Teoria dos Grafos

Aula 8

Componentes Conexos, Ordenação Topológica

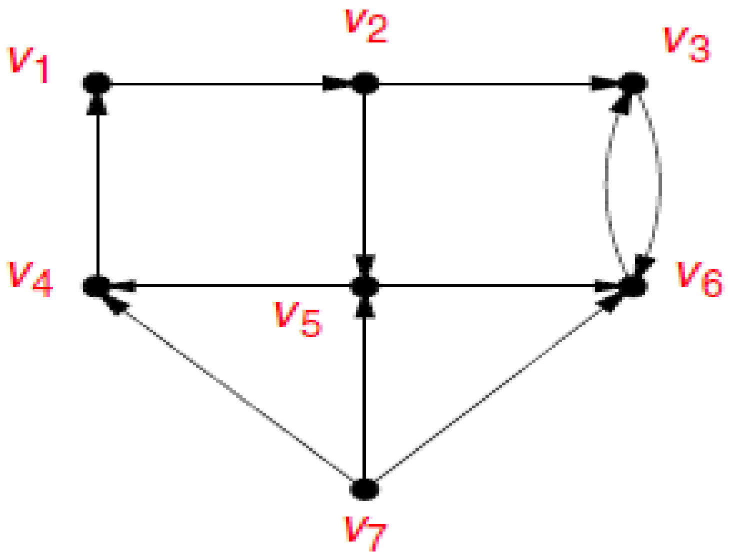
Marcos Muniz

Fecho Transitivo Direto

- Definição:
  - O fecho transitivo direto (FTD) de um vértice v é o conjunto de todos os vértices que podem ser atingidos por algum caminho iniciando em v.

Fecho Transitivo Direto

- Exemplo: O FTD do vértice v_5 do grafo abaixo é o conjunto $\{v_1, v_2, v_3, v_4, v_5, v_6\}$. Note que o próprio vértice faz parte do FTD já que ele é alcançável partindo-se dele mesmo.

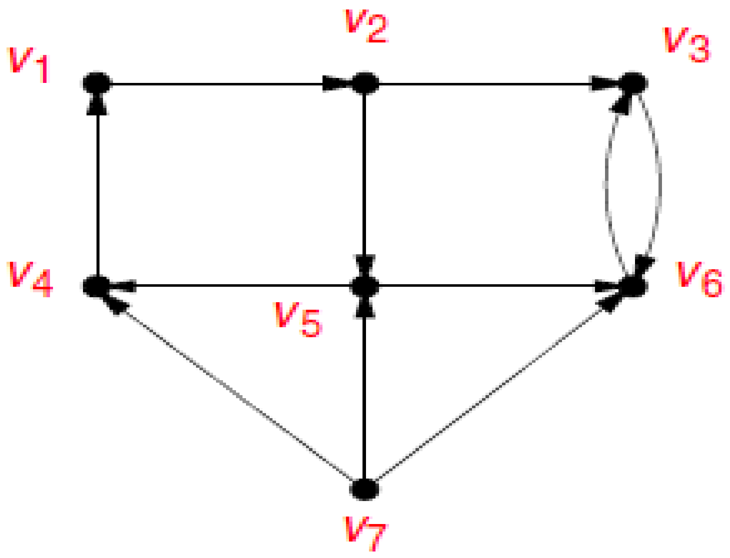


Fecho Transitivo Inverso

- Definição: O fecho transitivo inverso (FTI) de um vértice v é o conjunto de todos os vértices a partir dos quais se pode atingir v por algum caminho.

Fecho Transitivo Inverso

- Exemplo: O FTI do vértice v_5 do grafo abaixo é o conjunto $\{v_1, v_2, v_4, v_5, v_7\}$. Note que o próprio vértice faz parte do FTD já que ele é alcançável partindo-se dele mesmo.



Conectividade

- Informalmente um grafo é **conexo** (conectado) se for possível caminhar de qualquer vértice para qualquer outro vértice através de uma seqüência de arestas adjacentes.
- Definição: Seja G um grafo. Dois vértices v e w de G estão **conectados** se e somente se existe um caminho de v para w. Um grafo G é conexo se e somente se dado um par qualquer de vértice v e w em G, existe um caminho de v para w.

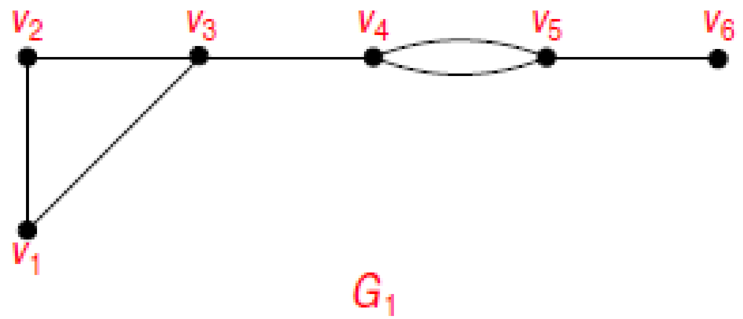
Conectividade

- Ou então:

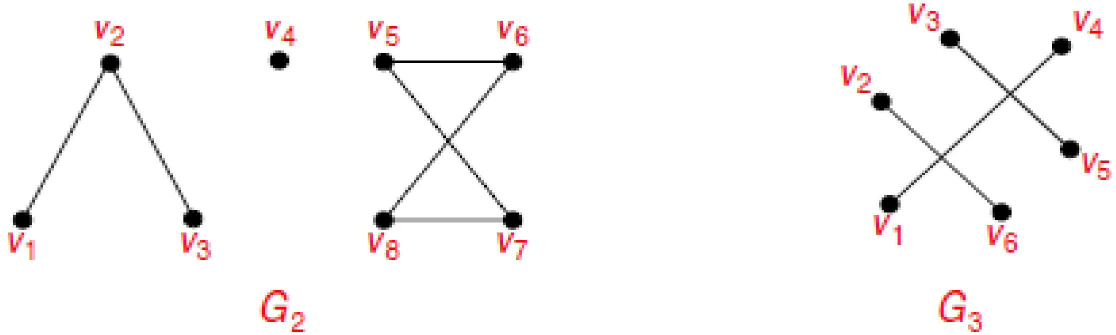
$$G \text{ é } conexo \Leftrightarrow \forall \text{ vértices } v, w \in V(G),$$
$$\exists \text{um caminho de } v \text{ para } w.$$

- Se a negação desta afirmação for tomada, é possível ver que um grafo não é conexo se e somente se existem dois vértices em G que não estão conectados por qualquer caminho.

Conectividade



Grafo conexo.



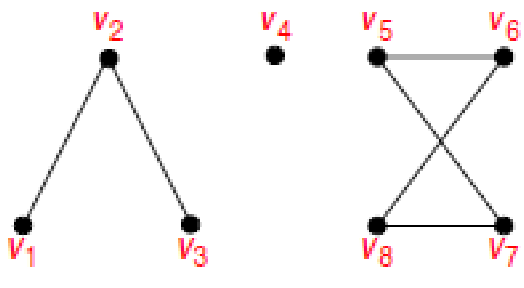
Grafos não conexos.

