

Melhores momentos

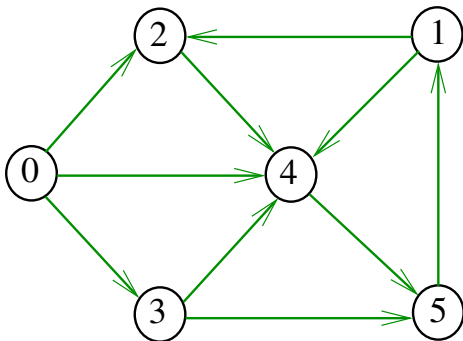
AULA 12

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 |
|------------------|---|---|---|---|---|---|
| $\text{dist}[v]$ | 0 | 3 | 1 | 1 | 1 | 2 |



Busca em largura

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

O algoritmo

visita **s**,

depois visita vértices à *distância 1* de **s**,

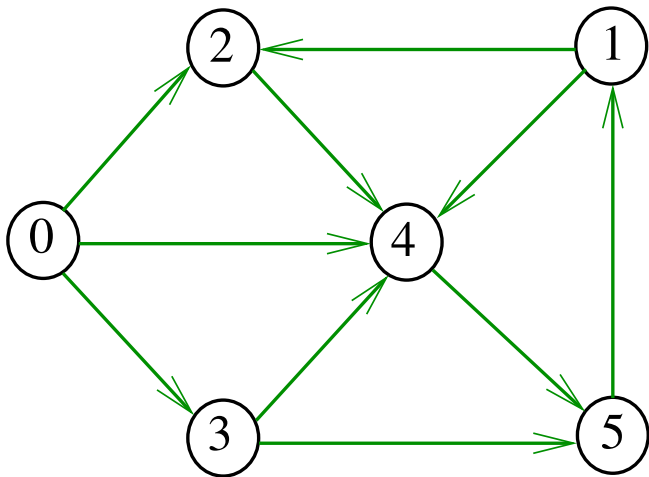
depois visita vértices à *distância 2* de **s**,

depois visita vértices à *distância 3* de **s**,

e assim por diante

Simulação

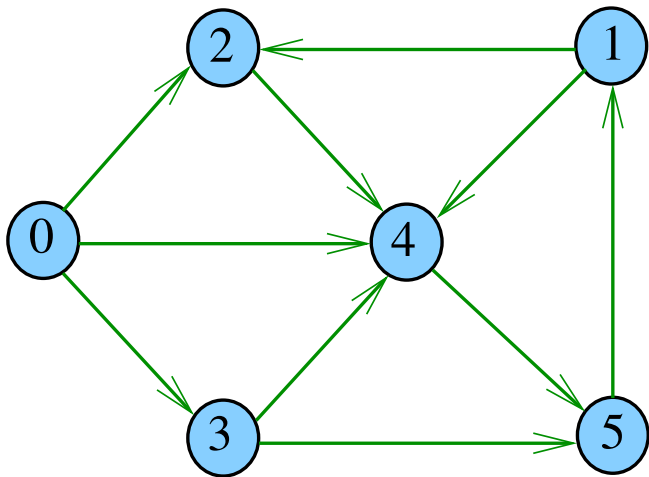
| i | 0 | 1 | 2 | 3 | 4 | 5 | v | 0 | 1 | 2 | 3 | 4 | 5 |
|--------|---|---|---|---|---|---|------------------|---|---|---|---|---|---|
| $q[i]$ | | | | | | | $\text{dist}[v]$ | | | | | | |



Simulação

| i | 0 | 1 | 2 | 3 | 4 | 5 |
|--------|---|---|---|---|---|---|
| $q[i]$ | | | | | | |

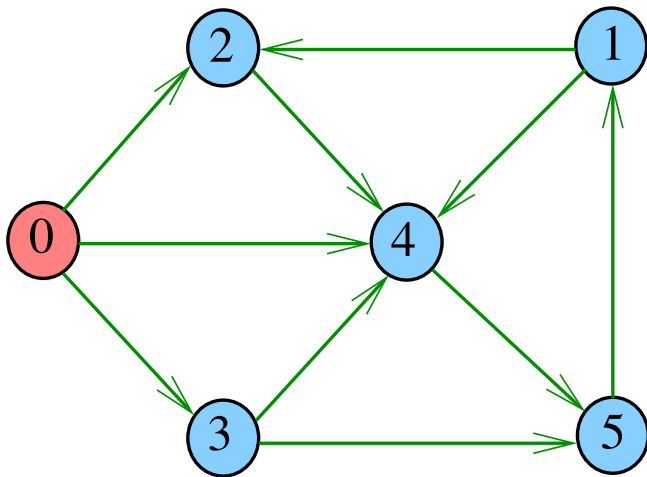
| v | 0 | 1 | 2 | 3 | 4 | 5 |
|------------------|---|---|---|---|---|---|
| $\text{dist}[v]$ | 6 | 6 | 6 | 6 | 6 | 6 |



Simulação

| i | 0 | 1 | 2 | 3 | 4 | 5 |
|--------|---|---|---|---|---|---|
| $q[i]$ | 0 | | | | | |

