Fonte: ash.atozviews.com

Compacto de alguns dos melhores momentos

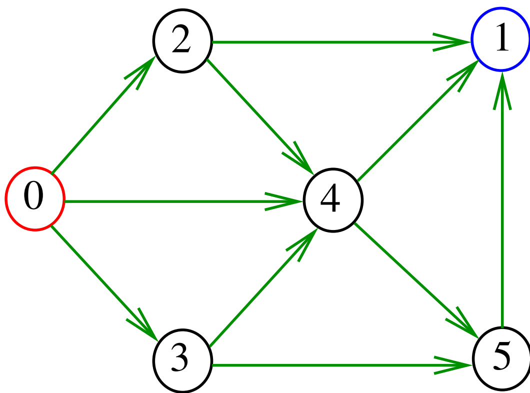
AULA 20

Busca ou varredura

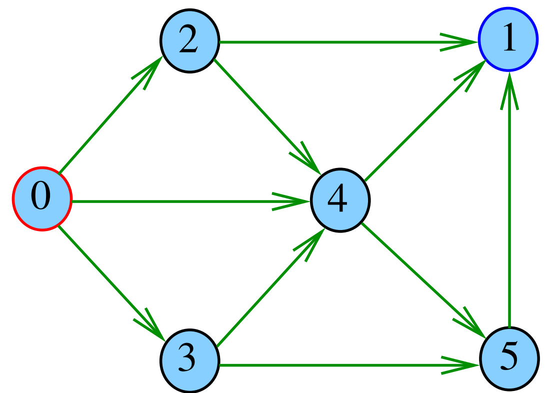
Um algoritimo de **busca** (ou **varredura**) examina, sistematicamente, todos os vértices e todos os arcos de um digrafo.

Cada arco é examinado **uma só vez**.
Despois de visitar sua ponta inicial o algoritmo percorre o arco e visita sua ponta final.

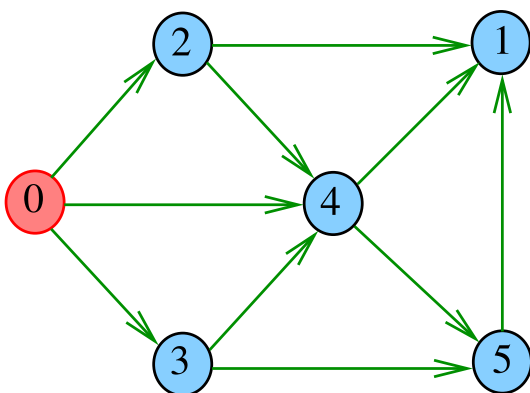
DFSpaths(G, 0)



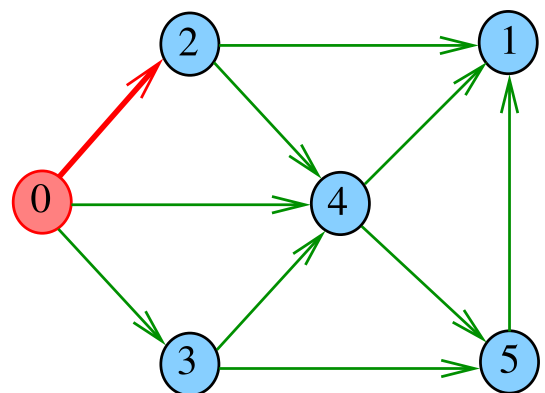
DFSpaths(G, 0)



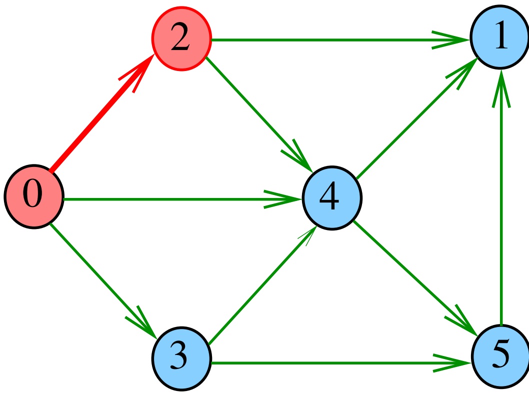
dfs(G, 0)



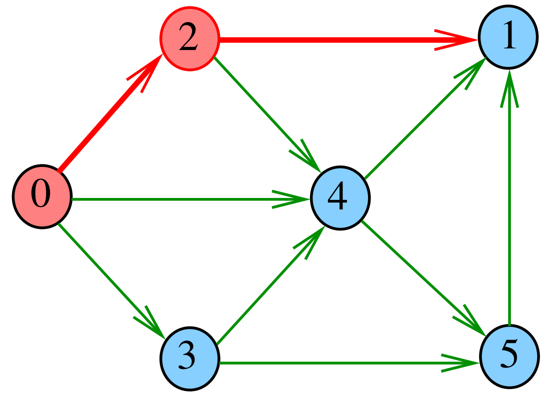
dfs(G, 0)



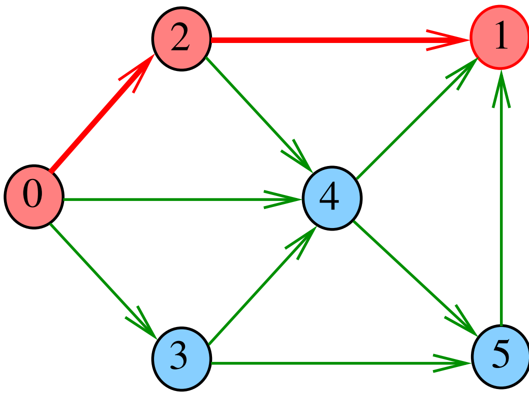
`dfs(G, 2)`



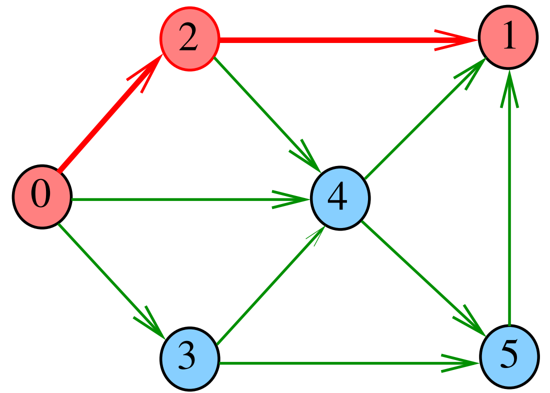
`dfs(G, 2)`



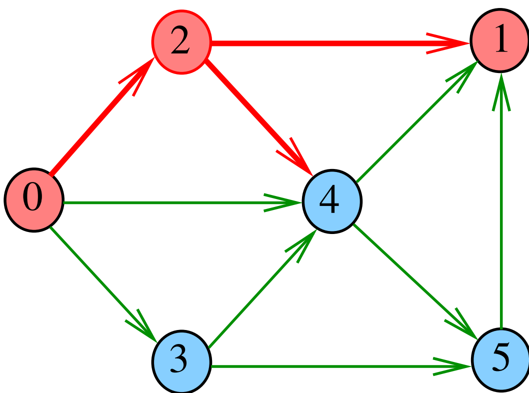
`dfs(G, 1)`



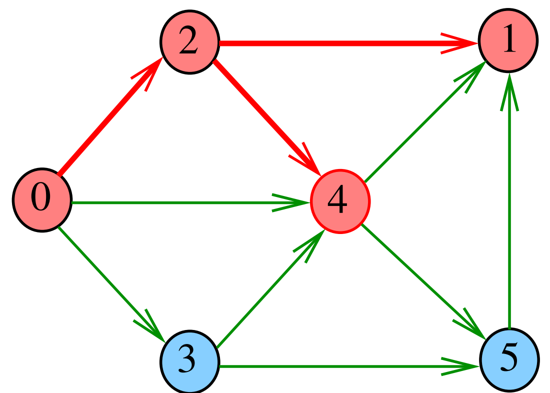
`dfs(G, 2)`



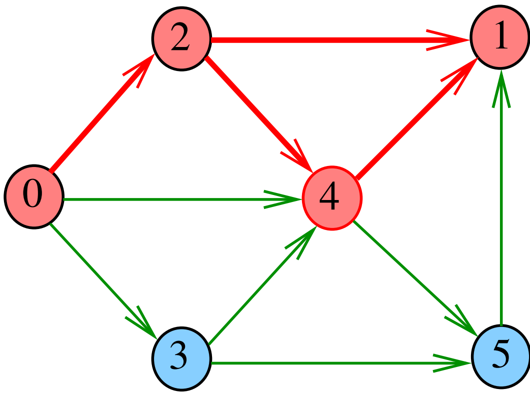
`dfs(G, 2)`



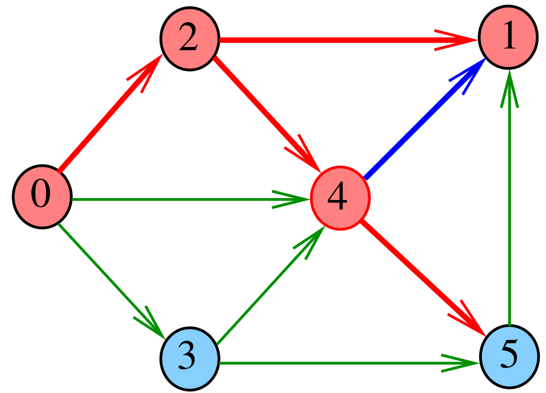
`dfs(G, 4)`



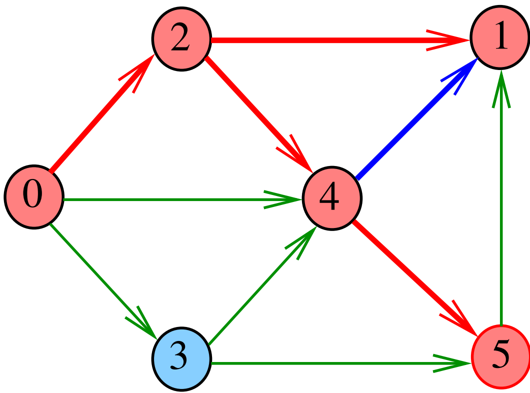
dfs(G, 4)



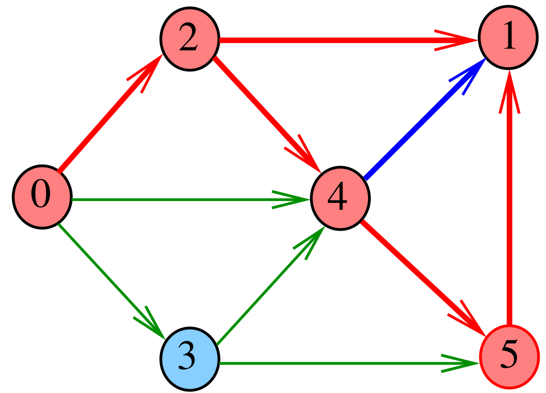
dfs(G, 4)



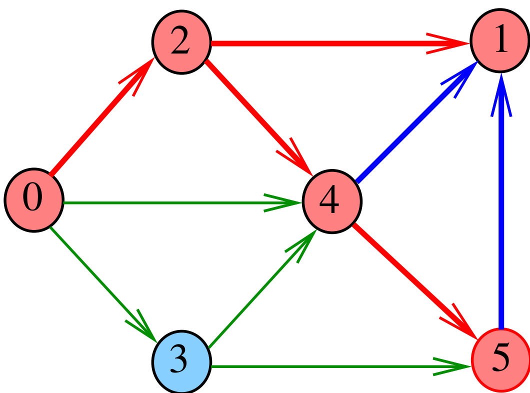
dfs(G, 5)



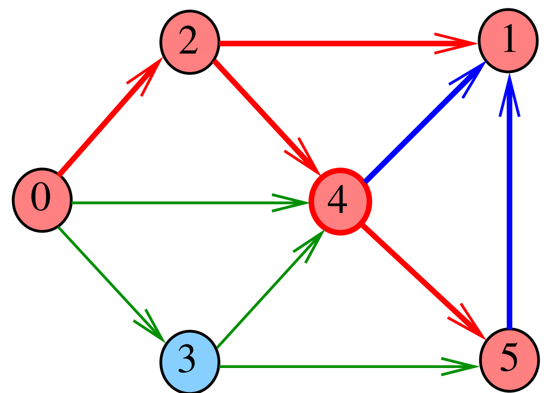
dfs(G, 5)



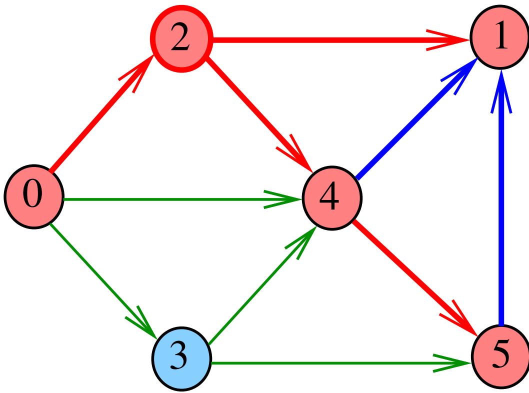
dfs(G, 5)



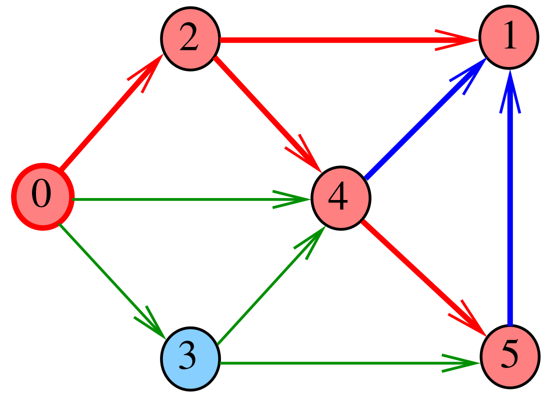
dfs(G, 4)



