Compacto dos melhores momentos

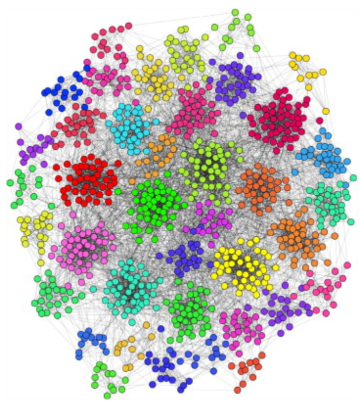
AULA 23

BFS versus DFS

- busca em largura usa **fila**, busca em profundidade usa **pilha**
- a busca em largura é descrita em **estilo iterativo**, enquanto a busca em profundidade é descrita, usualmente, em **estilo recursivo**
- busca em largura começa tipicamente num **vértice especificado**, a busca em profundidade, o próprio **algoritmo escolhe o vértice** inicial
- a busca em largura apenas **visita os vértices que podem ser atingidos** a partir do vértice inicial, a busca em profundidade, tipicamente, **visita todos os vértices** do digrafo

AULA 23

Componentes de grafos

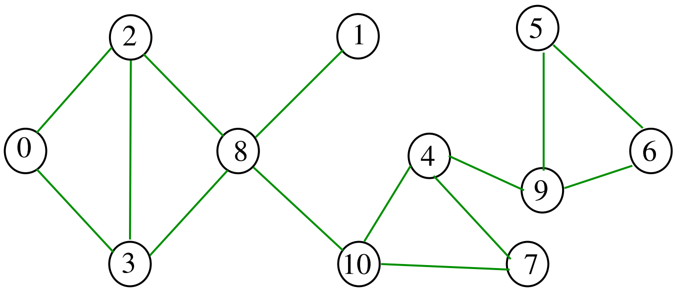


Fonte: `Personalized PageRank Clustering: A graph clustering algorithm based on random walks`

Grafos conexos

Um grafo é **conexo** se e somente se, para cada par (`s`,`t`) de seus vértices, existe um caminho com origem `s` e término `t`

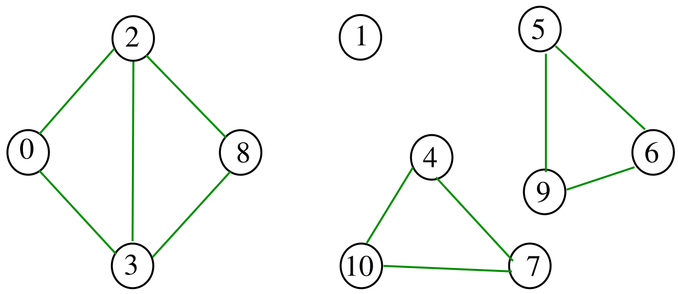
Exemplo: um grafo conexo



Componentes de grafos

Um **componente** (= *component*) de um grafo é o subgrafo conexo maximal

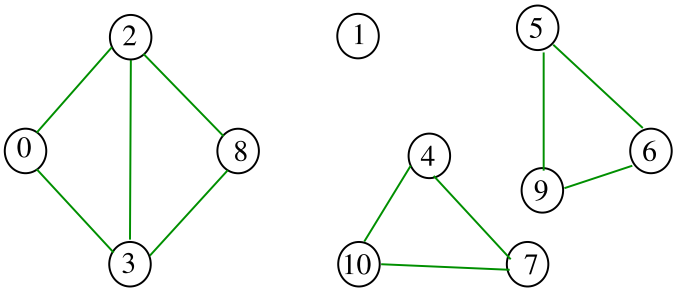
Exemplo: grafo com 4 componentes (conexos)



Contando componentes

Problema: calcular o número de componente

Exemplo: grafo com 4 componentes



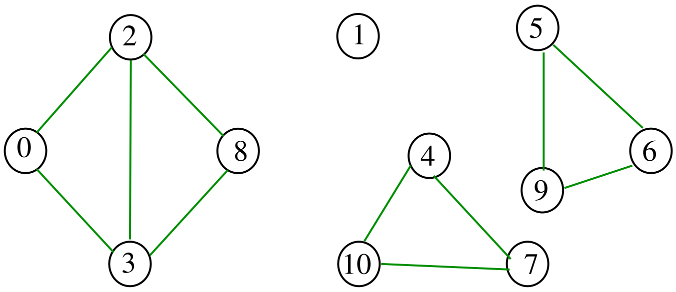
Cálculo das componentes de grafos

O classe `DFScc` determina o número de componentes do grafo `G`.

