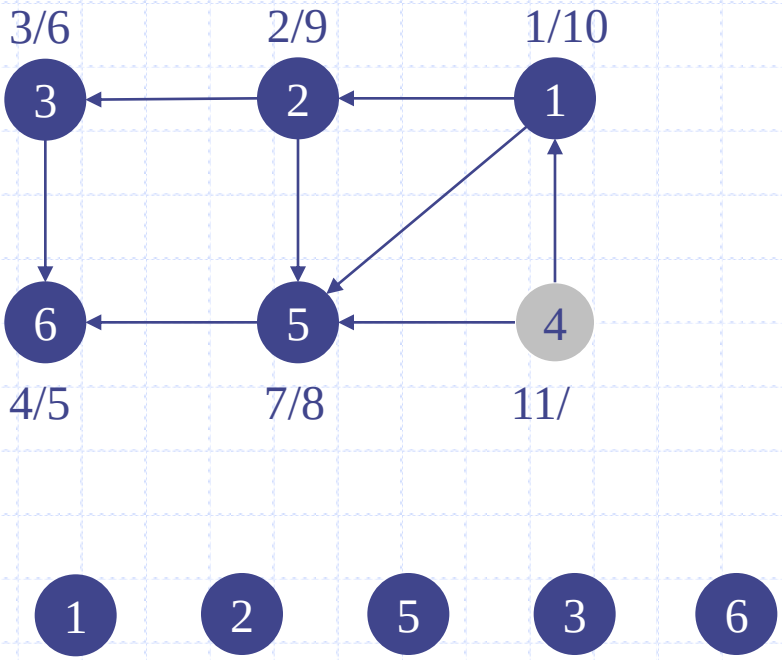


2 5 3 6

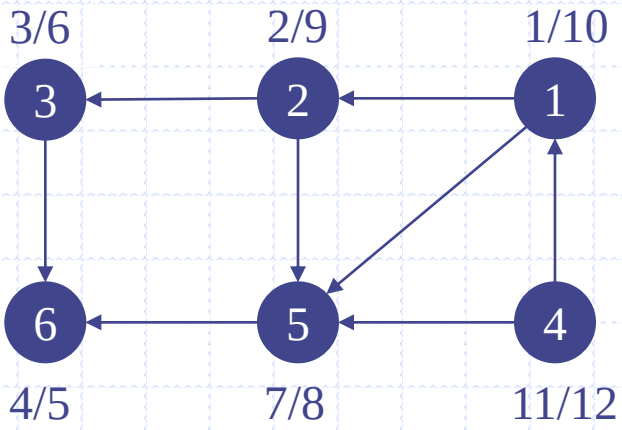
Ordenação Topológica



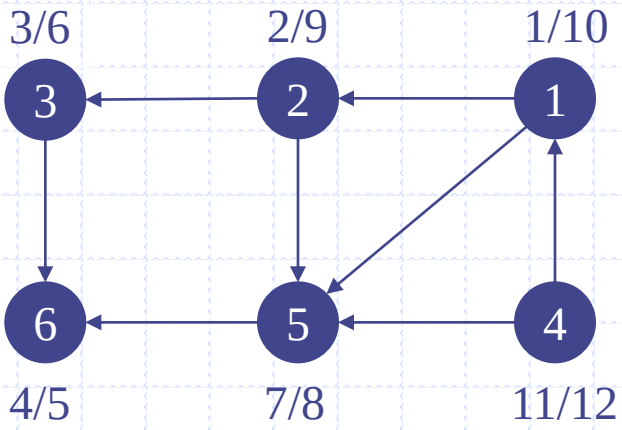
Ordenação Topológica

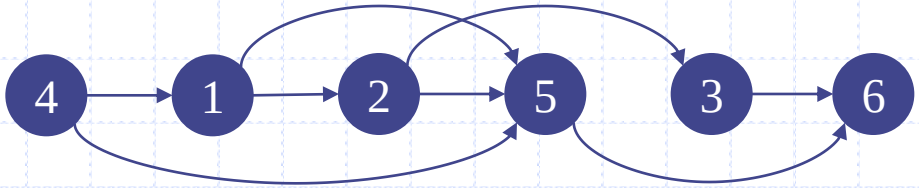


Ordenação Topológica



Ordenação Topológica



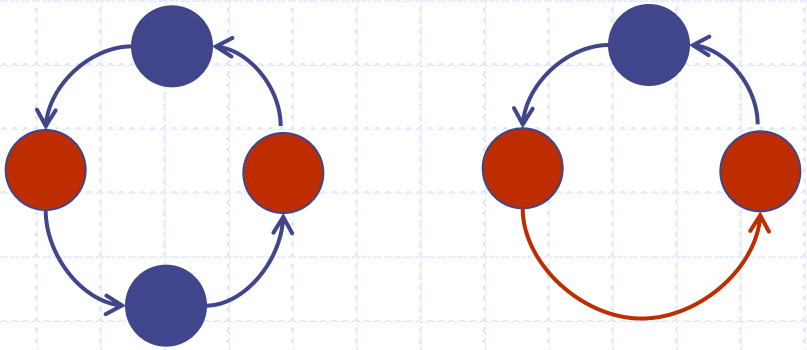


Ordenação Topológica

- A complexidade do algoritmo de ordenação topológica em um DAG é a mesma da busca em profundidade, ou seja:
  - $\mathcal{O}(|V|^2)$ para matrizes de adjacência, e;
  - $\mathcal{O}(|V|+|A|)$ para listas de adjacência.
- Inserir um elemento na fila é $\mathcal{O}(1)$.

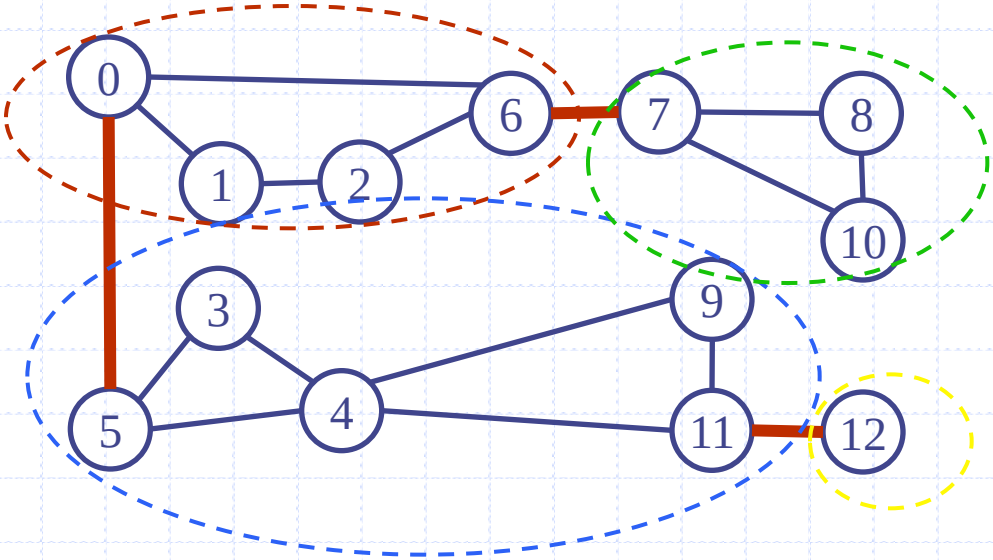
Grafos bi-coloridos, bipartidos ou com ciclos de tamanho ímpar

- Esses três problemas são equivalentes:
  - Os dois primeiros são diferentes nomenclaturas para o mesmo problema;
  - Qualquer grafo com um ciclo de tamanho ímpar é claramente não bi-colorível.

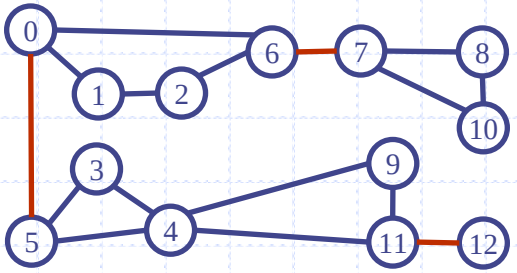


Pontes

- Uma **ponte** em um grafo é uma aresta que, se removida, separaria um grafo conexo em dois grafos disjuntos.