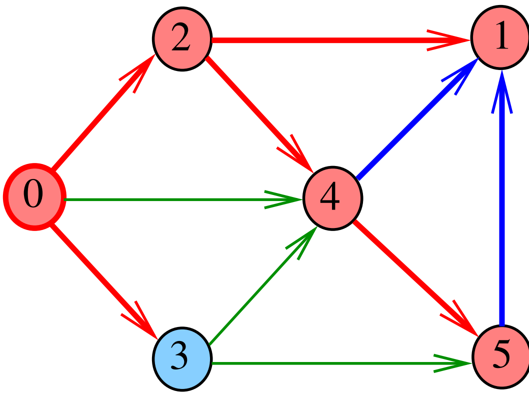
`dfs(G, 2)`



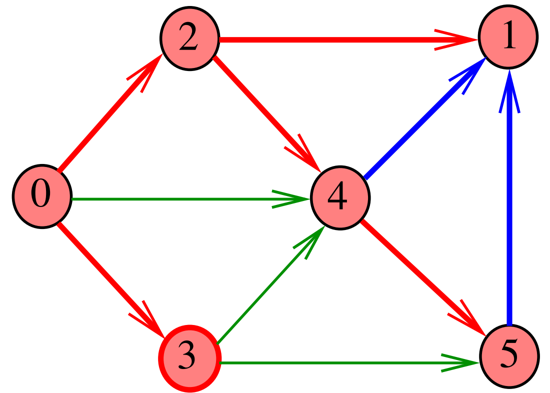
`dfs(G, 0)`



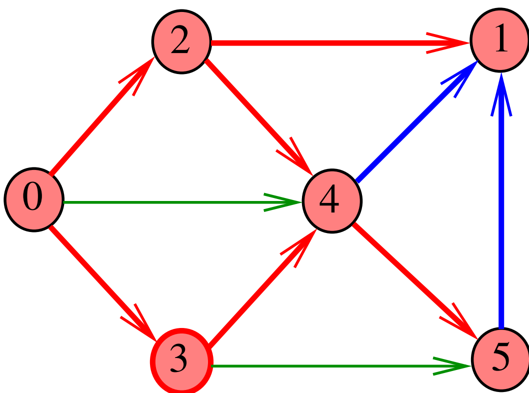
`dfs(G, 0)`



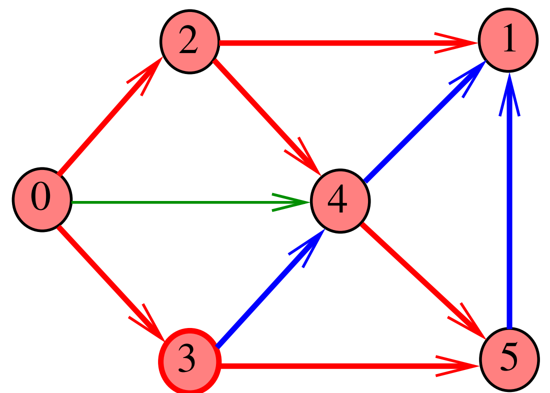
`dfs(G, 3)`



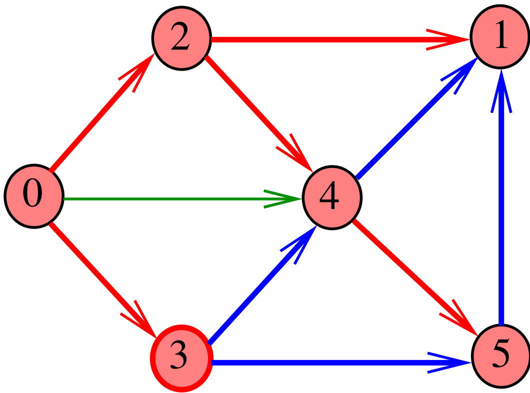
`dfs(G, 3)`



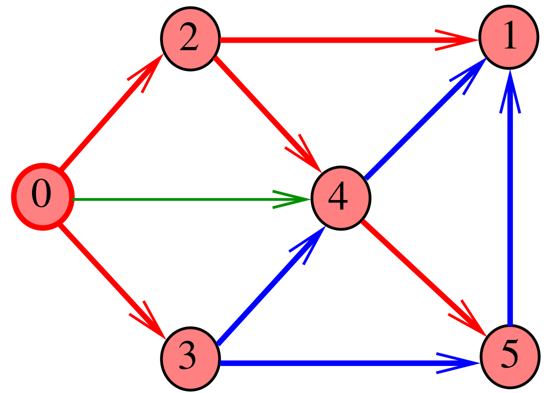
`dfs(G, 3)`



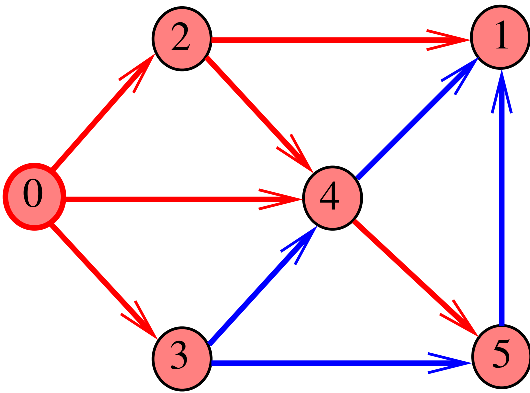
`dfs(G, 3)`



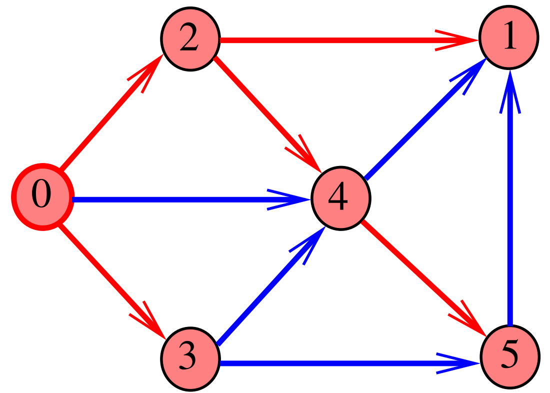
`dfs(G, 0)`



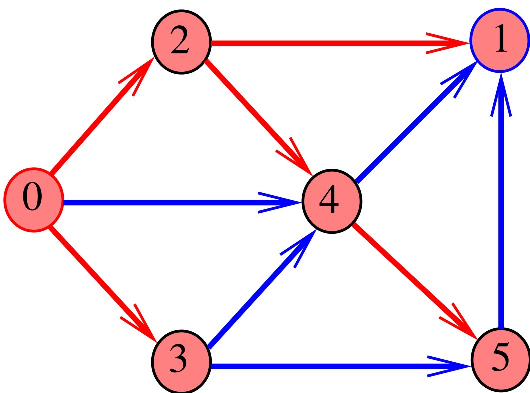
`dfs(G, 0)`



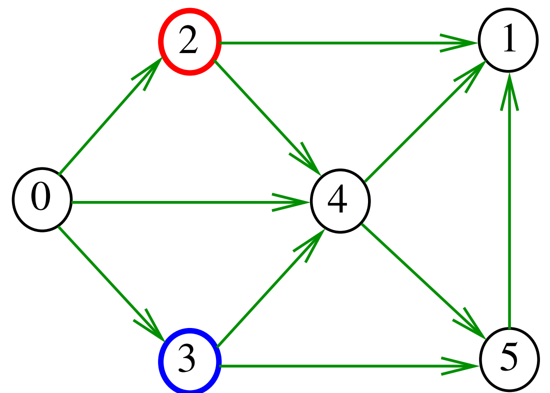
`dfs(G, 0)`



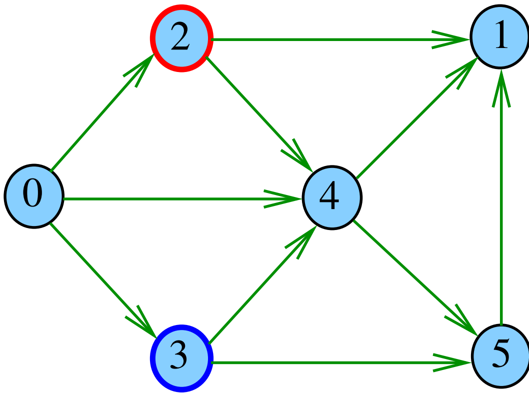
`DFSpaths(G, 0)`



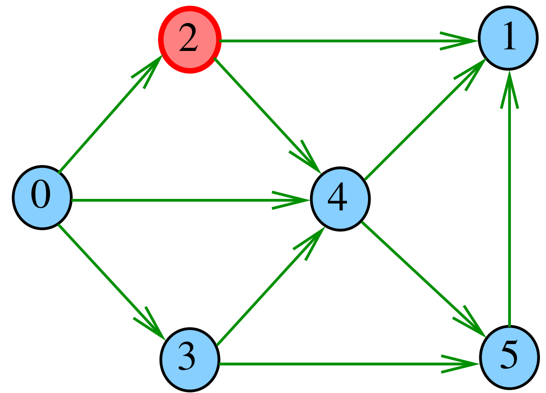
`DFSpaths(G, 2)`



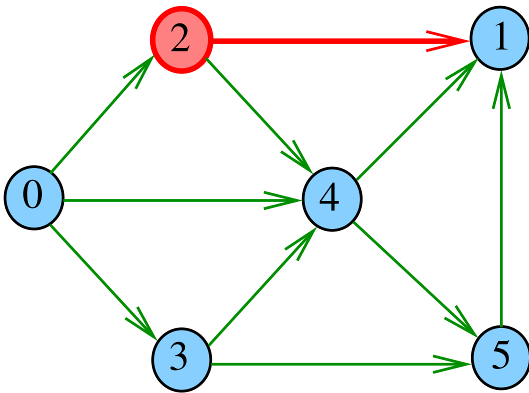
DFSpaths(G, 2)



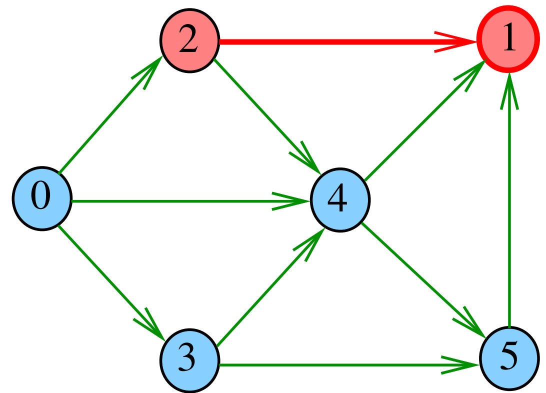
dfs(G, 2)



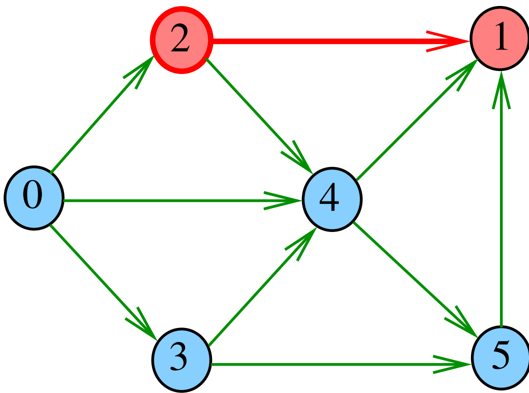
dfs(G, 2)



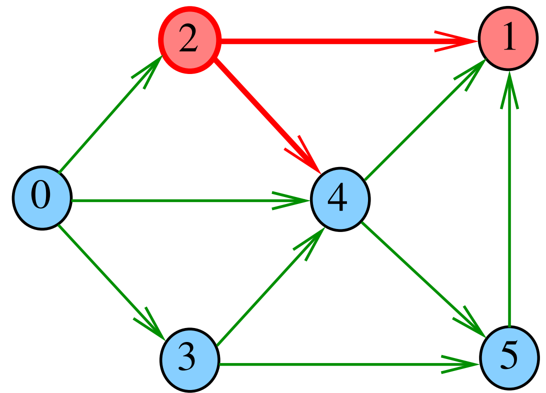
dfs(G, 1)



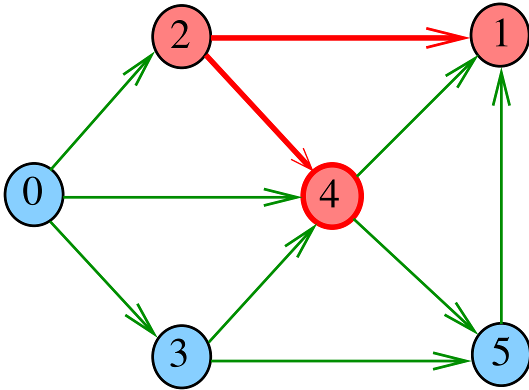
dfs(G, 2)



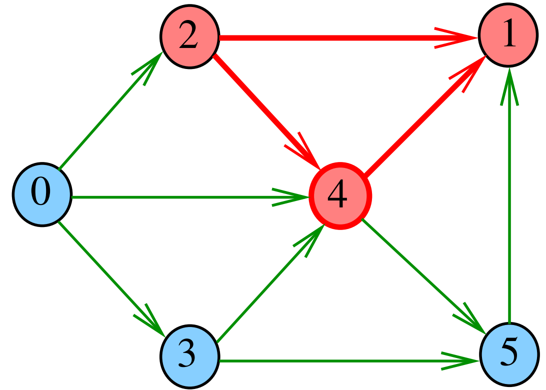
dfs(G, 2)



