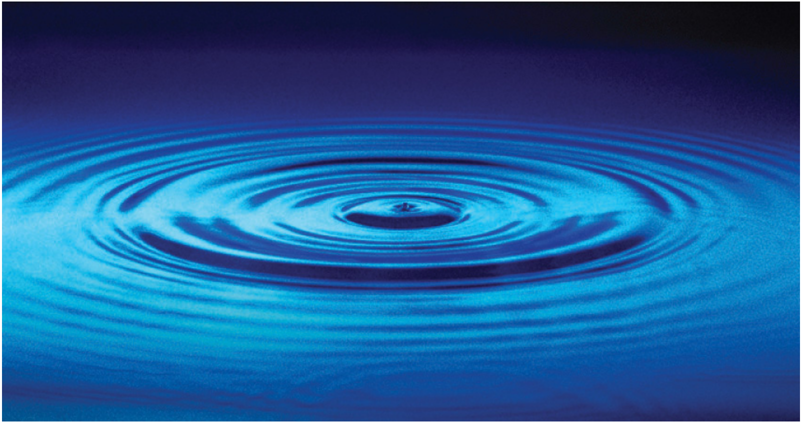
AULA 22

Busca em largura



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

Busca ou varredura

Um algoritimo 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

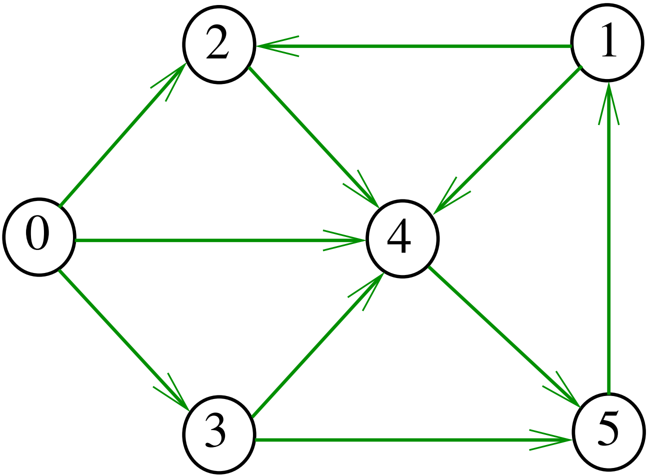
A **busca em largura** (=*breadth-first search search* = *BFS*) começa por um vértice, digamos s, especificado pelo usuário.

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

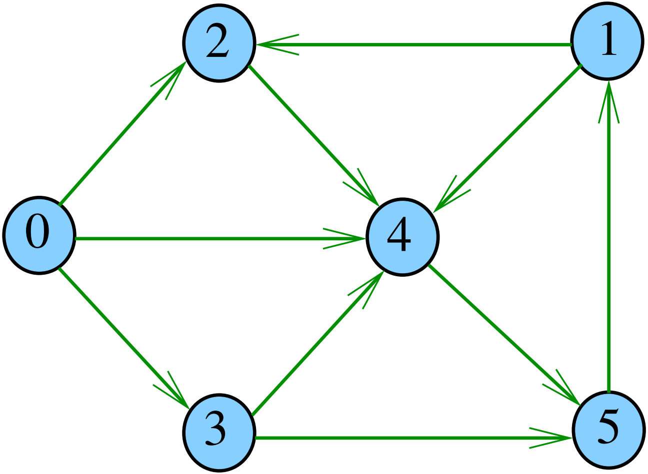
Simulação

i	0	1	2	3	4	5
q[i]						



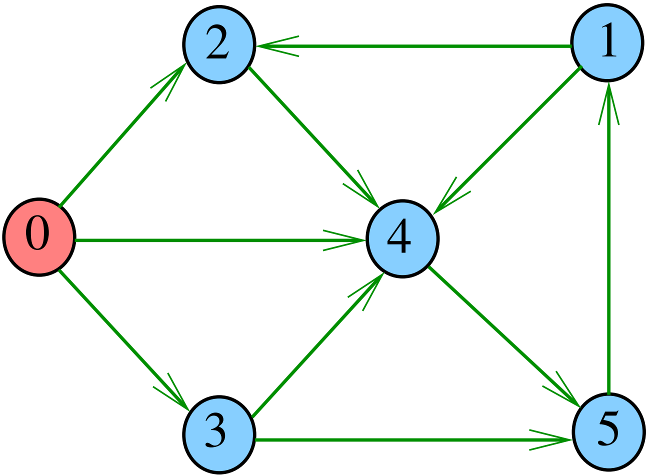
Simulação

i	0	1	2	3	4	5
q[i]						