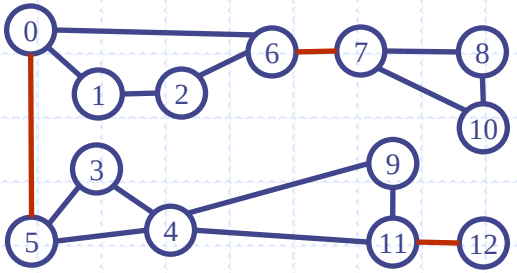


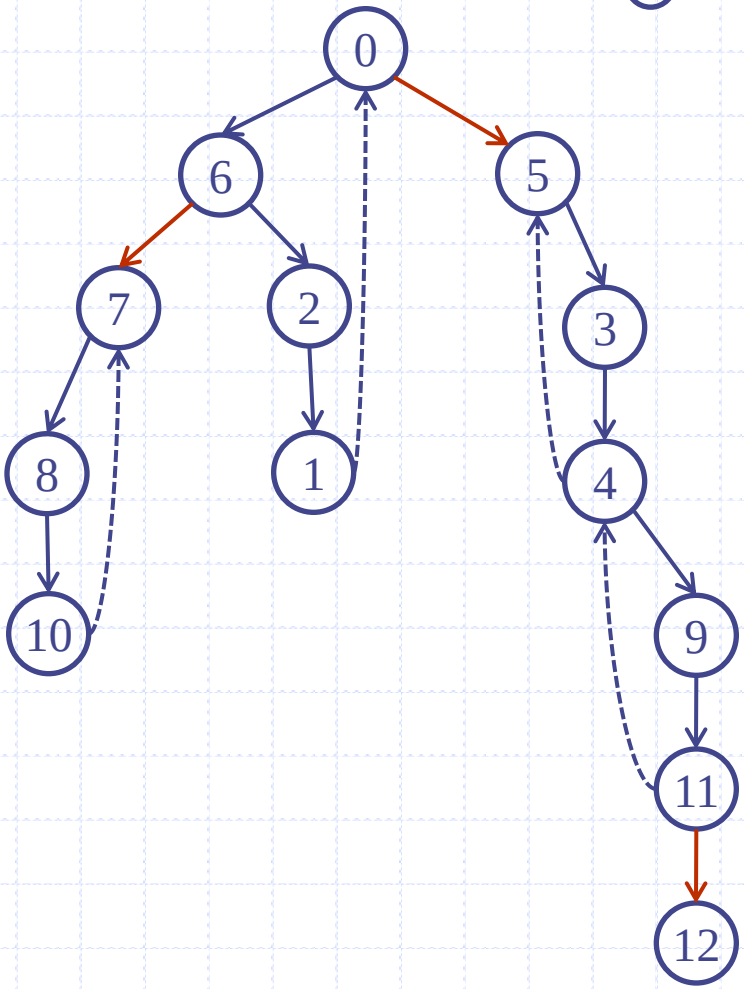
Pontes



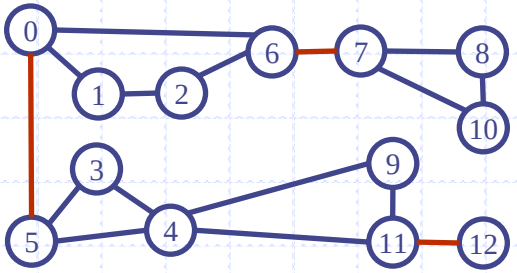
- Em uma árvore de busca em profundidade, uma aresta *v*-*w* é uma ponte se e somente se não existem arestas de retorno que conectam um descendente de *w* a um ancestral de *v*.

