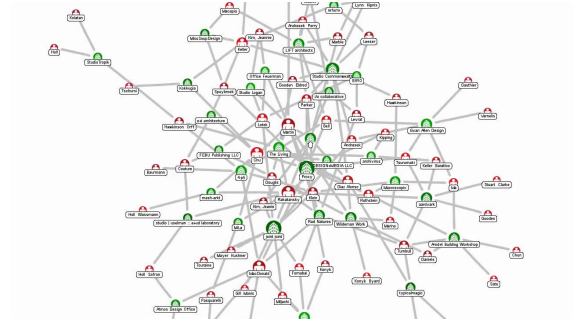


AULA 20

Digrafos



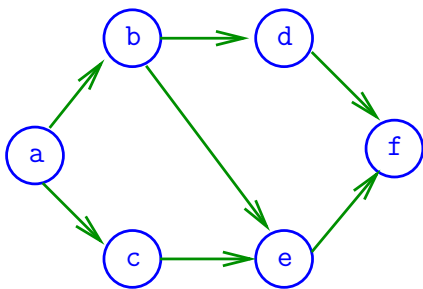
Fonte: Force Directed Graph

Referências: Directed graphs (SW): slides, vídeo.

Digrafos

Um **digrafo** (*directed graph*) consiste de um conjunto de **vértices** (bolas) e um conjunto de **arcos** (flechas)

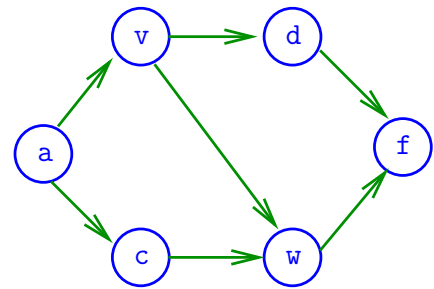
Exemplo: representação de um digrafo



Arcos

Um **arco** é um par ordenado de vértices

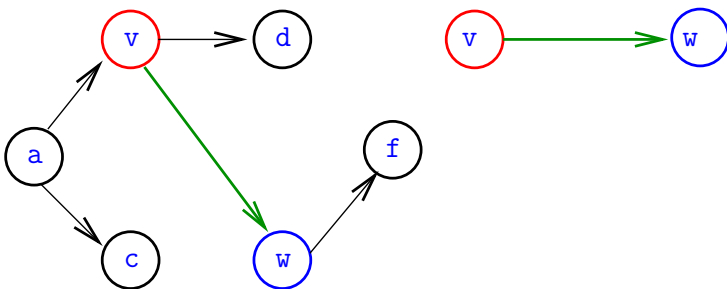
Exemplo: v e w são vértices e $v-w$ é um arco



Ponta inicial e final

Para cada arco $v-w$, o vértice v é a **ponta inicial** e w é a **ponta final**

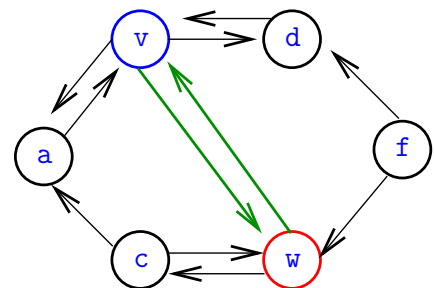
Exemplo: v é ponta inicial e w é ponta final de $v-w$



Arcos anti-paralelos

Dois arcos são **anti-paralelos** se a ponta inicial de um é ponta final do outro

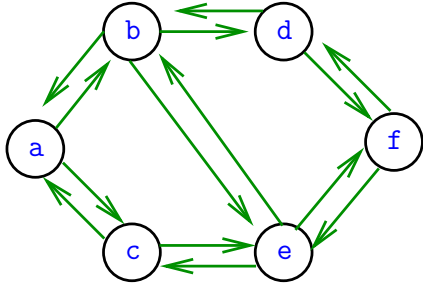
Exemplo: $v-w$ e $w-v$ são anti-paralelos



Digrafos simétricos

Um digrafo é **simétrico** se cada um de seus arcos é anti-paralelo a outro

Exemplo: digrafo simétrico



Navigation icons

Número de arcos

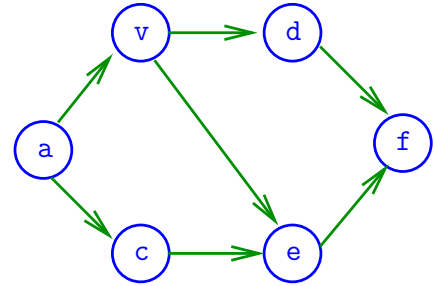
Quantos arcos, no máximo, tem um digrafo com V vértices?

Graus de entrada e saída

grau de entrada de v = no. arcos com **ponta final** v

grau de saída de v = no. arcos com **ponta inicial** v

Exemplo: v tem grau de entrada 1 e de saída 2



Navigation icons

Número de arcos

Quantos arcos, no máximo, tem um digrafo com V vértices?

A resposta é $V \times (V - 1) = \Theta(V^2)$

digrafo **completo** = todo par ordenado de vértices distintos é arco

digrafo **denso** = tem “muitos” muitos arcos

digrafo **esparso** = tem “poucos” arcos

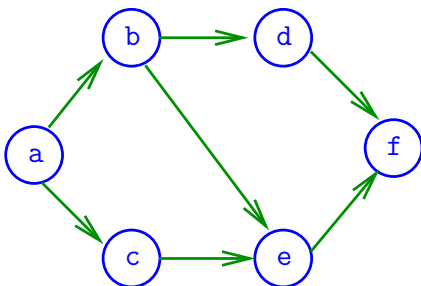
Navigation icons

Navigation icons

Especificação

Digrafos podem ser especificados através de sua lista de arcos

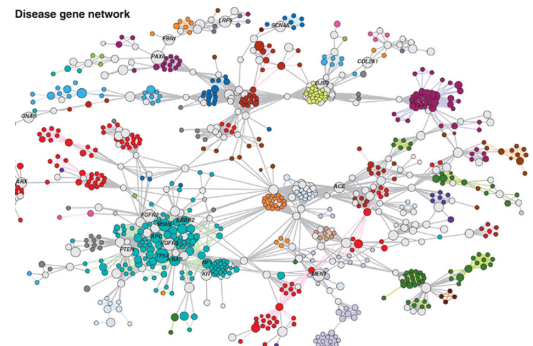
Exemplo:



d-f
b-d
a-c
b-e
e-f
a-b

Navigation icons

Grafos



Fonte: [Scaling Computation of Graph Structured Data with NScale](#)

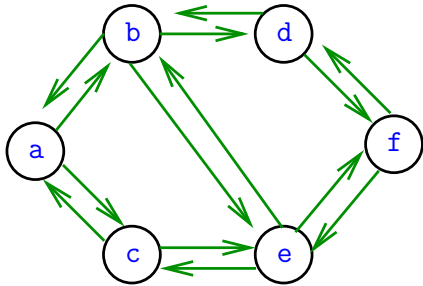
Referências: [Undirected graphs \(SW\)](#): slides, vídeo.

Navigation icons

Grafos

Um **grafo** é um digrafo **simétrico**

Exemplo: um grafo

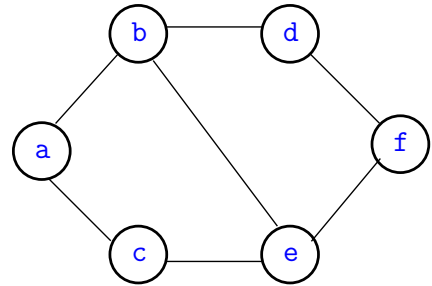


< > < > < > < > < > < > < > < > < >

Grafos

Um **grafo** é um digrafo **simétrico**

Exemplo: representação usual

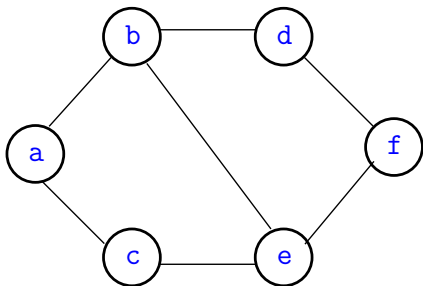


< > < > < > < > < > < > < > < > < >

Arestas

Uma **aresta** é um par de arcos anti-parallelas.

Exemplo: b-a e a-b são a **mesma** aresta

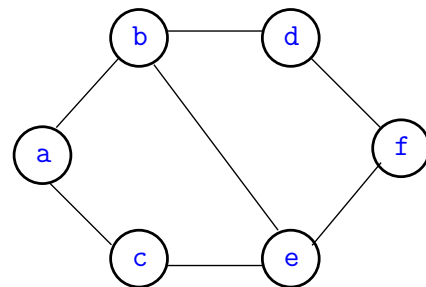


< > < > < > < > < > < > < > < > < >

Especificação

Grafos podem ser especificados através de sua lista de arestas

Exemplo:



f-d
b-d
c-a
e-b
e-f
a-b

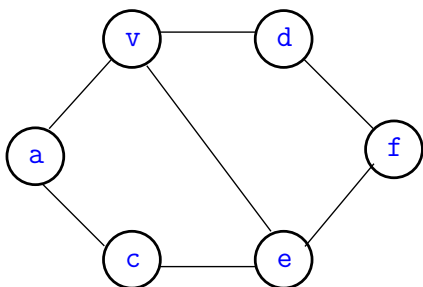
< > < > < > < > < > < > < > < > < >

Graus de vértices

Em um grafo

grau de v = número de arestas com ponta em v

Exemplo: v tem grau 3



< > < > < > < > < > < > < > < > < >

Número de arestas

Quantas arestas, no máximo, tem um grafo com V vértices?

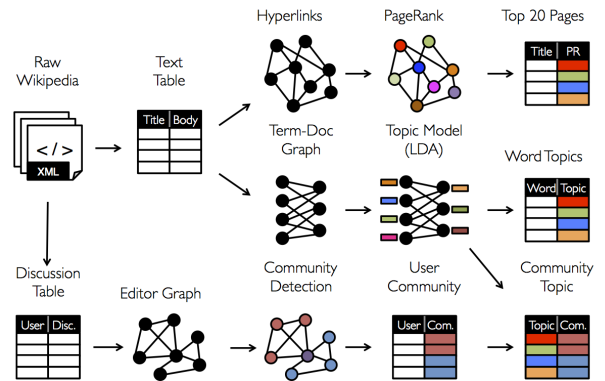
Número de arestas

Quantas arestas, no máximo, tem um grafo com V vértices?