Além disso, ela armazena no vetor `id[]` o número da componente a que o vértice pertence: se o vértice `v` pertence a `k`-ésima componente então `id[v] == k-1`

Exemplo

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



Classe DFScc: esqueleto

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

   public DFScc(Graph G) {...}
   private void dfs(Digraph G, int v) {}
   public boolean connected(int v, int w)
   {...}
   public int id(int v) {...}
```

DFScc

Determina as componentes de um dado grafo `G`.

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

DFScc: dfs()

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

DFScc: connected(), id(), count()

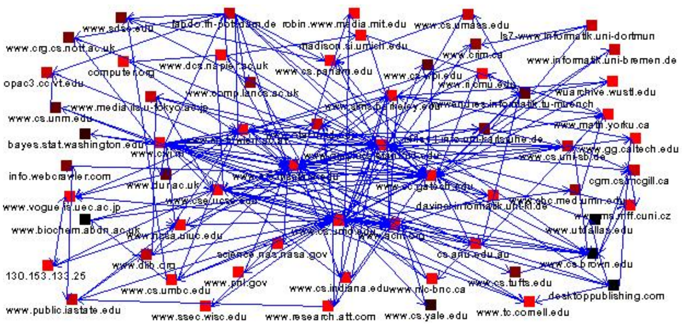
```
public int id(int v) { // CC
   return id[v];
}

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

public int count(int v) { // CC
   return count;
}
```

Consumo de tempo

O consumo de tempo de DFScc para vetor de listas de adjacência é $O(V + E)$.

O consumo de tempo de DFScc para matriz de adjacências é $O(V^2)$.

Componentes fortemente conexos

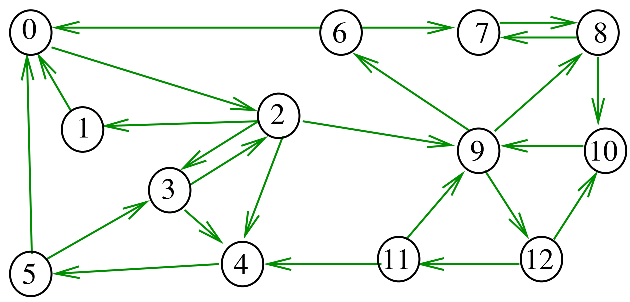


Fonte: A System for Collecting and Analyzing Topic-Specific Web Information

Digrafos fortemente conexos

Um digrafo é **fortemente conexo** se e somente se para cada par {s,t} de seus vértices, existem caminhos de s a t e de t a s

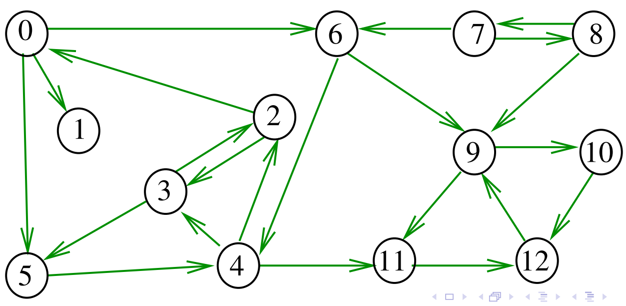
Exemplo: um digrafo fortemente conexo



Componentes fortemente conexos

Um componente **fortemente conexo** (= *strongly connected component* (SCC)) é um conjunto maximal de vértices W tal que o digrafo induzido por W é fortemente conexo

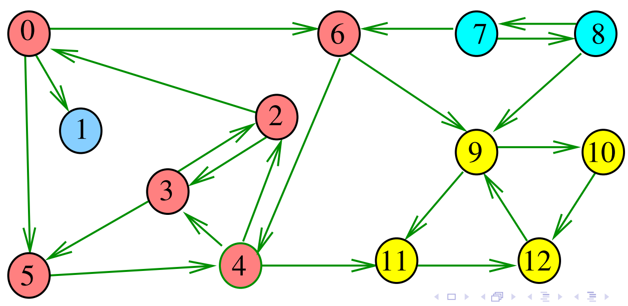
Exemplo: 4 componentes fortemente conexos



Componentes fortemente conexos

Um componente **fortemente conexo** (= *strongly connected component* (SCC)) é um conjunto maximal de vértices W tal que o digrafo induzido por W é fortemente conexo

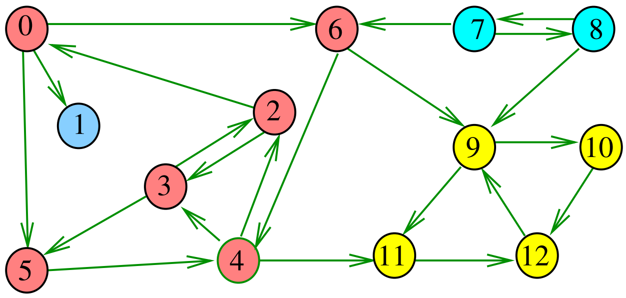
Exemplo: 4 componentes fortemente conexos



Determinando componentes f.c.

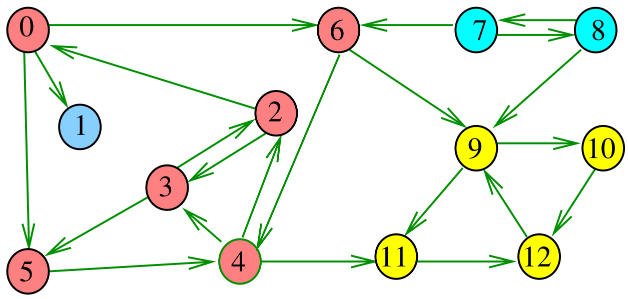
Problema: determinar os componentes fortemente conexos

Exemplo: 4 componentes fortemente conexos



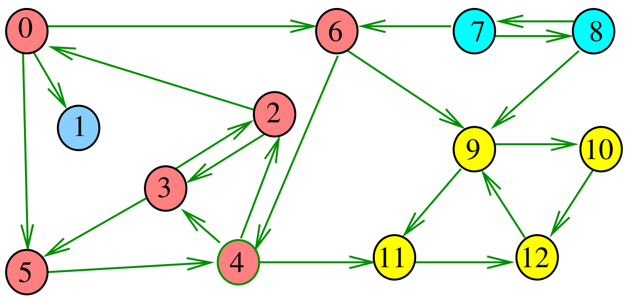
Exemplo

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



Exemplo

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



Força Bruta: esqueleto

