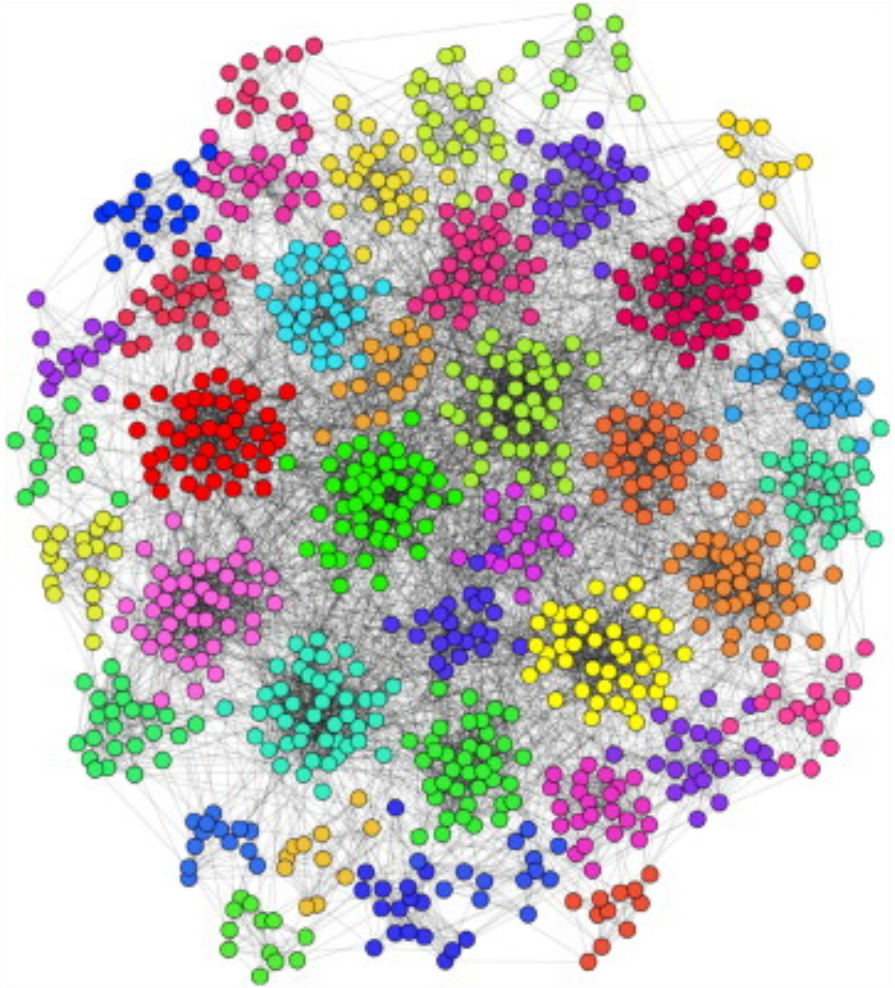
# MAC0323 Algoritmos e Estruturas de Dados II

Edição 2020 – 2

# AULA 21

# Componentes de grafos

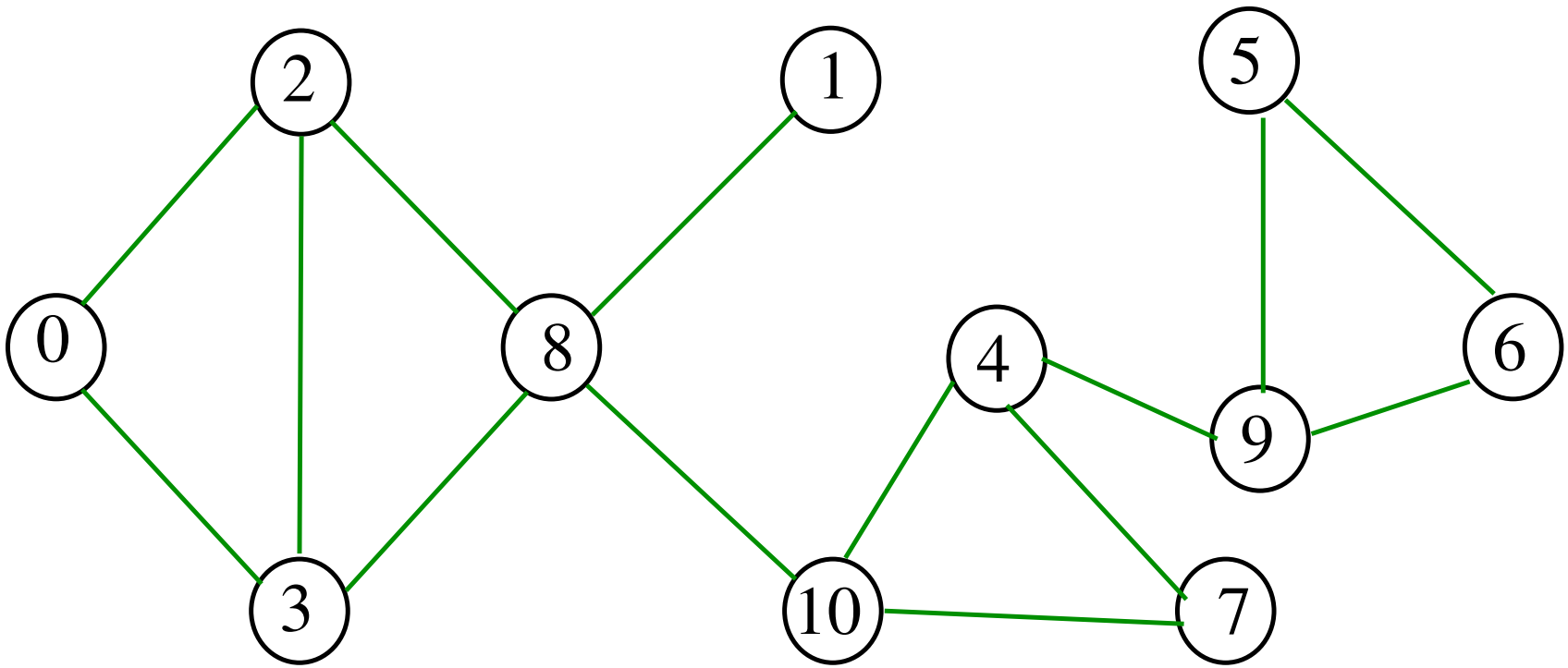


Fonte: Personalized PageRank Clustering: A graph clustering algorithm based on random walks

# 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`.

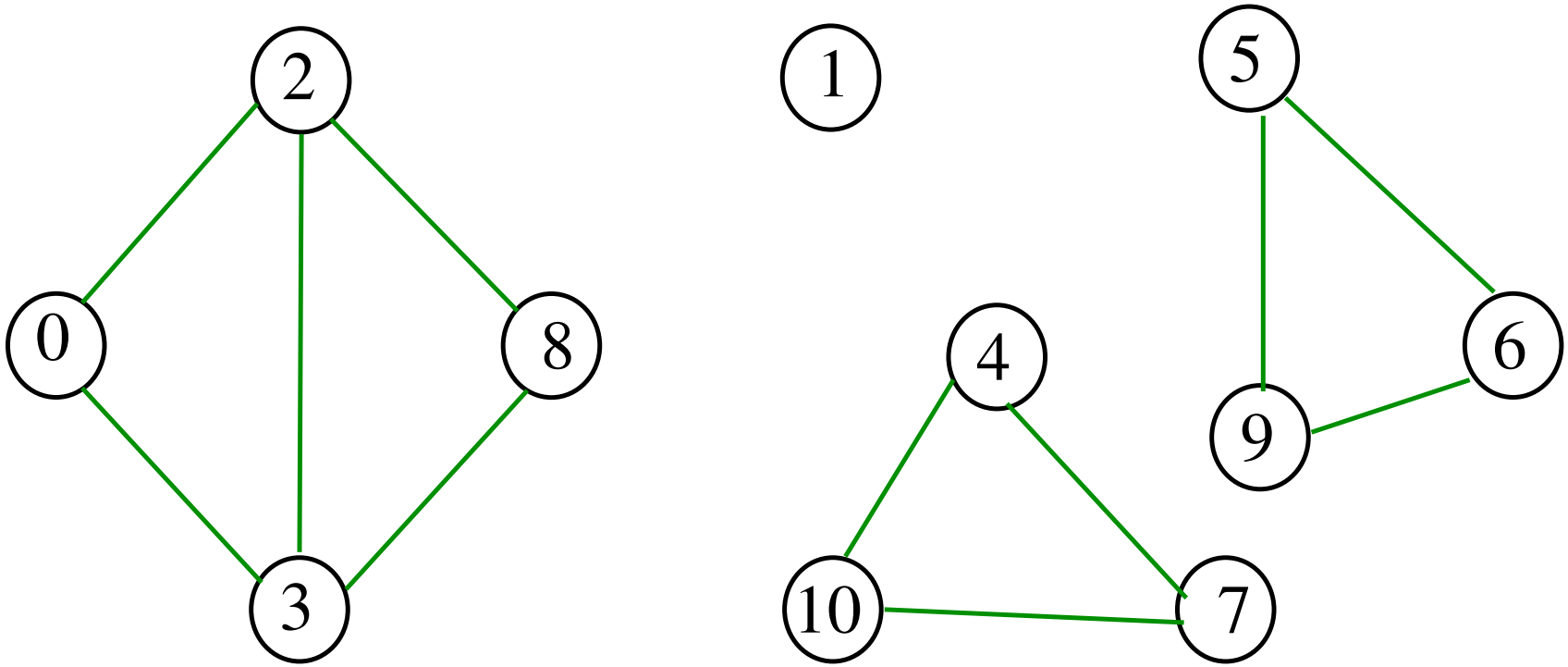
Exemplo: um grafo conexo



# Componentes de grafos

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

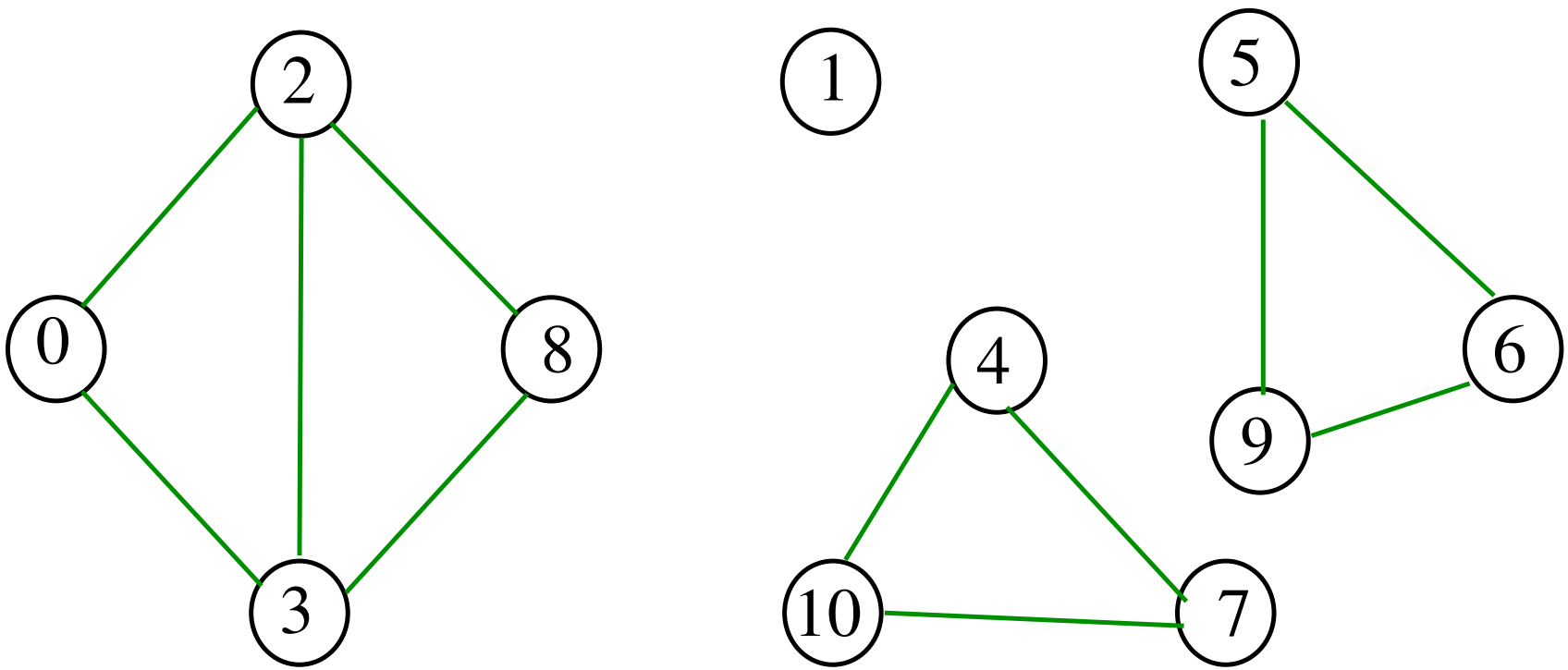
Exemplo: grafo com 4 componentes (conexos)



# Contando componentes

Problema: calcular o número de componente.