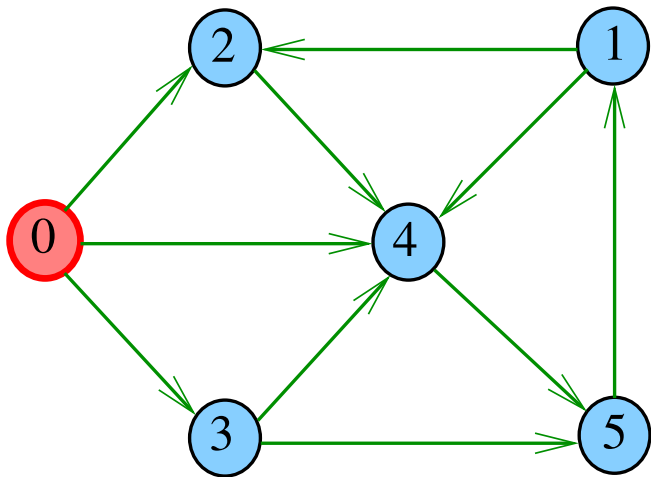
| v | 0 | 1 | 2 | 3 | 4 | 5 |
|------------------|---|---|---|---|---|---|
| $\text{dist}[v]$ | 6 | 6 | 6 | 6 | 6 | 6 |



Simulação

| i | 0 | 1 | 2 | 3 | 4 | 5 |
|--------|---|---|---|---|---|---|
| $q[i]$ | 0 | | | | | |

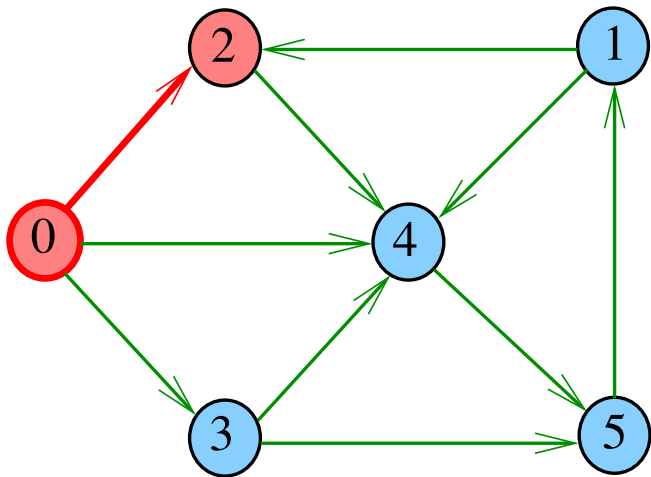
| v | 0 | 1 | 2 | 3 | 4 | 5 |
|------------------|---|---|---|---|---|---|
| $\text{dist}[v]$ | 0 | 6 | 6 | 6 | 6 | 6 |



Simulação

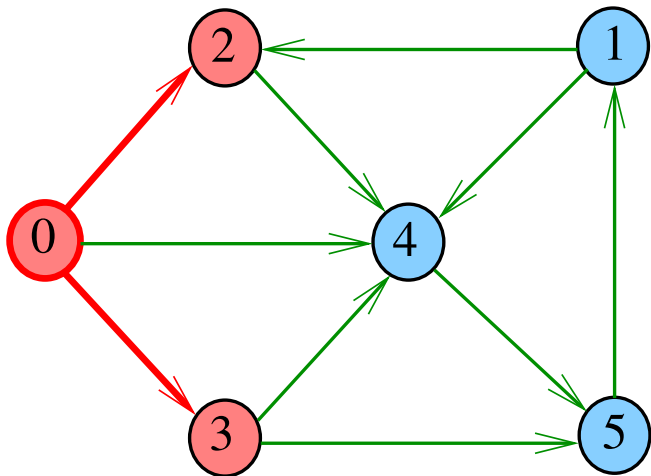
| i | 0 | 1 | 2 | 3 | 4 | 5 |
|--------|---|---|---|---|---|---|
| $q[i]$ | 0 | 2 | | | | |

| v | 0 | 1 | 2 | 3 | 4 | 5 |
|------------------|---|---|---|---|---|---|
| $\text{dist}[v]$ | 0 | 6 | 1 | 6 | 6 | 6 |



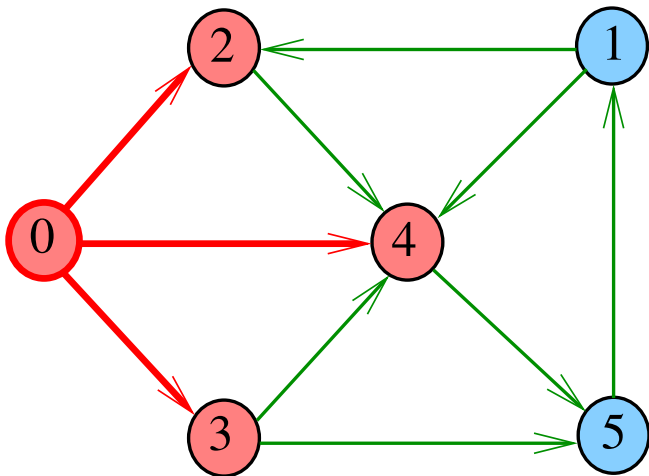
Simulação

| i | 0 | 1 | 2 | 3 | 4 | 5 | v | 0 | 1 | 2 | 3 | 4 | 5 |
|--------|---|---|---|---|---|---|------------------|---|---|---|---|---|---|
| $q[i]$ | 0 | 2 | 3 | | | | $\text{dist}[v]$ | 0 | 6 | 1 | 1 | 6 | 6 |



Simulação

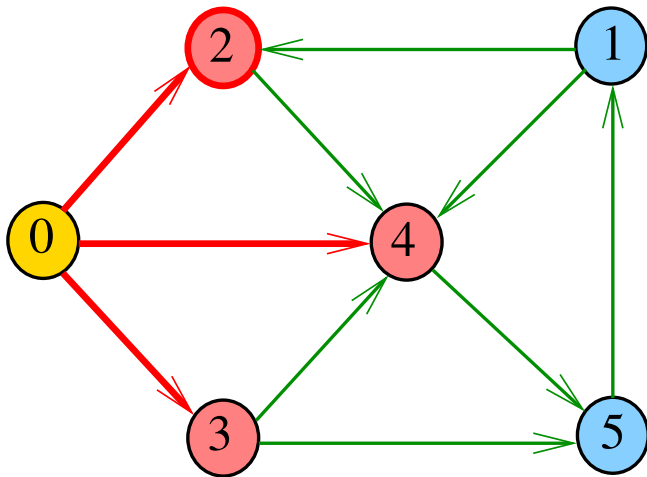
| i | 0 | 1 | 2 | 3 | 4 | 5 | v | 0 | 1 | 2 | 3 | 4 | 5 |
|--------|---|---|---|---|---|---|------------------|---|---|---|---|---|---|
| $q[i]$ | 0 | 2 | 3 | 4 | | | $\text{dist}[v]$ | 0 | 6 | 1 | 1 | 1 | 6 |