```
public class SCCforcaBruta {
   private DFScc cc;
   public SCCforcaBruta(Digraph G) {...}
   public boolean sConnected(int v, int w)
   {...}
   public int id(int v) {...}
   public int count(int v) {...}
}
```

Força Bruta

```
public SCCforcaBruta(Digraph G) {
   Graph H = new Graph(G.V());
   for (int v = 0; v < G.V(); v++) {
      DFSpaths dfsV = new DFSpaths(G, v);
      for(int w=v+1; w < G.V(); w++) {
         DFSpaths dfsW=new DFSpaths(G,w);
         if (dfsV.hasPath(w) &&
             dfsW.hasPath(v))
             H.addEdge(v, w);
   }
   cc = new DFScc(H);
}
```

stronglyConnected

```
public int id(int v) { // SCC
   return cc.id(v);
}

public boolean sConnected(int v,int w) {
   return cc.connected(v, w);
}

public int count(int v) { // SCC
   return cc.count;
}
```

Consumo de tempo

O consumo de tempo de `SCCforcaBruta` para vetor de listas de adjacência é $O(V^2(V+E))$.

O consumo de tempo de `SCCforcaBruta` para matriz de adjacência é $O(V^4)$.

Algoritmos Tarjan, Kosaraju e Sharir

Robert Endre Tarjan (1972), Sambasiva Rao Kosaraju (1978) e Micha Sharir (1981) desenvolveram algoritmos que consomem tempo $O(V+E)$ para calcular os componentes f.c. de um digrafo G

Esses algoritmos utilizam DFS de uma maneira fundamental.

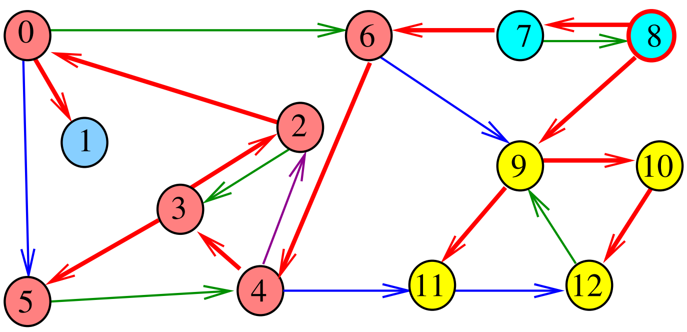
Tarjan realiza apenas um passo DFS sobre o digrafo.

Kosaraju e Sharir fazem duas passadas DFS.