Exemplo: grafo com 4 componentes



# Cálculo das componentes de grafos

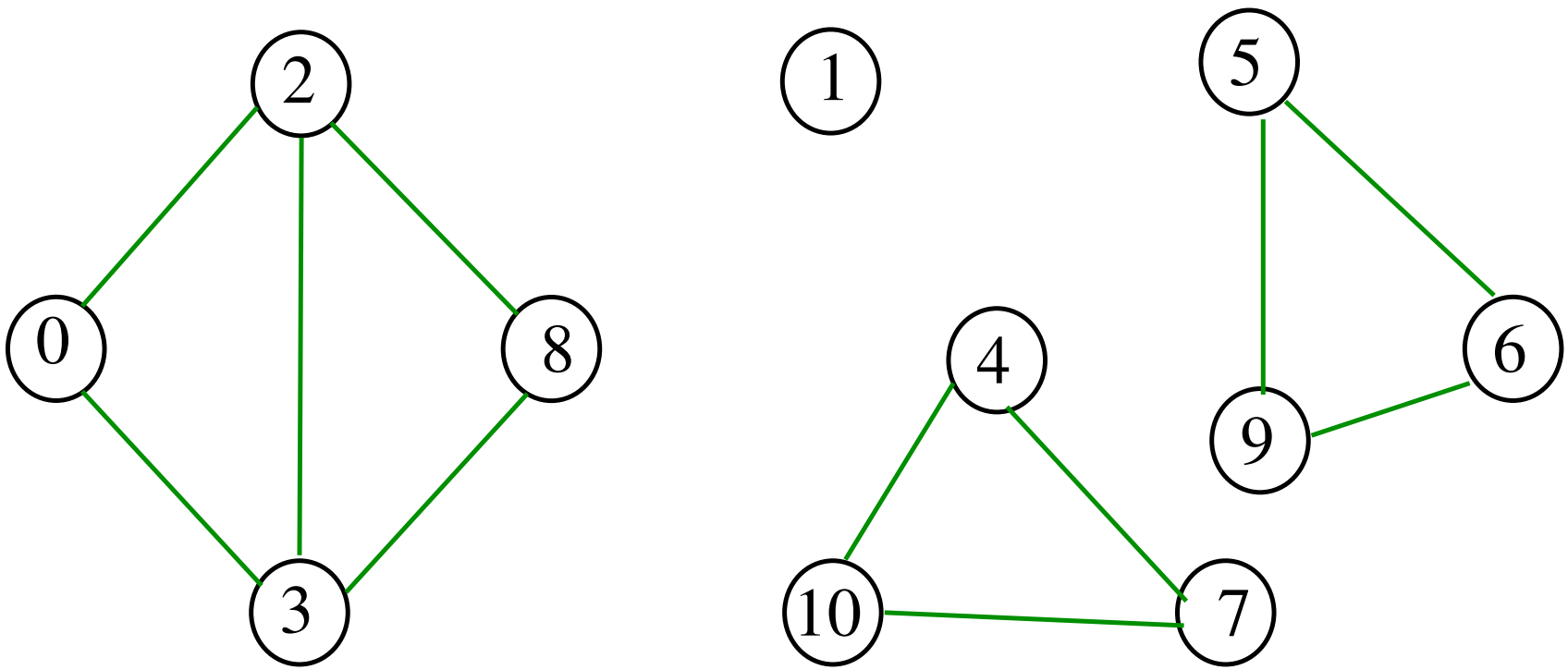
Para determinar o número de componentes
do grafo `G`, vamos armazenar no vetor `id[]`
o número da componente a que o vértice pertence:

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 no 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 |



# DFScc: estrutura

```
static struct dfscc {
    int  count;
    int  *id;
    bool *marked;                    /* DFS */
    int  *edgeTo;                    /* DFS */
};

typedef struct dfscc *dfsCC;

dfsCC DFSccInit(Graph G) {
    dfsCC cc = mallocSafe(sizeof(*cc));
    cc->count = 0;
    cc->id = mallocSafe(G->V*sizeof(int));

    /* DFSPathsInit */

    ...
}
```

# Como obter as componentes de um grafo

```
dfsCC DFScc(Graph G) {...}
static void dfs(Graph G, int v, dfsCC cc) {}
bool connected(dfsCC cc, int v, int w) {...}
int id(dfsCC cc, int v) {...}
int count(dfsCC cc) {...}
```

# DFScc

Determina as componentes de um dado grafo G.

```
dfsCC DFScc(Graph G) {
   dfsCC cc = DFSccInit(G);
   for (int v = 0; v < G->V; v++)
      if (!cc->marked[v]) {
         dfs(G, v, cc);   /* determina componente de v */
         (cc->count)++;
      }
   return cc;
}
```

# DFScc: dfs()

```
static void dfs(Graph G, int v, dfsCC cc) {
   Link w;
   cc->marked[v] = true;
   cc->id[v] = cc->count;
   for (w = G->adj[v]; w != NULL; w = w->next)
      if (!cc->marked[w->vertex]) {
         cc->edgeTo[w->vertex] = v;
         dfs(G, w->vertex, cc);
      }
}
```

