

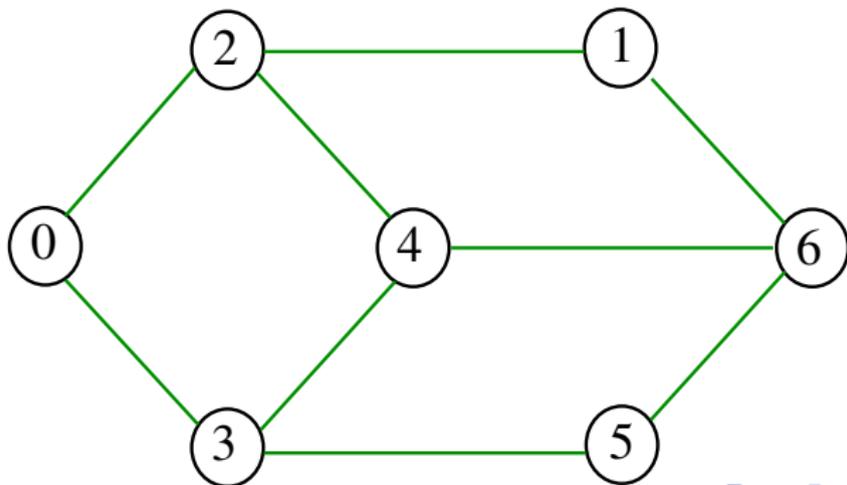
# Grafos bipartidos e ciclos ímpares

S 18.5

# 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

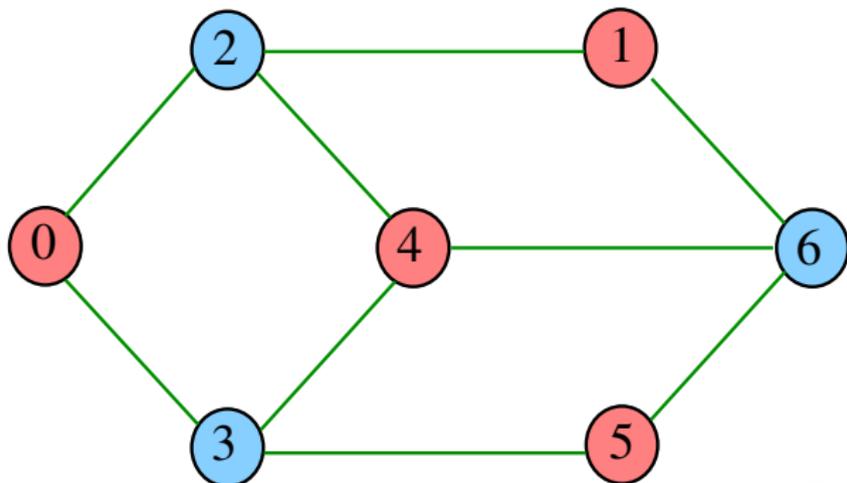
Exemplo:



# 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

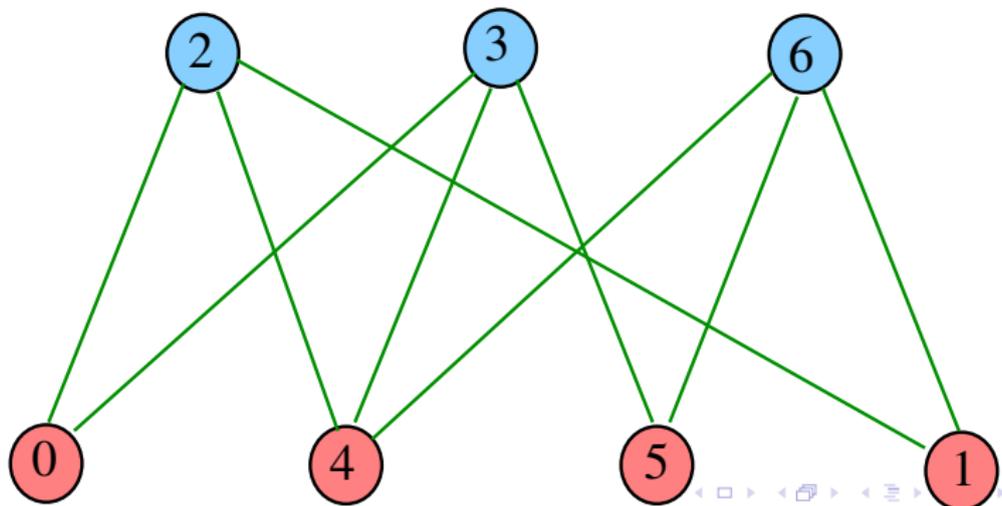
Exemplo:



# 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:



# Florestas e bipartições

## Teorema 1.

*Toda floresta é um grafo bipartido.*

Prova: Basta mostrar que toda árvore é.