Discutiremos o algoritmo de Kosaraju e Sharir.

Propriedade

Vértices de um componente fortemente conexo são uma **subarborescência** em uma floresta DFS



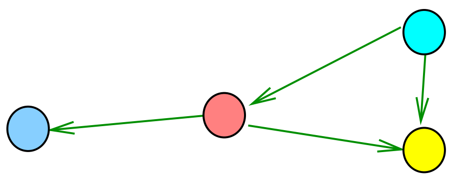
Digrafos dos componentes

O **digrafo dos componentes** de G tem um vértice para cada componente fortemente conexo e um arco U-W se G possui um arco com ponta inicial em U e ponta final em W

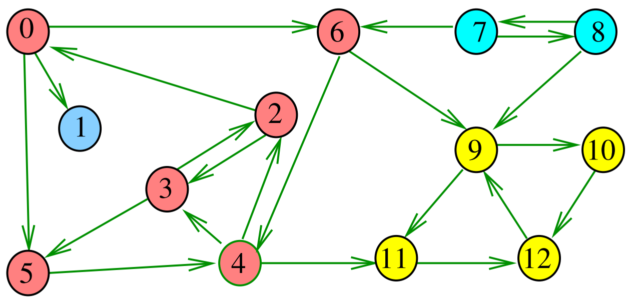
Digrafos dos componentes

O **digrafo dos componentes** de G tem um vértice para cada componente fortemente conexo e um arco U-W se G possui um arco com ponta inicial em U e ponta final em W

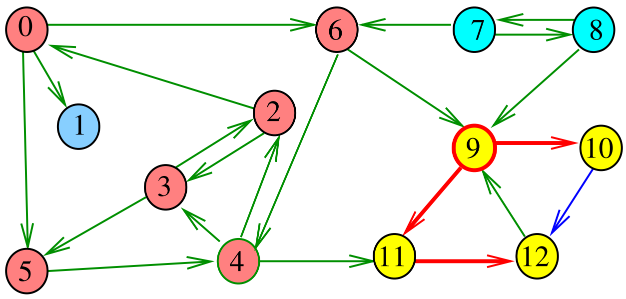
Digrafo dos componente é um DAG



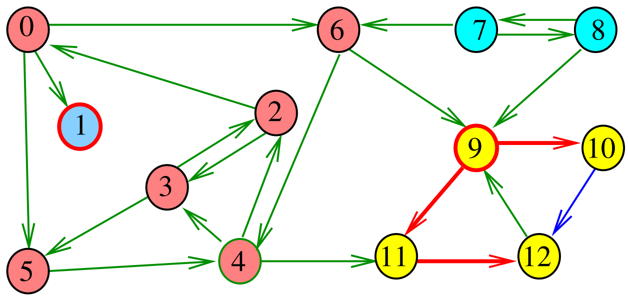
Ideia . . . G e DFS



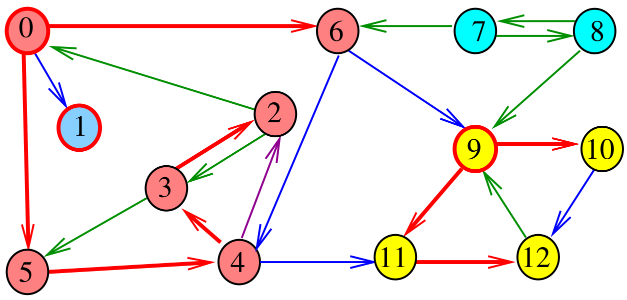
Ideia . . . G e DFS



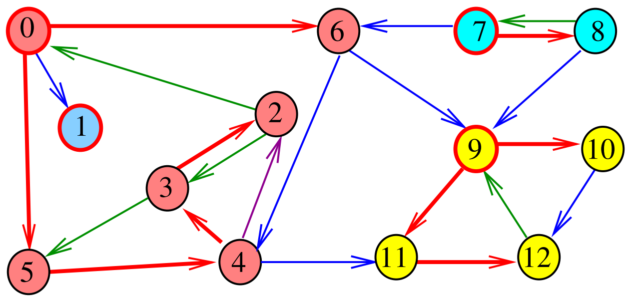
Ideia . . . G e DFS



Ideia . . . G e DFS

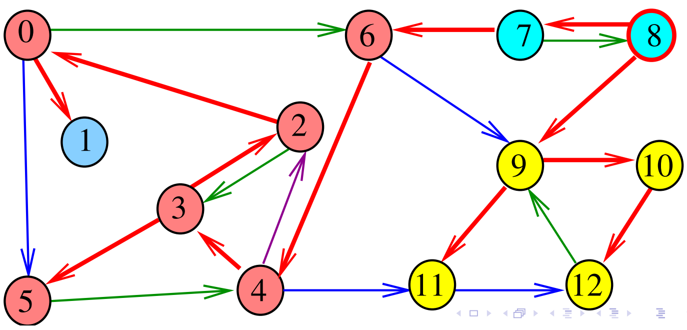


Ideia . . . G e DFS



Numeração pós-ordem

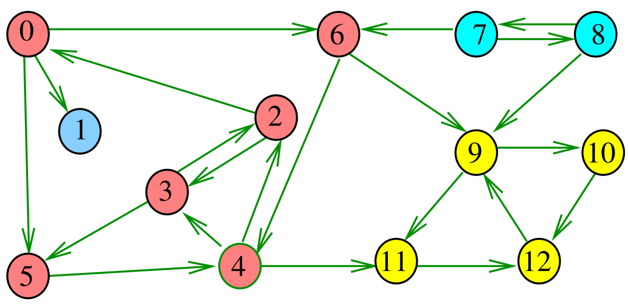
pós[v] = numeração pós-ordem de v
sóp[i] = vértice de numeração pós-ordem i
pós[W] = maior numeração pós-ordem de um vértice em W



Propriedade

Um digrafo G e seu digrafo reverso R têm os mesmos componente fortemente conexos

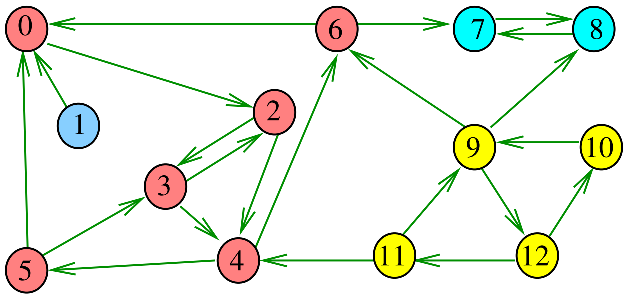
Exemplo: Digrafo G



Propriedade

Um digrafo G e seu digrafo reverso R têm os mesmos componente fortemente conexos

Exemplo: Digrafo reverso R de G



G, G reverso, DFS e pós []

Fato. Se pós[v] > pós[w] e existem um caminho de w a v, então existe um caminho de v a w.

Em outras palavras:

Fato. Se pós[v] > pós[w] e existem um caminho de w a v, então v e w estão em um mesmo componente fortemente conexo..

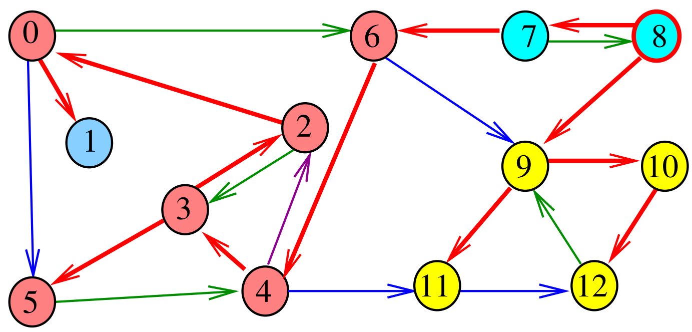
G, G reverso, DFS e pós []

Algoritmo de Kosaraju: aplique DFS no grafo reverso R de G e compute pós []. Em seguida

- pegue o vértice v tal que pós[v] é máximo (= pós [] reversa);
- determine o conjunto W = {w : existe caminho de v a w em G}
- para w em W existe em R um caminho de w a v.
- **Fato** ⇒ W forma um componente f.c. de R, e portanto de G;
- remova W de G e pegue o vértice v tal pós[v]. . .

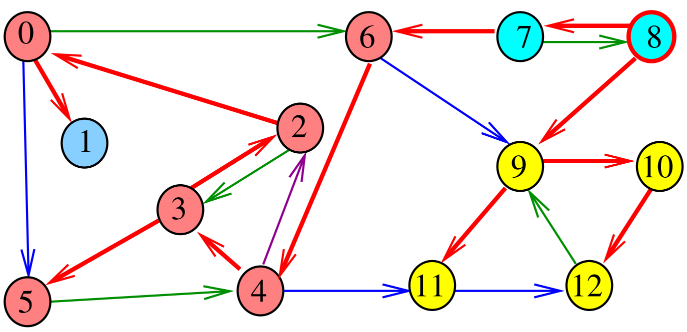
Exemplo

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



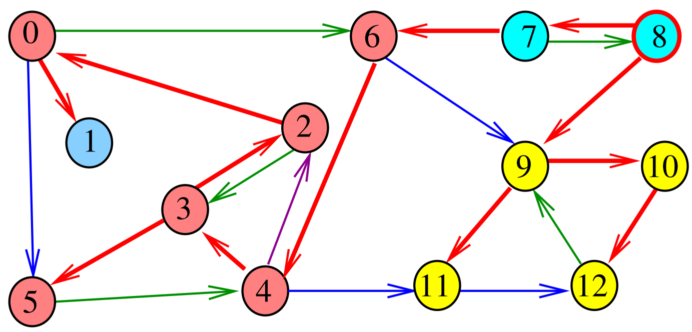
Exemplo

i	0	1	2	3	4	5	6	7	8	9	10	11	12
sóp[i]	12	10	11	**9**	5	**1**	0	2	3	4	**6**	7	**8**



Exemplo

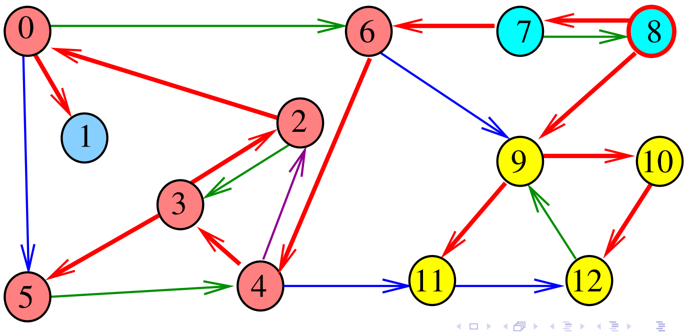
pós[{7, **8**}] = 12
pós[{0, 2, 3, 4, 5, **6**}] = 10
pós[{**1**}] = 5
pós[{**9**, 10, 11, 12}] = 3



Numeração pós-ordem e componentes f.c.

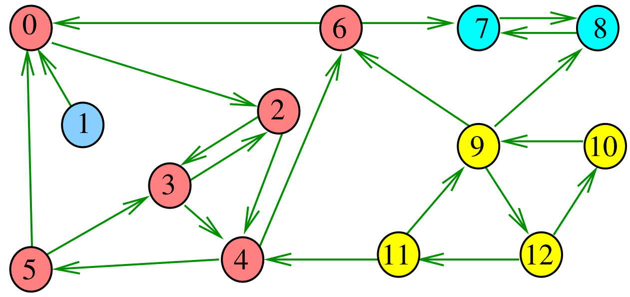
Se U e W são componentes f.c. e existe arco com ponta inicial em U e ponta final em W, então

$$\text{pós}[U] > \text{pós}[W]$$