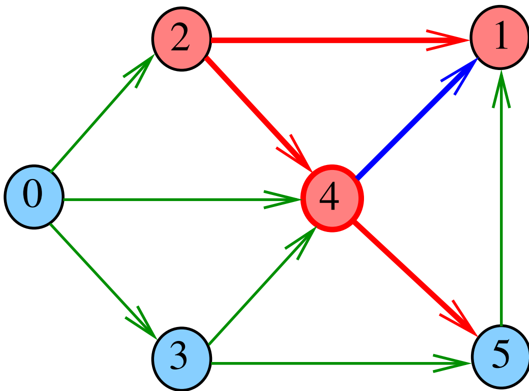
`dfs(G, 4)`



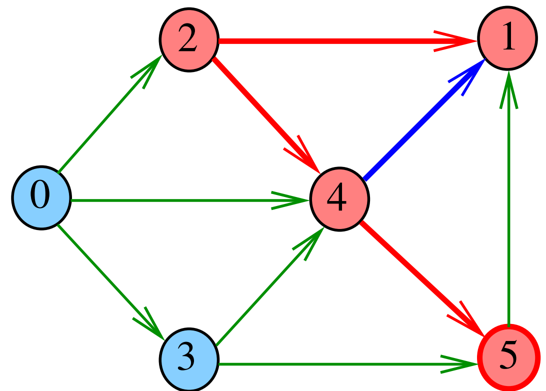
`dfs(G, 4)`



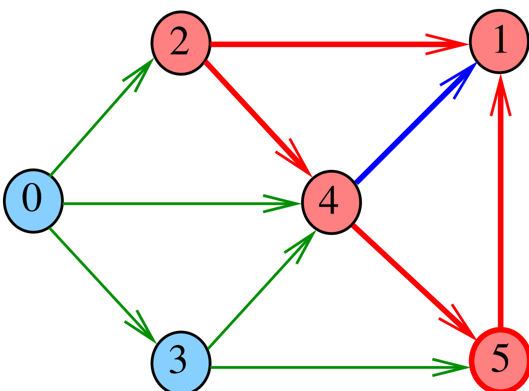
`dfs(G, 4)`



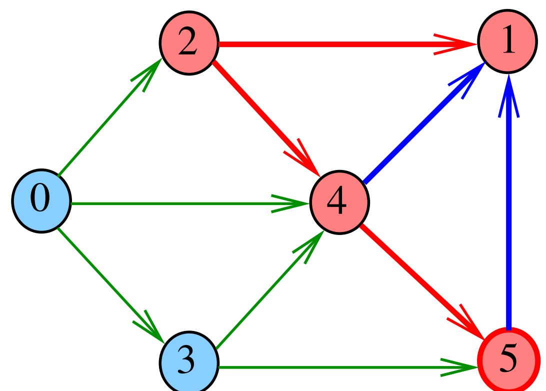
`dfs(G, 5)`



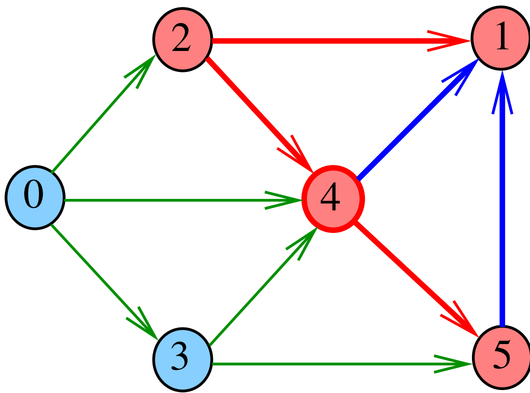
`dfs(G, 5)`



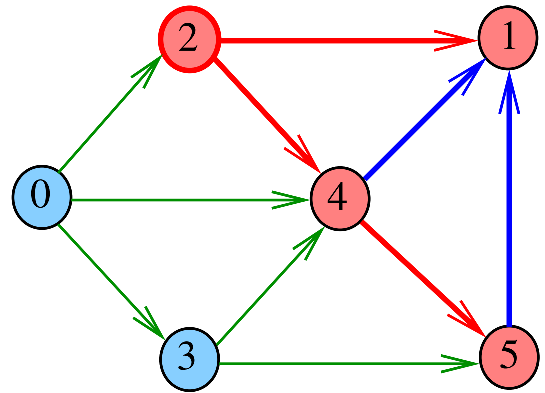
`dfs(G, 5)`



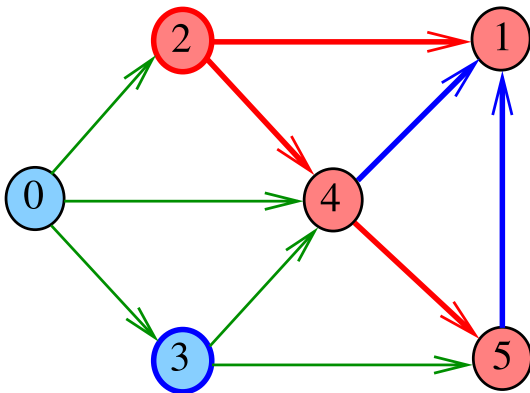
dfs(G, 4)



dfs(G, 2)



DFSpaths(G, 2)



Consumo de tempo

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

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

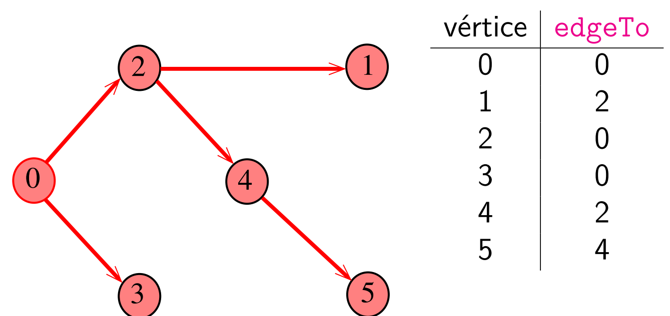
Consumo de tempo

O consumo de tempo da função dfs() para matriz de adjacências é $O(V^2)$.

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

Caminhos no computador

Um arborescência pode ser representada através de um **vetor de pais**: edgeTo[w] é o pai de w
Se r é a raiz, então edgeTo[r]=r



vértice	edgeTo
0	0
1	2
2	0
3	0
4	2
5	4

AULA 21

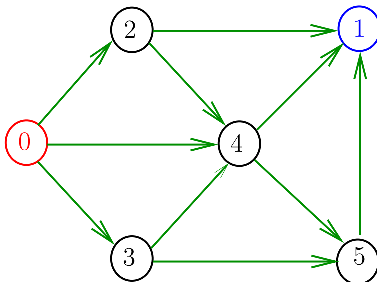
Certificados

Fonte: Free Printable Computer Achievement Award Certificates

Procurando um caminho

Problema: dados um digrafo G e dois vértices s e t decidir se existe um caminho de s a t

Exemplo: para $s = 0$ e $t = 1$ a resposta é SIM



Certificados

Como é possível 'verificar' a resposta?

Como é possível 'verificar' que existe caminho?

Como é possível 'verificar' que não existe caminho?

Certificados

Como é possível 'verificar' a resposta?

Como é possível 'verificar' que existe caminho?

Como é possível 'verificar' que não existe caminho?

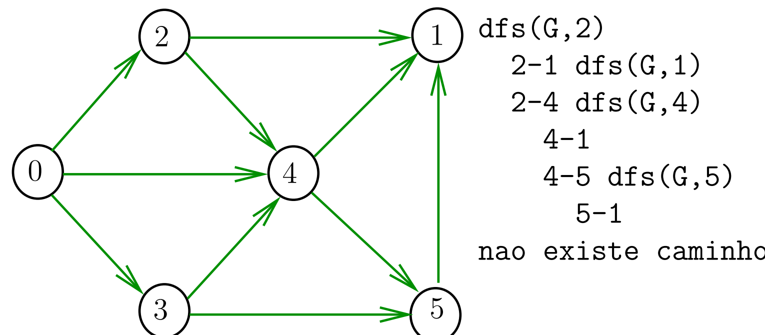
Veremos questões deste tipo freqüentemente

Elas terão um papel **suuupeeer** importante no final de MAC0338 Análise de Algoritmos e em MAC0414 Autômatos, Computabilidade e Complexidade

Elas estão relacionadas com o Teorema da Dualidade visto em MAC0315 Otimização Linear

Certificado de inexistência

Como é possível demonstrar que o problema não tem solução?



DFSpath(G,2,3)

Cortes (= *cuts*)

Um **corte** é uma bipartição do conjunto de vértices
Um arco **pertence** ou **atravessa** um corte (S, T) se tiver uma ponta em S e outra em T

Exemplo 1: arcos em vermelho estão no corte (S, T)

Cortes (= *cuts*)

Um **corte** é uma bipartição do conjunto de vértices
Um arco **pertence** ou **atravessa** um corte (S, T) se tiver uma ponta em S e outra em T

Exemplo 2: arcos em vermelho estão no corte (S, T)

st-Cortes (= st-*cuts*)

Um corte (S, T) é um **st-corte** se

> *s está em S e t está em T*

Exemplo: (S, T) é um 1-3-corte um 2-5-corte ...

Certificado de inexistência

Para demonstrarmos que **não existe** um caminho de s a t basta exibirmos um st-corte (S, T) em que

> ***todo arco*** *no corte tem ponta inicial em T e ponta final em S*

Certificado de inexistência

Exemplo: certificado de que não há caminho de 2 a 3

Conclusão

Para quaisquer vértices s e t de um digrafo, vale uma e apenas umas das seguintes afirmações:

- existe um caminho de s a t
- existe st-corte (S, T) em que todo arco no corte tem ponta inicial em T e ponta final em S.

Fonte: Yin and yang (Wikipedia)

E DFSpaths e BFSpaths com isso?

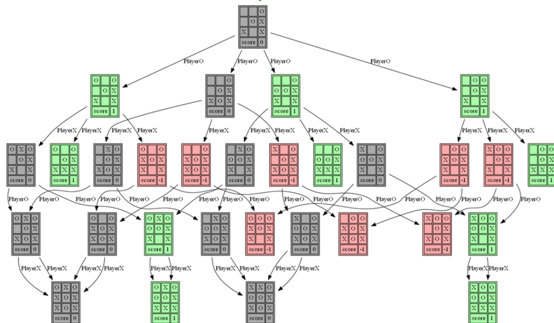
No código das classes DFSpaths e BFSpaths se existe um caminho de s a t ele esta representado no vetor edgeTo[].

No código da classes DFSpaths e BFSpaths se não existe um caminho de s a t um st-corte separando s de t está representado no vetor marked[].

Em ambos os casos podemos fazer um trecho de código que verifica a resposta em tempo proporcional a V + E.

Anatomia de busca em profundidade

Classificação dos arcos



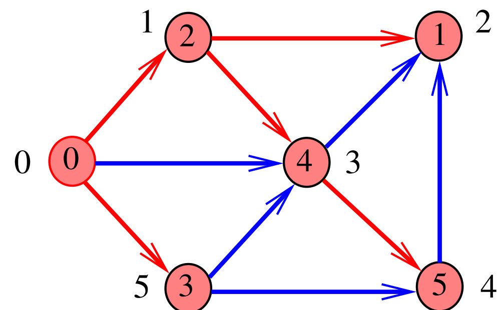
Fonte: Using Minimax (with the full game tree) to implement the machine players ...

Referências: CLRS 22

Arcos da arborescência

Arcos da arborescência são os arcos v-w que dfs() percorre para visitar w pela primeira vez

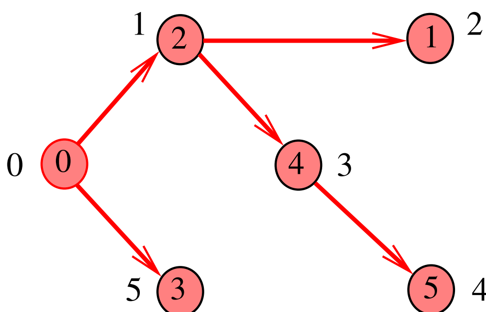
Exemplo: arcos em vermelho são arcos da arborescência



Arcos da arborescência

Arcos da arborescência são os arcos v-w que dfs() percorre para visitar w pela primeira vez

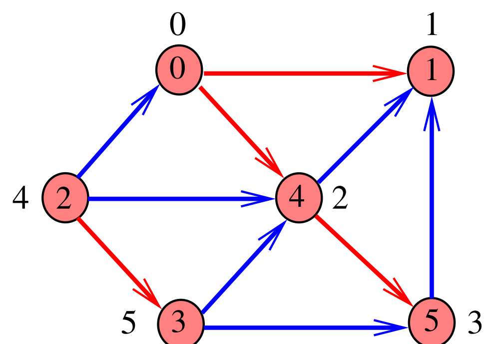
Exemplo: arcos em vermelho são arcos da arborescência



Floresta DFS

Conjunto de arborescências é a **floresta da busca em profundidade** (= *DFS forest*)

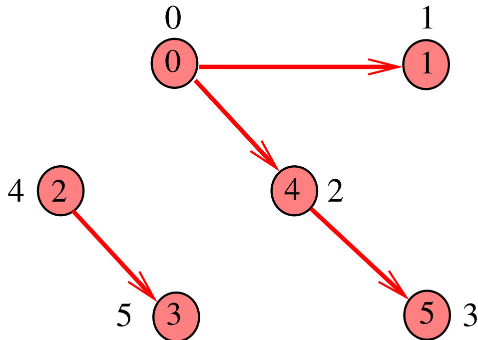
Exemplo: arcos em vermelho formam a floresta DFS



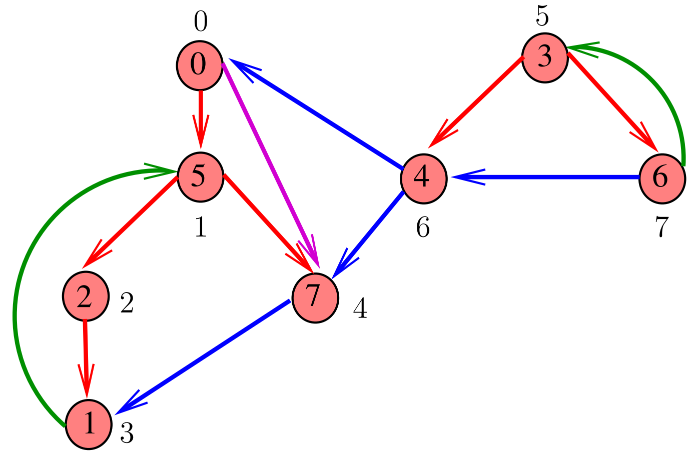
Floresta DFS

Conjunto de arborescências é a **floresta da busca em profundidade** (= *DFS forest*)

Exemplo: arcos em vermelho formam a floresta DFS

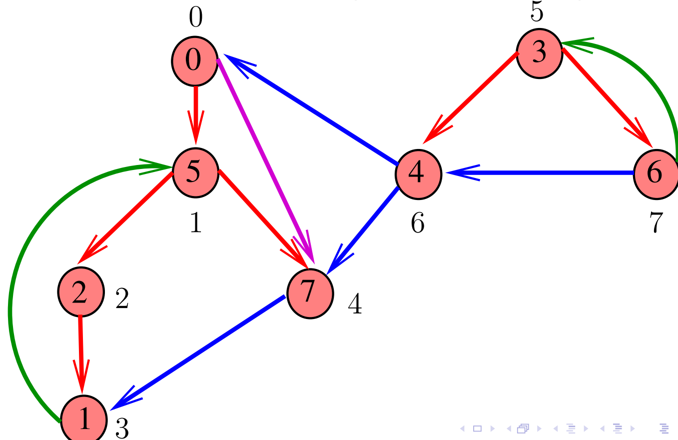


Classificação dos arcos



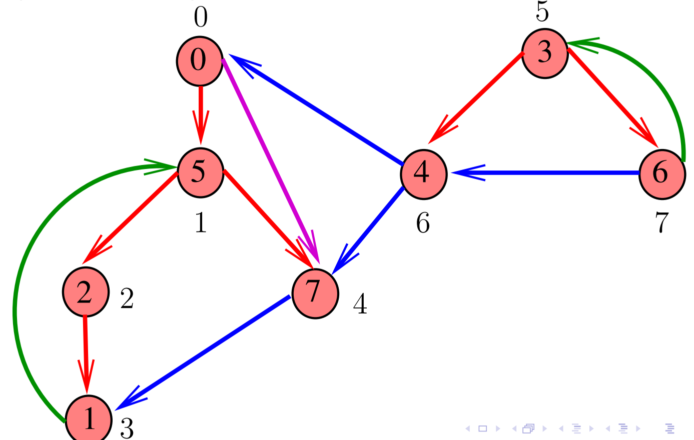
Arcos de arborescência

`v-w` é **arco de arborescência** se foi usado para visitar `w` pela primeira vez (arcos **vermelhos**)