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

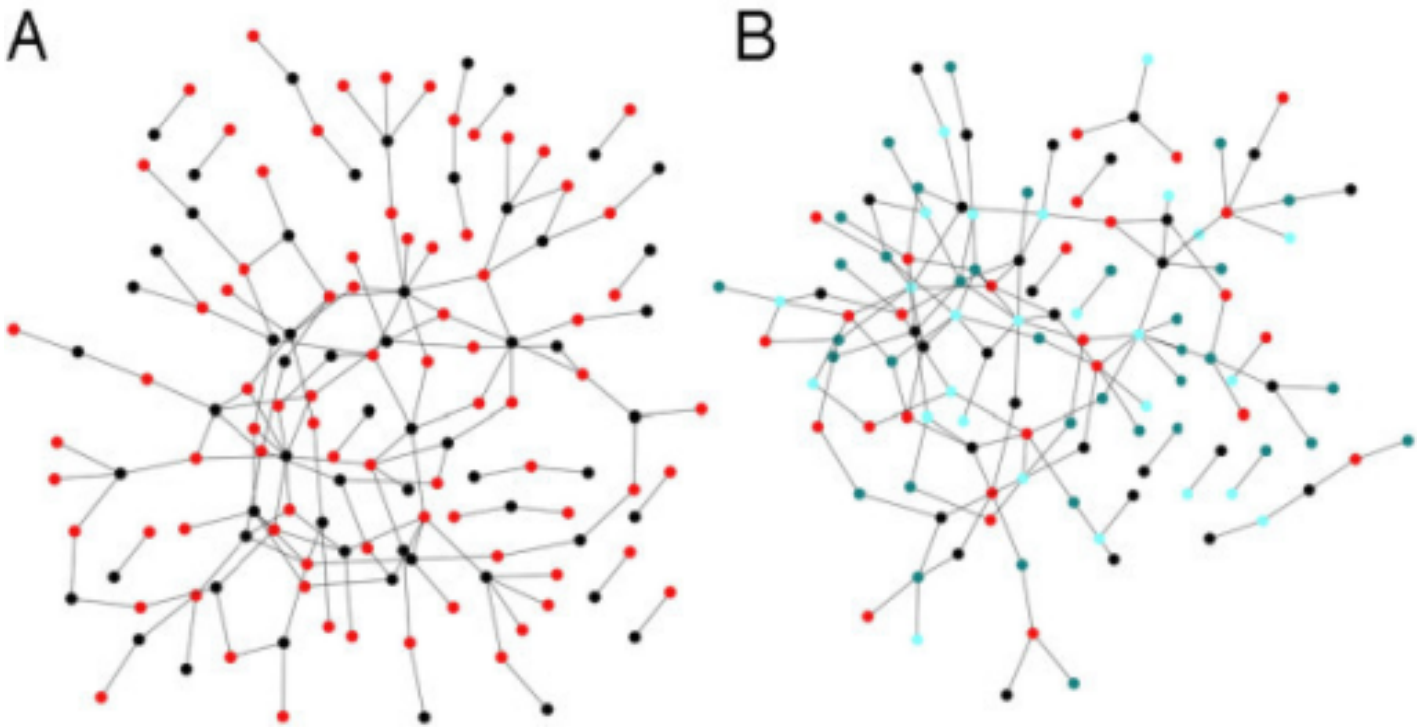
```
int id(dfsCC cc, int v) {
   return cc->id[v];
}

bool connected(dfsCC cc, int v, int w) {
   return cc->id[v] == cc->id[w];
}

int count(dfsCC cc) {
   return cc->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 $O(V^2)$.

# grafos bipartidos e ciclos ímpares

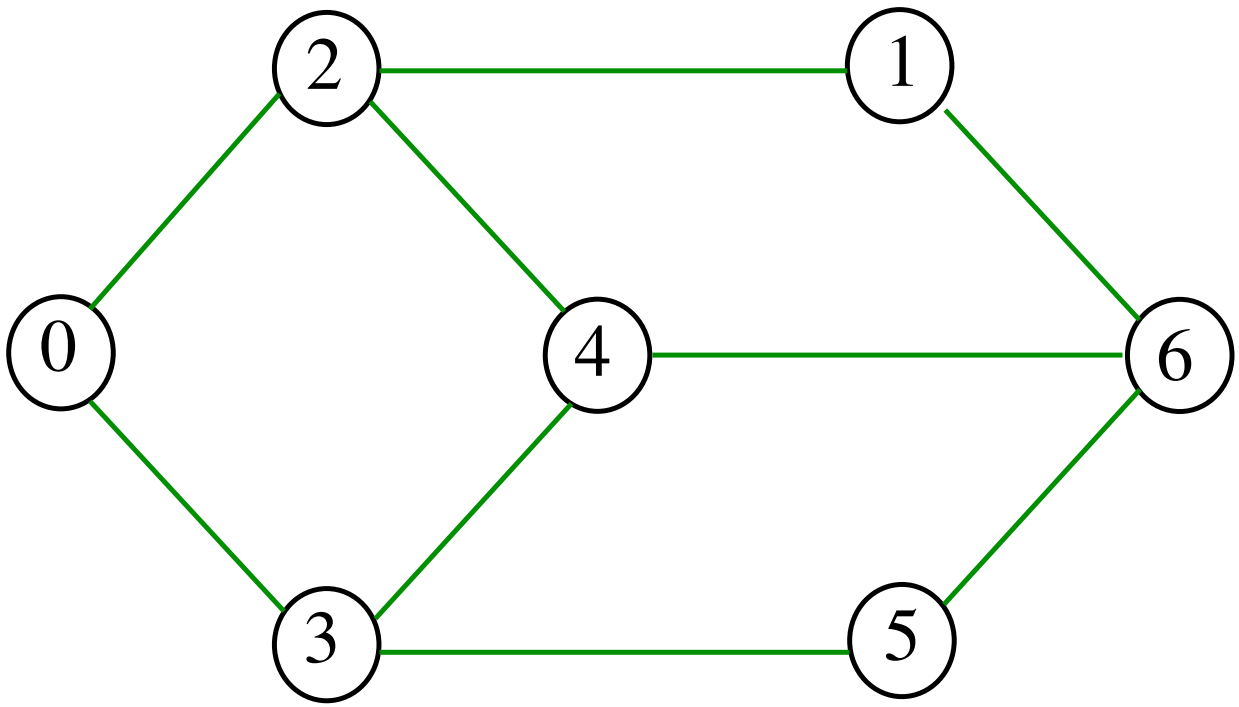



Fonte: Modularity and anti-modularity in networks with arbitrary degree distribution

# 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.

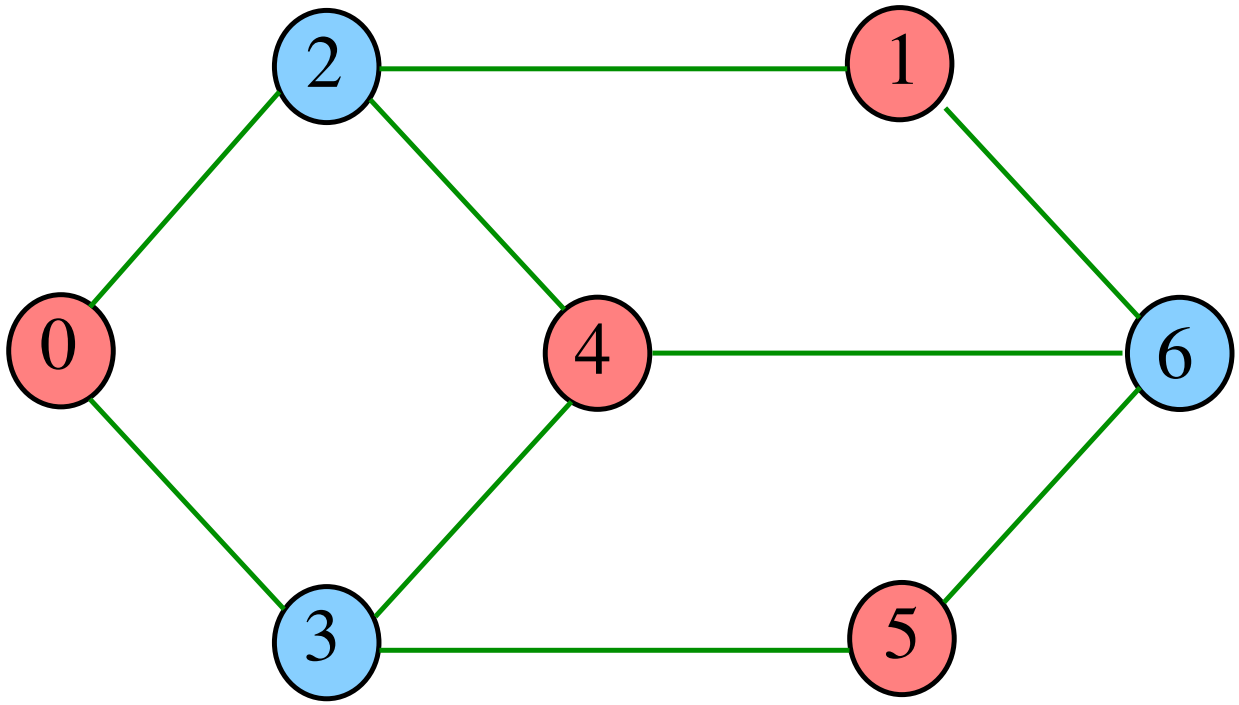
Exemplo:



# 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.

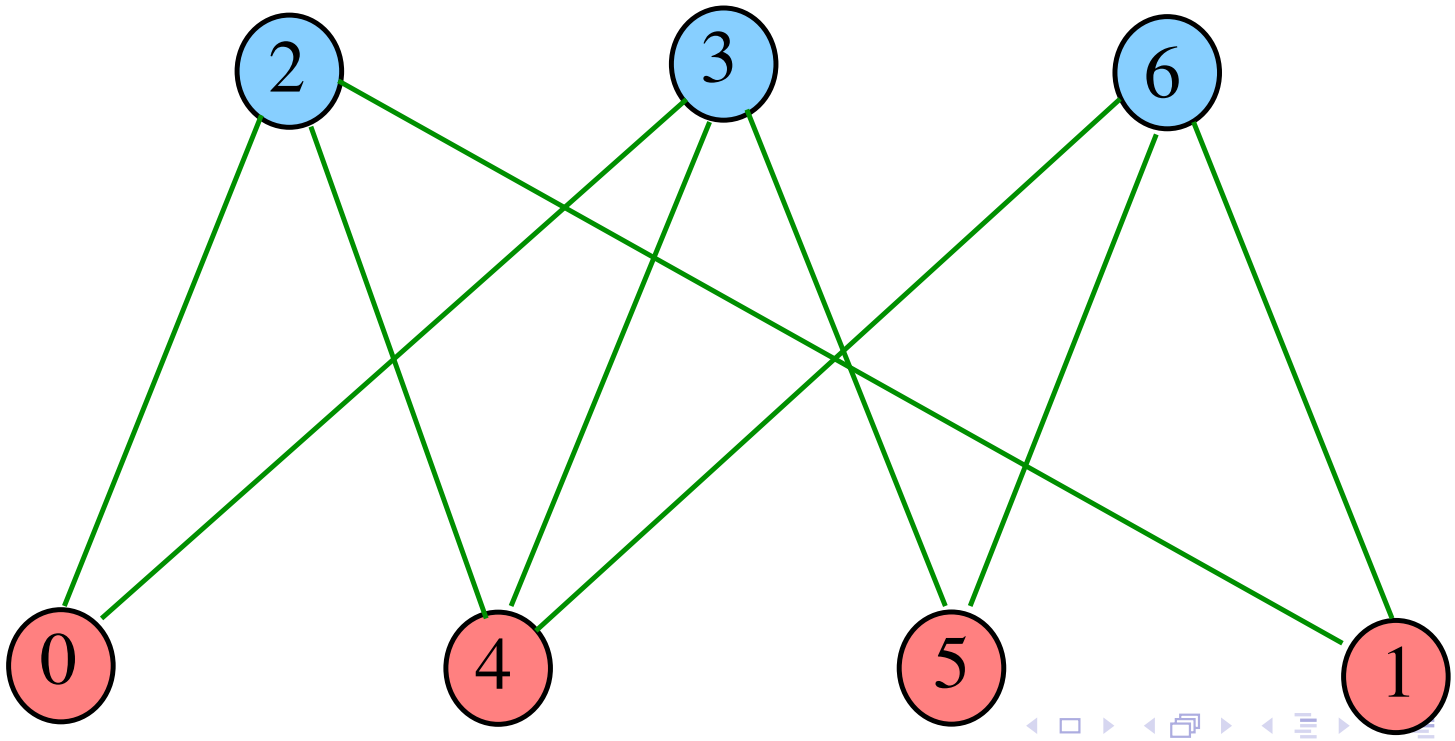
Exemplo:



# 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:



# Rotina DFSbipartite

A rotina 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.

# Rotina DFSbipartite

A rotina 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:

```
static bool color = mallocSafe(G->V*sizeof(bool));
```

# DFSbipartite: esqueleto

```
static bool *marked;                    /* DFS */
static int  *edgeTo;                    /* DFS */

static bool *color;                /* TwoColor */
static bool isTwoColorable = true;

static Stack cycle = stackInit();
static int onCycle = -1;

void DFSbipartite(Graph G) {...}
static void dfs(Digraph G, int v) {...}
bool isBipartite() {...}
Stack oddCycle() {...}
```

# DFSbipartite

```
void DFSbipartite(Graph G) {
   DFSInit(G);
   color = mallocSafe(G->V*sizeof(bool));
   for (int v = 0; v < G->V; v++)
      if (!marked(v)) {
         dfs(G, v);
      }
}
```

DFSInit(G): aloca e inicializa os vetores da DFS.

# DFSbipartite: dfs()

```
static void dfs(Digraph G, int v) {
    Link w;
    marked[v] = true;
    for (w = G->adj[v]; w != NULL; w = w->next)
        if (!marked[w->vertex]) {
            color[w->vertex] = !color[v];
            edgeTo[w->vertex] = v;
            dfs(G, w->vertex);
            if (!isBipartite()) return;
        }
```

# DFSbipartite: dfs()

```
static void dfs(Digraph G, int v) {
    Link w;
    marked[v] = true;
    for (w = G->adj[v]; w != NULL; w = w->next)
        if (!marked[w->vertex]) {
            color[w->vertex] = !color[v];
            edgeTo[w->vertex] = v;
            dfs(G, w->vertex);
            if (!isBipartite()) return;
        } else if (color[v] == color[w->vertex]) {
            isTwoColorable = false;
            onCycle = v;
            edgeTo[w->vertex] = v; /* fecha o ciclo */
        }
}
```

# DFSbipartite

```
bool isBipartite() {
   return isTwoColorable;
}

Stack oddCycle() {
   if (isTwoColorable) return NULL;
   if (!stackEmpty(cycle)) return cycle;

   for (int x = edgeTo[onCycle];
            x != onCycle; x = edgeTo[x])
      push(cycle, x);
   push(cycle, onCycle);

   return cycle;
}
```

# Consumo de tempo

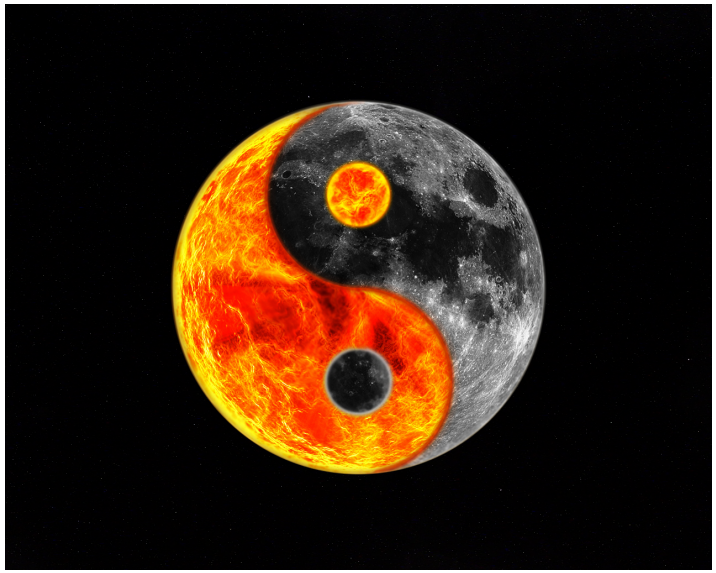
A rotina DFSbipartite, para vetor de listas de adjacência, consome tempo $O(V + E)$ para decidir se um grafo é bipartido.

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

# Certificado

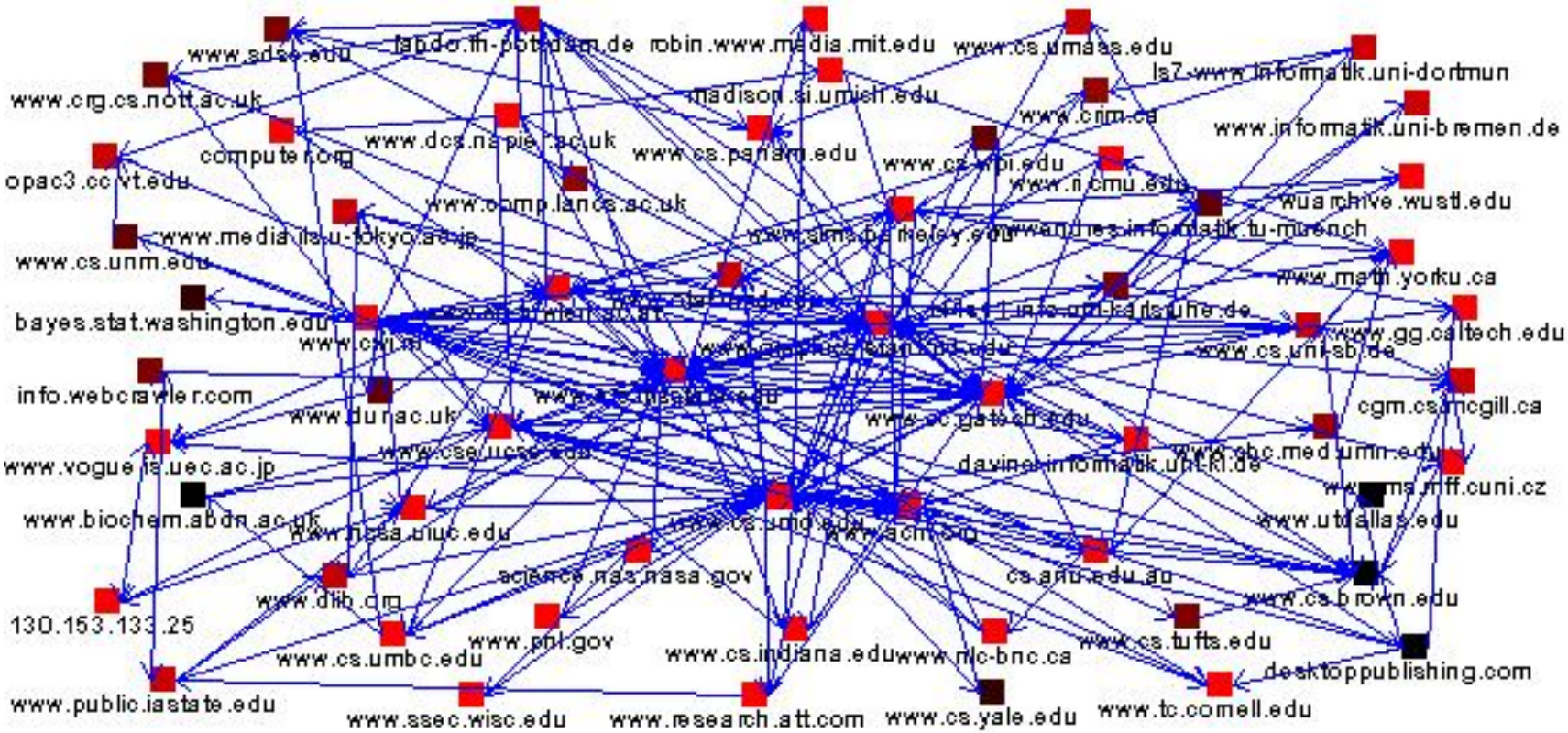
Para todo grafo G, vale uma e apenas uma das seguintes afirmações:

- G possui um ciclo ímpar;
- G é bipartido.



Fonte: Yin and Yang Yoga Workshop

# Componentes fortemente conexos

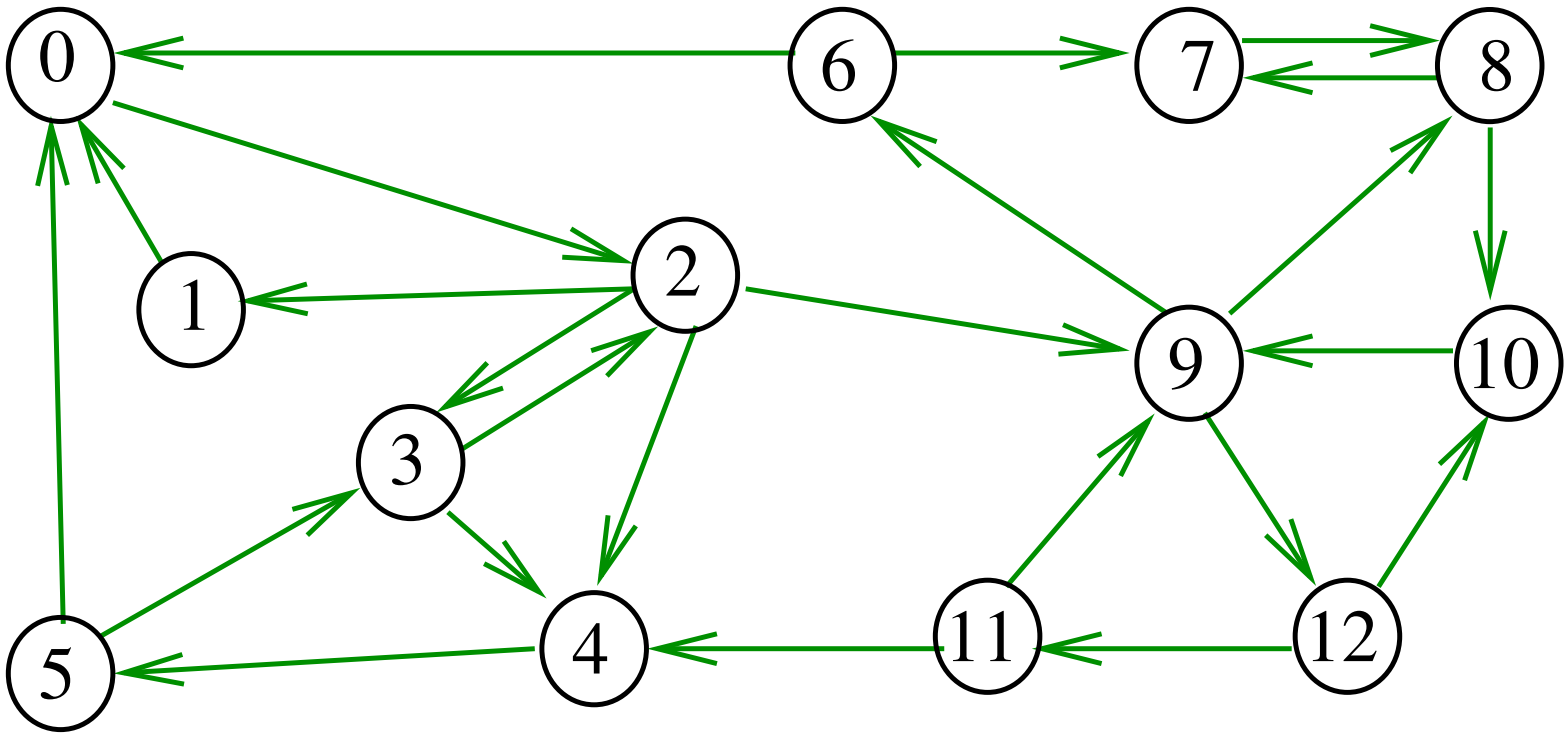



Fonte: A System for Collecting and Analyzing Topic-Specific Web Information

# 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



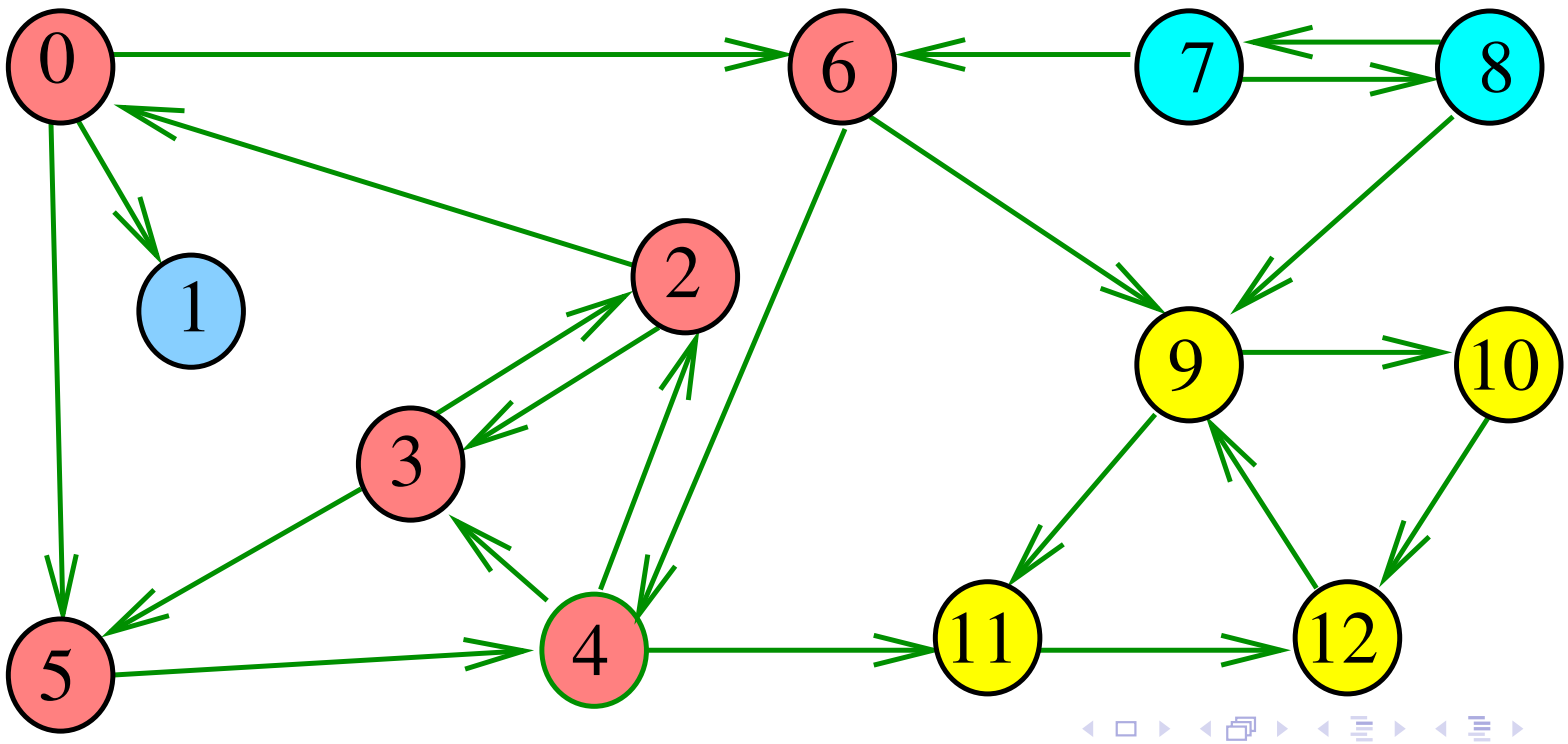
# Componentes fortemente conexos

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.

# 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.

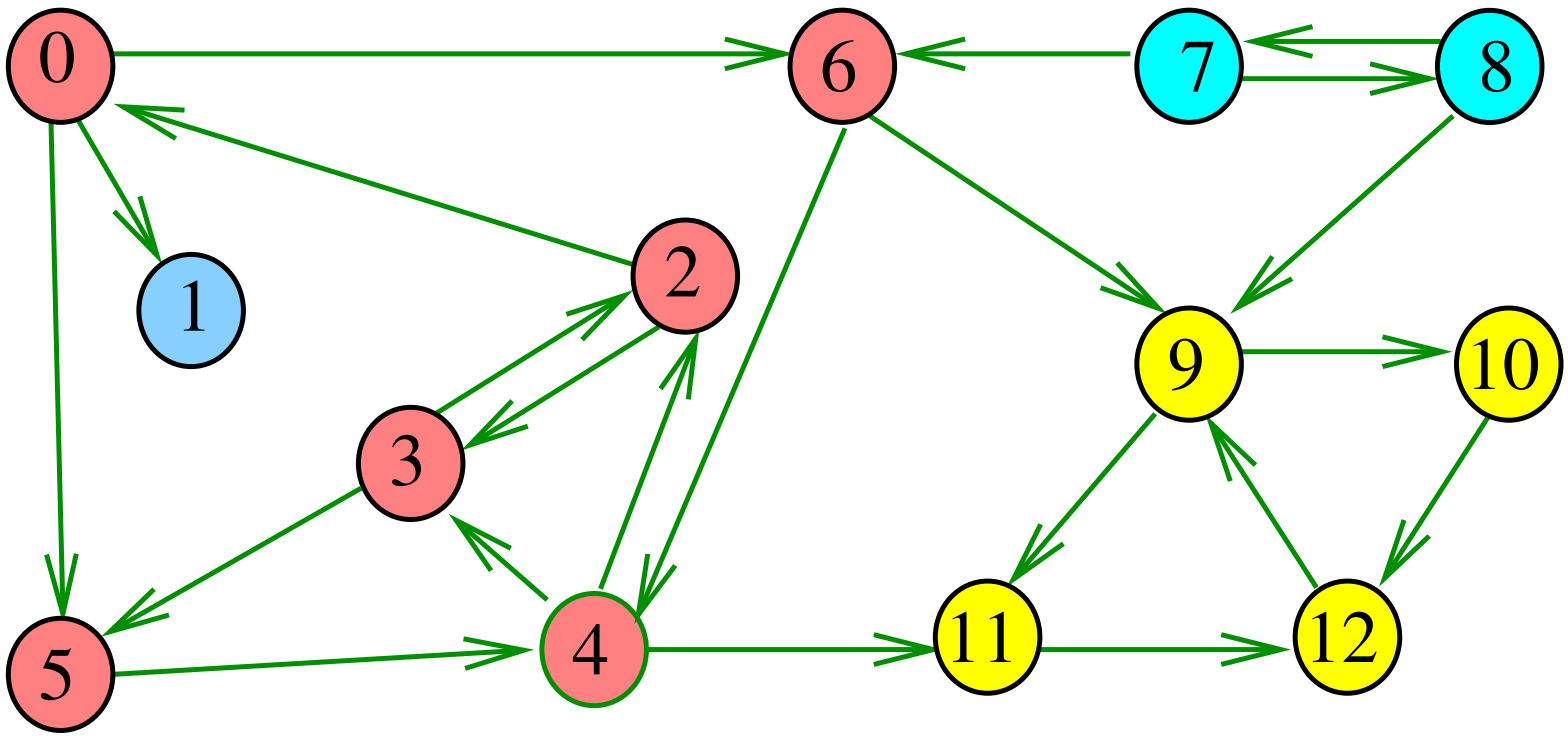
Exemplo: 4 componentes fortemente conexos



# Determinando componentes f.c.

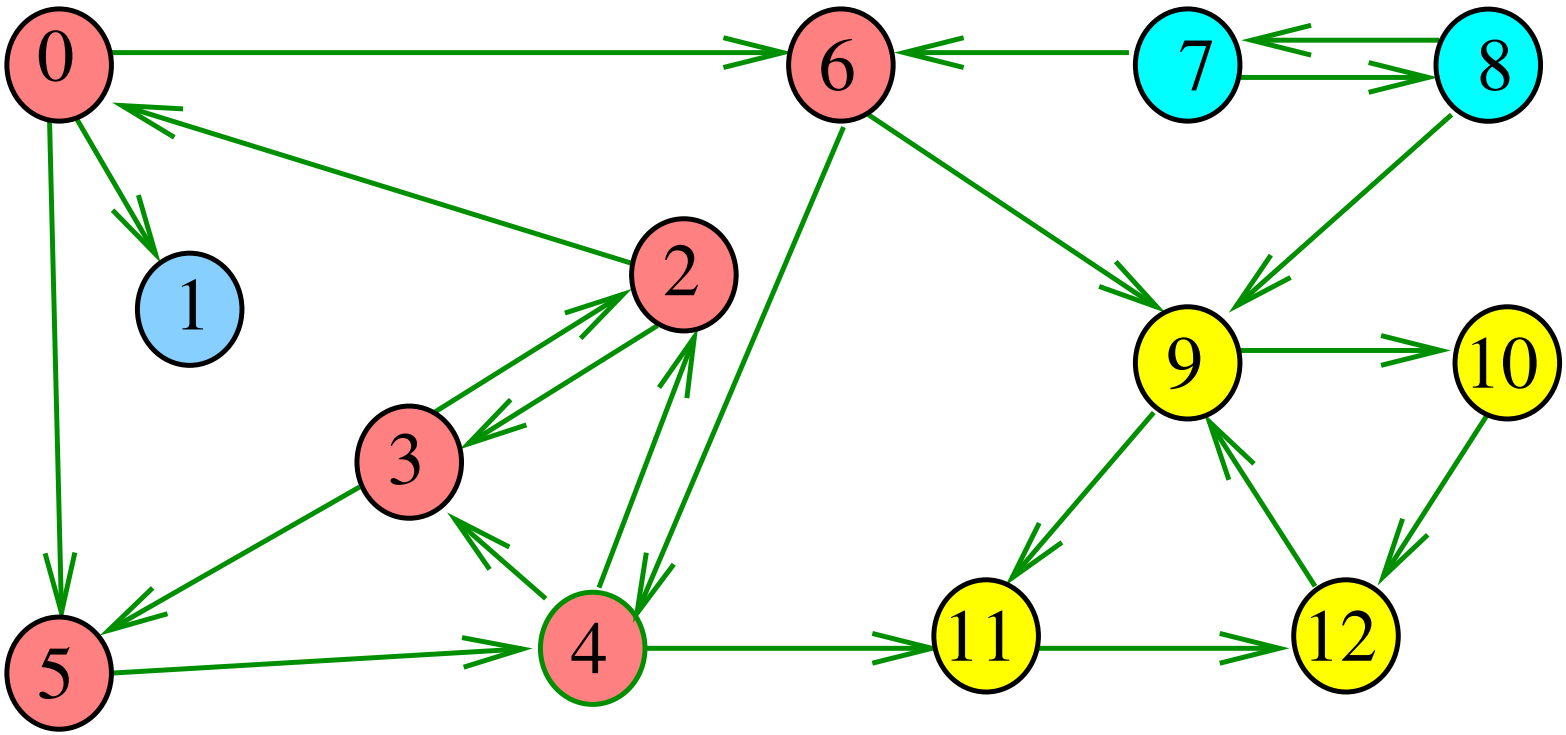
Problema: determinar os componentes fortemente conexos.

Exemplo: 4 componentes fortemente conexos



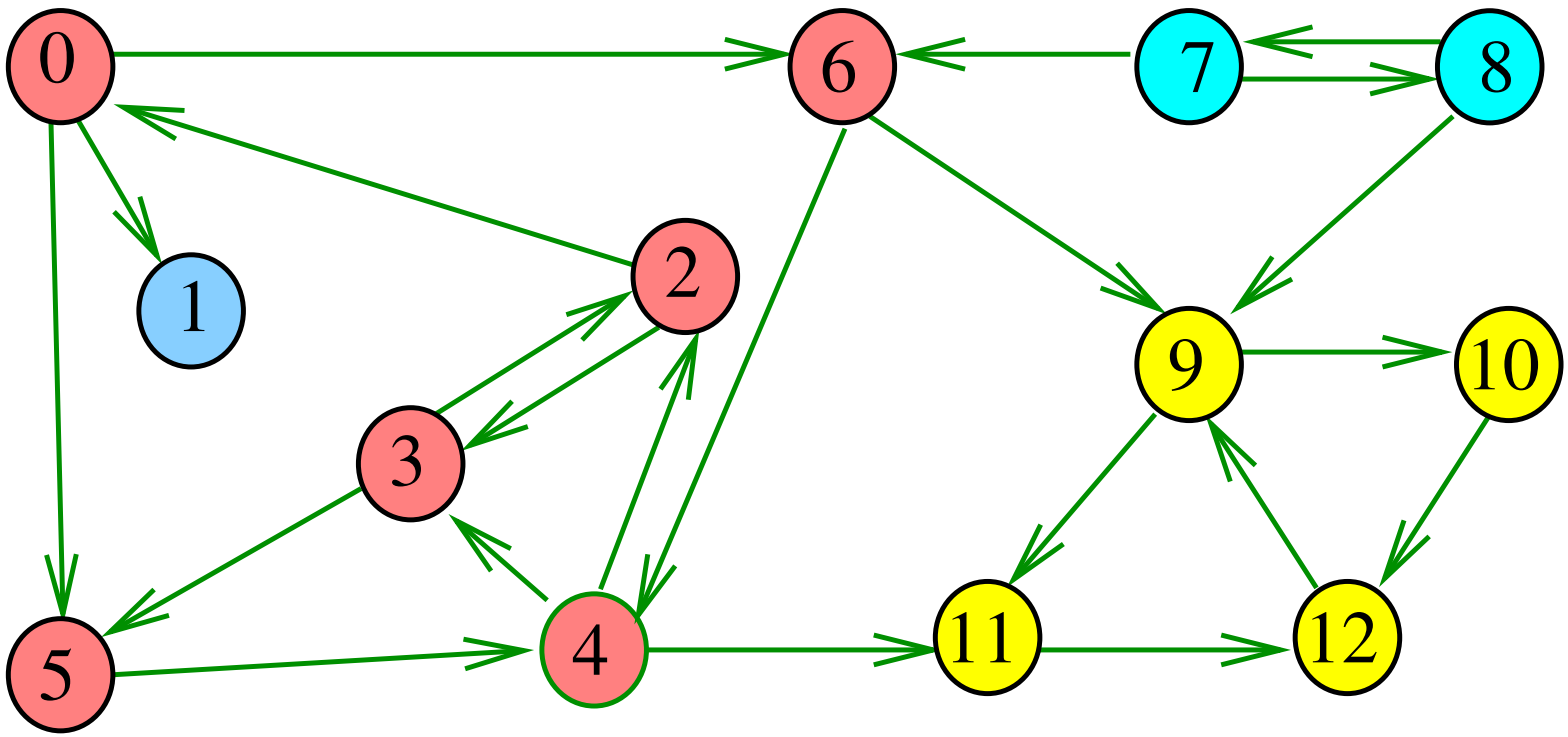
# Exemplo

| 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