Escolha um vértice  $v$ , e, para todo vértice  $w$ , faça:

$$\text{cor}[w] = \text{dist}(v, w) \bmod 2.$$

Se  $u$  e  $w$  são adjacentes, o caminho de  $v$  ao mais distante passa pelo mais próximo e usa a aresta. Assim, as distâncias a  $v$  diferem de 1, e eles têm cores diferentes.

# Florestas e bipartições

## Teorema 1.

*Toda floresta é um grafo bipartido.*

**Prova:** Basta mostrar que toda árvore é.

Escolha um vértice  $v$ , e, para todo vértice  $w$ , faça:

$$\text{cor}[w] = \text{dist}(v, w) \bmod 2.$$

Se  $u$  e  $w$  são adjacentes, o caminho de  $v$  ao mais distante passa pelo mais próximo e usa a aresta. Assim, as distâncias a  $v$  diferem de 1, e eles têm cores diferentes.

## GRAPHtwocolor

Supomos que nossos grafos têm no máximo  $\text{maxV}$  vértices

```
int color[maxV];
```

A função devolve **1** se o grafo **G** é bipartido e devolve **0** em caso contrário

Se **G** é **bipartido**, a função atribui uma "cor" a cada vértice de **G** de tal forma que toda aresta tenha **pontas de cores diferentes**

As cores dos vértices, **0** e **1**, são registradas no vetor color indexado pelos vértices

```
int GRAPHtwocolor (Graph G);
```

## GRAPHtwocolor

```
int GRAPHtwocolor (Graph G) {  
    Vertex v;  
1   for (v = 0; v < G->V; v++)  
2       color[v] = -1;  
3   for (v = 0; v < G->V; v++)  
4       if (color[v] == -1)  
5           if (dfsRclr(G, v, 0) == 0)  
6               return 0;  
7   return 1;  
}
```

## dfsRclr

```
int dfsRclr(Graph G, Vertex v, int c){
    link p;
1   color[v] = 1-c;
2   for (p=G->adj[v];p!=NULL;p=p->next) {
3       Vertex w = p->w;
4       if (color[w]==1-c || /*ciclo ímpar*/
5           color[w]==-1 && !dfsRclr(G,w,1-c))
6           return 0;
    }
7   return 1;
}
```

# Consumo de tempo

O consumo de tempo da função `GRAPHtwocolor` para **vetor de listas de adjacência** é  $O(V + A)$ .