Conectividade

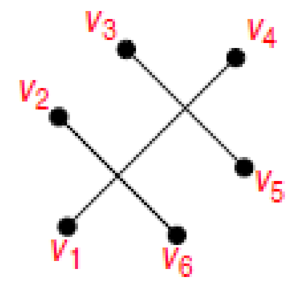
- Seja G um grafo conexo:
  - Se G é conexo, então quaisquer dois vértices distintos de G podem ser conectados por um trajeto simples.
  - Se vértices v e w são parte de um circuito de G e uma aresta é removida do circuito, ainda assim existe um trajeto de v para w em G.
  - Se G é conexo e contém um circuito, então uma aresta do circuito pode ser removida sem desconectar G.

Conectividade

- Os grafos



G_2



G_3

- Possuem três “partes” cada um, sendo cada parte um grafo conexo.
- Um **componente conexo** de um grafo é um subgrafo conexo de maior tamanho possível.

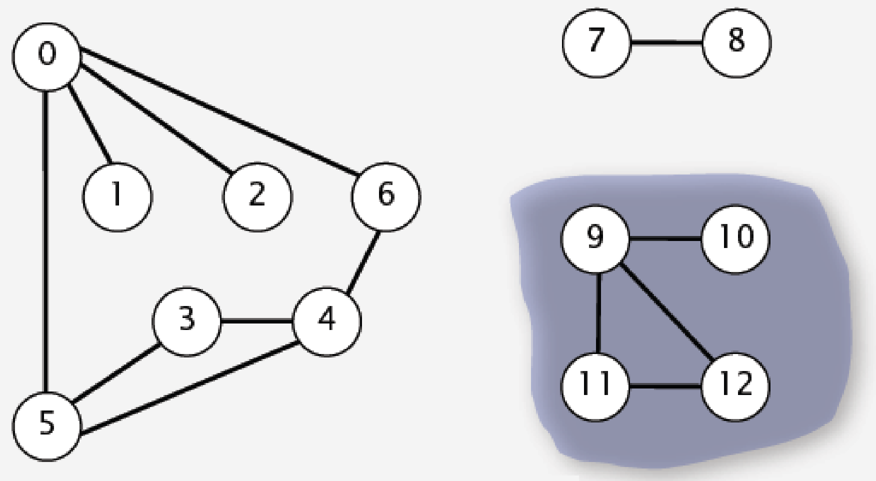
Componentes Conexos

- **Definição(Loureiro):** Um grafo H é um componente conexo de um grafo G se e somente se:
  1. H é um subgrafo de G;
  2. H é conexo;
  3. Nenhum subgrafo conexo I de G tem H como um subgrafo e I contém vértices ou arestas que não estão em H.

  Obs: Um grafo pode ser visto como a união de seus componentes conexos.

Componentes Conexos

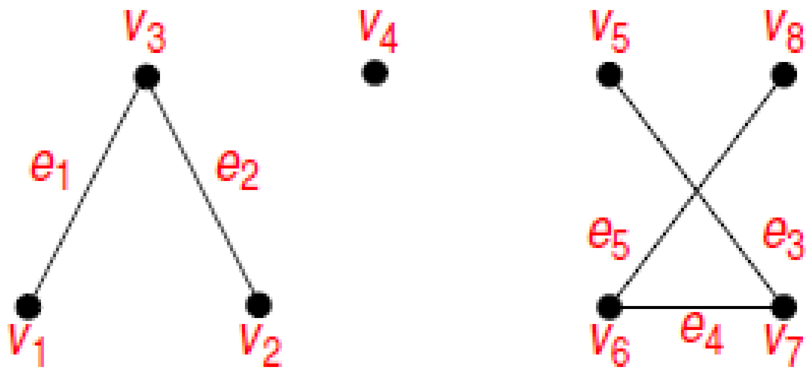
- **Definição(Sedgewick):** Um **componente conexo** é um conjunto máximo de vértices conectados.



v	id[v]
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	1
8	1
9	2
10	2
11	2
12	2

Componentes Conexos

- Os componentes conexos do grafo G abaixo são:



- G possui três componentes conexos:

$$H_1\colon V_1 = \{v_1, v_2, v_3\} \;\; E_1 = \{e_1, e_2\}$$

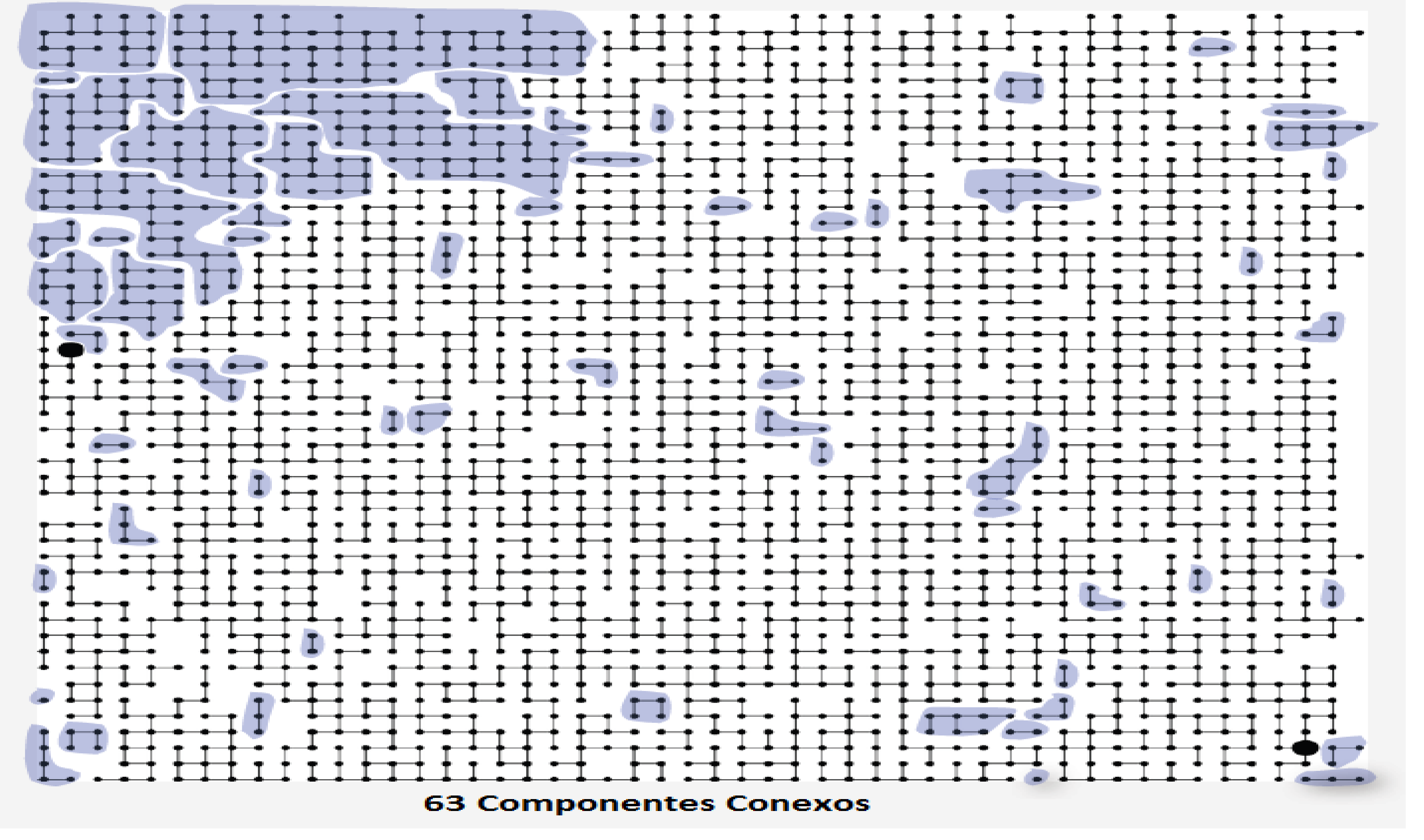
$$H_2\colon V_2 = \{v_4\} E2 = \emptyset;$$

$$H_3\colon V_3 = \{v_5, v_6, v_7, v_8\} \, E3 = \{e_3, e_4, e_5\}$$

Componentes Conexos

- A relação “está conectado a” é:
  - Reflexiva: v está conectado a v.
  - Simétrica: se v está conectado a w, então w está conectado a v.
  - Transitiva: se v está conectado a w, então w está conectado a x, então v está conectado a x.