A resposta é $V \times (V - 1) / 2 = \Theta(V^2)$

grafo **completo** = todo par **não**-ordenado de vértices distintos é aresta

Digrafos no computador



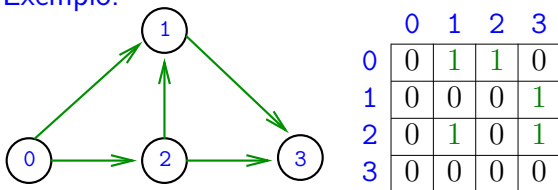
Fonte: GraphX Programming Guide

Matriz de adjacência de digrafos

Matriz de adjacência de um digrafo tem linhas e colunas indexadas por vértices:

$adj[v][w] = 1$ se $v \rightarrow w$ é um arco
 $adj[v][w] = 0$ em caso contrário

Exemplo:



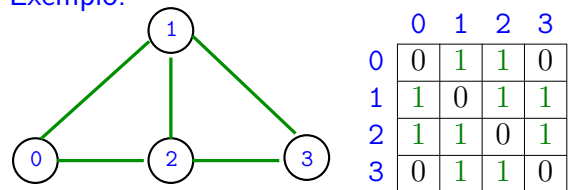
Consumo de espaço: $\Theta(V^2)$ fácil de implementar

Matriz de adjacência de grafos

Matriz de adjacência de um grafo tem linhas e colunas indexadas por vértices:

$adj[v][w] = 1$ se $v-w$ é um aresta
 $adj[v][w] = 0$ em caso contrário

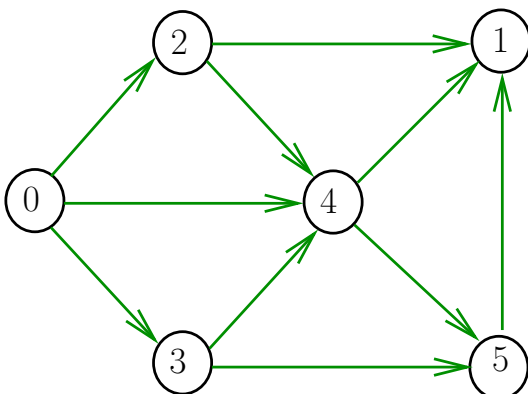
Exemplo:



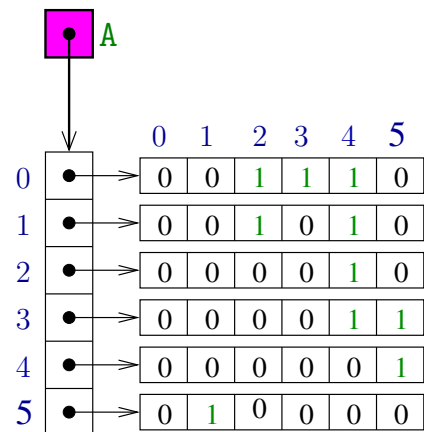
Consumo de espaço: $\Theta(V^2)$ fácil de implementar

Digrafo

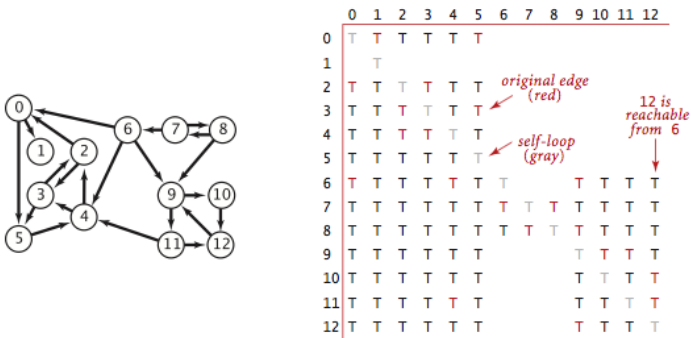
Digraph G



Estruturas de dados

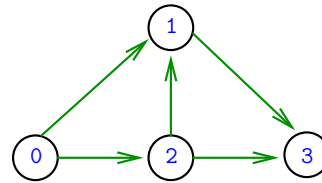


Digrafos no algs4



Matriz de incidência de digrafos

Uma **matriz de incidências** de um digrafo tem **linhas** indexadas por **vértices** e **colunas** por **arcos** e cada entrada $[k][vw]$ é -1 se $k = v$, $+1$ se $k = w$, e 0 em caso contrário.



	0-1	0-2	2-1	2-3	1-3
0	-1	-1	0	0	0
1	+1	0	+1	0	-1
2	0	+1	-1	-1	0
3	0	0	0	+1	+1

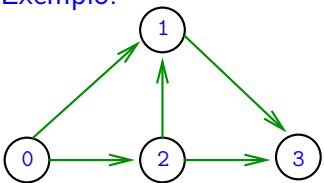
Consumo de espaço: $\Theta(nm)$

Interessante do ponto de vista de **otimização linear**.

Vetor de listas de adjacência de digrafos

Na representação de um digrafo através de **listas de adjacência** tem-se, para cada vértice v , uma lista dos vértices que são vizinhos v .

Exemplo:



- 0: 1, 2
- 1: 3
- 2: 1, 3
- 3:

Consumo de espaço: $\Theta(V + A)$

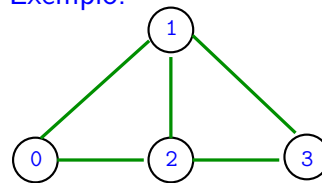
(linear)

Manipulação eficiente

Vetor de lista de adjacência de grafos

Na representação de um grafo através de **listas de adjacência** tem-se, para cada vértice v , uma lista dos vértices que são pontas de arestas incidentes a v .

Exemplo:



- 0: 1, 2
- 1: 3, 0, 2
- 2: 1, 3, 0
- 3: 1, 2

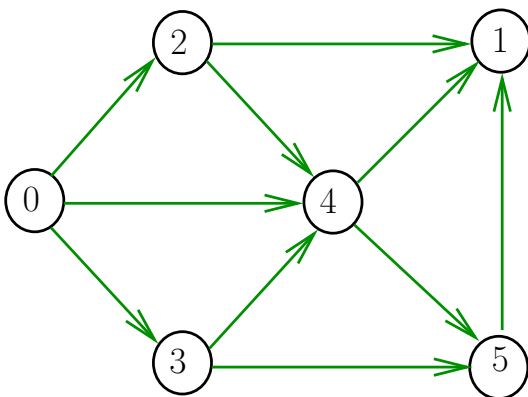
Consumo de espaço: $\Theta(V + A)$

(linear)

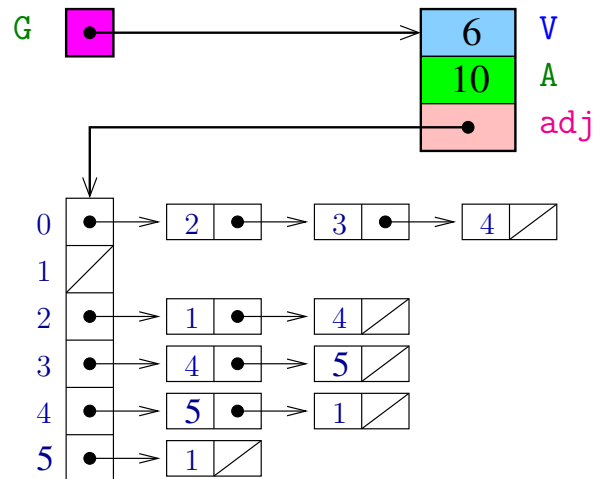
Manipulação eficiente

Digrafo

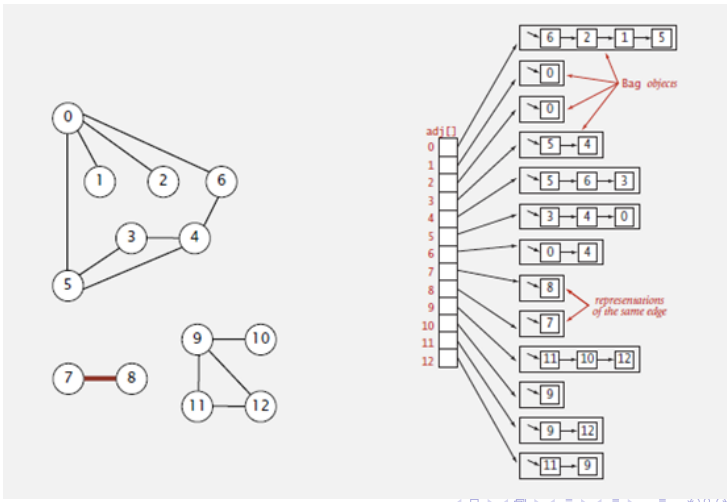
Digraph G



Estruturas de dados



Grafos no algs4



Digraph

```
public Digraph(int V) {
    this.V = V;
    this.E = 0;
    indegree = new int[V];
    adj = (Bag<Integer>[] ) new Bag[V];
    for (int v = 0; v < V; v++) {
        adj[v] = new Bag<Integer>();
    }
}
```

Digraph

```
// retorna o grau de saída de v
public int outdegree(int v) {
    return adj[v].size();
}
// retorna o grau de entrada de v
public int indegree(int v) {
    return indegree[v];
}
```

Esqueleto da classe Digraph

```
public class Digraph {
    private int V; // no. vértices
    private int E; // no. arcos
    private Bag<Integer>[] adj;
    private int[] indegree;
    public Digraph(int V) {...}
    public int V() { return V; }
    public int E() { return E; }
    public void addEdge(int v, int w) { }
    public Iterable<Integer> adj(int v) { }
    public int outdegree(int v) {...}
    public int indegree(int v) {...}
    public Digraph reverse() { ...}
}
```

Digraph

```
// insere um arco
public void addEdge(int v, int w) {
    adj[v].add(w);
    indegree[w]++;
    E++;
}
// retorna a lista de adjacência de v
public Iterable<Integer> adj(int v) {
    return adj[v];
}
```

Digraph

```
// retorna o sigrafo reverso
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;
}
```

Caminhos em digrafos

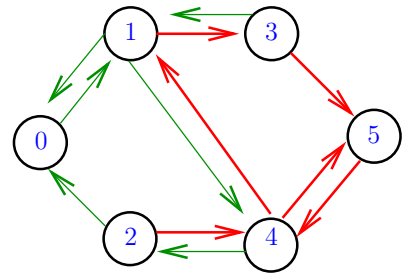


Fonte: Finding Your Way & Making You A Priority

Caminhos

Um **caminho** num digrafo é qualquer seqüência da forma $v_0-v_1-v_2-\dots-v_{k-1}-v_k$, onde $v_{k-1}-v_k$ é um arco para $k = 1, \dots, p$.

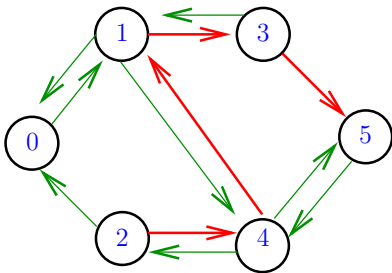
Exemplo: 2-4-1-3-5-4-5 é um caminho com **origem** 2 é **término** 5



Caminhos simples

Um caminho é **simples** se não tem vértices repetidos

Exemplo: 2-4-1-3-5 é um caminho simples de 2 a 5



Procurando caminhos

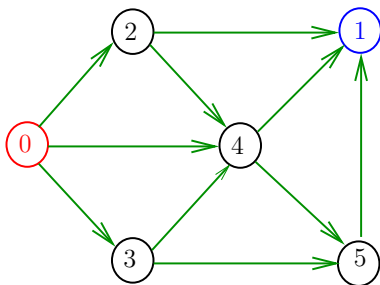


Fonte: Minecraft maze created by Carl Eklof (algs4)

Procurando um caminho

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

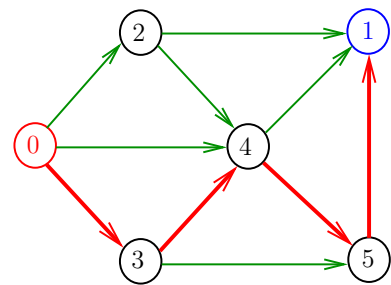
Exemplo: para $s = 0$ e $t = 1$ a resposta é SIM



Procurando um caminho

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

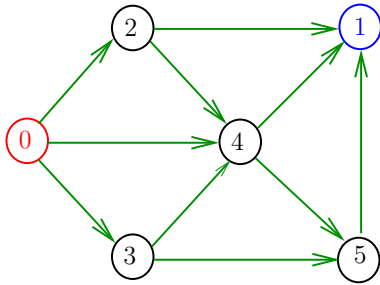
Exemplo: para $s = 0$ e $t = 1$ a resposta é SIM