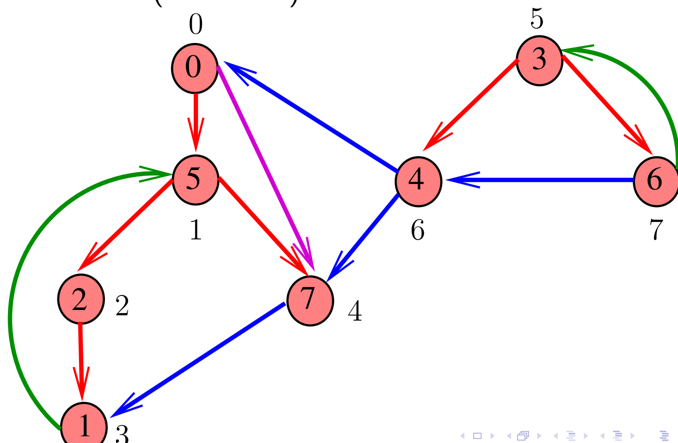
Arcos de retorno

`v-w` é **arco de retorno** se `w` é ancestral de `v` (arcos **verdes**)



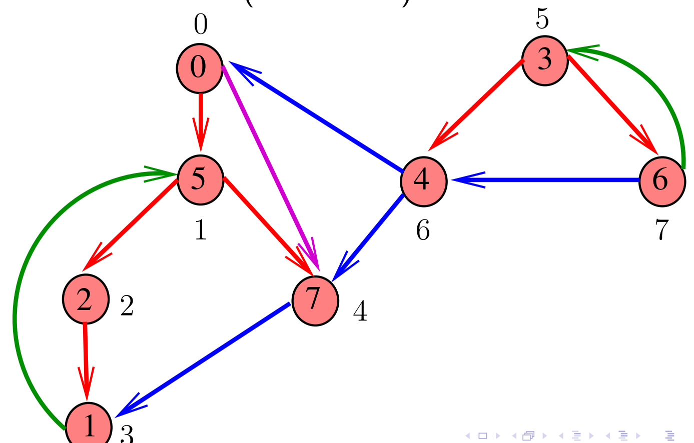
Arcos descendentes

`v-w` é **descendente** se `w` é descendente de `v`, mas não é filho (arco **roxo**)



Arcos cruzados

`v-w` é **arco cruzado** se `w` não é ancestral nem descendente de `v` (arcos **azuis**)



Busca DFSanatomia (CLRS)

O digrafo G têm G.V() vértices

```
private int time;
private int[] d = new int[G.V()];
private int[] f = new int[G.V()];
```

DFSanatomia visita todos os vértices e arcos do digrafo G.

A função registra em d[v] o 'momento' em que v foi descoberto e em f[v] o momento em que ele foi completamente examinado

DFSanatomia: esqueleto

```
public class DFSanatomia {
   private boolean[] marked;
   private int[] edgeTo;
   private int time;
   private int[] d; // discovered
   private int[] f; // finished
   private Queue<Integer> pre;// pré-ordem
   private Queue<Integer> pos;// pós-ordem
   // pós-ordem reversa
   private Stack<Integer> revPos;
```

DFSanatomia: esqueleto

```
   // métodos
   public DFSanatomia(Graph G) {...}
   private void dfs(Digraph G, int v){...}
   public Iterable<Integer> pre() {...}
   public Iterable<Integer> pos() {...}
   public Iterable<Integer> revPos() {...}
}
```

DFSanatomia: construtor

```
public DFSanatomia(Graph G) {
   marked = new boolean[G.V()];
   edgeTo = new int[G.V()];
   d = new int[G.V()];
   f = new int[G.V()];
   pre = new Queue<Integer>();
   pos = new Queue<Integer>();
   revPos = new Stack<Integer>();
   for (int v = 0; v < G.V(); v++)
      if (!marked(v)) {
         dfs(G,v);
      }
}
```

DFSanatomia: dfs()

```
private void dfs(Digraph G, int v) {
   marked[v] = true;
   d[v] = time++; // descoberto
   pre.enqueue(v); // pré-ordem
   for (int w : G.adj[v]) {
      if (!marked(w)) {
         edgeTo[w] = v;
         dfs(G, w);
      }
   }
   pos.enqueue(v); // pós-ordem
   revPos.push(v); // pós-ordem reversa
   f[v] = time++; // terminamos
}
```

DFSanatomia: pre(), pos() e revPos()

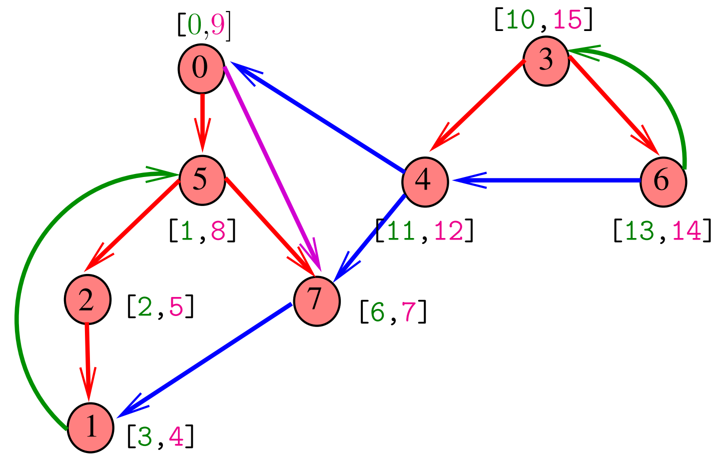
```
public Iterable<Integer> pre() {
   return pre;
}
public Iterable<Integer> pos() {
   return pos;
}
public Iterable<Integer> revPos() {
   return revPos;
}
```

Consumo de tempo

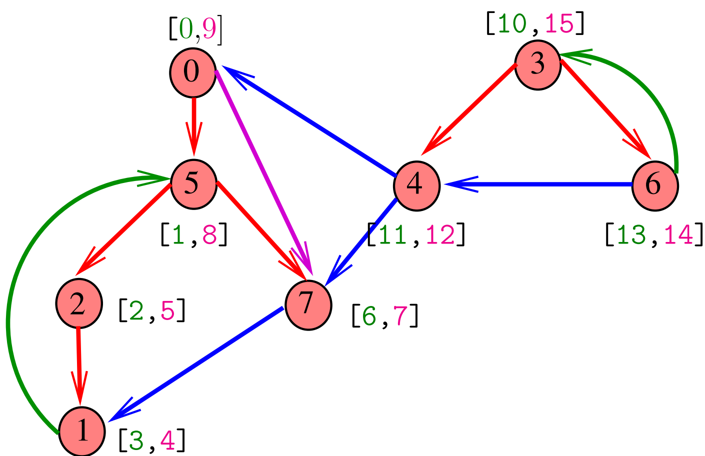
A classe DFSanatomia, para vetor de listas de adjacência, consome tempo $O(V + E)$.

A classe DFSanatomia, para matriz de adjacências, consome tempo $O(V^2)$.

Busca DFS (CLRS)

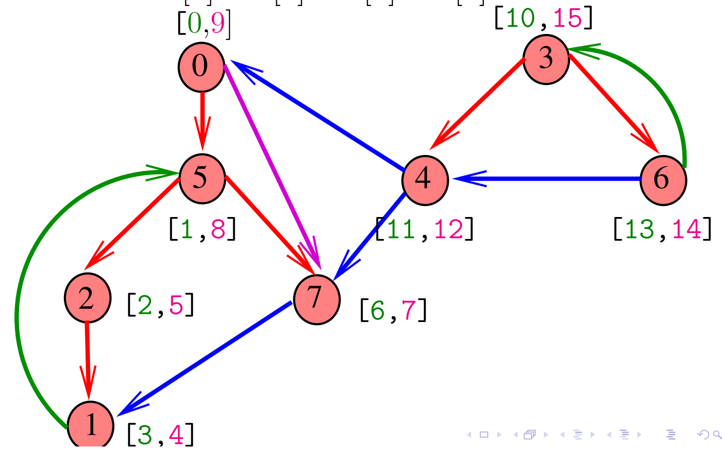


Classificação dos arcos



Arcos de arborescência ou descendentes

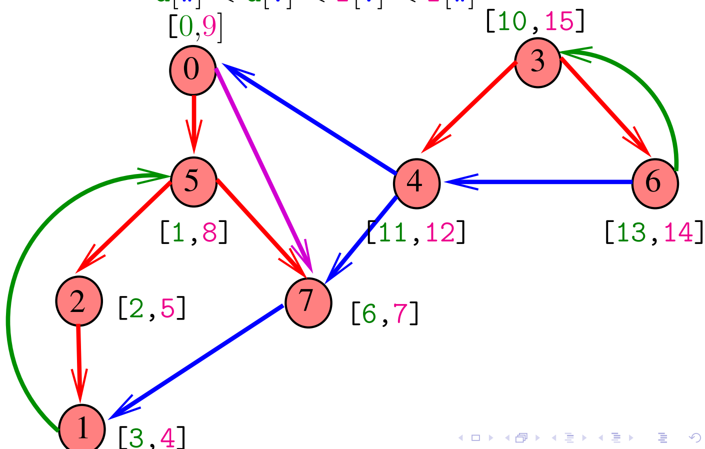
v-w é **arco de arborescência** ou **descendente** se e somente se $d[v] < d[w] < f[w] < f[v]$



Arcos de retorno

v-w é **arco de retorno** se e somente se

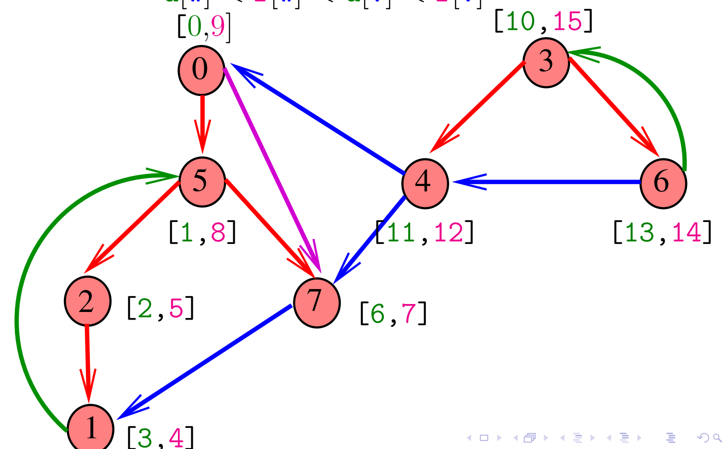
$$d[w] < d[v] < f[v] < f[w]$$



Arcos cruzados

v-w é arco **cruzado** se e somente se

$$d[w] < f[w] < d[v] < f[v]$$

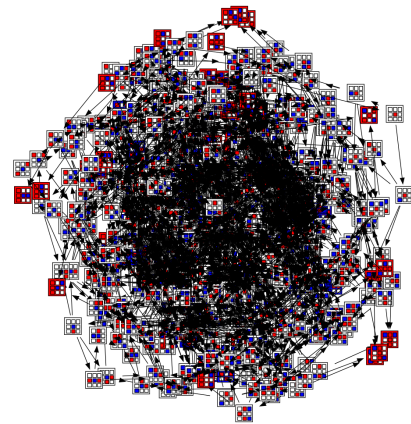


Conclusões

v-w é:

- **arco de arborescência** se e somente se $d[v] < d[w] < f[w] < f[v]$ e $edgeTo[w] = v$;
- **arco descendente** se e somente se $d[v] < d[w] < f[w] < f[v]$ e $edgeTo[w] \neq v$;
- **arco de retorno** se e somente se $d[w] < d[v] < f[v] < f[w]$;
- **arco cruzado** se e somente se $d[w] < f[w] < d[v] < f[v]$;

Ciclos em digrafos

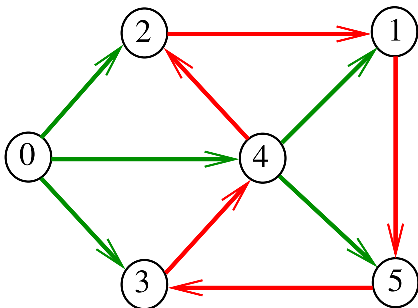


Fonte: laying out a large graph with graphviz

Ciclos

Um **ciclo** num digrafo é qualquer sequência da forma v_0–v_1–v_2–...–v_{k-1}–v_p, onde v_{k-1}-v_k é um arco para $k = 1, \ldots, p$ e $v_0 = v_p$.

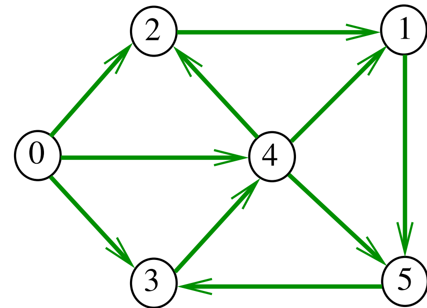
Exemplo: 2-1-5-3-4-2 é um ciclo



Procurando um ciclo

Problema: decidir se dado digrafo G possui um ciclo

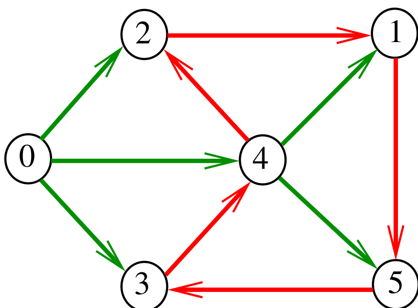
Exemplo: para o grafo a seguir a resposta é **SIM**



Procurando um ciclo

Problema: decidir se dado digrafo G possui um ciclo

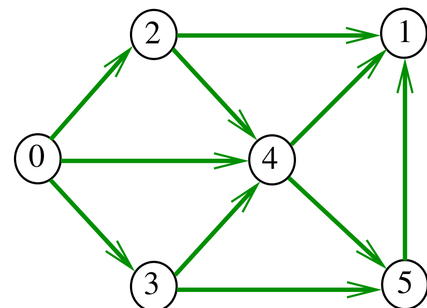
Exemplo: para o grafo a seguir a resposta é **SIM**



Procurando um ciclo

Problema: decidir se dado digrafo G possui um ciclo

Exemplo: para o grafo a seguir a resposta é **NÃO**



`DirectedCycle` café com leite

Recebe um digrafo `G` e decide se existe um ciclo.

