CES-11

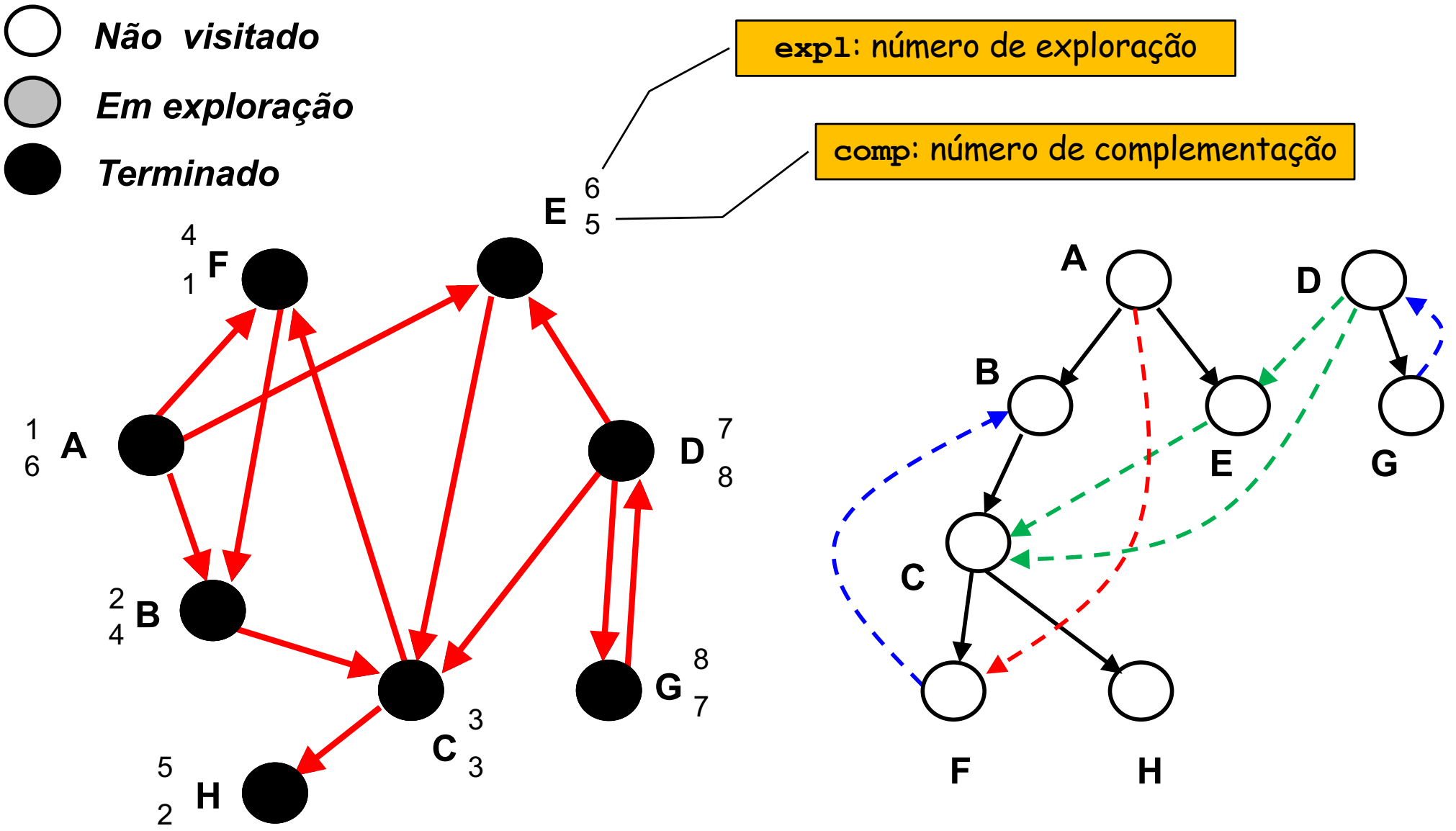
Algoritmos e Estruturas de Dados

Carlos Alberto Alonso Sanches
Juliana de Melo Bezerra

Ideia de Tarjan (1972)

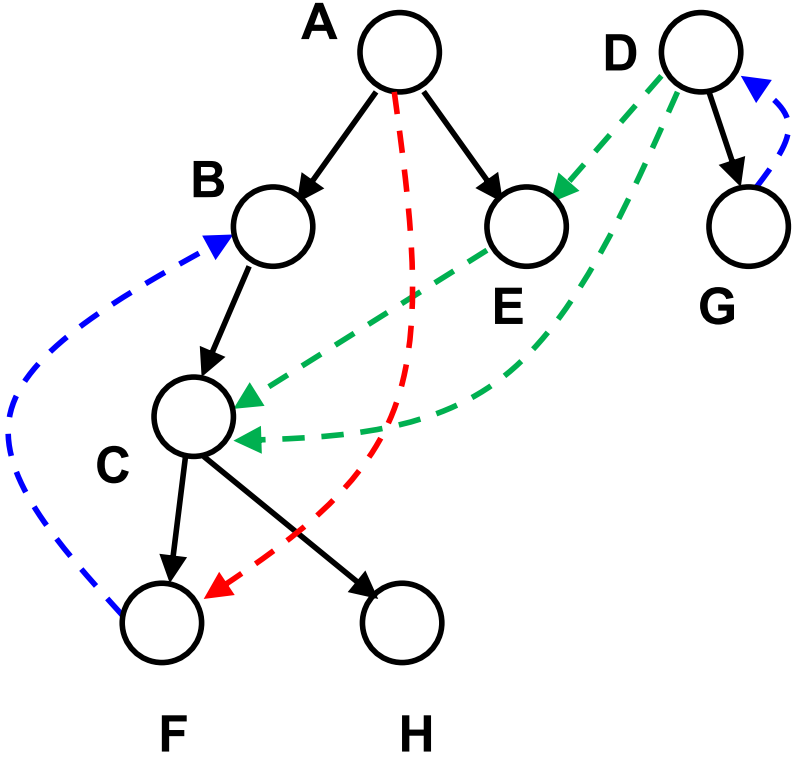
- Durante a *exploração em profundidade* de um digrafo, podemos numerar seus vértices de acordo com o início e o término dessa exploração.
- As diferentes situações permitem estabelecer uma classificação dos arcos.