Procurando um caminho

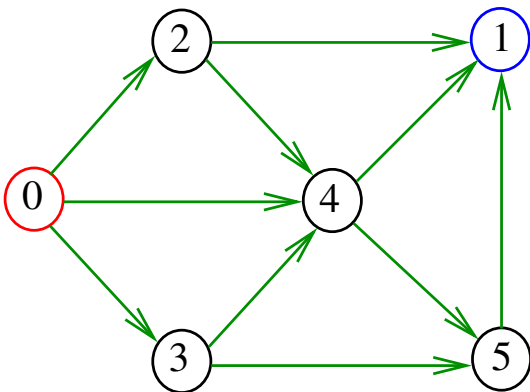
Problema: dados um digrafo G e dois vértices s e t decidir se existe um caminho de s a t

Exemplo: para $s = 5$ e $t = 4$ a resposta é **NÃO**



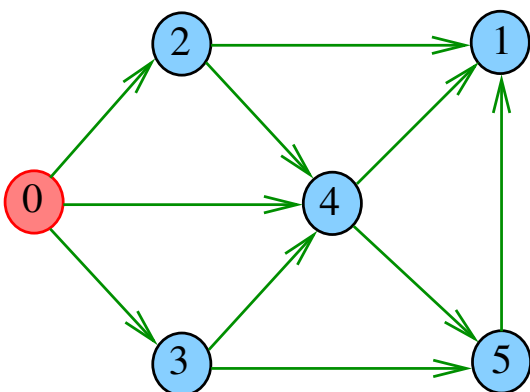
Navigation icons

DFSpaths($G, 0$)



Navigation icons

dfs($G, 0$)



Navigation icons

DFSpaths

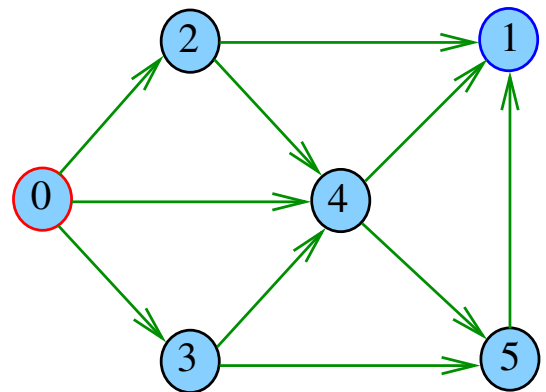
A classe **DFSpaths** recebe um digrafo G e um vértice s e **determina** todos os vértices alcançáveis a partir de s .

A classe implementa a técnica chamada **busca em profundidade** (*Depth First Search*).

```
public class DFSpaths{
    public void DFSpaths(Digraph G,int s){}
    // retorna true se há caminho de s a v
    public boolean hasPath(int v){ ...}
    private void dfs(Digraph G, int v) { }
}
```

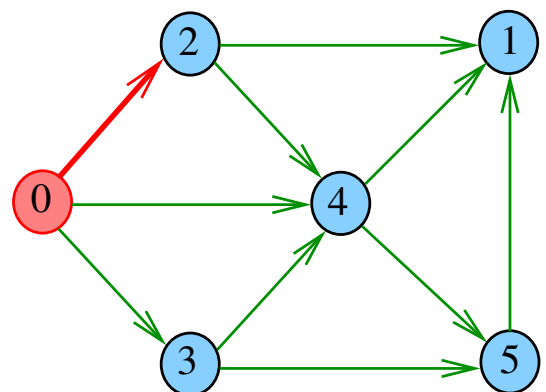
Navigation icons

DFSpaths($G, 0$)



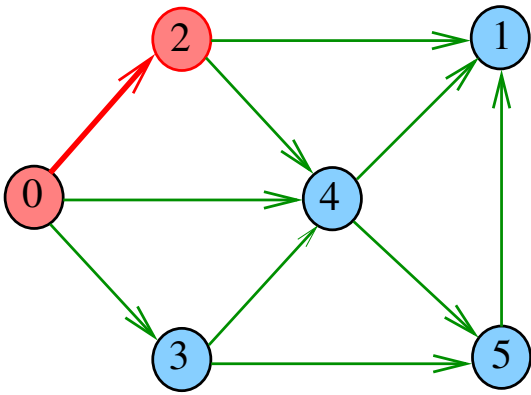
Navigation icons

dfs($G, 0$)

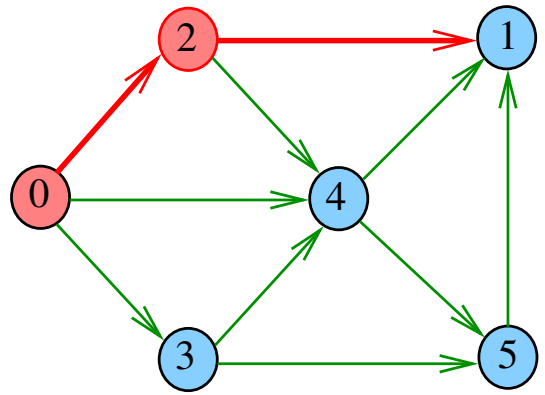


Navigation icons

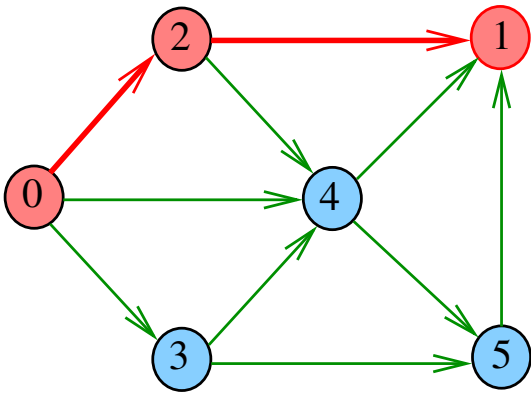
dfs(G, 2)



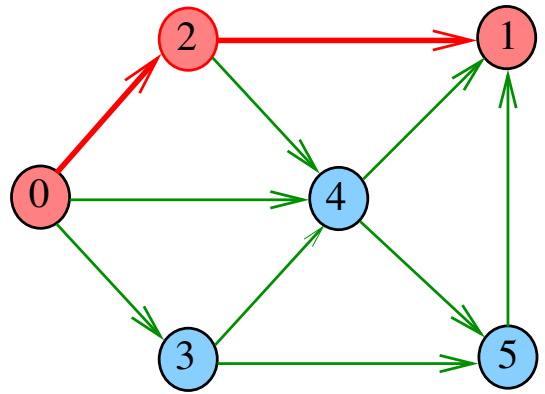
dfs(G, 2)



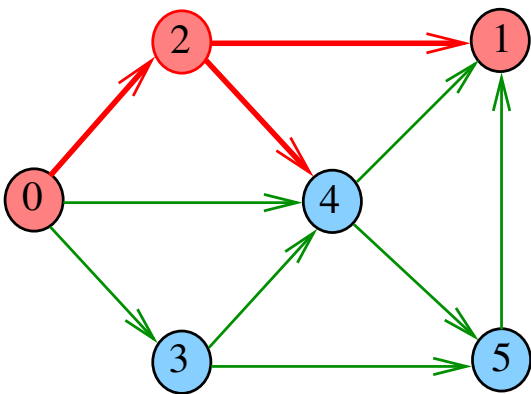
dfs(G, 1)



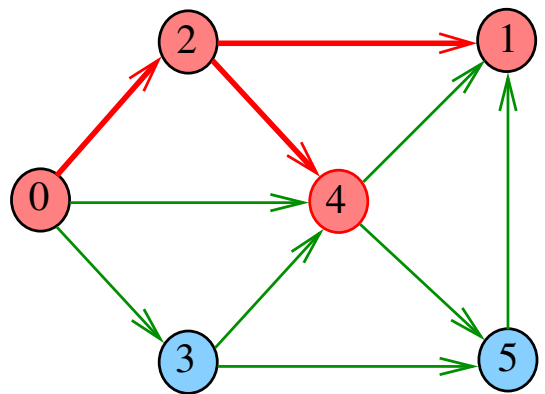
dfs(G, 2)



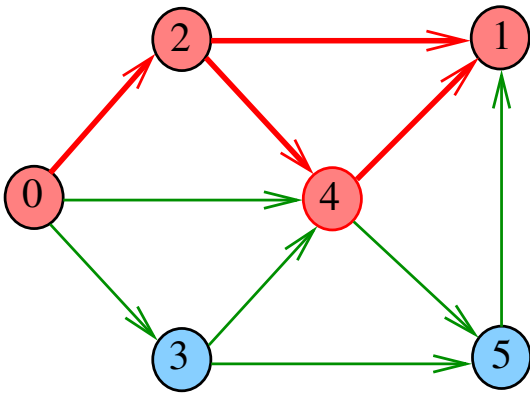
dfs(G, 2)



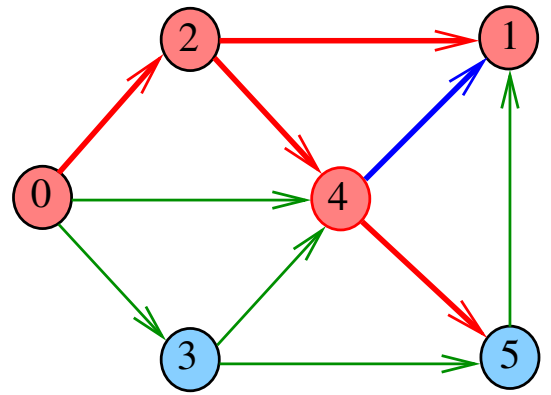
dfs(G, 4)



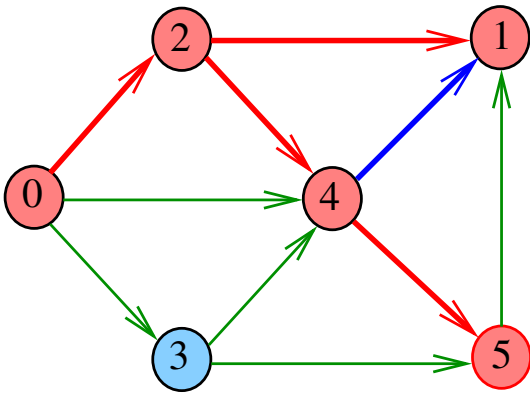
dfs(G, 4)



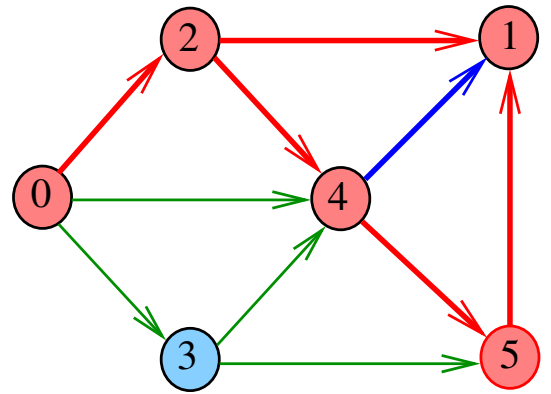
dfs(G, 4)



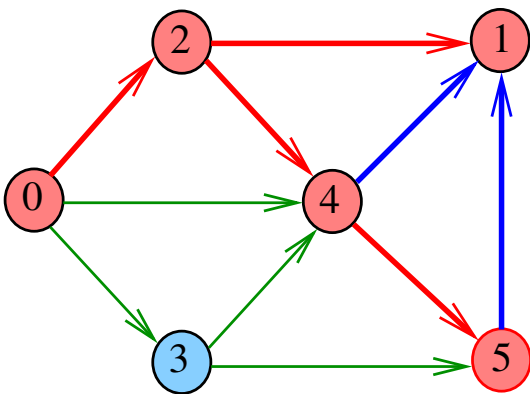
dfs(G, 5)



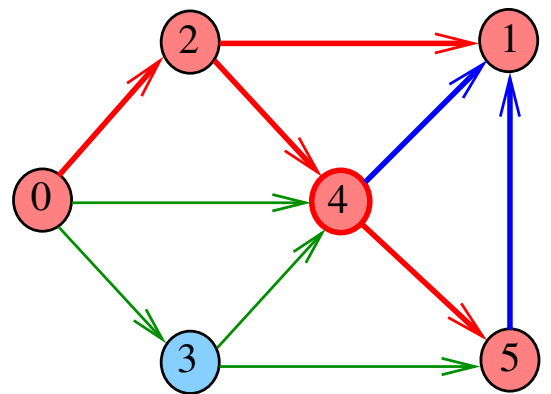
dfs(G, 5)



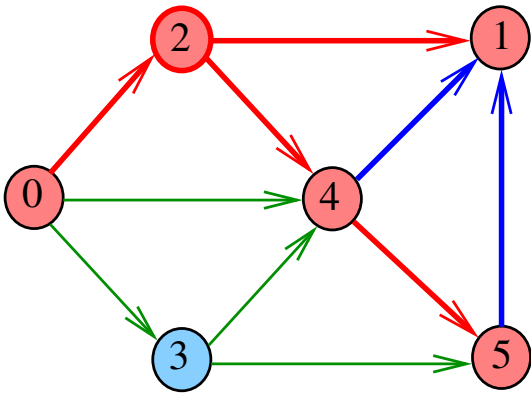
dfs(G, 5)



dfs(G, 4)

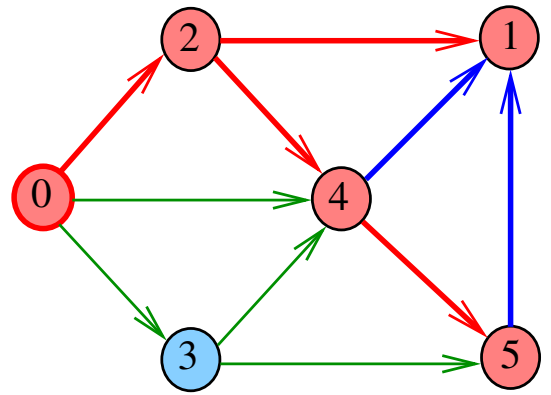


dfs(G, 2)



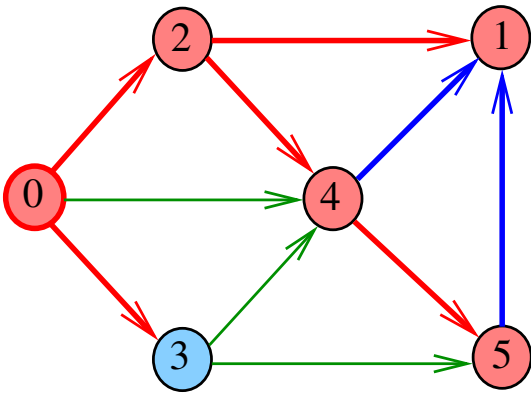
< > < > < > < > < > < >

dfs(G, 0)



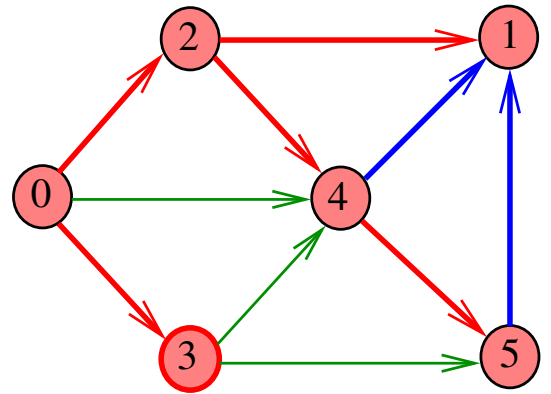
< > < > < > < > < > < >

dfs(G, 0)



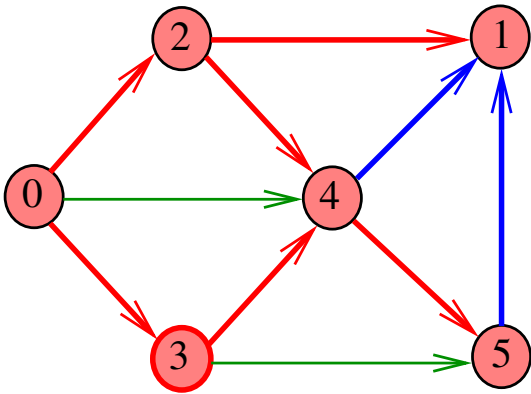
< > < > < > < > < > < >

dfs(G, 3)



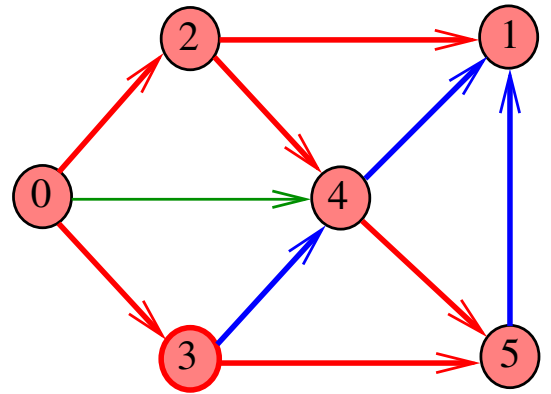
< > < > < > < > < > < >

dfs(G, 3)



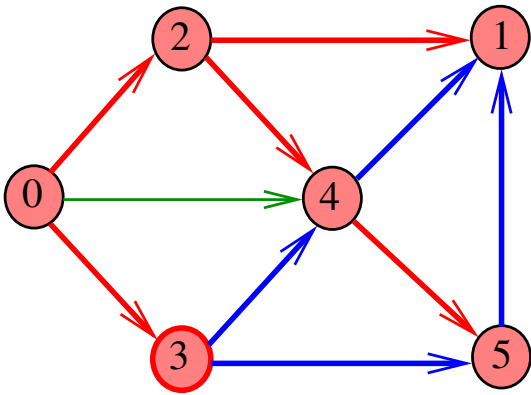
< > < > < > < > < > < >

dfs(G, 3)



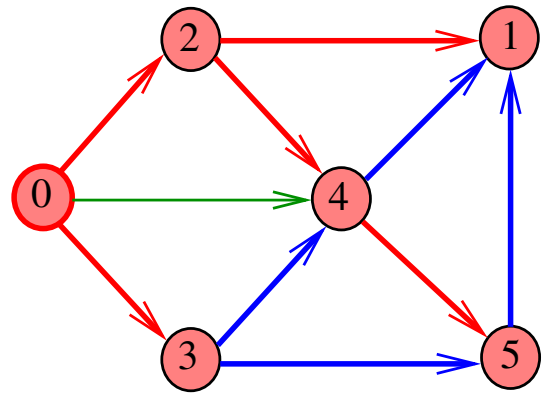
< > < > < > < > < > < >

dfs(G, 3)



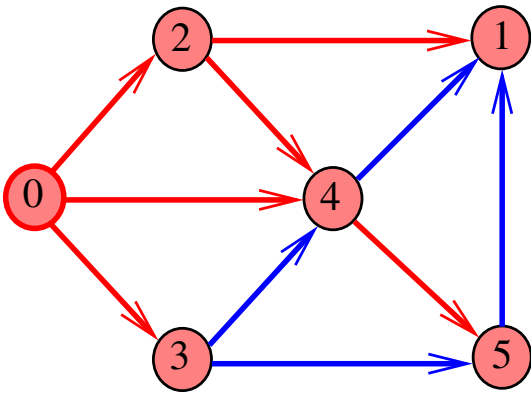
Navigation icons

dfs(G, 0)



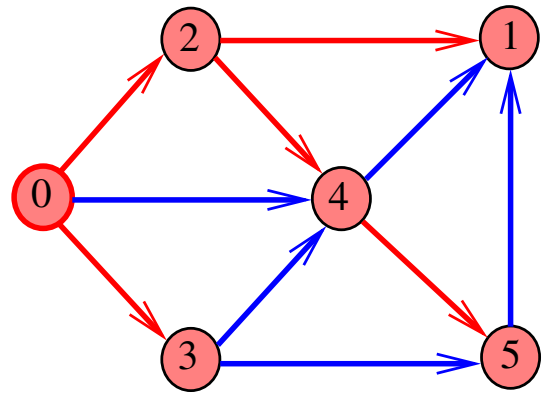
Navigation icons

dfs(G, 0)



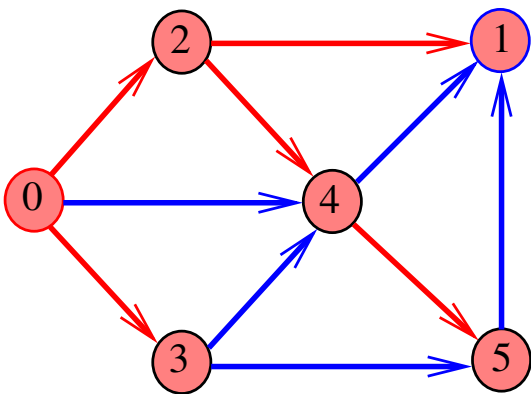
Navigation icons

dfs(G, 0)



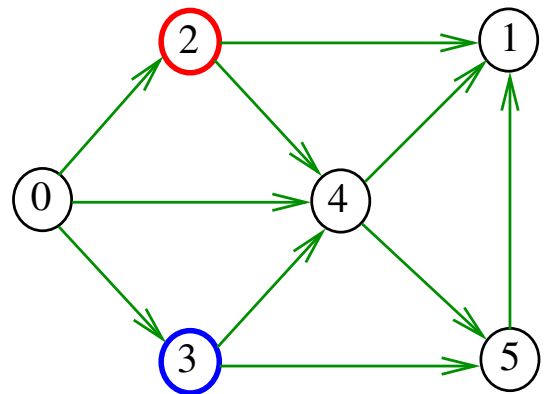
Navigation icons

DFSpaths(G, 0)



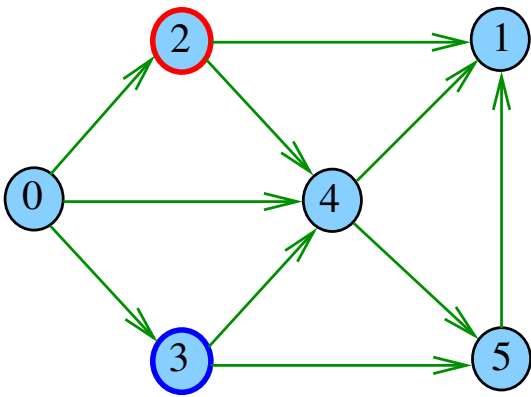
Navigation icons

DFSpaths(G, 2)



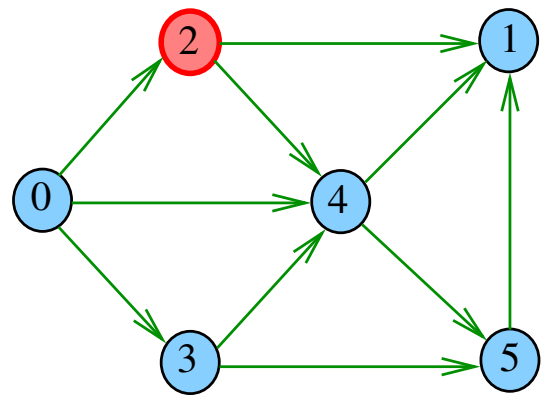
Navigation icons

DFSpaths(G, 2)