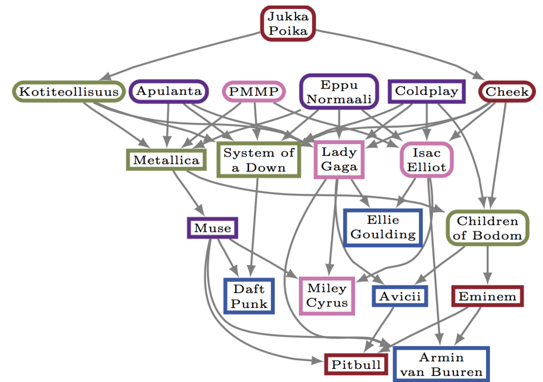
Para cada arco `u-v` podemos fazer

```
DFSpaths dfs = new DFSpaths(G, v);
```

e verificar se `dfs.hasPath(u)`

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

Digrafos acíclicos (DAGs)

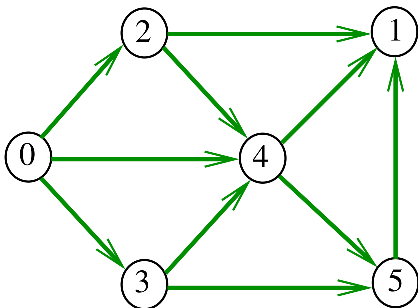


Fonte: `Comparing directed acyclic graphs`

DAGs

Um digrafo é **acíclico** se não tem ciclos
Digrafos acíclicos também são conhecidos como DAGs (= *directed acyclic graphs*)

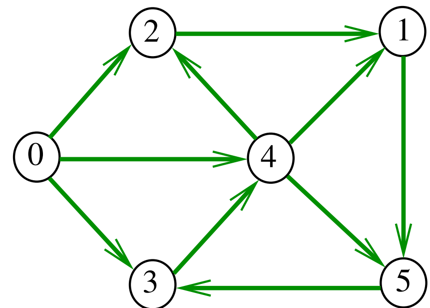
Exemplo: um digrafo acíclico



DAGs

Um digrafo é **acíclico** se não tem ciclos
Digrafos acíclicos também são conhecidos como DAGs (= *directed acyclic graphs*)

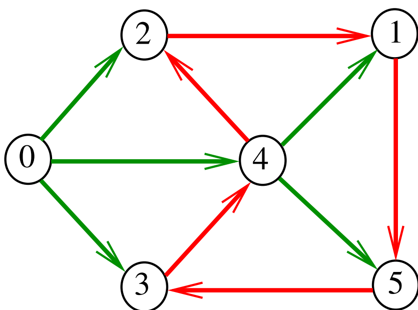
Exemplo: um digrafo que **não** é acíclico



DAGs

Um digrafo é **acíclico** se não tem ciclos
Digrafos acíclicos também são conhecidos como DAGs (= *directed acyclic graphs*)

Exemplo: um digrafo que **não** é acíclico



Ordenaçao topológica

Uma **permutação** dos vértices de um digrafo é uma seqüência em que cada vértice aparece uma e uma só vez

Uma **ordenação topológica** (= *topological sorting*) de um digrafo é uma permutação

`ts[0], ts[1],..., ts[V-1]`

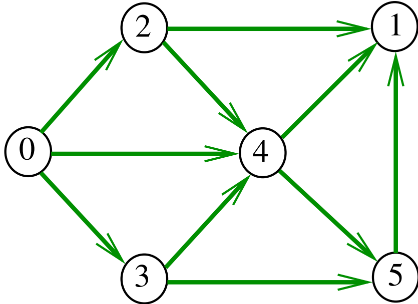
dos seus vértices tal que todo arco tem a forma

`ts[i]-ts[j]` com `i < j`

`ts[0]` é necessariamente uma **fonte**
`ts[V-1]` é necessariamente um **sorvedouro**

Exemplo

i	0	1	2	3	4	5
ts[i]	0	3	2	4	5	1



DAGs versus ordenação topológica

É evidente que digrafos com ciclos **não** admitem ordenação topológica.

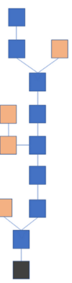
É menos evidente que **todo** DAG admite ordenação topológica.

A prova desse fato é um algoritmo que recebe qualquer digrafo e devolve

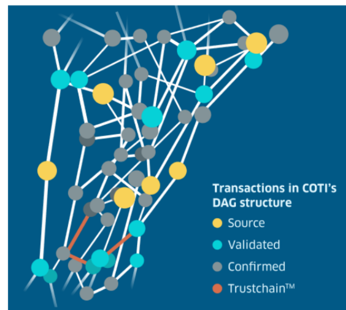
- um ciclo;
- uma ordenação topológica do digrafo.

Algoritmos de ordenação topológica

Blockchain vs Directed Acyclic Graph (DAG)



Blockchain



COTI's DAG

Fonte: Our Exciting Partnership With COTI

Algoritmo de eliminação de fontes

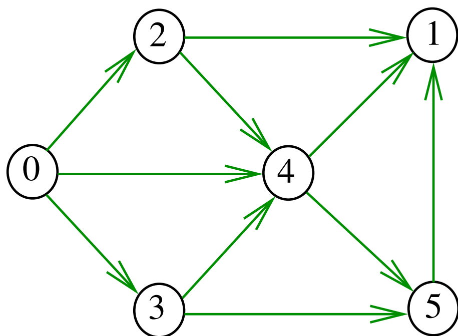
Armazena em `ts[0 . . i-1]` uma permutação de um subconjunto do conjunto de vértices de `G` e devolve o valor de `i`

Se `i` $=$ `G.V()` então `ts[0 . . i-1]` é uma ordenação topológica de `G`.
Caso contrário, `G` **não** é um DAG

```
int DAGts1 (Digraph G, Vertex ts[]);
```

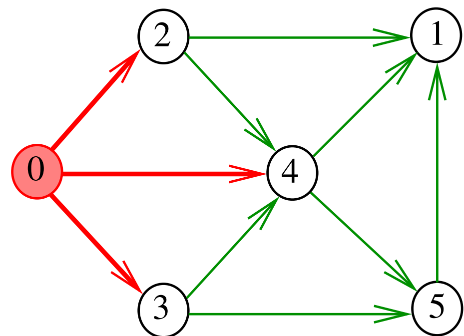
Exemplo

i	0	1	2	3	4	5
ts[i]						



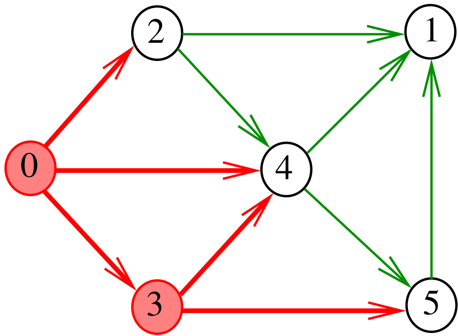
Exemplo

i	0	1	2	3	4	5
ts[i]	0					



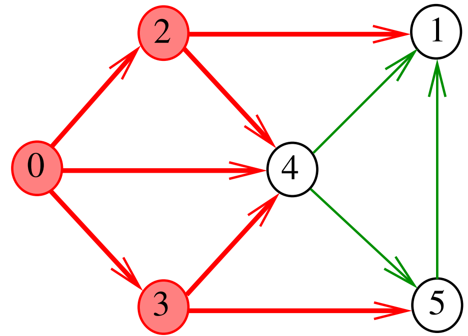
Exemplo

i	0	1	2	3	4	5
ts[i]	0	3				



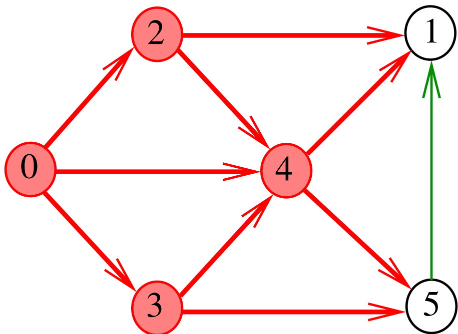
Exemplo

i	0	1	2	3	4	5
ts[i]	0	3	2			



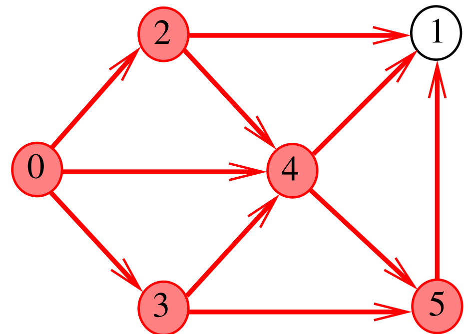
Exemplo

i	0	1	2	3	4	5
ts[i]	0	3	2	4		



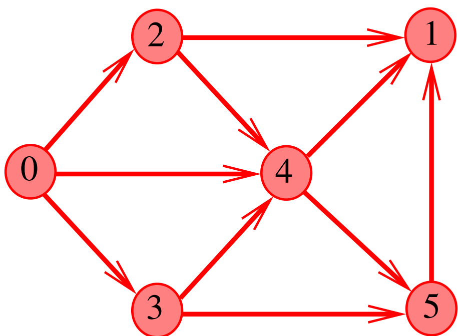
Exemplo

i	0	1	2	3	4	5
ts[i]	0	3	2	4	5	



Exemplo

i	0	1	2	3	4	5
ts[i]	0	3	2	4	5	1



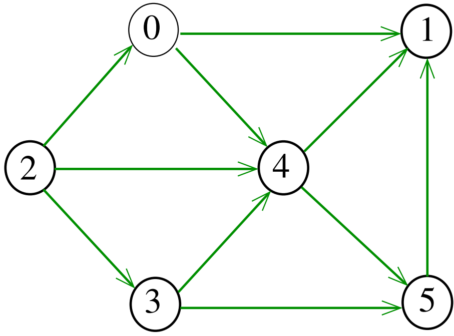
Consumo de tempo

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

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

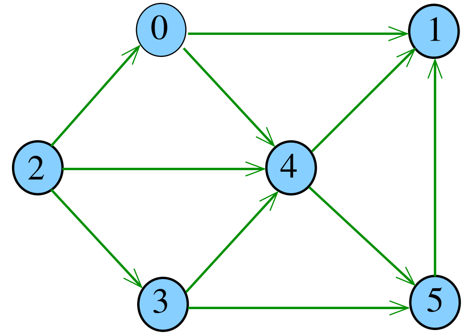
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]						



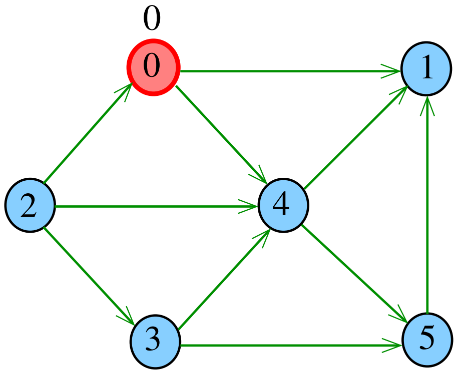
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]						



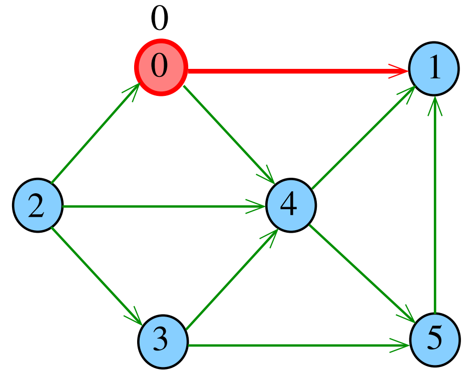
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]						



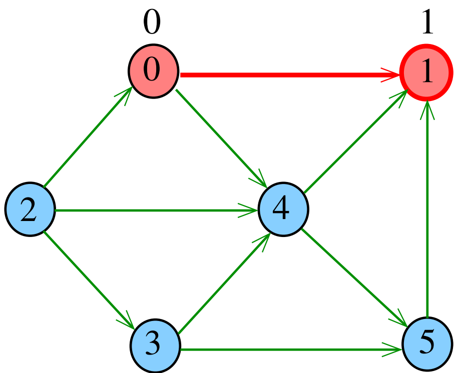
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]						



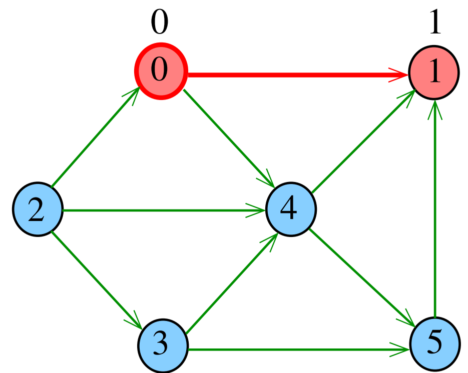
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]						



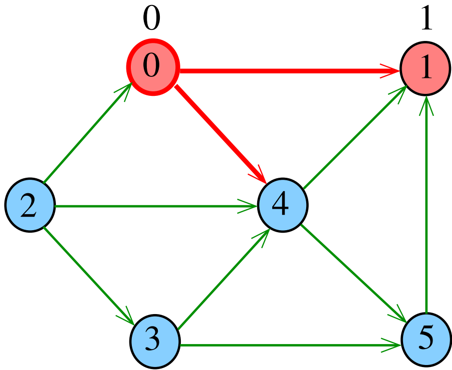
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]						1



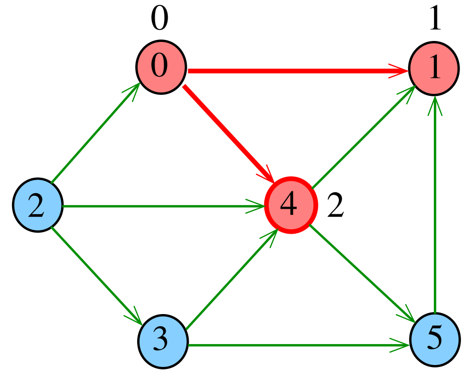
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]						1



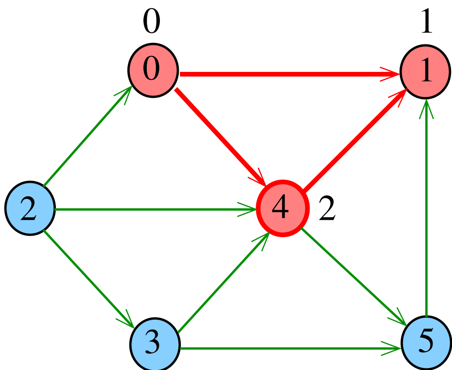
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]						1



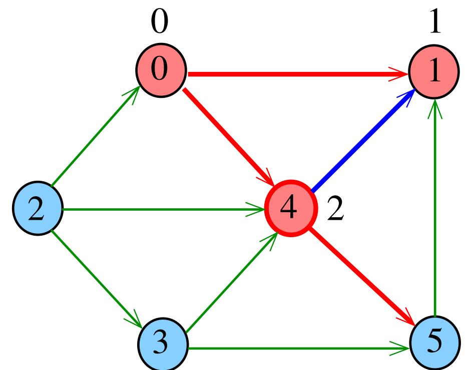
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]						1



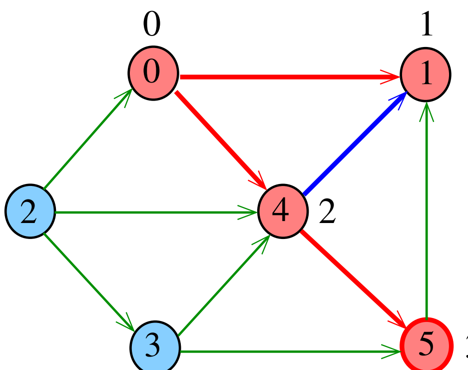
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]						1