# Conclusão

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

- $G$  possui um ciclo ímpar
- $G$  é bipartido

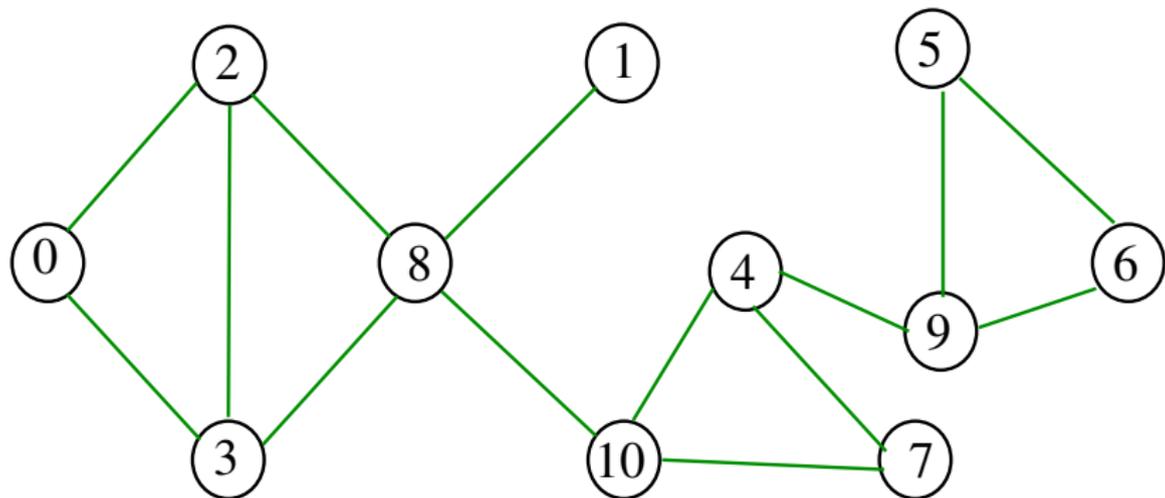
# Pontes em grafos e aresta-biconexão

## S 18.6

## Pontes em grafos

Uma aresta de um grafo é uma **ponte** (= *bridge* = *separation edge*) se ela é a única aresta que atravessa algum corte do grafo.

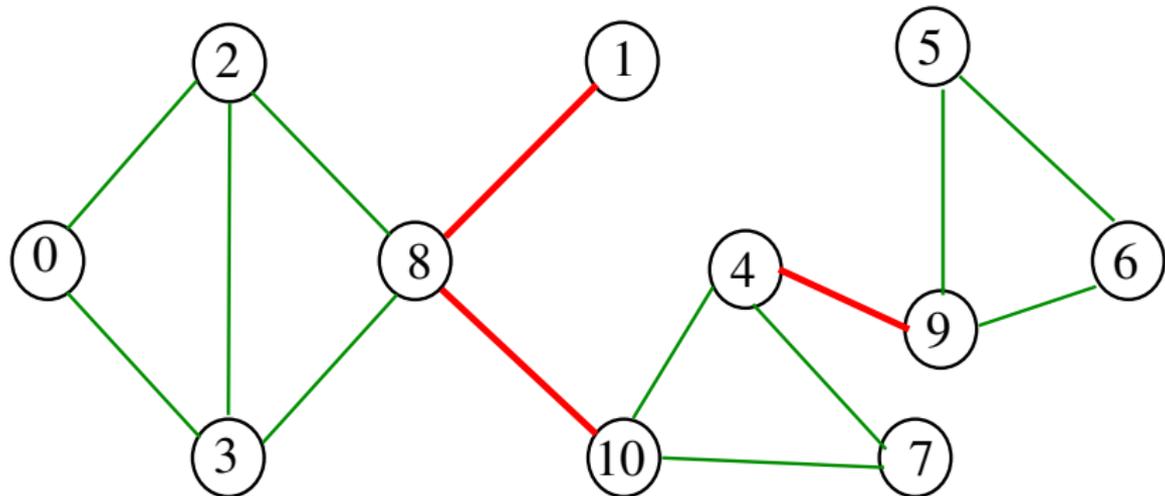
Exemplo:



## Pontes em grafos

Uma aresta de um grafo é uma **ponte** (= *bridge* = *separation edge*) se ela é a única aresta que atravessa algum corte do grafo.

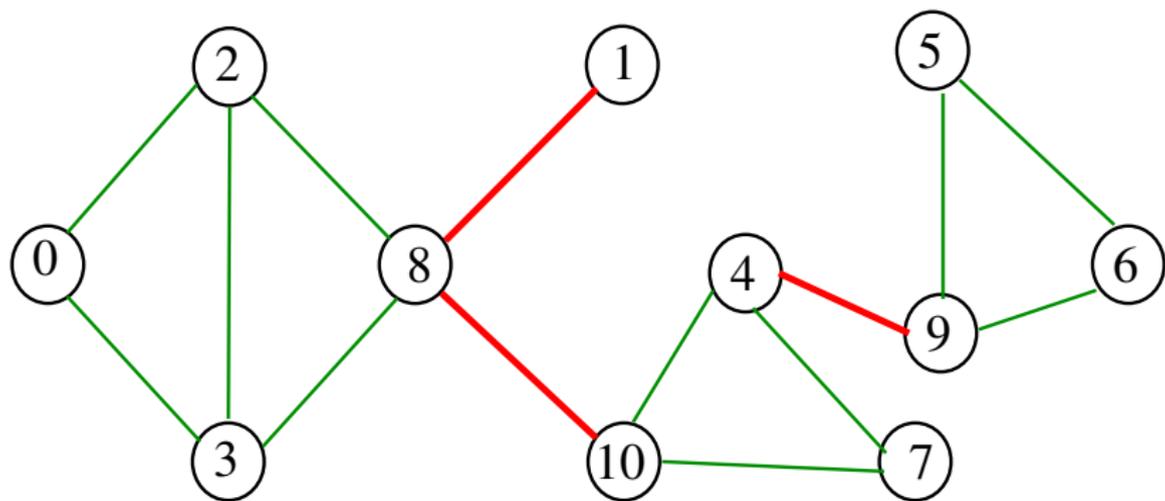
**Exemplo:** as arestas em **vermelho** são pontes



# Procurando pontes

**Problema:** encontrar as pontes de um grafo dado

**Exemplo:** as arestas em **vermelho** são pontes



# all\_bridges1

Recebe um grafo  $G$  e calcula o número `bcnt` de pontes do grafo  $G$  e imprime todas as pontes.

```
void all_bridges1 (Graph G);
```

# Primeiro algoritmo

```
void all_bridges1 (Graph G) {
    Vertex v, w; link p; int ligados;
1   for (v = 0; v < G->V; v++)
2       for(p=G->adj[v];p!=NULL;p=p->next){
3           w = p->w;
4           if (v < w) {
5               GRAPHremoveA(G,w,v);
6               ligados = DIGRAPHpath(G,w,v);
7               GRAPHinsertA(G,w,v);
8               if (!ligados) {
9                   bcnt++;
10                  output(v, w);
                }
            }
        }
    }
}
```

## Consumo de tempo

O consumo de tempo da função `all_bridges1` é  $A/2$  vezes o consumo de tempo da função `DIGRAPHpath`.

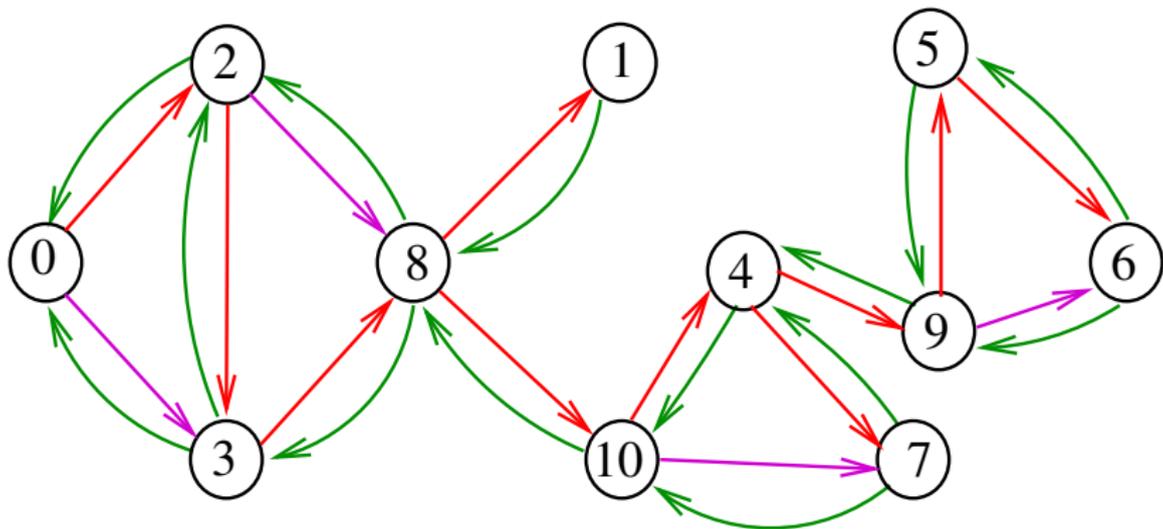
O consumo de tempo da função `all_bridges1` para **vetor de listas de adjacência** é  $O(A(V + A))$ .

O consumo de tempo da função `all_bridges1` para **matriz de adjacência** é  $O(AV^2)$ .