```
static dfsCC cc;
void SCCforcaBruta(Digraph G) {...}
bool sConnected(int v, int w) {...}
int idSCC(int v) {...}
int countSCC() {...}
```

Para cada par `v`, `w` com $v < w$,
vê se existe caminho de `v` a `w` e de `w` a `v`
usando `DFSpaths`.

# Força Bruta

H: grafo que diz se dois vértices estão na mesma c.f.

```
void SCCforcaBruta(Digraph G) {
   Graph H = newGraph(G->V);
   for (int v = 0; v < G->V; v++) {
      dfsPaths dfsV = DFSpaths(G, v);
      for (int w = v+1; w < G->V; w++) {
         dfsPaths dfsW = DFSpaths(G, w);
         if (hasPath(dfsV, w) &&
            hasPath(dfsW, v))
            addEdge(H, v, w);
   }
   cc = DFScc(H);
}
```

# stronglyConnected

```
int idSCC(int v) {
   return id(cc, v);
}

bool sConnected(int v, int w) {
   return connected(cc, v, w);
}

int countSCC() {
   return count(cc);
}
```

# 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.

# 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.

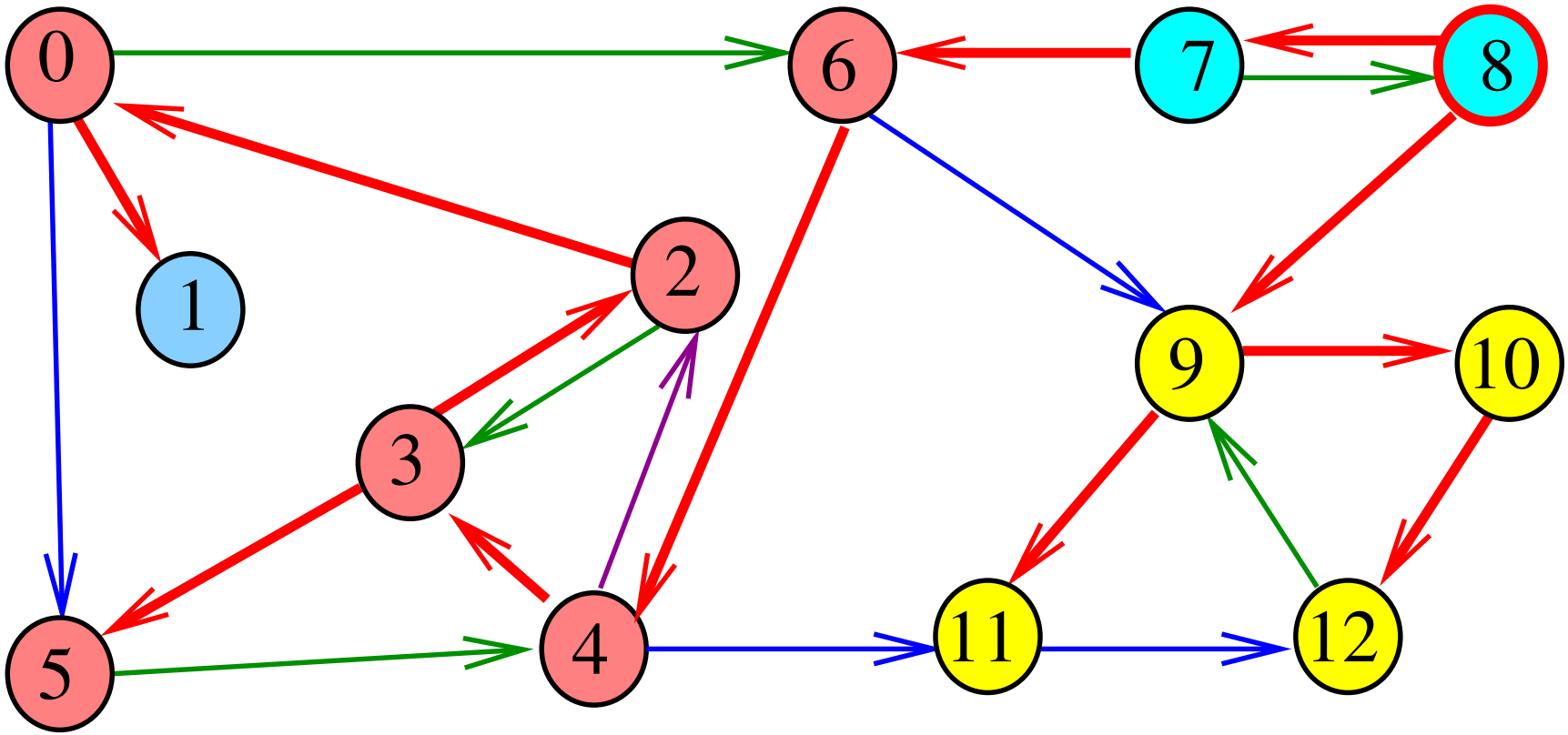
Tarjan realiza apenas um passo DFS sobre o digrafo.