Simulação

| i | 0 | 1 | 2 | 3 | 4 | 5 |
|--------|---|---|---|---|---|---|
| $q[i]$ | 0 | 2 | 3 | 4 | | |

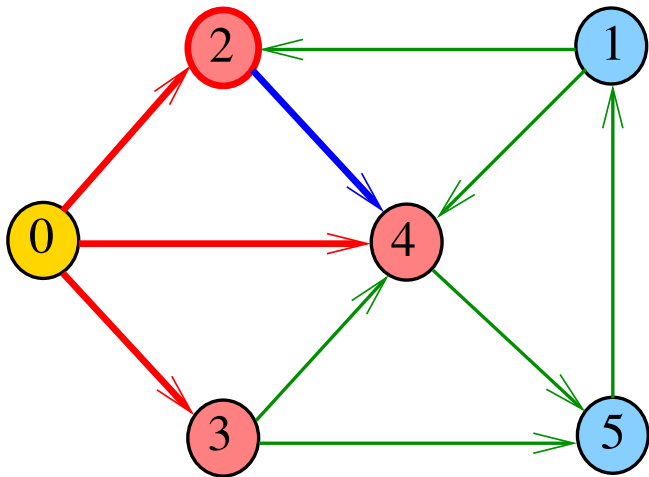
| v | 0 | 1 | 2 | 3 | 4 | 5 |
|------------------|---|---|---|---|---|---|
| $\text{dist}[v]$ | 0 | 6 | 1 | 1 | 1 | 6 |



Simulação

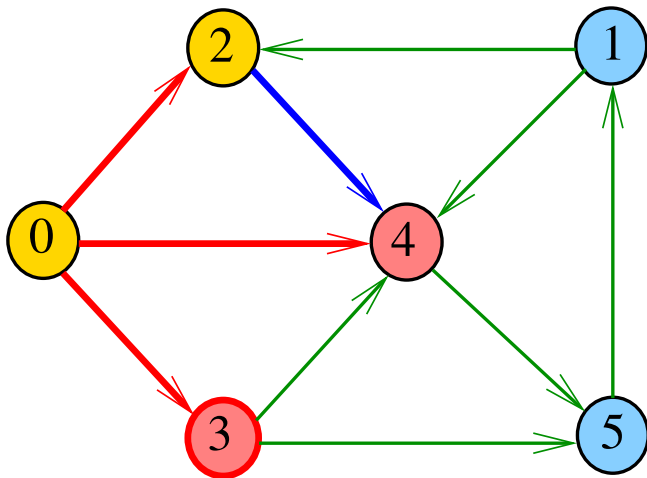
| i | 0 | 1 | 2 | 3 | 4 | 5 |
|--------|---|---|---|---|---|---|
| $q[i]$ | 0 | 2 | 3 | 4 | | |

| v | 0 | 1 | 2 | 3 | 4 | 5 |
|------------------|---|---|---|---|---|---|
| $\text{dist}[v]$ | 0 | 6 | 1 | 1 | 1 | 6 |



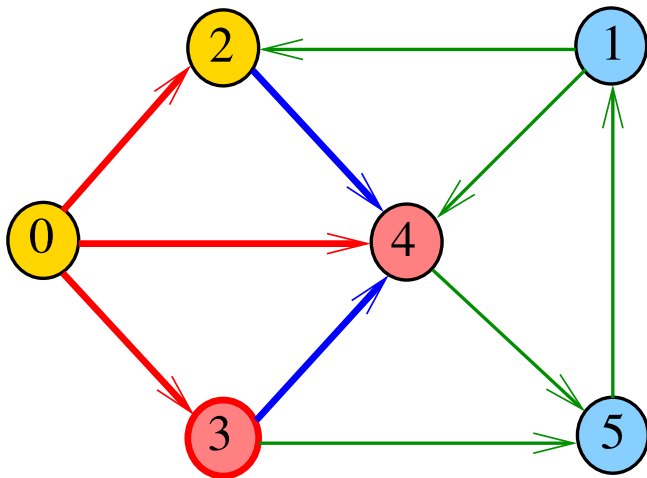
Simulação

| i | 0 | 1 | 2 | 3 | 4 | 5 | v | 0 | 1 | 2 | 3 | 4 | 5 |
|--------|---|---|---|---|---|---|------------------|---|---|---|---|---|---|
| $q[i]$ | 0 | 2 | 3 | 4 | | | $\text{dist}[v]$ | 0 | 6 | 1 | 1 | 1 | 6 |