Componentes Conexos



Componentes Conexos

- Objetivo: Separar os vértices em componentes conexos:

1. Inicialize todos os vértices como não marcados.
2. Para cada vértice v não marcado:
   - Execute a busca em profundidade para identificar todos os vértices de um mesmo componente.

Componentes Conexos

```
public class CC
{
   private boolean marked[];
   private int[] id;
   private int count;

   public CC(Graph G)
   {
      marked = new boolean[G.V()];
      id = new int[G.V()];
      for (int v = 0; v < G.V(); v++)
      {
         if (!marked[v])
         {
            dfs(G, v);
            count++;
         }
      }
   }

   public int count()
   public int id(int v)
   private void dfs(Graph G, int v)

}
```

id[v] - id do componente de v

número de componentes

Executa a busca em profundidade de cada vértice em cada componente

próximo slide

Componentes Conexos

```
public int count()
{  return count;  }

public int id(int v)
{  return id[v];  }

private void dfs(Graph G, int v)
{
   marked[v] = true;
   id[v] = count;
   for (int w : G.adj(v))
      if (!marked[w])
         dfs(G, w);
}
```

número de componentes

id do componente de v

todos os vértices descobertos na mesma chamada da busca em profundidade tem o mesmo id.

Componentes Conexos

Ordenação Topológica

- A busca em profundidade pode ser usada para executar uma ordenação topológica em um grafo dirigido acíclico(DAG – Directed Acyclic Graph).
- Uma ordenação topológica de um DAG $G = (V, E)$ é uma ordenação linear de todos os seus vértices, tal que se G contém uma $aresta\ (u, v)$, então u aparece antes de v na ordenação.

Ordenação Topológica

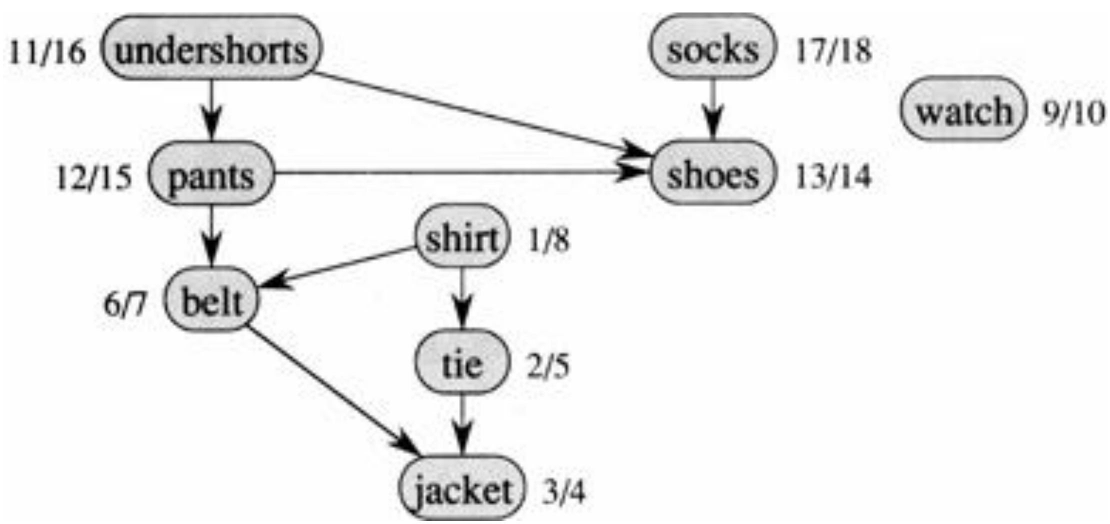
- Se o grafo não é acíclico, então não é possível nenhuma ordenação linear.
- Uma ordenação topológica de um grafo pode ser vista como uma ordenação de seus vértices ao longo de uma linha horizontal, de tal forma que todas as arestas orientadas sigam da esquerda para a direita.

Ordenação Topológica

- DAGs são usados em aplicações para indicar precedência entre eventos.
- O grafo abaixo (resultado de uma pesquisa DFS) mostra como um dado homem se veste pela manhã.
- Uma aresta orientada (u, v) no DAG indica que a peça de roupa u deve ser vestida antes da peça v.

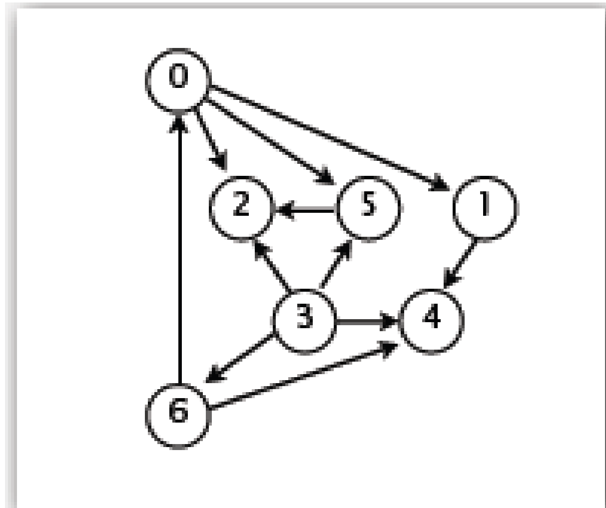
Ordenação Topológica

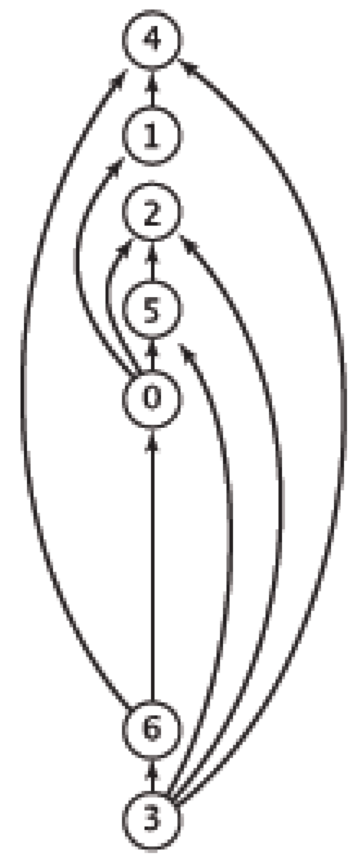
- Algumas peças devem ser vestidas antes de outras (e.g., meias antes dos sapatos);
- Outras, em qualquer ordem (e.g., meias e calças).
- Uma ordenação topológica desse DAG fornece uma ordem para o processode se vestir.