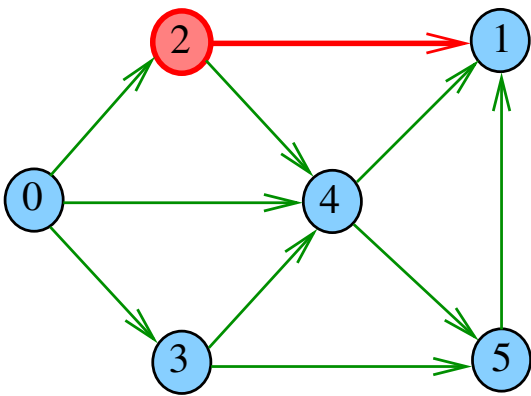
Navigation icons

dfs(G, 2)



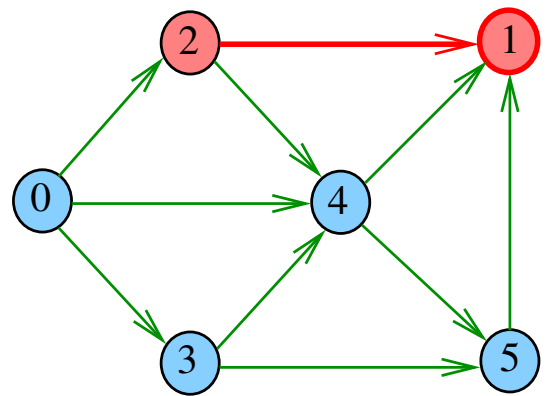
Navigation icons

dfs(G, 2)



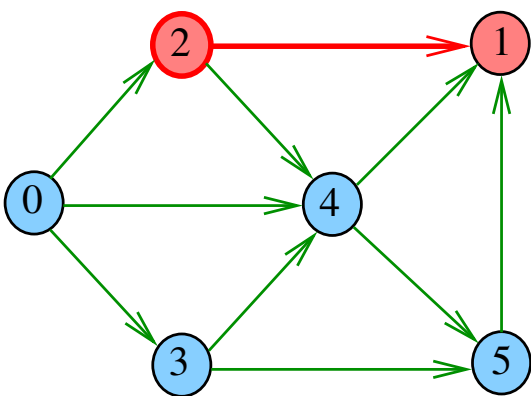
Navigation icons

dfs(G, 1)



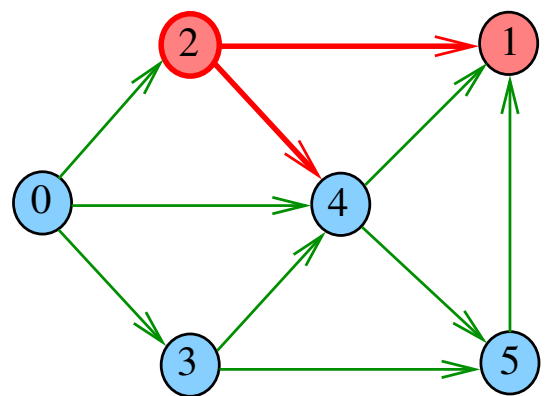
Navigation icons

dfs(G, 2)



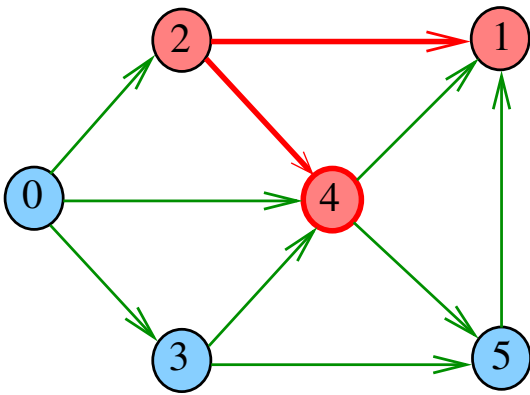
Navigation icons

dfs(G, 2)

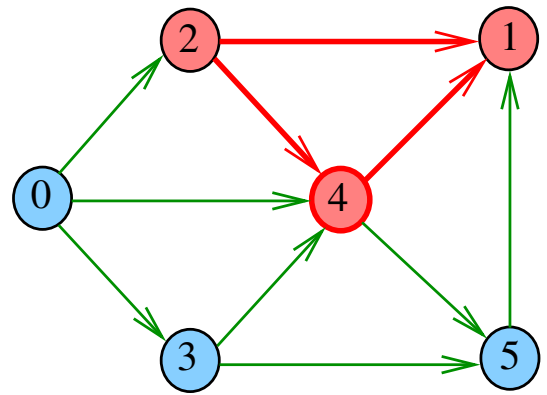


Navigation icons

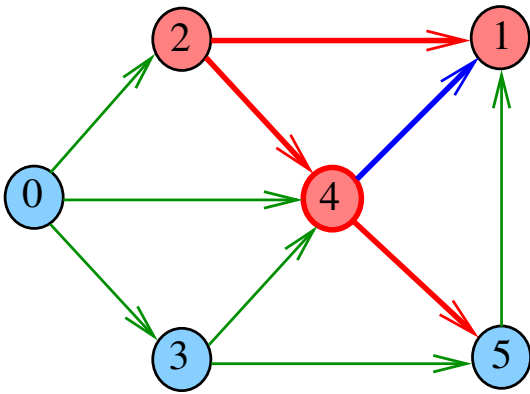
dfs(G, 4)



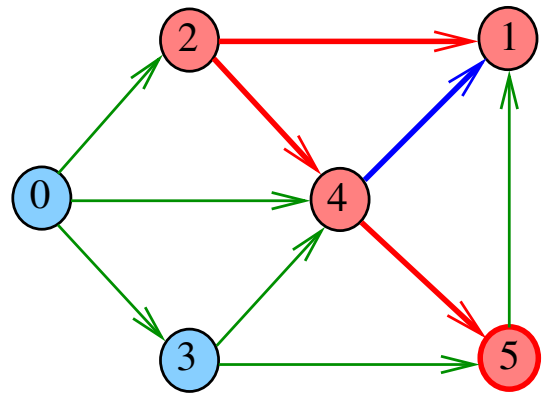
dfs(G, 4)



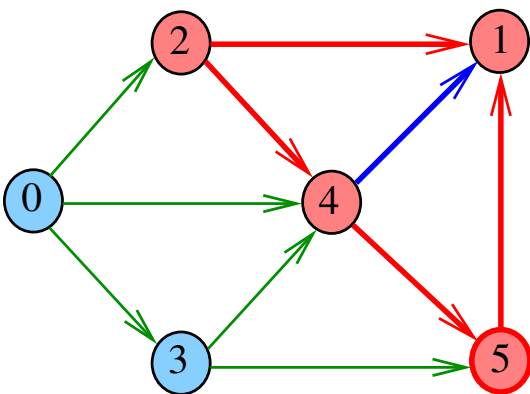
dfs(G, 4)



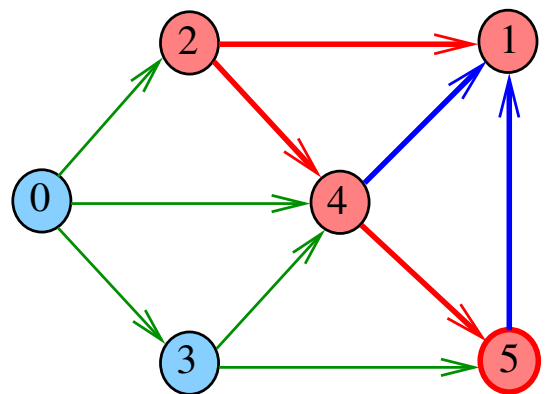
dfs(G, 5)



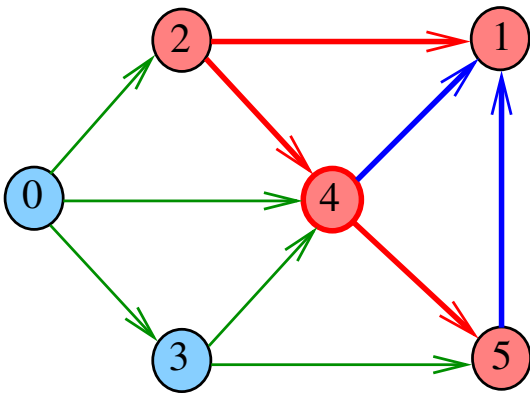
dfs(G, 5)



dfs(G, 5)

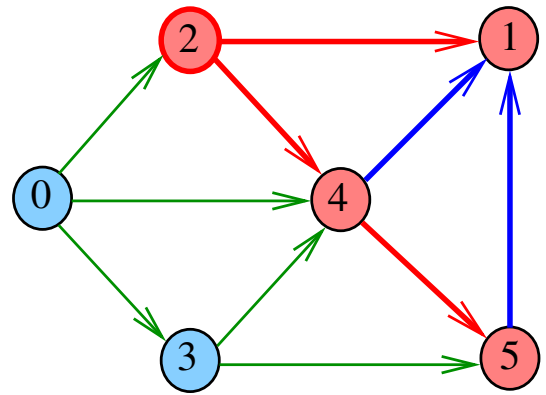


dfs(G, 4)



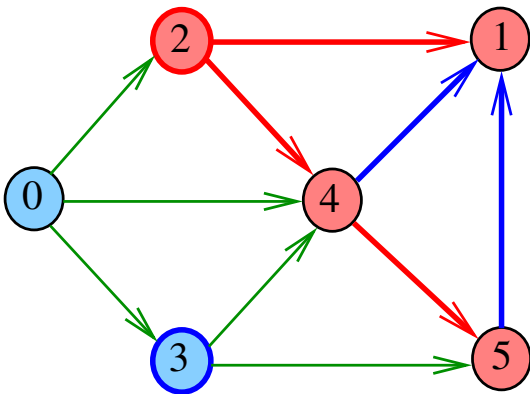
Navigation icons

dfs(G, 2)



Navigation icons

DFSpaths(G, 2)



Navigation icons

DFSpaths

```
public class DFSpaths {
    private final int s;
    private boolean[] marked;
    public DFSpaths(Digraph G, int s) {}
    private void dfs(Digraph G, int v) {}
    public boolean hasPath(int v) {}
}
```

Navigation icons

DFSpaths

DFSpaths

Encontra um caminho de s a todo vértice alcançável a partir de s .

```
public DFSpaths(Digraph G, int s) {
    marked = new boolean[G.V()];
    this.s = s;
    dfs(G, s);
}
```

Navigation icons

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

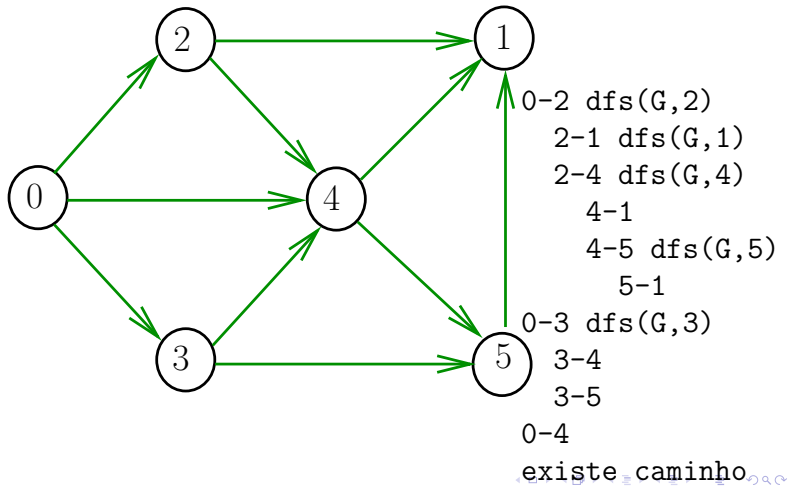
Navigation icons

DFSpaths

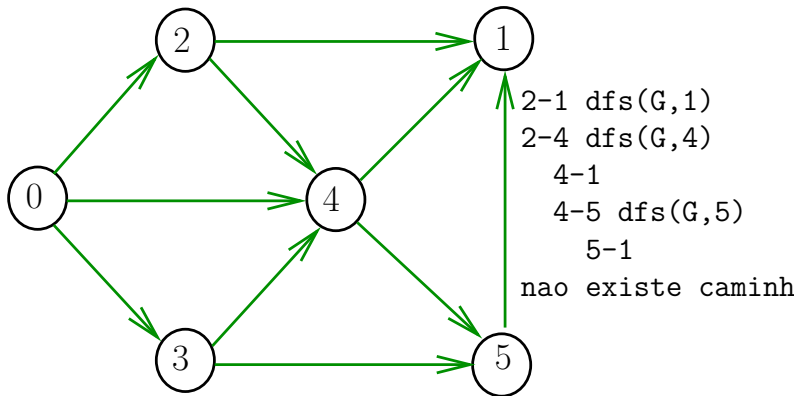
Há um caminho de s a v ?

```
public boolean hasPath(int v) {  
    return marked[v];  
}
```

DFSpaths(G, 0)



DFSpaths(G, 2)



Consumo de tempo

Qual é o consumo de tempo de `DFSpaths`?

