

AULAS anteriores

Busca em largura

S 18.7

Usamos **busca em profundidade** para encontrar:

- caminho de s a t ou **st-corte** (S, T) em que todo arco no **corte** tem ponta inicial em T e ponta final em S ;
- **ciclo** em digrafos ou **ordenação topológica**;
- **ciclo** em grafos;
- **componentes** de grafos;
- **bipartição** de grafos ou **ciclo ímpar**;
- **pontes** de grafos;
- **articulações** de grafos; e
- **componentes fortemente conexas** de digrafos.

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

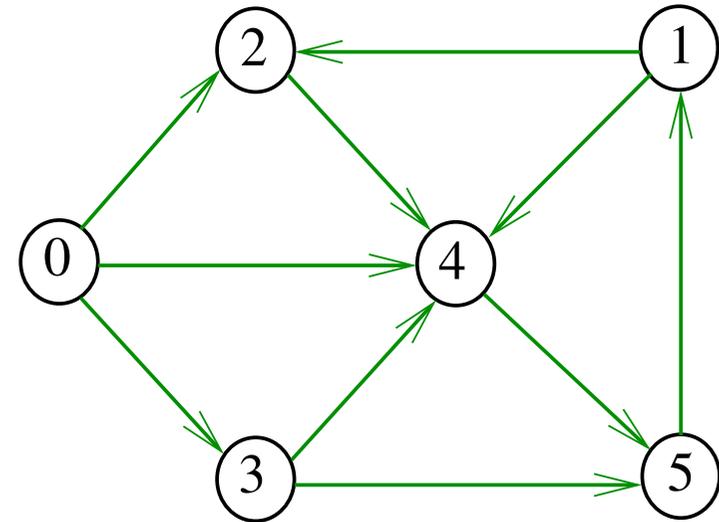
A **busca em largura** (= *breadth-first search* = *BFS*) começa por um vértice, digamos **s**, especificado pelo usuário.

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]						



DIGRAPHbfs

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

A ordem em que os vértices são visitados é registrada no vetor **lbl**. Se **v** é o **k**-ésimo vértices visitado então **lbl[v] == k-1**

A função usa uma **fila** de vértices

```
#define maxV 1024
static int cnt, lbl[maxV];
void DIGRAPHbfs (Digraph G, Vertex s)
```

Implementação de uma fila

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

/* QUEUE.h */
void QUEUEinit(int);
int QUEUEempty();
void QUEUEput(Item);
Item QUEUEget();
void QUEUEfree();
```

DIGRAPHbfs

```
void DIGRAPHbfs (Digraph G, Vertex s)
{
1  Vertex v, w;
2  cnt = 0;
3  for (v = 0; v < G->V; v++)
4      lbl[v] = -1;
5  QUEUEinit(G->V);
```

Relações invariantes

Digamos que um vértice v foi **visitado** se
 $lbl[v] \neq -1$

No início de cada iteração das linhas 8–13 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 V vértices

DIGRAPHbfs

```
6  lbl[s] = cnt++;
7  QUEUEput(s);
8  while (!QUEUEempty()) {
9      v = QUEUEget();
10     for (w=0; w < G->V; w++)
11         if (G->adj[v][w] == 1
12             && lbl[w] == -1) {
13             lbl[w] = cnt++;
14             QUEUEput(w);
15         }
16     }
17     QUEUEfree();
18 }
```

QUEUEinit e QUEUEempty

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

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

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