Ordenação Topológica

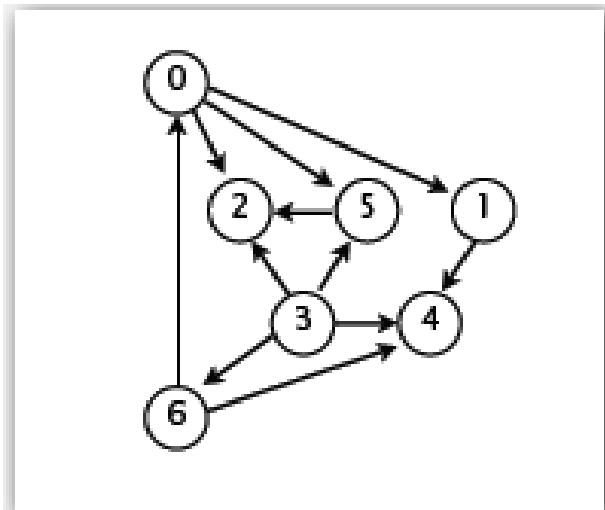
0→5 0→2
0→1 3→6
3→5 3→4
5→4 6→4
6→0 3→2
1→4

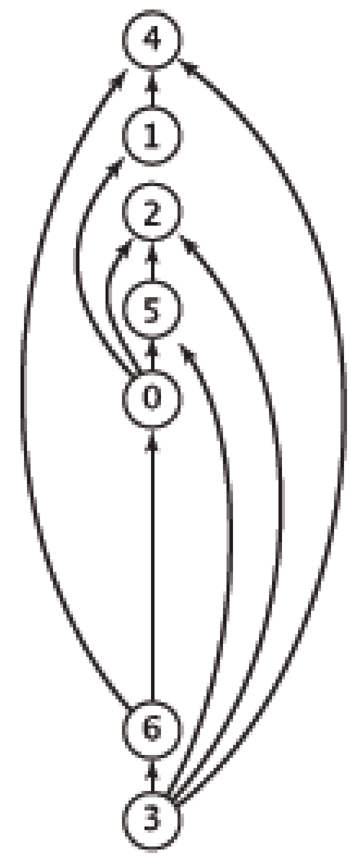




Ordenação Topológica

0→5	0→2
0→1	3→6
3→5	3→4
5→4	6→4
6→0	3→2
1→4	





Ordenação Topológica

```
public class DepthFirstOrder
{
   private boolean[] marked;
   private Stack<Integer> reversePost;

   public DepthFirstOrder(Digraph G)
   {
      reversePost = new Stack<Integer>();
      marked = new boolean[G.V()];
      for (int v = 0; v < G.V(); v++)
         if (!marked[v]) dfs(G, v);
   }

   private void dfs(Digraph G, int v)
   {
      marked[v] = true;
      for (int w : G.adj(v))
         if (!marked[w]) dfs(G, w);
      reversePost.push(v);
   }

   public Iterable<Integer> reversePost()
   {  return reversePost;  }
}
```

Retorna todos os vértices na ordem reversa

Ordenação Topológica

Ordenação Topológica

- Proposição: A busca em profundidade em ordem reversa de um grafo direcionado sem ciclo é uma ordenação topológica.
- Prova: Considere qualquer aresta $v \rightarrow w$. Quando a busca em profundidade é chamada:

1. dfs(w) já foi chamada antes e retornou. Desta maneira, w foi visitados antes de v.
2. dfs(w) ainda não foi chamada. dfs(w) será chamada diretamente or indiretamente por dfs(v) e irá terminar antes de dfs(v). Desta maneira, w será terminado antes de v.
3. dfs(w) já foi chamada, mas ainda não retornou. Não pode acontecer em um grafo direcionado acíclico: A pilha de execução da função contém um caminho de w para v, então $v \rightarrow w$ completaria um ciclo.

Ordenação Topológica

```
dfs(0)
   dfs(1)
      dfs(4)
      4 done
   1 done
   dfs(2)
   2 done
   dfs(5)
      check 2
   5 done
0 done
check 1
check 2
dfs(3)
   check 2
   check 4
   check 5
   dfs(6)
   6 done
3 done
check 4
check 5
check 6
done
```

Ex.: dfs(3)

Caso 1: check 2, check 4, check 5

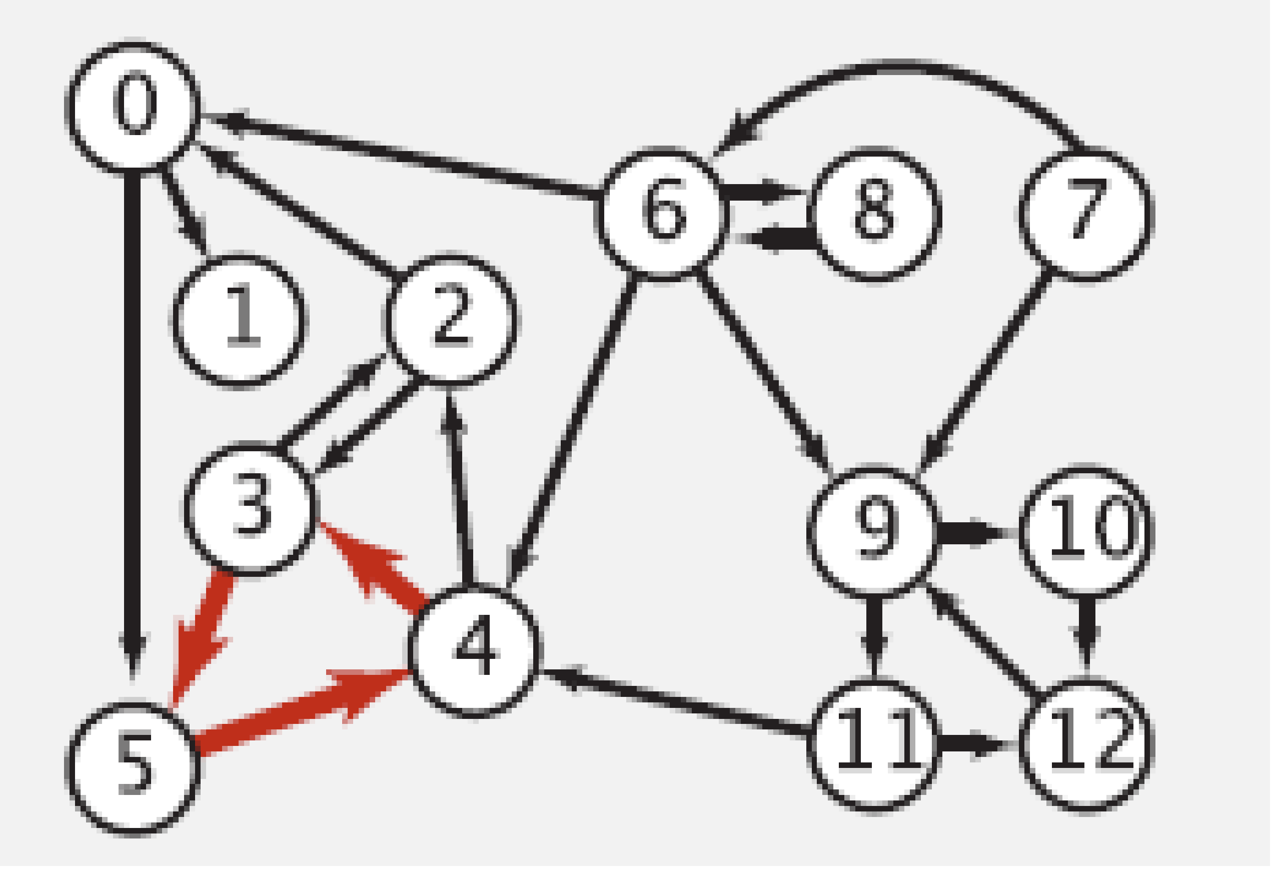
Caso 2: dfs(6), 6 done

Todos os vértices visitados por 3 serão finalizados antes que 3 termine. Desta maneira temos uma ordenação topológica.