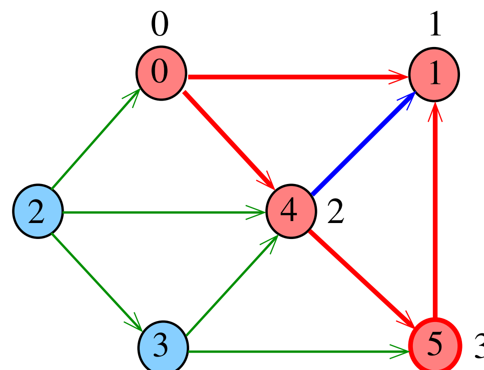
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]						1



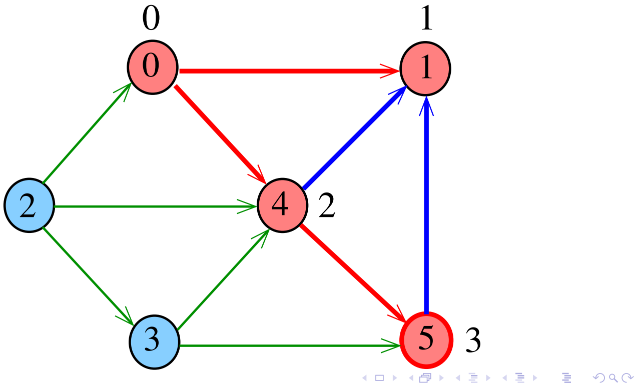
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]						1



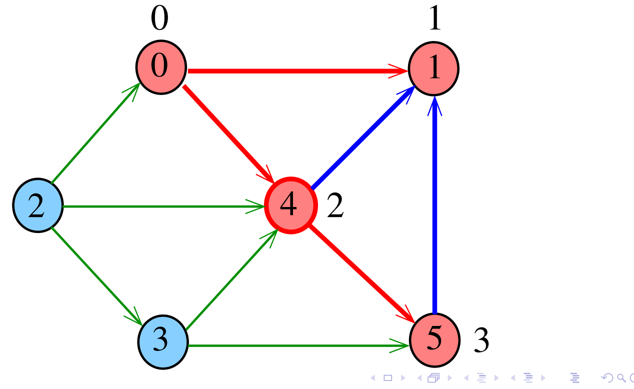
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]						1



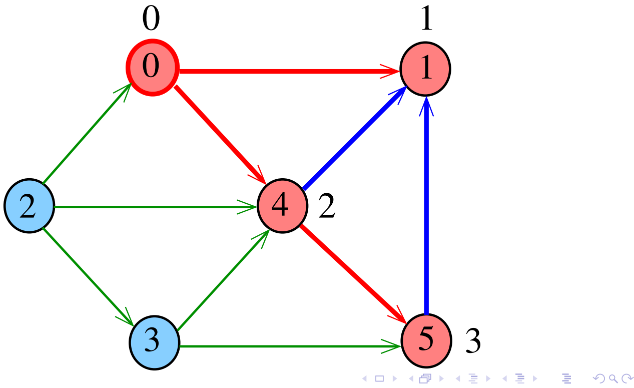
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]					5	1



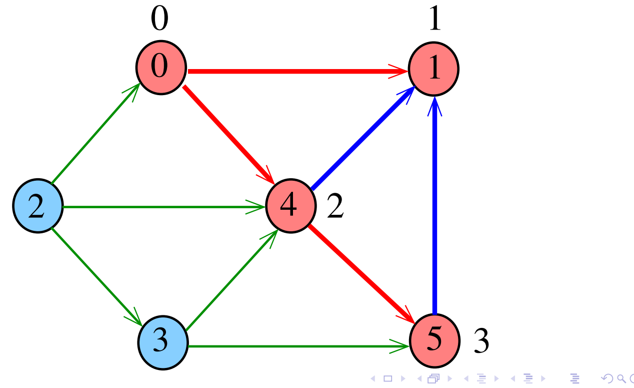
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]				4	5	1



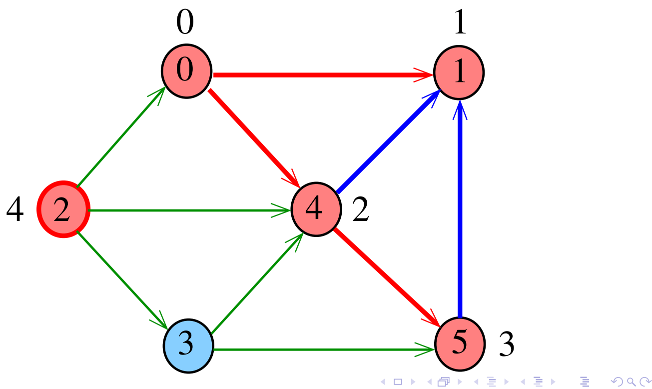
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]			0	4	5	1



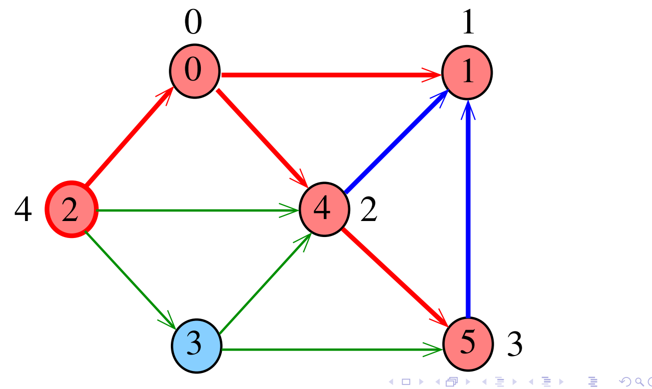
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]			0	4	5	1



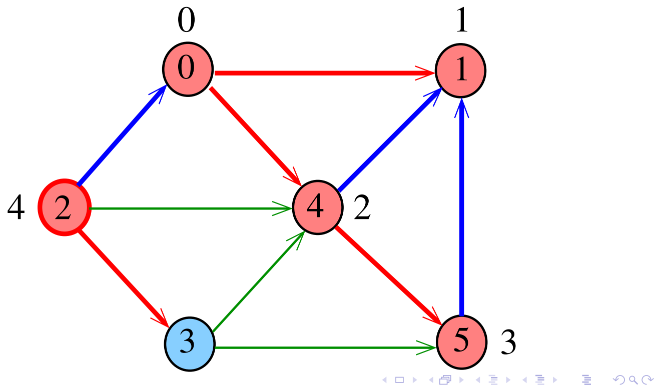
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]			0	4	5	1



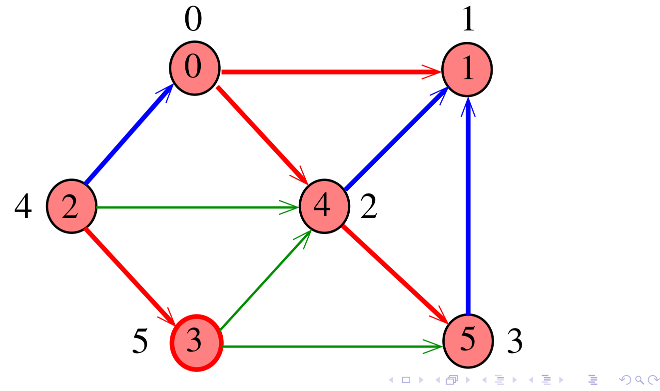
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]			0	4	5	1



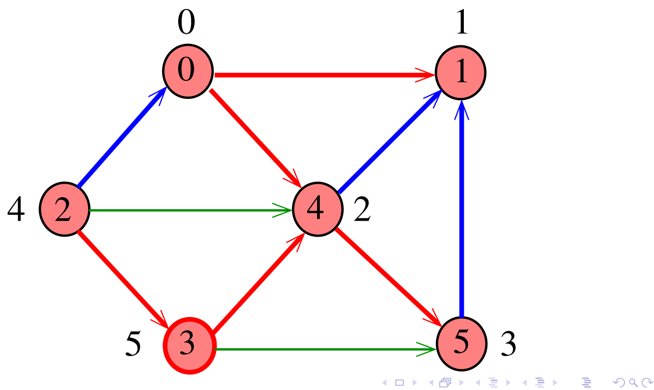
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]			0	4	5	1



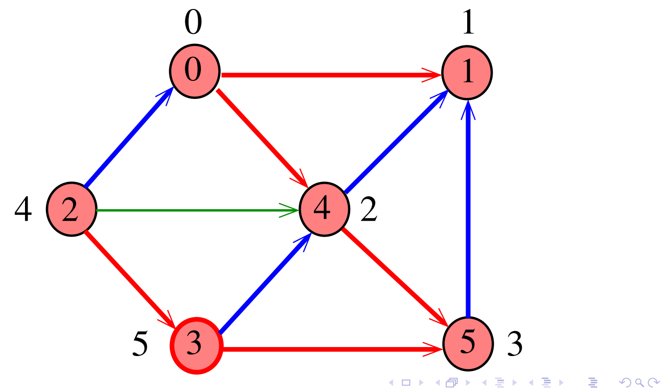
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]			0	4	5	1



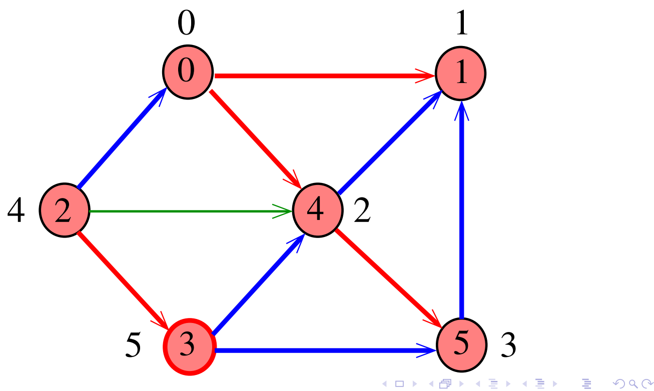
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]			0	4	5	1



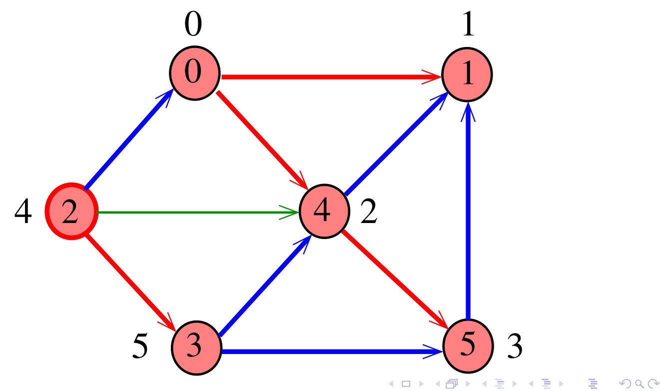
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]			0	4	5	1



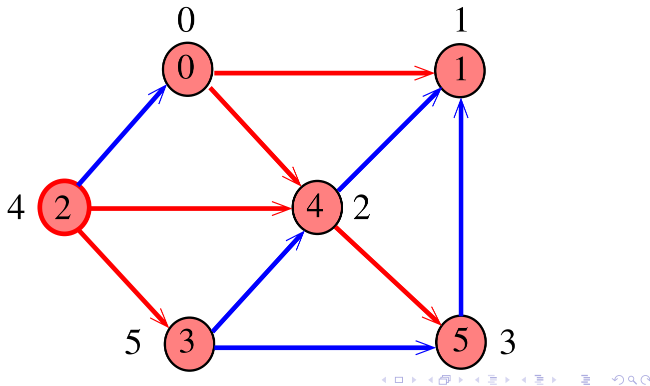
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]		3	0	4	5	1



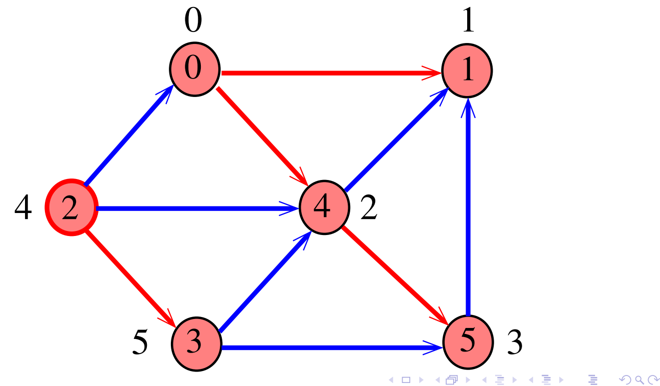
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]		3	0	4	5	1



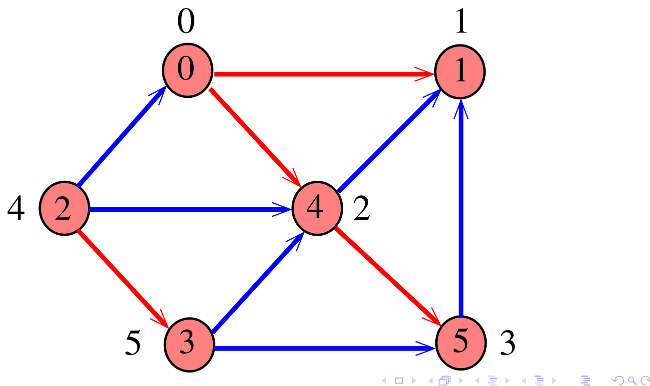
Algoritmo DFS

i	0	1	2	3	4	5
ts[i]		3	0	4	5	1



Algoritmo DFS

i	0	1	2	3	4	5
ts[i]	2	3	0	4	5	1



Classe DFStopological

A classe DFStopological decide se um dado digrafo G é um DAG.

```
private Stack<Integer> ts
private Stack<Integer> cycle;
private boolean[] onPath;
```

Se G é um DAG uma ordenação topológica de seus vértices é armazenada em ts.

Se G não é um DAG, cycle armazenará um ciclo de G.

onPath[v] é true se o vértice v está no *caminho ativo*.