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`.



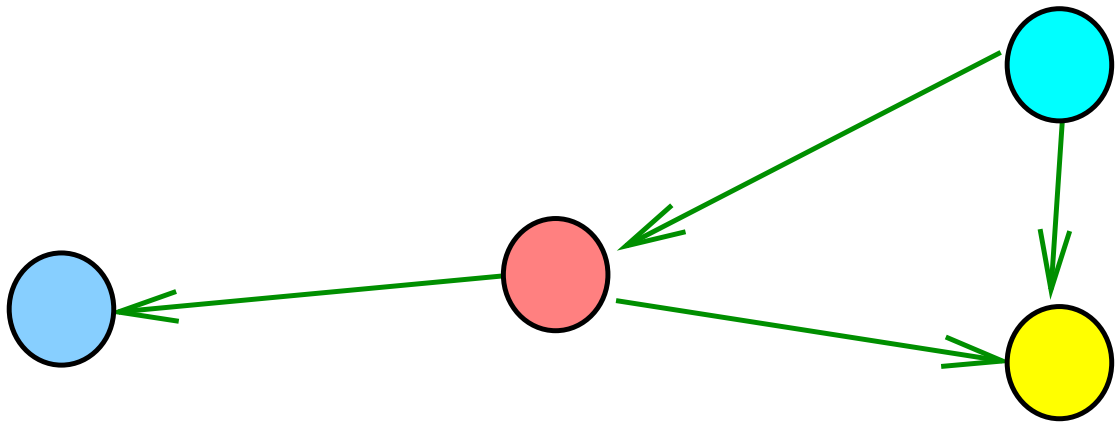
# Digrafos dos componentes

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

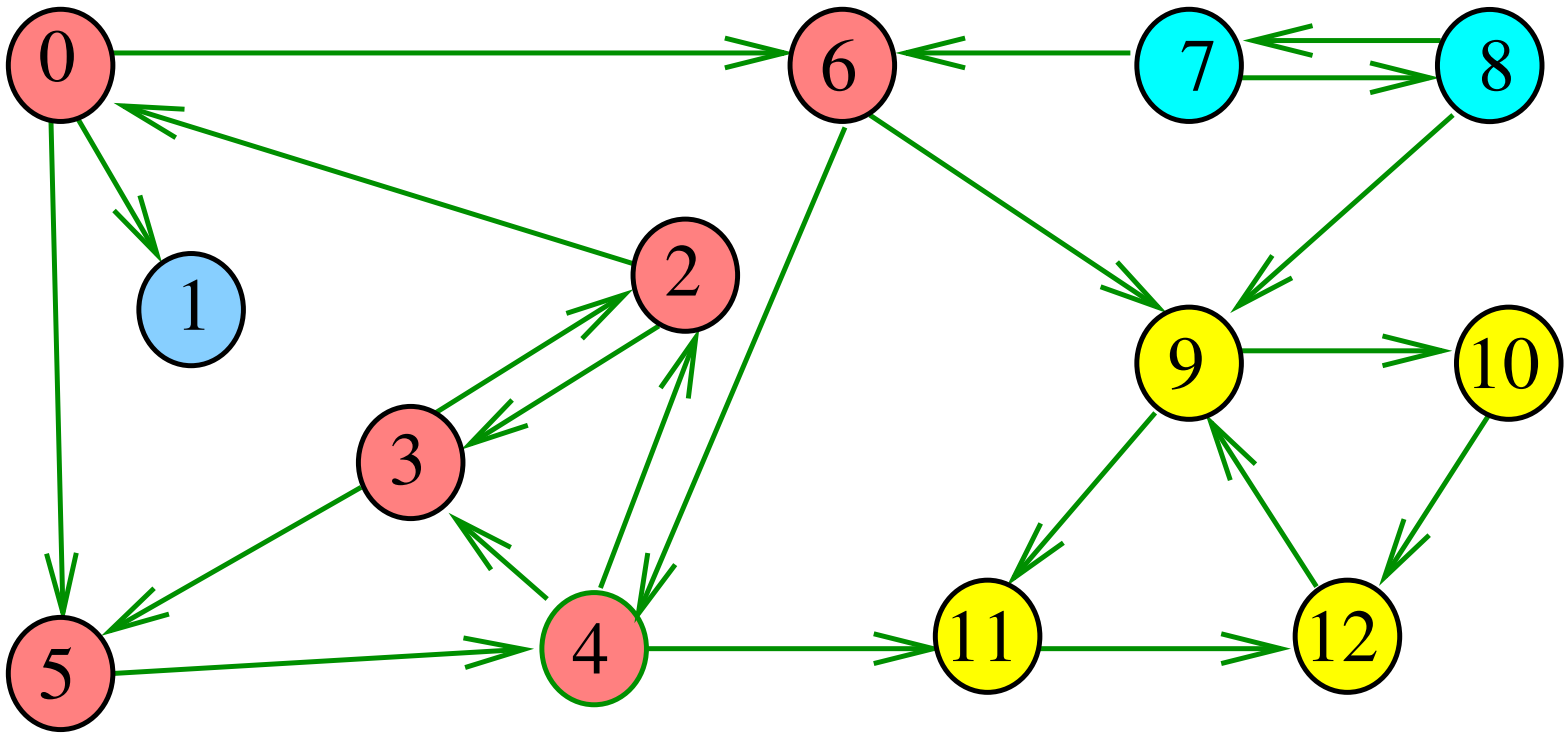
O **digrafo dos componentes** de G tem um vértice para cada componente fortemente conexo e um arco U-W se G possui um arco com ponta inicial em U e ponta final em W.

Digrafo dos componentes é um DAG!



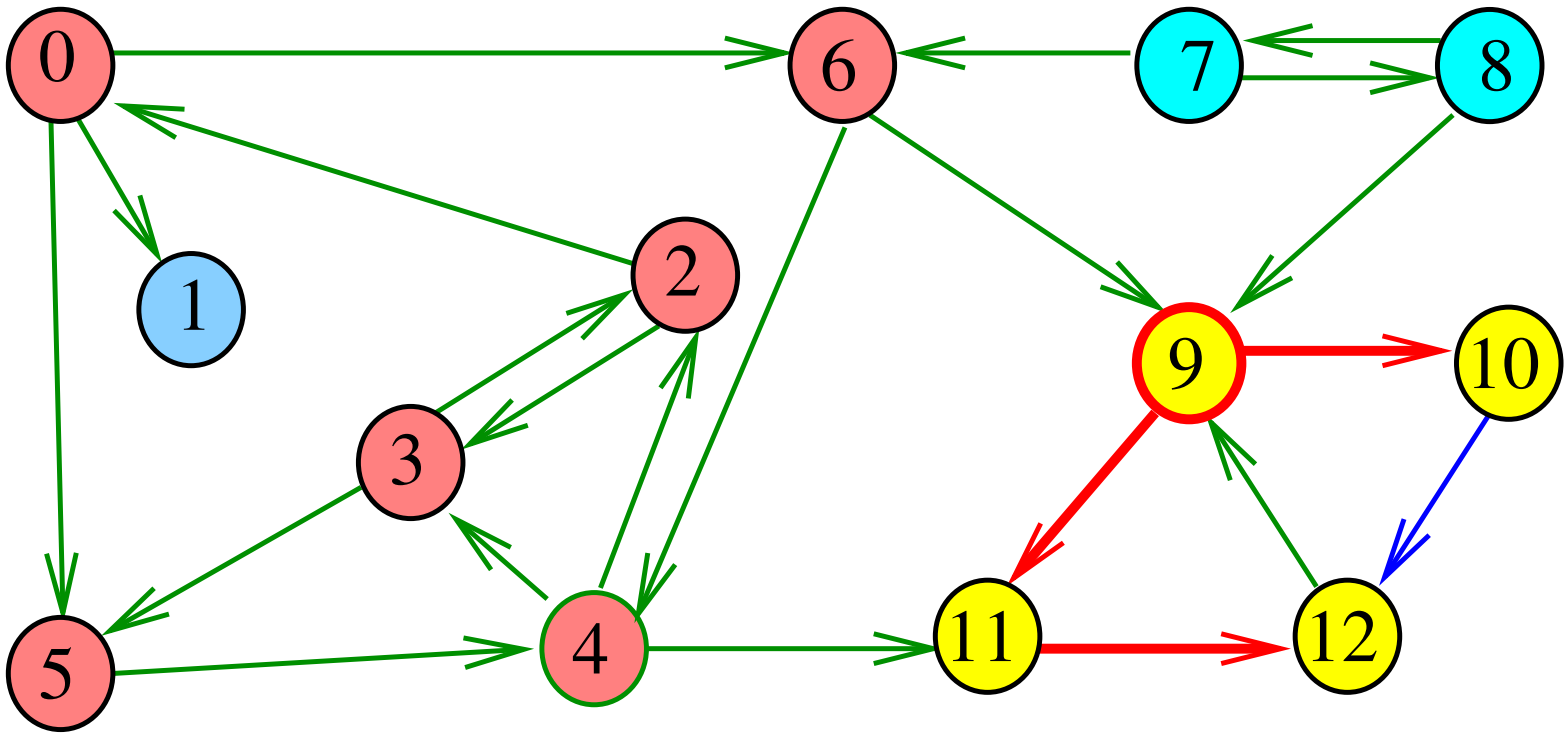
# Ideia . . . G e DFS

Visitar as componentes numa ordem topológica do digrafo das componentes...



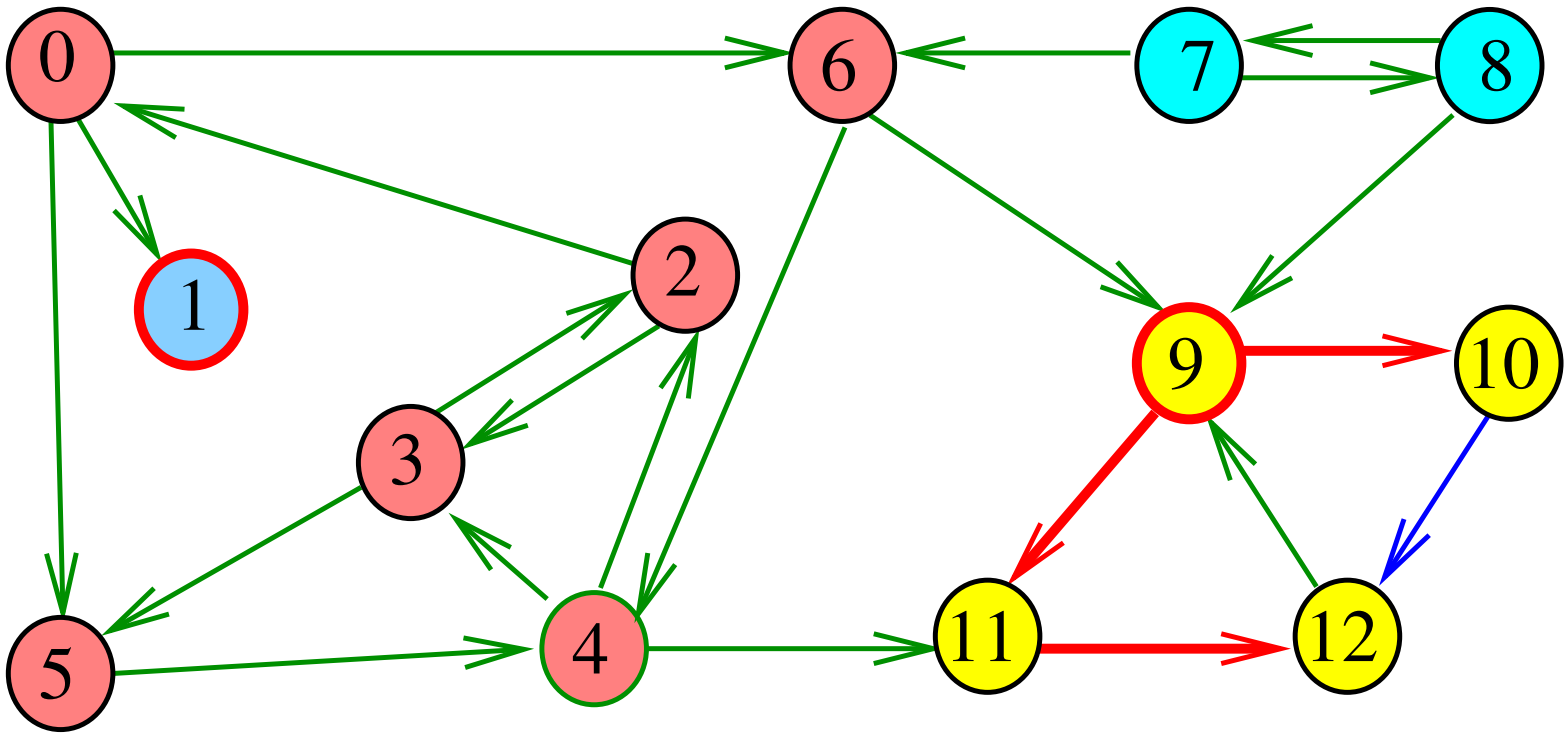
# Ideia . . . G e DFS

Visitar as componentes numa ordem topológica do digrafo das componentes...



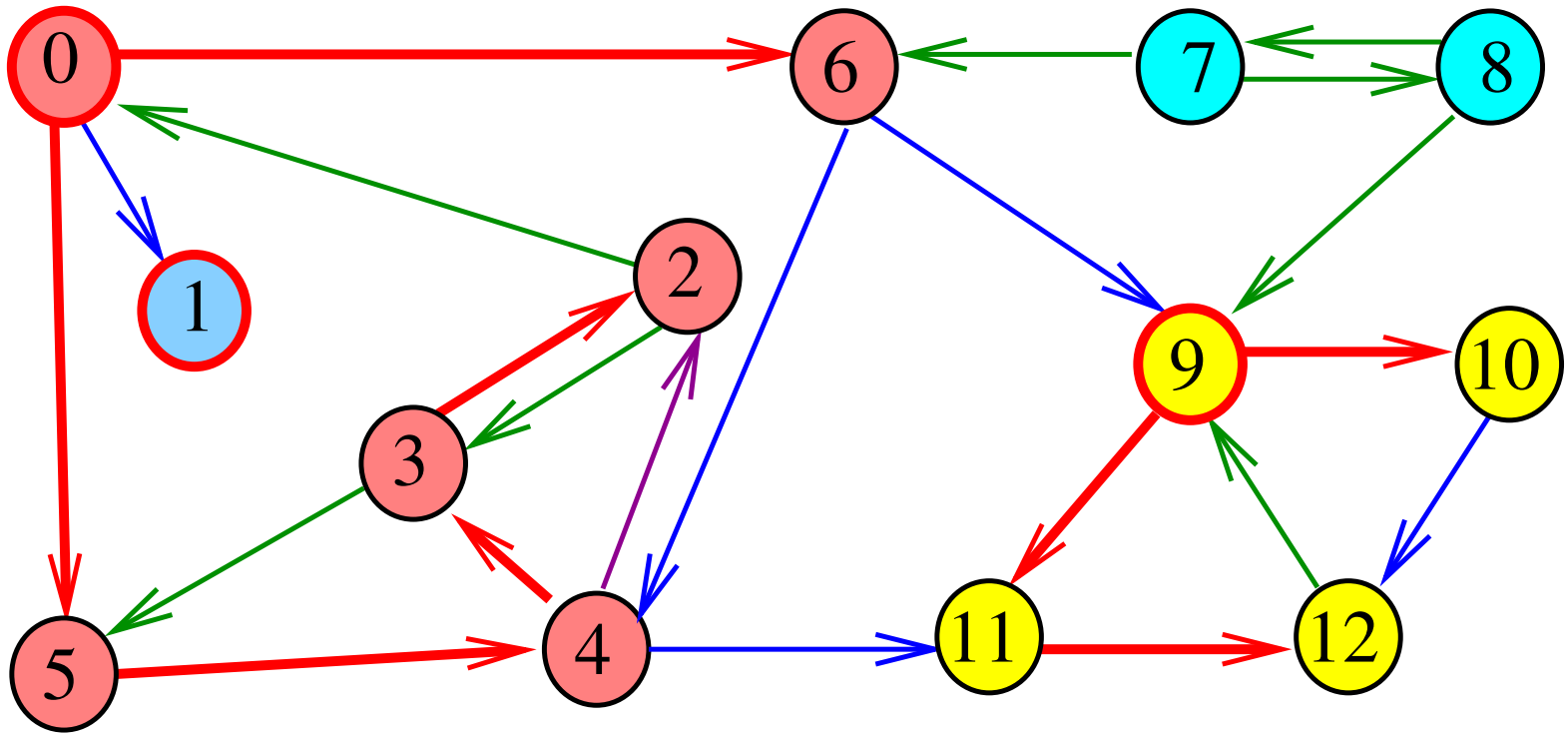
# Ideia . . . G e DFS

Visitar as componentes numa ordem topológica do digrafo das componentes...



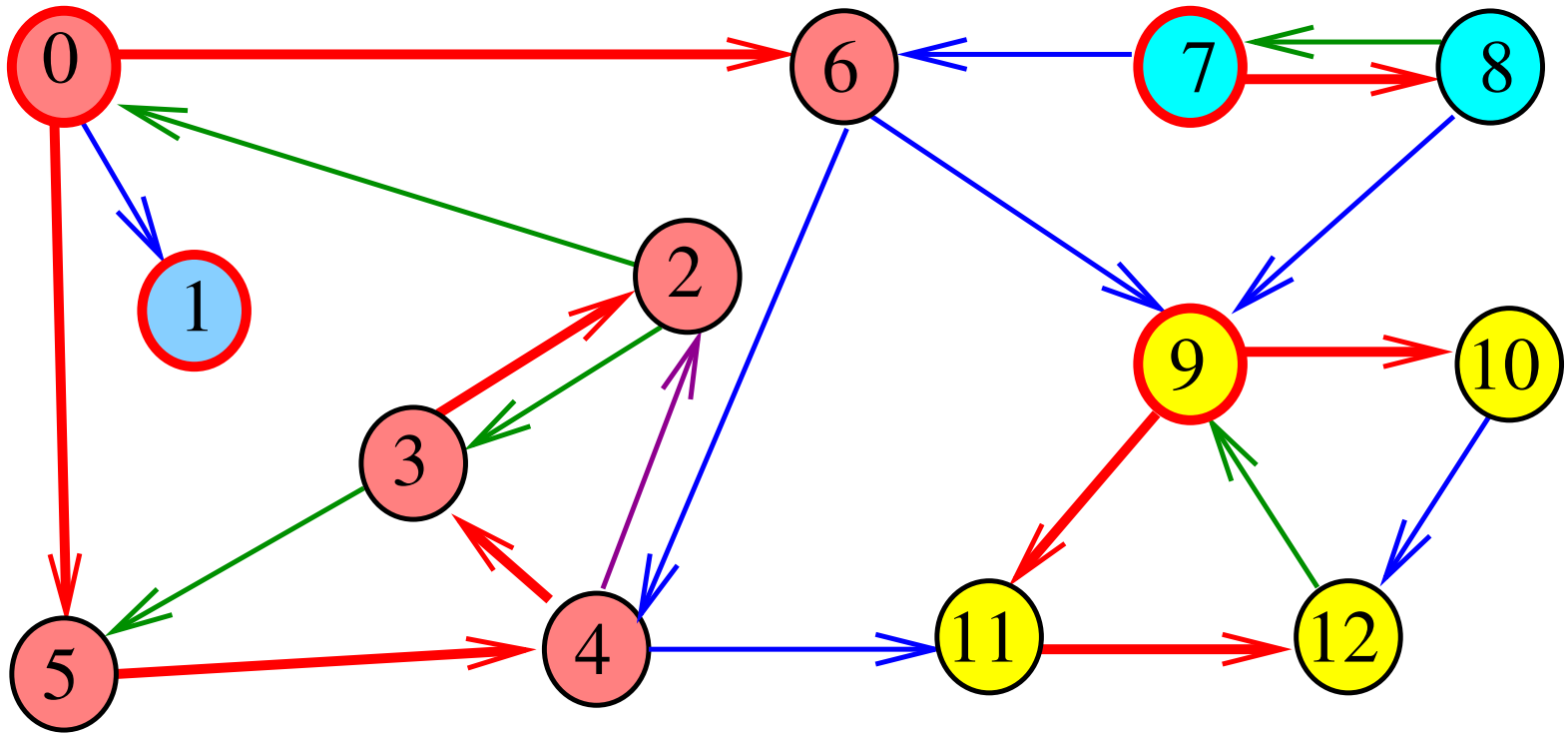
# Ideia . . . G e DFS

Visitar as componentes numa ordem topológica do digrafo das componentes...



# Ideia . . . G e DFS

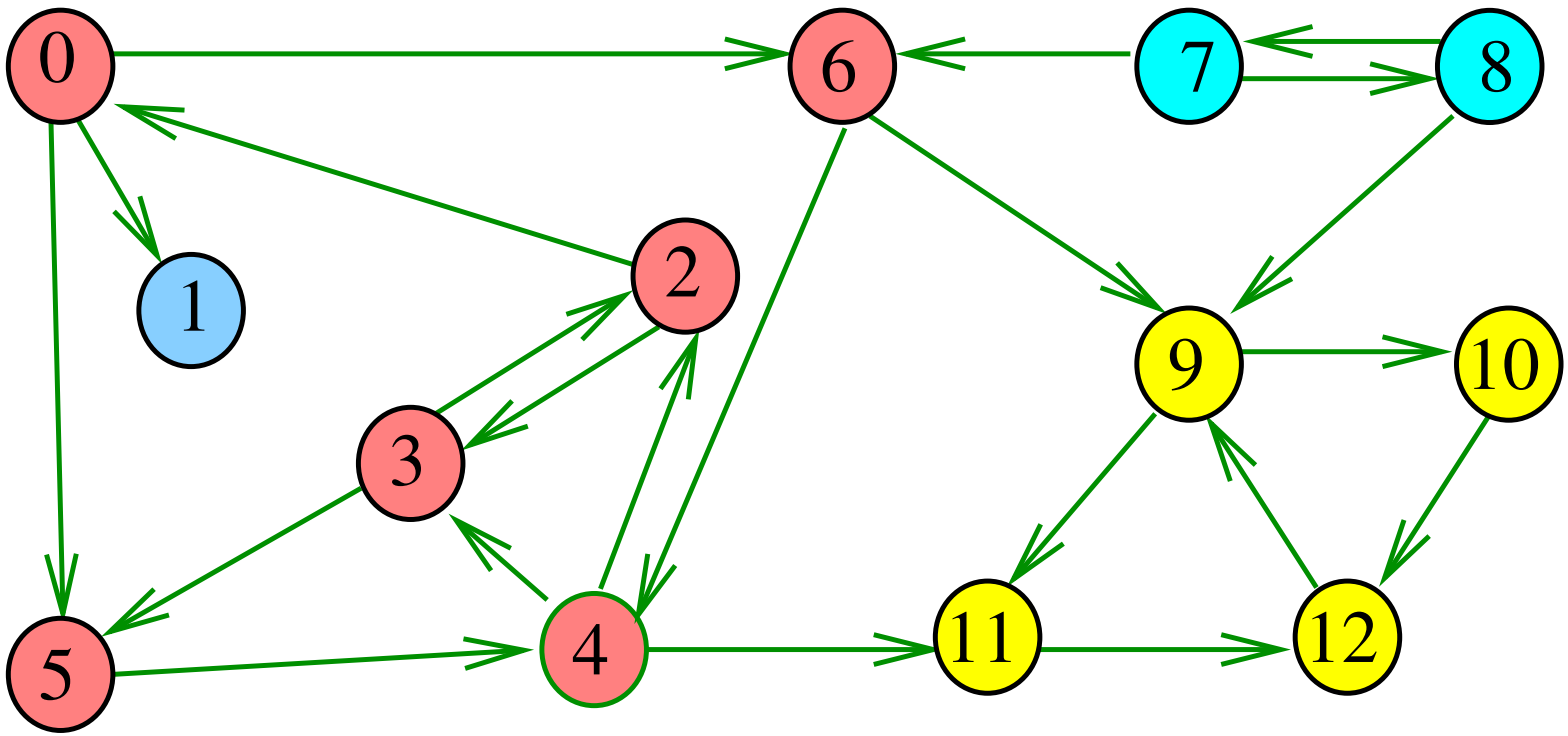