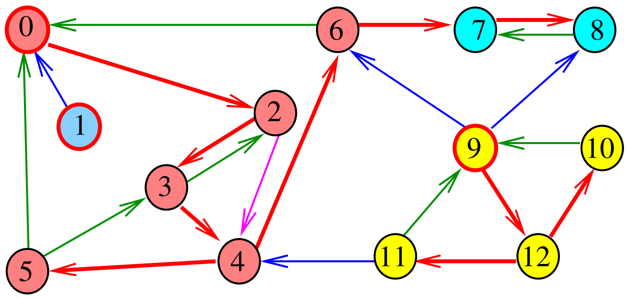
Digrafo reverso R

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



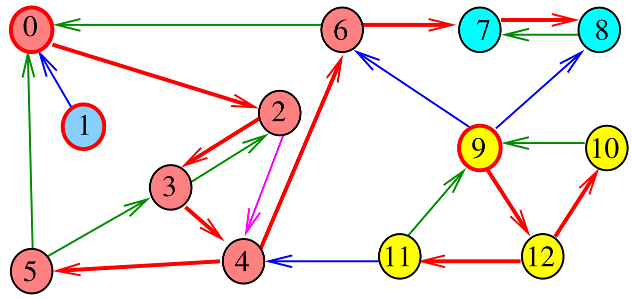
Digrafo reverso R e DFS

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



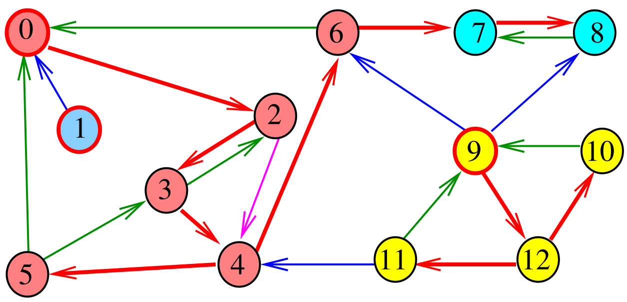
Digrafo reverso R e DFS

i	0	1	2	3	4	5	6	7	8	9	10	11	12
sóp[i]	8	**7**	6	5	4	3	2	**0**	**1**	10	11	12	**9**



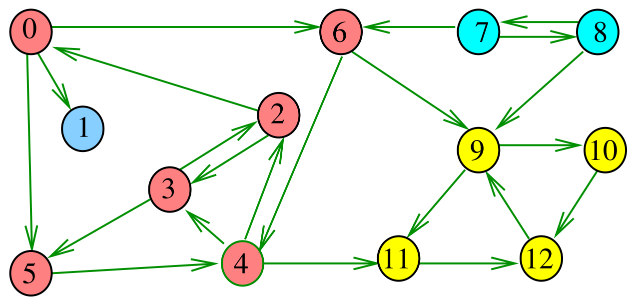
Digrafo reverso R e DFS

i	0	**1**	2	3	4	5	6	**7**	**8**	9	10	11	**12**
sóp[i]	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**0**	**1**	**10**	**11**	**12**	**9**



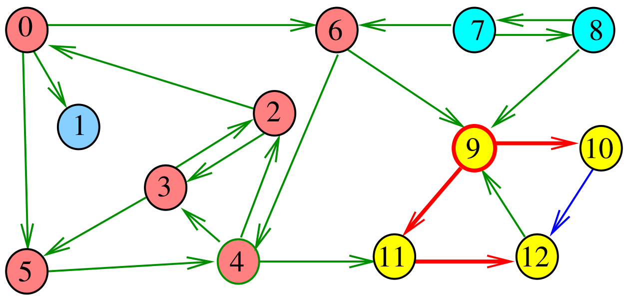
Digrafo G e DFS

i	0	**1**	2	3	4	5	6	**7**	**8**	9	10	11	**12**
sóp[i]	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**0**	**1**	**10**	**11**	**12**	**9**



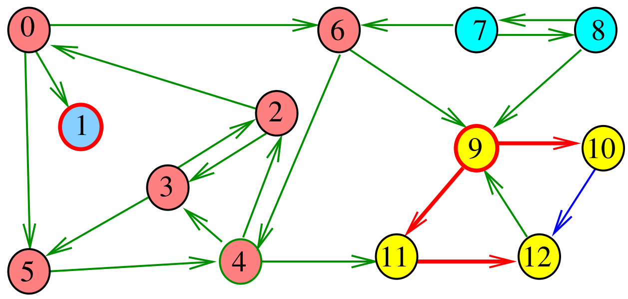
Digrafo G e DFS

i	0	1	2	3	4	5	6	7	8	9	10	11	12
sóp[i]	8	7	6	5	4	3	2	0	1	10	11	12	9



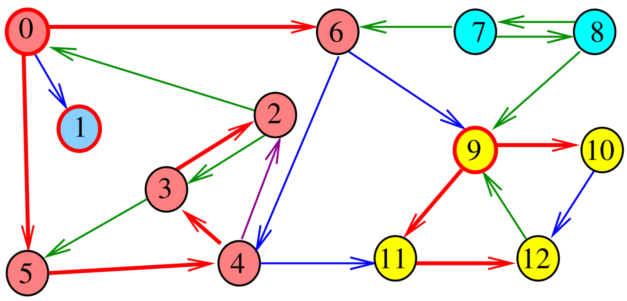
Digrafo G e DFS

i	0	1	2	3	4	5	6	7	8	9	10	11	12
sóp[i]	8	7	6	5	4	3	2	0	1	10	11	12	9



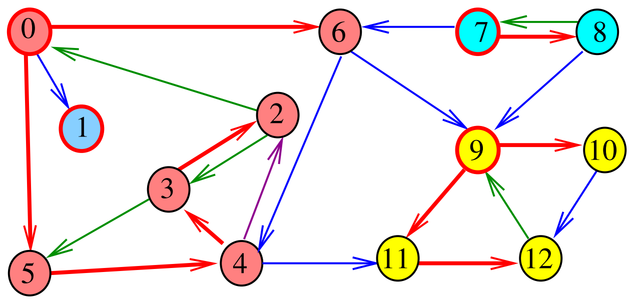
Digrafo G e DFS

i	0	1	2	3	4	5	6	7	8	9	10	11	12
sóp[i]	8	7	6	5	4	3	2	0	1	10	11	12	9



Digrafo G e DFS

i	0	1	2	3	4	5	6	7	8	9	10	11	12
sóp[i]	8	7	6	5	4	3	2	0	1	10	11	12	9



Algoritmo de Kosaraju e Sharir

A classe DFSscc calcula os componentes fortemente conexos do digrafo G

