

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

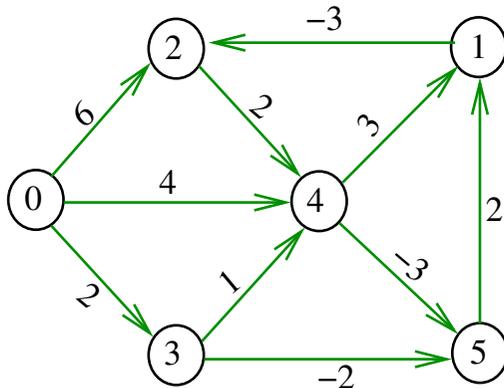
Vamos supor que esses números são do tipo **double**

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

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



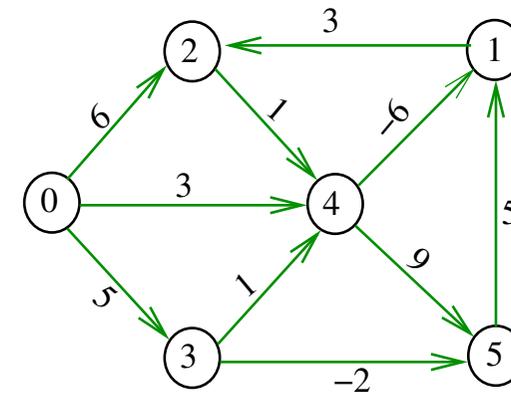
Custo de um caminho

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.



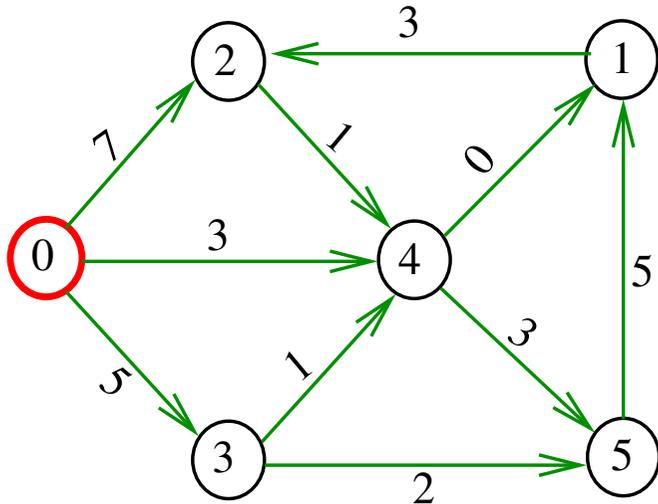
Problema

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:



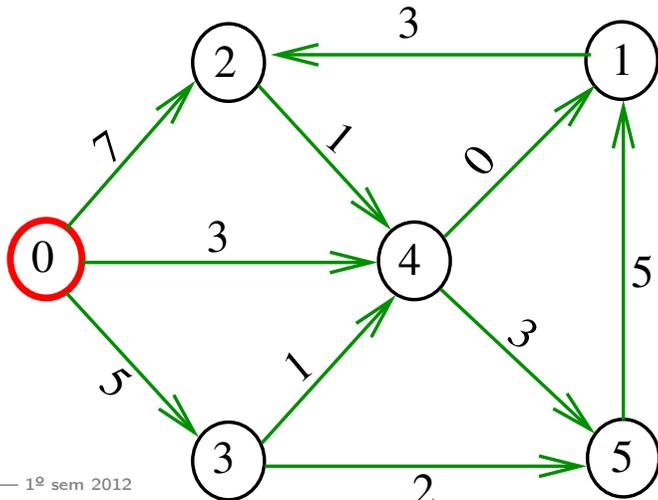
Sai:



Problema da SPT

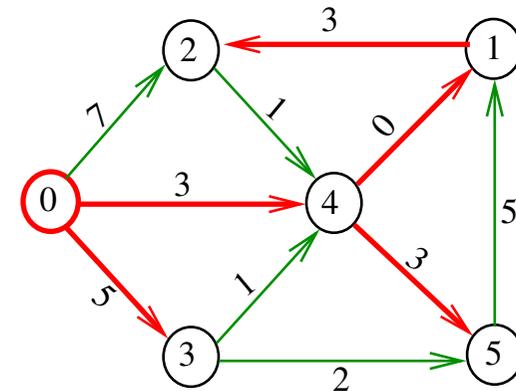
Problema: Dado um vértice s de um digrafo com custos **não-negativos** nos arcos, encontrar uma SPT com raiz s

Entra:



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



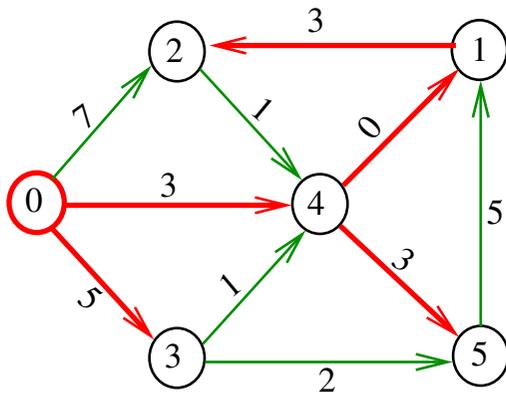
Algoritmo de Dijkstra

S 21.1 e 21.2

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



dijkstra

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

Calcula uma arborescência de caminhos mínimos com raiz s .

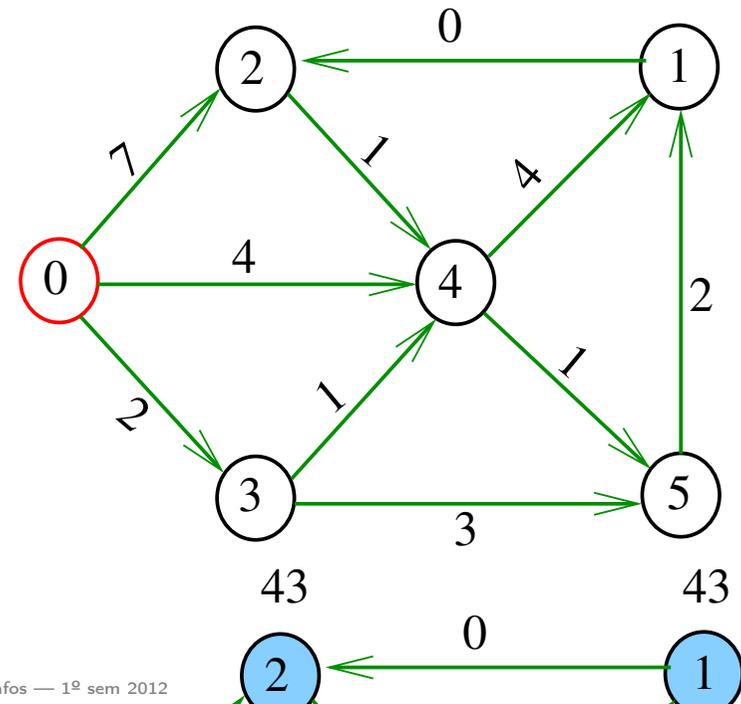
A arborescência é armazenada no vetor `parnt`

As distâncias em relação a s são armazenadas no vetor `cst`

void

```
dijkstra(Digraph G, Vertex s,  
         Vertex parnt[], double cst[]);
```

Simulação



Fila com prioridades

A função `dijkstra` usa uma fila com prioridades
A fila é manipulada pelas seguintes funções:

- `PQinit()`: inicializa uma fila de vértices em que cada vértice v tem prioridade $cst[v]$
- `PQempty()`: devolve 1 se a fila estiver vazia e 0 em caso contrário
- `PQinsert(v)`: insere o vértice v na fila
- `PQdelmin()`: retira da fila um vértice de prioridade mínima.
- `PQdec(w)`: reorganiza a fila depois que o valor de $cst[w]$ foi decrementado.