Exemplo


Não visitado
Em exploração
Terminado
expl: número de exploração
comp: número de complementação
E 6 5
4 1 F
1 6 A
D 7 8
2 4 B
G 8 7
C 3 3
5 2 H
A
D
B
E
G
C
F
H

Classificação dos arcos

- Classificação do arco <v,u>:
  - Árvore (T): u ainda não havia sido explorado, e será filho de v em T (`expl[u]=0`)
  - Retorno (B): u é antecessor de v em T, pois começou antes de v e ainda está em exploração (`expl[u]<expl[v] e comp[u]=0`)
  - Cruzamento (C): u está em outra árvore ou sub-árvore, pois começou antes de v e já foi explorado (`expl[u]<expl[v] e comp[u]>0`)
  - Avanço (F): u é descendente de v em T, pois começou depois de v e já foi explorado (`expl[u]>expl[v] e comp[u]>0`)



Algoritmo de Tarjan

```
Tarjan() {
  int ce = 0;
  int cc = 0;
  for v ∈ V {
      expl[v] = 0;
      comp[v] = 0;
  }
for v ∈ V
     if (expl[v] == 0)
         DFS(v);
}
```

Sua implementação deve receber as variáveis **ce** e **cc**

```
DFS(v) {
     expl[v] = ++ce;
     for <v,u> ∈ E
        if (expl[u] == 0) {
              tipo[<v,u>] = T;
              DFS(u);
        }
        else if (expl[u] > expl[v])
              tipo[<v,u>] = F;
          else if (comp[u] > 0)
              tipo[<v,u>] = C;
            else tipo[<v,u>] = B;
    comp[v] = ++cc;
}
```

Complexidade de tempo: $\Theta(n+m)$

Exercício

- Considere um grafo *não orientado*, sem laços e sem arestas repetidas. Se aplicarmos o algoritmo de Tarjan nesse grafo, somente haverá arestas de árvore e de retorno. Por quê?

CES-11

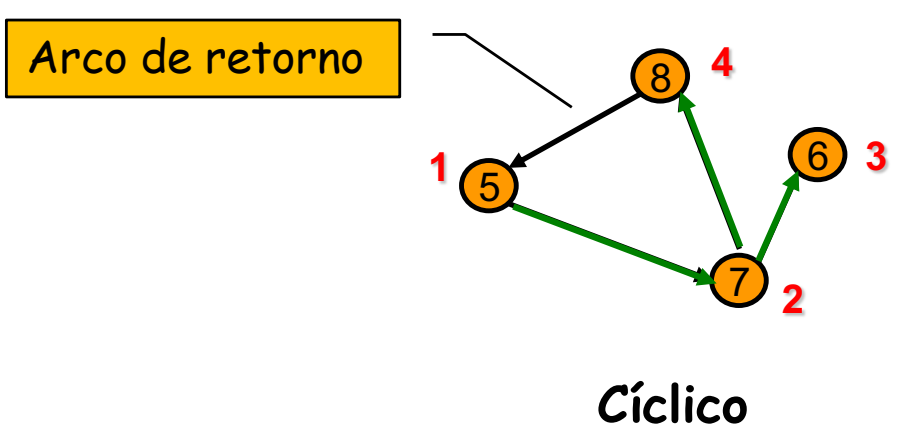
- Grafos
  - Conceitos gerais e representações
- Algoritmos em grafos
  - Exploração sistemática em largura
  - Caminhos mais curtos
  - Exploração sistemática em profundidade
  - Teste de aciclicidade
  - Ordenação topológica
  - Componentes fortemente conexos
  - Vértices e arestas de corte
  - Árvore geradora de custo mínimo

Teste de aciclicidade

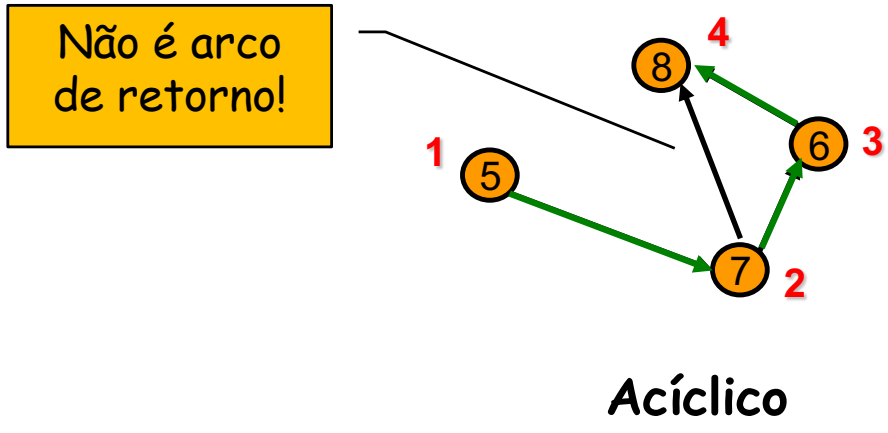
- Certas aplicações, como a ordenação topológica, exigem que o digrafo seja acíclico.
- Definição: um *digrafo acíclico* é chamado de **DAG**.
- Portanto, uma tarefa importante é verificar se um determinado digrafo é um DAG.
- A exploração em profundidade pode nos dar uma boa solução para esse problema.
- Concretamente, basta uma variação do algoritmo de Tarjan: se um arco de retorno for encontrado durante a exploração, então o digrafo será cíclico.

Ideia do algoritmo

- Ideia:
  - Manter uma pilha com os ancestrais do vértice que está sendo visitado, incluindo ele próprio.
  - Quando terminar a exploração dos seus vértices adjacentes, ele deverá ser retirado dessa pilha.
  - Se um arco de retorno for encontrado, o ciclo estará nessa pilha (desde o topo até o vértice atingido pelo arco).