```
private boolean[] marked;
private int[] id;
private int count; // no.  de scc
```

Ela armazena no vetor id[] o número do componente a que o vértice pertence: se o vértice v pertence ao k-ésimo componente então id[v] == k-1

Classe DFSscc: esqueleto

```
public class DFSscc {
   private boolean[] marked;
   private int count; // SCC
   private int[] id; // SCC
   public DFSscc(Graph G) {...}
   private void dfs(Digraph G, int v) {}
   public boolean sConnected(int v, int w)
   {...}
   public int id(int v) {...}
   public int count(int v) {...}
}
```

DFSscc

```
public DFSscc(Digraph G) {
  // computa uma pós-ordem reversa
  DFSanatomia dfs;
  dfs = new DFSanatomia(G.reverse());
  // contrói floresta DFS de G
  marked = new boolean[G.V()];
  id = new int[G.V()];
  for (int v:  dfs.revPos())
     if (!marked[v]) {
        dfs(G, v);
        count++;
     }
}
```

DFSscc: dfs()

```
// DFS on graph G
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);
  }
}
```

DFSscc

```
// no.  de comps fortemente conexos
public int count() {
  return count;
}
// v e w estão no mesmo comp f.c.?
public boolean sConnected(int v, int w) {
  return id[v] == id[w];
}
// id do comp fort.  conexo de v
public int id(int v) {
  return id[v];
}
```

Digraph: G.reverse()

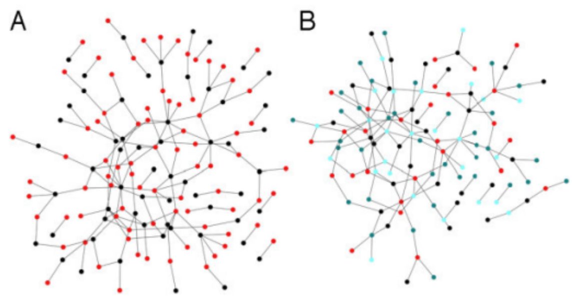
```
public Digraph reverse () {
  Digraph reverse = new Digraph(V);
  for (int v = 0; v < V; v++) {
     for (int w:  adj(v)) {
        reverse.addEdge(w, v);
     }
  }
  return reverse;
}
```

Consumo de tempo

O consumo de tempo de DFSscc para listas de adjacência é $O(V + E)$.

O consumo de tempo de DFSscc matriz de adjacências é $O(V^2)$.

Apêndice: grafos bipartidos e ciclos ímpares

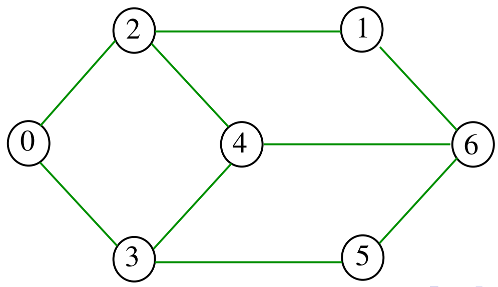


Fonte: Modularity and anti-modularity in networks with arbitrary degree distribution

Bipartição

Um grafo é **bipartido** (= *bipartite*) se existe uma bipartição do seu conjunto de vértices tal que cada aresta tem uma ponta em uma das partes da bipartição e a outra ponta na outra parte.

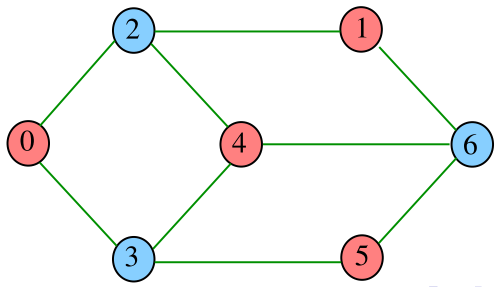
Exemplo:



Bipartição

Um grafo é **bipartido** (= *bipartite*) se existe uma bipartição do seu conjunto de vértices tal que cada aresta tem uma ponta em uma das partes da bipartição e a outra ponta na outra parte.

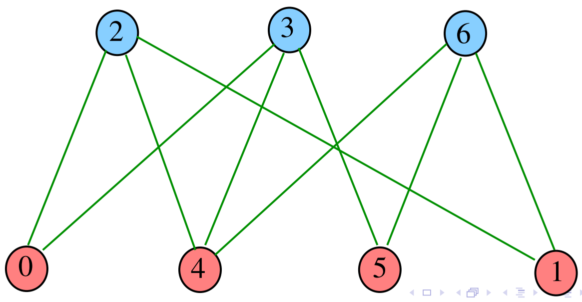
Exemplo:



Bipartição