Visitar as componentes numa ordem topológica do digrafo das componentes...



# Propriedade

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

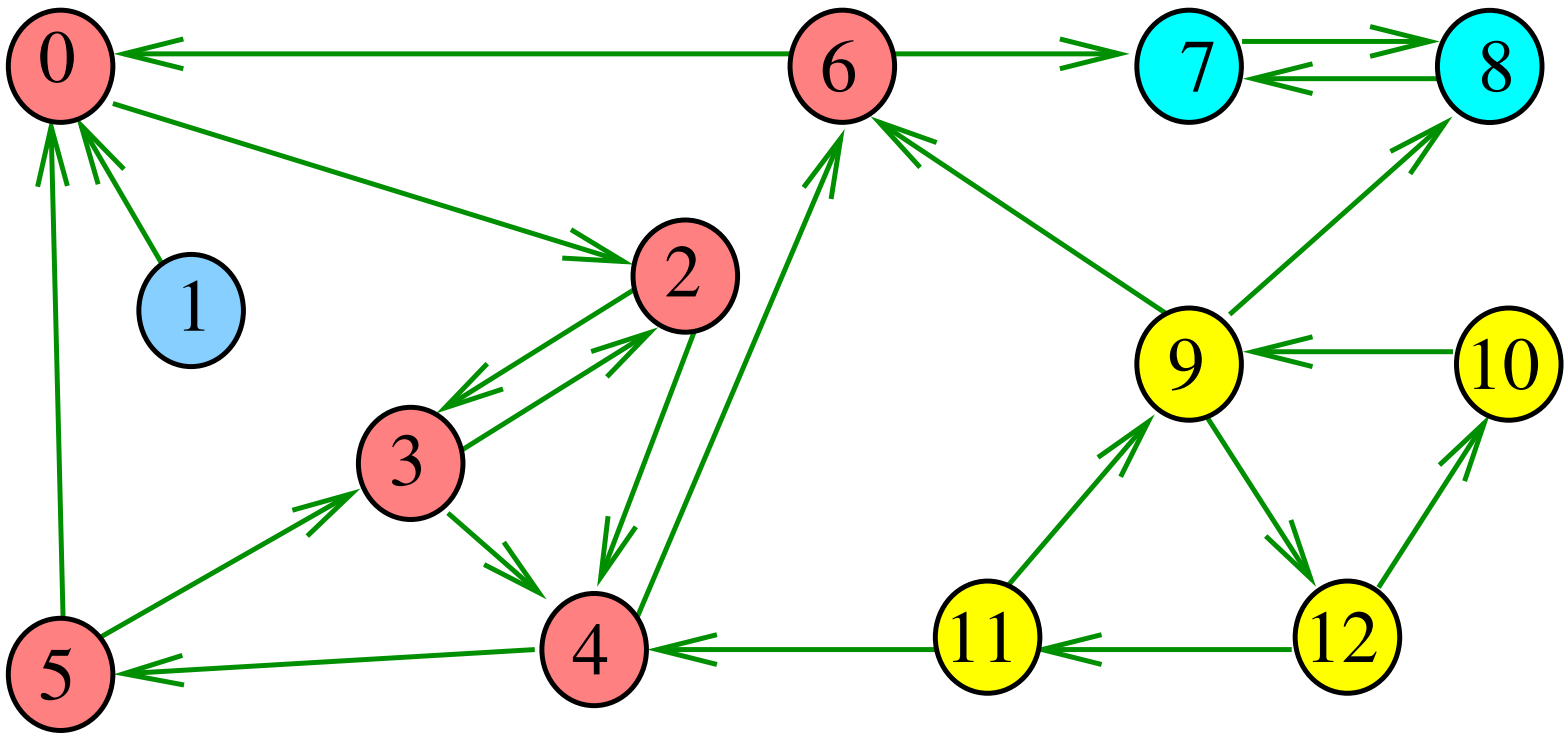
Exemplo: Digrafo G



# 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[]

pós[v] = numeração pós-ordem de v

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

# G, G reverso, DFS e pós[]

pós[v] = numeração pós-ordem de v

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

Em outras palavras:

**Fato.** Se pós[v] > pós[w] e existe 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

# 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
  (em ordem reversa de pós[]);
- determine o conjunto
  $W = \{w : \text{existe caminho de } v \text{ a } w \text{ em } G\}$;
- 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;

# 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
  (em ordem reversa de pós[]);
- determine o conjunto
  $W = \{w : \text{existe caminho de } v \text{ a } w \text{ em } G\}$;
- 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 que ...