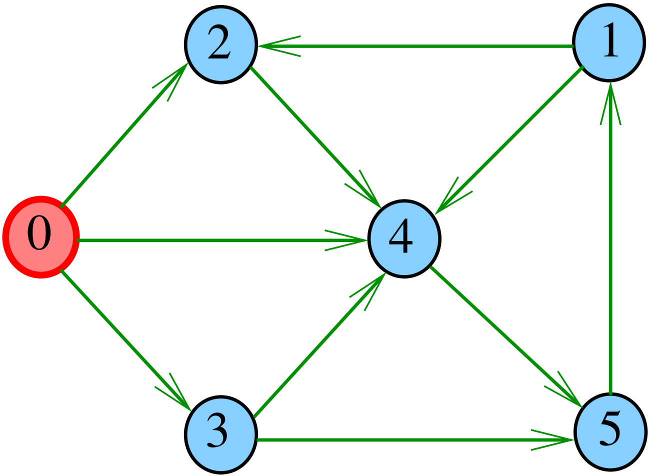
Simulação

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



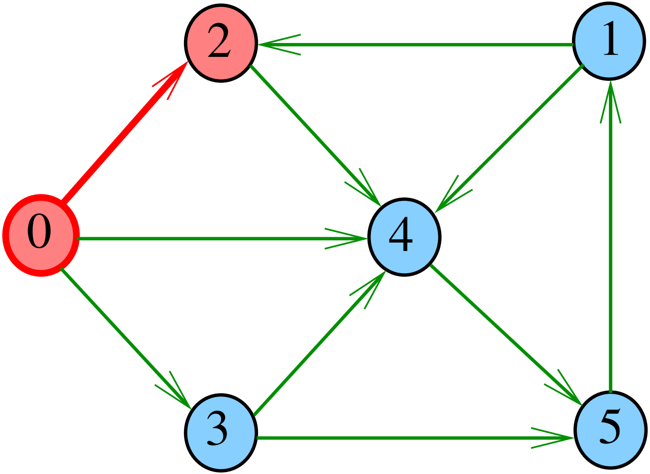
Simulação

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



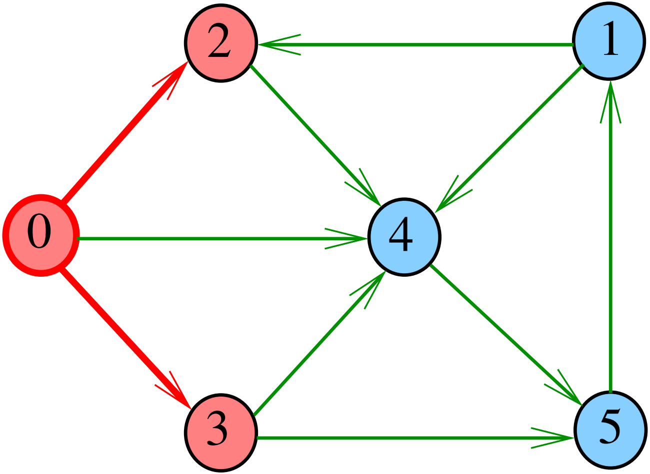
Simulação

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



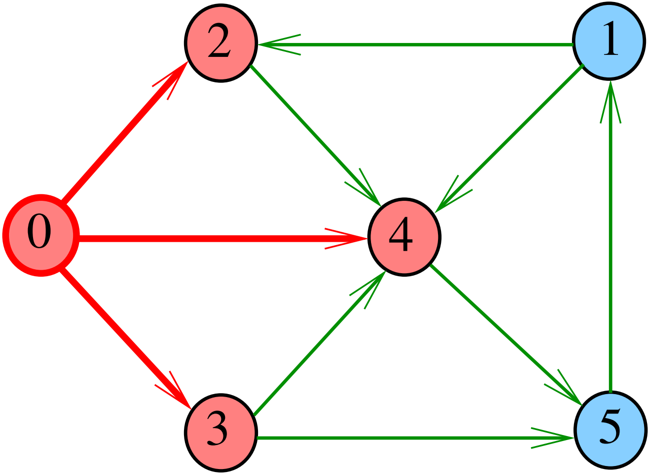
Simulação

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



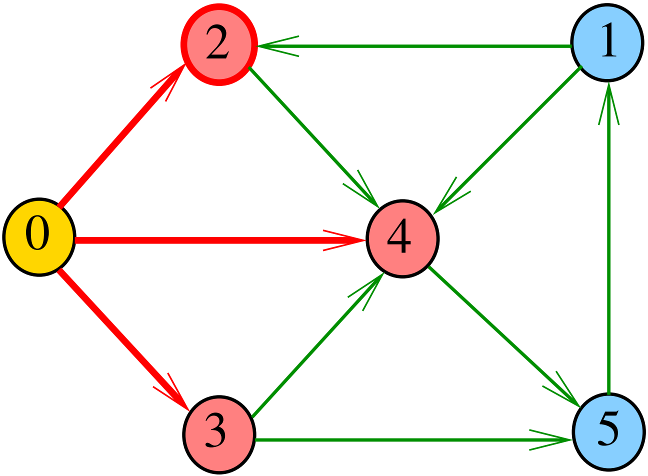
Simulação

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



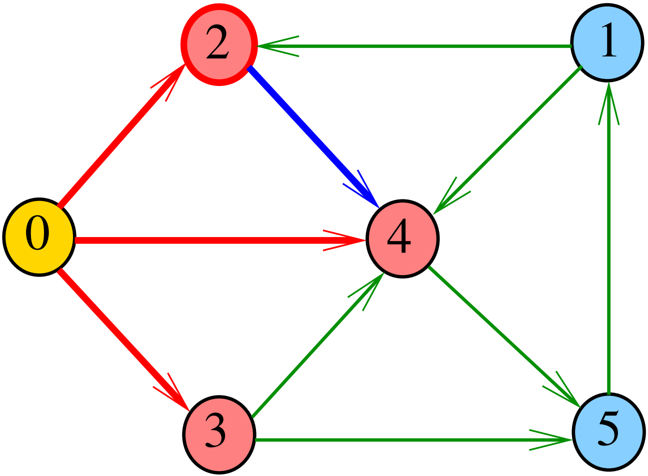
Simulação

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



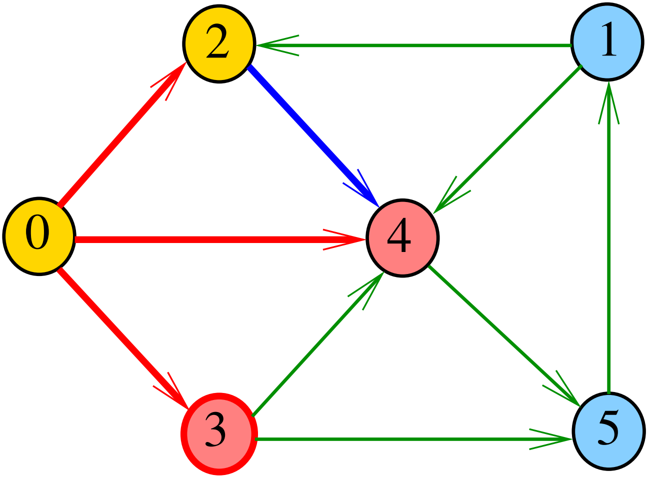
Simulação

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



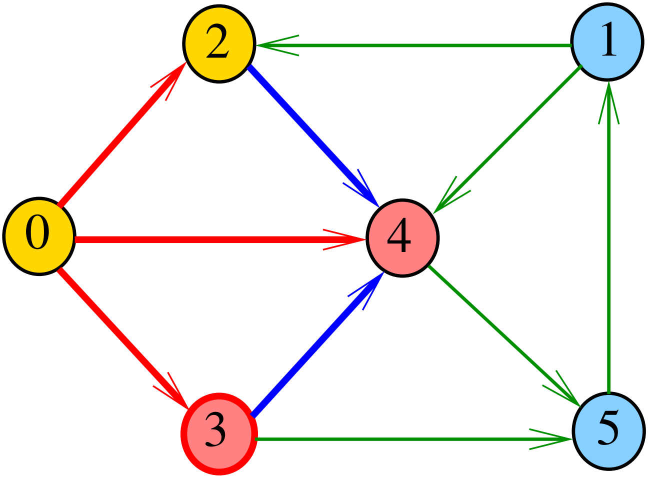
Simulação

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



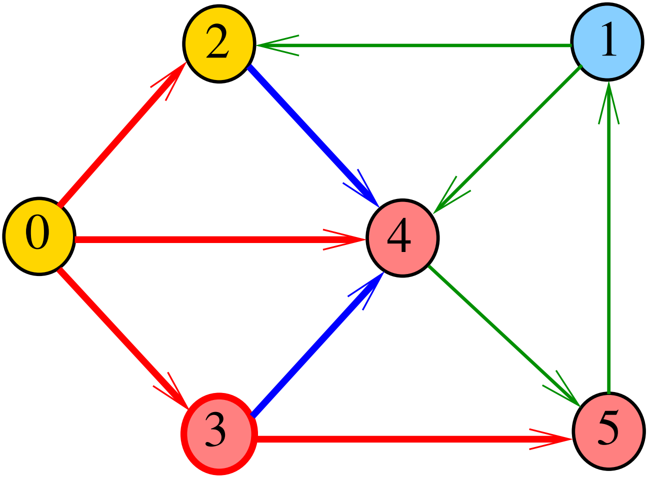
Simulação

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



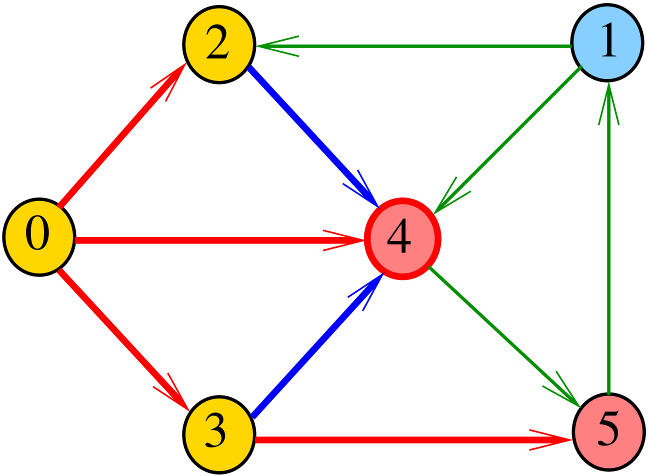
Simulação

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



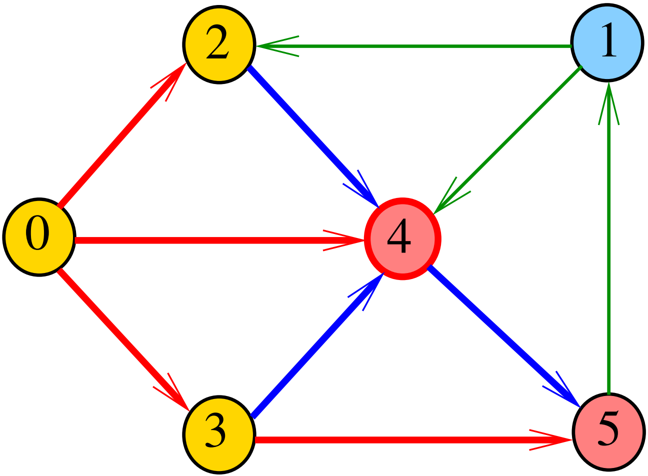
Simulação

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



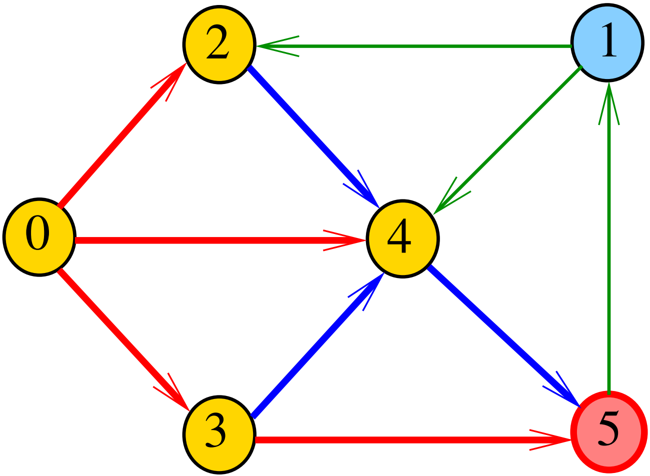
Simulação

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



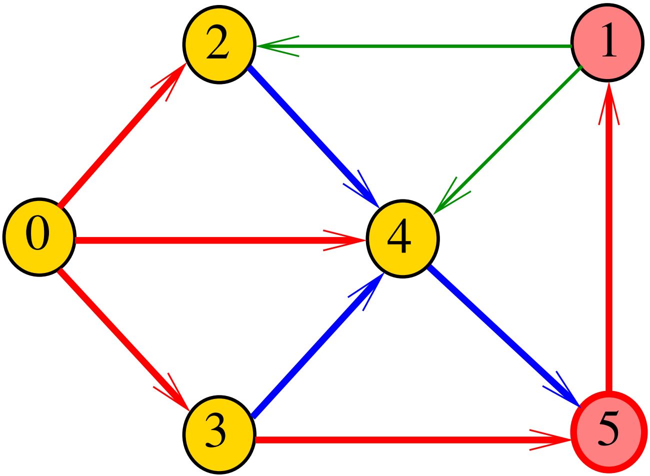
Simulação

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



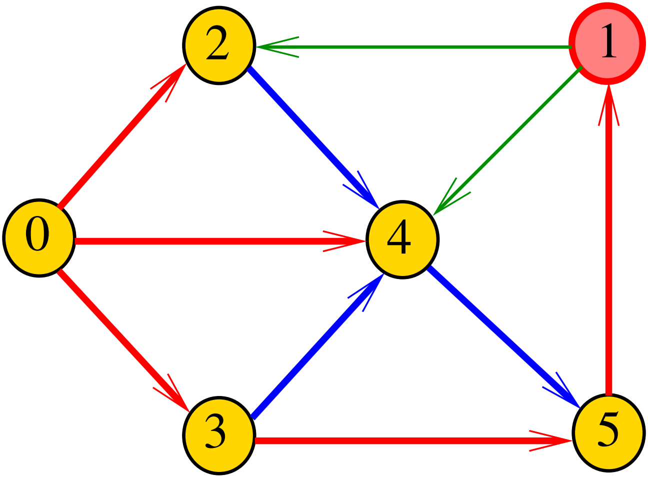
Simulação

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



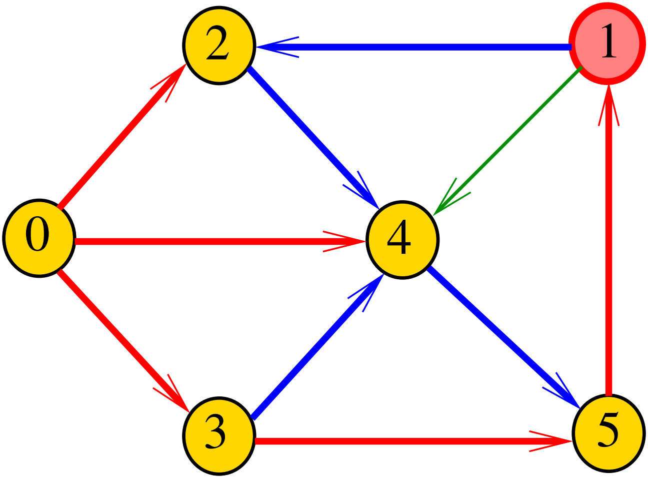
Simulação

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



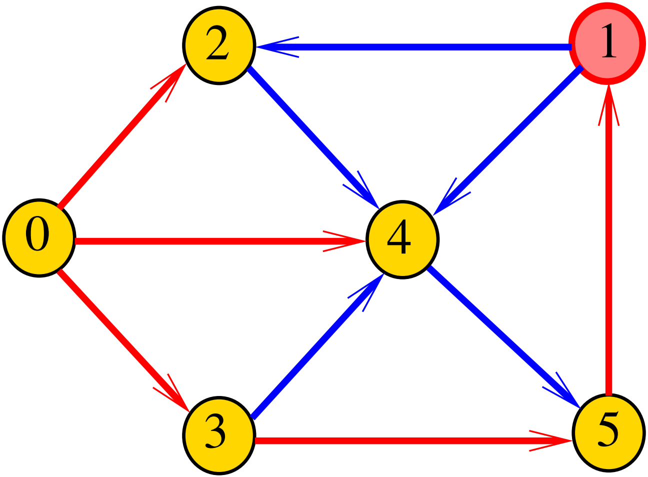
Simulação

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



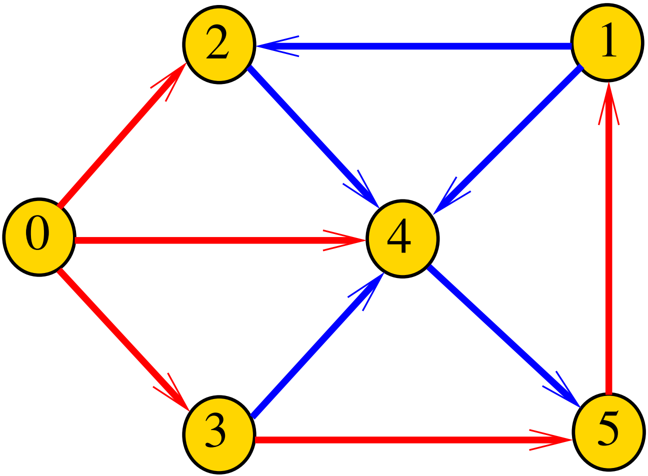
Simulação

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



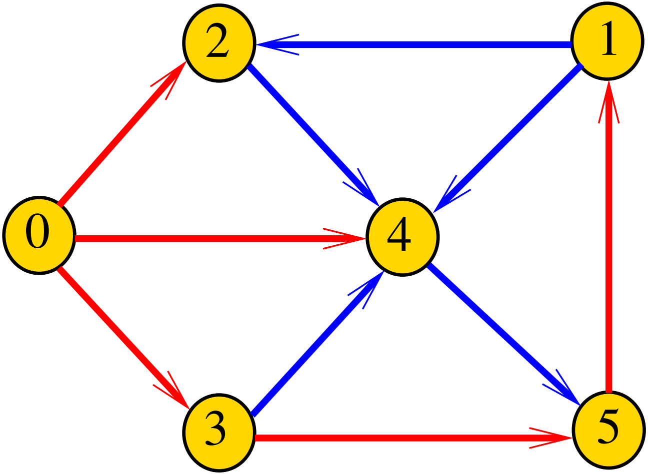
Simulação

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



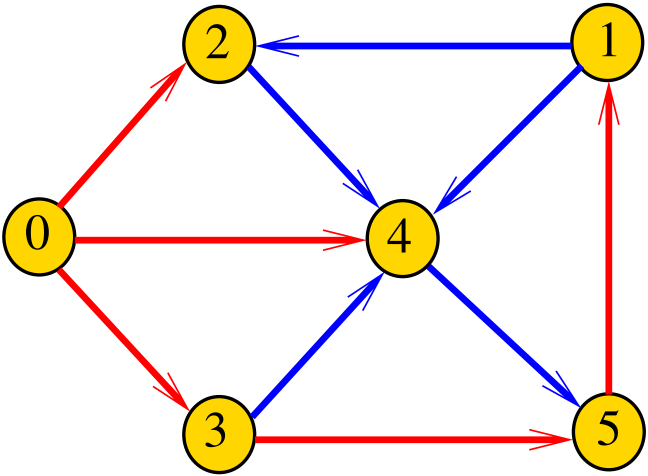
Arborescência da BFS

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



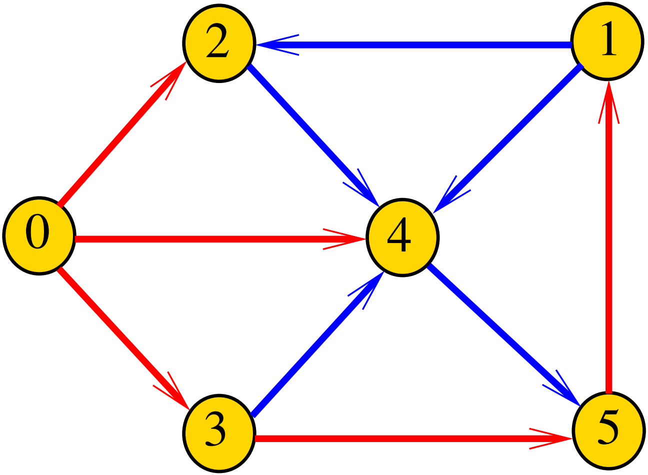
Arborescência da BFS

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



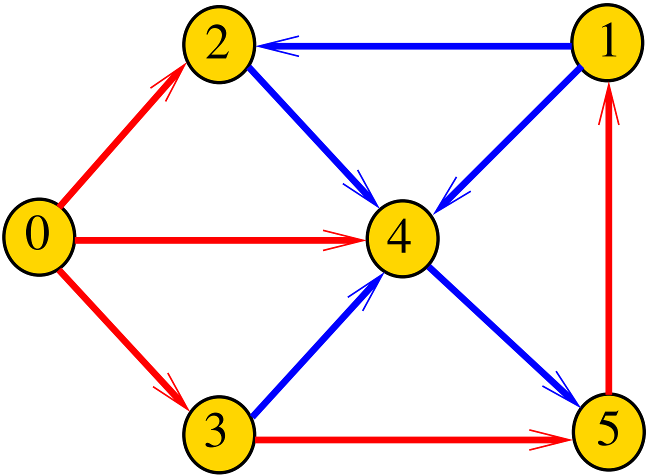
Representação da BFS

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



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: esqueleto

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

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()];
   distTo = new int[G.V()];
   this.s = s;
   bfs(G, s);
}
```

`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
```

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];
}
```

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;
}
```