DFStopological: esqueleto

```
public class DFStopological {
  private boolean[] marked;
  private int[] edgeTo;
  private boolean[] onPath;
  private Stack<Integer> ts;
  private Stack<Integer> cycle;
  private int onCycle= -1;
  public DFStopological(Digraph G) {...}
  private void dfs(Digraph G, int v){...}
  public boolean isDag() {...}
  public boolean hasCycle() {...}
  public Iterable<Integer> order() {...}
  public Iterable<Integer> cycle() {...}
}
```

DFStopological: construtor

Determina se um digrafo G é acíclico, e portanto seu vértices tem uma ordem topológica, ou tem um ciclo.

```
public DFStopological(Digraph G) {
  marked = new boolean[G.V()];
  edgeTo = new int[G.V()];
  ts = new Stack<Integer>();
  onPath = new boolean[G.V()];
  for (int v = 0; v < G.V(); v++)
     if (!marked[v] && onCycle == -1) {
        dfs(G,v);
     }
}
```

DFStopological: dfs()

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

DFStopological: hasCycle(), isDag()

```
// G contém um ciclo ?
public boolean hasCycle() {
  return onCycle != -1;
}

// G é um DAG ?
public boolean isDag() {
  return onCycle == -1;
}
```

DFStopological: cycle()

Retorna um ciclo como iterável se G possui um ciclo ou null em caso contrário.

```
public Iterable<Integer> cycle() {
  if (!hasCycle()) 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;
}
```

DFStopological: order()

Retorna uma ordem topológica dos vértices de G como iterável, se G é um DAG.

```
public Iterable<Integer> order() {
  if (!isDag()) return null;
  return ts;
}
```

Consumo de tempo

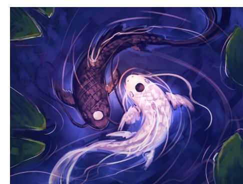
O consumo de tempo da função DFStopological para vetor de listas de adjacência é $O(V + E)$.

A classe DFStopological, para matriz de adjacências, consome tempo $O(V^2)$.

Conclusão

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

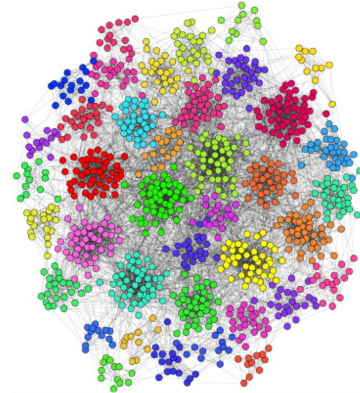
- G possui um ciclo
- G é um DAG e, portanto, admite uma ordenação topológica



Fonte: Avatar: The Last Airbender

Fonte: Relationship of Time-Activity-Adjusted Particle Number Concentration with Blood Pressure

Componentes de grafos

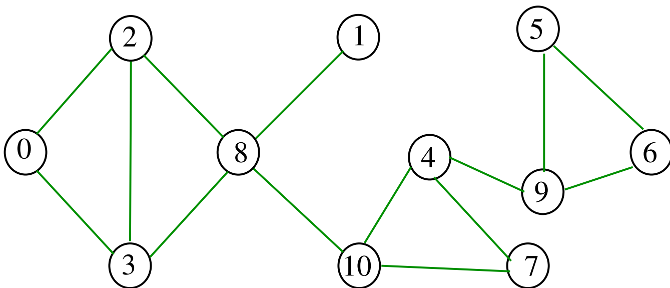


Fonte: `Personalized PageRank Clustering: A graph clustering algorithm based on random walks`

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`

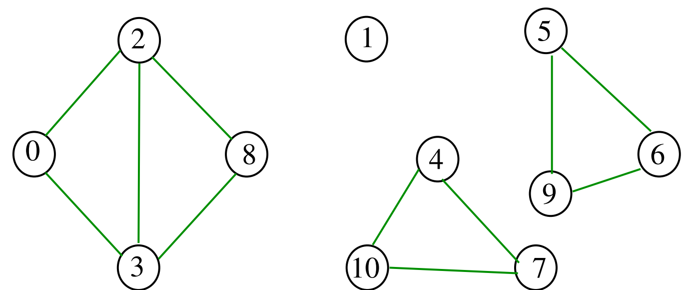
Exemplo: um grafo conexo



Componentes de grafos

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

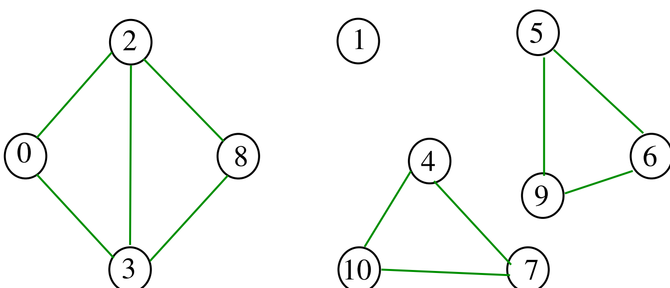
Exemplo: grafo com 4 componentes (conexos)



Contando componentes

Problema: calcular o número de componente

Exemplo: grafo com 4 componentes



Cálculo das componentes de grafos

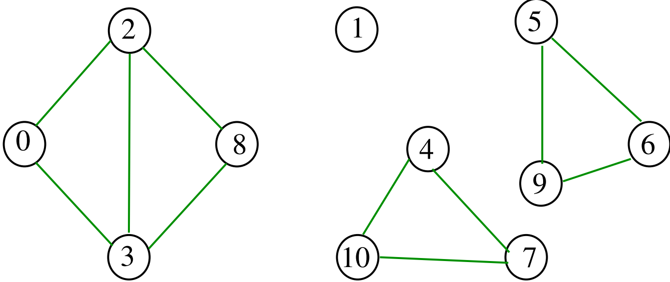
O classe `DFScc` determina o número de componentes do grafo `G`.

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`

A classe `Graph` é idêntica a classe `Digraph` onde `addEdge(v,w)` insere do digrafo os arcos `v-w` e `w-v`.

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(Graph 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(Graph 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(Graph 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

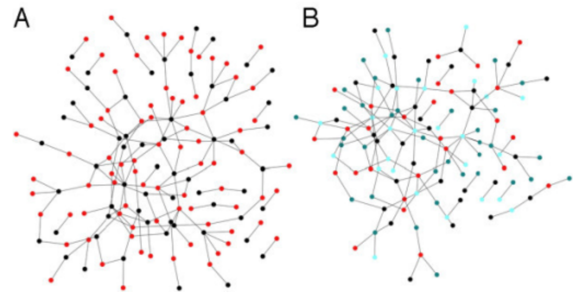
O consumo de tempo de DFScc para vetor de listas de adjacência é $O(V + E)$.

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

Fonte: ash.atozviews.com

APÊNDICE

grafos bipartidos e ciclos ímpares

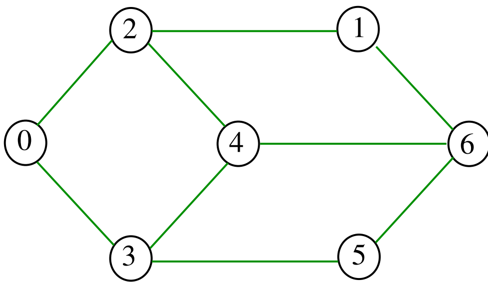


Fonte: `Modularity and anti-modularity in networks with arbitrary degree distribution`

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.

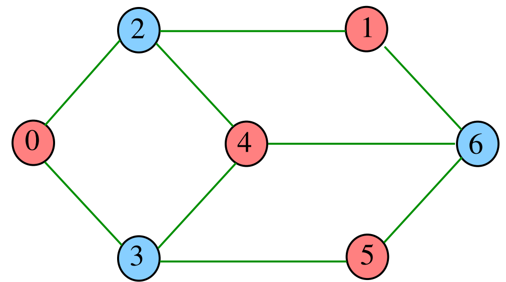
Exemplo:



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.

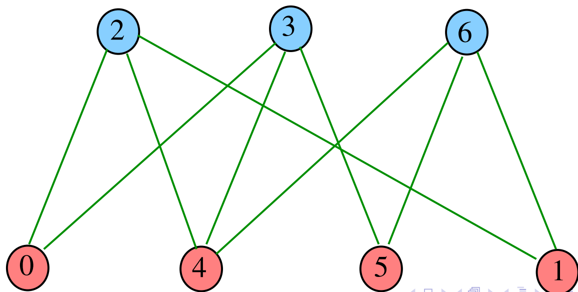
Exemplo:



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

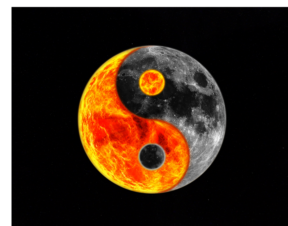
A classe DFSbipartite, para vetor de listas de adjacência, consome tempo $O(V + E)$ para decidir se um grafo é bipartido.

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

Certificado

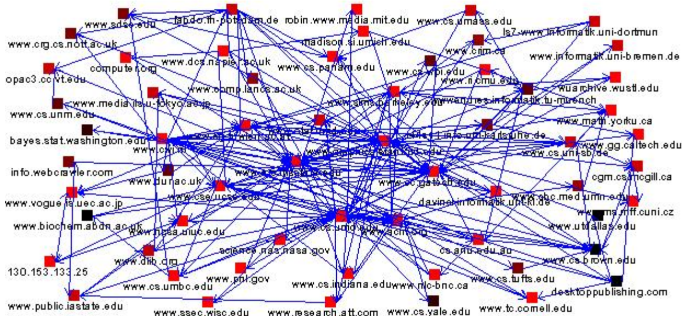
Para todo grafo G, vale uma e apenas umas das seguintes afirmações:

- G possui um ciclo ímpar
- G é bipartido



Fonte: Yin and Yang Yoga Workshop

Componentes fortemente conexos

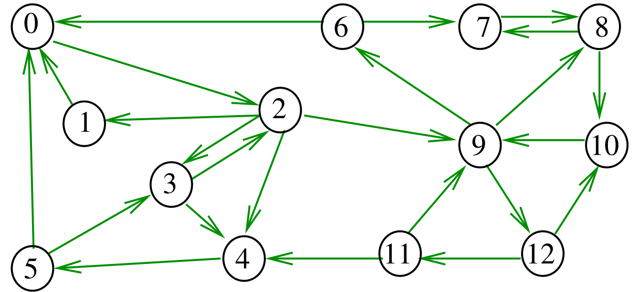


Fonte: `A System for Collecting and Analyzing Topic-Specific Web Information`

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`

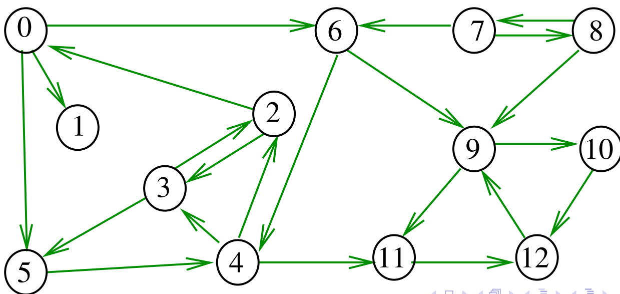
Exemplo: um digrafo fortemente conexo



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

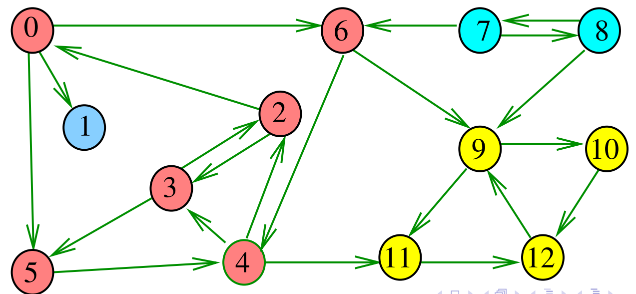
Exemplo: 4 componentes fortemente conexos



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

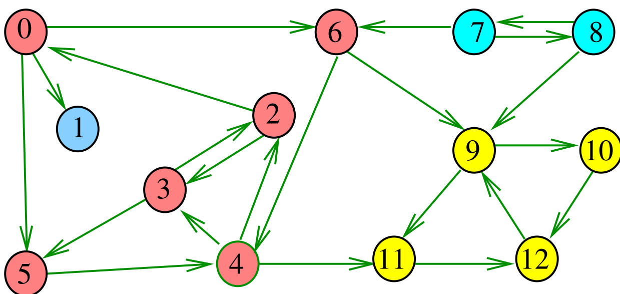
Exemplo: 4 componentes fortemente conexos



Determinando componentes f.c.

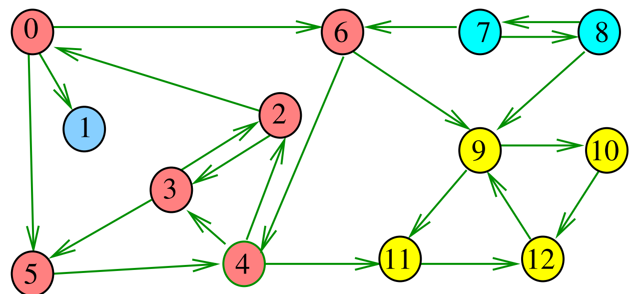
Problema: determinar os componentes fortemente conexos

Exemplo: 4 componentes fortemente conexos



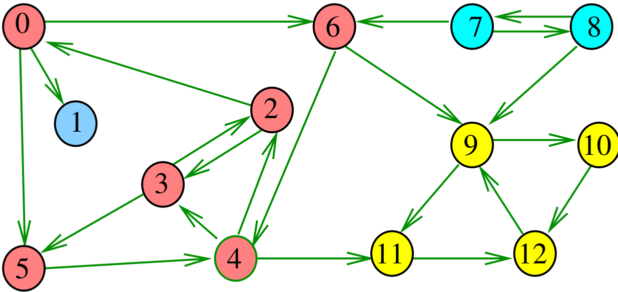
Exemplo

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.

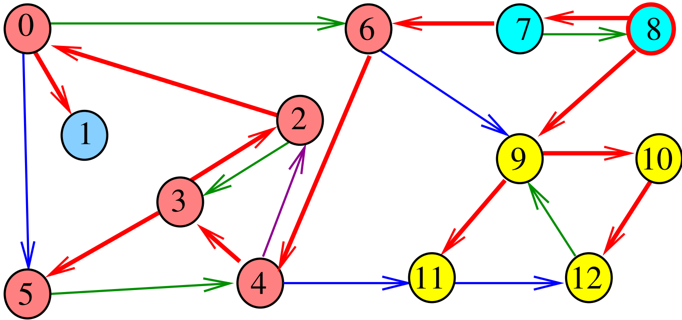
Tarjan realiza apenas um passo DFS sobre o digrafo.

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



I

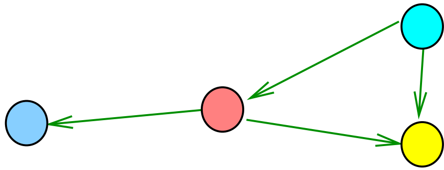
Digrafos dos componentes

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

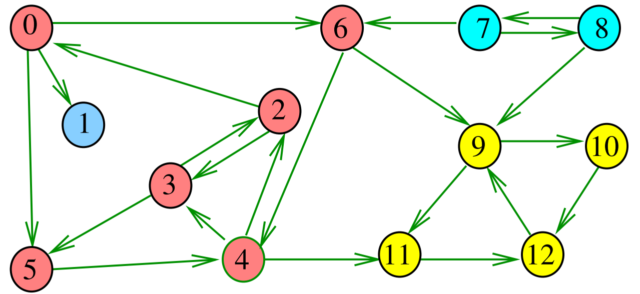
Digrafos dos componentes

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

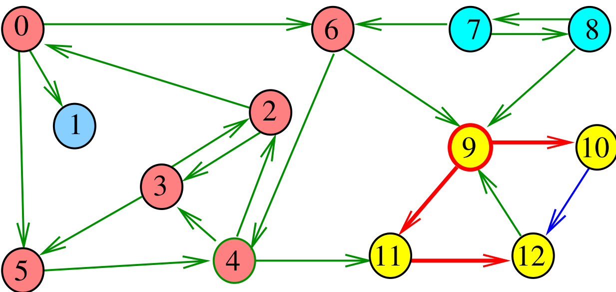
Digrafo dos componente é um DAG



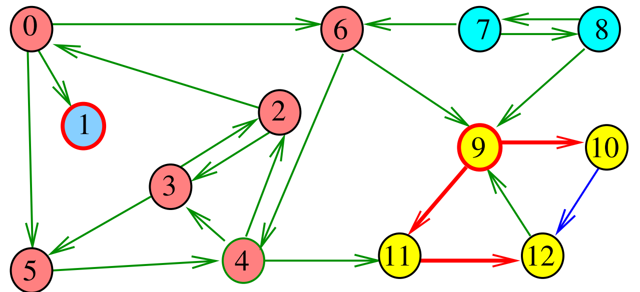
Ideia . . . G e DFS



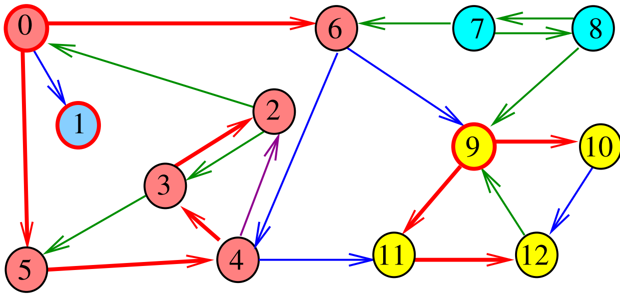
Ideia . . . G e DFS



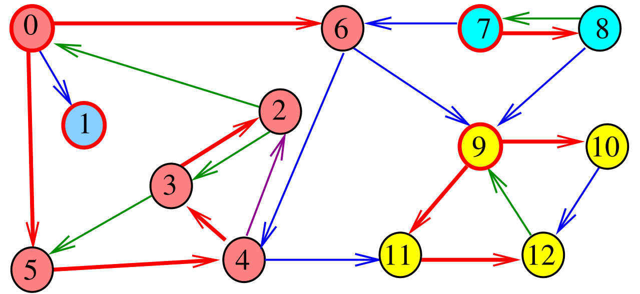
Ideia . . . G e DFS



Ideia . . . G e DFS

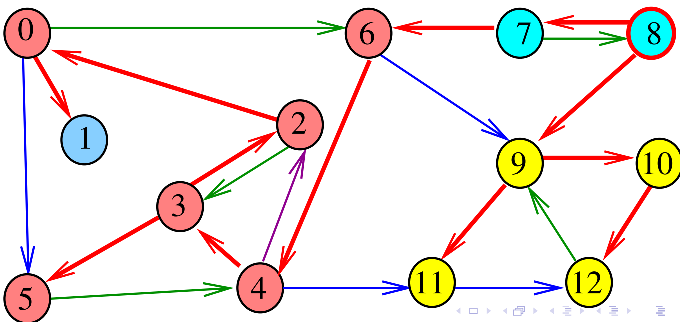


Ideia . . . G e DFS



Numeração pós-ordem

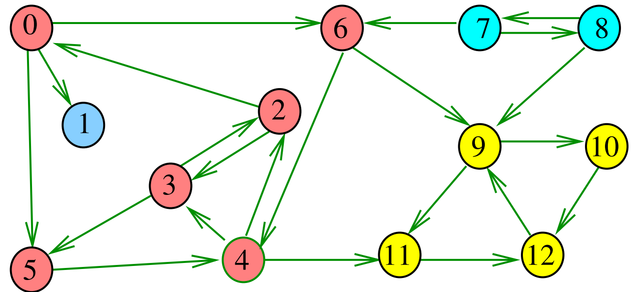
pós[v] = numeração pós-ordem de v
sóp[i] = vértice de numeração pós-ordem i
pós[W] = maior numeração pós-ordem de um vértice em W



Propriedade

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

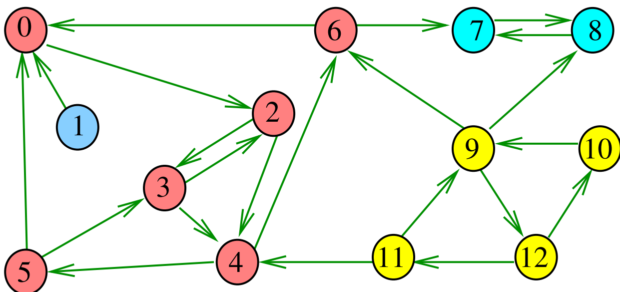
Exemplo: Digrafo G



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..

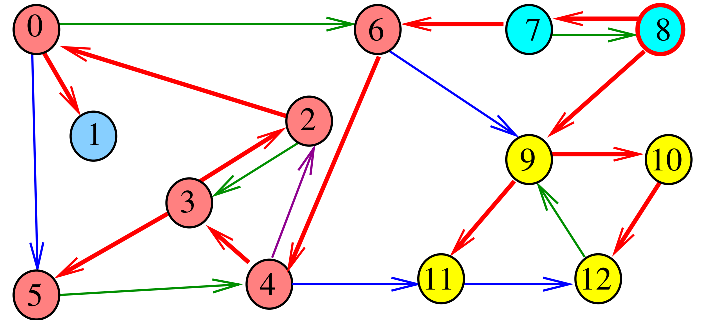
G, G reverso, DFS e pós[]

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** $\Rightarrow$ W forma um componente f.c. de R, e portanto de G;
- remova W de G e pegue o vértice v tal pós[v]...

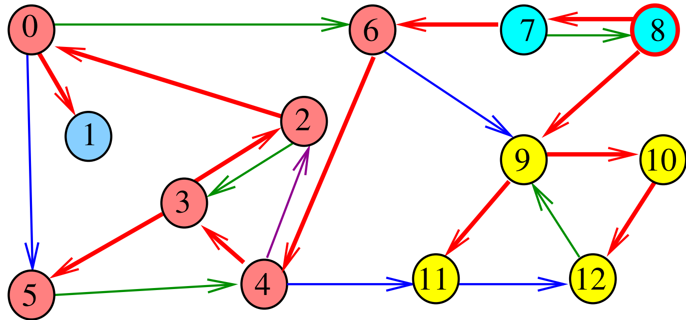
Exemplo

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



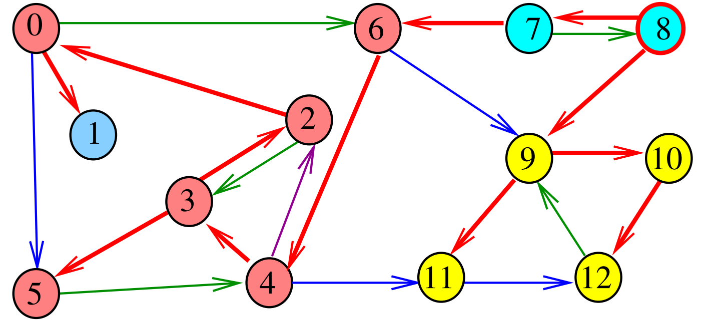
Exemplo

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

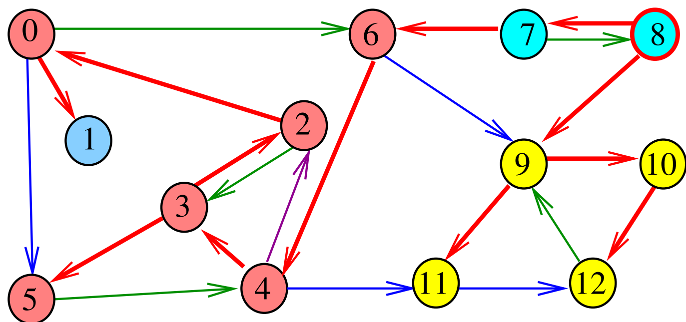
pós$[\{7, \mathbf{8}\}] = 12$
pós$[\{0, 2, 3, 4, 5, \mathbf{6}\}] = 10$
pós$[\{\mathbf{1}\}] = 5$
pós$[\{\mathbf{9}, 10, 11, 12\}] = 3$



Numeração pós-ordem e componentes f.c.

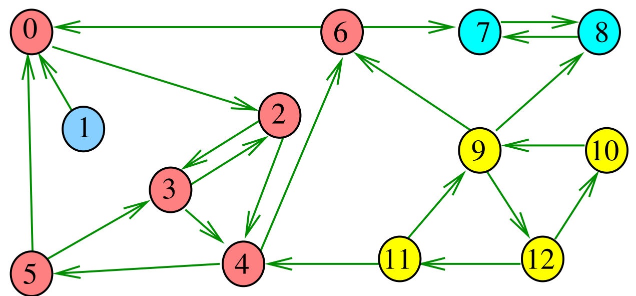
Se U e W são componentes f.c. e existe arco com ponta inicial em U e ponta final em W, então

$$\text{pós}[U] > \text{pós}[W]$$



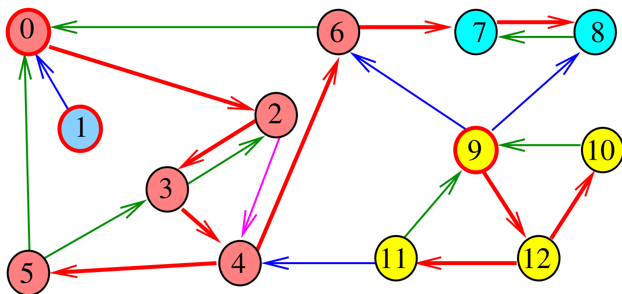
Digrafo reverso R

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



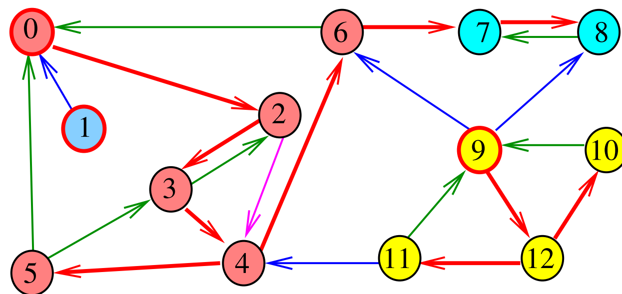
Digrafo reverso R e DFS

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



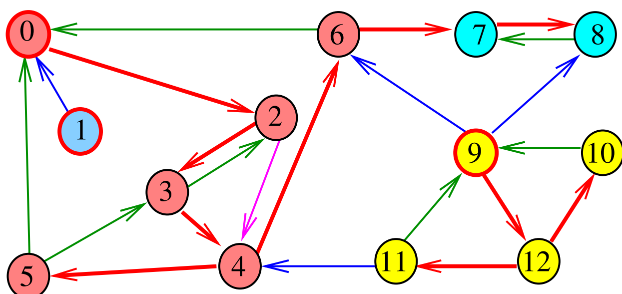
Digrafo reverso R e DFS

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



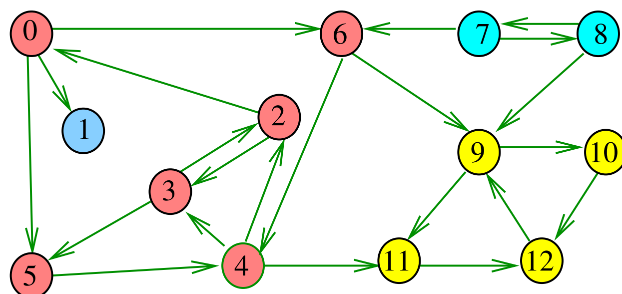
Digrafo reverso R e DFS

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



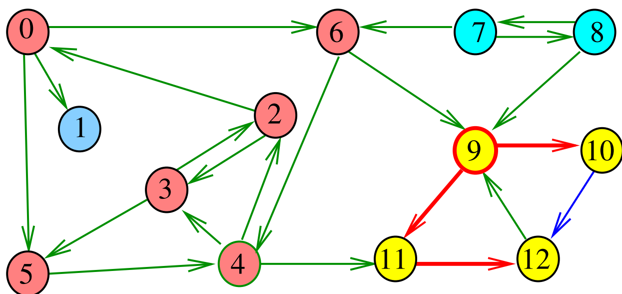
Digrafo G e DFS

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



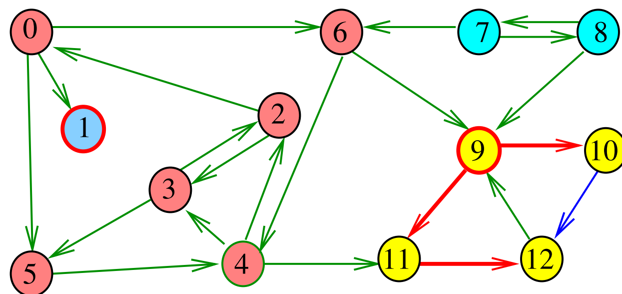
Digrafo G e DFS

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



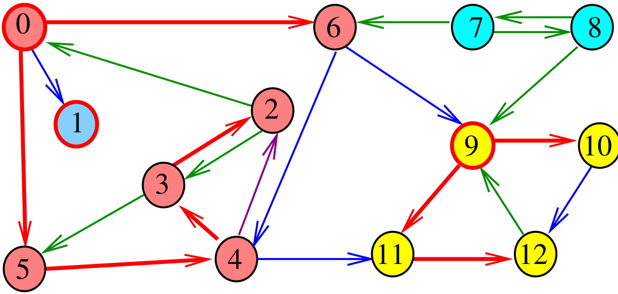
Digrafo G e DFS

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



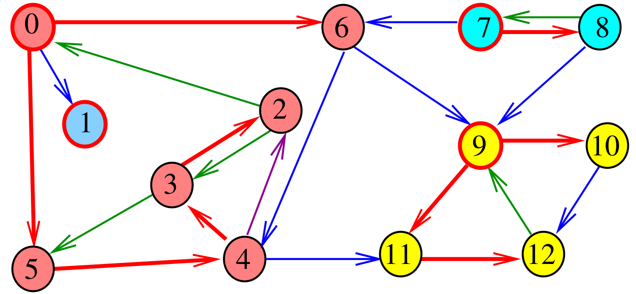
Digrafo G e DFS

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 $O(V^2)$.