```

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

Item QUEUEget() {
    return q[inicio++];
}

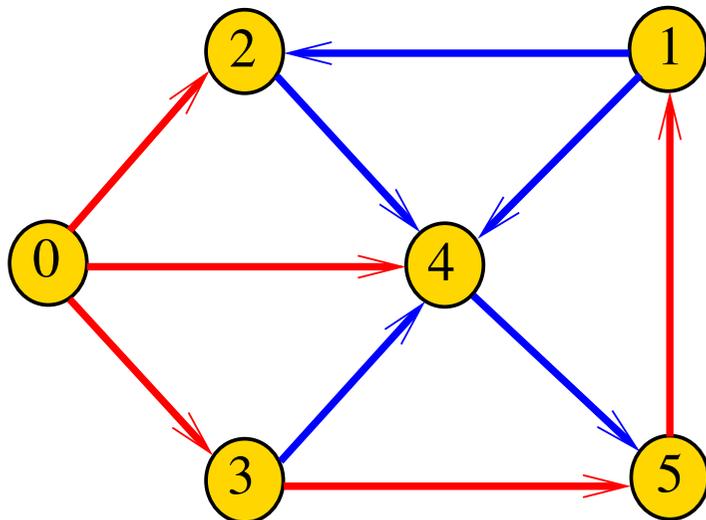
void QUEUEfree() {
    free(q);
}
    
```

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

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

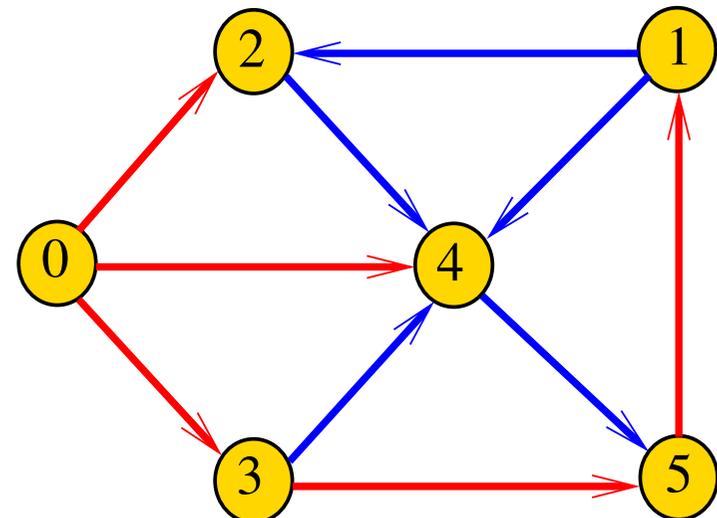
Arborescência da BFS

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



Representação da BFS

Podemos representar essa arborescência explicitamente por um vetor de pais **parnt**



DIGRAPHbfs

```
#define maxV 1024
static int cnt, lbl[maxV];
static Vertex parnt[maxV];
void DIGRAPHbfs (Digraph G, Vertex s)
{
1  Vertex v, w;
2  cnt = 0;
3  for (v = 0; v < G->V; v++)
4      lbl[v] = -1;
5  QUEUEinit(G->V);
6  lbl[s] = cnt++;
7  parnt[s] = s;
```

Algoritmos em Grafos — 1º sem 2014

17 / 1

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**, na busca em profundidade, o próprio **algoritmo escolhe o vértice inicial**
- a busca em largura visita apenas os **vértices que podem ser atingidos** a partir do vértice inicial, a busca em profundidade visita, tipicamente, **todos os vértices** do digrafo

Algoritmos em Grafos — 1º sem 2014

19 / 1

DIGRAPHbfs

```
8  QUEUEput(s);
9  while (!QUEUEempty()) {
10     v = QUEUEget();
11     for (w=0; w < G->V; w++)
12         if (G->adj[v][w] == 1
13             && lbl[w] == -1) {
14                 lbl[w] = cnt++;
15                 parnt[w] = v;
16                 QUEUEput(w);
17             }
18     }
19  QUEUEfree();
20 }
```

Algoritmos em Grafos — 1º sem 2014

18 / 1

Caminhos mínimos

S 18.7

Algoritmos em Grafos — 1º sem 2014

19 / 1

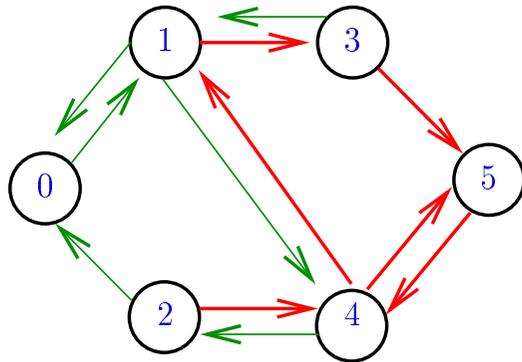
Algoritmos em Grafos — 1º sem 2014

20 / 1

Comprimento

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

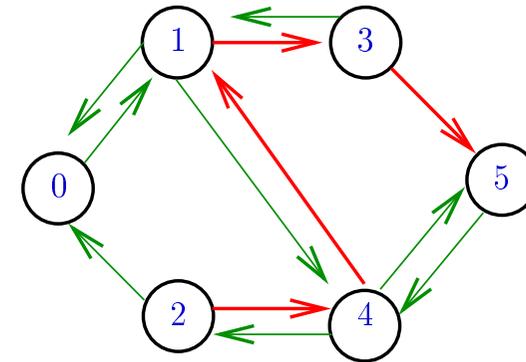
Exemplo: 2-4-1-3-5-4-5 tem comprimento 6



Comprimento

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

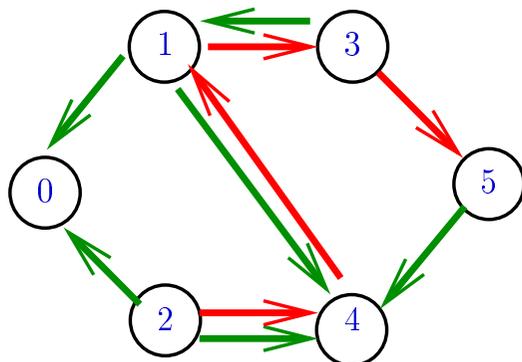
Exemplo: 2-4-1-3-5 tem comprimento 4



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



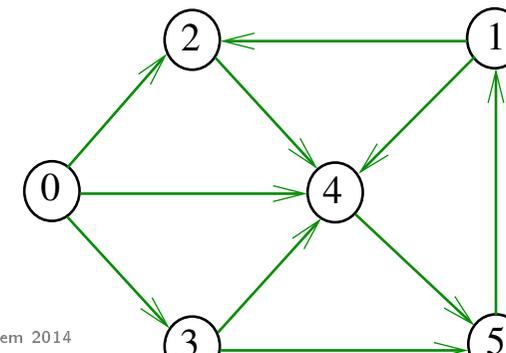
Exemplo: a distância de 0 a 2 é **infinita**

Calculando distâncias

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

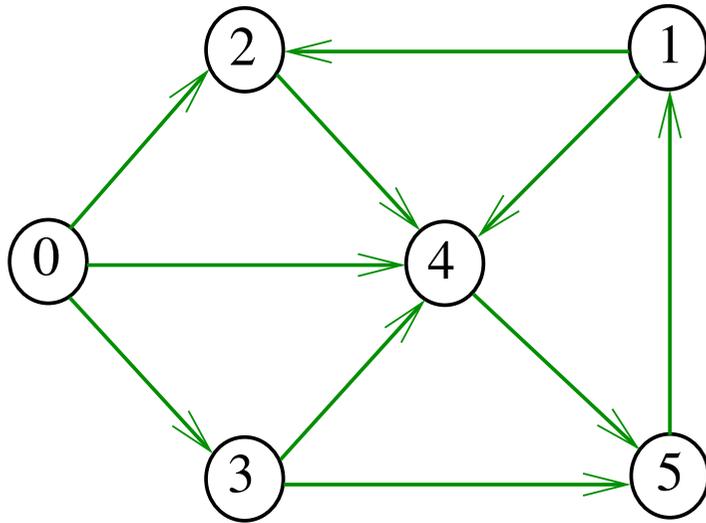
Exemplo: para $s = 0$

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



Simulação

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



DIGRAPHdist

```

void DIGRAPHbfs (Digraph G, Vertex s)
{
  1 Vertex v, w;
  2 for (v = 0; v < G->V; v++) {
  3     dist[v] = INFINITO;
  3     parnt[v] = -1;
  }
  4 QUEUEinit(G->V);
  5 dist[s] = 0;
  6 parnt[s] = s;

```

DIGRAPHdist

DIGRAPHdist armazena no vetor **dist** a distância do vértice **s** a cada um dos vértices do grafo **G**

A distância 'infinita' é representada por **G->V**

```

#define maxV 1024
#define INFINITO G->V
static int dist[maxV];
static Vertex parnt [maxV];
void DIGRAPHdist (Digraph G, Vertex s);

```

DIGRAPHdist

```

  7 QUEUEput(s);
  8 while (!QUEUEempty()) {
  9     v = QUEUEget();
 10     for (w=0; w < G->V; w++)
 11         if (G->adj[v][w] == 1
 12             && dist[w] == INFINITO) {
 13             dist[w] = dist[v] + 1;
 14             parnt[w] = v;
 15             QUEUEput(w);
 16         }
 17     }
 18 QUEUEfree();

```

Relações invariantes

No início de cada iteração das linhas 8–13 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 $\text{dist}[x] \neq \infty$ então $\text{dist}[x]$ é a distância de s a x .

Consumo de tempo

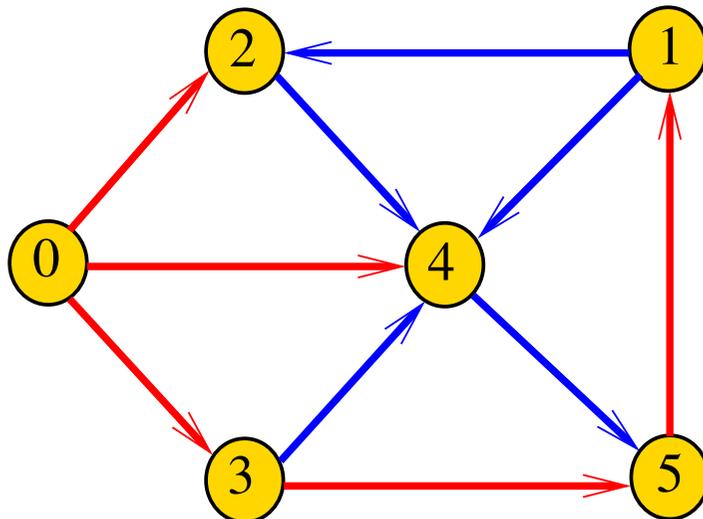
O consumo de tempo da função `DIGRAPHdist` para **vetor de listas de adjacência** é $O(V + A)$.

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

Arborescência da BFS

v	0	1	2	3	4	5
parnt	0	5	0	0	0	3

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



Condição de inexistência

Se no fim do algoritmo $\text{dist}[t] == \text{INFINITO}$ para algum vértice t , então

$$S = \{v : \text{dist}[v] < \text{INFINITO}\}$$

$$T = \{v : \text{dist}[v] == \text{INFINITO}\}$$

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