Consumo de tempo

Conclusão

Qual é o consumo de tempo de `DFSpaths`?

Qual é o consumo de tempo da função `dfs`?

O consumo de tempo de `DFSpaths` é $\Theta(V)$ mais o consumo de tempo da função `dfs()`.

Conclusão

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 $\sim O(V + E)$.

Conclusão

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



Fonte: [Tron Legacy Light Cycle Riders wallpaper](#)

Caminhos no computador

Como representar **caminhos** no computador?

Caminhos no computador

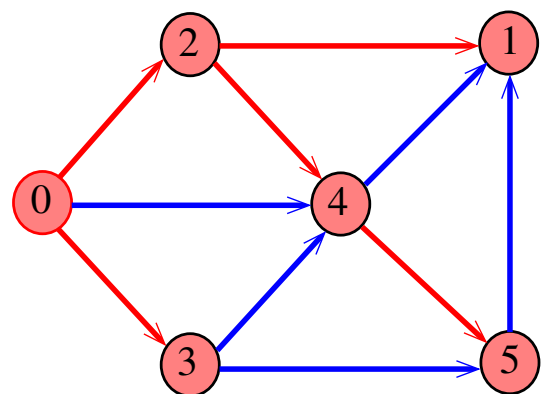
Uma maneira **compacta** de representar **caminhos** de um **vértice** a outros é uma arborescência

Uma **arborescência** é um digrafo em que

- ▶ existe **exatamente um vértice** com grau de entrada 0, a **raiz** da arborescência
- ▶ **não existem vértices** com grau de entrada maior que 1,
- ▶ **cada um dos vértices** é término de um caminho com origem no vértice **raiz**.

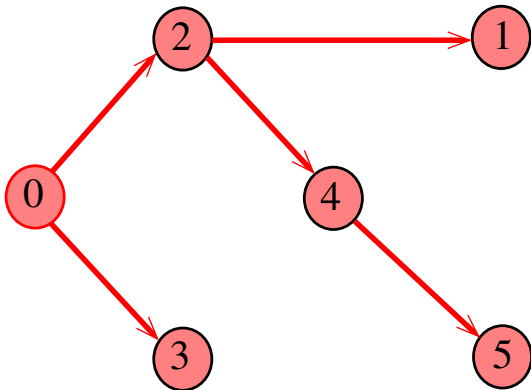
Arborescências

Exemplo: a raiz da arborescência é 0



Arborescências

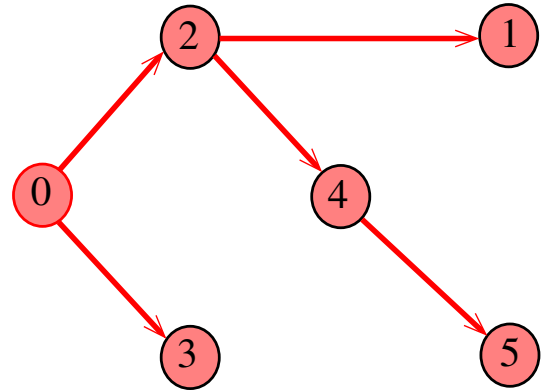
Exemplo: a raiz da arborescência é 0



Navigation icons

Arborescências

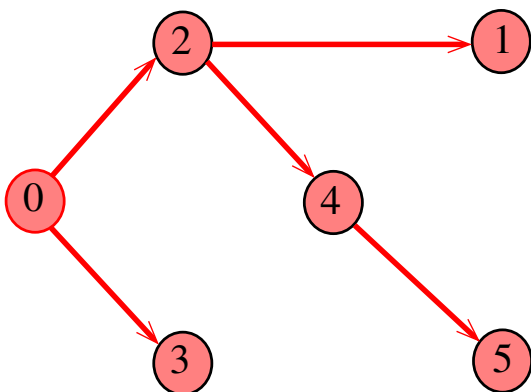
Propriedade: para todo vértice v , existe exatamente um caminho da raiz a v



Navigation icons

Arborescências

Todo vértice w , exceto a raiz, tem uma **pai**: o **único** vértice v tal que $v-w$ é um arco



Navigation icons

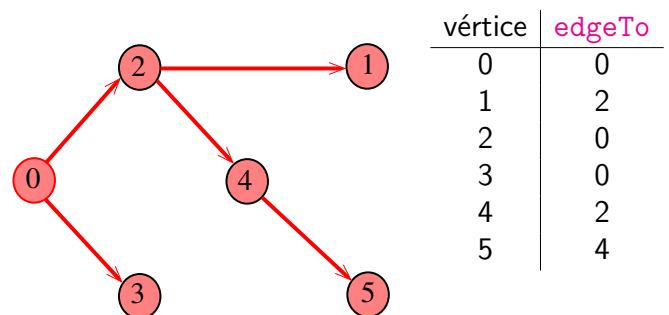
Caminho

Dado o vetor de pais, `edgeTo`, de uma arborescência, é fácil determinar o caminho que leva da raiz a um dado vértice v : basta inverter a sequência impressa pelo seguinte fragmento de código:

Navigation icons

Arborescências 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

Navigation icons

Caminho

Dado o vetor de pais, `edgeTo`, de uma arborescência, é fácil determinar o caminho que leva da raiz a um dado vértice v : basta inverter a sequência impressa pelo seguinte fragmento de código:

```
for (int x=v; edgeTo[x]!=x; x=edgeTo[x])
    StdOut.printf("%d-", x);
StdOut.printf("%d", x);
```

Navigation icons

DFSpaths: esqueleto

```
public class DFSpaths {
    private final int s;
    private boolean[] marked;
    private int[] edgeTo;
    public DFSpaths(Digraph G, int s) {}
    private void dfs(Digraph G, int v) {}
    public boolean hasPath(int v) {}
    public Iterable<Integer> pathTo(int v)
}
```

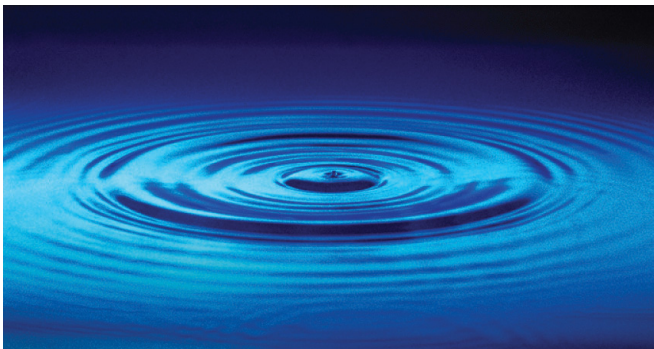
<> <> <> <> <> <> <>

DFSpaths: dfs()

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

<> <> <> <> <> <> <>

Busca em largura



Fonte: <http://catalog.flatworldknowledge.com/bookhub/>

<> <> <> <> <> <> <>

DFSpaths: construtor

Encontra um caminho de **s** a todo vértice alcançável a partir de **s**.

```
public DFSpaths(Digraph G, int s) {
    marked = new boolean[G.V()];
    edgeTo = new int[G.V()];
    this.s = s;
    dfs(G, s);
}
```

<> <> <> <> <> <> <>

DFSpaths: pathTo()

Retorna um **caminho** de **s** a **v** ou **null** se um tal caminho não existe.

```
public Iterable<Integer> pathTo(int v) {
    if (!hasPath(v)) return null;
    Stack<Integer> path =
        new Stack<Integer>();
    for (int x = v; x != s; x = edgeTo[x])
        path.push(x);
    path.push(s);
    return path;
}
```

<> <> <> <> <> <> <>

Busca ou varredura

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

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

<> <> <> <> <> <> <>

Busca em largura

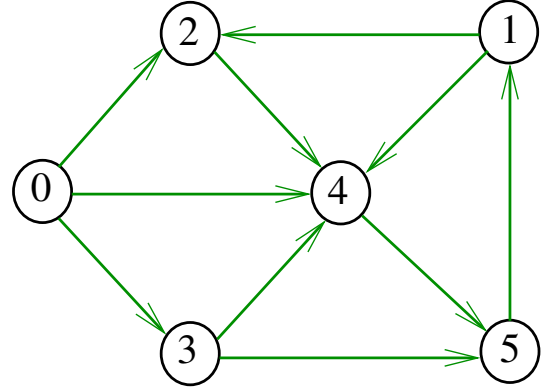
A **busca em largura** (= *breadth-first search search* = *BFS*) começa por um vértice, digamos **s**, especificado pelo usuário.

O algoritmo

visita **s**,
 depois visita vértices à **distância 1** de **s**,
 depois visita vértices à **distância 2** de **s**,
 depois visita vértices à **distância 3** de **s**,
 e assim por diante

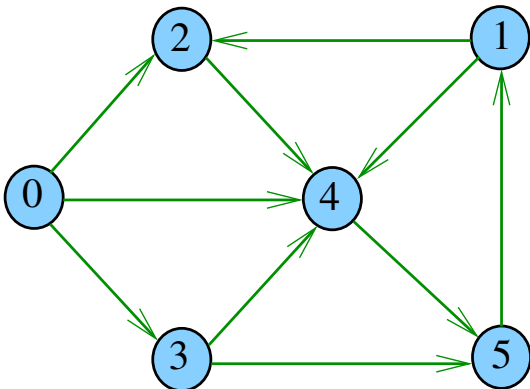
Simulação

i	0	1	2	3	4	5
q[i]						



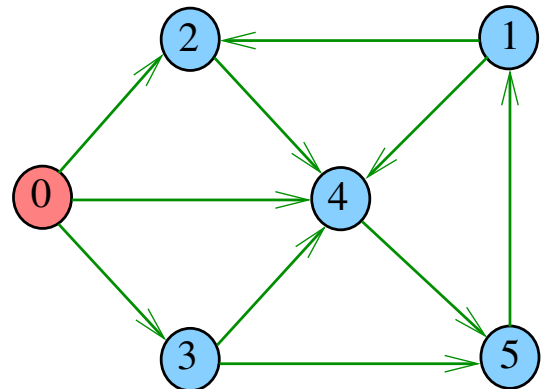
Simulação

i	0	1	2	3	4	5
q[i]						