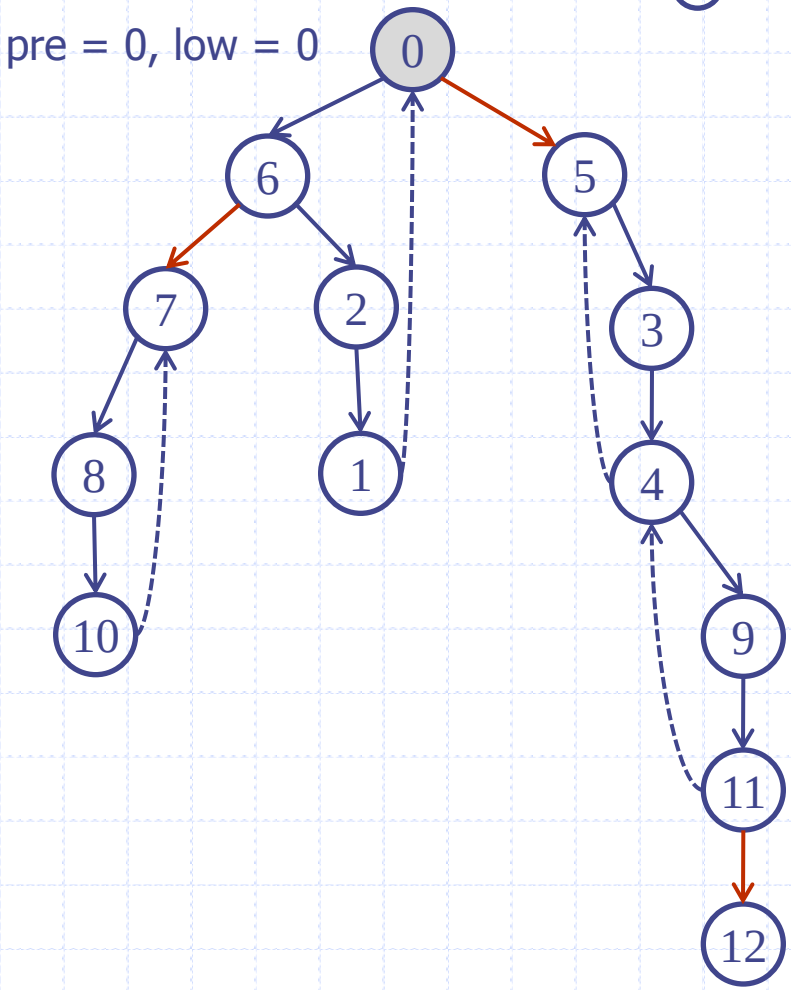
Pontes



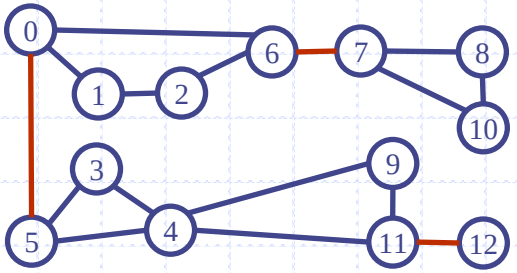


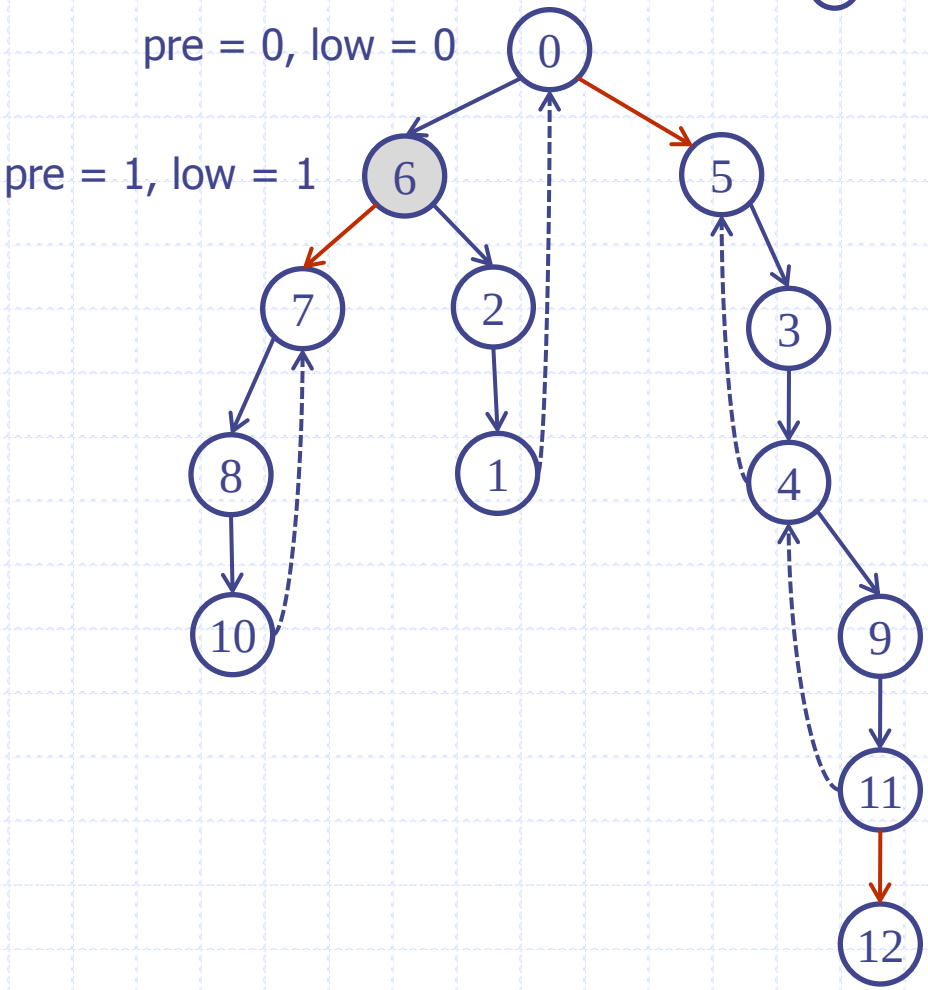
Pontes



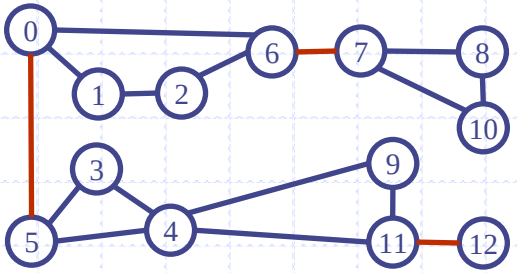


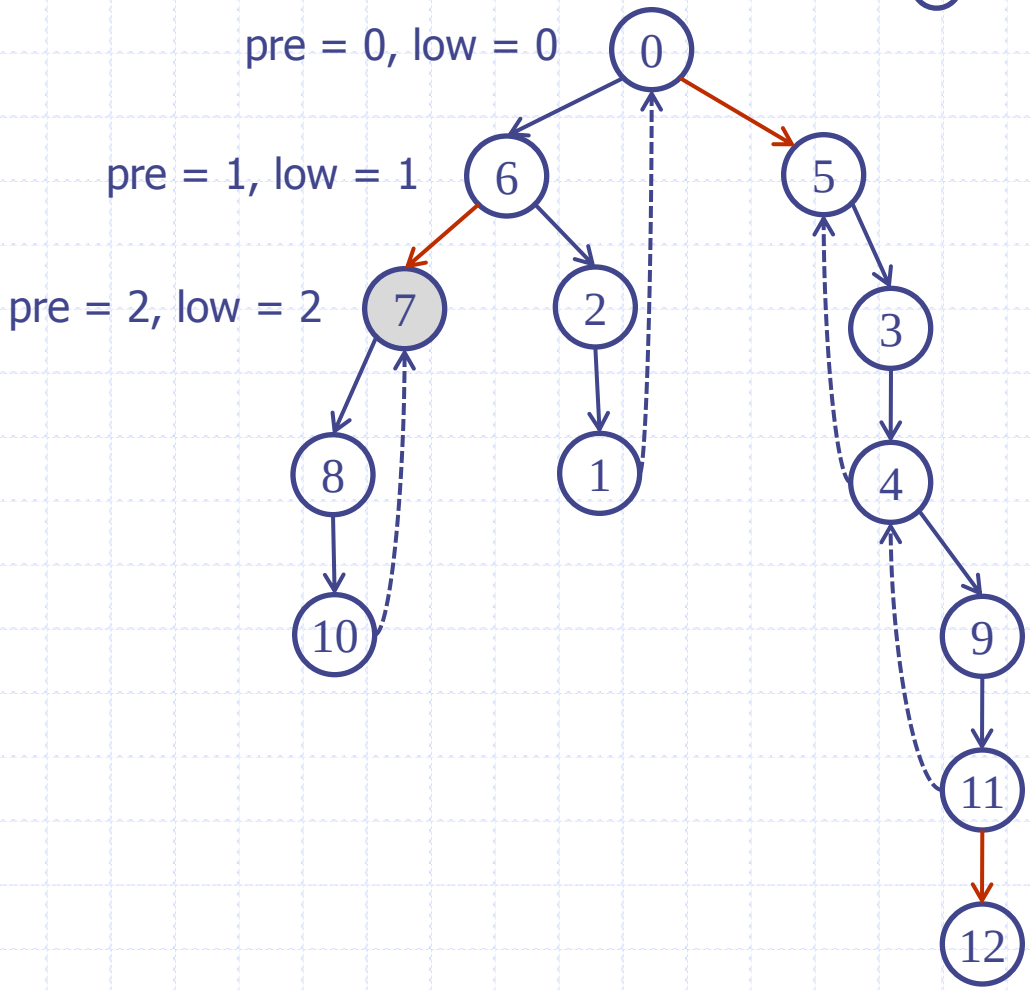
Pontes



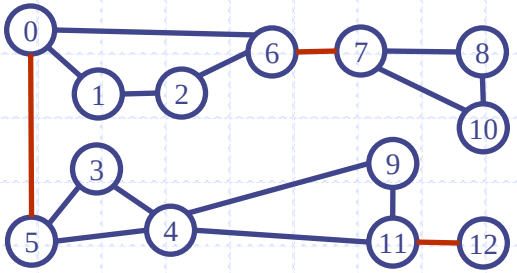


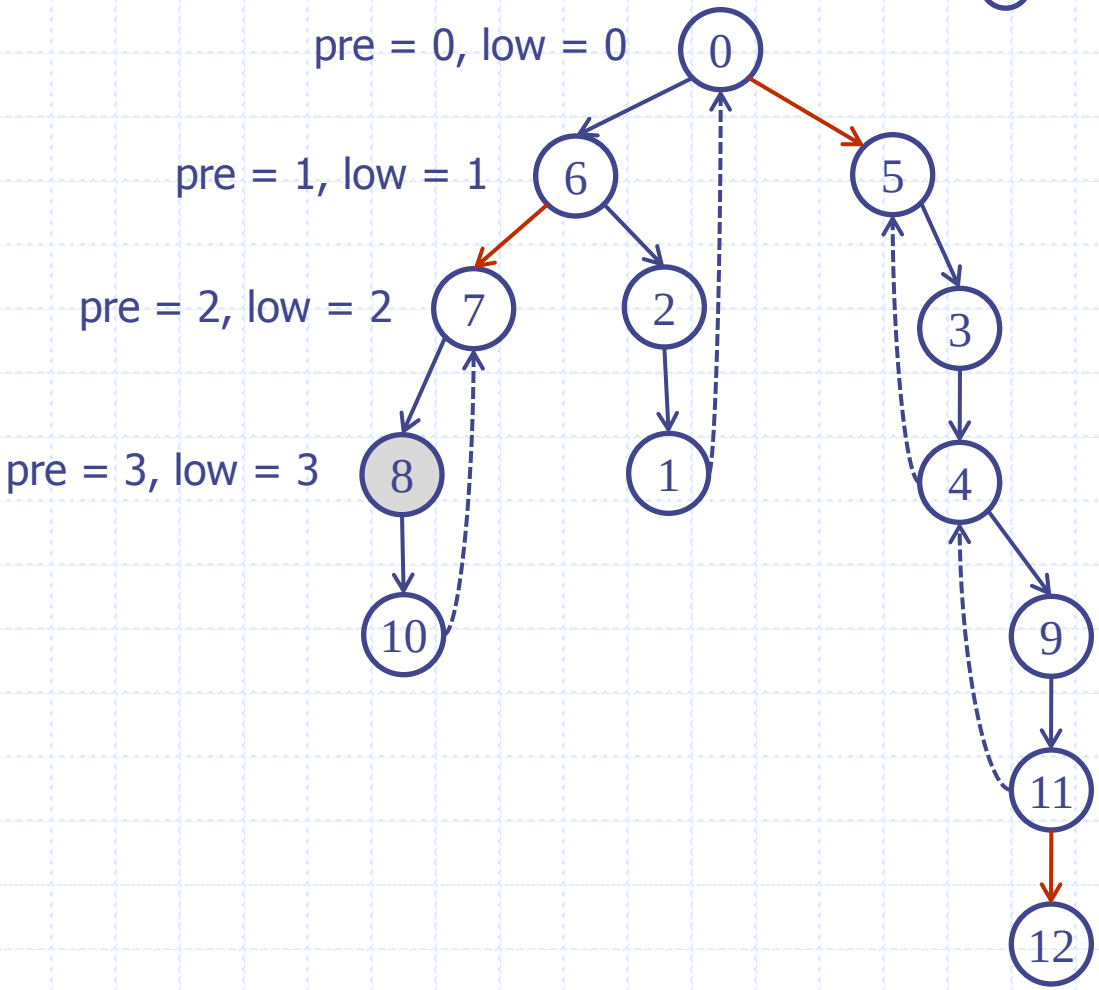
Pontes


0
6
7
8
1
2
10
3
9
5
4
11
12

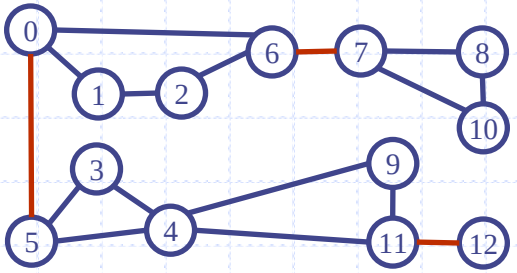

pre = 0, low = 0
0
pre = 1, low = 1
6
5
pre = 2, low = 2
7
2
3
8
1