dijkstra

```
#define INFINITO maxCST
void
dijkstra(Digraph G, Vertex s,
         Vertex parnt[], double cst[]);
{
1  Vertex v, w; link p;
2  for (v = 0; v < G->V; v++) {
3      cst[v] = INFINITO;
4      parnt[v] = -1;
5  }
5  PQinit(G->V);
6  cst[s] = 0;
7  parnt[s] = s;
```

Algoritmos em Grafos — 1º sem 2012

13 / 1

Algoritmos em Grafos — 1º sem 2012

14 / 1

dijkstra

```
8  PQinsert(s);
9  while (!PQempty()) {
10     v = PQdelmin();
11     for (p=G->adj[v]; p!=NULL; p=p->next)
12         if (cst[w=p->w]==INFINITO) {
13             cst[w]=cst[v]+p->cst;
14             parnt[w]=v;
15             PQinsert(w);
16         }
```

dijkstra

```
16     else
17         if (cst[w]>cst[v]+p->cst)
18             cst[w]=cst[v]+p->cst
19             parnt[w] = v;
20             PQdec(w);
21     }
21  PQfree();
}
```

Algoritmos em Grafos — 1º sem 2012

15 / 1

Algoritmos em Grafos — 1º sem 2012

16 / 1

Consumo de tempo

linha	número de execuções da linha
2-4	$\Theta(V)$
5	= 1 PQinit
6-7	= 1
8	= 1 PQinsert
9-10	$O(V)$ PQempty e PQdelmin
11	$O(A)$
12-14	$O(V)$
15	$O(V)$ PQinsert
16-19	$O(A)$
20	$O(A)$ PQdec
21	= 1 PQfree
total	= $O(V + A) + ???$

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

- 1 execução de `PQinit` e `PQfree`,
- $O(V)$ execuções de `PQinsert`,
- $O(V)$ execuções de `PQempty`,
- $O(V)$ execuções de `PQdelmin`, e
- $O(A)$ execuções de `PQdec`.

```
/* Item.h */
typedef Vertex Item;

/* QUEUE.h */
void PQinit(int);
int PQempty();
void PQinsert(Item);
Item PQdelmin();
void PQdec(Item);
void PQfree();
```

PQinit e PQempty

```
Item *q;
int inicio, fim;

void PQinit(int maxN) {
    q=(Item*)malloc(maxN*sizeof(Item));
    inicio = 0;
    fim = 0;
}

int PQempty() {
    return inicio==fim;
}
```

PQinsert e PQdelmin

```
void PQinsert(Item item){
    q[fim++] = item;
}

Item PQdelmin() {
    int i, j; Item x;
    i= inicio;
    for (j=i+1; j < fim; j++)
        if (cst[q[i]] > cst[q[j]]) i = j;
    x = q[i];
    q[i] = q[--fim];
    return x;
}
```

```

void PQdec(Vertex v) {
  /* faz nada */
}

void PQfree() {
  free(q);
}

```

PQinit	$\Theta(1)$
PQempty	$\Theta(1)$
PQinsert	$\Theta(1)$
PQdelmin	$O(V)$
PQdec	$\Theta(1)$
PQfree	$\Theta(1)$

Conclusão

O consumo de tempo da função `dijkstra` é $O(V^2)$.

Este consumo de tempo é ótimo para **digrafos densos**.

Consumo de tempo Min-Heap

	heap	d -heap	fibonacci heap
PQinsert	$O(\lg V)$	$O(\log_D V)$	$O(1)$
PQdelmin	$O(\lg V)$	$O(\log_D V)$	$O(\lg V)$
PQdec	$O(\lg V)$	$O(\log_D V)$	$O(1)$
dijkstra	$O(A \lg V)$	$O(A \log_D V)$	$O(A + V \lg V)$