Simulação

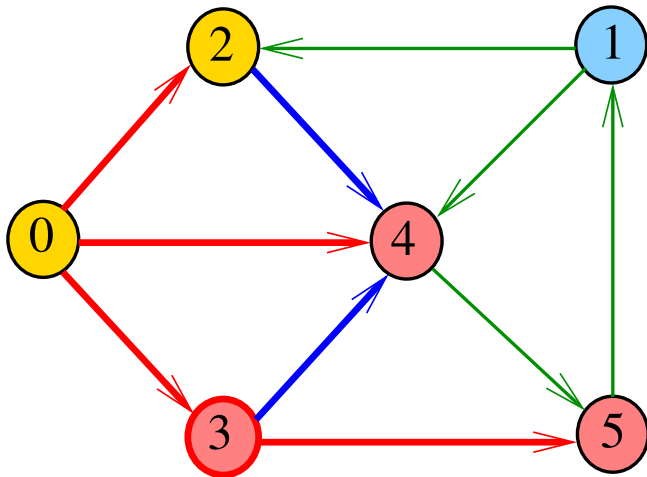
| i | 0 | 1 | 2 | 3 | 4 | 5 | v | 0 | 1 | 2 | 3 | 4 | 5 |
|--------|---|---|---|---|---|---|------------------|---|---|---|---|---|---|
| $q[i]$ | 0 | 2 | 3 | 4 | | | $\text{dist}[v]$ | 0 | 6 | 1 | 1 | 1 | 6 |



Simulação

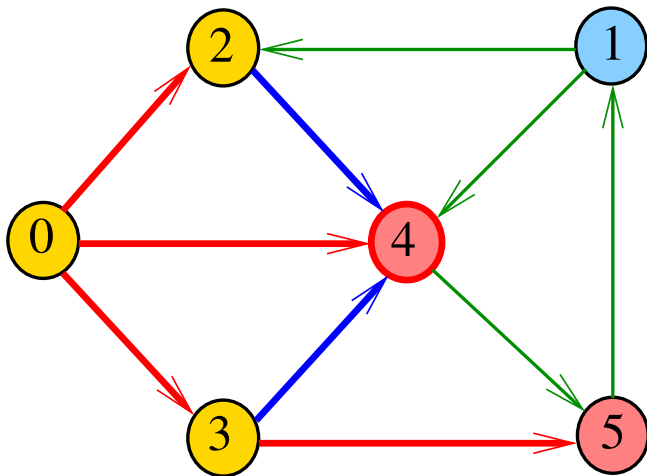
| i | 0 | 1 | 2 | 3 | 4 | 5 |
|--------|---|---|---|---|---|---|
| $q[i]$ | 0 | 2 | 3 | 4 | 5 | |

| v | 0 | 1 | 2 | 3 | 4 | 5 |
|------------------|---|---|---|---|---|---|
| $\text{dist}[v]$ | 0 | 6 | 1 | 1 | 1 | 2 |



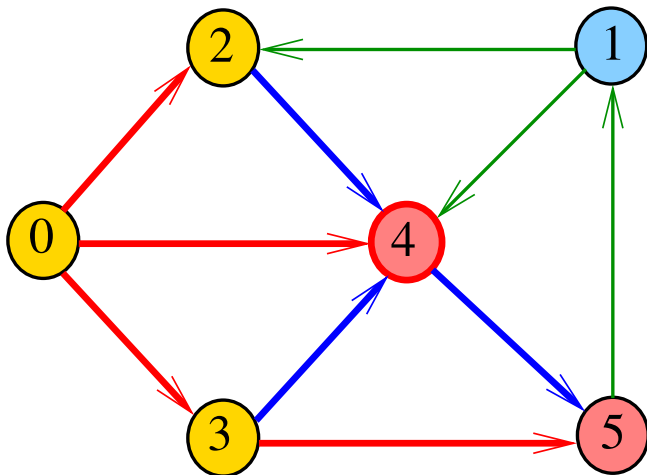
Simulação

| i | 0 | 1 | 2 | 3 | 4 | 5 | v | 0 | 1 | 2 | 3 | 4 | 5 |
|--------|---|---|---|---|---|---|------------------|---|---|---|---|---|---|
| $q[i]$ | 0 | 2 | 3 | 4 | 5 | | $\text{dist}[v]$ | 0 | 6 | 1 | 1 | 1 | 2 |



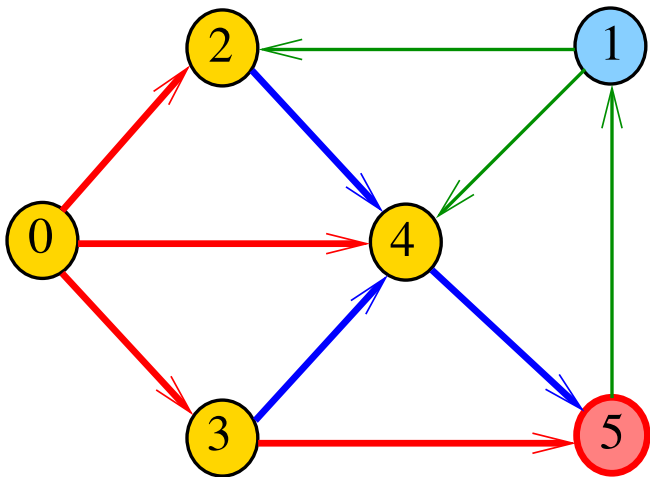
Simulação

| i | 0 | 1 | 2 | 3 | 4 | 5 | v | 0 | 1 | 2 | 3 | 4 | 5 |
|--------|---|---|---|---|---|---|------------------|---|---|---|---|---|---|
| $q[i]$ | 0 | 2 | 3 | 4 | 5 | | $\text{dist}[v]$ | 0 | 6 | 1 | 1 | 1 | 2 |