Ordenação Topológica

- Proposição: Um digrafo tem uma ordenação topológica se e somente se não possui ciclo direcionado.
- Prova:
  - Se existe um ciclo direcionado, então a ordenação topológica é impossível.
  - Se não existe ciclo direcionado, a busca em profundidade encontrará uma ordenação topológica.

Ordenação Topológica

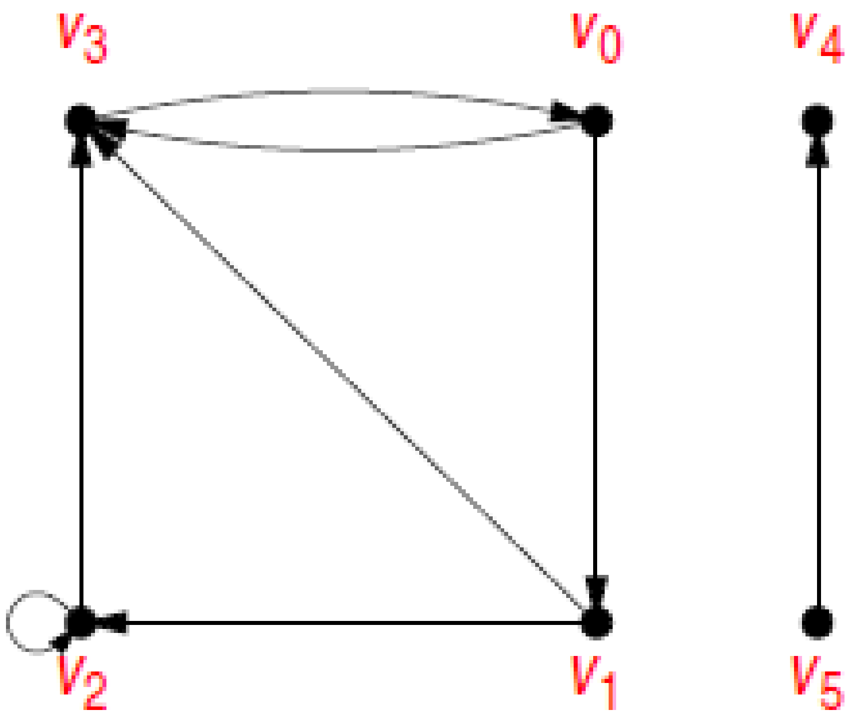


Componentes Fortemente Conexos

- Um grafo dirigido $G = (V, E)$ é **fortemente conexo** se cada dois vértices quaisquer são alcançáveis a partir um do outro.
- Os componentes fortemente conexos de um grafo dirigido são conjuntos de vértices sob a relação "são mutuamente alcançáveis".

uni-bh
A MARCA DA EDUCAÇÃO

Componentes Fortemente Conexos



Os componentes fortemente conexos do grafo ao lado são:

$$H_1 : V_1 = \{v_0, v_1, v_2, v_3\}$$
$$H_2 : V_2 = \{v_4\}$$
$$H_3 : V_3 = \{v_5\}$$

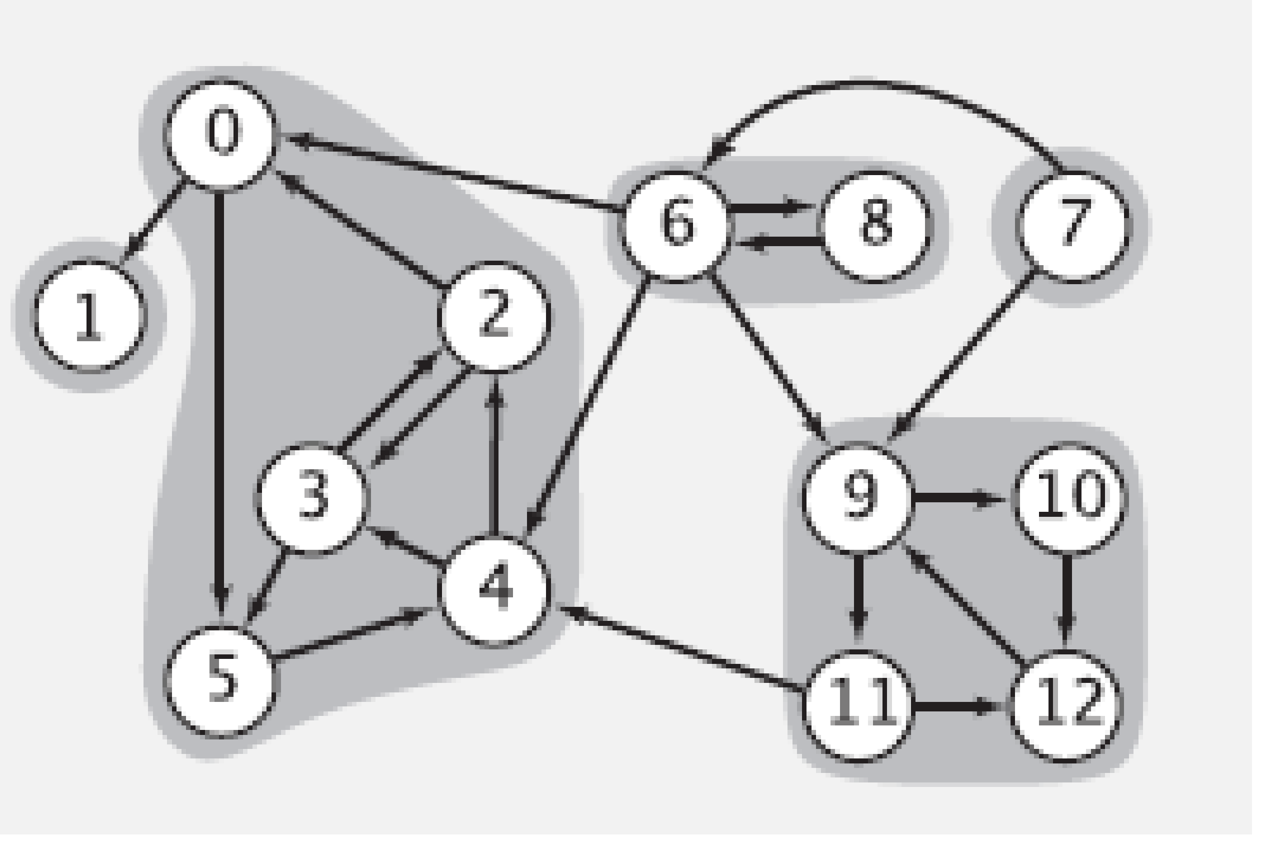
Observe que $\{v_4, v_5\}$ não é um componente fortemente conexo já que o vértice v_5 não é alcançável a partir do vértice v_4.

Componentes Fortemente Conexos

- A relação de conectividade forte é uma relação de equivalência:
  – Reflexiva: v está fortemente conectado a v.
  – Simétrica: se v está fortemente conectado a w, então w está fortemente conectado a v.
  – Transitiva: se v está fortemente conectado a w, então w está fortemente conectado a x, então v está fortemente conectado a x.

Componentes Fortemente Conexos

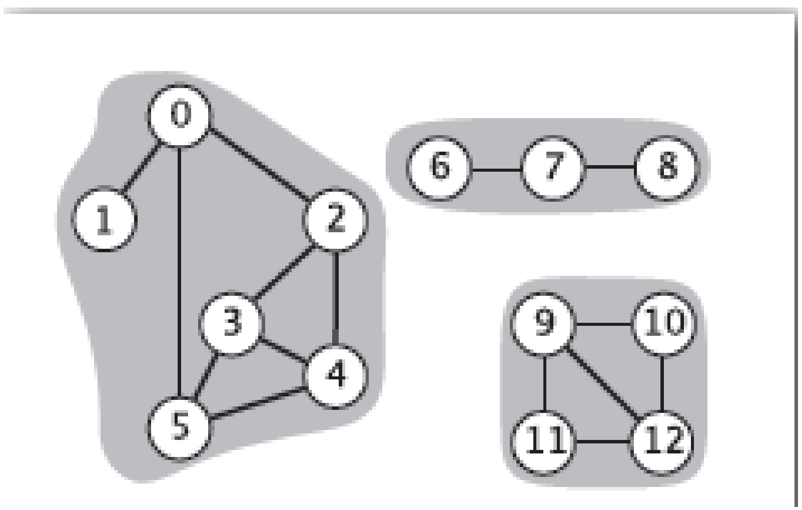
- Definição: Um componente forte é o maior sub-conjunto de vértices fortemente conexos.

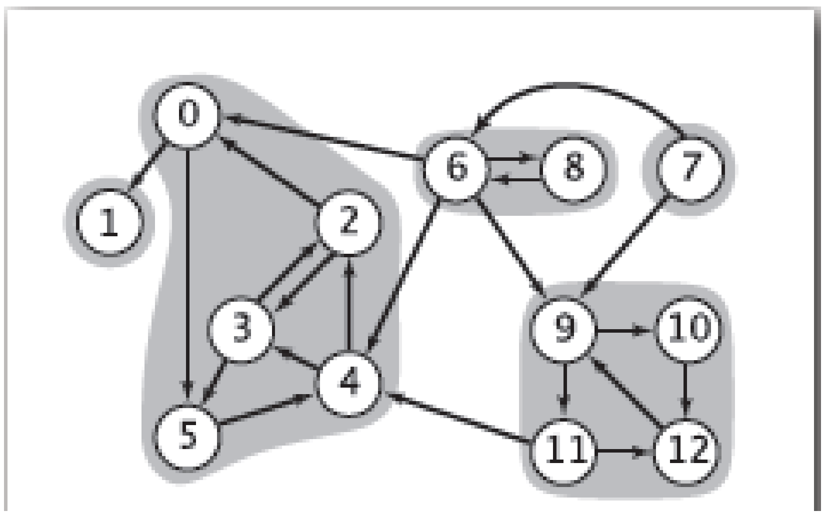


Componentes Conexos vs Componentes Fortemente Conexos

v e w são conectados se existe um caminho entre v e w

v e w são fortemente conexos se existe um caminho direcionado (trajeto) entre v e w e um trajeto entre w e v.





Algoritmo de Kosaraju

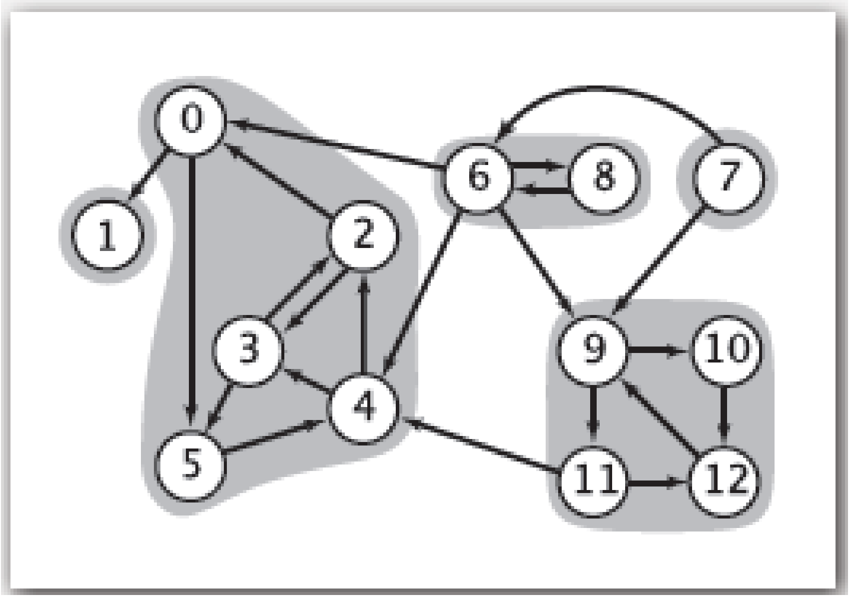
- Grafo reverso: Componentes fortes em G são componentes fortes em $G^{\mathbb{R}}$.
- Kernel do grafo direcionado acíclico: Contrai cada componente forte em um único vértice/
- Idéia:
  - Calcular a ordenação topológica no kernel do grafo direcionado acíclico.
  - Executar a busca em profundidade, considerando os vértices na ordenação topológica reversa.

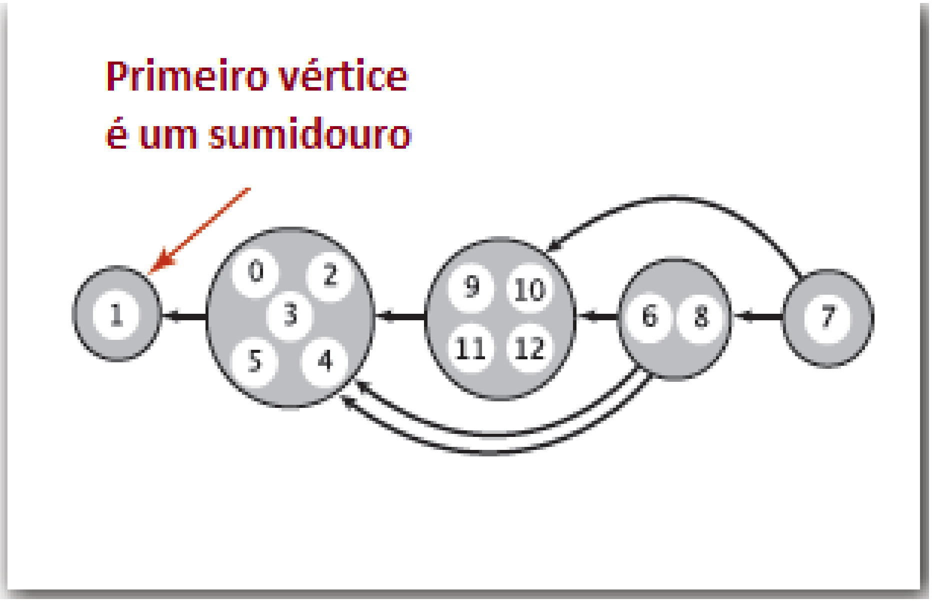
uni-bh A MARCA DA EDUCAÇÃO

Algoritmo de Kosaraju

Digrafo G com seus componentes fortes.

Kernel do Grafo direcionado acíclico considerando os vértices em ordenação topológica reversa.



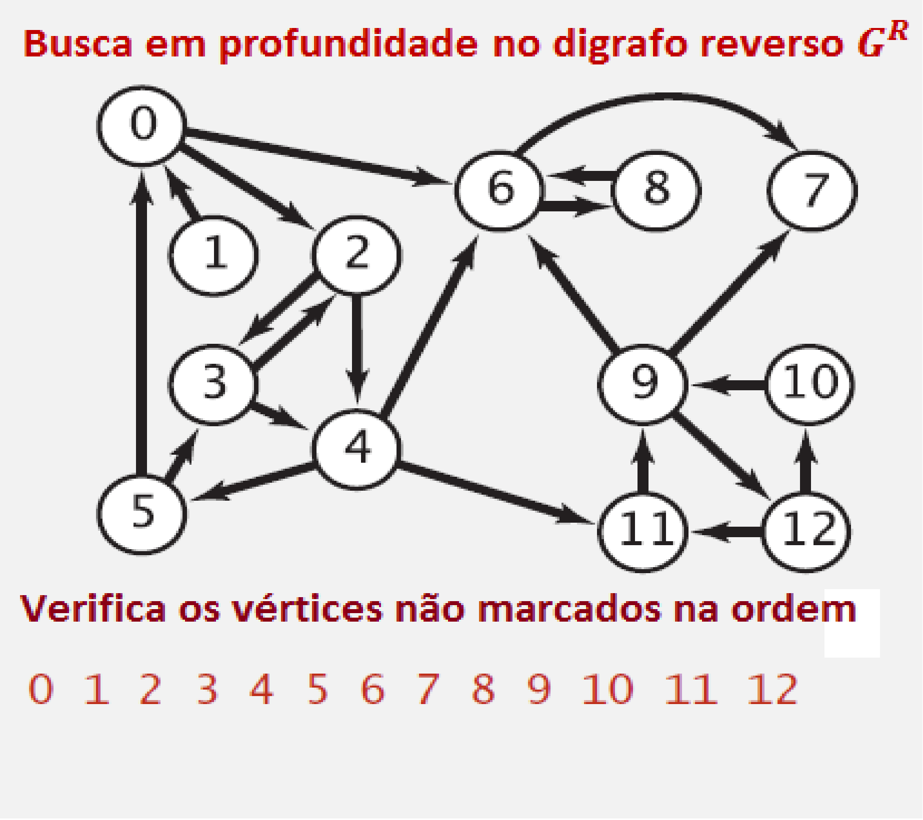


Algoritmo de Kosaraju

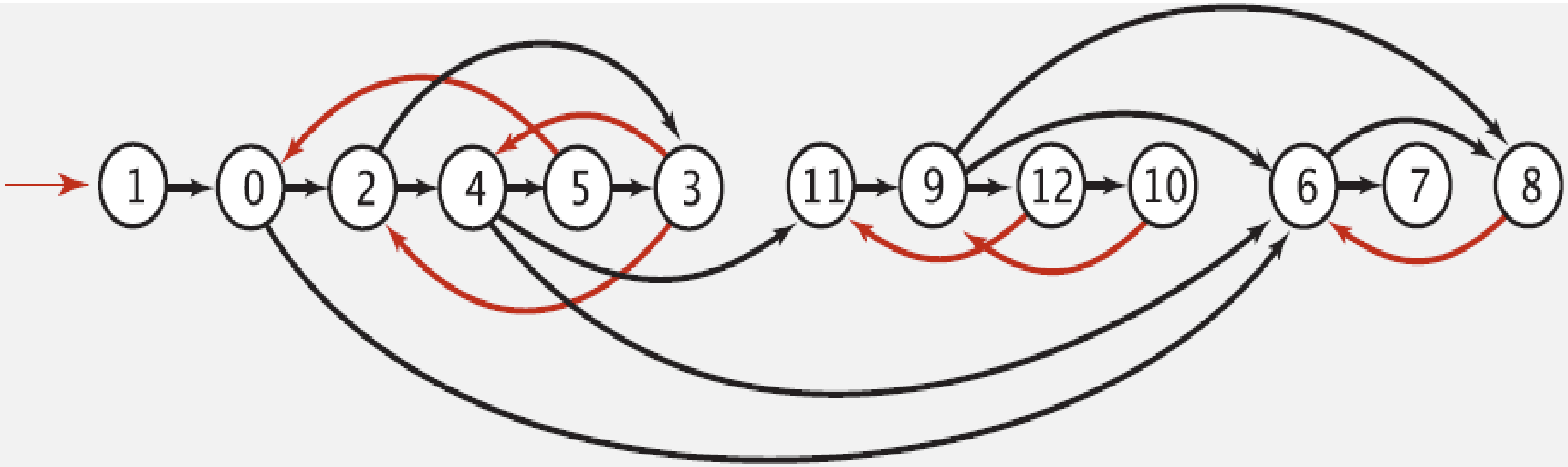
- Algoritmo:
- Execute a busca em profundidade sobre $G^{\mathbb{R}}$ para calcular a pós-ordenação reversa.
- Execute a busca em profundidade sobre G, considerando os vértices na ordem retornada pela primeira busca em profundidade.

Ordenação Topológica

- Execute a busca em profundidade sobre $G^{\mathbb{R}}$ para calcular a pós-ordenação reversa.



Ordenação Topológica



Pós-ordem reversa para uso na segunda busca em profundidade

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

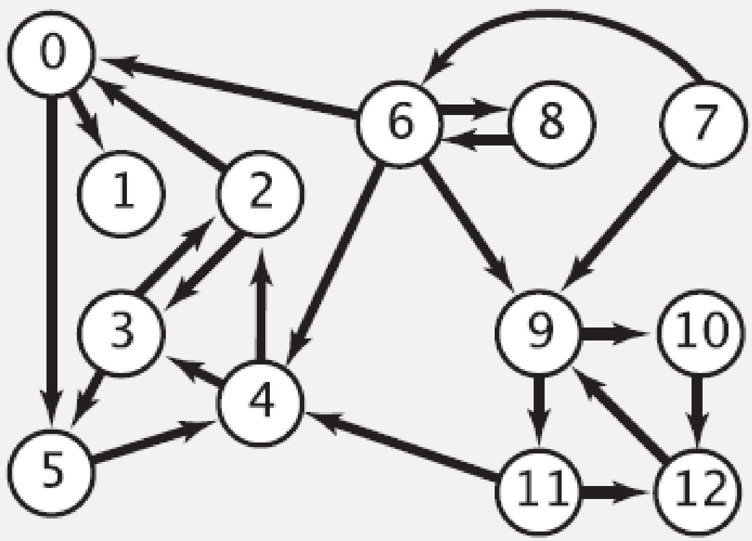
Ordenação Topológica

```
dfs(0)
│  dfs(6)
│  │  dfs(8)
│  │  │  check 6
│  │  8 done
│  │  dfs(7)
│  │  7 done
│  6 done
│  dfs(2)
│  │  dfs(4)
│  │  │  dfs(11)
│  │  │  │  dfs(9)
│  │  │  │  │  dfs(12)
│  │  │  │  │  │  check 11
│  │  │  │  │  │  dfs(10)
│  │  │  │  │  │  │  check 9
│  │  │  │  │  │  10 done
│  │  │  │  │  12 done
│  │  │  │  │  check 7
│  │  │  │  │  check 6
│  │  │  │
. . .
```

Ordenação Topológica

- Execute a busca em profundidade sobre G, considerando os vértices na ordem retornada pela primeira busca em profundidade.