Um grafo é **bipartido** (= *bipartite*) se existe uma bipartição do seu conjunto de vértices tal que cada aresta tem uma ponta em uma das partes da bipartição e a outra ponta na outra parte.

Exemplo:



Class DFSbipartite

A classe decide se um dado grafo `G` é bipartido.

Nossos grafos têm `G.V()` vértices.

Se `G` é bipartido, o método `dfs()` atribui uma "cor" a cada vértice de `G` de tal forma que toda aresta tenha pontas de cores diferentes

As cores dos vértices, true e false, são registradas no vetor `color` indexado pelos vértices:

```
private boolean color=new boolean[G.V()];
```

DFSbipartite: esqueleto

```
public class DFSbipartite {
  private boolean[] marked;
  private int[] edgeTo;
  private boolean[] color; // TwoColor
  private boolean isTwoColorable= true;
  private Stack<Integer> cycle;
  private int onCycle = -1;
  public DFSbipartite(Graph G) {...}
  private void dfs(Digraph G, int v){...}
  public boolean isBipartite() {...}
  public Iterable<Integer> cycle() {...}
}
```

DFSbipartite

```
public DFSbipartite(Graph G) {
  marked = new boolean[G.V()];
  edgeTo = new int[G.V()];
  color = new boolean[G.V()];
  for (int v = 0; v < G.V(); v++)
     if (!marked(v)) {
        dfs(G,v);
     }
}
```

DFSbipartite: dfs()

```
private void dfs(Digraph G, int v) {
   marked[v] = true;
   for (int w :  G.adj(v)) {
       if (!marked(w)) {
          color[w] = !color[v];
          edgeTo[w] = v;
          dfs(G, w);
          if (hasCycle()) return;
       } else if (color[v] == color[w]) {
          isTwoColorable= false;
          onCycle = v;
          edgeTo[v] = w; // fecha o ciclo
       }
   }
```

DFSbipartite

```
public boolean isBipartite() {
   return isTwoColorable;
}

public Iterable<Integer> cycle() {
   if (isTwoColorable) return null;
   if (cycle != null) return cycle;
   cycle = new Stack<Integer>();
   for (int x=edgeTo[onCycle]; x!=onCycle;
            x = edgeTo[x])
      cycle.push(x);
   cycle.push(onCycle);
   return cycle;
}
```

Consumo de tempo

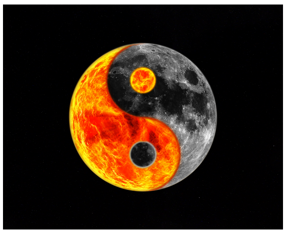
A classe DFSbipartite, para vetor de listas de adjacência, consome tempo $O(V+E)$ para decidir se um grafo é bipartido.

A classe DFSbipartite, para matriz de adjacências, consome tempo $O(V^2)$ para decidir se um grafo é bipartido.

Certificado

Para todo grafo G, vale uma e apenas umas das seguintes afirmações:

- G possui um ciclo ímpar
- G é bipartido



Fonte: Yin and Yang Yoga Workshop