4
10
9
11
12

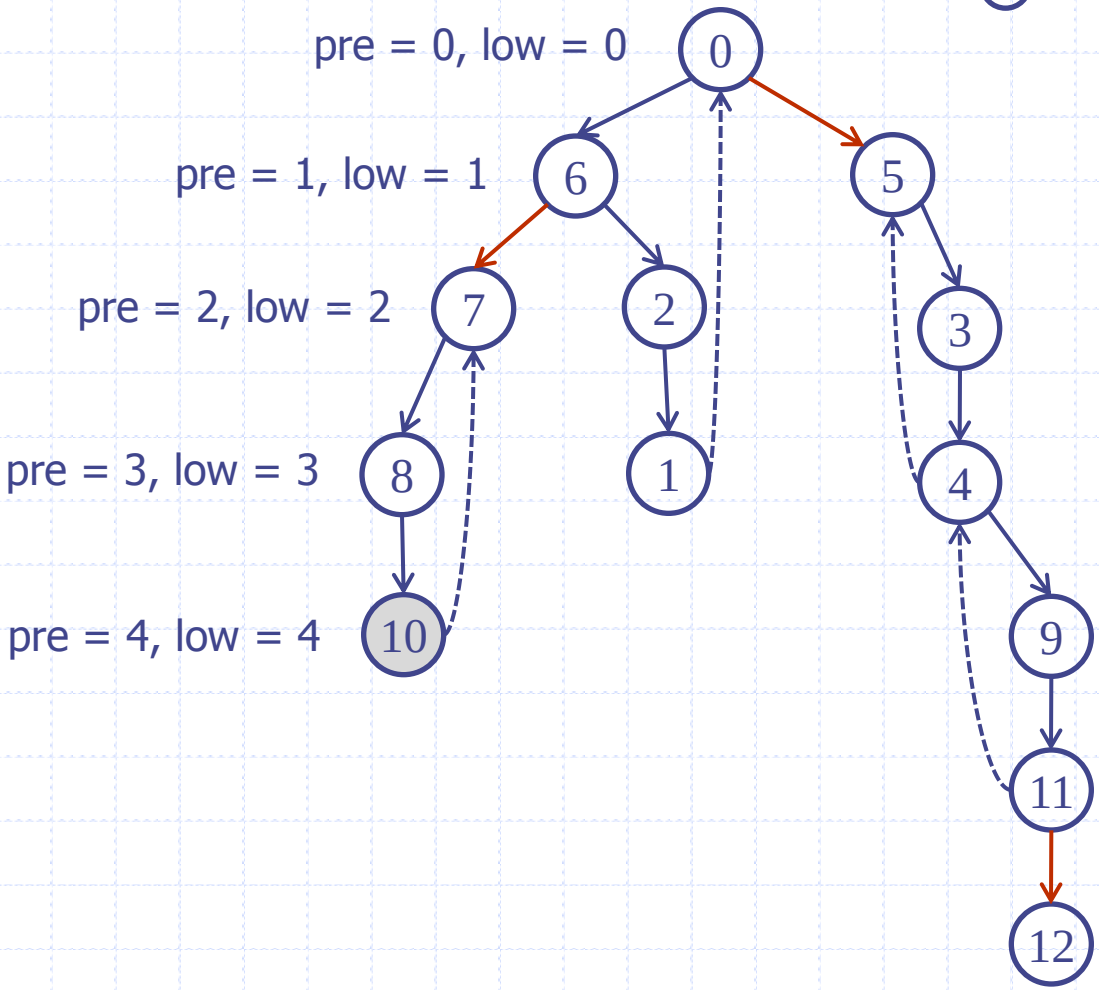
Pontes


0
6
7
8
1
2
10
3
9
5
4
11
12

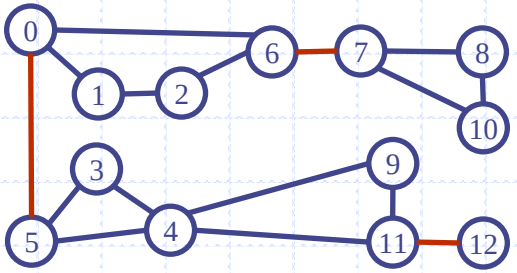

pre = 0, low = 0
0
pre = 1, low = 1
6
5
pre = 2, low = 2
7
2
3
pre = 3, low = 3
8
1
4
10
9
11
12

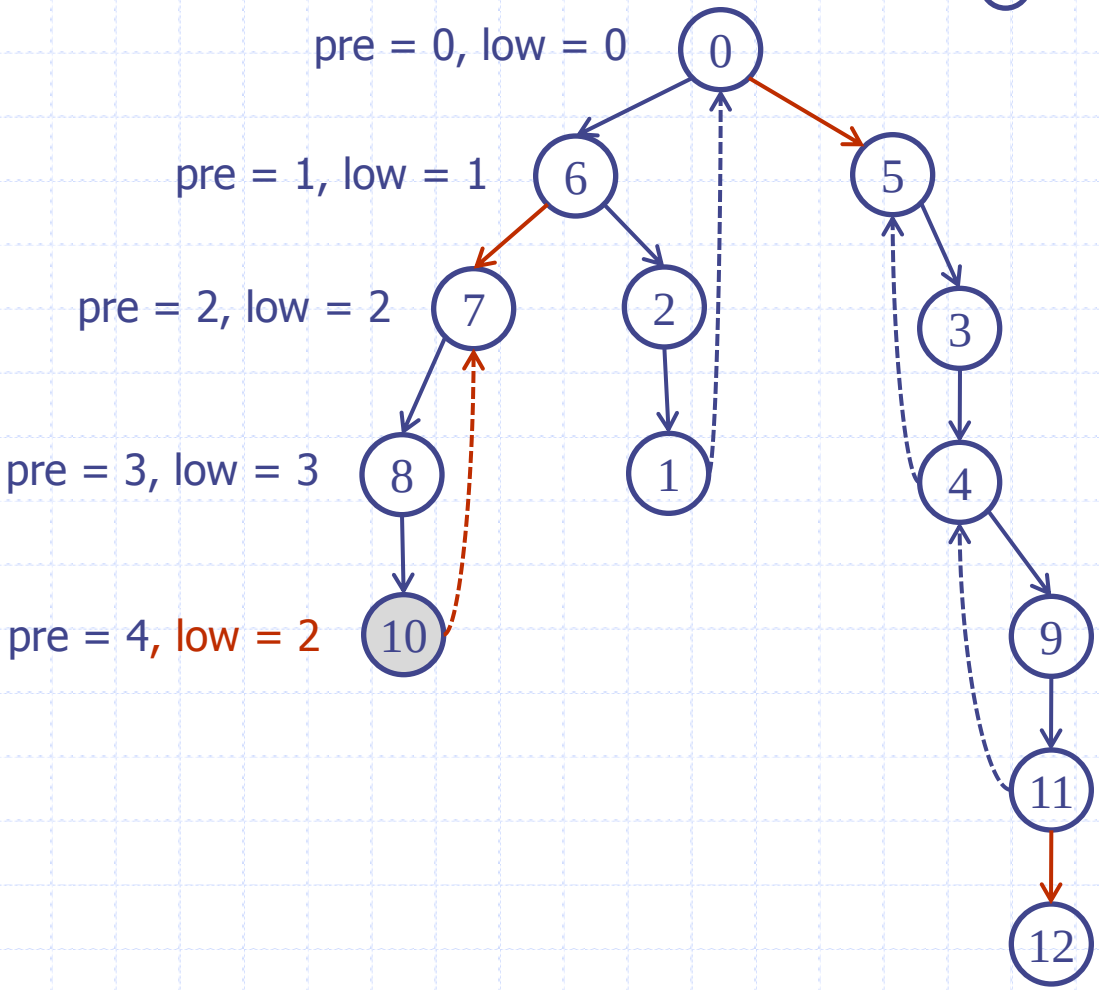
Pontes


0
6
7
8
1
2
10
3
9
5
4
11
12

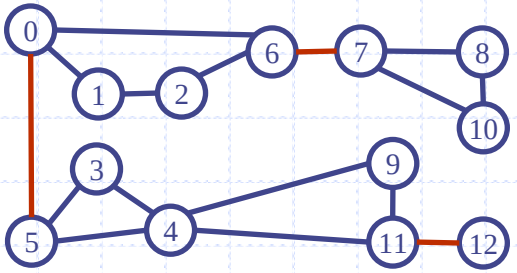

pre = 0, low = 0
pre = 1, low = 1
pre = 2, low = 2
pre = 3, low = 3
pre = 4, low = 4
0
6
5
7
2
3
8
1
4
10
9
11
12

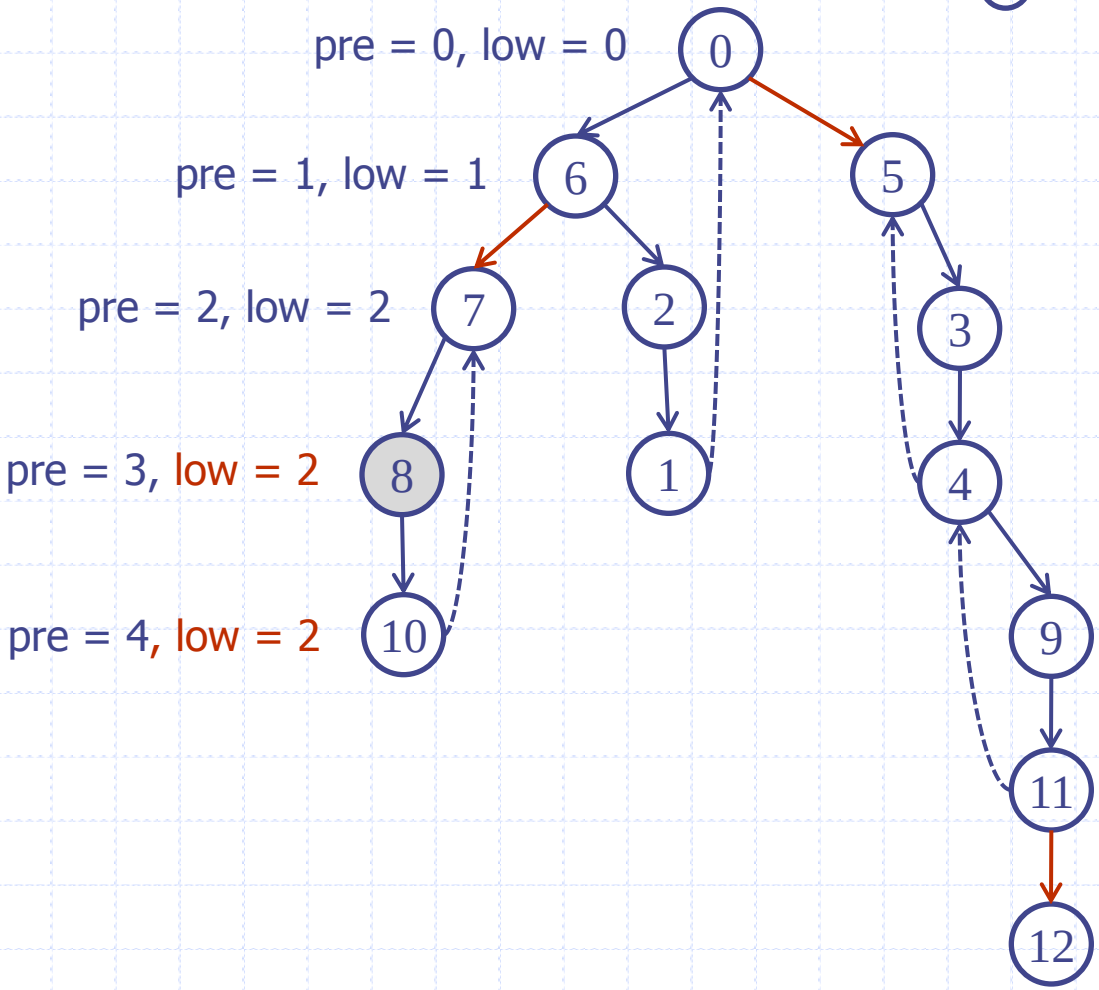
Pontes


0
6
7
8
1
2
10
3
9
5
4
11
12

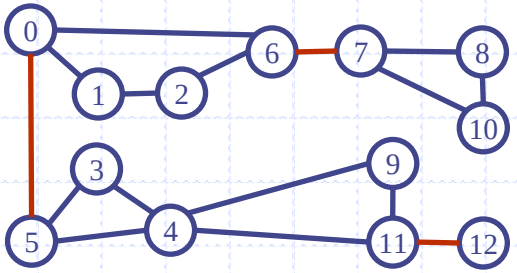

pre = 0, low = 0
0
pre = 1, low = 1
6
5
pre = 2, low = 2
7
2
3
pre = 3, low = 3
8
1
4
pre = 4, low = 2
10
9
11
12

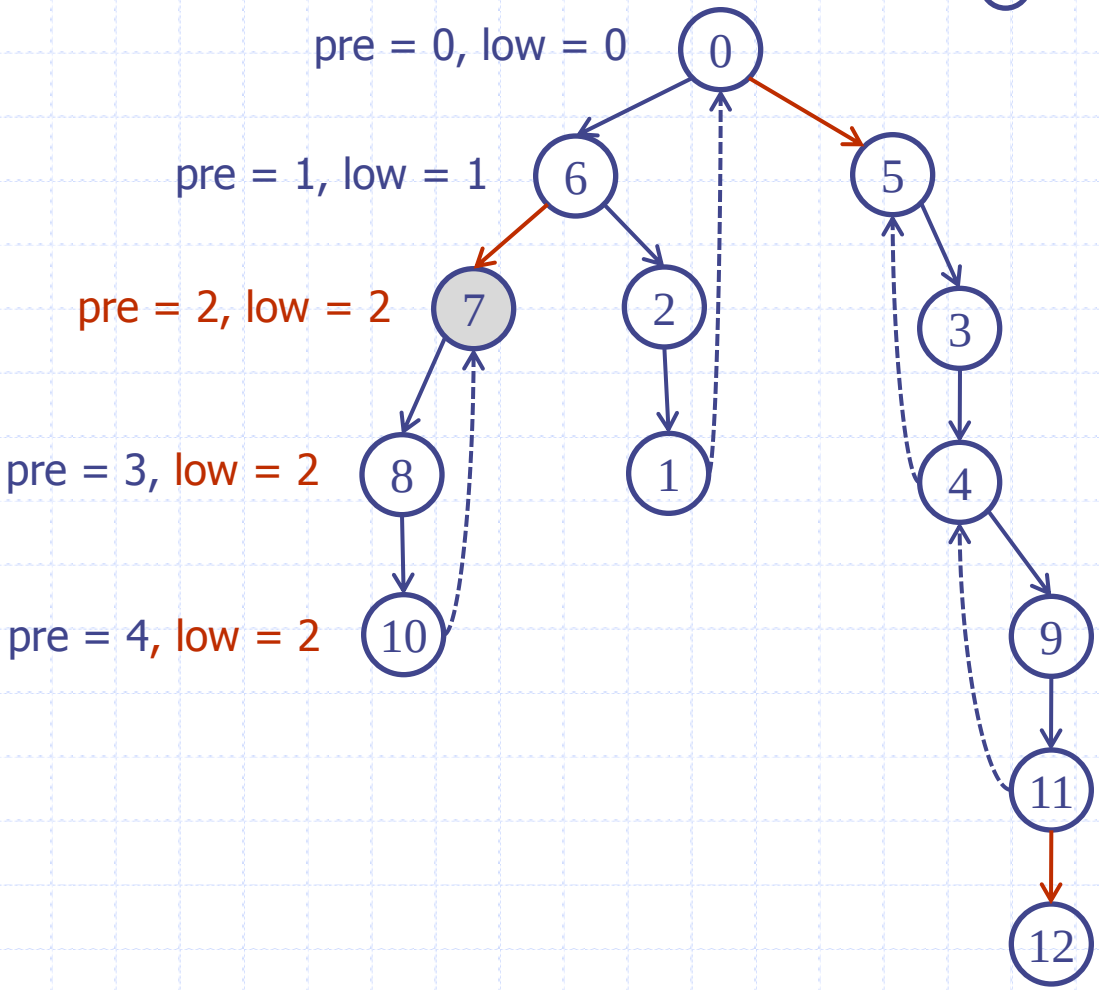
Pontes


0
6
7
8
1
2
10
3
9
5
4
11
12

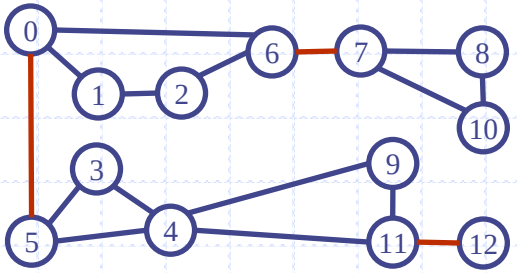

pre = 0, low = 0
0
pre = 1, low = 1
6
5
pre = 2, low = 2
7
2
3
pre = 3, low = 2
8
1
4
pre = 4, low = 2
10
9
11
12