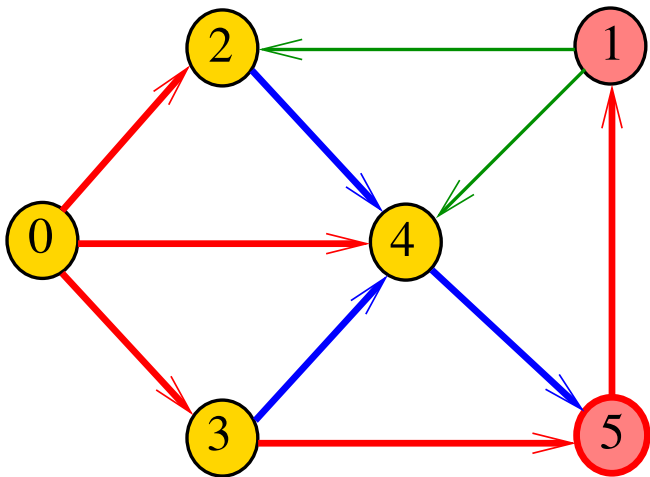
Simulação

| i | 0 | 1 | 2 | 3 | 4 | 5 | v | 0 | 1 | 2 | 3 | 4 | 5 |
|--------|---|---|---|---|---|---|------------------|---|---|---|---|---|---|
| $q[i]$ | 0 | 2 | 3 | 4 | 5 | | $\text{dist}[v]$ | 0 | 6 | 1 | 1 | 1 | 2 |



Simulação

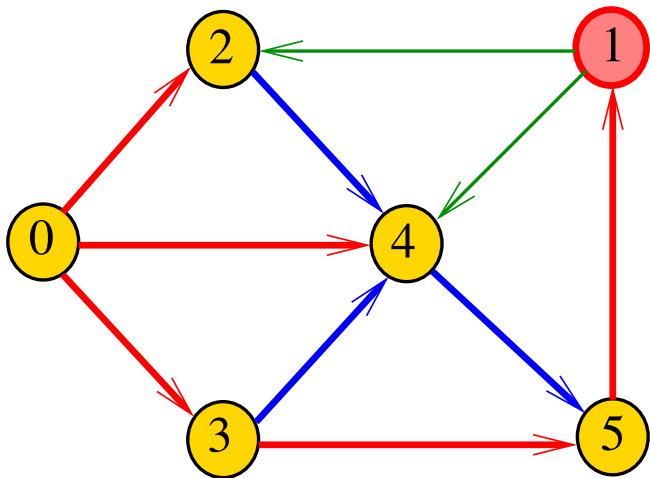
| i | 0 | 1 | 2 | 3 | 4 | 5 | v | 0 | 1 | 2 | 3 | 4 | 5 |
|--------|---|---|---|---|---|---|------------------|---|---|---|---|---|---|
| $q[i]$ | 0 | 2 | 3 | 4 | 5 | 1 | $\text{dist}[v]$ | 0 | 3 | 1 | 1 | 1 | 2 |



Simulação

| i | 0 | 1 | 2 | 3 | 4 | 5 |
|--------|---|---|---|---|---|---|
| $q[i]$ | 0 | 2 | 3 | 4 | 5 | 1 |

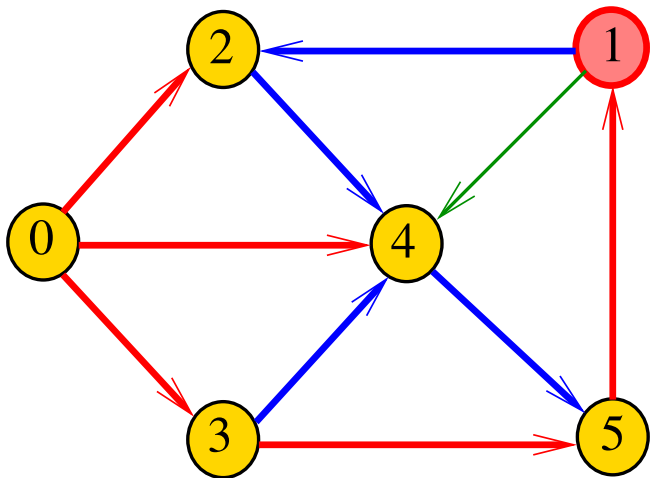
| v | 0 | 1 | 2 | 3 | 4 | 5 |
|------------------|---|---|---|---|---|---|
| $\text{dist}[v]$ | 0 | 3 | 1 | 1 | 1 | 2 |



Simulação

| i | 0 | 1 | 2 | 3 | 4 | 5 |
|--------|---|---|---|---|---|---|
| $q[i]$ | 0 | 2 | 3 | 4 | 5 | 1 |

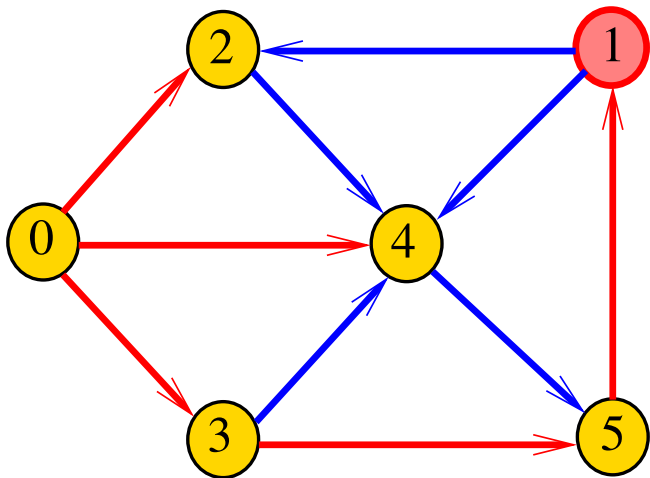
| v | 0 | 1 | 2 | 3 | 4 | 5 |
|------------------|---|---|---|---|---|---|
| $\text{dist}[v]$ | 0 | 3 | 1 | 1 | 1 | 2 |