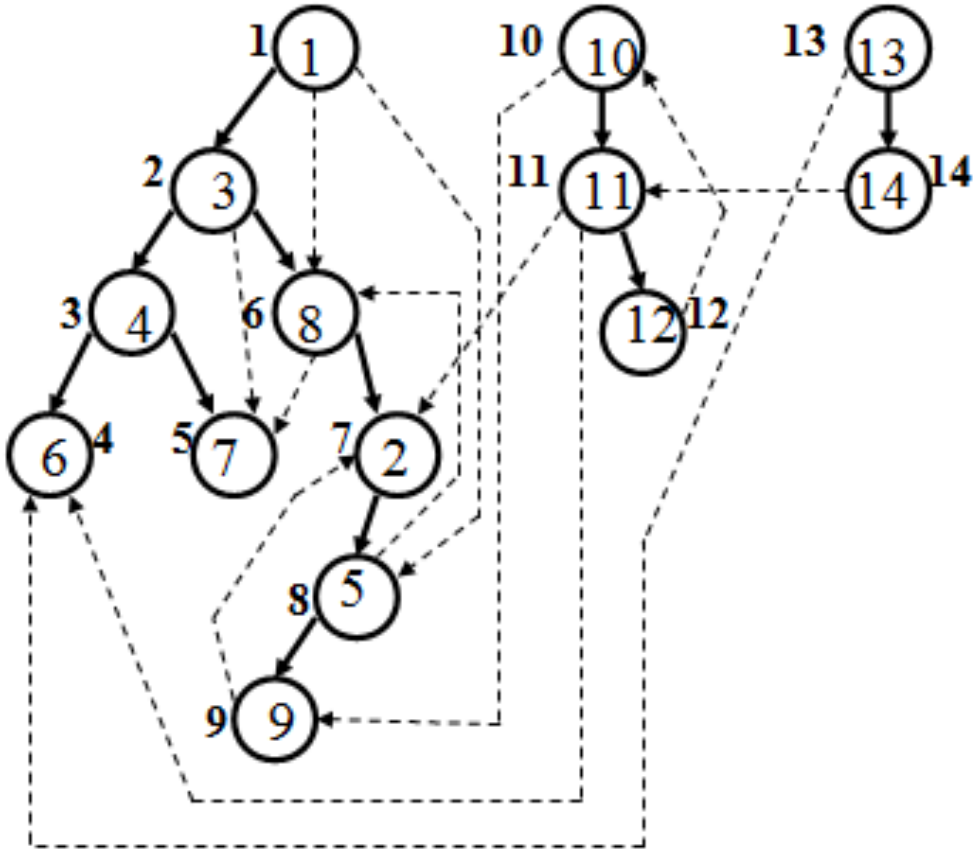
Cíclico



Acíclico

Um exemplo

- Considere a exploração do digrafo ao lado, começada no vértice 1.
- Ao explorar o vértice 5, estarão na pilha seus ancestrais 1, 3, 8 e 2, além dele mesmo.
- Na tentativa de explorar o vértice 8, encontra-se um arco de retorno.
- Logo, esse digrafo é cíclico.



Algoritmo

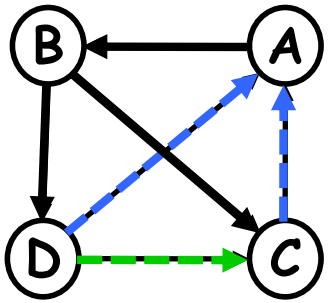
```
Aciclicidade() {
    pilha P;
    inicPilha(P);
    bool aciclico = true;
    int ce = 0;
    int cc = 0;
    for v ∈ V {
         expl[v] = 0;
         comp[v] = 0;
    }
    for v ∈ V
      if (expl[v] == 0)
          DFS(v);
    if (!aciclico)
        escrever ("Grafo é cíclico");
    else
        escrever ("Grafo é acíclico");
}
```

```
DFS(v) {
     expl[v] = ++ce;
     push(P,v);
     for <v,u> ∈ E
       if (expl[u] == 0)
           DFS(u);
       else
           if (expl[u]<expl[v] && comp[u]==0)
               aciclico = false;
               // ciclo está em P
               // desde o topo até u
     pop(P);
     comp[v] = ++cc;
}
```

Sua implementação deve receber as variáveis **ce**, **cc**, **P** e **aciclico**

Uma observação

- Como todo ciclo possui ao menos uma aresta de retorno, o algoritmo anterior verifica a existência ou não de ciclos em um digrafo.
- No entanto, ele não é capaz de identificar todos os ciclos...
- No exemplo abaixo, aplicando o algoritmo a partir de A em ordem alfabética, podem ser encontrados os ciclos ABC e ABD.



- No entanto, o ciclo ABDC não será identificado. Isso se deve ao fato de que, neste ciclo, também há uma aresta de cruzamento.

CES-11

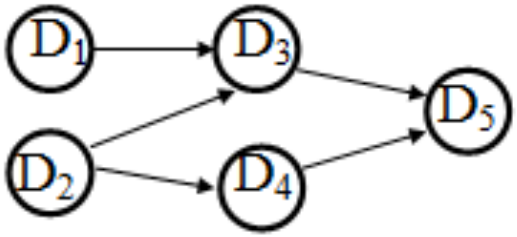
- Grafos
  - Conceitos gerais e representações
- Algoritmos em grafos
  - Exploração sistemática em largura
  - Caminhos mais curtos
  - Exploração sistemática em profundidade
  - Teste de aciclicidade
  - Ordenação topológica
  - Componentes fortemente conexos
  - Vértices e arestas de corte
  - Árvore geradora de custo mínimo

Ordenação topológica

- Como vimos, *ordenação topológica* é o processo de se ordenar os vértices de um DAG: se houver um arco do vértice i para o vértice j, então i aparecerá antes de j na ordenação.
- Aplicação muito comum: encontrar um escalonamento de tarefas.
- De modo geral, grandes projetos são formados por tarefas, entre as quais existe uma relação de dependência temporal.
- Através da ordenação topológica, obtém-se uma sequência válida de execução dessas tarefas. Isso é crítico quando poucas tarefas podem ser executadas simultaneamente.

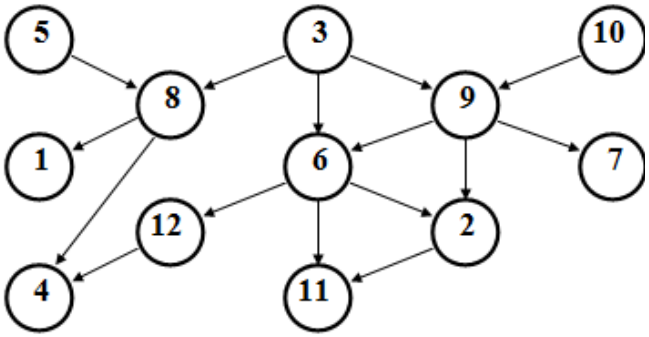
Uma aplicação

- A ordenação topológica poderia ser aplicada, por exemplo, em um curso com módulos semestrais, com várias disciplinas por módulos.
- O DAG representaria o sistema de pré-requisitos entre as disciplinas do curso, e cada ordenação topológica corresponderia a uma possível sequência em que as disciplinas poderiam ser cursadas.
- Exemplo: disciplinas D_1, D_2, D_3, D_4 e D_5.
  - (D1, D2, D3, D4, D5) e (D2, D4, D1, D3, D5) são ordenações topológicas do DAG abaixo.



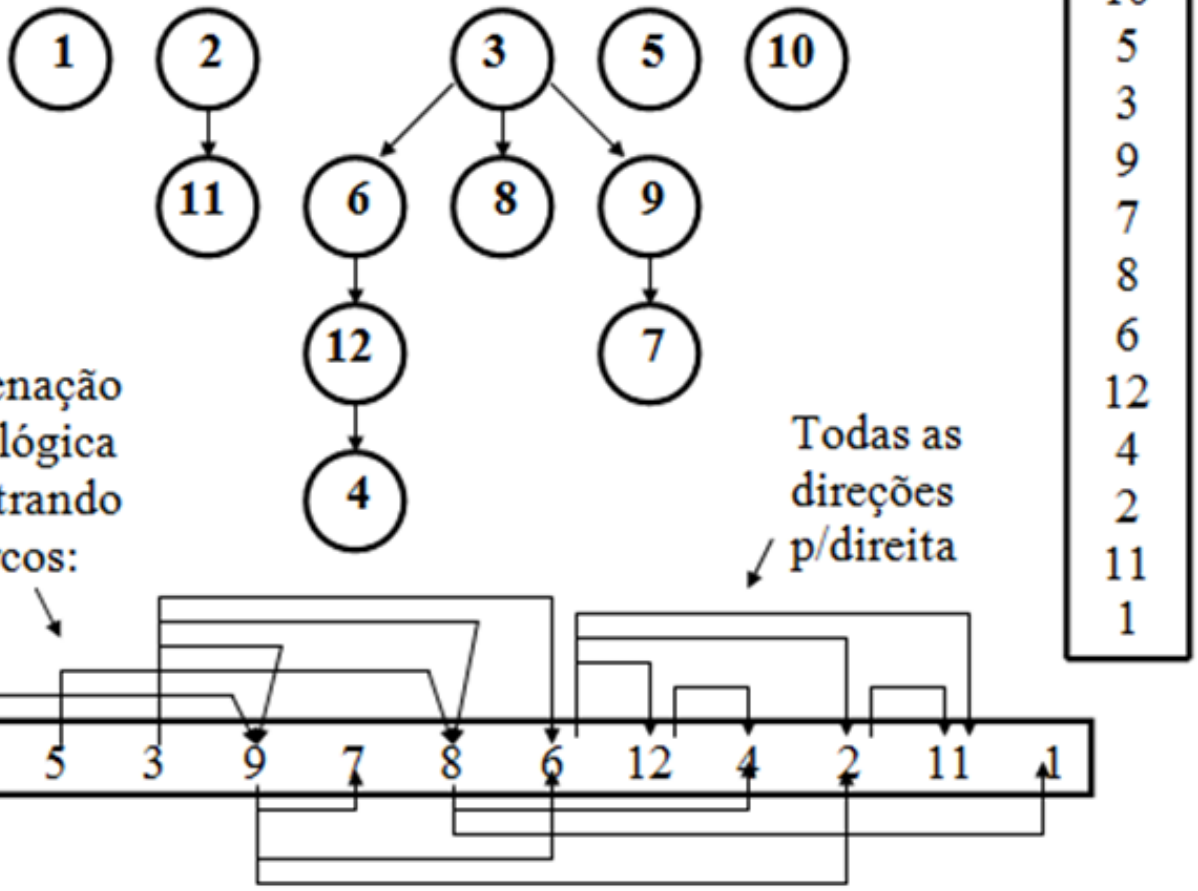
Exemplo de ordenação topológica

- Considere o DAG abaixo:



  - Vamos fazer sua exploração em profundidade dando prioridade sempre aos vértices de menor valor.

Sequência de término de exploração



Uma solução

- A ordenação topológica pode ser resolvida com uma simples variante da exploração em profundidade: basta utilizar o vetor de complementação em ordem inversa.

```
OrdemTopol(s) {
    int ce = 0;
    int cc = 0;
    for v ∈ V {
        expl[v] = 0;
        comp[v] = 0;
    }
    DFS(s);
    for v ∈ V
        if (expl[v] == 0)
            DFS(v);
    for v ∈ V
        escrever (v, |V|-comp[v]+1);
}
```

```
DFS(v) {
    expl[v] = ++ce;
    for <v,u> ∈ E
        if (expl[u] == 0)
            DFS(u);
    comp[v] = ++cc;
}
```

Complexidade de tempo: $O(n+m)$

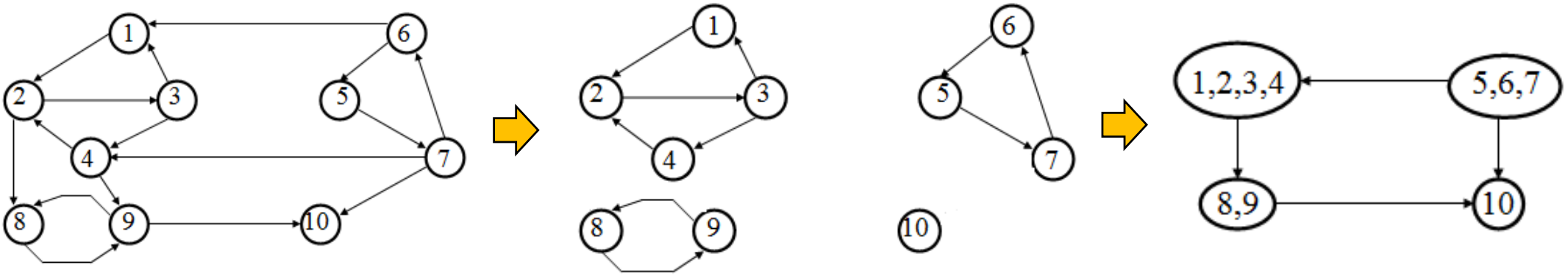
Usando uma pilha, seria possível imprimir os vértices já na ordem topológica. Como?

CES-11

- Grafos
  - Conceitos gerais e representações
- Algoritmos em grafos
  - Exploração sistemática em largura
  - Caminhos mais curtos
  - Exploração sistemática em profundidade
  - Teste de aciclicidade
  - Ordenação topológica
  - Componentes fortemente conexos
  - Vértices e arestas de corte
  - Árvore geradora de custo mínimo

Componentes fortemente conexos (CFC)

- Em um digrafo, os componentes fortemente conexos (CFC) são subconjuntos maximais de vértices conectados entre si, isto é, dados dois vértices v_i e v_j em um mesmo CFC, há um caminho de v_i a v_j e de v_j a v_i.



Digrafo | **CFCs do digrafo** | **Digrafo reduzido (é um DAG)**

Importante:
{1,2,3} não é um CFC, pois não é maximal

Componentes fortemente conexos (CFC)

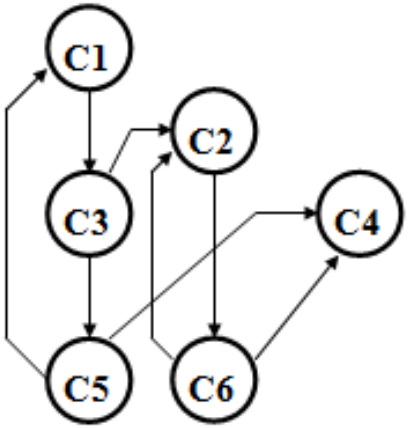
- Para que servem?
- Por exemplo, para subdividir problemas em digrafos (um subproblema semelhante para cada CFC)
- Exemplos:
  - Estudo de privacidade em sistemas de comunicação
  - Análise de circuitos eletrônicos: classes de equivalência
  - Análise do fluxo de controle para a validação de programas
  - Facilitar a paralelização de laços sequenciais

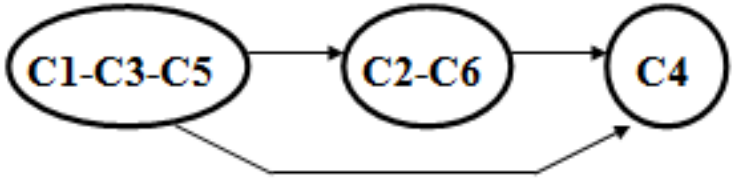
Componentes fortemente conexos (CFC)

- Exemplo: decomposição de laços sequenciais

```
    for (i = 1; i <= n; i++) {
C1:     A[i] = B[i] + C[i];
C2:     D[i] = E[i] + F[i];
C3:     E[i+1] = A[i] + G[i];
C4:     H[i] = F[i] + B[i];
C5:     B[i+1] = E[i+1] + M[i];
C6:     F[i+1] = D[i] + N[i];
    }
```

É possível representar em um digrafo as dependências temporais entre esses comandos





Digrafo dos CFC

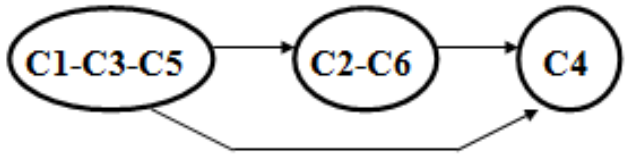
Componentes fortemente conexos (CFC)

- Laços decompostos de acordo com os CFC:

```
     for (i = 1; i <= n; i++) {
C1:      A[i] = B[i] + C[i];
C3:      E[i+1] = A[i] + G[i];
C5:      B[i+1] = E[i+1] + M[i];
     }

     for (i = 1; i <= n; i++) {
C2:      D[i] = E[i] + F[i];
C6:      F[i+1] = D[i] + N[i];
     }

     for (i = 1; i <= n; i++) {
C4:      H[i] = F[i] + B[i];
     }
```

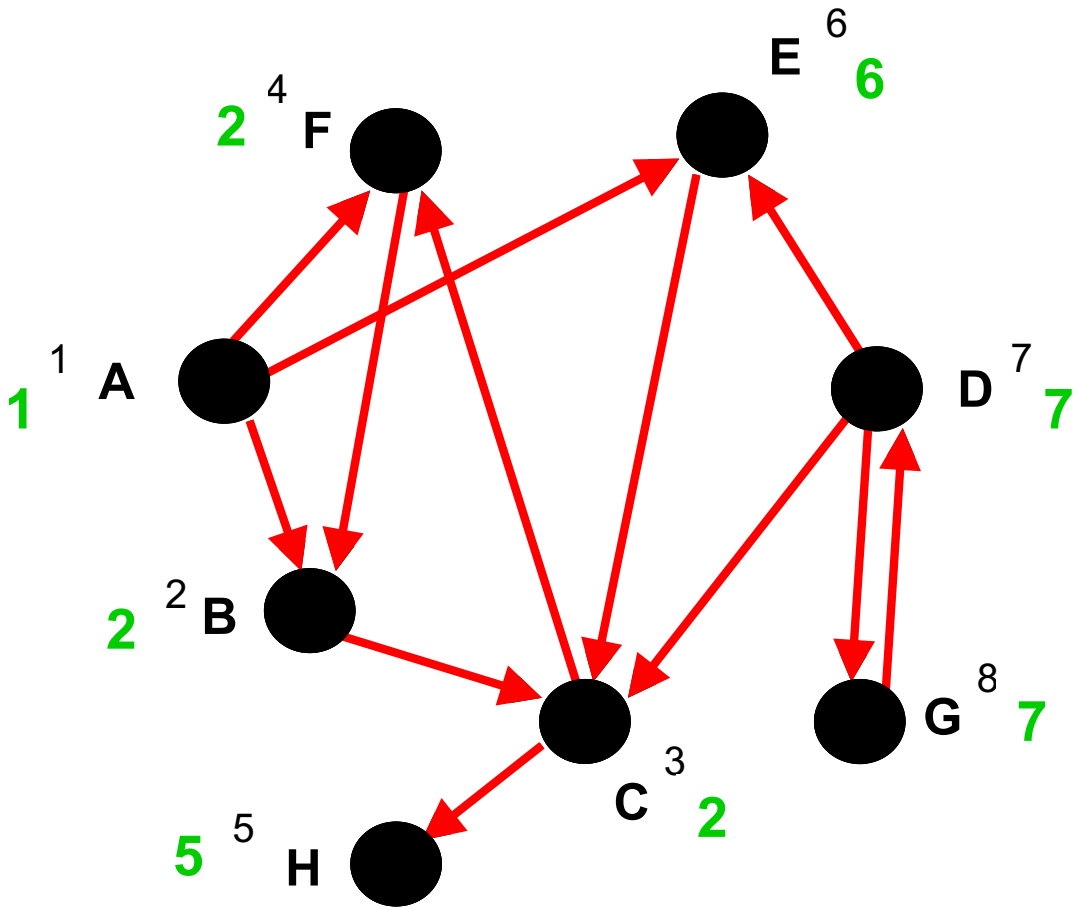


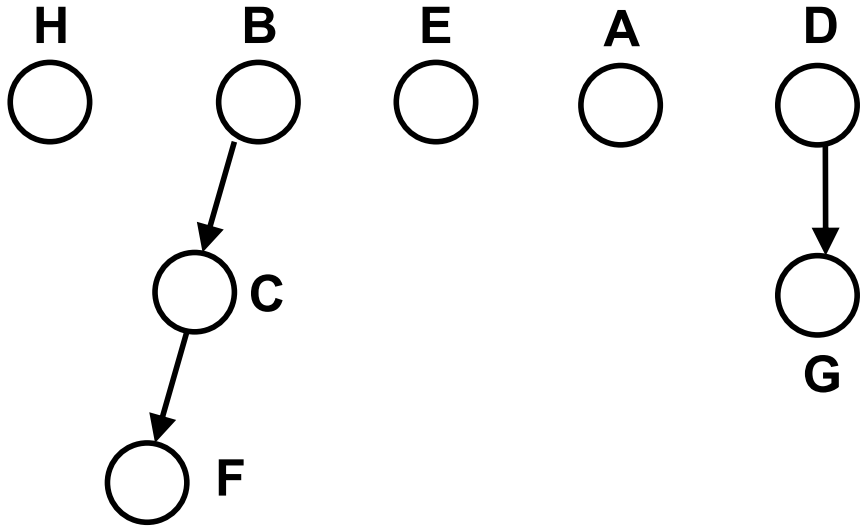
Geralmente, é mais fácil aplicar técnicas de paralelização em laços menores

Componentes fortemente conexos (CFC)

- É possível encontrar os componentes fortemente conexos de um digrafo através de uma variante do algoritmo de Tarjan.
- Ideia:
  - Considere a árvore T de exploração em profundidade e a numeração expl[v] para cada $v \in V$.
  - Os vértices que estão em exploração são empilhados (permanecerão na pilha até que seja encontrado o seu componente conexo).
  - Cada vértice v guardará CFC[v], que é o menor número de exploração entre os vértices na pilha que atingir durante sua exploração. Desse modo, ficará automaticamente associado a um componente conexo.
  - Quando a exploração do vértice v terminar, se expl[v] = CFC[v] então todos os vértices na pilha (desde o topo até v) pertencem a um mesmo componente, e podem ser desempilhados.

Exemplo





- Importante: nem todos os vértices de um mesmo componente terminam com o mesmo valor.
- Exemplo: acrescente um arco <C,A> nesse mesmo digrafo, e visite <C,F> antes.

Algoritmo

```
TarjanCFC() {
  pilha P;
  inicPilha(P);
  int ce = 0;
  for v ∈ V
     expl[v] = 0;
  for v ∈ V
     if (expl[v] == 0)
         DFSCFC(v);
}
```

```
DFSCFC(v) {
    expl[v] = ++ce;
    push(P,v);
    CFC[v] = expl[v];
    for <v,u> ∈ E
       if (expl[u] == 0) {
          DFSCFC(u);
          CFC[v] = min{CFC[v],CFC[u]};
       }
       else if (u ∈ P)
          CFC[v] = min{CFC[v],expl[u]};
    if (CFC[v] == expl[v])
       do {
          x = top(P);
          pop(P);
       } while (x != v);
}
```

Arco de árvore

Vértices sem componente definido

Complexidade de tempo: $O(n+m)$

CES-11

- Grafos
  - Conceitos gerais e representações
- Algoritmos em grafos
  - Exploração sistemática em largura
  - Caminhos mais curtos
  - Exploração sistemática em profundidade
  - Teste de aciclicidade
  - Ordenação topológica
  - Componentes fortemente conexos
  - Vértices e arestas de corte
  - Árvore geradora de custo mínimo

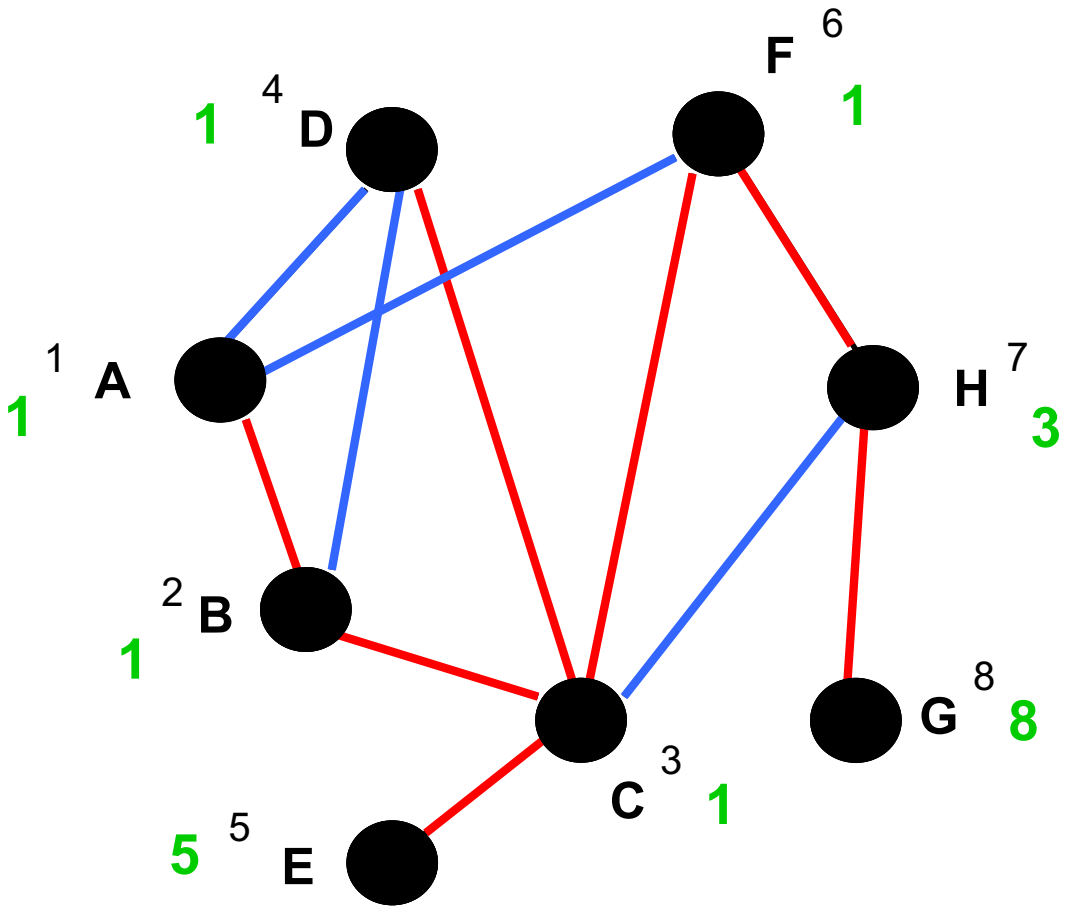
Vértices de corte

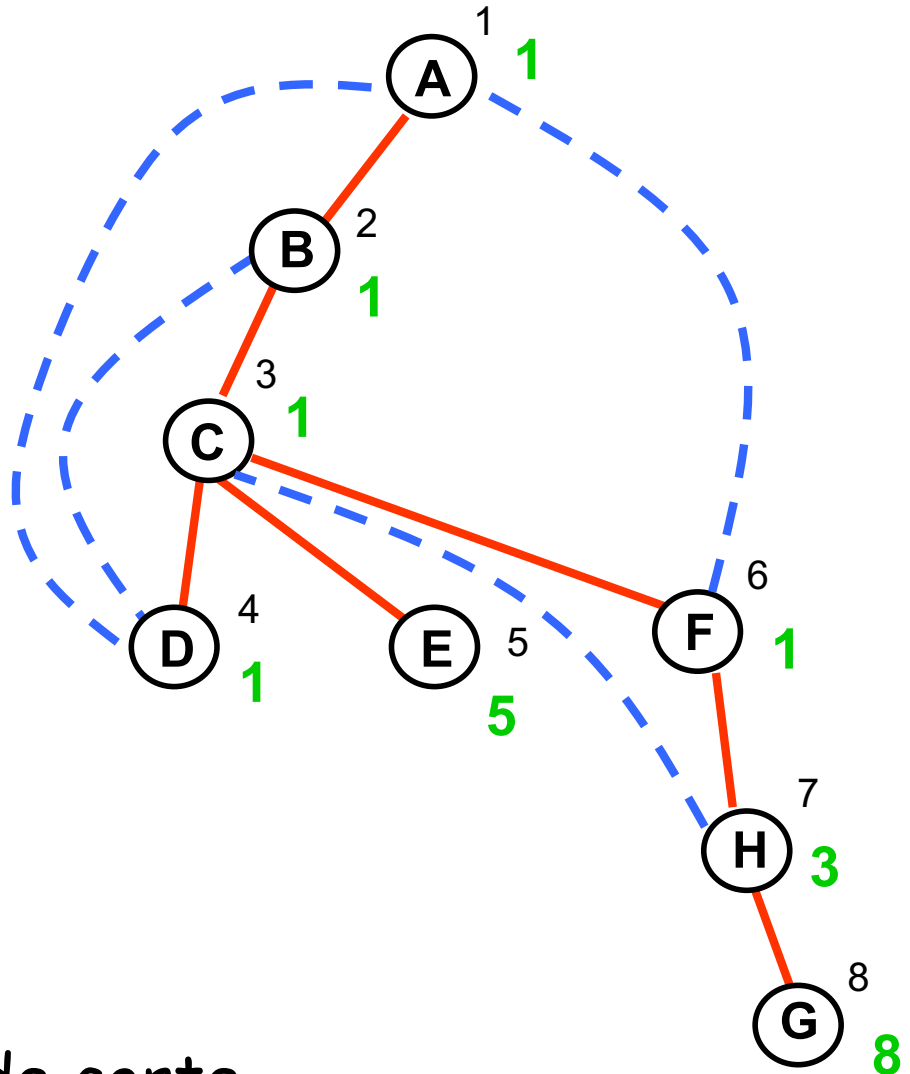
- Uma variante do algoritmo de Tarjan pode encontrar os vértices de corte (ou pontos de articulação) de um grafo G=(V,E) *conexo não-orientado, sem laços ou arestas repetidas*.

Desse modo, se fizéssemos uma exploração em profundidade a partir de cada vértice do grafo, poderíamos identificar todos os vértices de corte (no entanto, há outra solução mais eficiente)

- Ideia:
  - Considere a árvore T de exploração em profundidade e a numeração expl[v] para cada v $\in$ V.
  - **Raiz:** será vértice de corte se tiver pelo menos dois filhos em T.
  - **Demais vértices**:
    - v será vértice de corte se tiver algum filho sem retorno para nenhum dos ancestrais de v.
    - É calculado m[v] = mín{ expl[v], expl[x] }, onde x é um vértice que v (ou um de seus descendentes) atinge em T através de uma *única aresta de retorno*.
    - Portanto, v será vértice de corte se tiver algum filho u tal que m[u] ≥ expl[v].

Exemplo





C e H são vértices de corte

Algoritmo

```
TarjanVC(r) {
   // válido se for conexo e
   // não tiver laços ou
   // arestas repetidas
   int ce = 0;
   for v ∈ V {
      expl[v] = 0;
      pai[v] = null;
      nfilhos[v] = 0;
      VC[v] = false;
   }
   DFSVC(r);
   for v ∈ V-{r} {
      p = pai[v];
      VC[p] = VC[p] || (m[v]>=expl[p]);
   }
   VC[r] = (nfilhos[r] > 1);
   for v ∈ V
      if (VC[v]) v é vértice de corte
}
```

Trata as arestas de retorno, tanto na ida como na volta

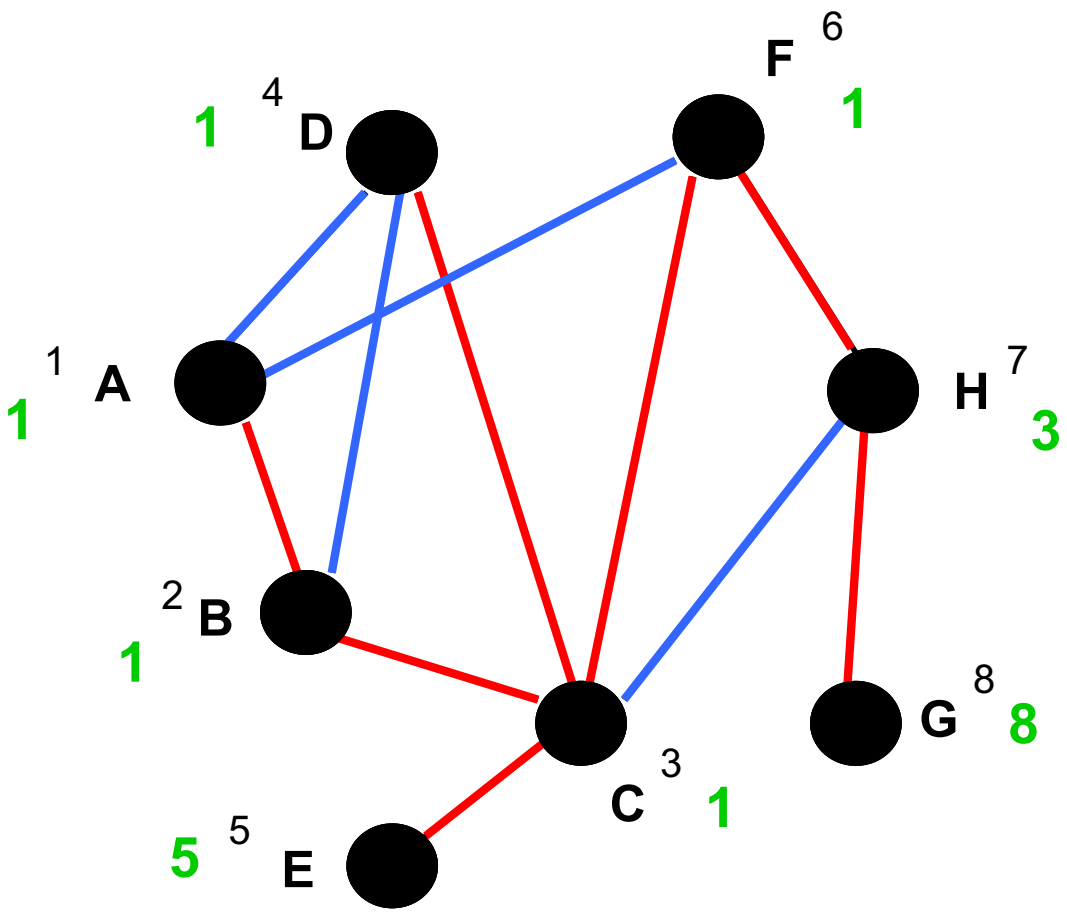
```
DFSVC(v) {
   expl[v] = ++ce;
   m[v] = expl[v];
   for <v,u> ∈ E
      if (expl[u] == 0) {
         pai[u] = v;
         nfilhos[v]++;
         DFSVC(u);
         m[v] = min{m[v],m[u]};
      }
      else // arestas de retorno
       if(u != pai[v])
         m[v] = min{m[v],expl[u]};
}
```

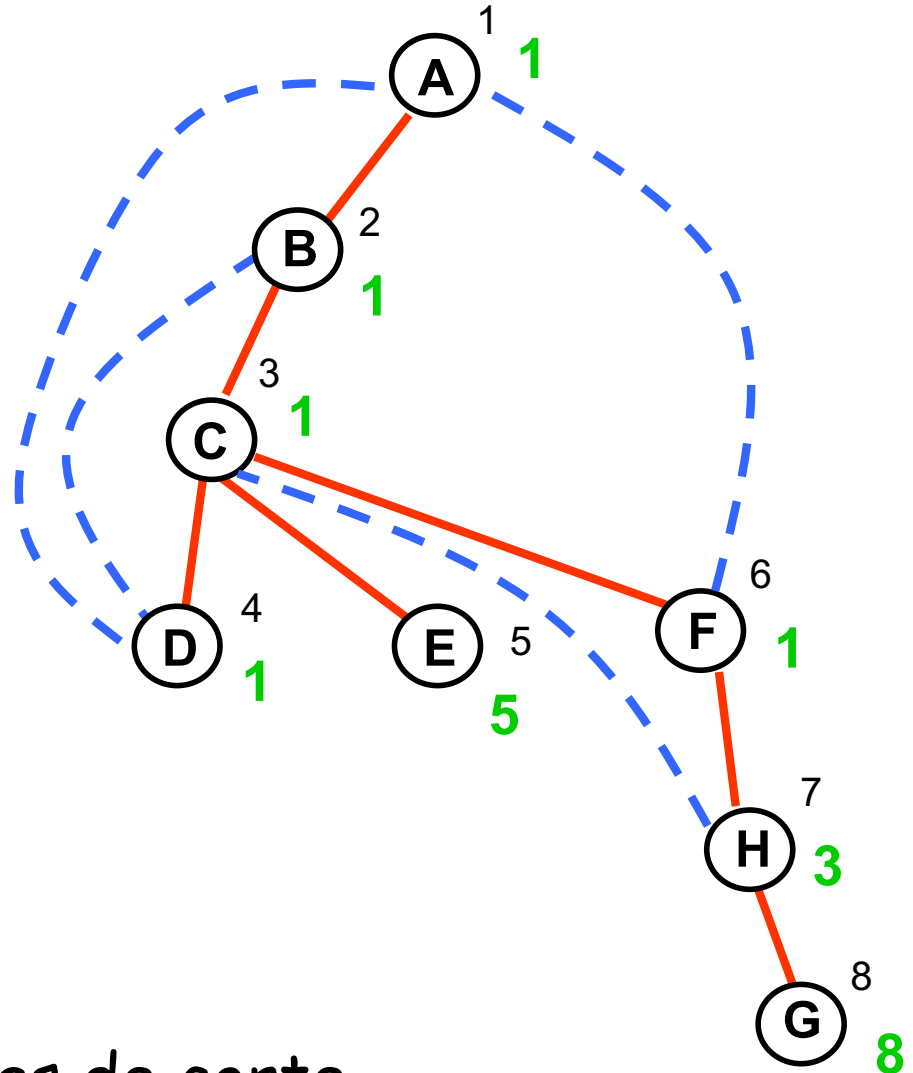
Complexidade de tempo: $O(n+m)$

Arestas de corte

- A identificação das arestas de corte (ou pontes) é realizada de maneira semelhante:
  - Encontrar uma árvore de exploração T, calculando as mesmas numerações expl e m para os vértices.
  - É fácil constatar que nenhuma aresta de retorno dessa exploração pode ser de corte.
  - Uma aresta <v,u> ϵ T será de corte se m[u] = expl[u].

No exemplo anterior





<C,E> e <H,G> são arestas de corte

Algoritmo

```
TarjanAC(r) {
  // válido se for conexo e
  // não tiver laços ou
  // arestas repetidas
  int ce = 0;
  for v ∈ V {
     expl[v] = 0;
     pai[v] = null;
  }
  DFSAC(r);
}
```

```
DFSAC(v) {
  expl[v] = ++ce;
  m[v] = expl[v];
  for <v,u> ∈ E
    if (expl[u] == 0) {
      pai[u] = v;
      DFSAC(u);
      m[v] = min{m[v],m[u]};
      if (m[u] == expl[u])
            <v,u> é aresta de corte
    }
    else // arestas de retorno
     if(u != pai[v])
       m[v] = min{m[v],expl[u]};
}
```

Complexidade de tempo: $O(n+m)$