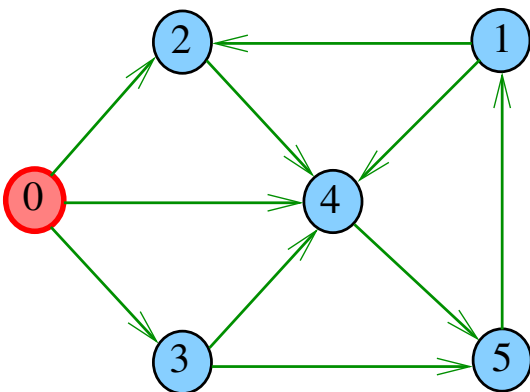
Simulação

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



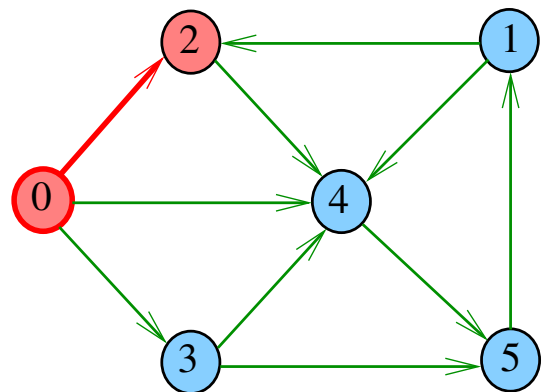
Simulação

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



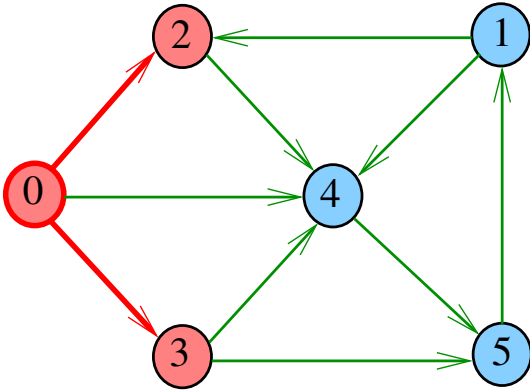
Simulação

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



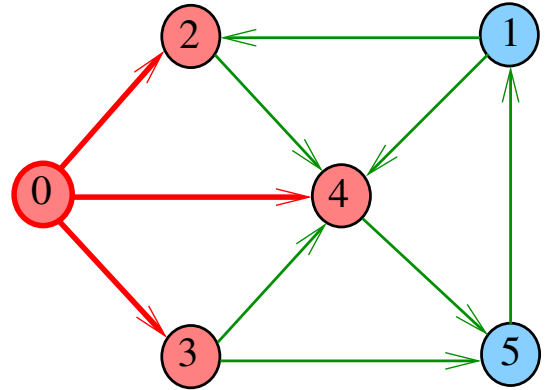
Simulação

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



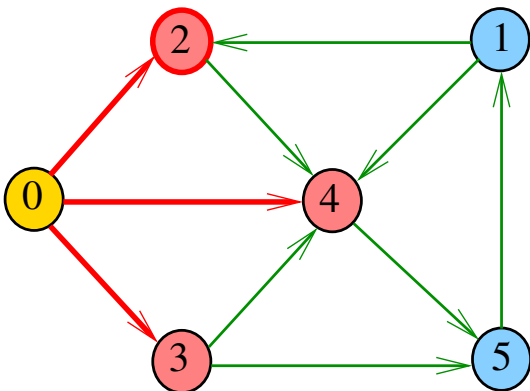
Simulação

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



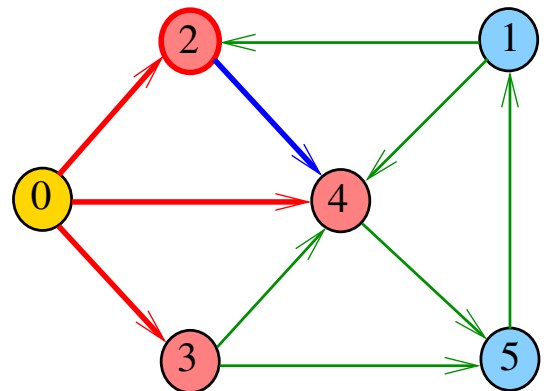
Simulação

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



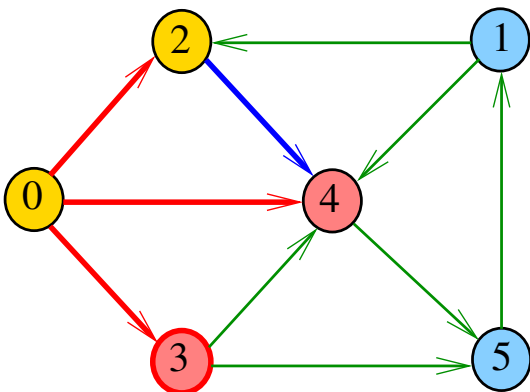
Simulação

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



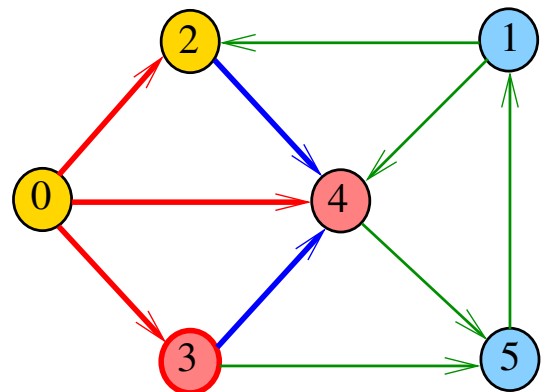
Simulação

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



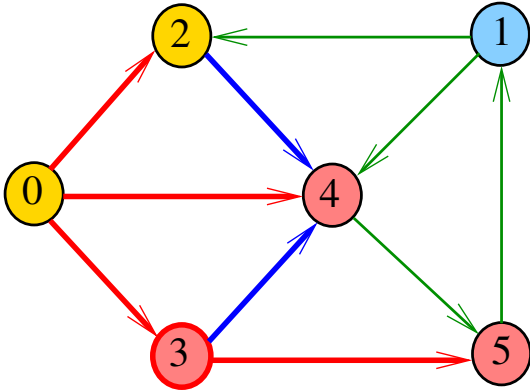
Simulação

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



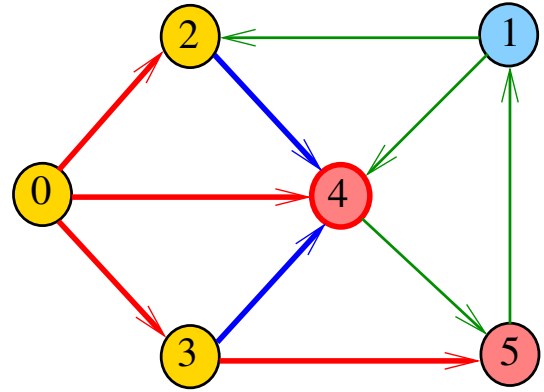
Simulação

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



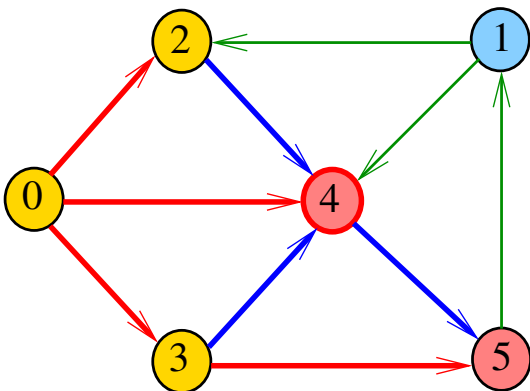
Simulação

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



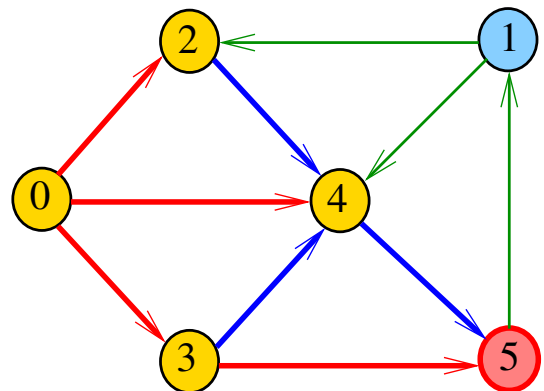
Simulação

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



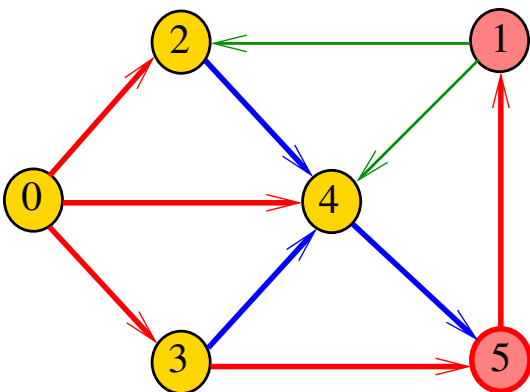
Simulação

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



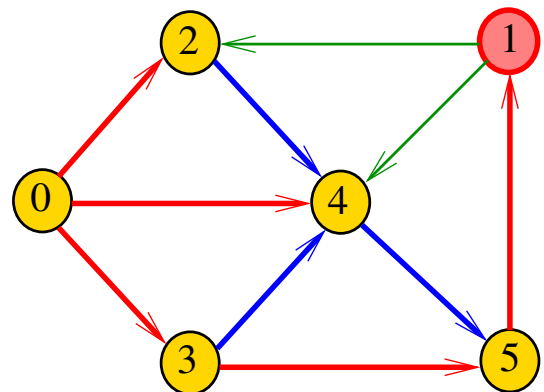
Simulação

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



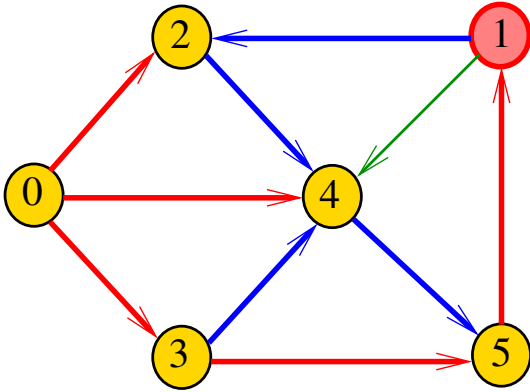
Simulação

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



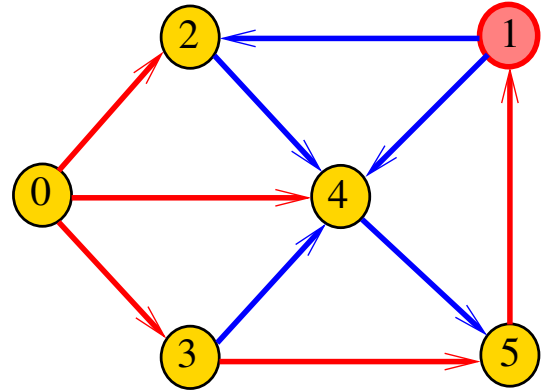
Simulação

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



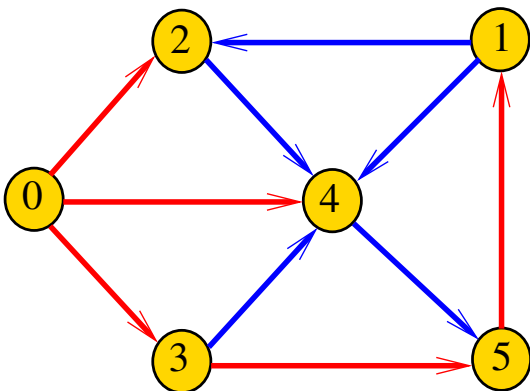
Simulação

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



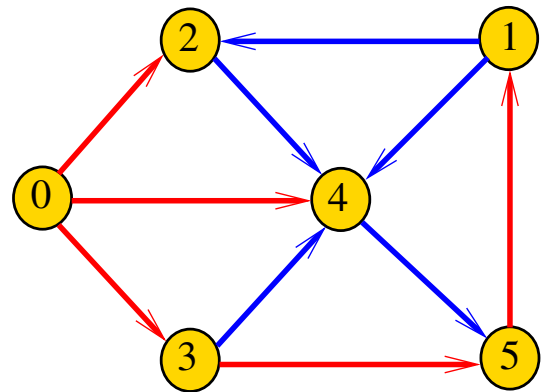
Simulação

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



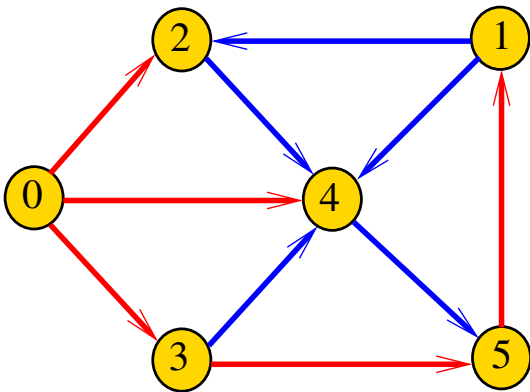
Arborescência da BFS

A busca em largura a partir de um vértice s descreve a arborescência com raiz s



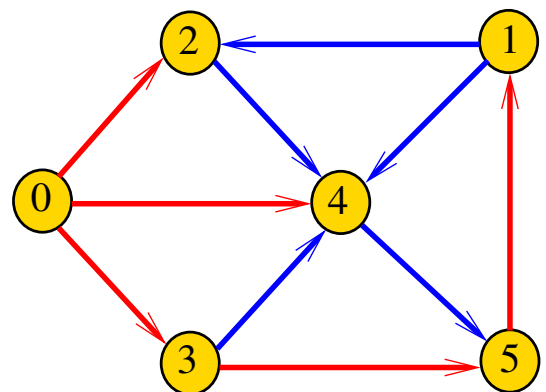
Arborescência da BFS

Essa arborescência é conhecida como **arborescência de busca em largura** (= *BFS tree*)



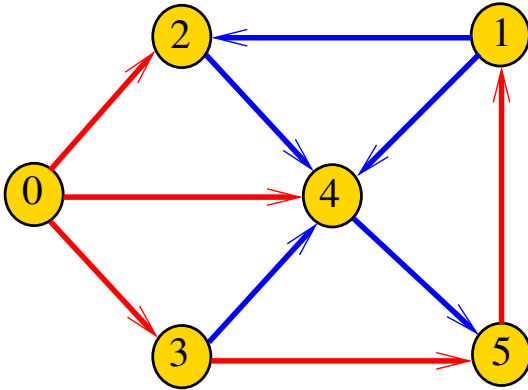
Representação da BFS

Podemos representar essa arborescência explicitamente por um vetor de pais `edgeTo []`



Representação da BFS

v	0	1	2	3	4	5
edgeTo	0	5	0	0	0	3



BFSpaths: esqueleto

```
public class BFSpaths {
    private final int s;
    private boolean[] marked;
    private int[] edgeTo;
    public BFSpaths(Digraph G, int s) {}
    private void bfs(Digraph G, int s) {}
    public boolean hasPath(int v) {}
    public Iterable<Integer> pathTo(int v) {}
}
```

bfs(): inicializações

```
private void bfs(Digraph G, int s) {
    Queue<Integer> q = new Queue<Integer>();
    marked[v] = true;
    q.enqueue(s);
    // aqui vem a iteração do próximo slide
}
```

Class BFSpaths

BFSpaths visita todos os vértices do digrafo **G** que podem ser alcançados a partir de **s**.

A visita aos vértices é registrada no vetor **marked[]**. Se **v** foi então **marked[v] == true**.

Para isso **BFSpaths** usa uma **fila** de vértices:

```
Queue<Integer> q = new Queue<Integer>();
```

BFSpaths

Encontra um caminho de **s** a todo vértice alcançável a partir de **s**.

```
public BFSpaths(Digraph G, int s) {
    marked = new boolean[G.V()];
    edgeTo = new int[G.V()];
    this.s = s;
    bfs(G, s);
}
```

bfs(): iteração

```
while (!q.isEmpty()) {
    int v = q.dequeue();
    for (int w : G.adj(v)) {
        if (!marked[w]) {
            edgeTo[w] = v;
            marked[w] = true;
            q.enqueue(w);
        }
    }
}
```


BFSpaths

Há um caminho de s a v ?

```
// Método copiado de DFSpaths.  
public boolean hasPath(int v) {  
    return marked[v];  
}
```

◀ ▶ ⏪ ⏩ 🔍 🔄

Relações invariantes

Digamos que um vértice v foi **visitado** se $\text{marked}[v] == \text{true}$

No início de cada iteração vale que

- ▶ todo vértice que está na fila já foi visitado;
- ▶ se um vértice v já foi visitado mas algum de seus vizinhos ainda não foi visitado, então v está na fila.

Cada vértice entra na fila no **máximo uma vez**. Portanto, basta que a fila tenha espaço suficiente para $G.V()$ vértices.

◀ ▶ ⏪ ⏩ 🔍 🔄

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

◀ ▶ ⏪ ⏩ 🔍 🔄

BFSpaths

Retorna um **caminho** de s a v ou **null** se um tal caminho não existe.

```
// Método copiado de DFSpaths.  
public Iterable<Integer> pathTo(int v) {  
    if (!hasPath(v)) return null;  
    Stack<Integer> path =  
        new Stack<Integer>();  
    for (int x = v; x != s; x = edgeTo[x])  
        path.push(x);  
    path.push(s);  
    return path;  
}
```

◀ ▶ ⏪ ⏩ 🔍 🔄

Consumo de tempo

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

O consumo de tempo da função **BFSpaths** para **matriz de adjacência** é $O(V^2)$.

◀ ▶ ⏪ ⏩ 🔍 🔄

Certificados



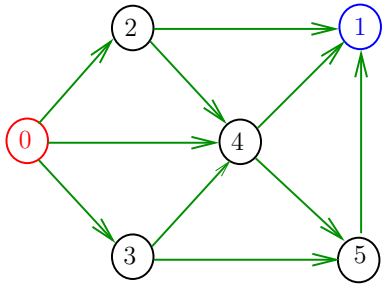
Fonte: [Free Printable Computer Achievement Award Certificates](#)

◀ ▶ ⏪ ⏩ 🔍 🔄

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

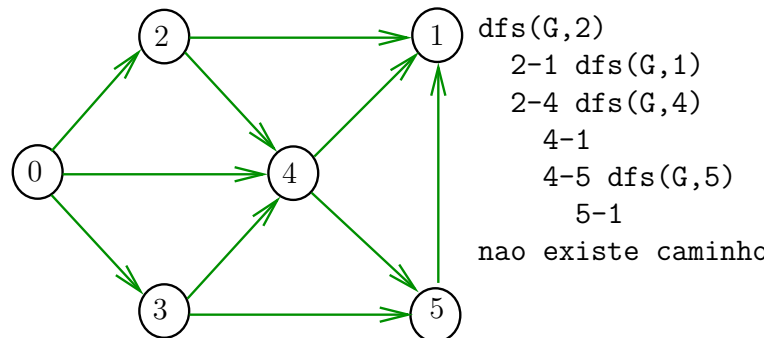
Elas terão um papel **suuupeeer** importante no final de [MAC0338 Análise de Algoritmos](#) e em [MAC0414 Autômatos, Computabilidade e Complexidade](#)

Elas estão relacionadas com o **Teorema da Dualidade** visto em [MAC0315 Otimização Linear](#)

◀ ▶ 🔍 ↺ ↻

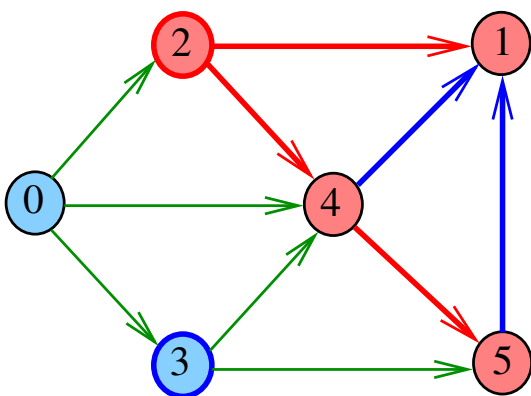
Certificado de inexistência

Como é possível demonstrar que o problema **não tem solução**?



◀ ▶ 🔍 ↺ ↻

DFSpath($G, 2, 3$)

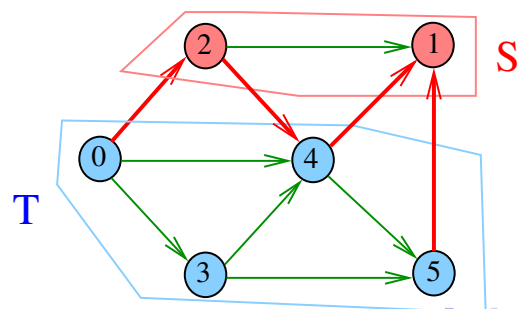


◀ ▶ 🔍 ↺ ↻

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)

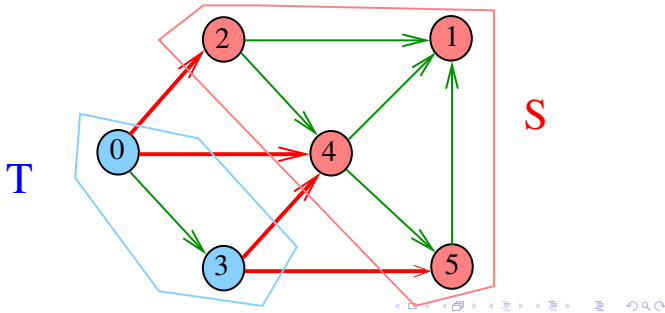


◀ ▶ 🔍 ↺ ↻

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)

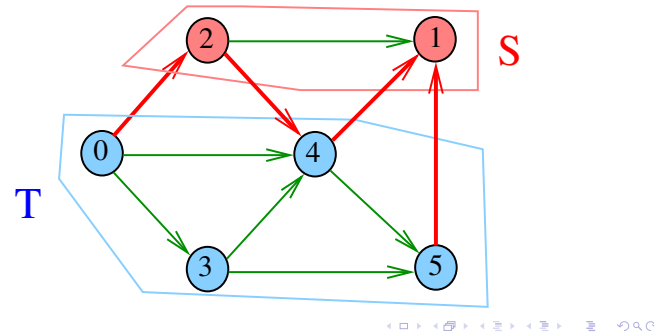


Certificado de inexistência

st-Cortes (= st-cuts)

Um corte (S, T) é um **st-corte** se
 s está em S e t está em T

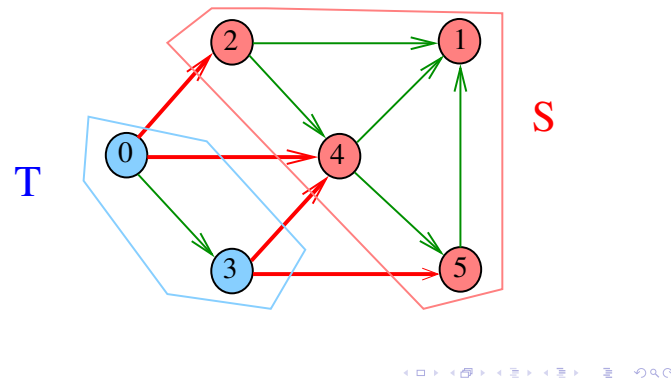
Exemplo: (S, T) é um 1-3-corte um 2-5-corte ...



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

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 uma 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 está representado no
vetor **edgeTo** [] .

No código das 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 $V + E$.