Relações invariantes

Digamos que um vértice `v` foi **visitado** se

```
marked[v] == 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.

Consumo de tempo

O consumo de tempo da função BFSpaths para vetor de listas de adjacência é $\mathrm{O}(\mathtt{V} + \mathtt{E})$.

O consumo de tempo da função BFSpathspara matriz de adjacência é $\mathrm{O}(\mathtt{V}^2)$.

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

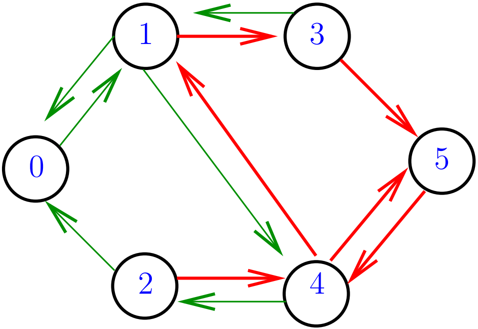
Caminhos mínimos

Fonte: http://vandanasanju.blogspot.com.br/

Comprimento

O **comprimento** de um caminho é o número de arcos no caminho, contanto-se as repetições

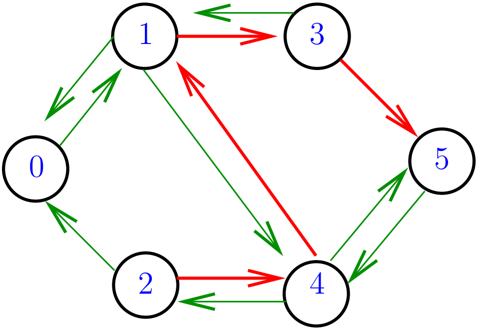
Exemplo: 2-4-1-3-5-4-5 tem comprimento 6



Comprimento

O **comprimento** de um caminho é o número de arcos no caminho, contanto-se as repetições.

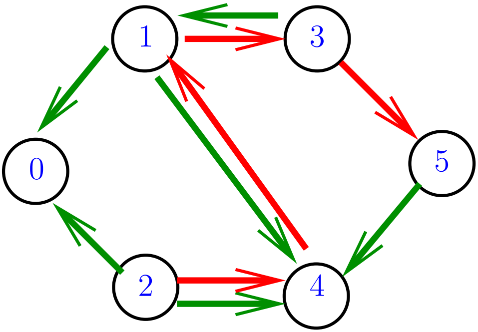
Exemplo: 2-4-1-3-5 tem comprimento 4



Distância

A **distância** de um vértice `s` a um vértice `t` é o menor comprimento de um caminho de `s` a `t`. Se não existe caminho de `s` a `t` a distância é **infinita**

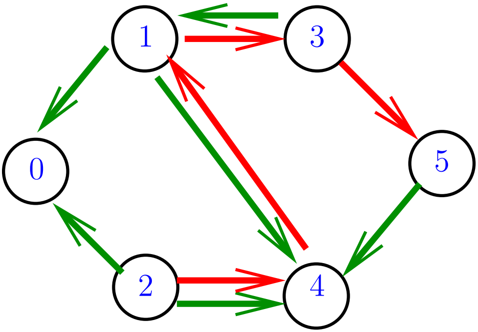
Exemplo: a distância de 2 a 5 é 4



Distância

A **distância** de um vértice `s` a um vértice `t` é o menor comprimento de um caminho de `s` a `t`. Se não existe caminho de `s` a `t` a distância é **infinita**

Exemplo: a distância de 0 a 2 é infinita

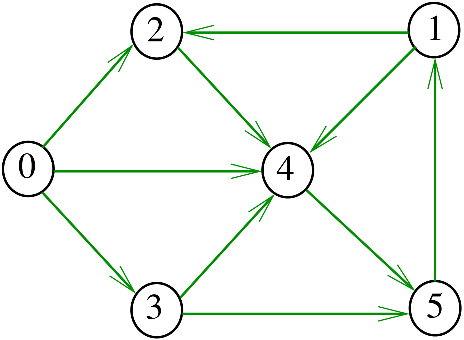


Calculando distâncias

Problema: dados um digrafo `G` e um vértice `s`, determinar a distância de `s` aos demais vértices do digrafo

Exemplo: para `s` $= 0$

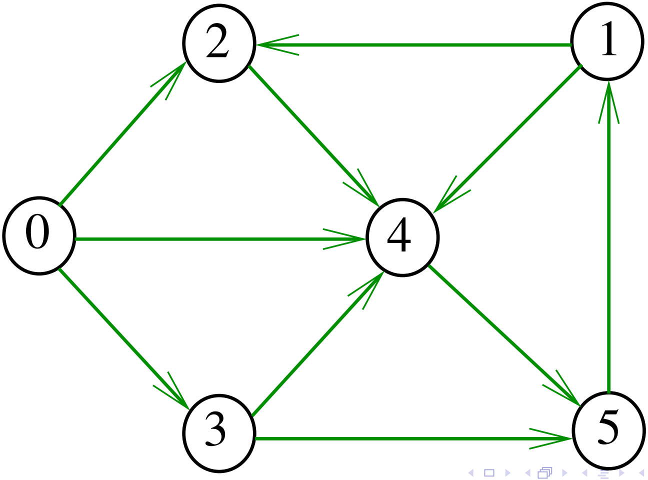
`v`	0	1	2	3	4	5
`distTo[v]`	0	3	1	1	1	2



Simulação

i	0	1	2	3	4	5
q[i]						

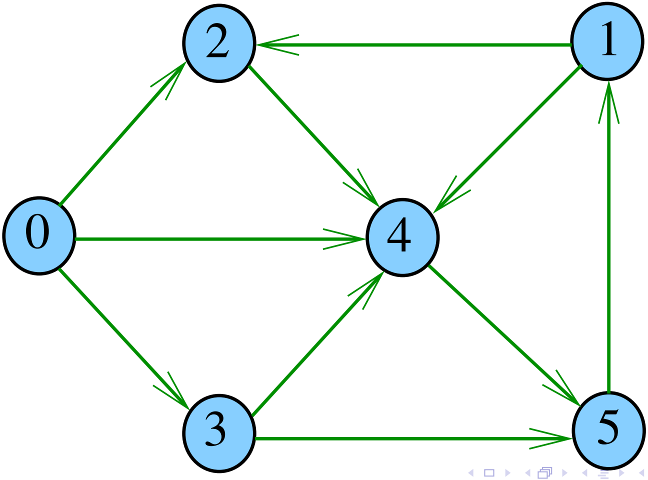
v	0	1	2	3	4	5
distTo[v]						



Simulação

i	0	1	2	3	4	5
q[i]						

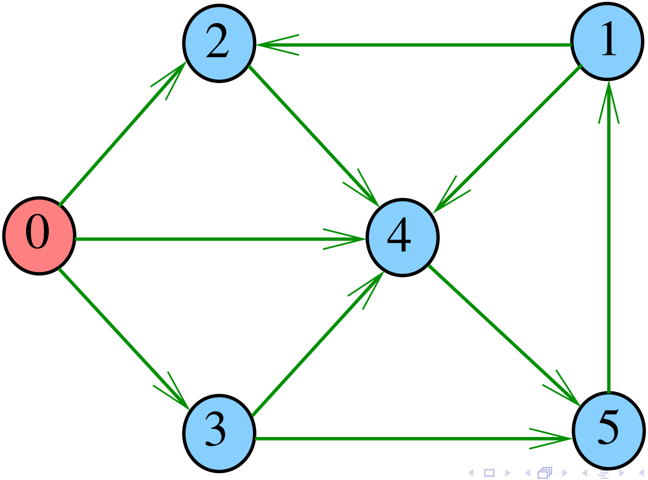
v	0	1	2	3	4	5
distTo[v]	6	6	6	6	6	6



Simulação

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

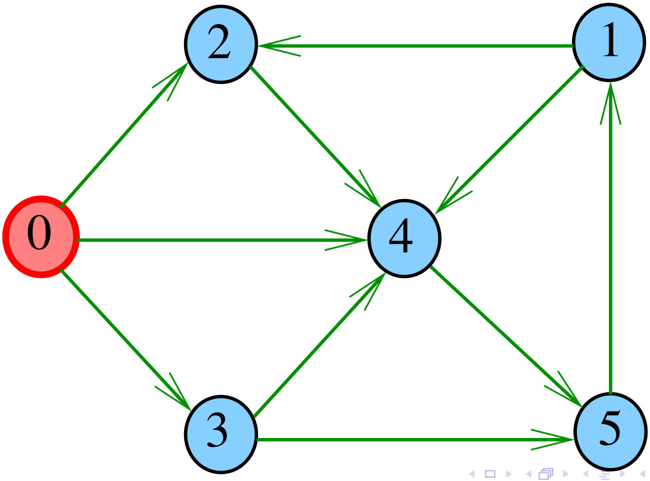
v	0	1	2	3	4	5
distTo[v]	6	6	6	6	6	6



Simulação

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

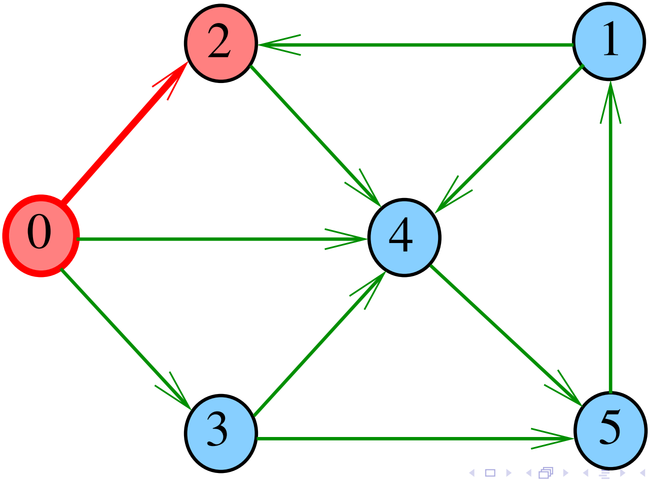
v	0	1	2	3	4	5
distTo[v]	**0**	6	6	6	6	6



Simulação

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

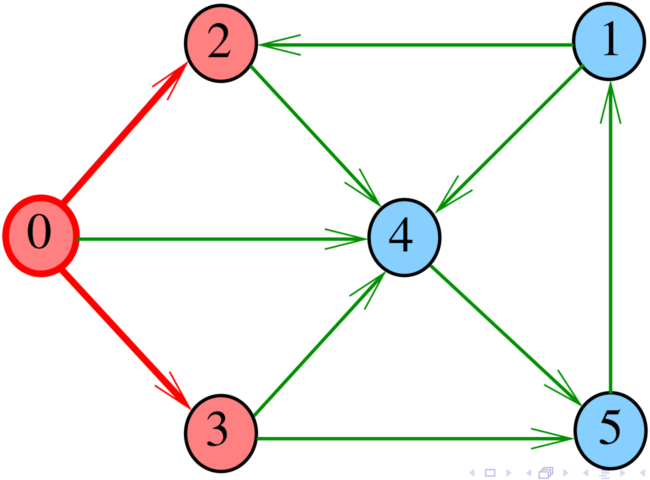
v	0	1	2	3	4	5
distTo[v]	0	6	**1**	6	6	6



Simulação

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

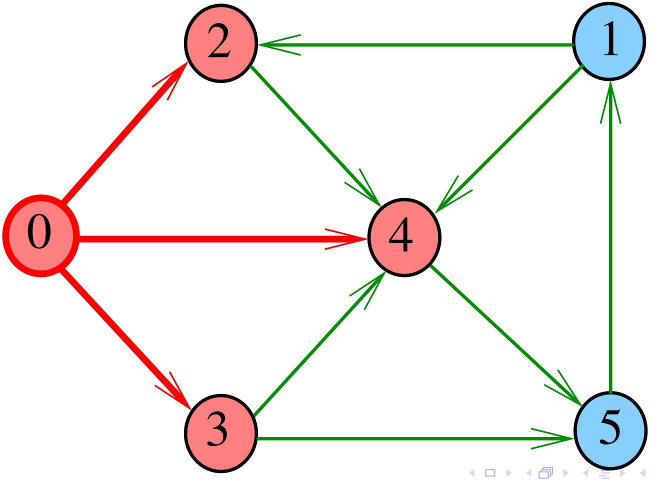
v	0	1	2	3	4	5
distTo[v]	0	6	1	**1**	6	6



Simulação

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

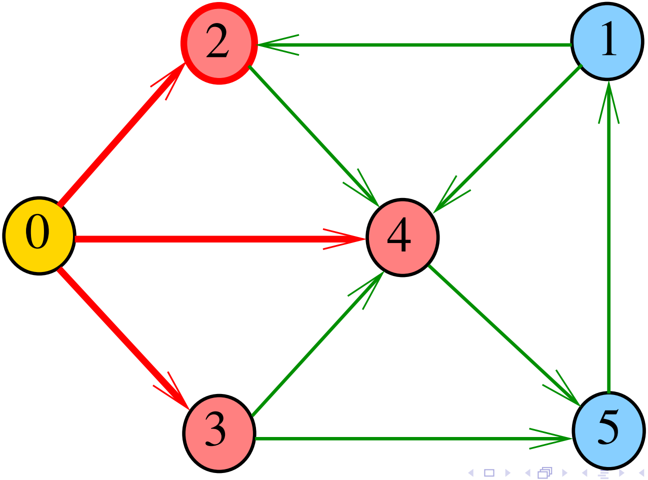
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	**1**	6



Simulação

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

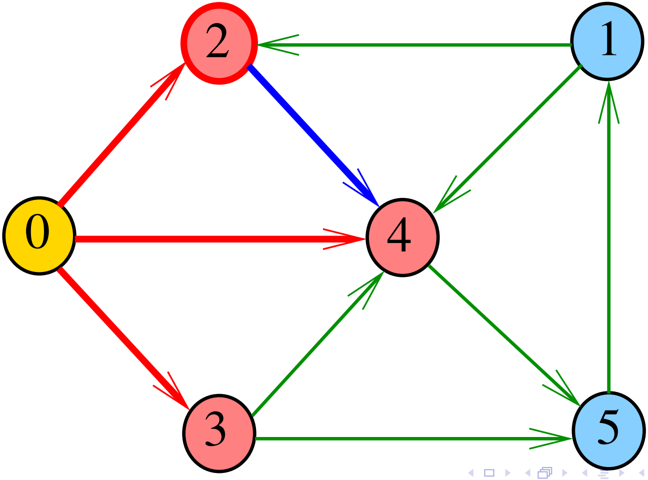
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	1	6



Simulação

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

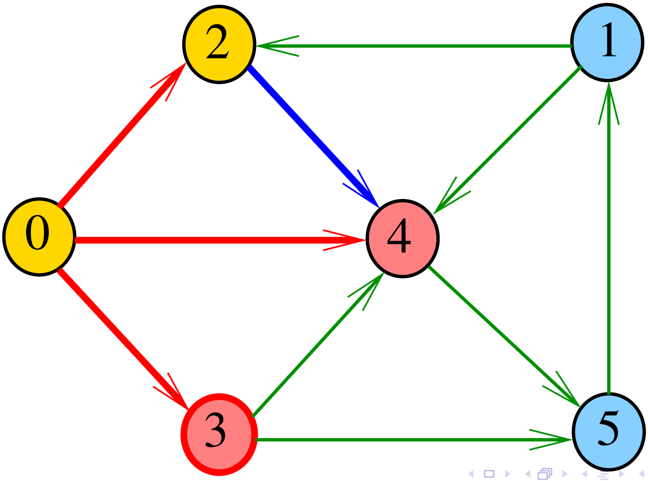
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	1	6



Simulação

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

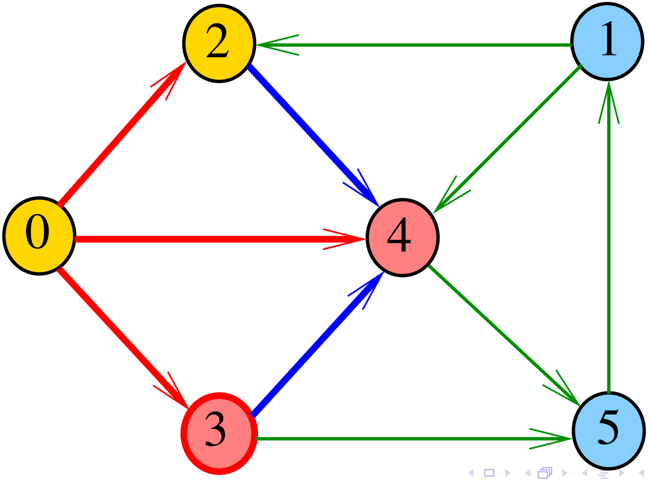
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	1	6



Simulação

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

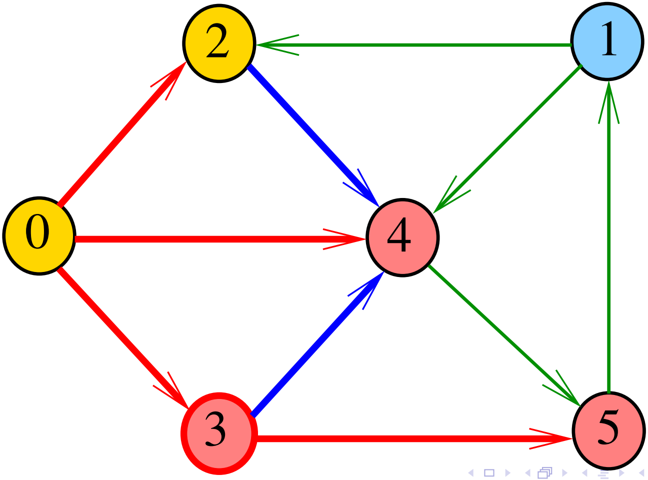
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	1	6



Simulação

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

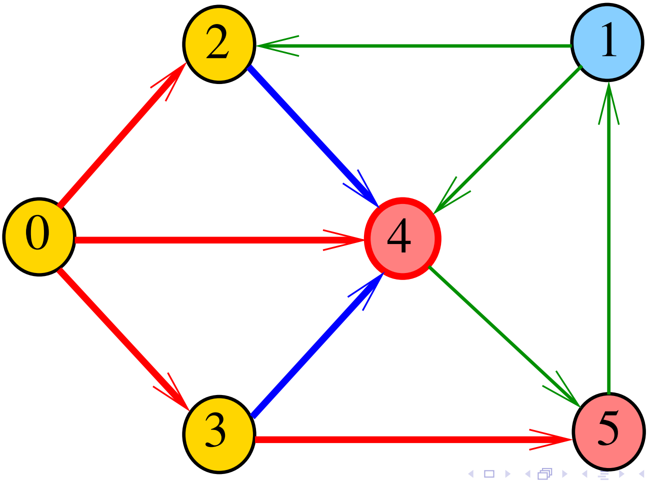
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	1	**2**



Simulação

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

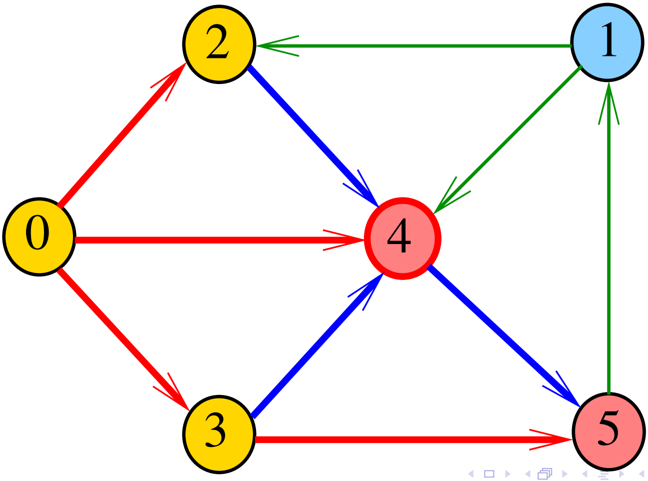
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	1	2



Simulação

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

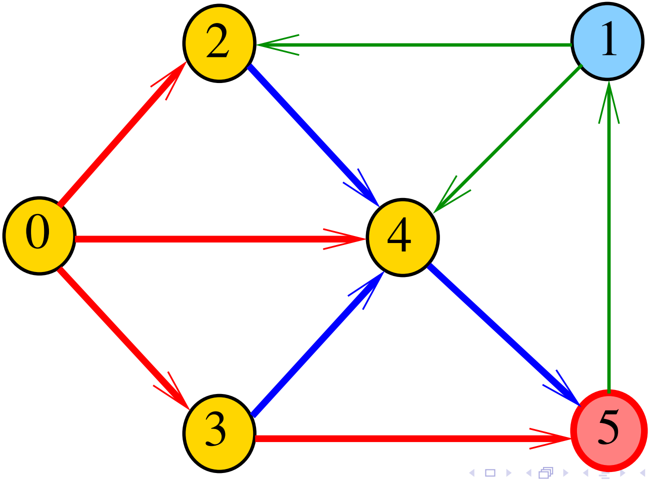
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	1	2



Simulação

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

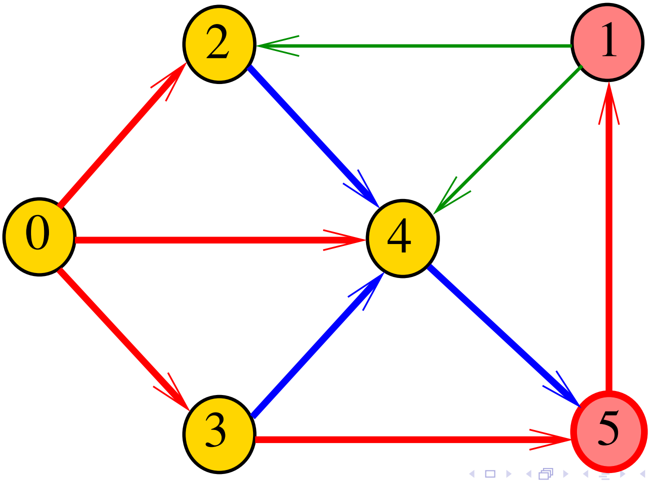
v	0	1	2	3	4	5
distTo[v]	0	6	1	1	1	2



Simulação

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

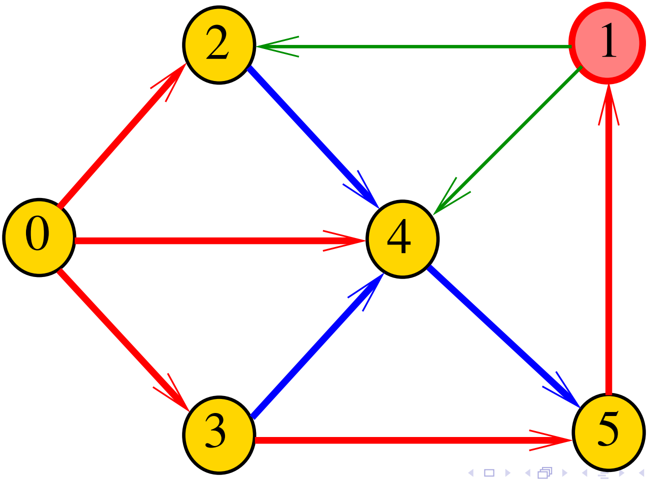
v	0	1	2	3	4	5
distTo[v]	0	3	1	1	1	2



Simulação

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

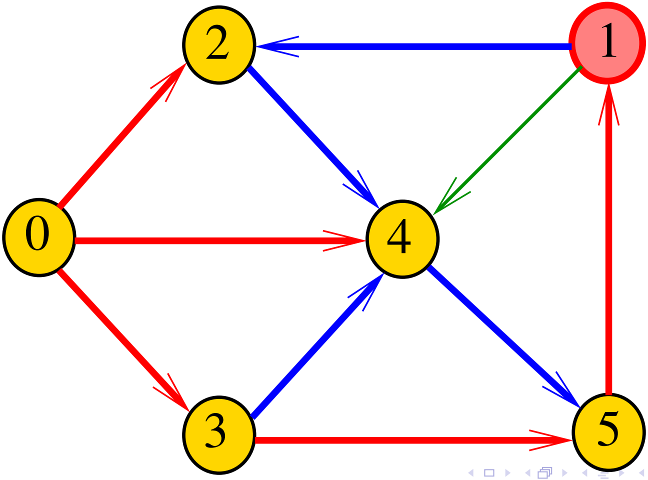
v	0	1	2	3	4	5
distTo[v]	0	3	1	1	1	2



Simulação

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

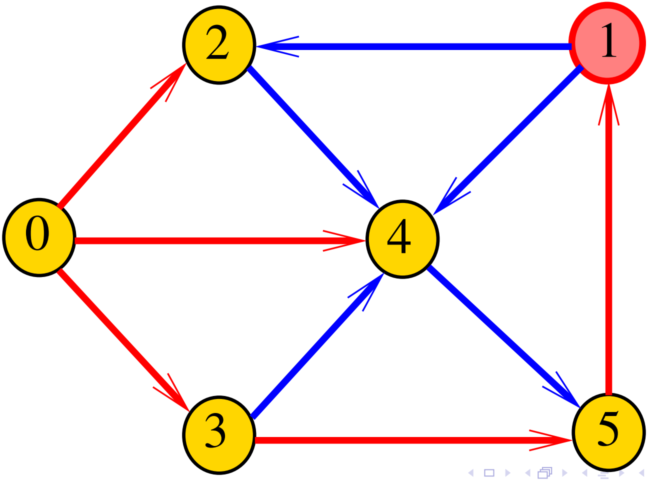
v	0	1	2	3	4	5
distTo[v]	0	3	1	1	1	2



Simulação

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

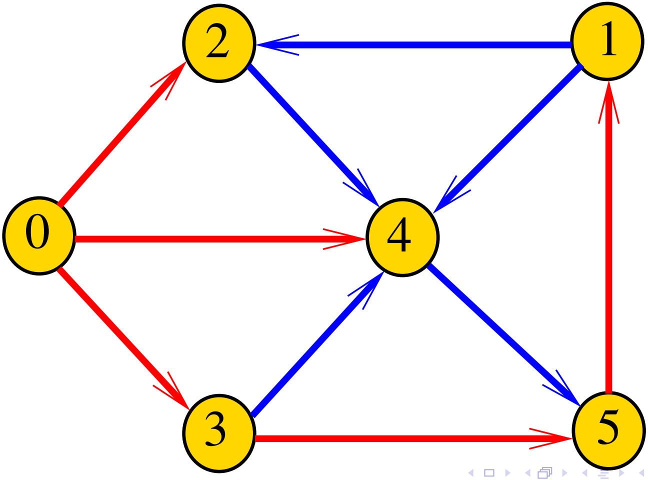
v	0	1	2	3	4	5
distTo[v]	0	3	1	1	1	2



Simulação

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

v	0	1	2	3	4	5
distTo[v]	0	3	1	1	1	2



Nova classe BFSpaths

BFSpaths armazena no vetor distTo[] a distância do vértice s a cada um dos vértices do digrafo G

A distância 'infinita' é representada por G.V()

```
private static final int INFINITY=G.V();
private int[] distTo = new int[G.V()];
```

Nova BFSpaths: esqueleto

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

BFSpaths

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

```
public BFSpaths(Digraph G, int s) {
   INFINITY = G.Vcor();
   marked = new boolean[G.V()];
   edgeTo = new int[G.V()];
   distTo = new int[G.V()];
   this.s = s;
   for (int v = 0; v < G.V(); v++)
      distTo[v] = INFINITY;
   bfs(G, s);
}
```

bfs(): inicializações

```
private void bfs(Digraph G, int s) {
    Queue<Integer> q= new Queue<Integer>();
    marked[v] = true;
    distTo[s] = 0;
    q.enqueue(s);

    // aqui vem a iteração do próximo slide
```

bfs(): iteração

```
while (!q.isEmpty()) {
    int v = q.dequeue();
    for (int w :  G.adj(v)) {
        if (!marked[w]) {
        if (distTo[w] == INFINITY) {
            edgeTo[w] = v;
            distTo[w] = distTo[v] + 1;
            marked[w] = true;
            q.enqueue(w);
        }
    }
}
}
```

BFSpaths

Há um caminho de s a v?

```
// Método adaptado de DFSpaths.
public boolean hasPath(int v) {
   return distTo[v] < INFINITY;
}

// retorna o número de arcos em um
// caminho mínimo de s a t
public int distTo(int v) {
   return distTo[v];
}
```

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;
}
```

Relações invariantes

No início de cada iteração a fila consiste em

> *zero ou mais vértices à distância* `d` *de* `s`*,*
> *seguidos de zero ou mais vértices à*
> *distância* `d`*+1 de* `s`*,*

para algum `d`

Isto permite concluir que, no início de cada iteração, para todo vértice `x`, se `distTo[x] != G.V()` então `distTo[x]` é a distância de `s` a `x`

Consumo de tempo

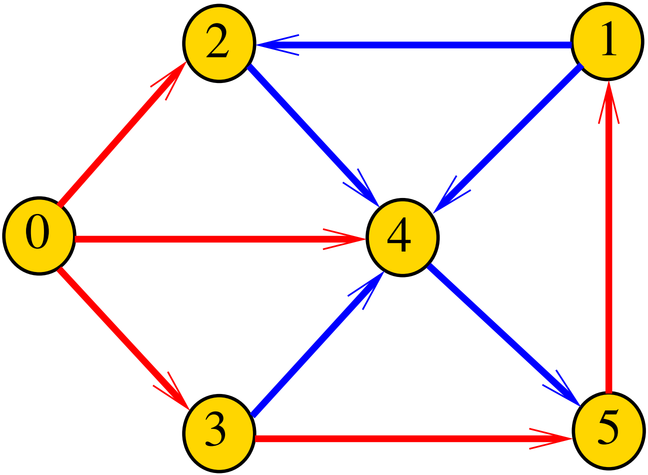
O consumo de tempo de BFSpaths para vetor de listas de adjacência é $O(V + E)$.

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

Arborescência da BFS

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

v	0	1	2	3	4	5
distTo	0	3	1	1	1	2



Condição de inexistência

Se `distTo[t]==INFINITO` para algum vértice `t`, então

$$S = \{v : \mathtt{distTo}[v] < \mathtt{INFINITO}\}$$
$$T = \{v : \mathtt{distTo}[v] == \mathtt{INFINITO}\}$$

formam um `st`-corte (S, T) em que todo arco no corte tem ponta inicial em `T` e ponta final em `S`

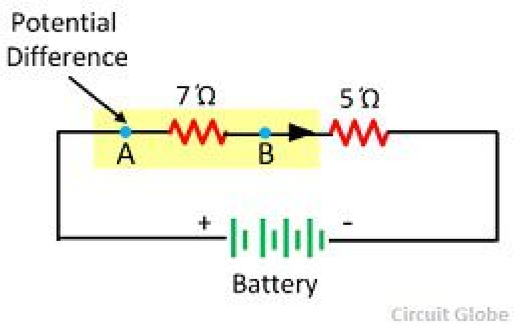
Conclusão

Para quaisquer vértices $\texttt{s}$ e $\texttt{t}$ de um digrafo, vale uma e apenas umas das seguintes afirmações:

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

Fonte: `Yin Yang Bonsai vector image`

Apêndice: 1-potenciais



Fonte: Difference Between Electromotive Force & Potential Difference

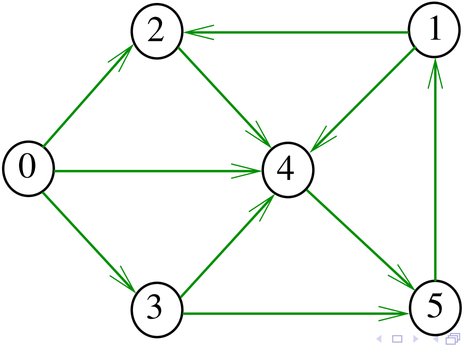
1-potenciais

Um **1-potencial** é um vetor y indexado pelos vértices do digrafo tal que

$$y[w] - y[v] \leq 1 \text{ para todo arco } v\text{-}w$$

Exemplo:

v	0	1	2	3	4	5
y[v]	1	1	1	1	1	1



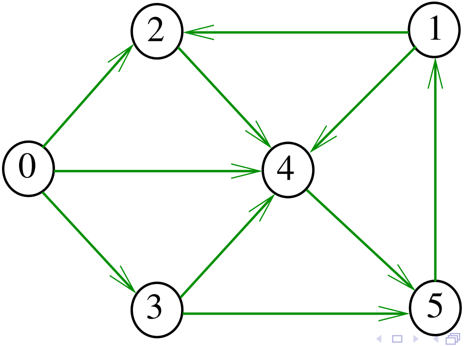
1-potenciais

Um **1-potencial** é um vetor y indexado pelos vértices do digrafo tal que

$$y[w] - y[v] \leq 1 \text{ para todo arco } v\text{-}w$$

Exemplo:

v	0	1	2	3	4	5
y[v]	1	2	2	1	1	2



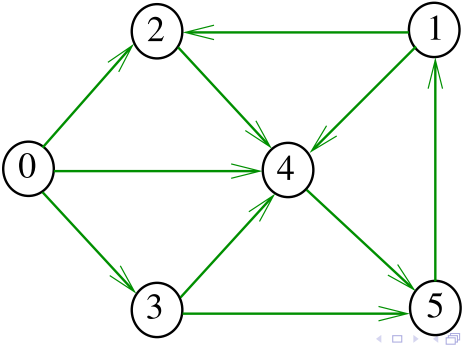
Propriedade dos 1-potencias

Lema da dualidade. Se y é um 1-potencial e P é um caminho de s a t, então

$$y[t] - y[s] \leq |P|$$

Exemplo:

v	0	1	2	3	4	5
y[v]	6	6	6	7	7	7



Conseqüência

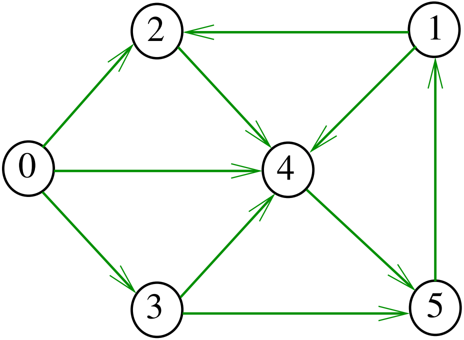
Se $\mathtt{P}$ é um caminho de $\mathtt{s}$ a $\mathtt{t}$ e $\mathtt{y}$ é um 1-potencial tais que

$$|\mathtt{P}| = \mathtt{y}[\mathtt{t}] - \mathtt{y}[\mathtt{s}],$$

então $\mathtt{P}$ é um caminho **mínimo** e $\mathtt{y}$ é um 1-potencial tal que $\mathtt{y}[\mathtt{t}] - \mathtt{y}[\mathtt{s}]$ é **máximo**

Exemplo

v	0	1	2	3	4	5
y[v]	0	3	1	1	1	2



Invariantes

Abaixo está escrito `y` no papel de `distTo[]`
Na classe `BFSpaths`, no início `while` do método `bfs()` valem as seguintes invariantes:

(i0) para cada arco `v-w` na arborescência BFS tem-se que $\mathtt{y}[\mathtt{w}] - \mathtt{y}[\mathtt{v}] = 1$;

(i1) $\mathtt{edgeTo}[\mathtt{s}] = \mathtt{s}$ e $\mathtt{y}[\mathtt{s}] = 0$;

(i2) para cada vértice `v`,
$\mathtt{y}[\mathtt{v}] \neq \mathtt{G.V()} \Leftrightarrow \mathtt{edgeTo}[\mathtt{v}] \neq -1$;

(i3) para cada vértice `v`, se $\mathtt{y}[\mathtt{v}] \neq \mathtt{G.V}()$ então existe um caminho de `s` a `v` na arborescência BFS.

Invariantes (continuação)

Abaixo está escrito y no papel de `distTo[]`
Na linha

```
int v = q.dequeue();
```

do método `bfs()` vale a seguinte relação invariante:

(i4) para cada arco `v-w` se

$$y[w] - y[v] > 1$$

então `v` está na fila.

Correção de BFSpaths

Início da última iteração:

- $\mathtt{y}$ é um 1-potencial, por (i4)
- se $\mathtt{y}[\mathtt{t}] \neq \mathtt{G.V}()$, então $\mathtt{edgeTo}[\mathtt{t}] \neq -1$ [(i2)]. Logo, de (i3), segue que existe um $\mathtt{st}$-caminho $\mathtt{P}$ na arborescência $\mathtt{BFS}$. (i0) e (i1) implicam que
$$|\mathtt{P}| = \mathtt{y}[\mathtt{t}] - \mathtt{y}[\mathtt{s}] = \mathtt{y}[\mathtt{t}].$$
Da propriedade dos 1-potencias, concluímos que $\mathtt{P}$ é um $\mathtt{st}$-caminho de comprimento mínimo
- se $\mathtt{y}[\mathtt{t}] = \mathtt{G.V}()$, então (i1) implica que $\mathtt{y}[\mathtt{t}] - \mathtt{y}[\mathtt{s}] = \mathtt{G.V}()$ e da propriedade dos 1-potencias concluímos que não existe caminho de $\mathtt{s}$ a $\mathtt{t}$ no grafo

Tipo teorema da dualidade

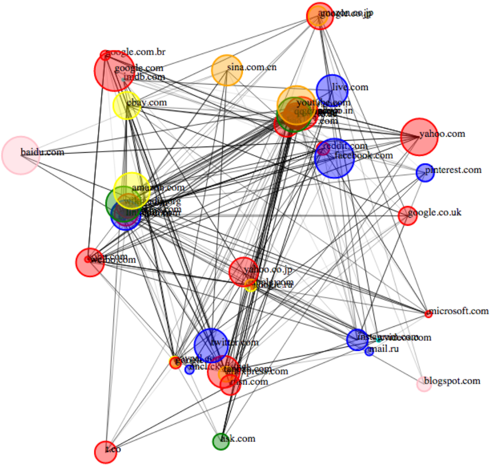
Da propriedade dos 1-potenciais (lema da dualidade) e da correção de `bfs()` concluímos o seguinte:

Se s e t são vértices de um digrafo e t está ao alcance de s então

$$\min\{|\mathtt{P}| : \mathtt{P} \text{ é um } \mathtt{st}\text{-caminho}\}$$
$$= \max\{\mathtt{y}[\mathtt{t}] - \mathtt{y}[\mathtt{s}] : \mathtt{y} \text{ é um } 1\text{-potencial}\}.$$

Fonte: Yin Yang Meaning

Custos nos arcos



Fonte: Force-directed graph drawing

Digrafos com custos nos arcos

Muitas aplicações associam um número a cada arco de um digrafo

Diremos que esse número é o **custo** ou **peso** do arco

Vamos supor que esses números são do tipo `double` na classe `Arc`.

```
private final double weight;
```

Classe Arc: esqueleto

```
public class Arc {
   private final int v;
   private final int w;
   private final double weight;

   public Arc(int v, int w,
                    double weight) {...}
   public int from() {...}
   public int to() {...}
   public double weight() {...}
   public String toString() {...}
}
```

Arc

O construtor `Arc` recebe dois vértices `v` e `w` e um valor `weight` e produz a representação de um arco com ponta inicial `v` e ponta final `w` e peso `weight`.

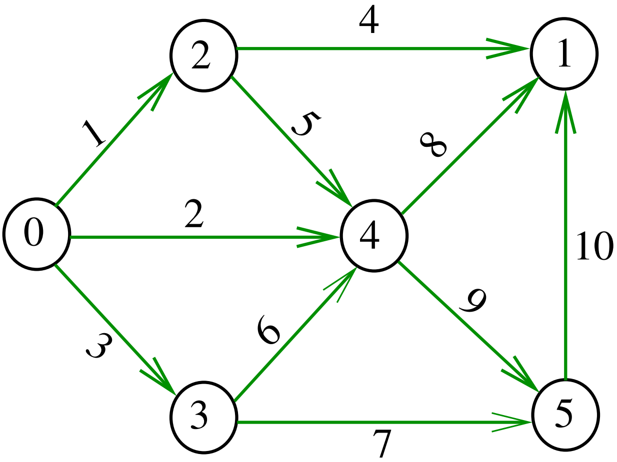
```
public Arc(int v, int w,
           double weight) {
   this.v = v;
   this.w = w;
   this.weight = weight;
}
```

Arc

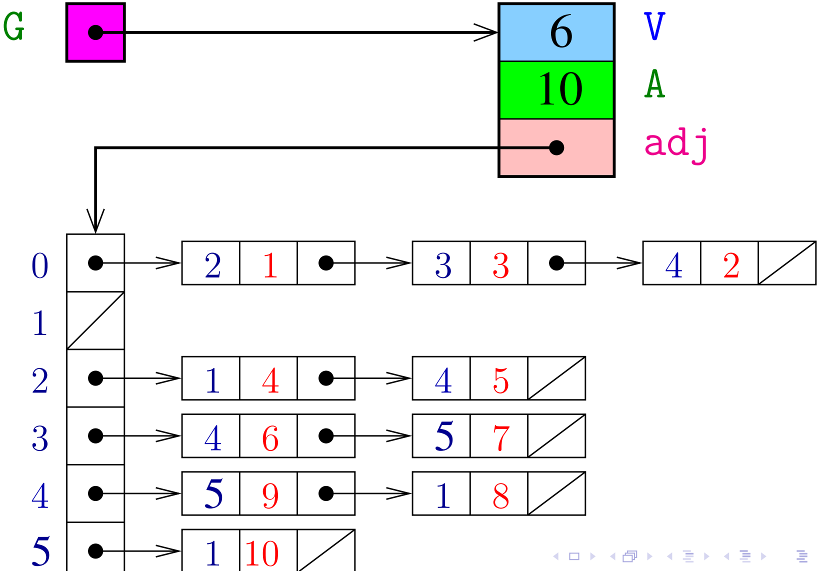
```
// retorna a ponta inicial do arco
public int from() {
   return v;
}
// retorna a ponta final do arco
public int to() {
   return w;
}
// retorna o custo/peso do arco
public double weight() {
   return weight;
}
```

Digrafo com pesos

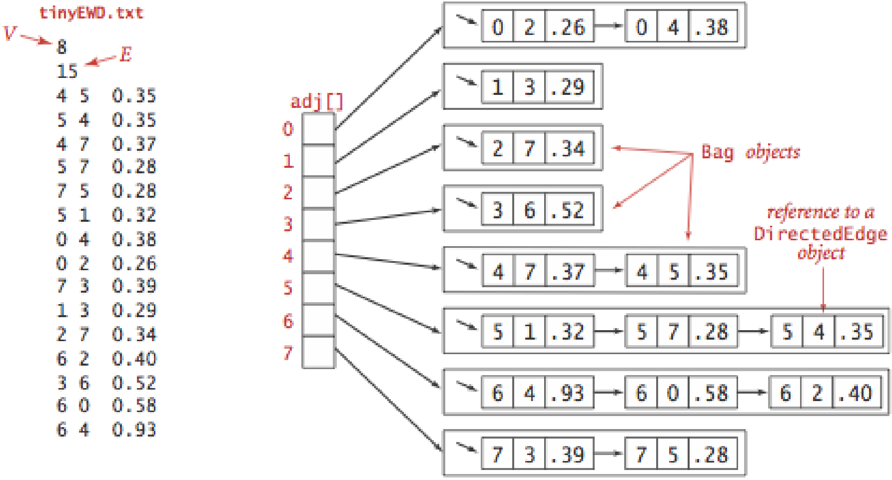
EdgeWeightedDigraph G



Estruturas de dados



Enquanto isso... no algs4



Edge-weighted digraph representation

Classe

A estrutura `EdgeWeightedDigraph` do `algs4` representa um digrafo com pesos nos arcos.

- `V` é o número de vértices
- `E` é o número de arcos do digrafo
- `adj` é uma referência para vetor de listas de adjacência

A lista de adjacência de um vértice `v` é composta por nós do tipo `Arc`.

Um `next` de `Bag` é uma referência para um `Arc`

Cada nó da lista contém `v`, um vizinho `w` de `v`, **o custo** do arco `v-w` e o endereço do nó seguinte da lista

Classe EWDigraph: esqueleto

```
public class EWDigraph{
   private final int V;
   private int E;
   private Bag<Arc>[] adj;
   private int[] indegree;
   public EWDigraph(int V) {...}
   public int V() { return V; }
   public int E() { return E; }
   public void addEdge(int v, int w) { }
   public Iterable<Integer> adj(intv) { }
   public int outdegree(int v) {...}
   public int indegree(int v) {...}
}
```

Construtor de EWDigraph

Constrói um digrafo com V vértice e zero arcos.

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

V() e E()

```
public int V() {
      return V;
}
public int E() {
      return E;
}
```

addEdge() e adj()

Insere um arco e no digrafo G.

```
public addEdge(Arc e) {
    int v = e.from();
    int w = e.to();
    adj[v].add(e);
    E++;
}

public Iterable<Arc> adj(int v) {
    return adj[v];
}
```

edges()

O código abaixo retorna os arcos em G como uma coleção iterável.

```
public Iterable<Arc> edges() {
   Bag<Arc> bag;
   bag = new Bag<Arc>();
   for (int v = 0; v < V; v++)
       for (Arc e:  adj[v])
           bag.add(e);
   return bag;
}
```

Caminhos de custo mínimo

Fonte: The Shortest Distance WoW Achievement Fast

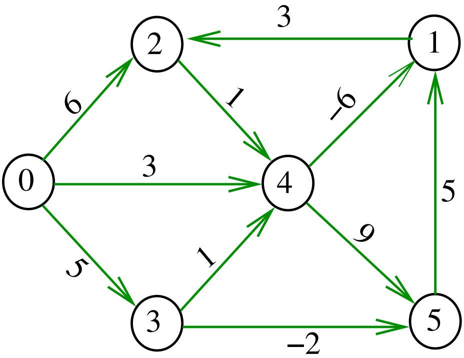
Custo de um caminho

Custo de um caminho é soma dos custos de seus arcos

Custo do caminho 0–2–4–5 é 16.

Custo do caminho 0–2–4–1–2–4–5 é 14.

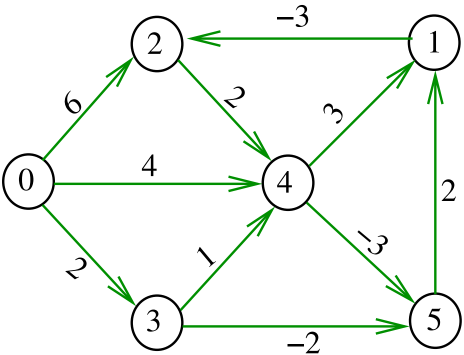
Custo do caminho 0–2–4–1–2–4–1–2–4–5 é 12.



Caminho mínimo

Um caminho `P` tem **custo mínimo** se o custo de `P` é menor ou igual ao custo de todo caminho com a mesma origem e término

O caminho `0-3-4-5-1-2` é mínimo, tem custo -1



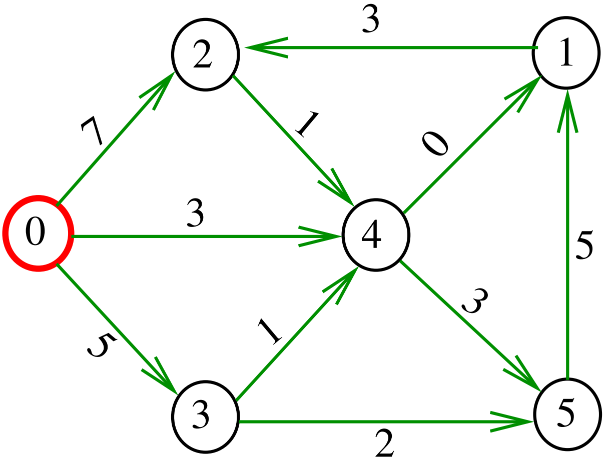
Problema

Problema dos Caminhos Mínimos com Origem Fixa (*Single-source Shortest Paths Problem*):

> *Dado um vértice s de um digrafo com custos **não-negativos** nos arcos, encontrar, para cada vértice t que pode ser alcançado a partir de s, um caminho mínimo simples de s a t.*

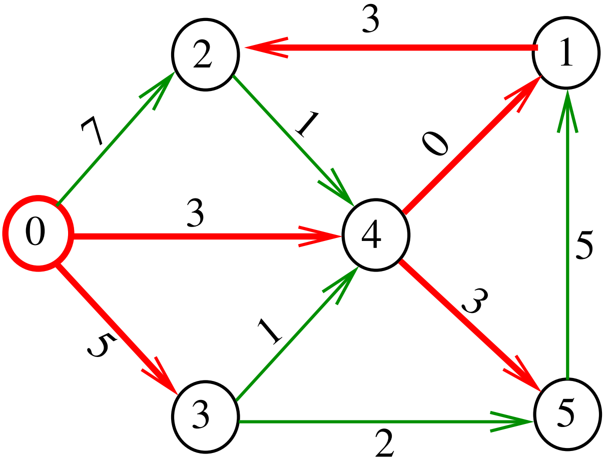
Exemplo

Entra:



Exemplo

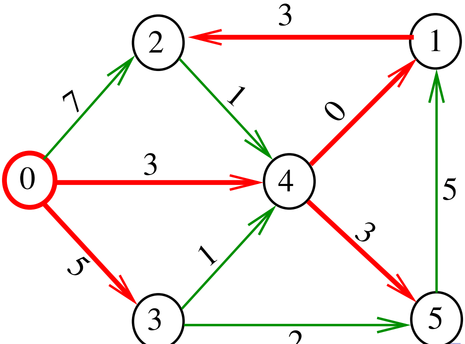
Sai:



Arborescência de caminhos mínimos

Uma arborescência com raiz **s** é de **caminhos mínimos** (= *shortest-paths tree* = *SPT*) se para todo vértice **t** que pode ser alcançado a partir de **s**,

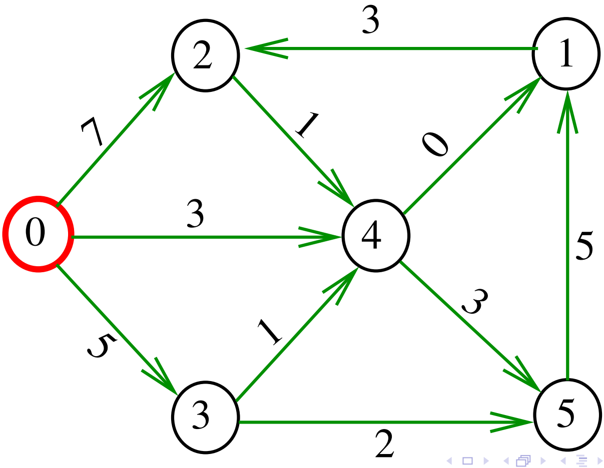
> *o único caminho de s a t na arborescência é um caminho mínimo*



Problema da SPT

Problema: Dado um vértice s de um digrafo com custos **não-negativos** nos arcos, encontrar uma SPT com raiz s

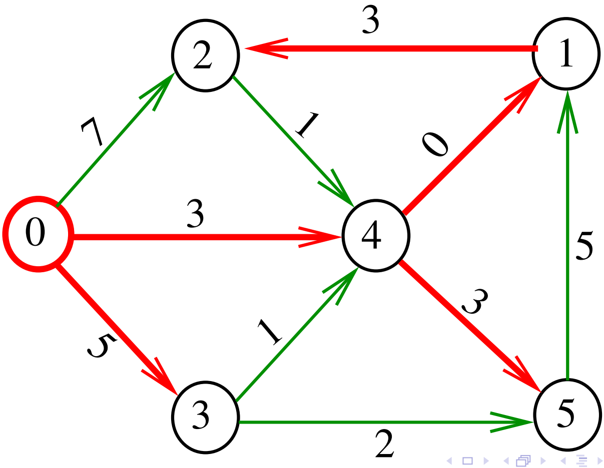
Entra:



Problema da SPT

Problema: Dado um vértice s de um digrafo com custos **não-negativos** nos arcos, encontrar uma SPT com raiz s

Sai:



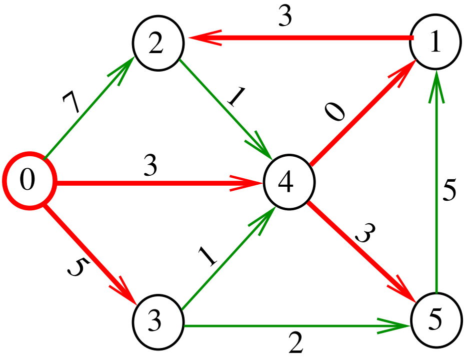
Algoritmo de Dijkstra

Fonte: What S So Great About A World Flight Paths Map Spatial Ly In

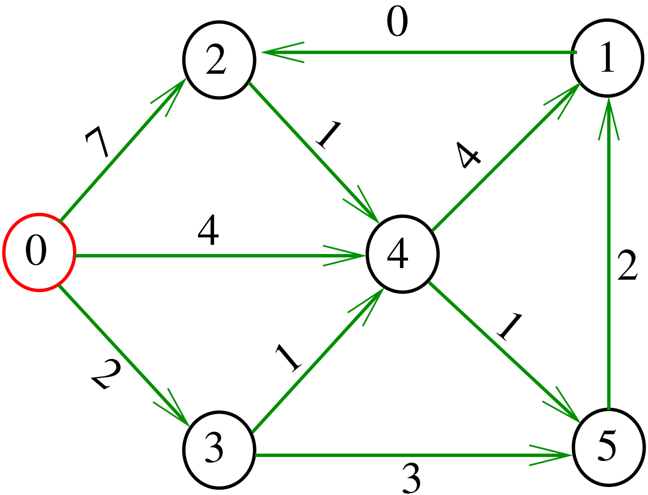
Problema

O algoritmo de Dijkstra resolve o problema da SPT:

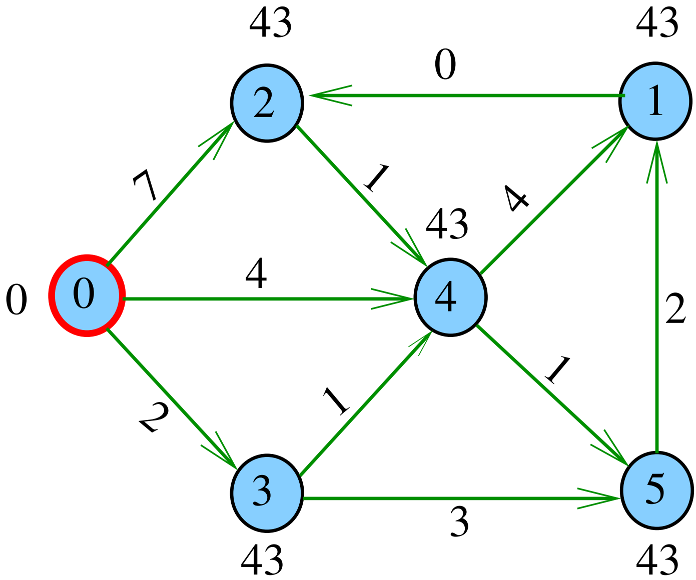
> *Dado um vértice s de um digrafo com custos não-negativos nos arcos, encontrar uma SPT com raiz s*



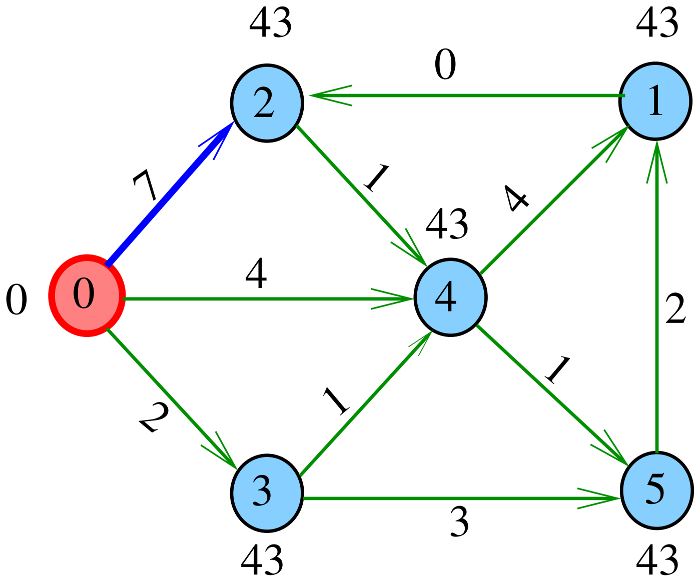
Simulação



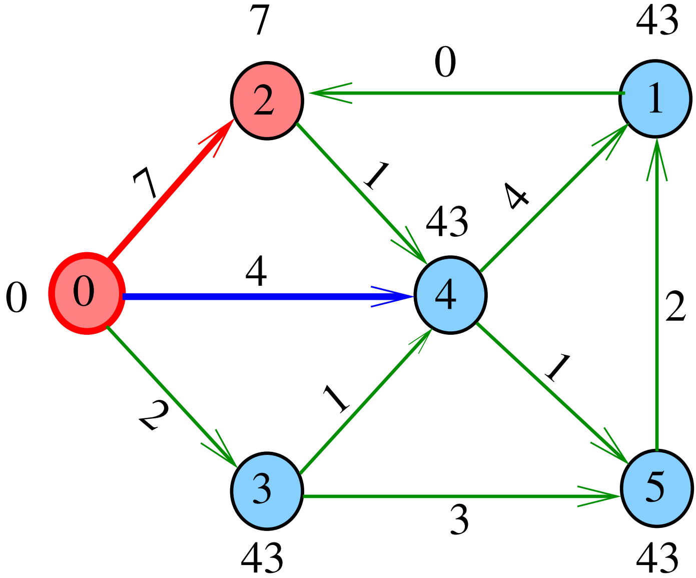
Simulação



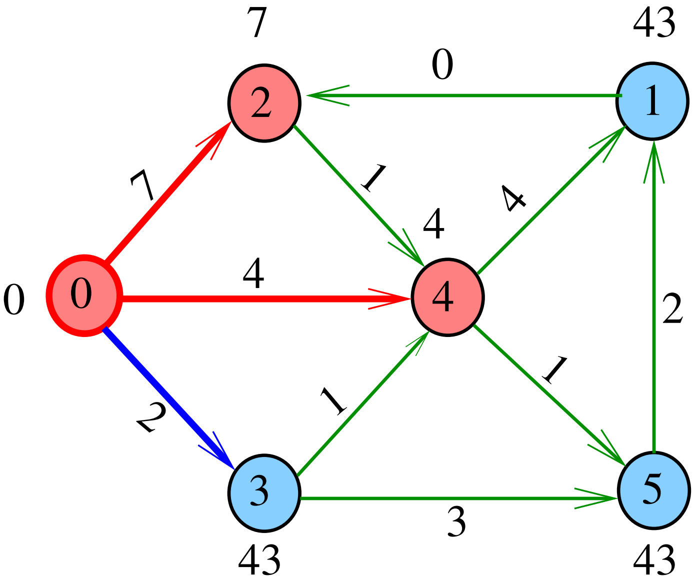
Simulação



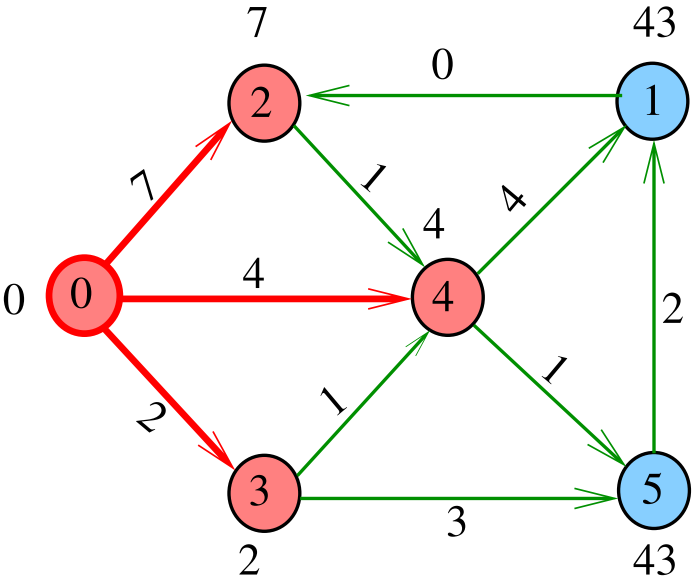
Simulação



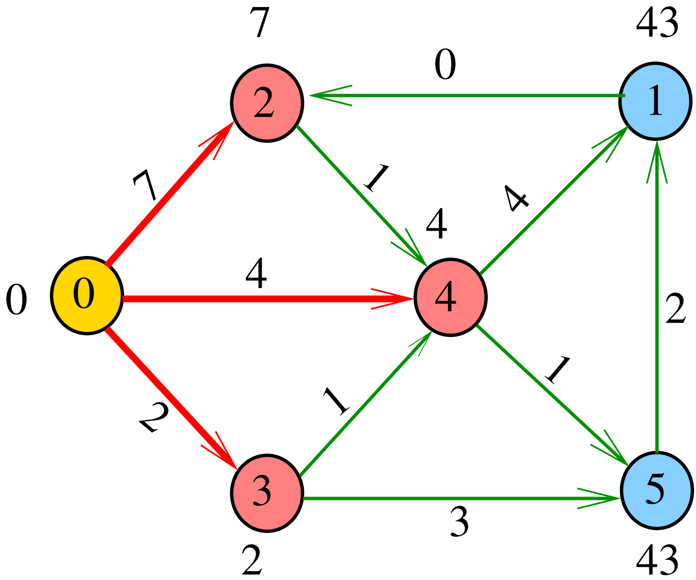
Simulação



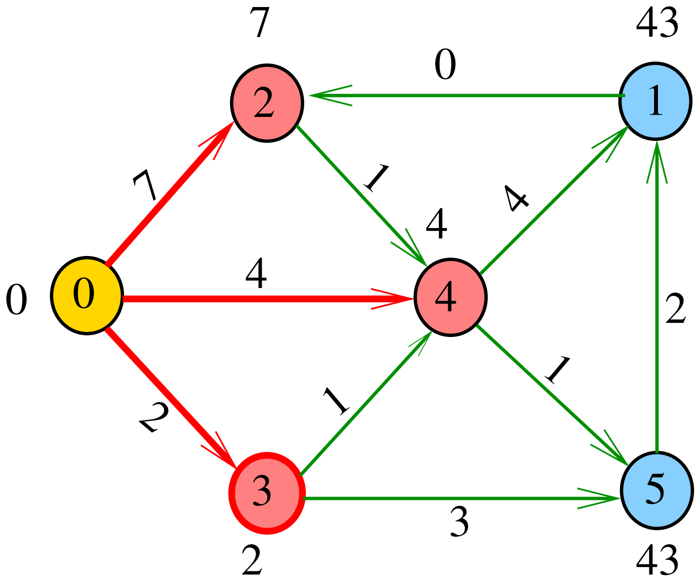
Simulação



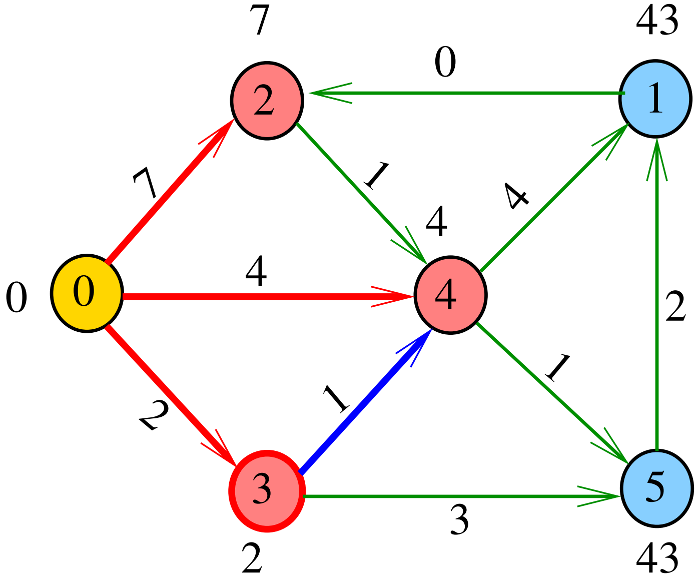
Simulação



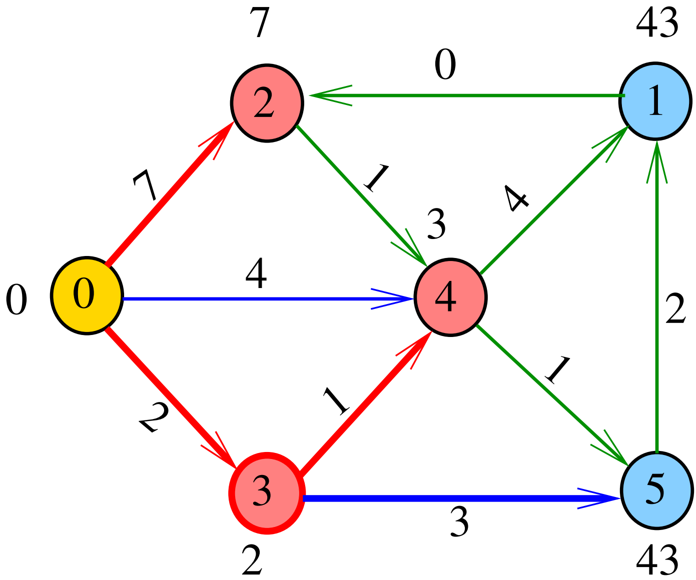
Simulação



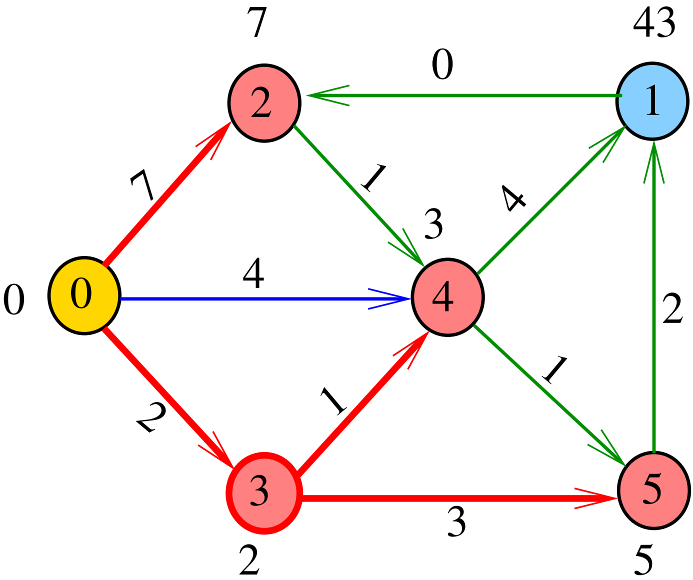
Simulação



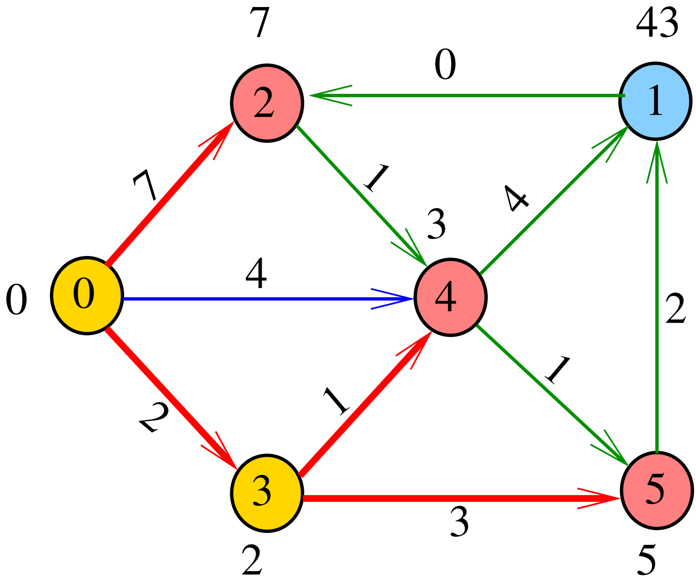
Simulação



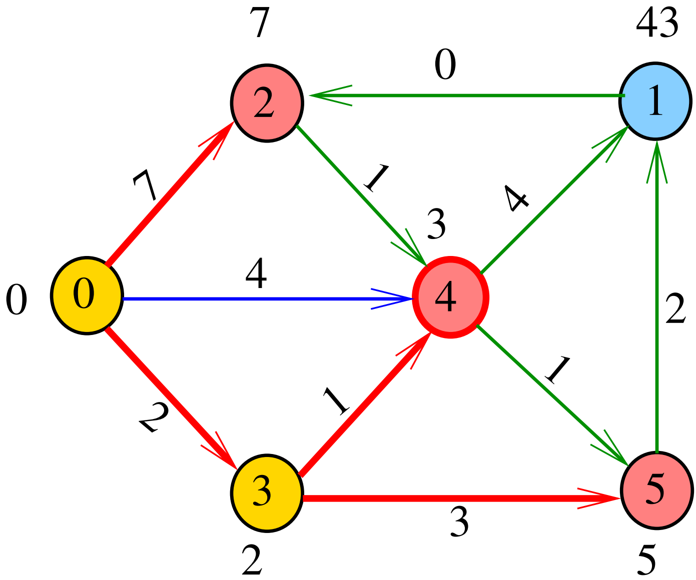
Simulação



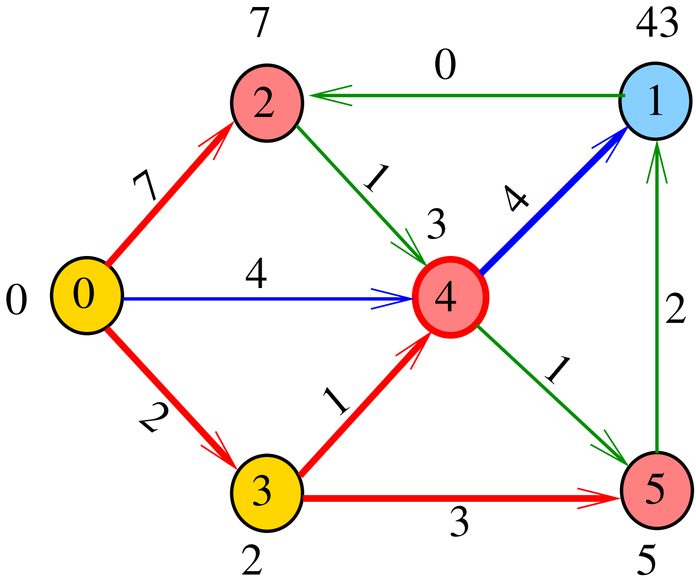
Simulação



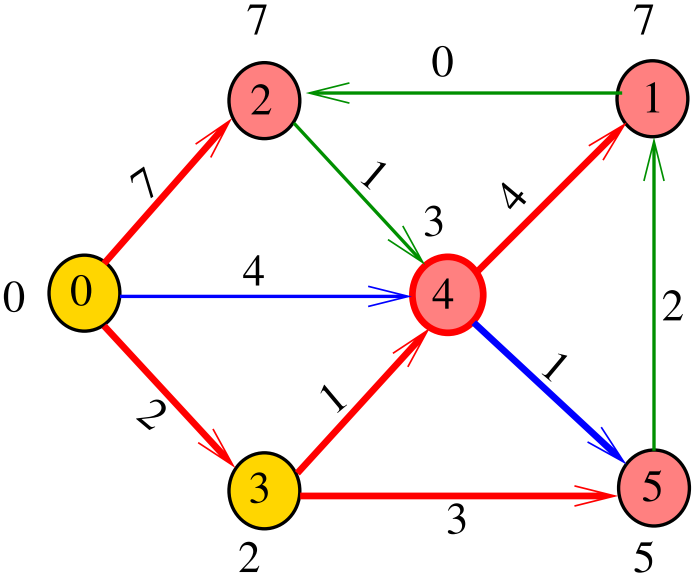
Simulação



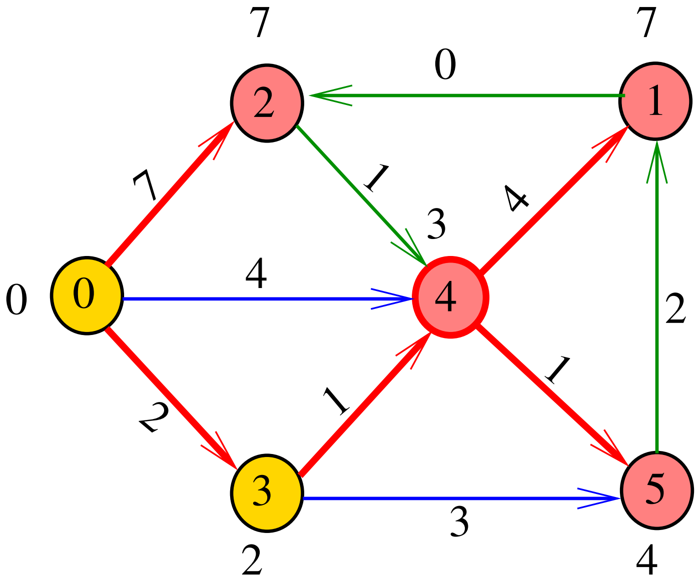
Simulação



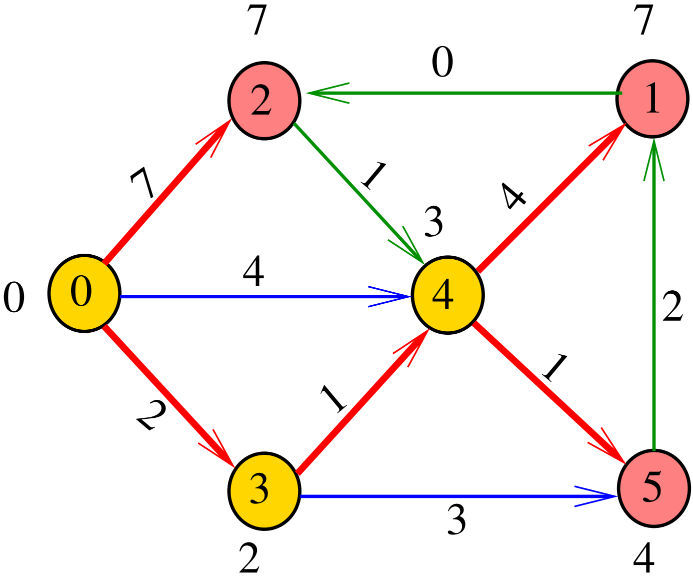
Simulação



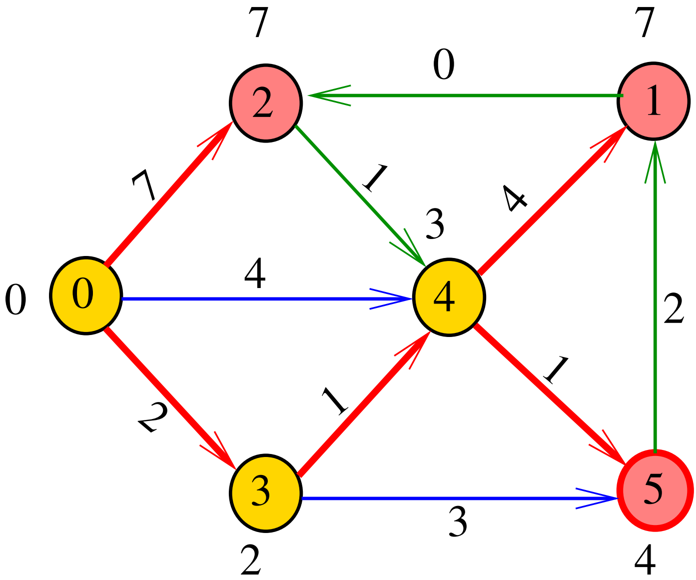
Simulação



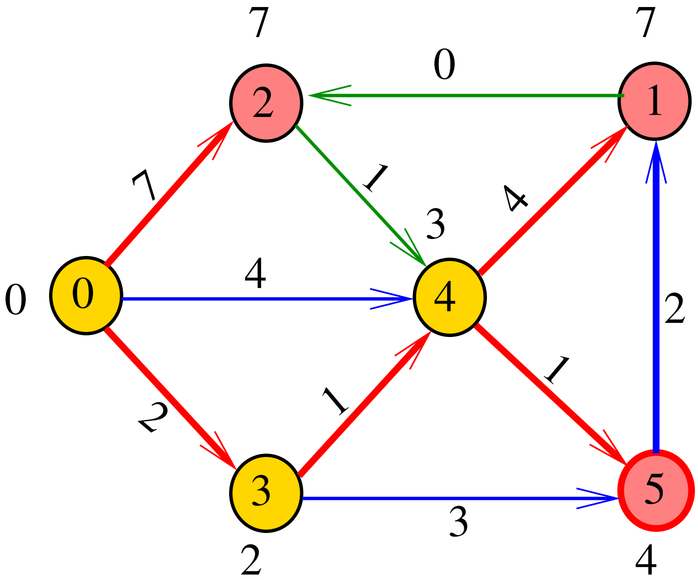
Simulação



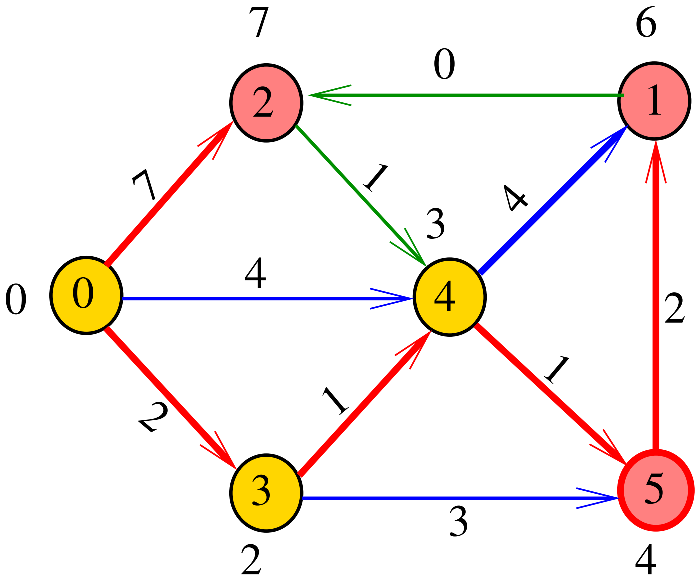
Simulação



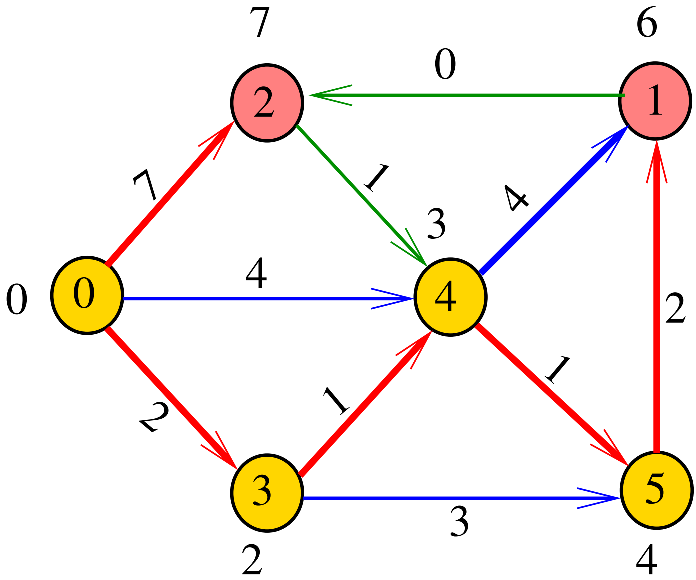
Simulação



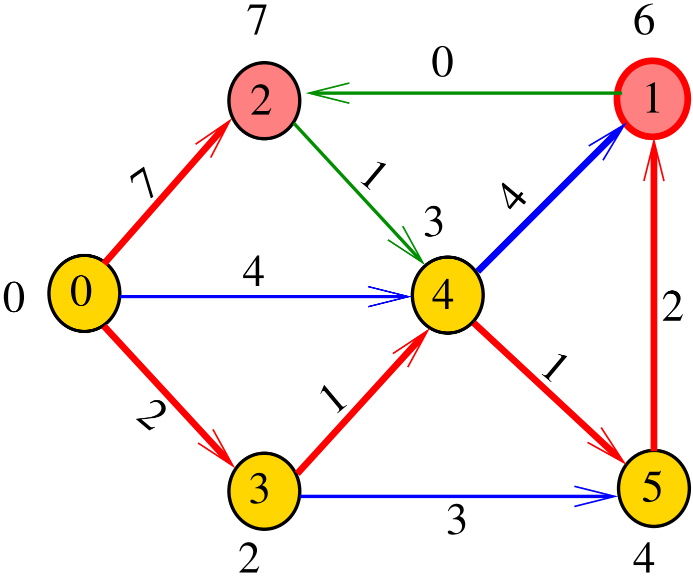
Simulação



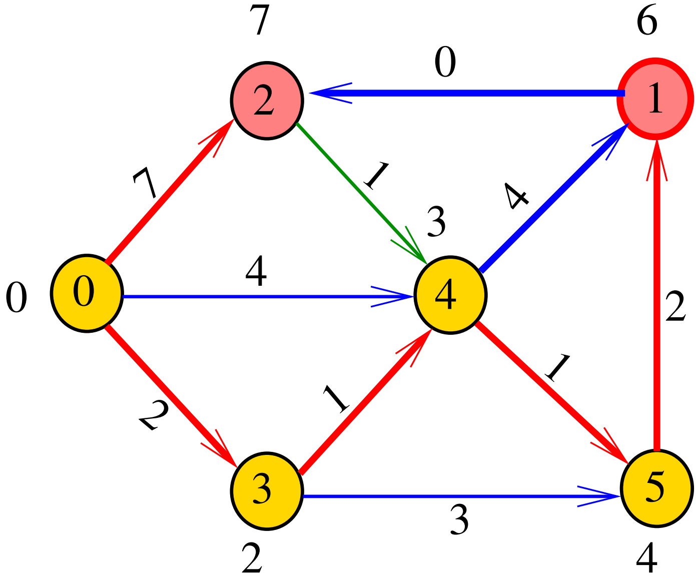
Simulação



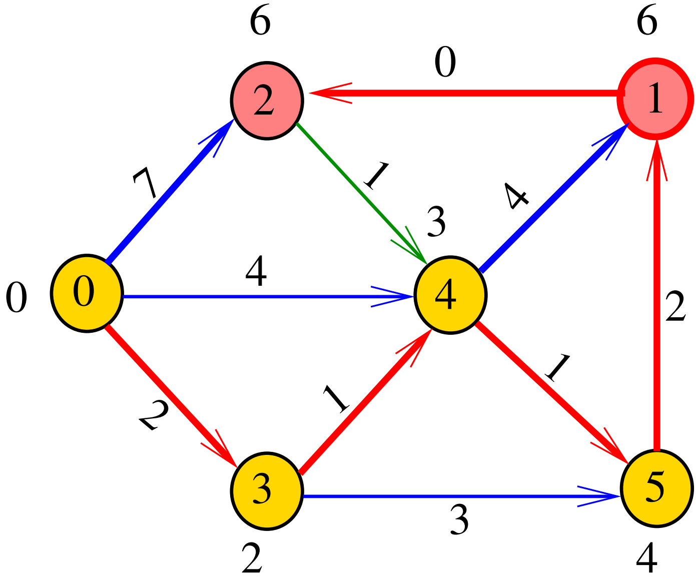
Simulação



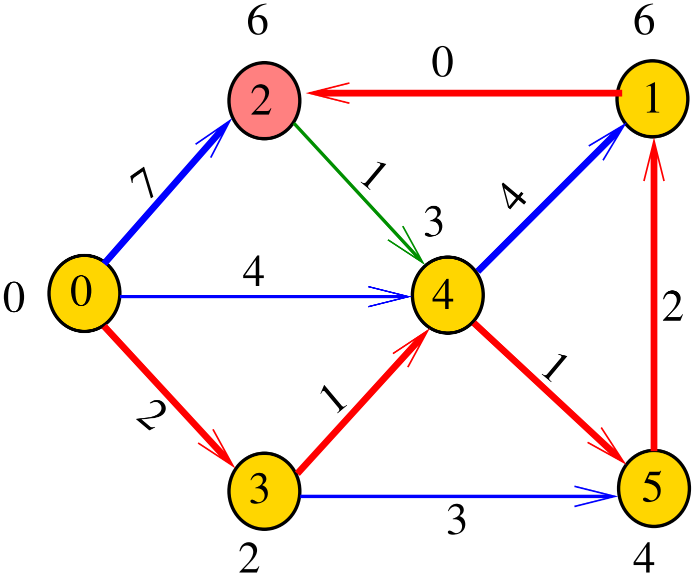
Simulação



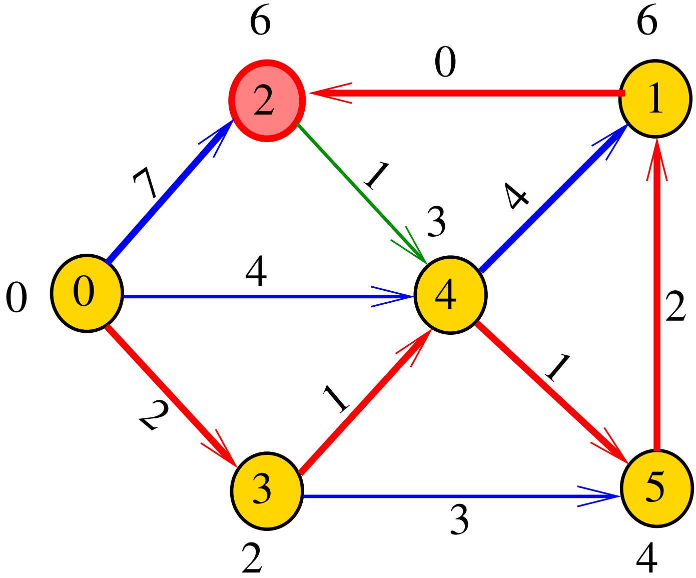
Simulação



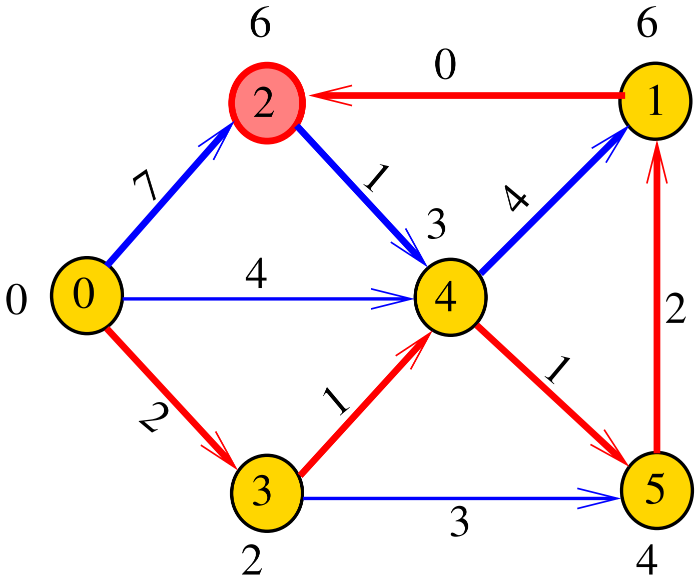
Simulação



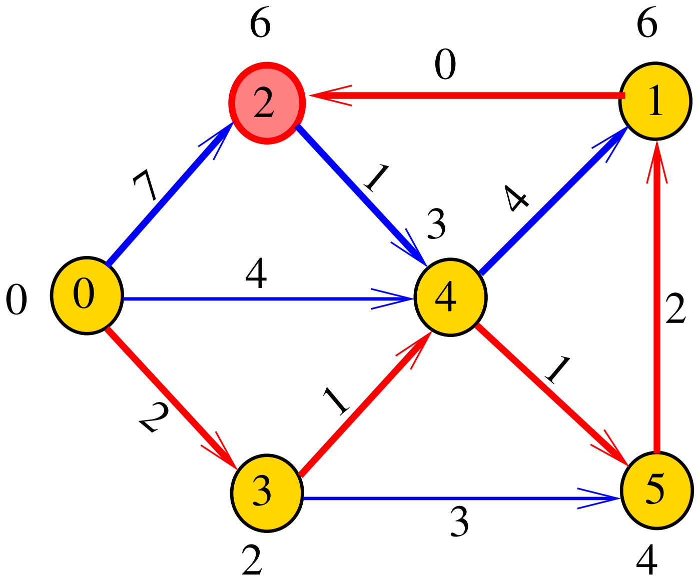
Simulação



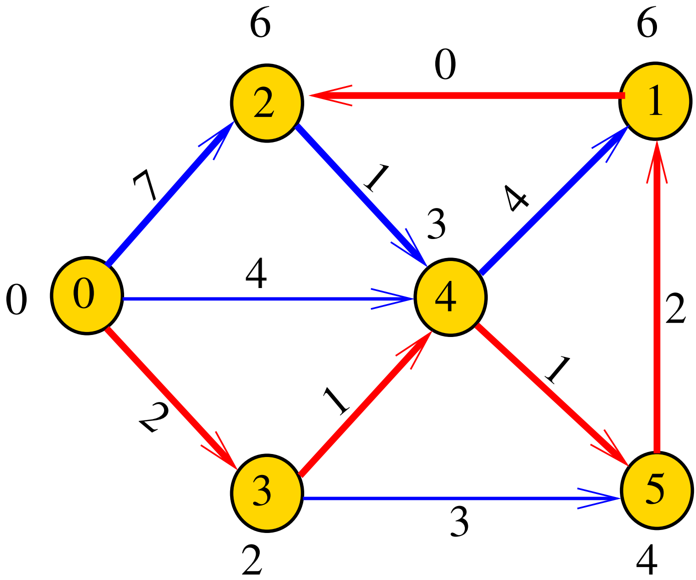
Simulação



Simulação



Simulação



Dijkstra

Recebe digrafo `G` com custos não-negativos nos arcos e um vértice `s`

Calcula uma arborescência de caminhos mínimos com raiz `s`.

A arborescência é armazenada no vetor `edgeTo[]`.

As distâncias em relação a `s` são armazenadas no vetor `distTo[]`

```
private double[] distTo;
private DirectedEdge[] edgeTo;
```

Fila priorizadas

A classe `Dijkstra` usa uma fila priorizada

```
private IndexMinPQ<Double> pq;
```

A fila é manipulada pelos métodos:

- `IndexMaxPQ<Double>()`: fila de vértices em que cada vértice `v` tem prioridade `distTo[v]`
- `isEmpty()`: a fila está vazia?
- `contains(v)`: `v` está na fila?
- `insert(v,valor)`: insere `v` com prior. `valor`
- `delMin()`: retorna um vértice de prioridade mínima.
- `decreaseKey(w,valor)`: reorganiza a fila depois que o valor de `distTo[w]` foi decrementado.

Classe Dijkstra: esqueleto

```
public class Dijkstra {
   private static final int INFINITY;
   private final int s;
   private Arc[] edgeTo;
   private double[] distTo;

   public Dijkstra(EWDigraph G, int s) {}
   private void dijkstra(EWDigraph G,
                         int s) {}
   public boolean hasPath(int v) {}
   public boolean distTo(int v) {}
   public Iterable<Arc> pathTo(int v)
}
```

Dijkstra

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

```
public Dijkstra(EWDigraph G, int s) {
   INFINITY = Double.POSITIVE_INFINITY;
   edgeTo = new Arc[G.V()];
   distTo = new double[G.V()];
   this.s = s;
   for (int v = 0; v < G.V(); v++)
      distTo[v] = INFINITY;
   dijkstra(G, s);
}
```

dijkstra(): inicializações

```
private void dijkstra(EWDigraph G, int s)
{
   IndexMaxPQ<Double> pq =
         new IndexMaxPQ<Double>(G.V());
   distTo[s] = 0;
   pq.insert(s, distTo[s]);

   // aqui vem a iteração do próximo slide
```

bfs(): iteração

```
while (!pq.isEmpty()) {
    int v = pq.delMin();
    for (Arc e :  G.adj(v)) {
        int v = e.from(), w = e.to();
        Double d = distTo[v]+e.weight();
        if (distTo[w] > d)
            edgeTo[w] = e;
            distTo[w] = d;
            if (pq.contains(w))
                pq.decreaseKey(w, d);
            else pq.insert(w, d);
        }
    }
}
```

Dijkstra

Há um caminho de s a v?

```
// Método copiado de BFSpaths.
public boolean hasPath(int v) {
   return distTo[v] < INFINITY;
}

// retorna o comprimento de um
// caminho mínimo de s a t
public int distTo(int v) {
   return distTo[v];
}
```

Dijkstra

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

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

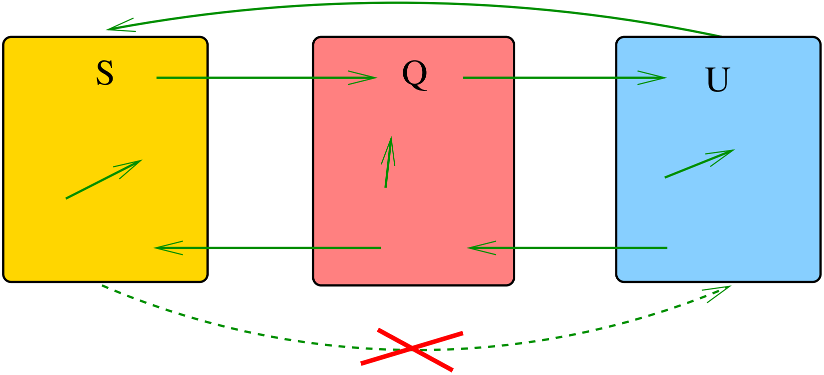
Relações invariantes

S = vértices examinados

Q = vértices visitados = vértices na fila

U = vértices ainda não visitados

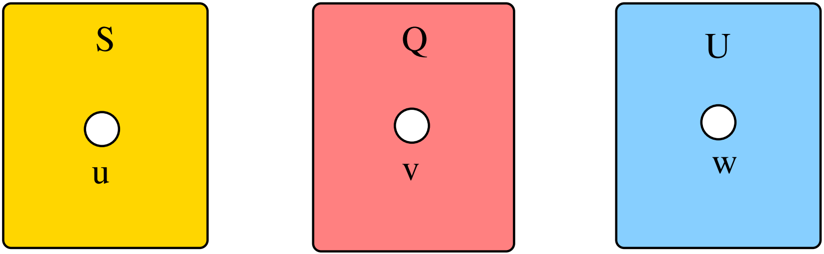
(i0) não existe arco v-w com v em S e w em U



Relações invariantes

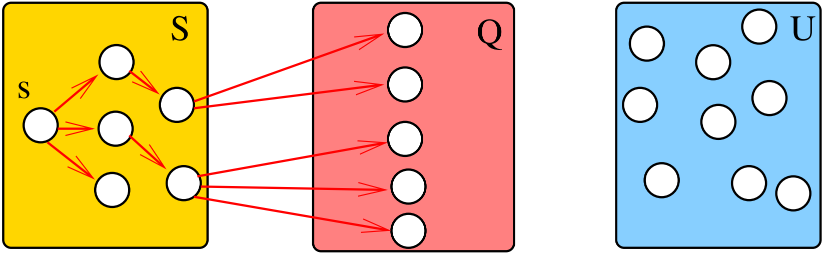
(i1) para cada `u` em `S`, `v` em `Q` e `w` em `U`

$$\texttt{distTo}[\texttt{u}] \leq \texttt{distTo}[\texttt{v}] \leq \texttt{distTo}[\texttt{w}]$$



Relações invariantes

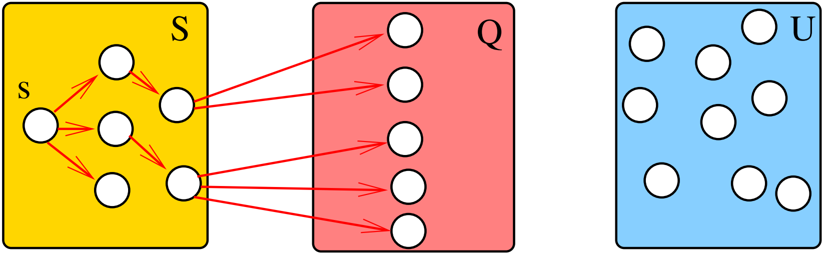
(i2) O vetor `edgeTo` restrito aos vértices de `S` e `Q` determina um árborescência com raiz `s`



Relações invariantes

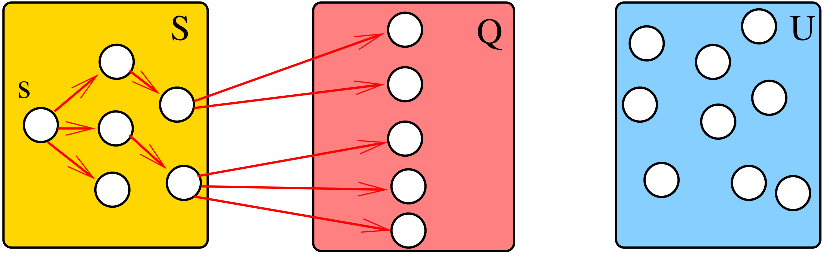
(i3) Para arco `v-w` na arborescência vale que

$$\texttt{distTo}[\texttt{w}] = \texttt{distTo}[\texttt{v}] + \text{custo do arco } \texttt{vw}$$

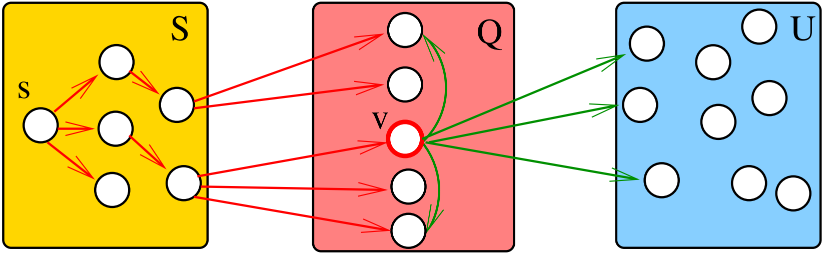


Relações invariantes

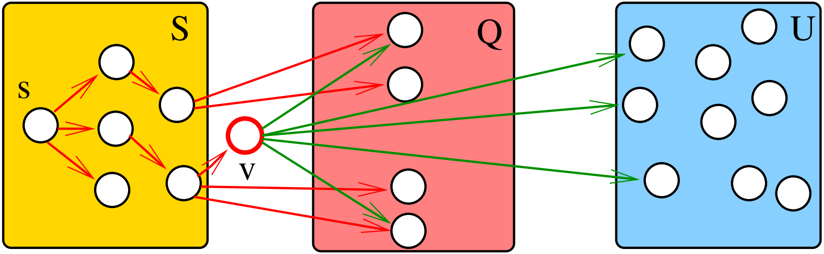
(i3) Para cada vértice `v` em `S` vale que `distTo[v]` é o custo de um caminho mínimo de `s` a `v`.



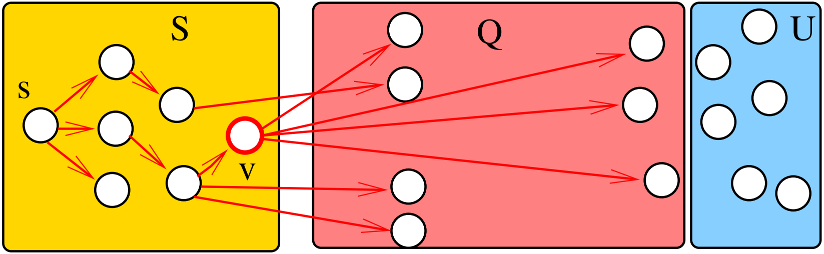
Iteração



Iteração



Iteração



Consumo de tempo

O consumo de tempo da função dijkstra é $O(V + E)$ mais o consumo de tempo de

$=$	1	execução de	IndexMaxPQ<Double>,
$\leq$	V	execuções de	insert(),
$\leq$	V	execuções de	isEmpty(),
$\leq$	V	execuções de	delMin(), e
$\leq$	E	execuções de	contains(),
$\leq$	E	execuções de	decreaseKey().

Consumo de tempo MIN-HEAP

`IndexMaxPQ`	$\Theta(V)$
`isEmpty`	$\Theta(1)$
`insert`	$\Theta(\lg V)$
`delMin`	$O(\lg V)$
`decreaseKey`	$\Theta(\lg V)$
`contains`	$\Theta(1)$

Conclusão

O consumo de **Dijkstra** é $O(E \lg V)$.

Para **grafos densos** podemos alcançar consumo de tempo ótimo . . . detalhes **MAC0328** Algoritmos em Grafos.

Consumo de tempo Min-Heap

	heap	d-heap	fibonacci heap
insert	$O(\lg V)$	$O(\log_D V)$	$O(1)$
delMin	$O(\lg V)$	$O(\log_D V)$	$O(\lg V)$
decreaseKey	$O(\lg V)$	$O(\log_D V)$	$O(1)$
Dijkstra	$O(E \lg V)$	$O(E \log_D V)$	$O(E + V \lg V)$

Consumo de tempo Min-heap

	bucket heap	radix heap
insert	$O(1)$	$O(\lg(VC)R$
delMin	$O(C)$	$O(\lg(VC))$
decreaseKey	$O(1)$	$O(E + V\lg(VC))$
Dijkstra	$O(E + VC)$	$O(E + V\lg(VC))$

$C =$ maior custo de um arco.