Simulação

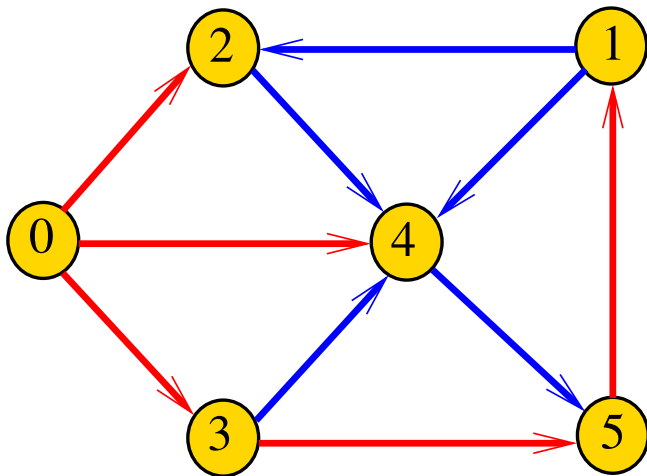
| i | 0 | 1 | 2 | 3 | 4 | 5 |
|--------|---|---|---|---|---|---|
| $q[i]$ | 0 | 2 | 3 | 4 | 5 | 1 |

| v | 0 | 1 | 2 | 3 | 4 | 5 |
|------------------|---|---|---|---|---|---|
| $\text{dist}[v]$ | 0 | 3 | 1 | 1 | 1 | 2 |



Simulação

| i | 0 | 1 | 2 | 3 | 4 | 5 | v | 0 | 1 | 2 | 3 | 4 | 5 |
|--------|---|---|---|---|---|---|------------------|---|---|---|---|---|---|
| $q[i]$ | 0 | 2 | 3 | 4 | 5 | 1 | $\text{dist}[v]$ | 0 | 3 | 1 | 1 | 1 | 2 |



DIGRAPHdist

```
#define INFINITO G->V /* ou maxV */
static int dist[maxV];
static Vertex parnt[maxV];
void DIGRAPHdist (Digraph G, Vertex s) {
1  Vertex v, w; link p;
2  for (v = 0; v < G->V; v++)
3      dist[v] = INFINITO;
4      parnt[v] = -1;
    }
5  QUEUEinit(G->V);
6  dist[s] = 0;
7  parnt[s] = s;
```


DIGRAPHdist

```
8  QUEUEput(s);
9  while (!QUEUEempty()) {
10     v = QUEUEget();
11     for(p=G->adj[v]; p!=NULL; p=p->next)
12         if (dist[w=p->w] == INFINITO){
13             dist[w] = dist[v] + 1;
14             parnt[w] = v;
15             QUEUEput(w);
16         }
17     }
18  QUEUEfree();
19 }
```

AULA 13

Custos nos arcos

S 20.1

Digrafos com custos nos arcos

Muitas aplicações associam um número a cada arco de um digrafo

Diremos que esse número é o custo da arco

Vamos supor que esses números são do tipo **double**

```
typedef struct {  
    Vertex v;  
    Vertex w;  
    double cst;  
} Arc;
```

ARC

A função **ARC** recebe dois vértices **v** e **w** e um valor **cst** e devolve um arco com ponta inicial **v** e ponta final **w** e custo **cst**

```
Arc ARC (Vertex v, Vertex w, double cst)  
{  
1    Arc e;  
2    e.v = v;    e.w = w;  
3    e.cst = cst;  
4    return e;  
}
```

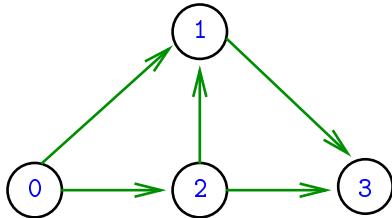
Matriz de adjacência

Matriz de adjacência indica a presença ausência e custo dos arcos:

se $v-w$ é um arco, $\text{adj}[v][w]$ é seu custo

se $v-w$ não é arco, $\text{adj}[v][w] = \text{maxCST}$

Exemplo:



| | 0 | 1 | 2 | 3 |
|---|---|----|---|-----|
| 0 | * | .3 | 2 | * |
| 1 | * | * | * | .42 |
| 2 | * | 1 | * | .12 |
| 3 | * | * | * | * |

* indica maxCST

Estrutura **digraph**

A estrutura **digraph** representa um digrafo

adj é um ponteiro para a matriz de adjacência

V contém o número de vértices

A contém o número de arcos do digrafo.

```
struct digraph {  
    int V;  
    int A;  
    double **adj;  
};
```