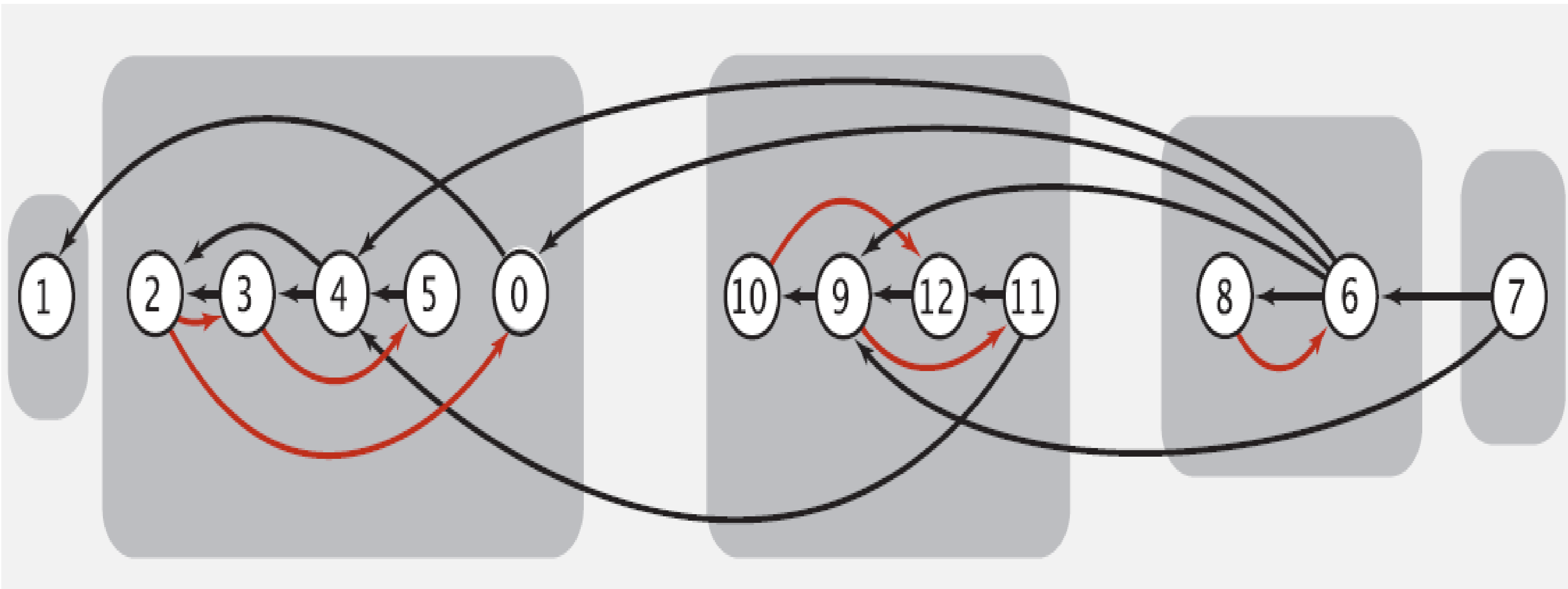
Busca em profundidade no digrafo original G



Visita os vértices na seguinte ordem

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

Ordenação Topológica



Ordenação Topológica

```
dfs(1)
1 done
```

```
dfs(0)
|  dfs(5)
|  |  dfs(4)
|  |  |  dfs(3)
|  |  |  |  check 5
|  |  |  |  dfs(2)
|  |  |  |  |  check 0
|  |  |  |  |  check 3
|  |  |  |  2 done
|  |  |  3 done
|  |  |  check 2
|  |  4 done
|  5 done
|  check 1
0 done
check 2
check 4
check 5
check 3
```

```
dfs(11)
|  check 4
|  dfs(12)
|  |  dfs(9)
|  |  |  check 11
|  |  |  dfs(10)
|  |  |  |  check 12
|  |  |  10 done
|  |  9 done
|  12 done
11 done
check 9
check 12
check 10
```

Ordenação Topológica

```
dfs(6)
|  check 9
|  check 4
|  dfs(8)
|  |  check 6
|  8 done
|  check 0
6 done
```

```
dfs(7)
|  check 6
|  check 9
7 done
check 8
```

Ordenação Topológica

```
public CC(Graph G)
{
   marked = new boolean[G.V()];
   id = new int[G.V()];

   for (int v = 0; v < G.V(); v++)
   {
      if (!marked[v])
      {
         dfs(G, v);
         count++;
      }
   }
}
```

```
public KosarajuSCC(Digraph G)
{
   marked = new boolean[G.V()];
   id = new int[G.V()];
   DepthFirstOrder dfs = new DepthFirstOrder(G.reverse());
   for (int v : dfs.reversePost())
   {
      if (!marked[v])
      {
         dfs(G, v);
         count++;
      }
   }
}
```

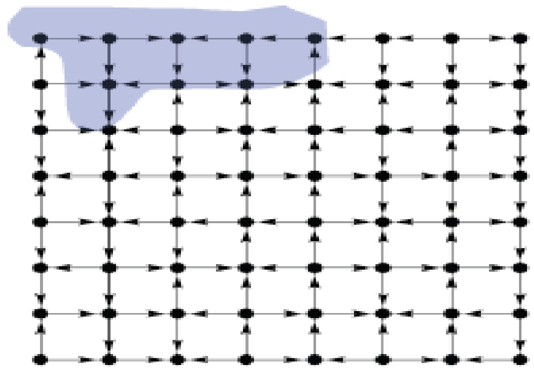
Ordenação Topológica

```
private void dfs(Graph G, int v)
{
   marked[v] = true;
   id[v] = count;
   for (int w : G.adj(v))
      if (!marked[w])
         dfs(G, w);
}

public boolean connected(int v, int w)
{  return id[v] == id[w];  }
```

```
private void dfs(Digraph G, int v)
{
   marked[v] = true;
   id[v] = count;
   for (int w : G.adj(v))
      if (!marked[w])
         dfs(G, w);
}

public boolean stronglyConnected(int v, int w)
{  return id[v] == id[w];  }
```

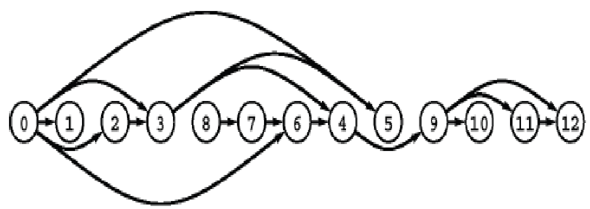
Resumo dos algoritmos

Alcançabilidade por uma única fonte



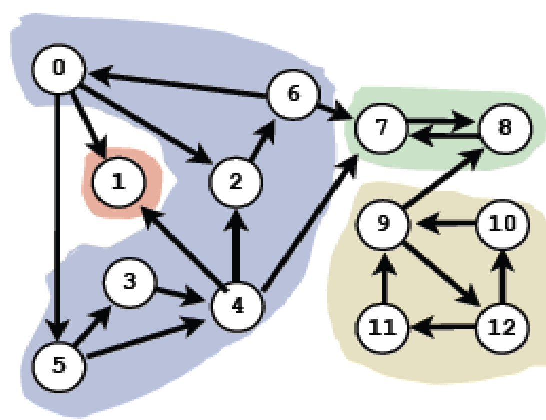
Busca em profundidade

Ordenação Topológica



Busca em profundidade

Componentes fortes



Algoritmo de Kosaraju
Busca em largura (2 vezes)