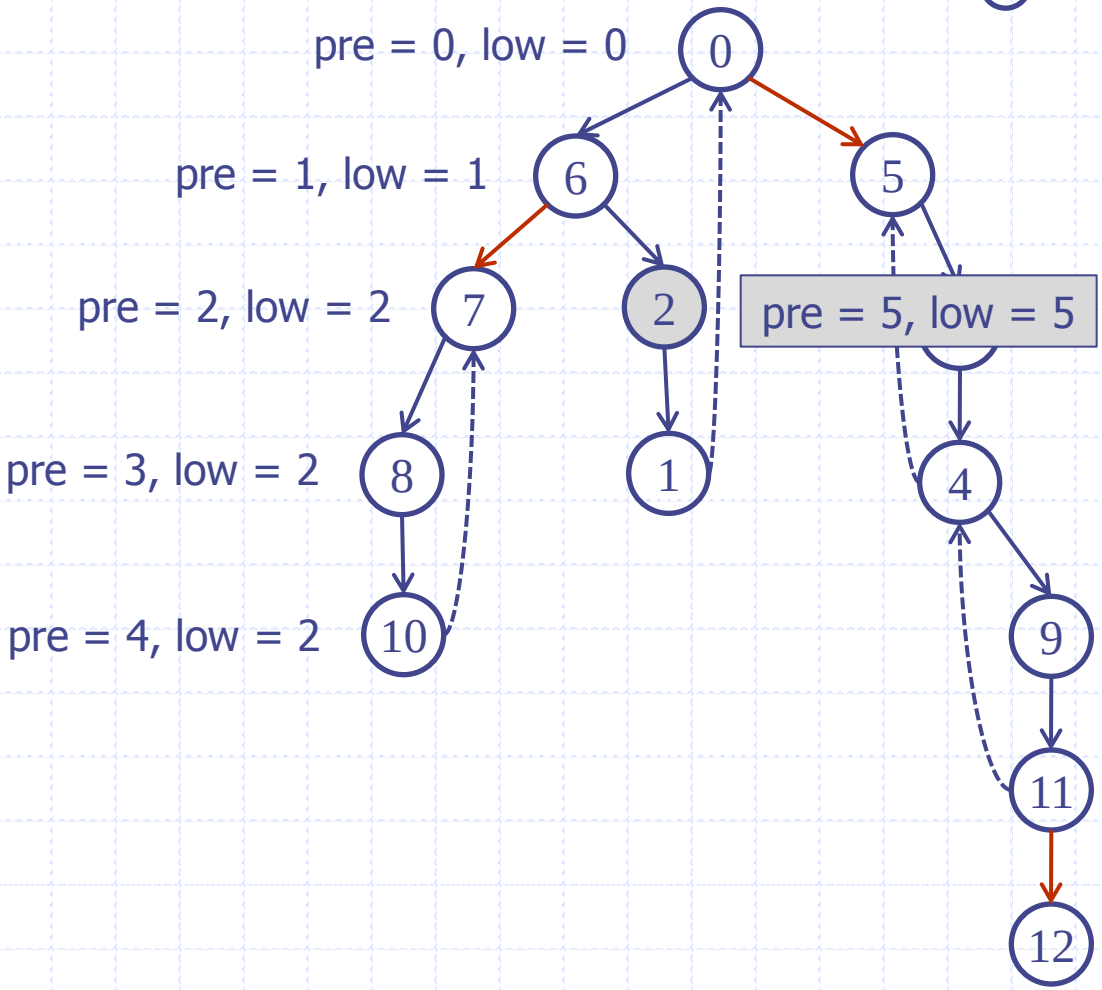
Pontes


0
6
7
8
1
2
10
3
9
5
4
11
12

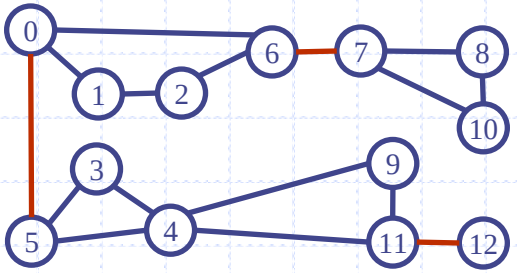

pre = 0, low = 0
0
pre = 1, low = 1
6
5
pre = 2, low = 2
7
2
3
pre = 3, low = 2
8
1
4
pre = 4, low = 2
10
9
11
12

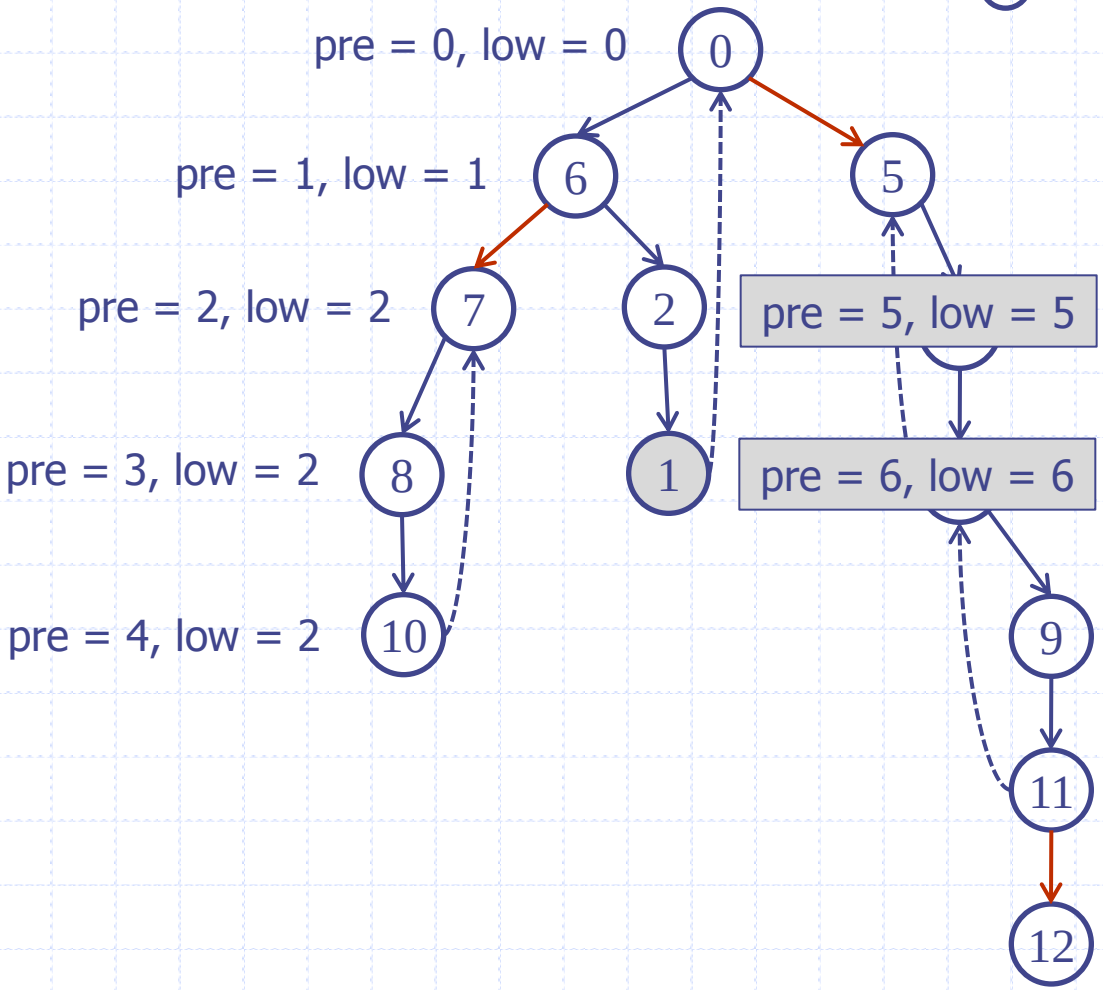
Pontes


0
6
7
8
1
2
10
3
9
5
4
11
12

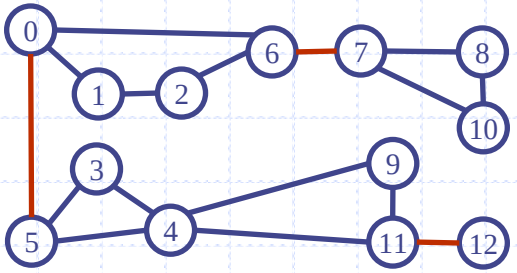

pre = 0, low = 0
0
pre = 1, low = 1
6
5
pre = 2, low = 2
7
2
pre = 5, low = 5
pre = 3, low = 2
8
1
4
pre = 4, low = 2
10
9
11
12