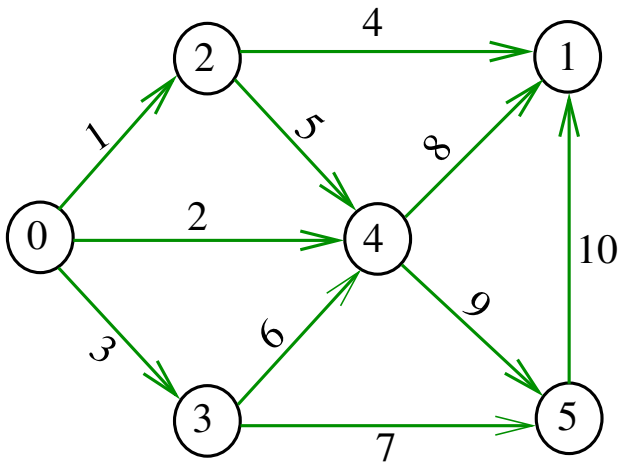
Estrutura Digraph

Um objeto do tipo **Digraph** contém o endereço de um **digraph**

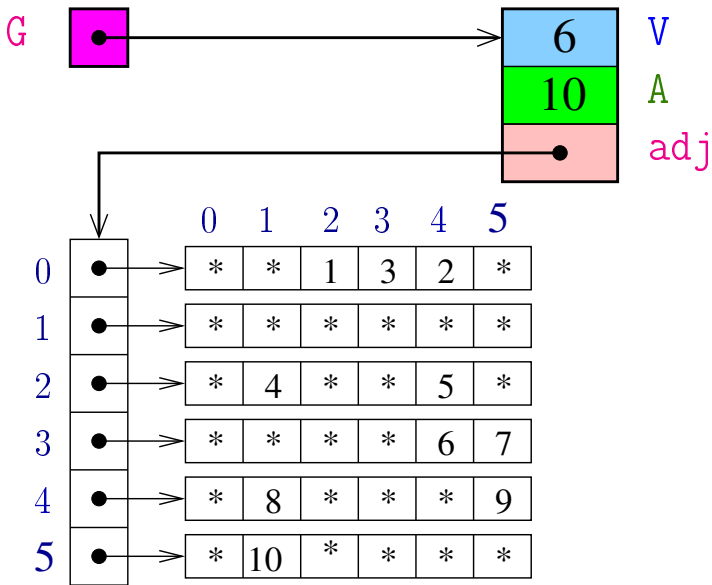
```
typedef struct digraph *Digraph;
```


Digrafo

Digraph G



Estruturas de dados



MATRIXdouble

Aloca uma matriz com linhas $0 \dots r-1$ e colunas $0 \dots c-1$, cada elemento da matriz recebe valor **val**

```
double **  
MATRIXdouble(int r, int c, double val) {  
0     Vertex i, j;  
1     double **m = malloc(r*sizeof(double*));  
2     for (i = 0; i < r; i++)  
3         m[i] = malloc(c*sizeof(double));  
4     for (i = 0; i < r; i++)  
5         for (j = 0; j < c; j++)  
6             m[i][j] = val;  
7     return m;  
}
```

DIGRAPHinit

Devolve (o endereço de) um novo digrafo com vértices $0, \dots, V-1$ e nenhum arco.

```
Digraph DIGRAPHinit (int V) {  
0     Digraph G = malloc(sizeof *G);  
1     G->V = V;  
2     G->A = 0;  
3     G->adj = MATRIXdouble(V, V, maxCST);  
4     return G;  
}
```

DIGRAPHinsertA

Inserir um arco $v-w$ de custo cst no digrafo G

Se $v=w$ ou o digrafo já tem arco $v-w$, não faz nada

void

DIGRAPHinsertA(Digraph G , Vertex v , Vertex w ,
double cst)

```
{  
    if ( $v \neq w$  &&  $G \rightarrow adj[v][w] == \text{maxCST}$ ) {  
         $G \rightarrow adj[v][w] = cst$ ;  
         $G \rightarrow A++$ ;  
    }  
}
```

Vetor de listas de adjacência

A lista de adjacência de um vértice **v** é composta por nós do tipo **node**

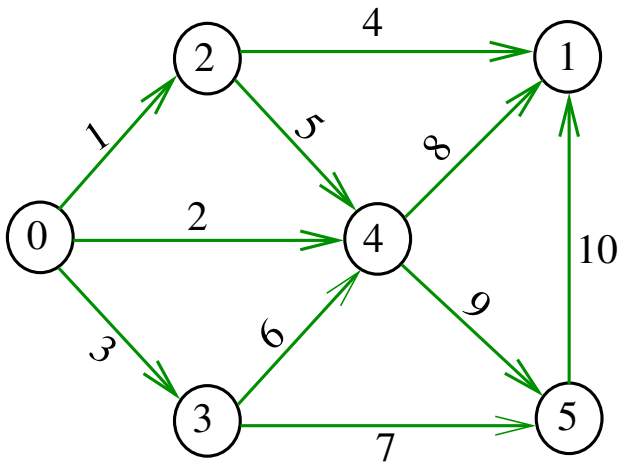
Um **link** é um ponteiro para um **node**

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

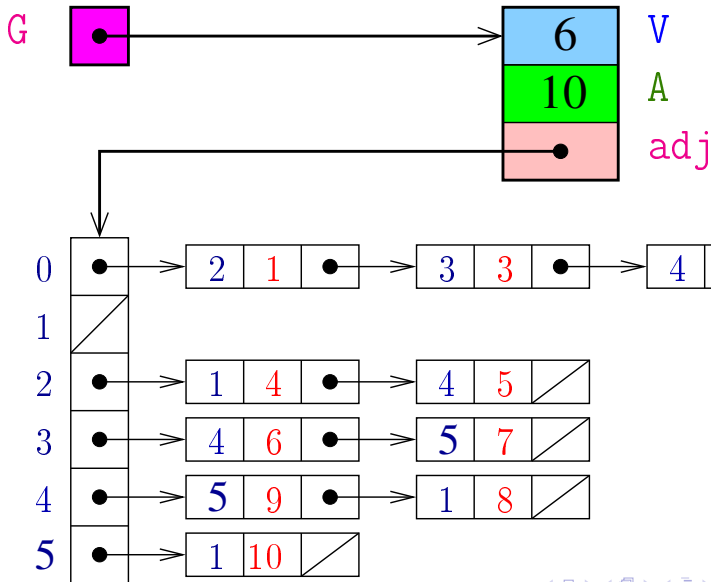
```
typedef struct node *link;
struct node {
    Vertex w;
    double cst;
    link next;
};
```

Digrafo

Digraph G



Estruturas de dados



Estrutura digraph

A estrutura **digraph** representa um digrafo

V contém o número de vértices

A contém o número de arcos do digrafo

adj é um ponteiro para vetor de listas de
adjacência

```
struct digraph {  
    int V;  
    int A;  
    link *adj;  
};
```

Estrutura Digraph

Um objeto do tipo **Digraph** contém o endereço de um **digraph**

```
typedef struct digraph *Digraph;
```