# Pontes e busca em profundidade

Em uma floresta DFS, um dos dois arcos de cada ponte será um arco da **arborescência**

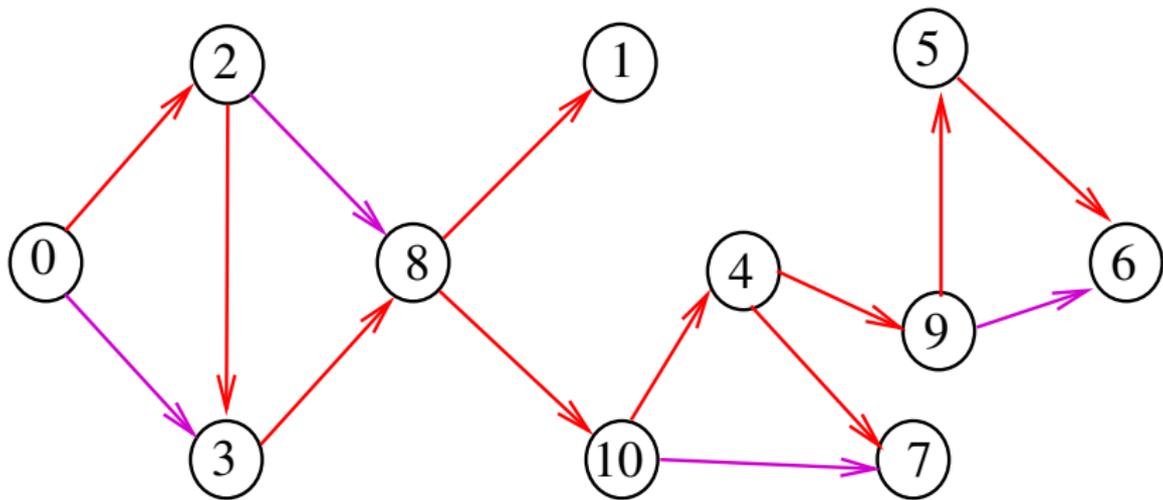
**Exemplo:** arcos em **vermelho** são da arborescência



## Pontes e busca em profundidade

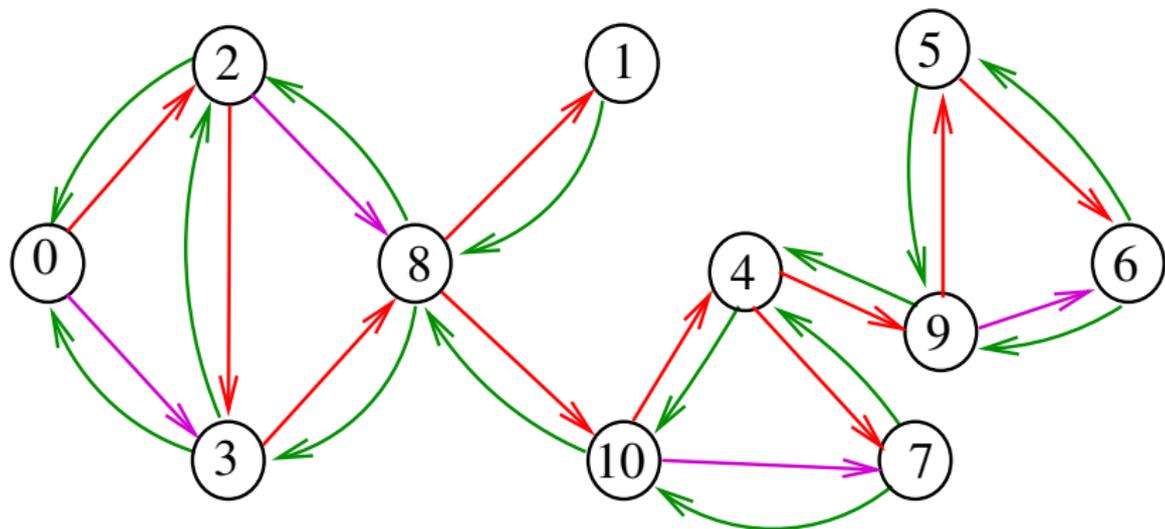
Em uma floresta DFS, um dos dois arcos de cada ponte será um arco da **arborescência**

**Exemplo:** arcos em **vermelho** são da arborescência



## Propriedade

Um arco  $v-w$  da floresta DFS faz parte (juntamente com  $w-v$ ) de uma ponte se e somente se não existe arco de retorno que ligue um descendente de  $w$  a um ancestral de  $v$



## Numeração pré-ordem

$v$	0	1	2	3	4	5	6	7	8	9	10
pre[v]	0	<b>4</b>	1	2	6	8	9	10	3	<b>7</b>	<b>5</b>