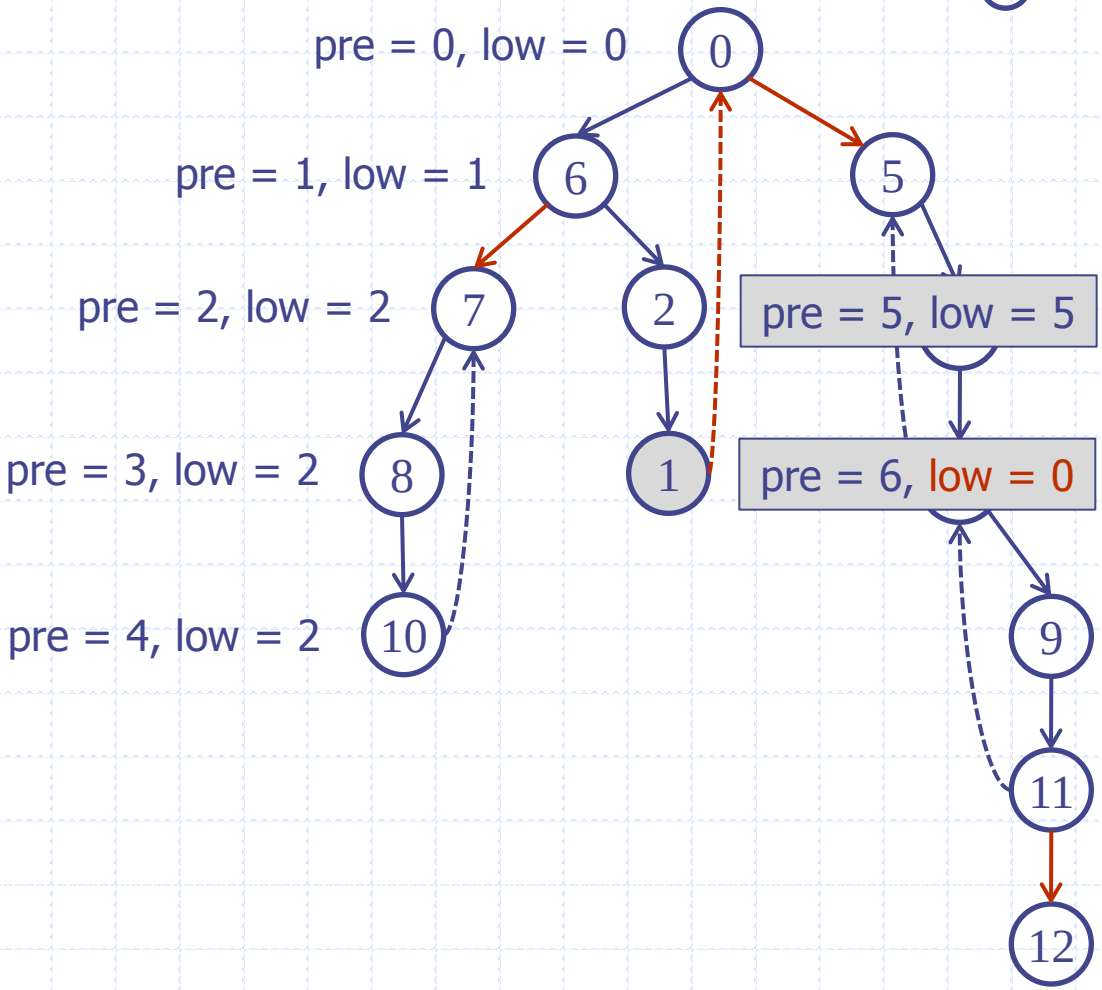
Pontes


0
6
7
8
1
2
10
3
9
5
4
11
12

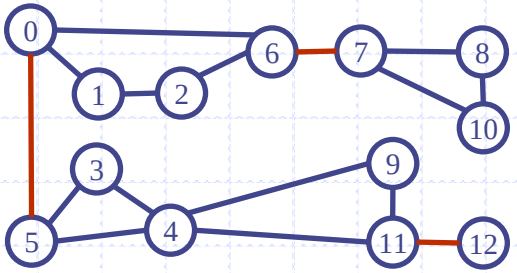

pre = 0, low = 0
0
pre = 1, low = 1
6
5
pre = 2, low = 2
7
2
pre = 5, low = 5
pre = 3, low = 2
8
1
pre = 6, low = 6
pre = 4, low = 2
10
9
11
12

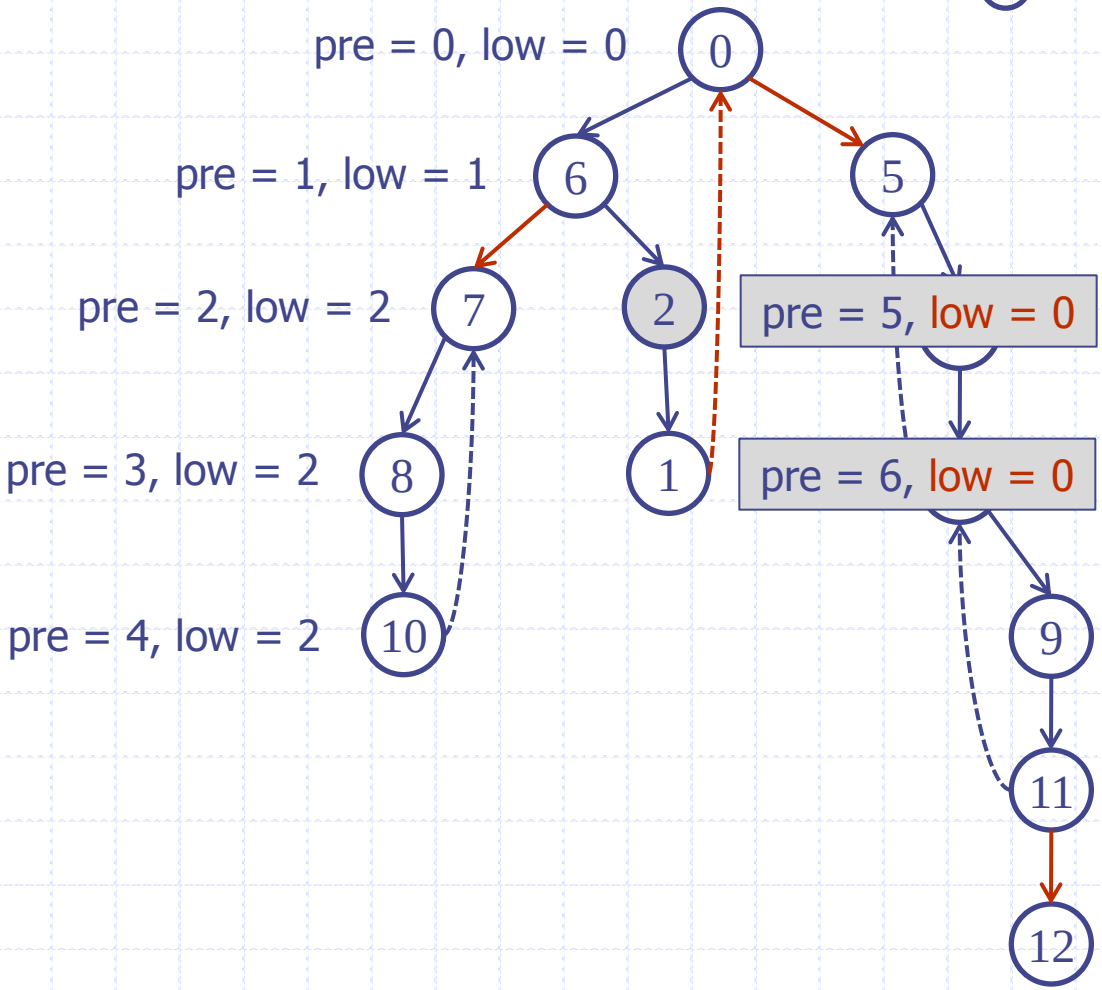
Pontes


0
6
7
8
1
2
10
3
9
5
4
11
12

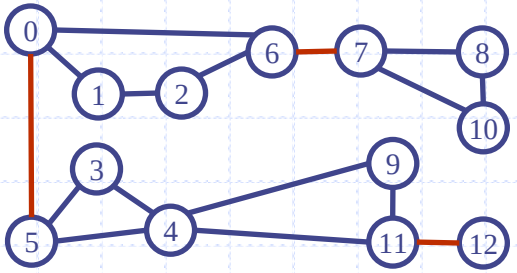

pre = 0, low = 0
0
pre = 1, low = 1
6
5
pre = 2, low = 2
7
2
pre = 5, low = 5
pre = 3, low = 2
8
1
pre = 6, low = 0
pre = 4, low = 2
10
9
11
12

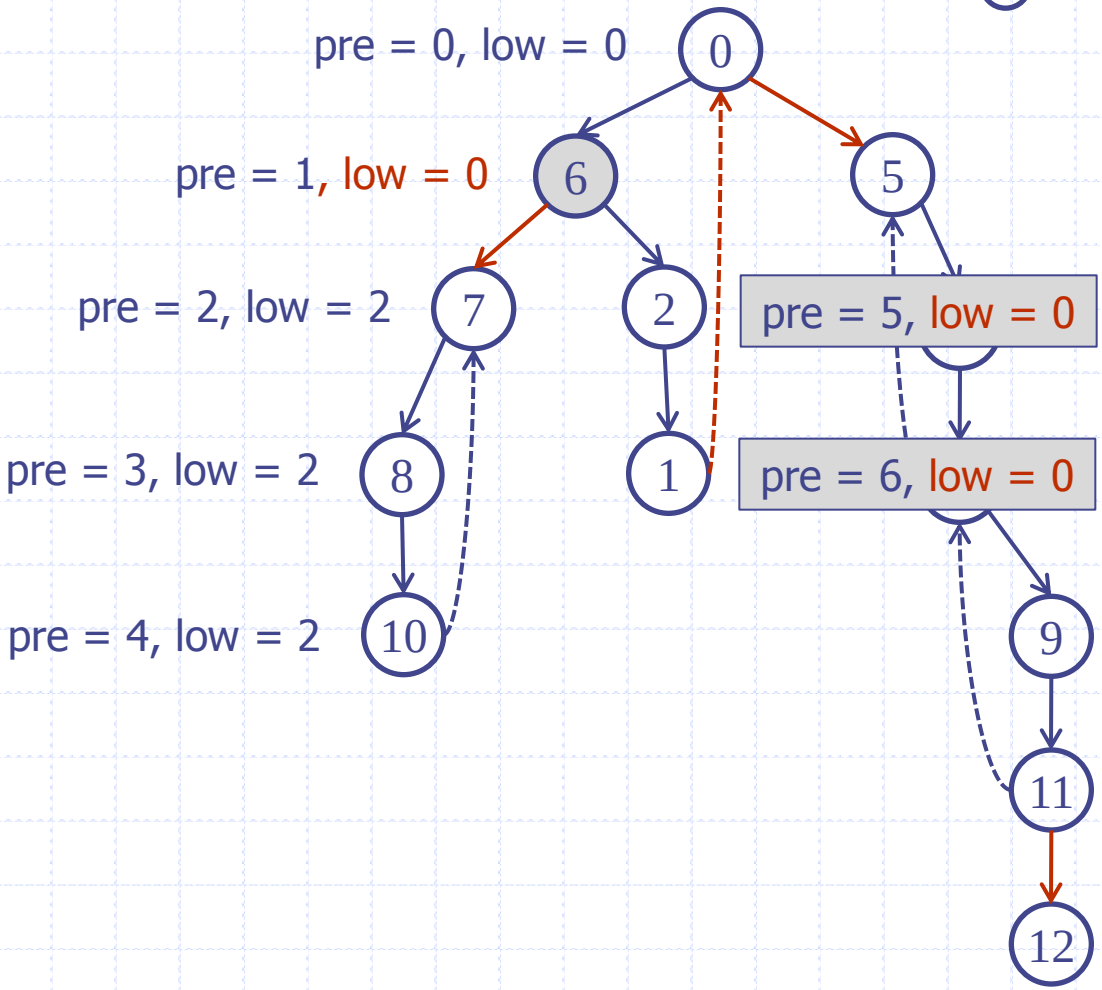
Pontes


0
6
7
8
1
2
10
3
9
5
4
11
12


pre = 0, low = 0
0
pre = 1, low = 1
6
5
pre = 2, low = 2
7
2
pre = 5, low = 0
pre = 3, low = 2
8
1
pre = 6, low = 0
pre = 4, low = 2
10
9
11
12

Pontes


0
6
7
8
1
2
10
3
9
5
4
11
12


pre = 0, low = 0
0
pre = 1, low = 0
6
5
pre = 2, low = 2
7
2
pre = 5, low = 0
pre = 3, low = 2
8
1
pre = 6, low = 0
pre = 4, low = 2
10
9
11
12

Pontes: algoritmo

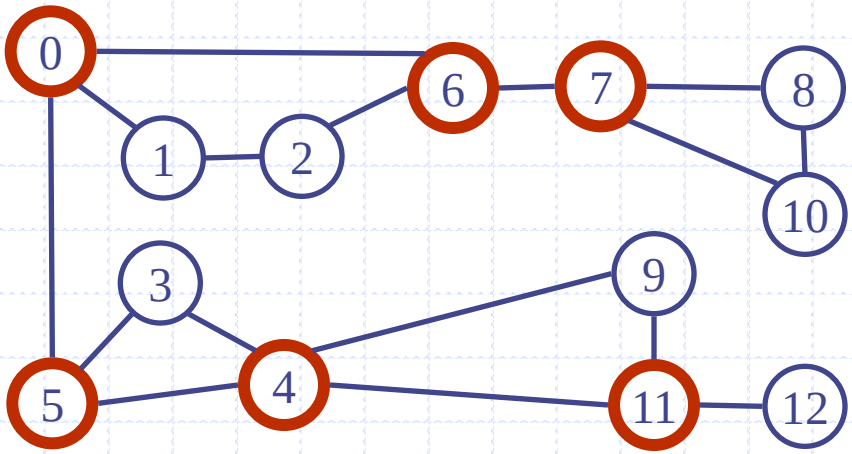
```
void bridge(int v, int ant) {
    int adj;

    pre[v] = cnt++;                                  // cnt iniciado com 0
    low[v] = pre[v];
    for (adj=0; adj<N; i++)
        if (m[v][adj] > 0)
            if (pre[adj] == -1) {                    // pre iniciado com -1
                bridge(adj, v);
                if (low[v] > low[adj])
                    low[v] = low[adj];
                if (low[adj] == pre[adj]) {
                    bcnt++;                          // bcnt iniciado com 0
                    printf("%d-%d\n", v, adj);
                }
            } else if (adj != ant)
                if (low[v] > pre[adj])
                    low[v] = pre[adj];
}
```

Pontos de articulação

- Um vértice ou **ponto de articulação** (ou ainda vértice de separação) é um vértice que, se removido, separaria o grafo em pelo menos dois subgrafos disjuntos.
- Existem pontos de articulação associados a pontes, mas existem outros não associados.

Pontos de articulação



Pontos de articulação

- Um grafo é dito ser biconectado se cada par de vértices é conectado por dois caminhos disjuntos.
- Um grafo é biconectado se e somente se ele não possui pontos de articulação.

Pontos de articulação

- Para encontrar pontos de articulação pode-se usar um algoritmo similar ao de pontes:
  - Um ponto de articulação é desprovido de uma aresta de retorno que liga um descendente a um ancestral.
  - O problema é a raiz da árvore de busca em profundidade:
    - A raiz de uma árvore de busca em profundidade é um ponto de articulação se e somente se ela possuir dois ou mais filhos.

Pontos de articulação: algoritmo

```
void articulation_point(int v, int ant, int root) {
    int adj;

    pre[v] = cnt++;                          // cnt inicializado com 0
    low[v] = pre[v];
    for (adj=0; adj<N; adj++)
        if (m[v][adj] > 0)
            if (pre[adj] == -1) {            // pre inicializado com -1
                if (root)
                    child_root++;            // child_root inicializado com 0
                articulation_point(adj, v, 0);
                if (low[v] > low[adj])
                    low[v] = low[adj];
                if (low[adj] >= pre[v])
                    art_vertex[v] = 1;       // art_vertex inicializado com 0
            } else if (adj != ant)
                if (low[v] > pre[adj])
                    low[v] = pre[adj];
}
```

```
articulation_point(0, -1, 1);
art_vertex[0] = child_root > 1;
```

Bicoloring (UVa 10004)

- *Popularity: A, Success rate: high, Level: 1*
- In 1976 the "Four Color Map Theorem" was proven with the assistance of a computer. This theorem states that every map can be colored using only four colors, in such a way that no region is colored using the same color as a neighbor region. Here you are asked to solve a simpler similar problem.
- You have to decide whether a given arbitrary connected graph can be bicolored. That is, if one can assign colors (from a palette of two) to the nodes in such a way that no two adjacent nodes have the same color. To simplify the problem you can assume:

Bicoloring (UVa 10004)

- no node will have an edge to itself.
- the graph is nondirected. That is, if a node *a* is said to be connected to a node *b*, then you must assume that *b* is connected to *a*.
- the graph will be strongly connected. That is, there will be at least one path from any node to any other node.

Input

- The input consists of several test cases. Each test case starts with a line containing the number *n* (1 < *n* < 200) of different nodes. The second line contains the number of edges *l*. After this, *l* lines will follow, each containing two numbers that specify an edge between the two nodes that they represent. A node in the graph will be labeled using a number *a* ( ). An input with *n* = 0 will mark the end of the input and is not to be processed.

Output

- You have to decide whether the input graph can be bicolored or not, and print it as shown below.

Bicoloring (UVa 10004)

- **Sample Input**

```
3
3
0 1
1 2
2 0
9
8
0 1
0 2
0 3
0 4
0 5
0 6
0 7
0 8
0
```

Sample Output

NOT BICOLORABLE.
BICOLORABLE.

Referências

- Batista, G. & Campello, R.
  - Slides disciplina *Algoritmos Avançados*, ICMC-USP, 2007.
- Goodrich, M. T. & Tamassia, R. & Mount, D.
  - http://ww3.datastructures.net
- Sedgewick, R.
  - *Algorithms in C – Part 5 – Graph Algorithms – Third Edition*, Addison-Wesley, 2002.
- Skiena, S. S. & Revilla, M. A.
  - *Programming Challenges – The Programming Contest Training Manual*. Springer, 2003.