NEW

NEW recebe um vértice **w**, um custo **cst** e o endereço **next** de um nó e devolve o endereço **x** de um novo nó com $x \rightarrow w = w$, e $x \rightarrow cst = cst$ e $x \rightarrow next = next$

```
link NEW (Vertex w, double cst, link next)
{
    link x = malloc(sizeof *x);
    x->w = w;
    x->cst = cst;
    x->next = next;
    return x;
}
```

DIGRAPHInit

Devolve (o endereço de) um novo digrafo com vértices $0, \dots, V-1$ e nenhum arco

```
Digraph DIGRAPHInit (int V) {  
0     Vertex v;  
1     Digraph G = malloc(sizeof *G);  
2     G->V = V;  
3     G->A = 0;  
4     G->adj = malloc(V * sizeof(link));  
5     for (v = 0; v < V; v++)  
6         G->adj[v] = NULL;  
7     return G;  
}
```

DIGRAPHinsertA

Inserir um arco $v-w$ de custo cst no digrafo G .

Se $v == w$ ou o digrafo já tem arco $v-w$; não faz nada

void

```
DIGRAPHinsertA(Digraph G, Vertex v, Vertex w,  
               double cst)
```

```
{  
    link p;  
    if (v == w) return;  
    for (p = G->adj[v]; p != NULL; p = p->next)  
        if (p->w == w) return;  
    G->adj[v] = NEW(w, cst, G->adj[v]);  
    G->A++;  
}
```

DIGRAPHinsertA

O código abaixo transfere a responsabilidade de evitar laços e arcos paralelos ao cliente/usuário

void

DIGRAPHinsertA (**Digraph** **G**, **Vertex** **v**, **Vertex** **w**,
 double **cst**)

{

G->**adj**[**v**] = **NEW**(**w**, **cst**, **G**->**adj**[**v**]);

G->**A**++;

}

Caminhos de custo mínimo

S 21.0 e 21.1

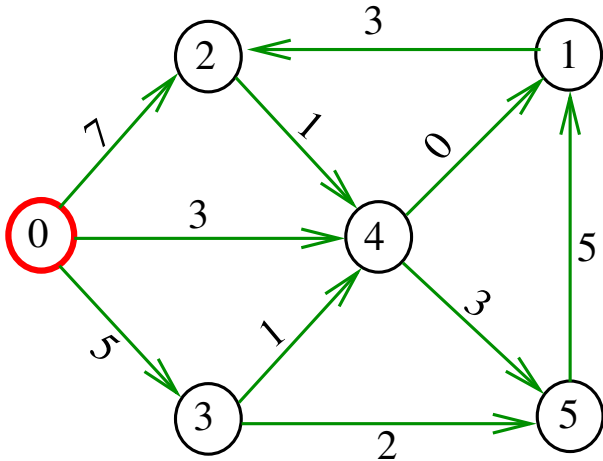
Problema

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 .

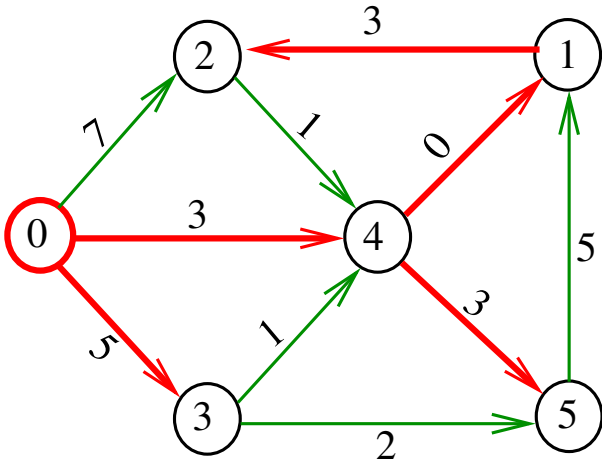
Exemplo

Entra:



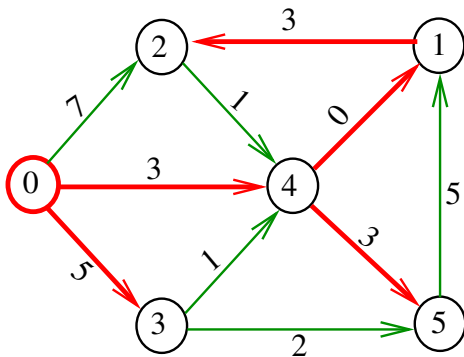
Exemplo

Sai:



Arborescência de caminhos mínimos

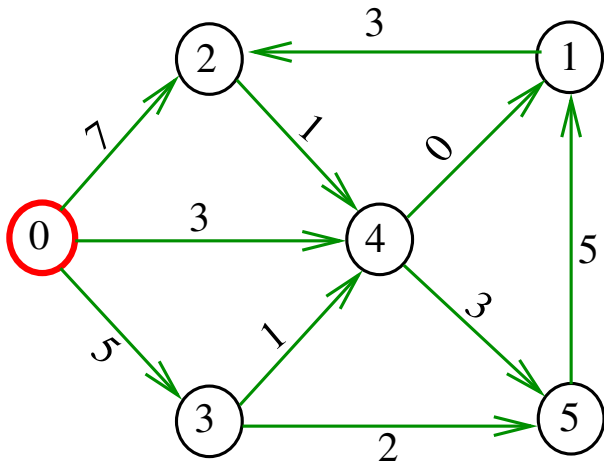
Uma arborescência com raiz **s** é de **caminhos mínimos** (= *shortest-paths tree* = *SPT*) se para todo vértice **t** que pode ser alcançado a partir de **s**,
o único caminho de s a t na arborescência é um caminho mínimo



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:

