Gli algoritmi di visita dei grafi

Gianpiero Cabodi e Paolo Camurati

Dip. Automatica e Informatica
Politecnico di Torino

Algoritmi di visita

Visita di un grafo G=(V, E):

- a partire da un vertice dato, seguendo gli archi con una certa strategia, elencare i vertici incontrati, eventualmente aggiungendo altre informazioni.

Algoritmi:

- in profondità (depth-first search, DFS)
- in ampiezza (breadth-first search, BFS).

Visita in profondità

Dato un grafo (connesso o non connesso), a partire da un vertice s:

- visita **tutti** i vertici del grafo (raggiungibili da s e non)
- etichetta ogni vertice v con tempo di scoperta/ tempo di fine elaborazione `pre[v]/post[v]`
- etichetta ogni arco:
  - grafi orientati: T(tree), B(backward), F(forward), C(cross)
  - grafi non orientati: T(tree), B(backward)
- genera una foresta di alberi della visita in profondità, memorizzata in un vettore `st`.

Principi base

Profondità: espande l'ultimo vertice scoperto che ha ancora vertici non ancora scoperti adiacenti.

Scoperta di un vertice: prima volta che si incontra nella visita (discesa ricorsiva, visita in pre-order).

Completamento: fine dell'elaborazione del vertice (uscita dalla ricorsione, visita in post-order).

Scoperta/Completamento: tempo discreto che avanza mediante contatore `time`.

I vertici si distinguono (concettualmente) in:

- bianchi: non ancora scoperti
- grigi: scoperti, ma non completati
- neri: scoperti e completati.

Per ogni vertice si memorizza:

- il tempo di scoperta `pre[i]` e il tempo di fine elaborazione `post[i]`
- il padre nella visita in profondità `st[i]`.

Esempio

`time = -1`   `st`

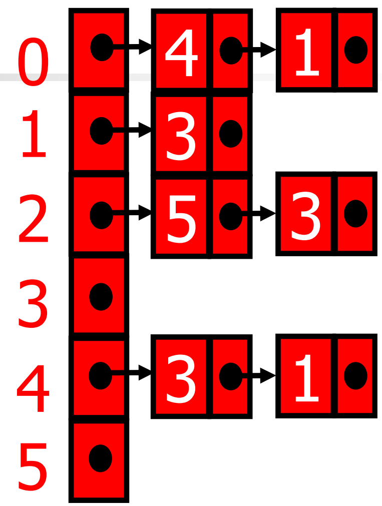
in.txt

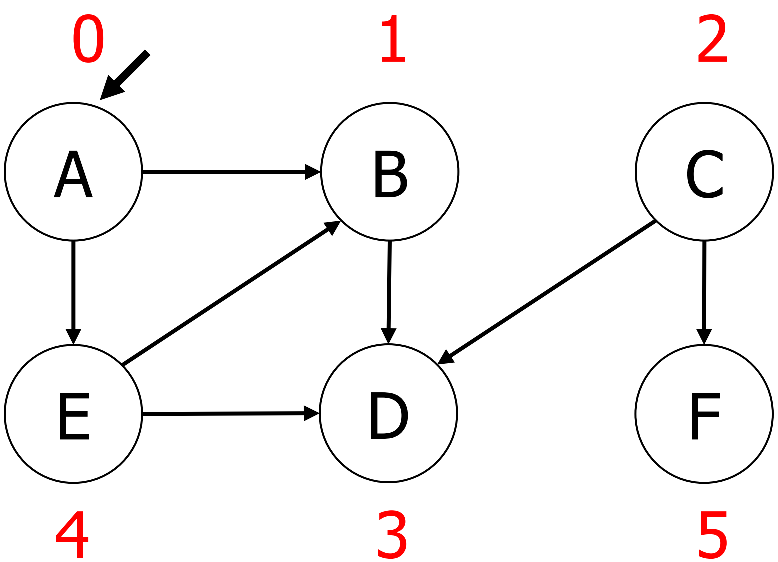
```
A B
C D
A E
C F
B D
E B
E D
```

ST

0	A
1	B
2	C
3	D
4	E
5	F

Lista delle adiacenze





st

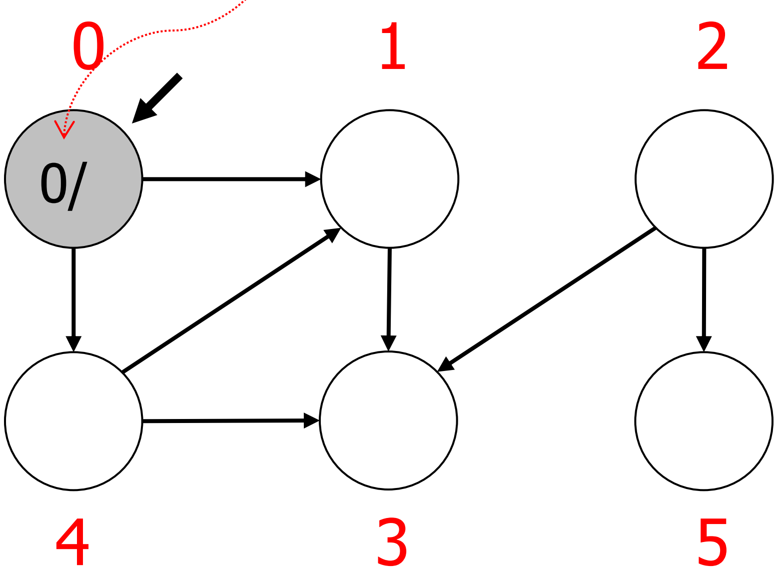
0	1	2	3	4	5
-1	-1	-1	-1	-1	-1

time = 0

st

0

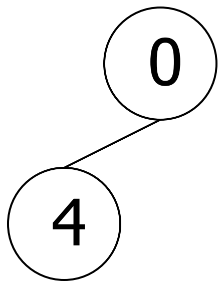
pre[i]/post[i]

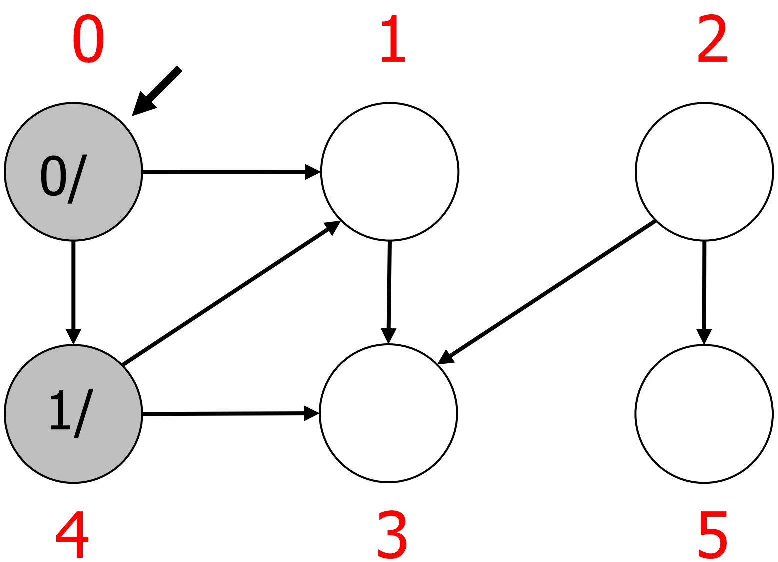


st	0	-1	-1	-1	-1	-1
	0	1	2	3	4	5

time = 1

st

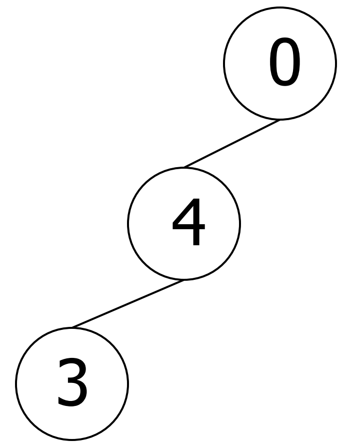


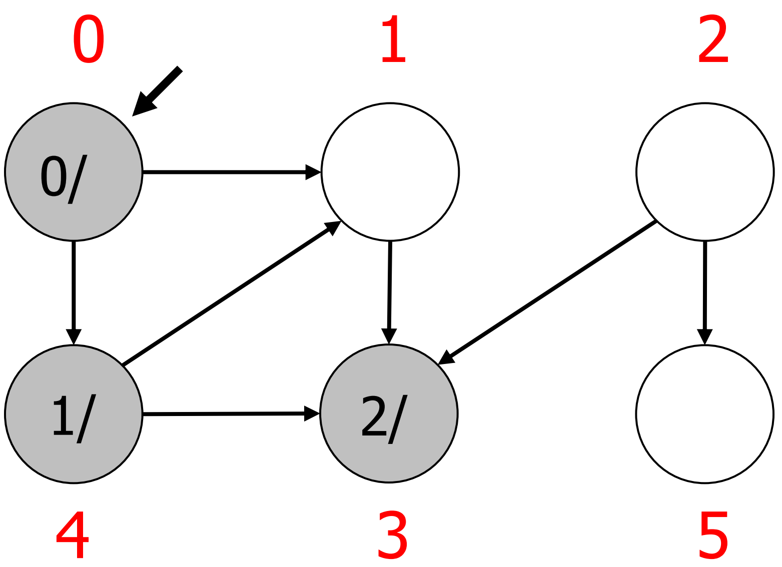


	0	1	2	3	4	5
st	0	-1	-1	-1	0	-1

time = 2

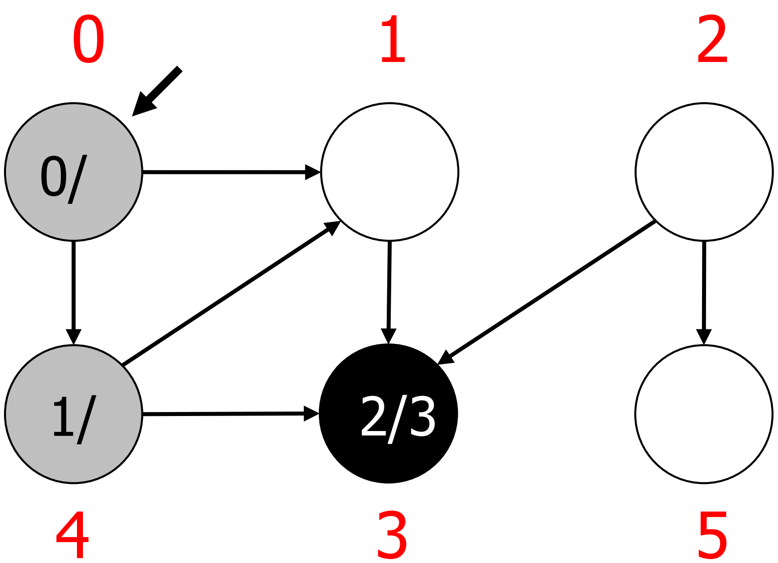
st



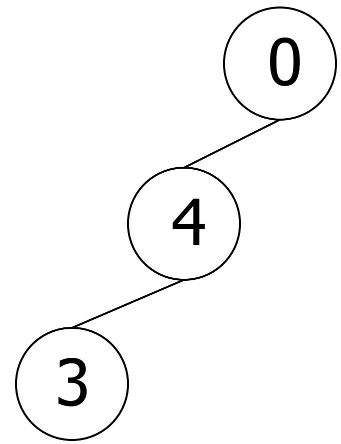


st	0	-1	-1	4	0	-1
	0	1	2	3	4	5

`time = 3`



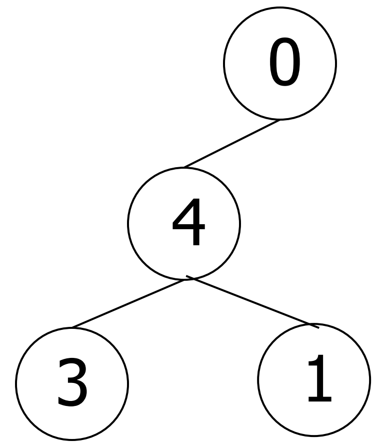
`st`

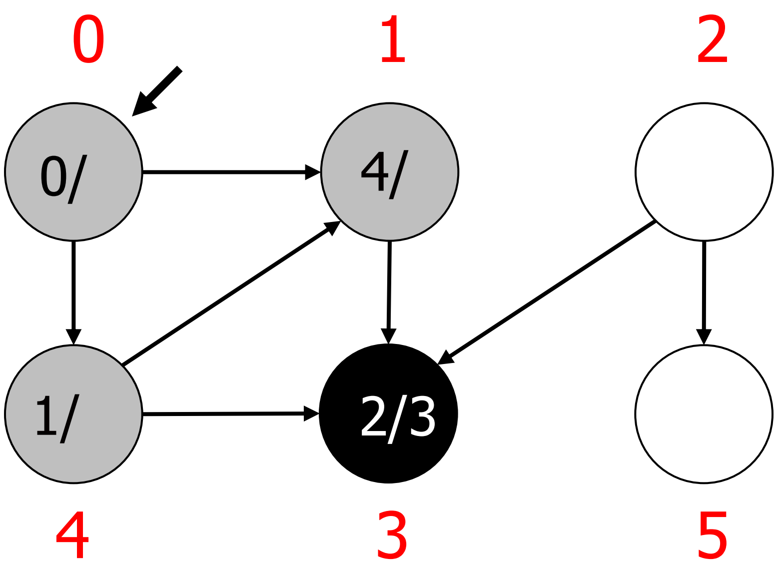


`st`	0	-1	-1	4	0	-1
	0	1	2	3	4	5

time = 4

st

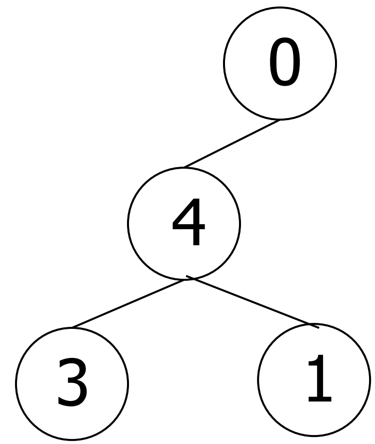


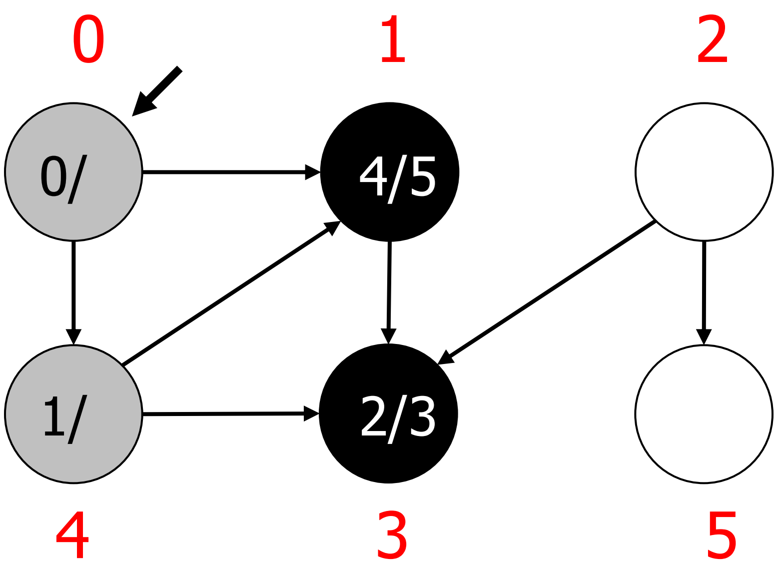


st	0	4	-1	4	0	-1
	0	1	2	3	4	5

time = 5

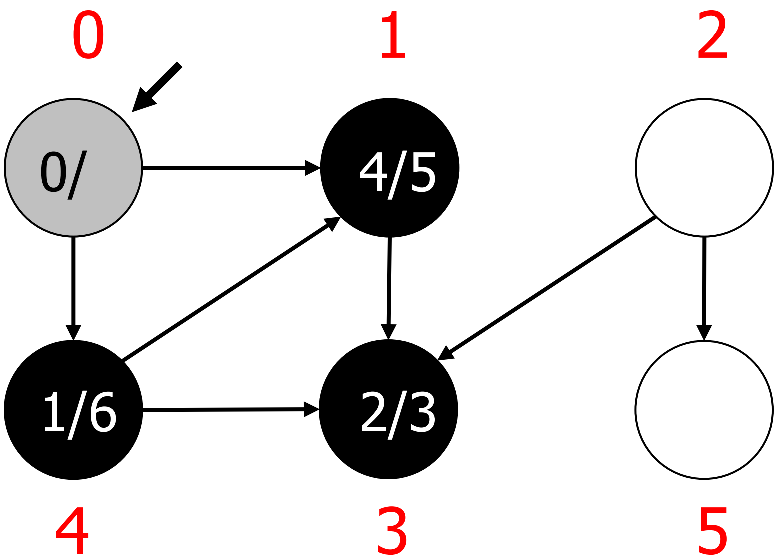
st

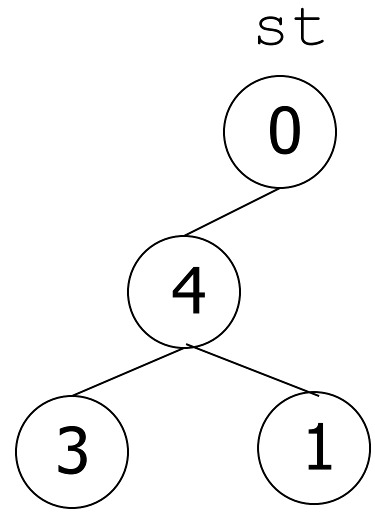




	0	1	2	3	4	5
st	0	4	-1	4	0	-1

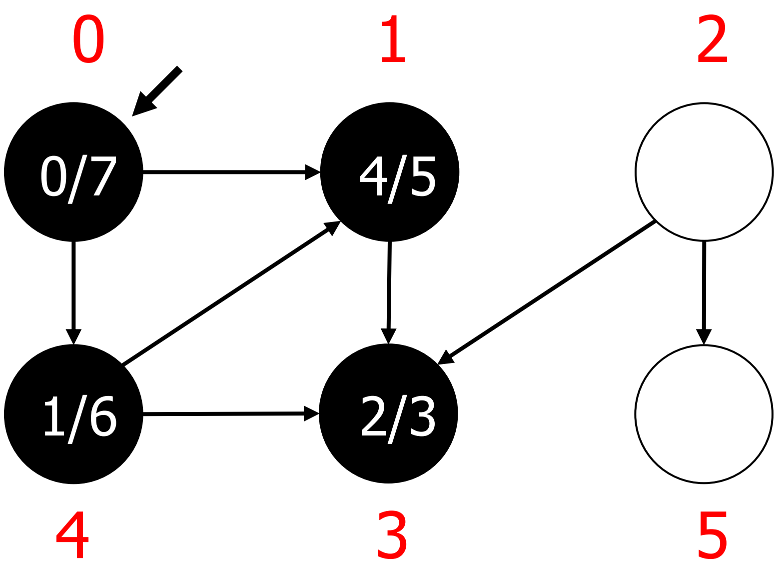
time = 6



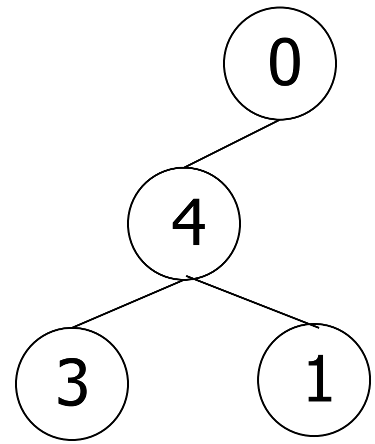


	0	1	2	3	4	5
st	0	4	-1	4	0	-1

time = 7



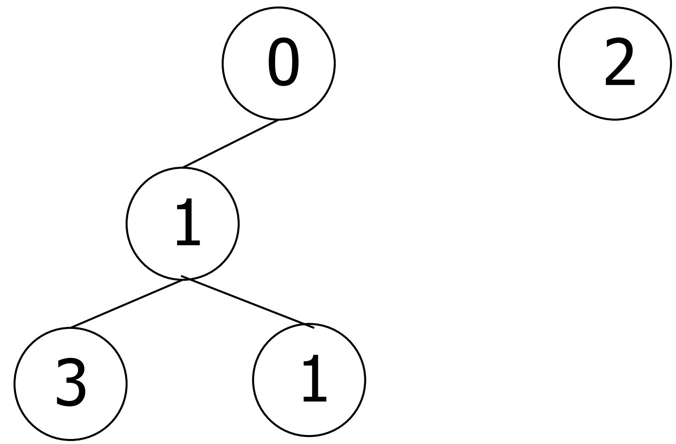
st

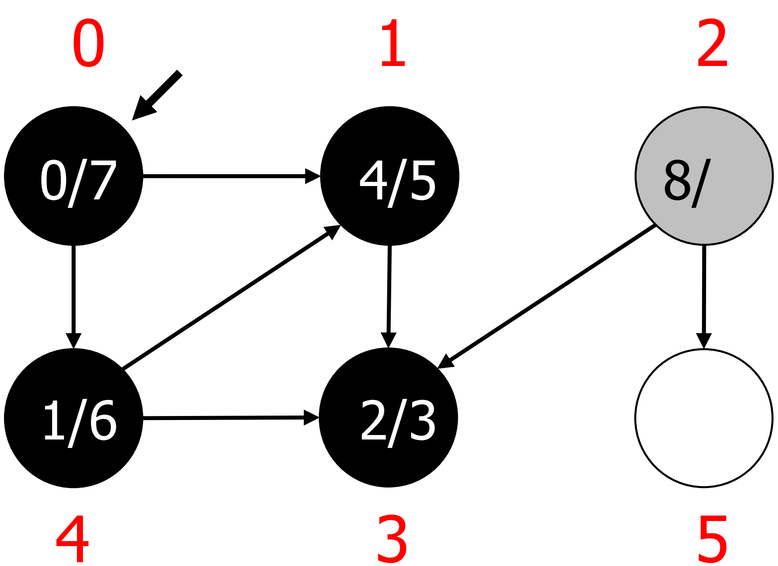


	0	1	2	3	4	5
st	0	4	-1	4	0	-1

`time` = 8

`st`

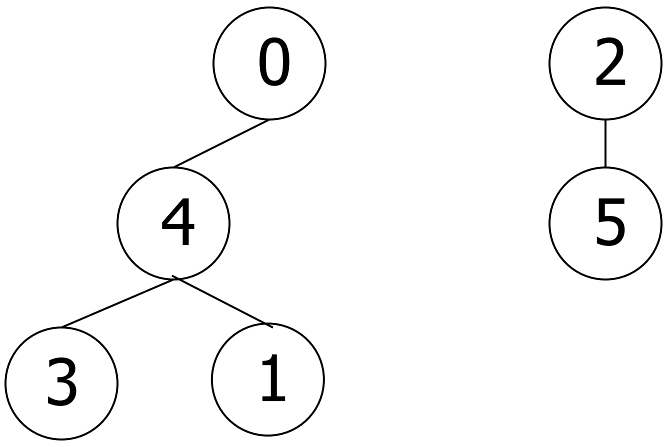


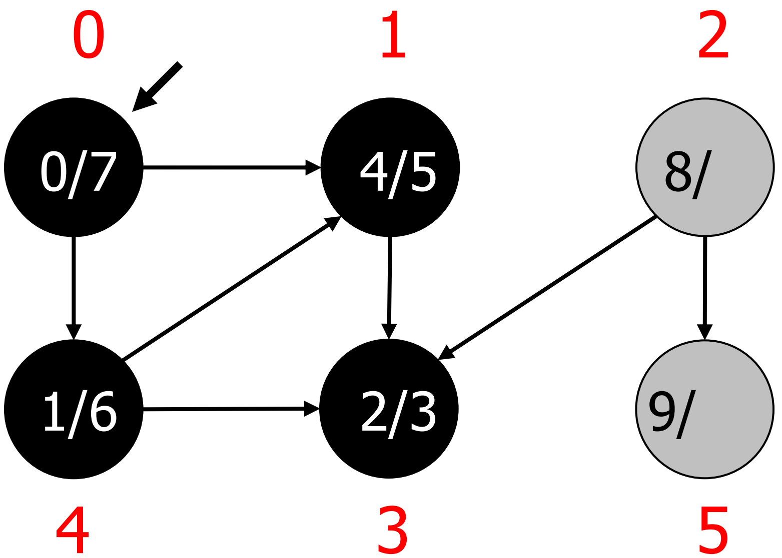


`st`	0	4	2	4	0	-1
	0	1	2	3	4	5

time = 9

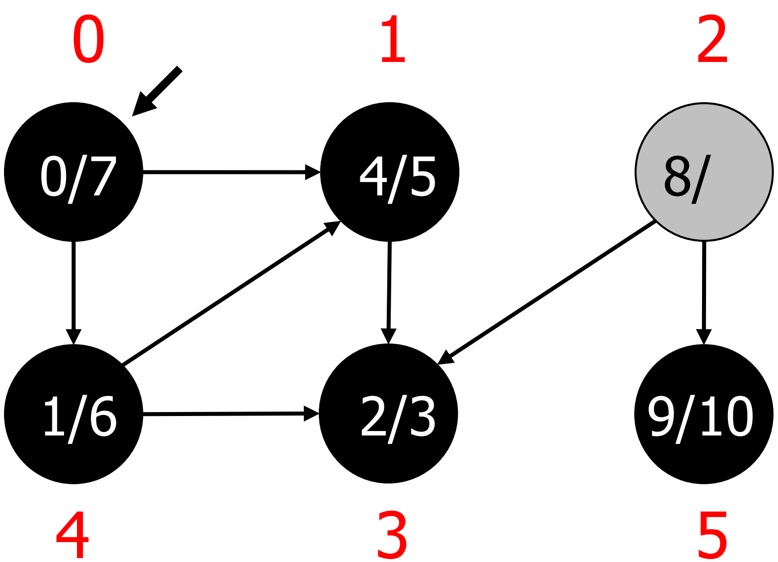
st



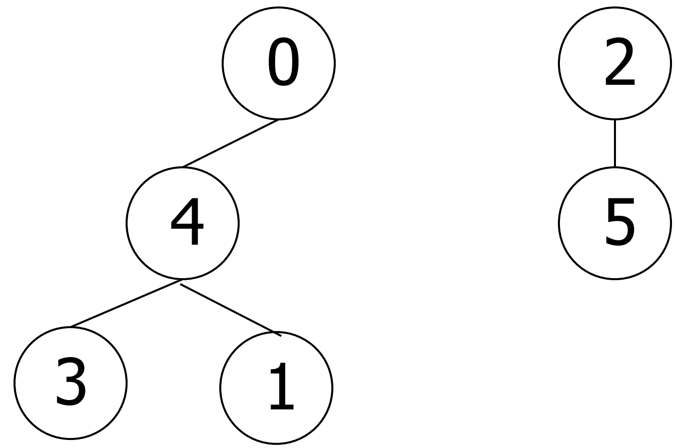


	0	1	2	3	4	5
st	0	4	2	4	0	2

time = 10



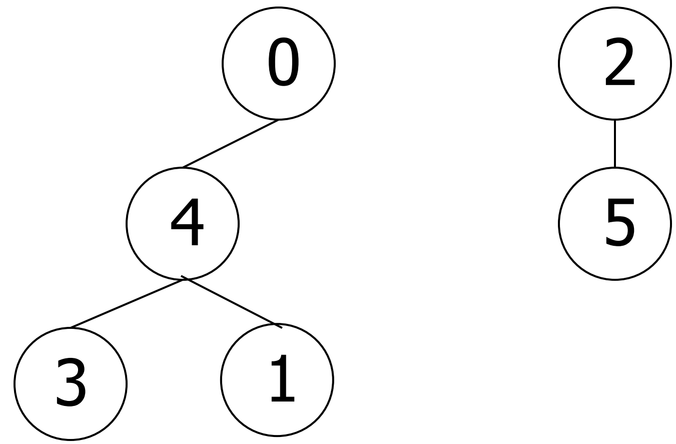
st

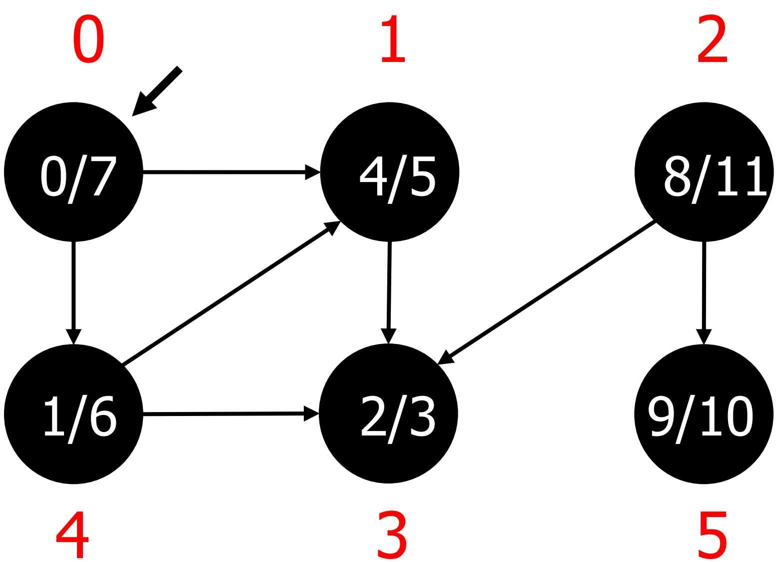


	0	1	2	3	4	5
st	0	4	2	4	0	2

time = 11

st





	0	1	2	3	4	5
st	0	4	2	4	0	2

Classificazione degli archi

Grafo orientato:

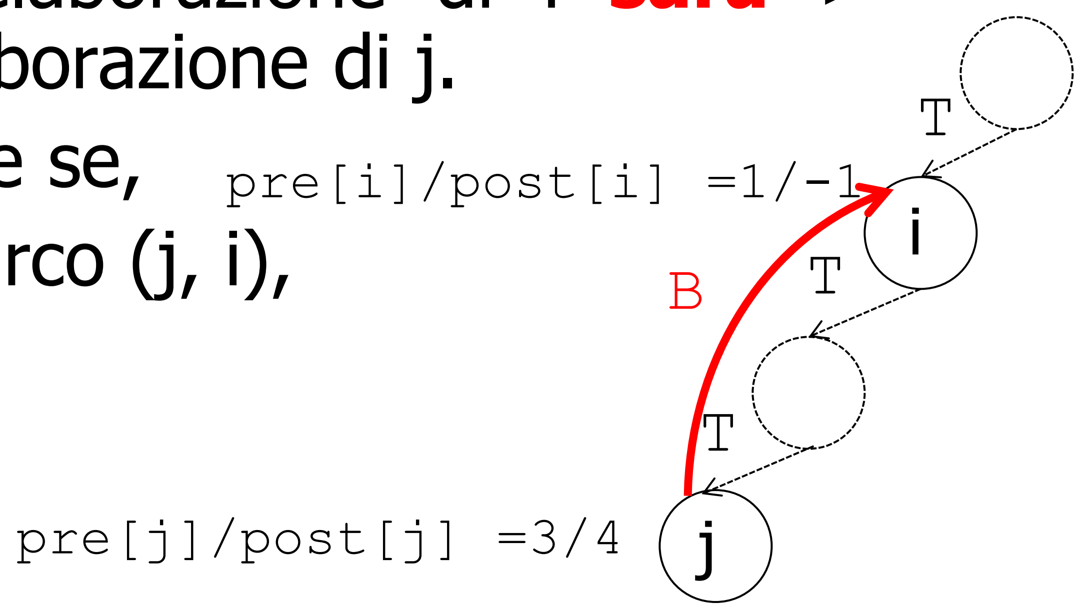
- T: archi dell'albero della visita in profondità
- B: connettono un vertice j ad un suo antenato i nell'albero:

  tempo di fine elaborazione di i **sarà** > tempo di fine elaborazione di j.

  Equivale a testare se,

  quando scopro l'arco (j, i),

  `post[i] == -1`

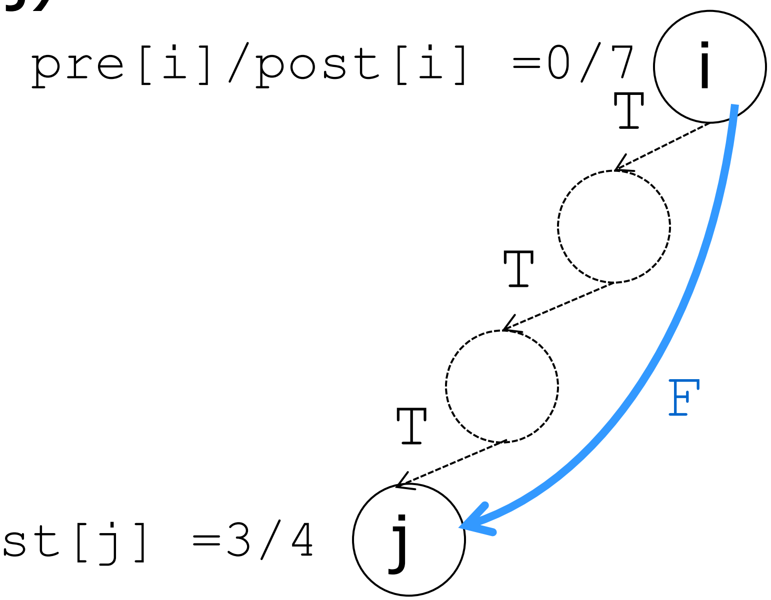


- F: connettono un vertice i ad un suo discendente j nell'albero:

  tempo di scoperta di i **è** < tempo di scoperta di j

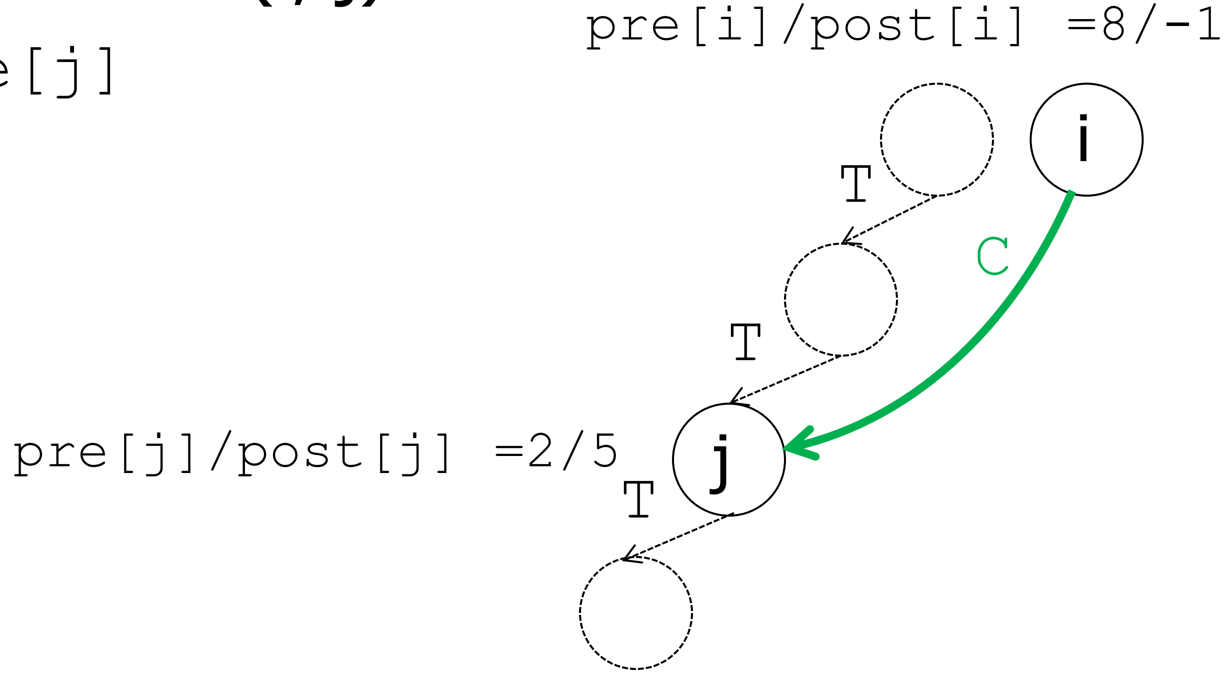
  quando scopro l'arco (i, j)

  `pre[i] < pre[j]`



- C: archi rimanenti, per cui tempo di scoperta di i **è** > tempo di scoperta di j quando scopro l'arco (i, j)

pre[i] > pre[j]



Grafo non orientato: solo archi T e B.

Esempio

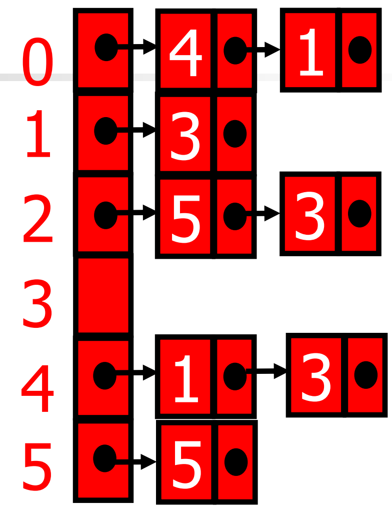
`time = -1`   `st`

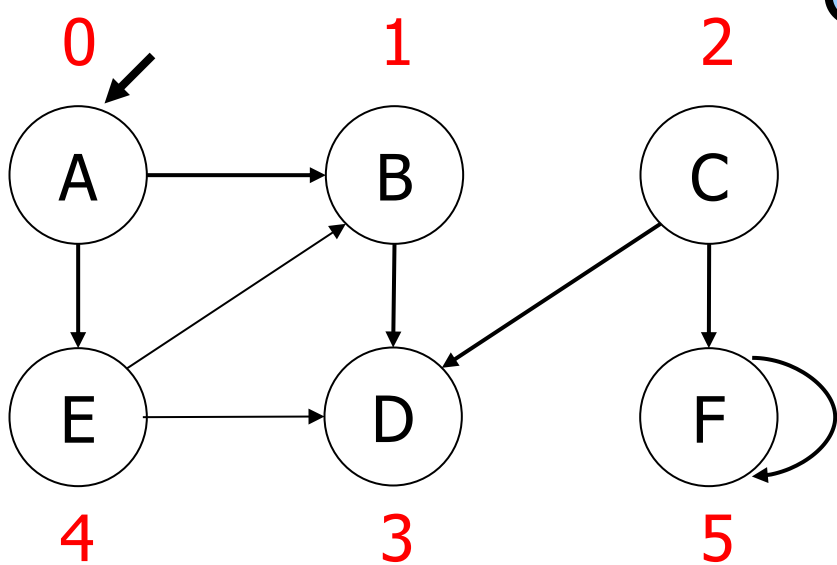
in.txt

```
A B
C D
A E
C F
B D
E D
E B
F F
```

	ST
0	A
1	B
2	C
3	D
4	E
5	F

Lista delle adiacenze





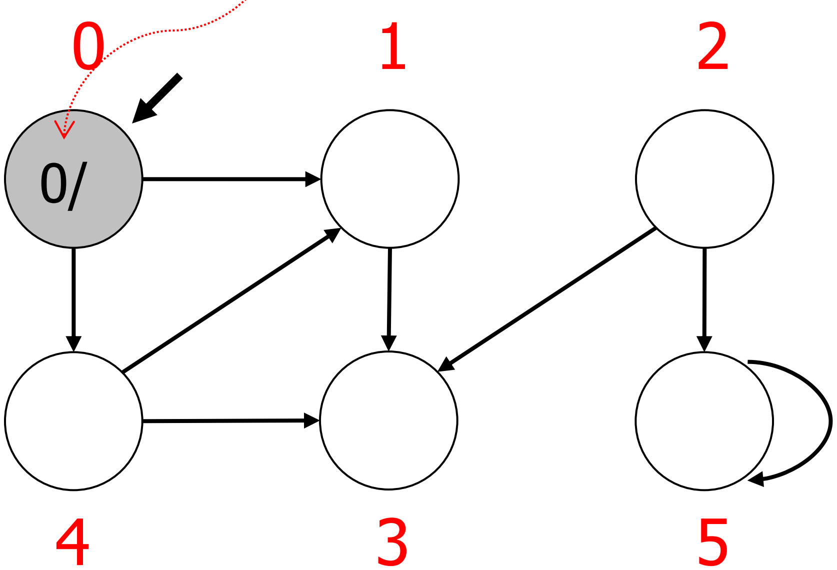
st	-1	-1	-1	-1	-1	-1
	0	1	2	3	4	5

`time = 0`

`st`

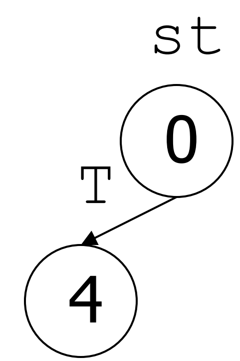
0

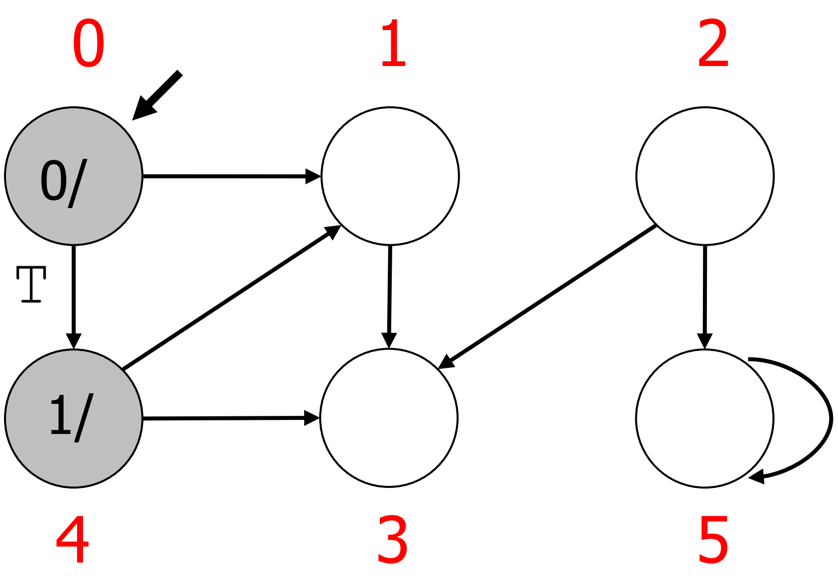
`pre[i]/post[i]`



st	0	-1	-1	-1	-1	-1
	0	1	2	3	4	5

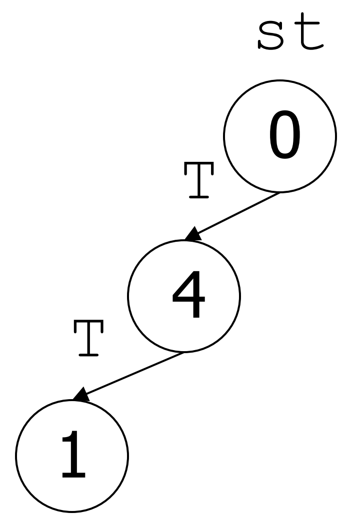
time = 1

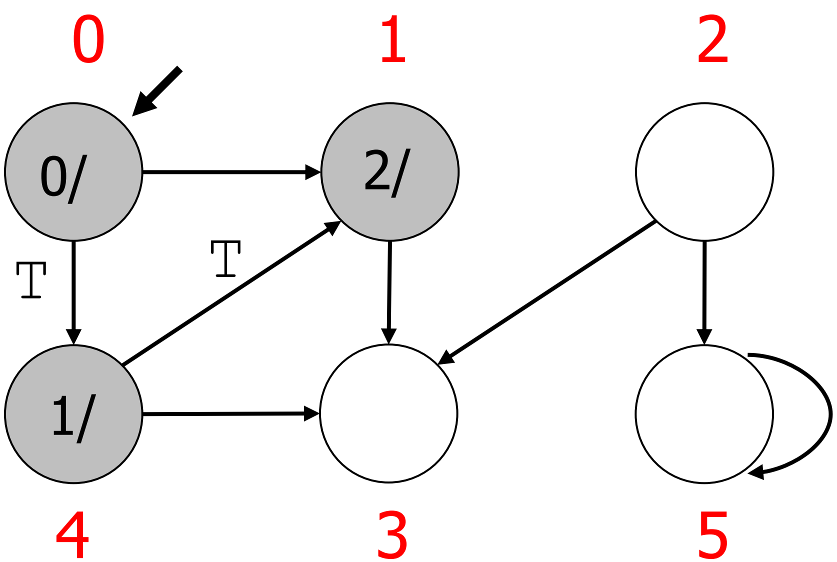




st					
0	-1	-1	-1	0	-1
0	1	2	3	4	5

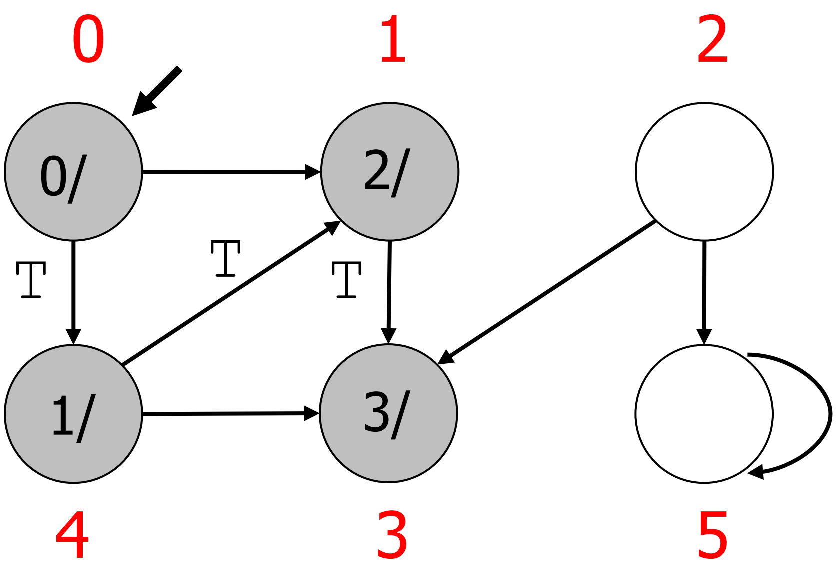
time = 2

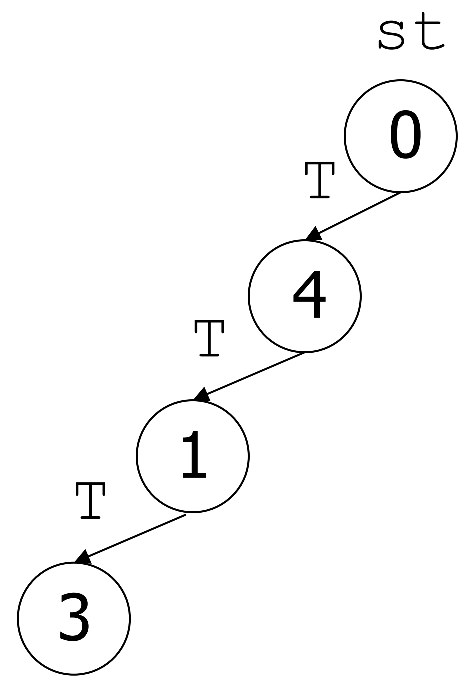




	0	1	2	3	4	5
st	0	4	-1	-1	0	-1

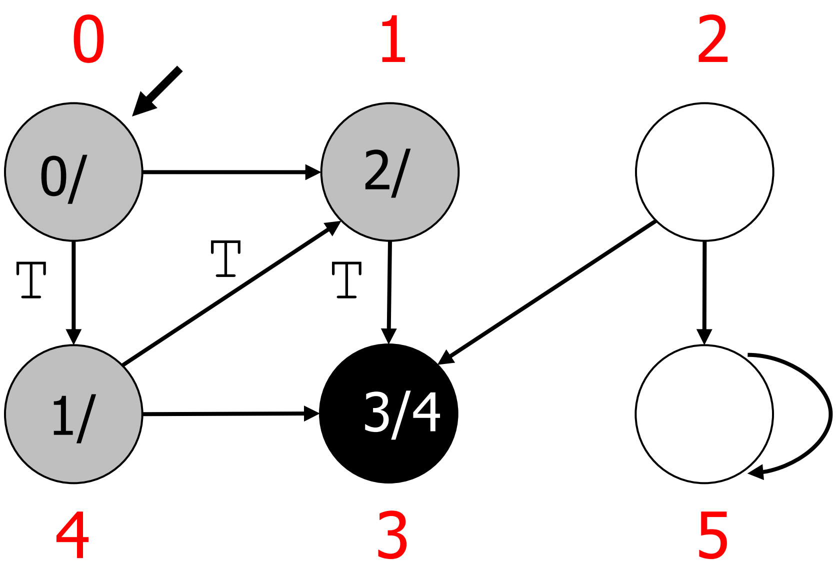
time = 3

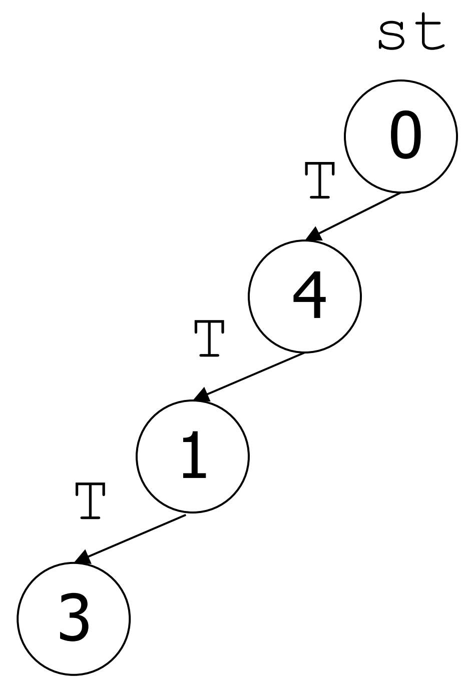




st	0	4	-1	1	0	-1
	0	1	2	3	4	5

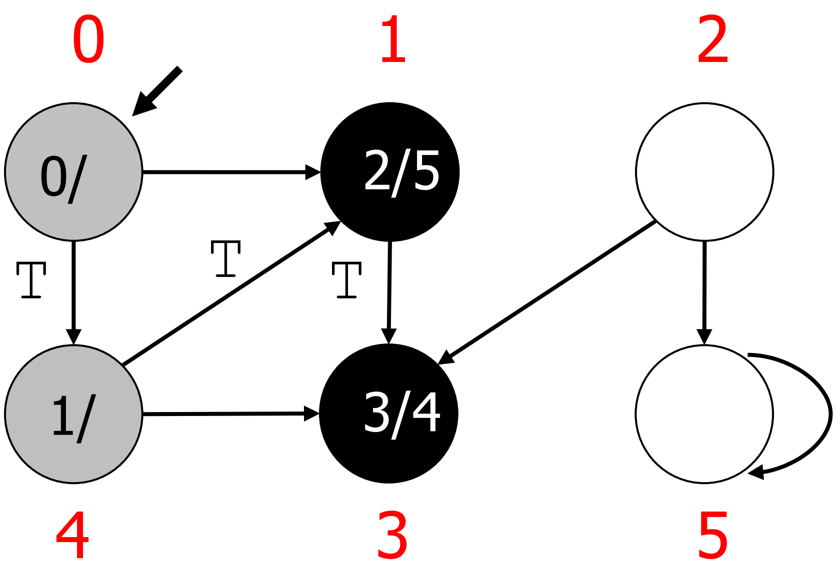
time = 4



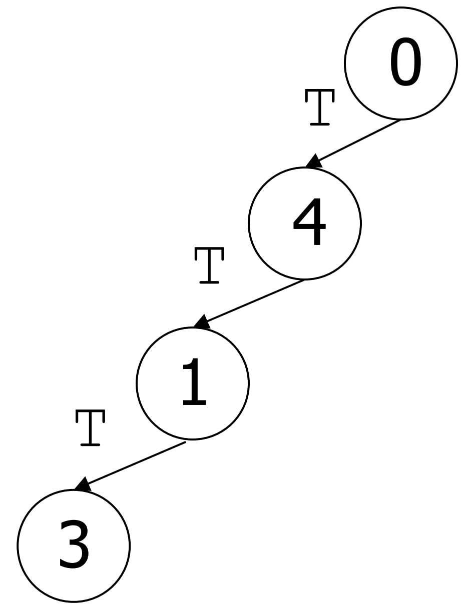


	0	1	2	3	4	5
st	0	4	-1	1	0	-1

time = 5

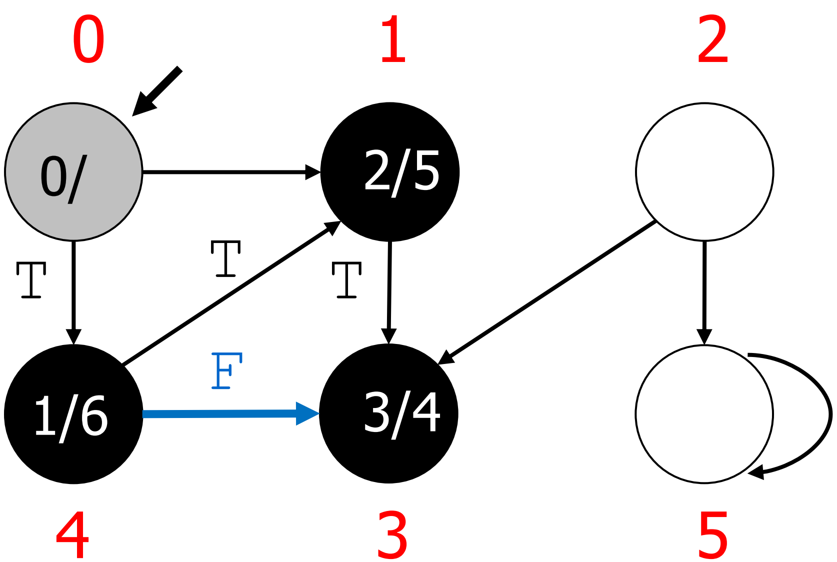


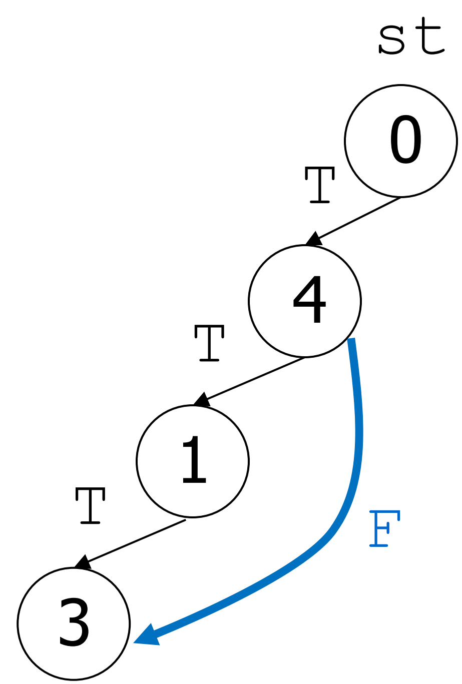
st



st	0	4	-1	1	0	-1
	0	1	2	3	4	5

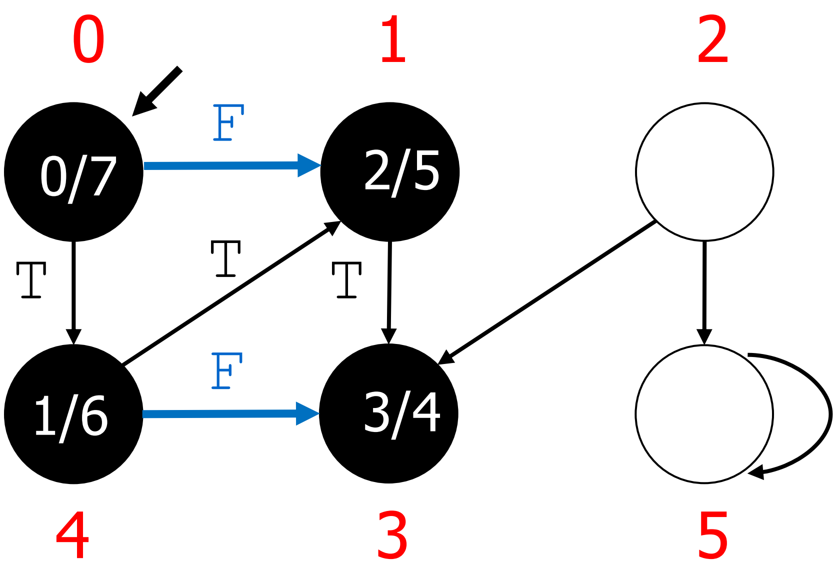
time = 6

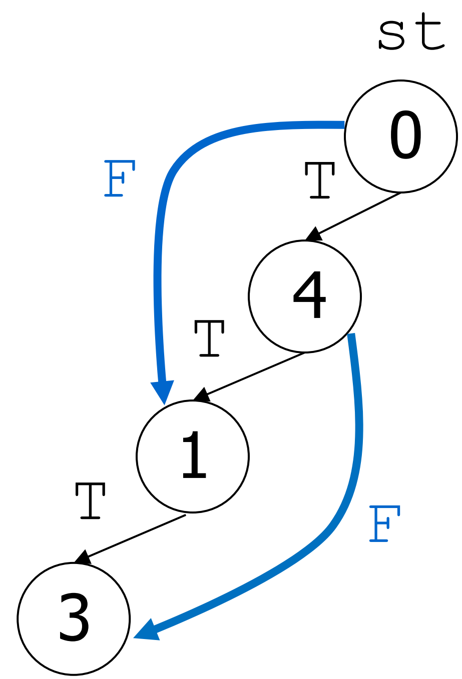




	0	1	2	3	4	5
st	0	4	-1	1	0	-1

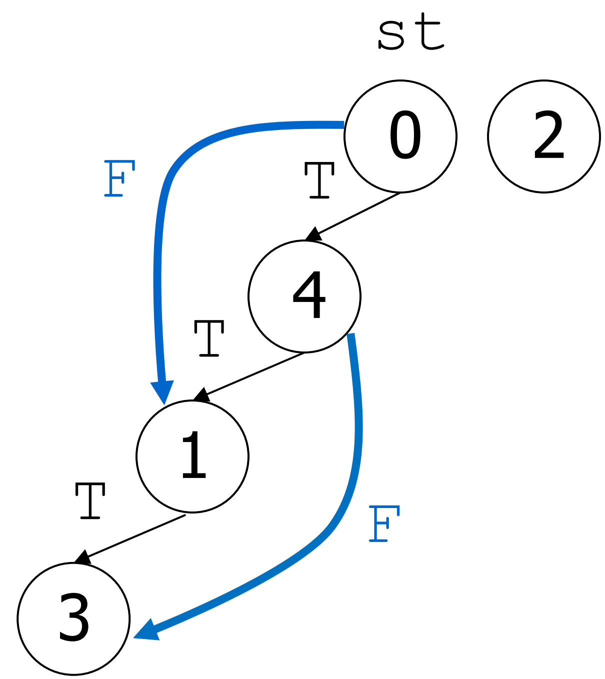
time = 7

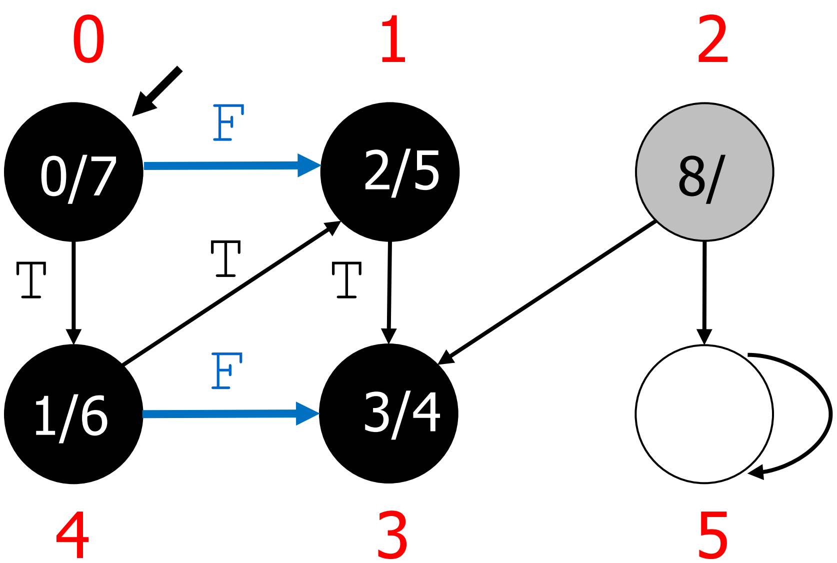




st	0	4	-1	1	0	-1
	0	1	2	3	4	5

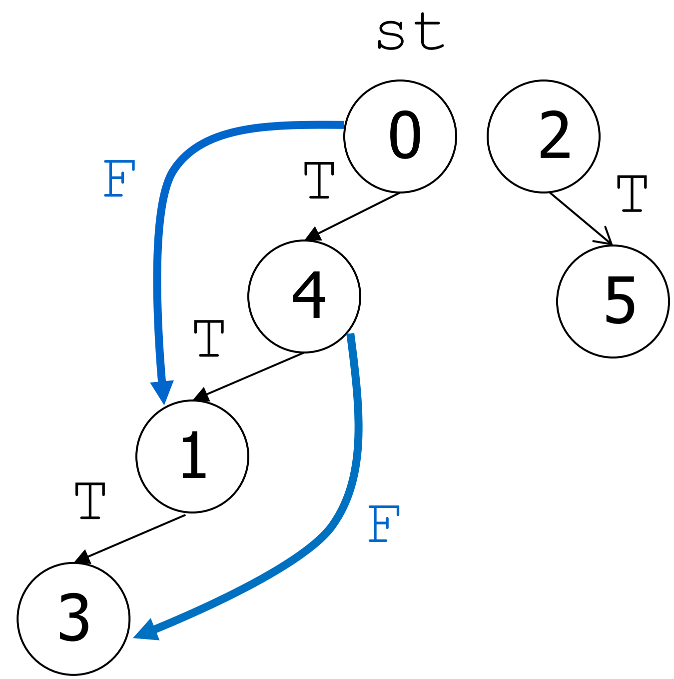
time = 8

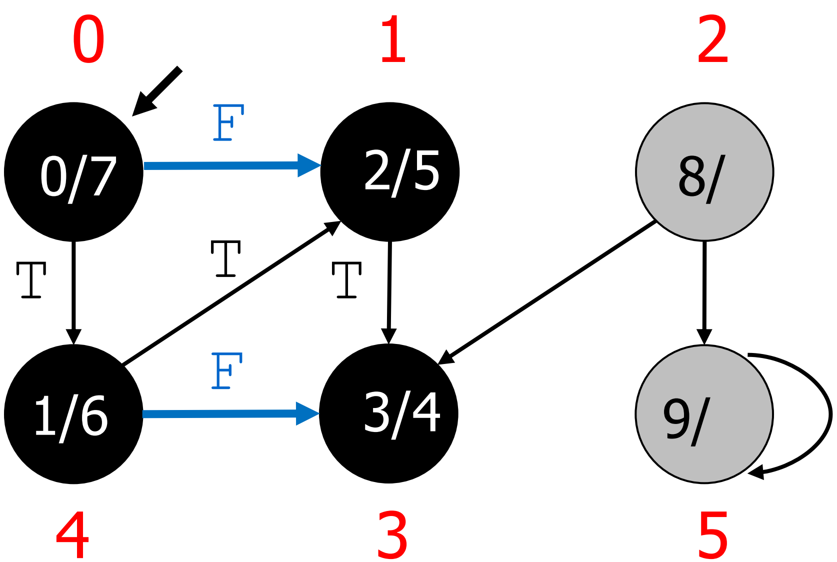




st	0	4	2	1	0	-1
	0	1	2	3	4	5

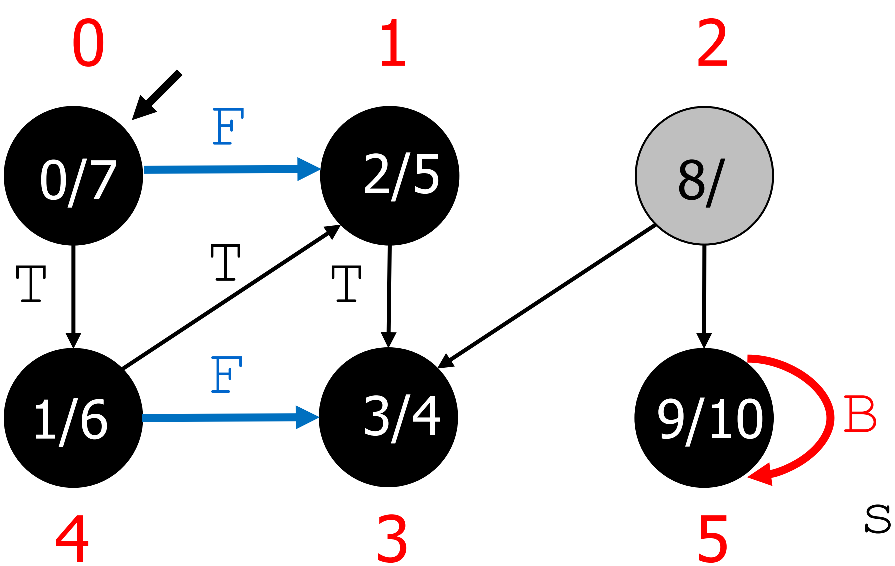
time = 9

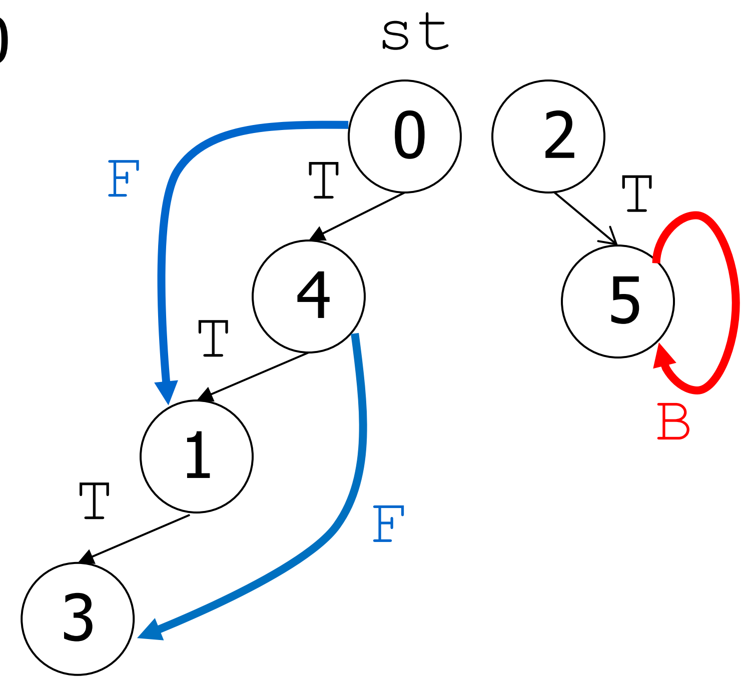




st	0	4	2	1	0	2
	0	1	2	3	4	5

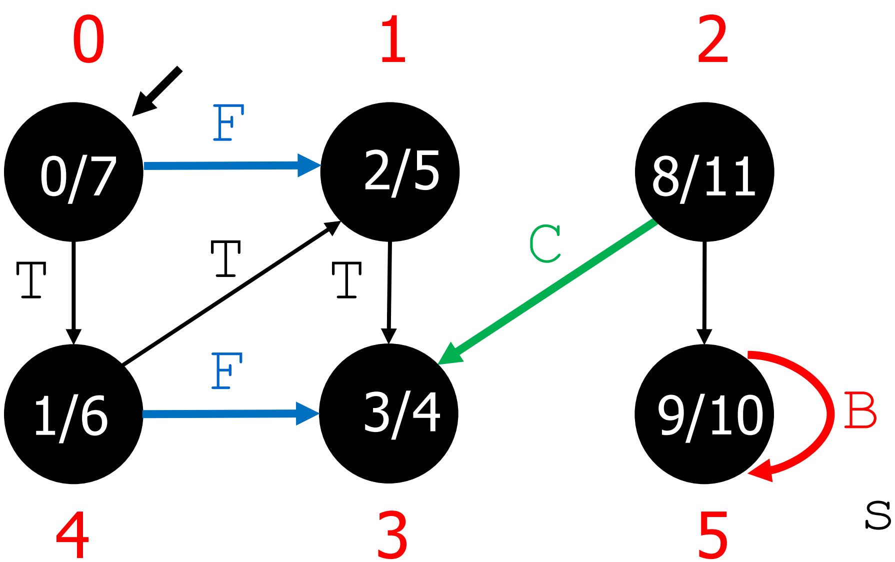
time = 10

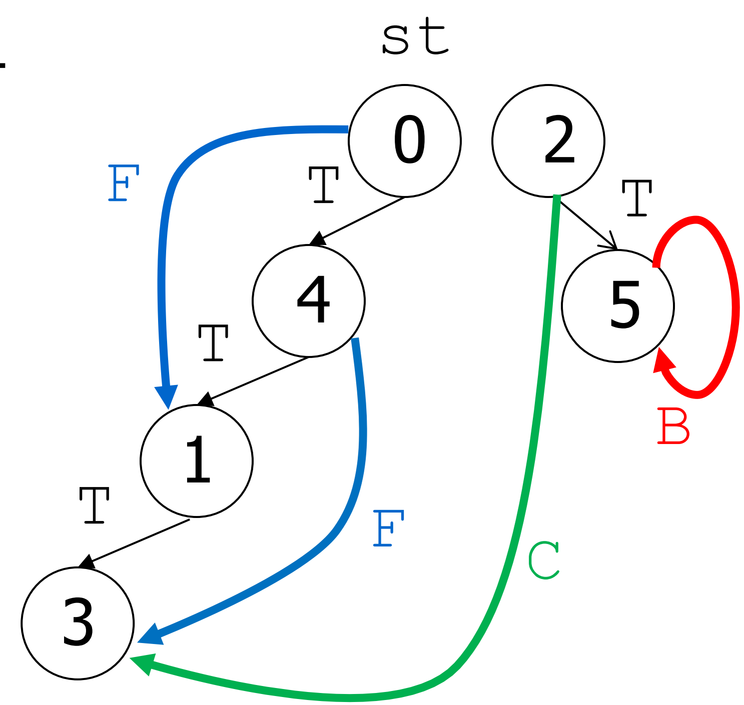




st	0	4	2	1	0	2
	0	1	2	3	4	5

time = 11





	0	1	2	3	4	5
st	0	4	2	1	0	2

Algoritmo

wrapper

- `GRAPHdfs`: funzione che visita tutti i vertici di un grafo, richiamando la procedura ricorsiva `dfsR`. Termina quando tutti i vertici sono neri.
- `dfsR`: funzione che visita in profondità a partire da un vertice `v` identificato fittiziamente come `EDGEcreate(v,v)`.

  Termina quando ha visitato in profondità tutti i nodi raggiungibili da `v`.

NB: alcuni autori chiamano visita in profondità la sola `dfsR`.

Strutture dati

- grafo non pesato come lista delle adiacenze
- vettori dove per ciascun vertice:
  - si registra il tempo di scoperta (numerazione in preordine dei vertici) `pre[i]`
  - si registra il tempo di completamento (numerazione in postordine dei vertici) `post[i]`
  - si registra il padre per la costruzione della foresta degli alberi della visita in profondità: `st[i]`
- contatore `time` per tempi di scoperta/completamento
- `time, *pre, *post` e `*st` sono locali alla funzione `GRAPHdfs` e passati by reference alla funzione ricorsiva `dfsR`.

```
void GRAPHdfs(Graph G) {
  int v, time=0, *pre, *post, *st;
  pre = malloc(G->V * sizeof(int));
  post = malloc(G->V * sizeof(int));
  st = malloc(G->V * sizeof(int));
  for (v=0;v<G->V;v++) {pre[v]=-1; post[v]=-1; st[v]=-1; }
  for (v=0; v < G->V; v++)
    if (pre[v]==-1)
      dfsR(G,EDGEcreate(v,v),&time,pre,post,st);
  printf("discovery/endprocessing time labels \n");
  for (v=0; v < G->V; v++)
    printf("%s:%d/%d\n",STretrieve(G->tab,v),pre[v],post[v]);
  printf("resulting DFS tree \n");
  for (v=0; v < G->V; v++)
     printf("%s's parent: %s \n", STretrieve(G->tab, v),
                STretrieve(G->tab, st[v]));
}
```

```
void dfsR(Graph G, Edge e, int *time,
          int *pre, int *post, int *st){
  link t;
  int v, w = e.w;
  Edge x;

  if (e.v != e.w)
    printf("(%s, %s): T\n",STretrieve(G->tab,e.v),
            STretrieve(G->tab, e.w)) ;
  st[e.w] = e.v;
  pre[w] = (*time)++;
  for (t = G->adj[w]; t != G->z; t = t->next)
    if (pre[t->v] == -1)
      dfsR(G, EDGEcreate(w, t->v), time, pre, post, st);
    else {
      v = t->v;
      x = EDGEcreate(w, v);
```

condizione di terminazione implicita della ricorsione grafi non orientati

```
      if (pre[w] < pre[v])
       printf("(%s, %s): B\n", STretrieve(G->tab, x.v),
                STretrieve(G->tab,x.w)) ;
     if (post[v] == -1)
       printf("(%s, %s): B\n", STretrieve(G->tab, x.v),
               STretrieve(G->tab, x.w));
     else
       if (pre[v] > pre[w])
         printf("(%s, %s): F\n", STretrieve(G->tab, x.v),
                 STretrieve(G->tab, x.w));
       else
         printf("(%s, %s): C\n", STretrieve(G->tab, x.v),
                 STretrieve(G->tab, x.w));
   }
  post[w] = (*time)++;
}
```

grafi orientati

Complessità (lista adiacenze)

$\Theta(|V|)$

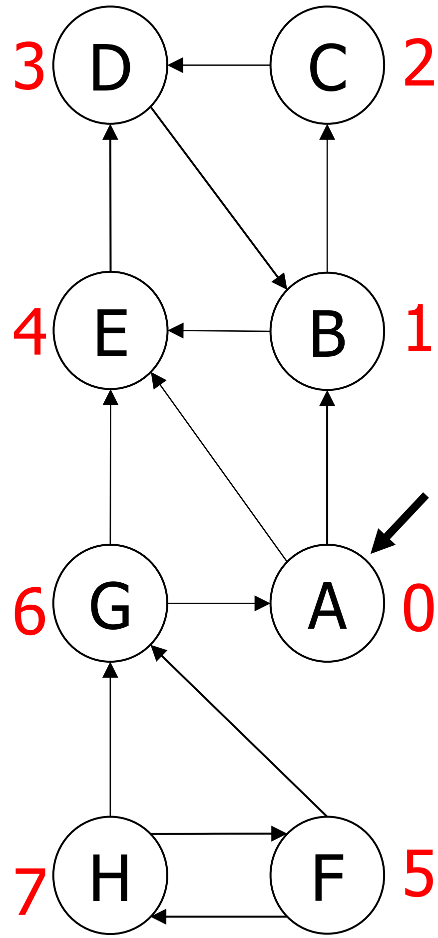
$\Theta(|E|)$

- Inizializzazione
- visita ricorsiva da u
- T(n) = $\Theta(|V|+|E|)$.
- Con la matrice delle adiacenze: T(n) = $\Theta(|V|^2)$.

Esempio

in.txt

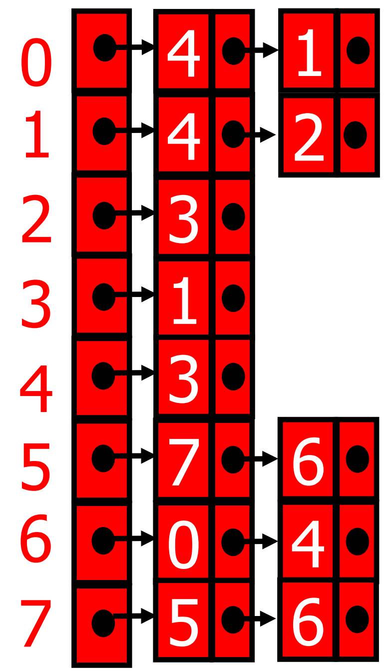
A B
B C
C D
A E
F G
F H
D B
E D
B E
G E
H G
H F
G A

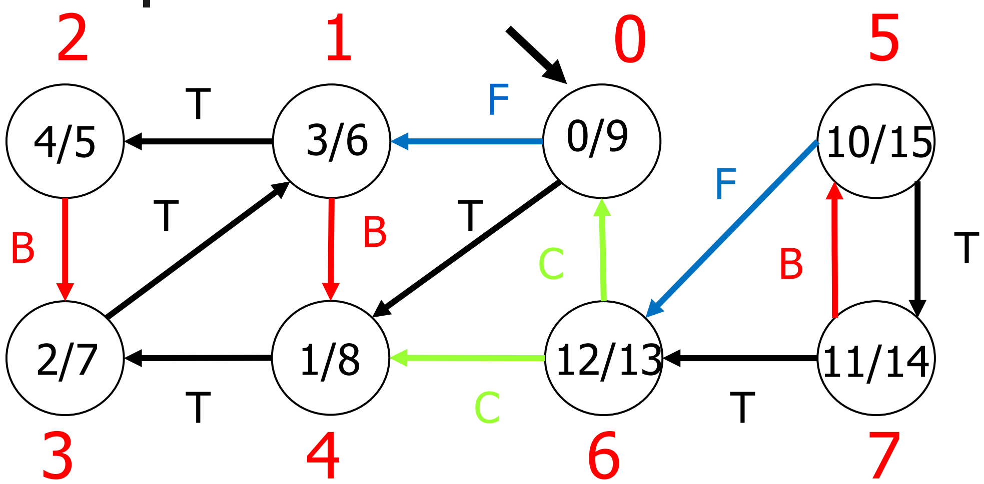


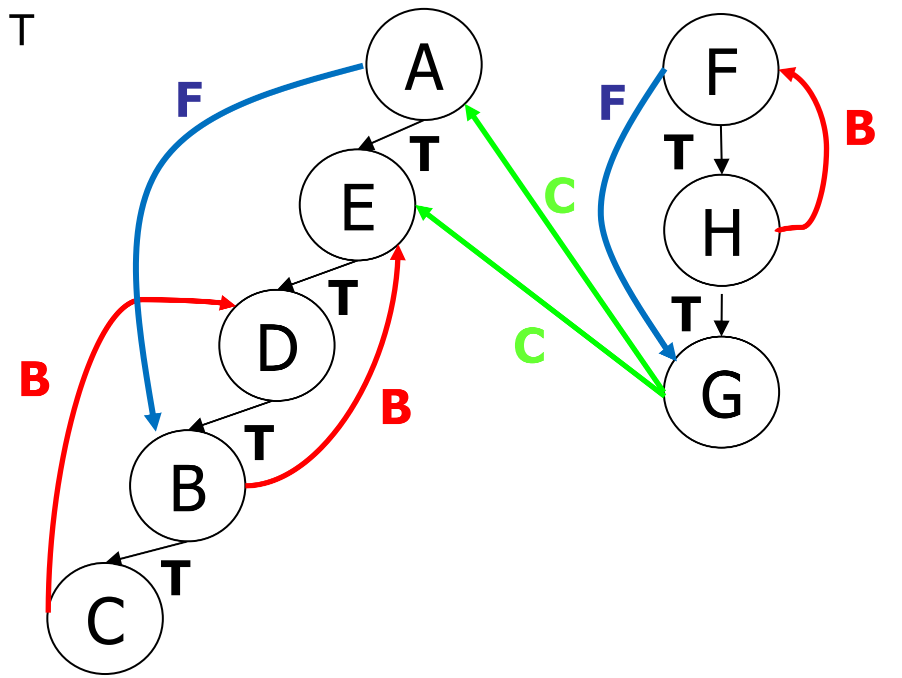
ST

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H

Lista delle adiacenze

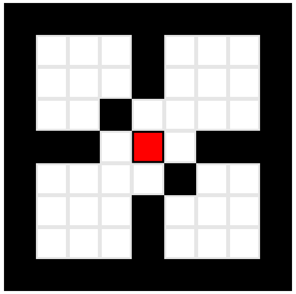



2
1
0
5
4/5
3/6
0/9
10/15
2/7
1/8
12/13
11/14
3
4
6
7
T
F
B
C


A
E
D
B
C
F
H
G
T
F
B
C

Applicazione: flood fill

- Scopo: colorare un'intera area di pixel connessi con lo stesso colore (Bucket Tool)
- DFS a partire dal pixel sorgente (seed), terminazione quando si incontra una frontiera (boundary):



http://en.wikipedia.org

Sedgewick, Wayne, Algorithms Part I & II, www.coursera.org

Visita in ampiezza

A partire da un vertice s:

- determina tutti i vertici raggiungibili da s, quindi non visita necessariamente tutti i vertici a differenza della DFS
- calcola la distanza minima da s di tutti i vertici da esso raggiungibili.
- genera un albero della visita in ampiezza.

Ampiezza: espande tutta la frontiera tra vertici già scoperti/non ancora scoperti.

Principi base

Scoperta di un vertice: prima volta che si incontra nella visita.

Vertici:

- bianchi: non ancora scoperti
- grigi: scoperti, ma non completati
- neri: scoperti e completati.

Dato un vertice u, il vettore `st[u]` registra il padre di u nell'albero della visita.

Strutture dati

- grafo non pesato come matrice delle adiacenze
- coda `Q` dei vertici grigi (esterna al grafo)
- vettore `st` dei padri nell'albero di visita in ampiezza
- vettore `pre` dei tempi di scoperta dei vertici
- contatore `time` del tempo
- `time`, `*pre` e `*st` sono locali alla funzione `GRAPHbfs` e passati by reference alla funzione `bfs`.

wrapper

Algoritmo

Algoritmo:

- estrai un vertice dalla coda
- metti in coda tutti i vertici bianchi ad esso adiacenti (metti in coda tutti gli archi che puntano ai vertici ancora bianchi ad esso adiacenti)
- ripeti finché la coda si svuota

`bfs`: funzione che visita in ampiezza a partire da un vertice di partenza.

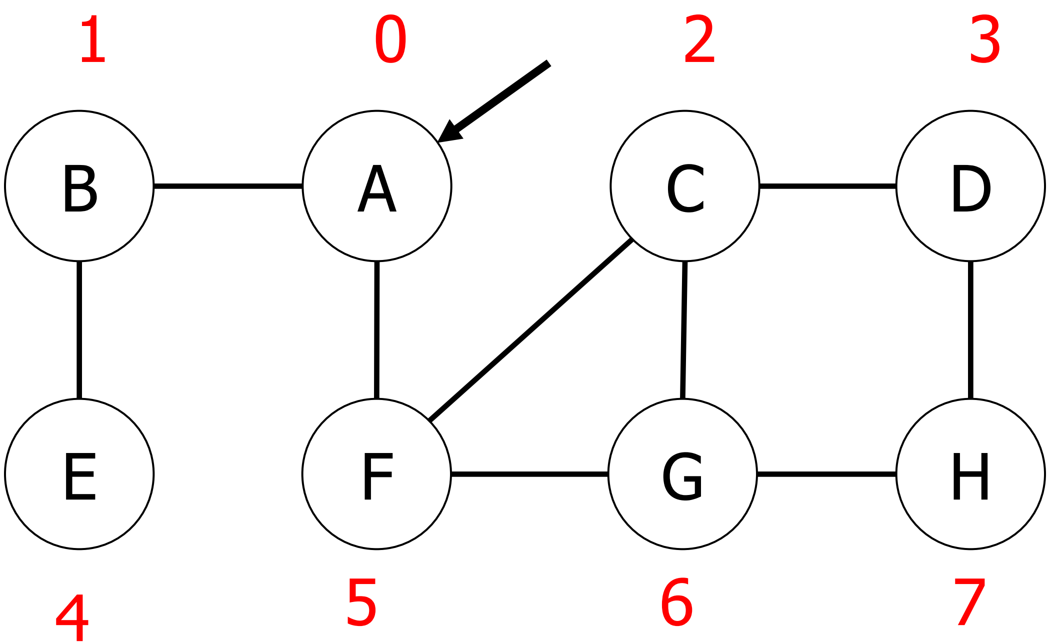
Esempio

in.txt

```
A B
C D
B E
C F
F G
D H
A F
C G
G H
```

ST

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H



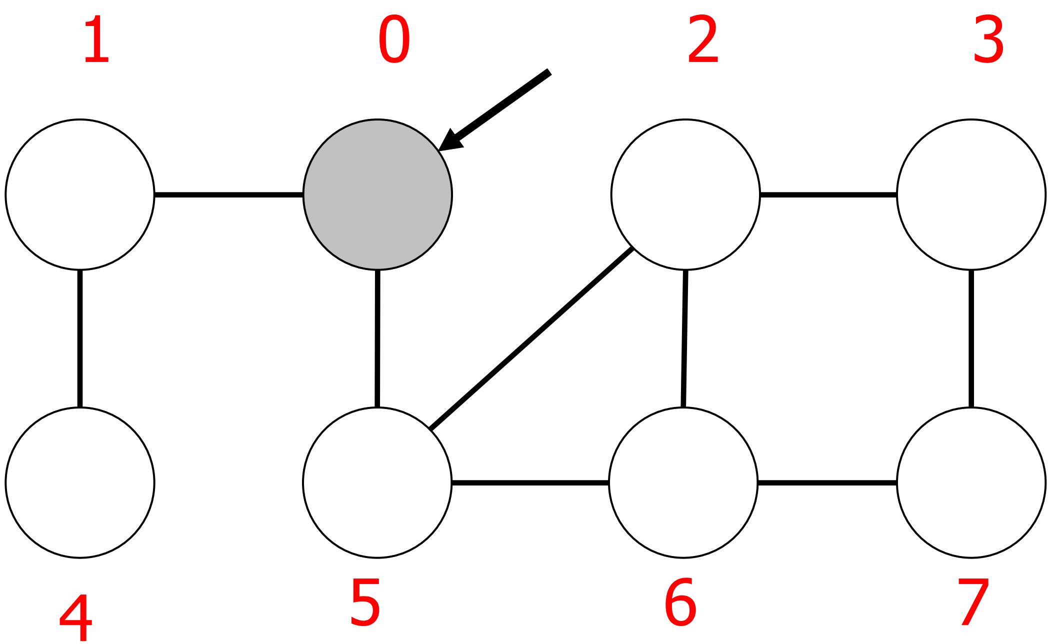
st

-1	-1	-1	-1	-1	-1	-1	-1
0	1	2	3	4	5	6	7

Nella coda Q sono riportati i vertici bianchi, non gli archi

Q 0

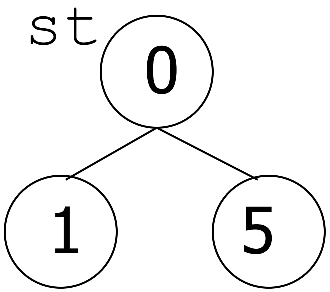
st 0

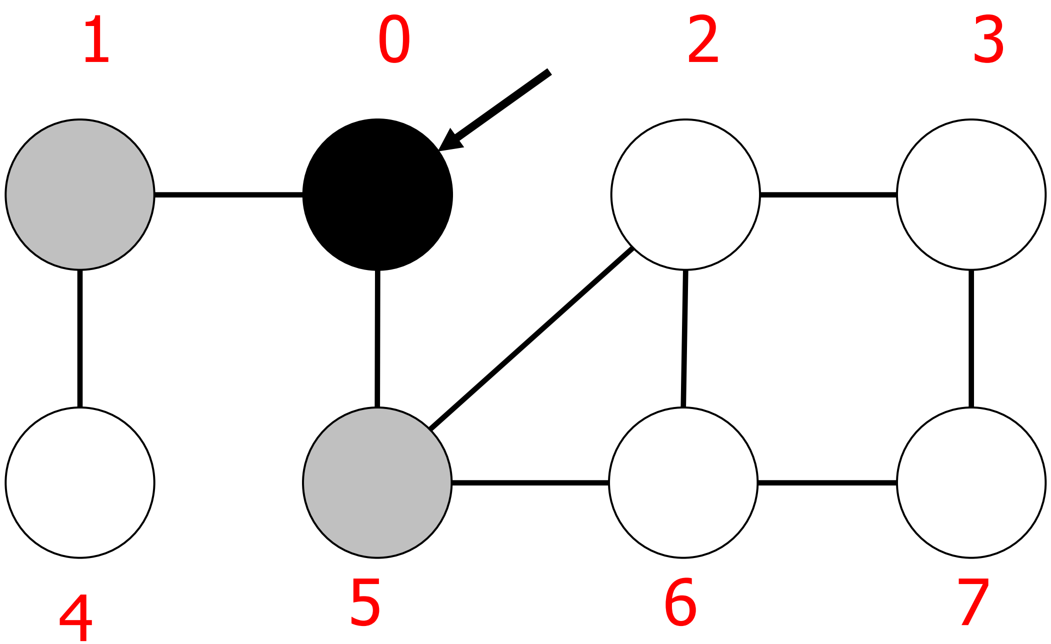


st

0	1	2	3	4	5	6	7
0	-1	-1	-1	-1	-1	-1	-1

Q		
	1	5



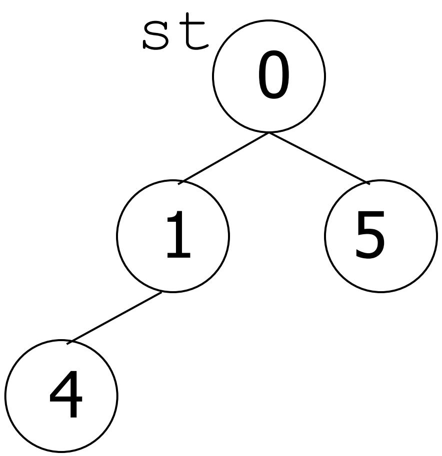


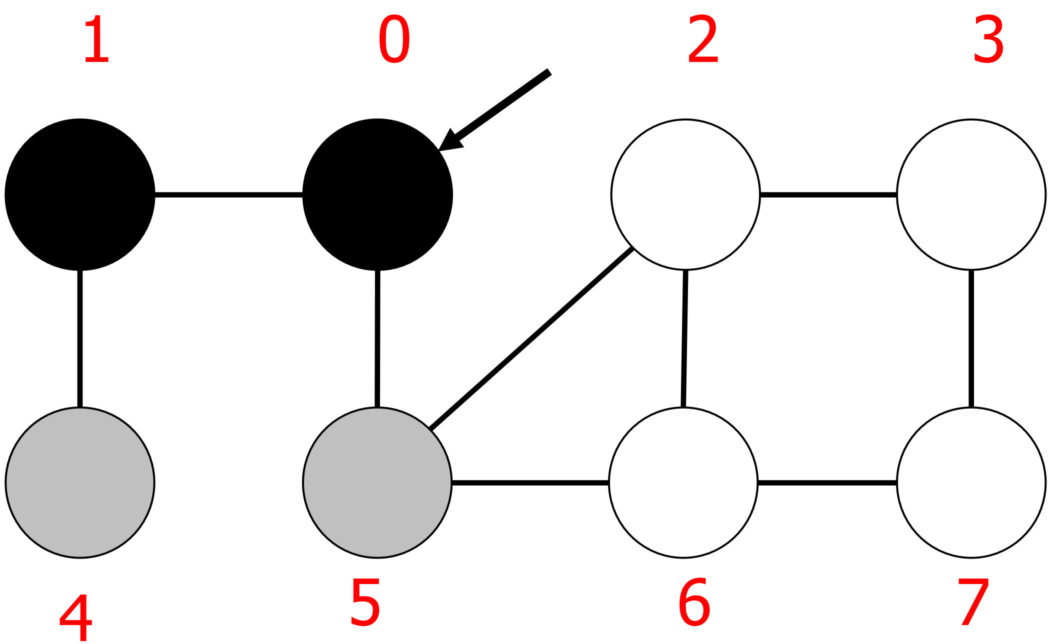
st

0	1	2	3	4	5	6	7
0	0	-1	-1	-1	0	-1	-1

Q

5	4

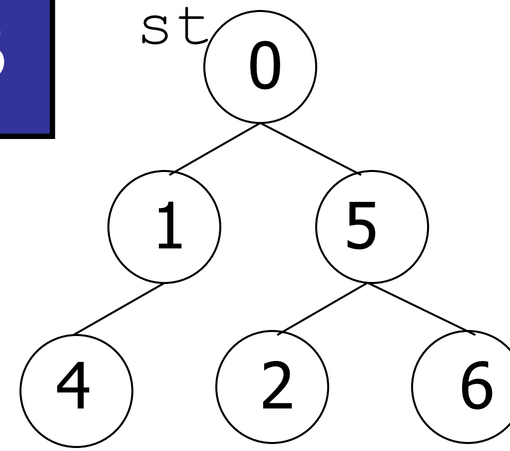


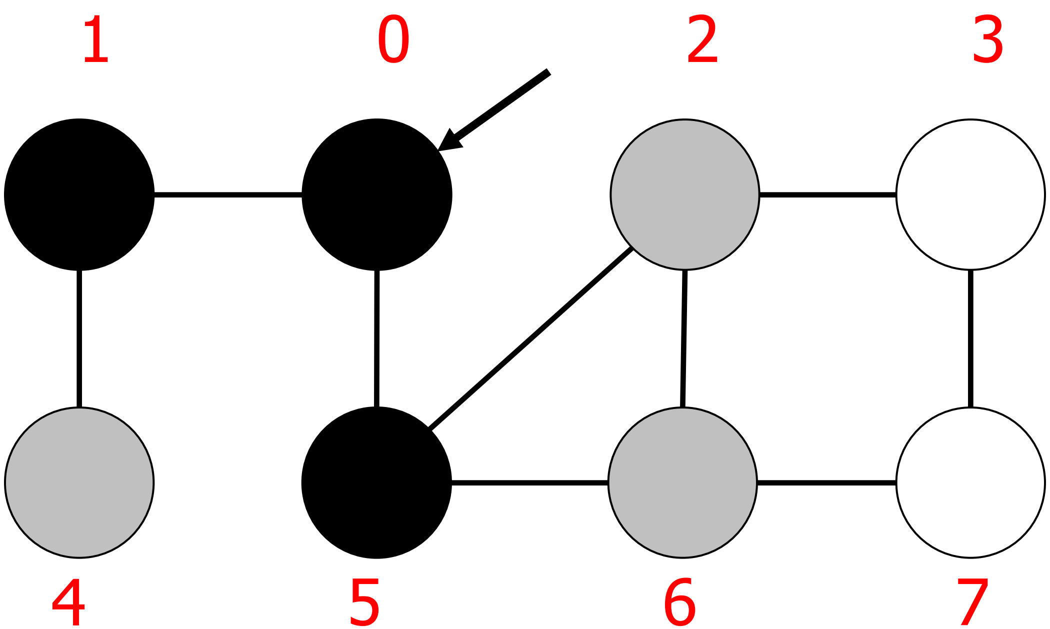


st

0	1	2	3	4	5	6	7
0	0	-1	-1	1	0	-1	-1

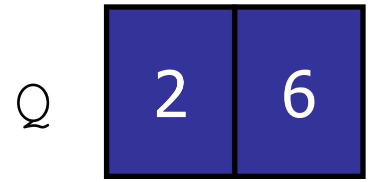
Q			
	4	2	6

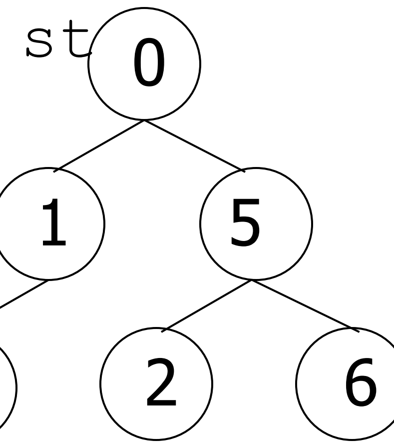


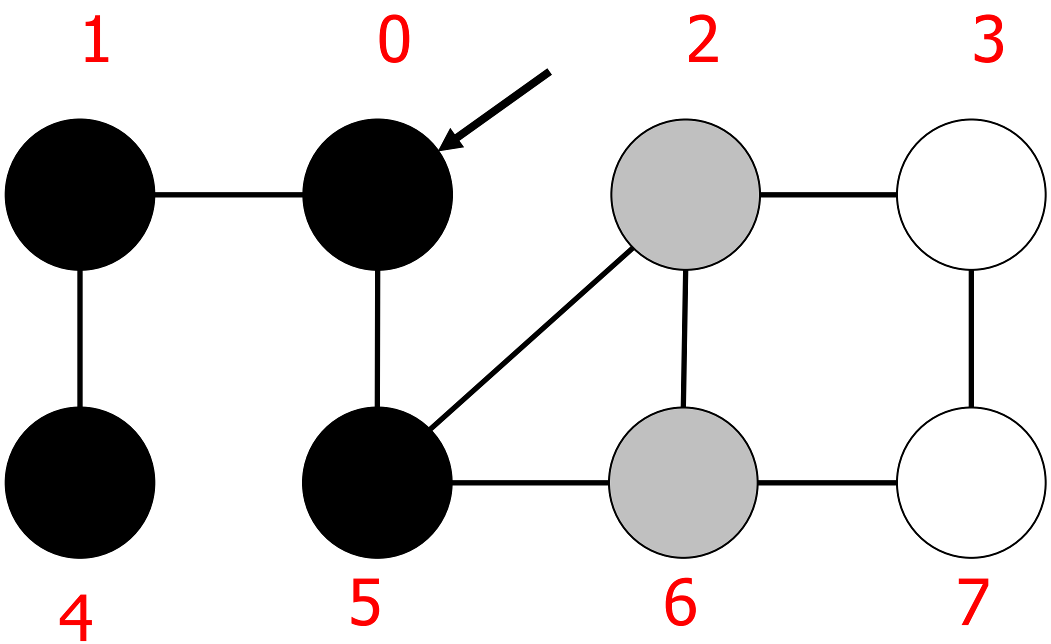


st

0	1	2	3	4	5	6	7
0	0	5	-1	1	0	5	-1





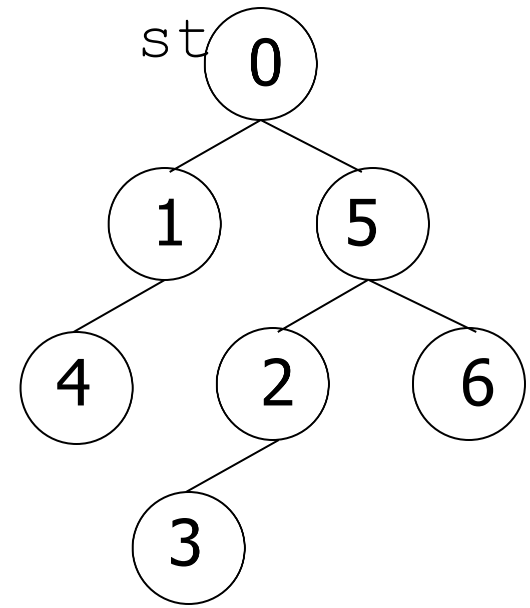


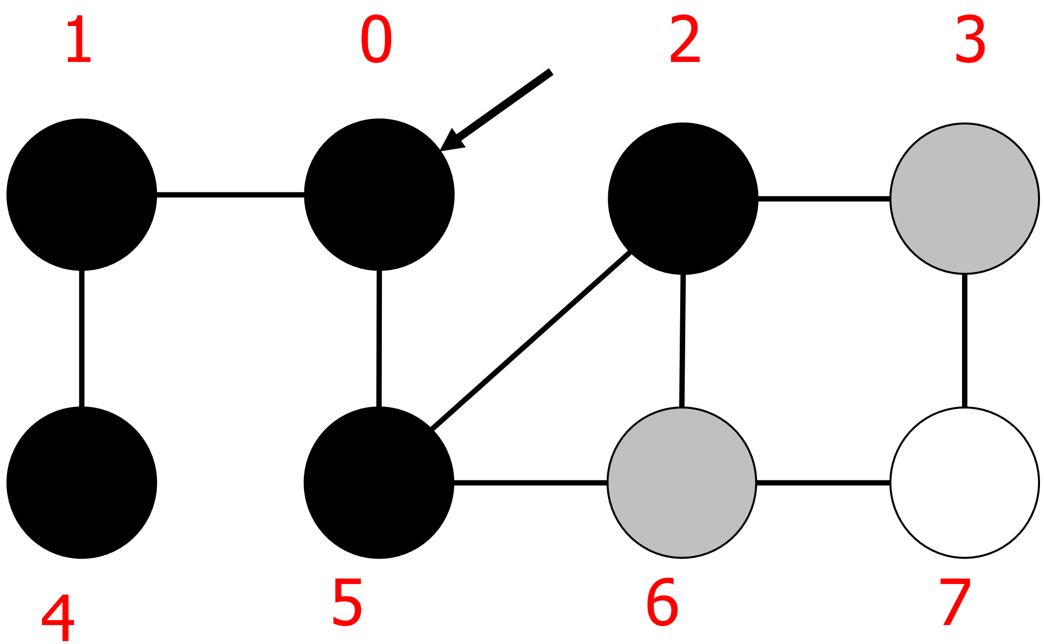
st

0	1	2	3	4	5	6	7
0	0	5	-1	1	0	5	-1

Q

6	3

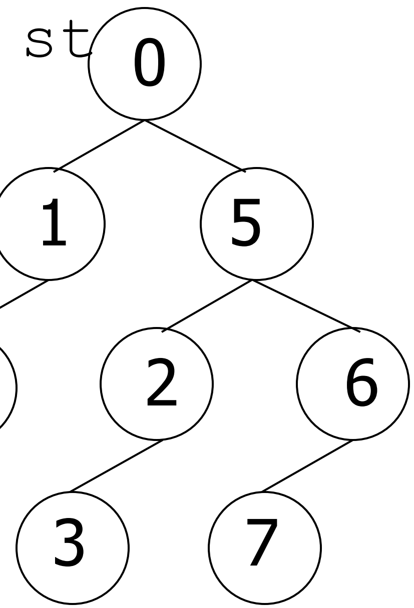


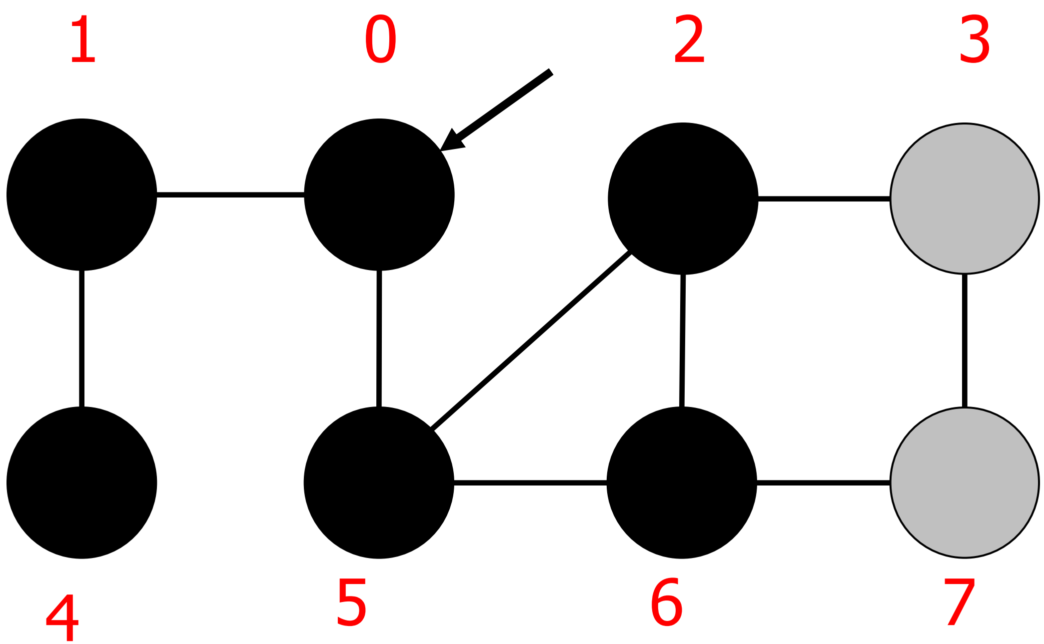


st

0	0	5	2	1	0	5	-1
0	1	2	3	4	5	6	7

Q		
	3	7





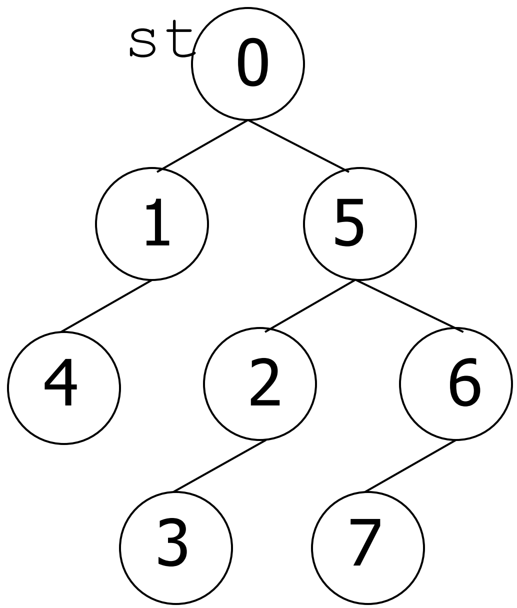
st

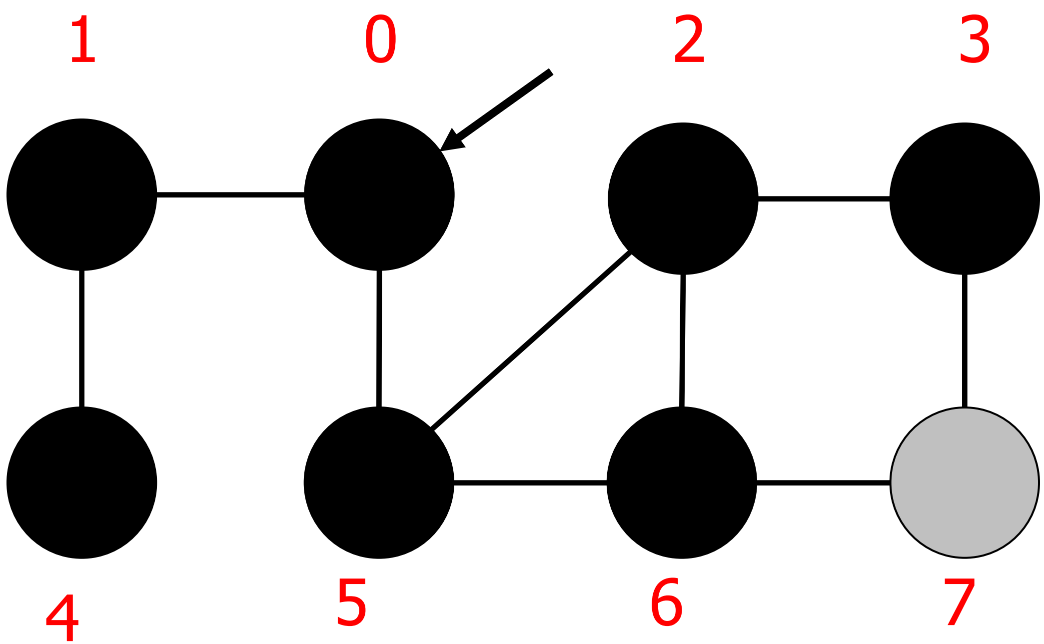
0	1	2	3	4	5	6	7
0	0	5	2	1	0	5	6

Q

7

st

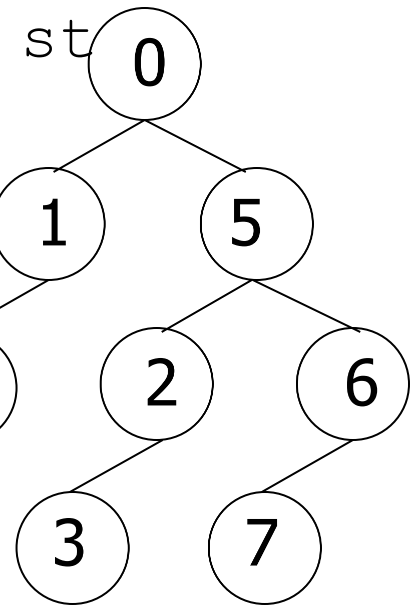


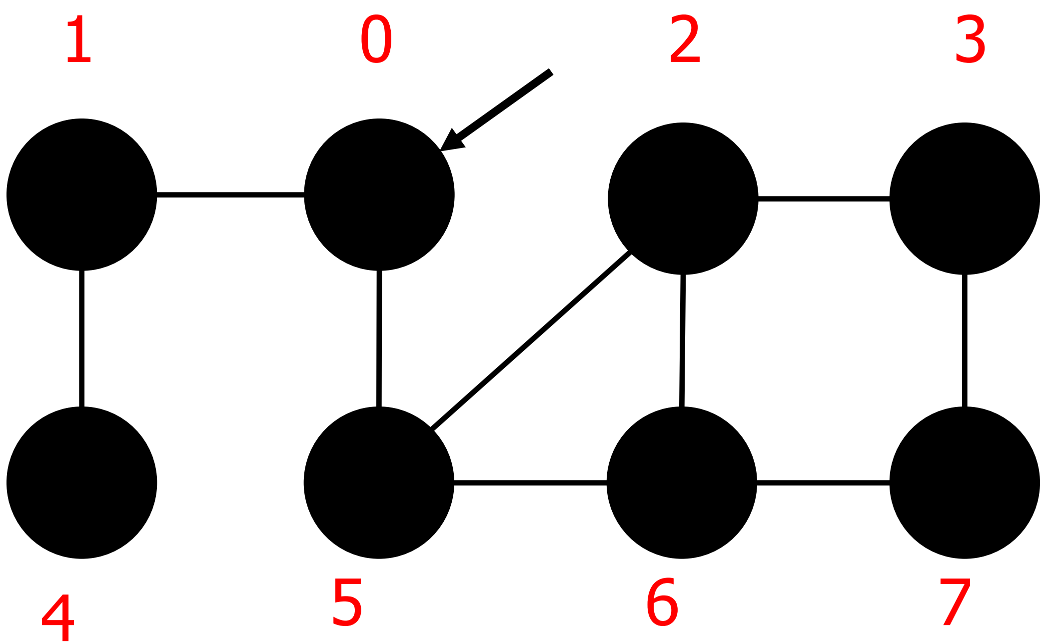


st

0	0	5	2	1	0	5	6
0	1	2	3	4	5	6	7

Q





st

0	0	5	2	1	0	5	6
0	1	2	3	4	5	6	7

```
void GRAPHbfs(Graph G) {
  int v, time=0, *pre, *st;
  pre = malloc(G->V*sizeof(int));
  st = malloc(G->V*sizeof(int));
  for (v=0; v < G->V; v++) {
    pre[v] = -1;
    st[v] = -1;
  }
  bfs(G, EDGEcreate(0,0), &time, pre, st);
  printf("\n Resulting BFS tree \n");
  for (v=0; v < G->V; v++)
    if (st[v] != -1)
      printf("%s's parent is: %s\n", STretrieve(G->tab, v),
                STretrieve(G->tab, st[v]));
}
```

```
void bfs(Graph G, Edge e, int *time, int *pre, int *st) {
  int v;
  Q q = Qinit();
  Qput(q, e);
  while (!Qempty(q))
    if (pre[(e = Qget(q)).w] == -1) {
      pre[e.w] = (*time)++;
      st[e.w] = e.v;
      for (v = 0; v < G->V; v++)
        if (G->adj[e.w][v] == 1)
          if (pre[v] == -1)
            Qput(q, EDGEcreate(e.w, v));
    }
}
```

Matrice delle adiacenze

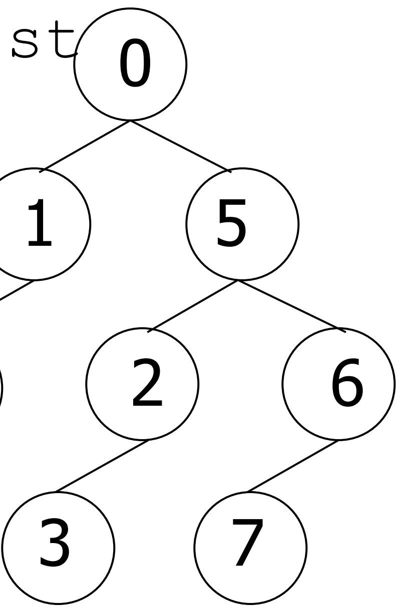
Complessità

- Operazioni sulla coda
- Scansione della matrice delle adiacenze $T(n) = \Theta(|V|^2)$.
- Con la lista delle adiacenze: $T(n) = O(|V|+|E|)$.

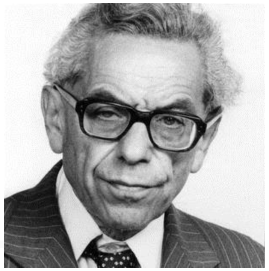
Proprietà

Cammini minimi: la visita in ampiezza determina la minima distanza tra s e ogni vertice raggiungibile da esso.

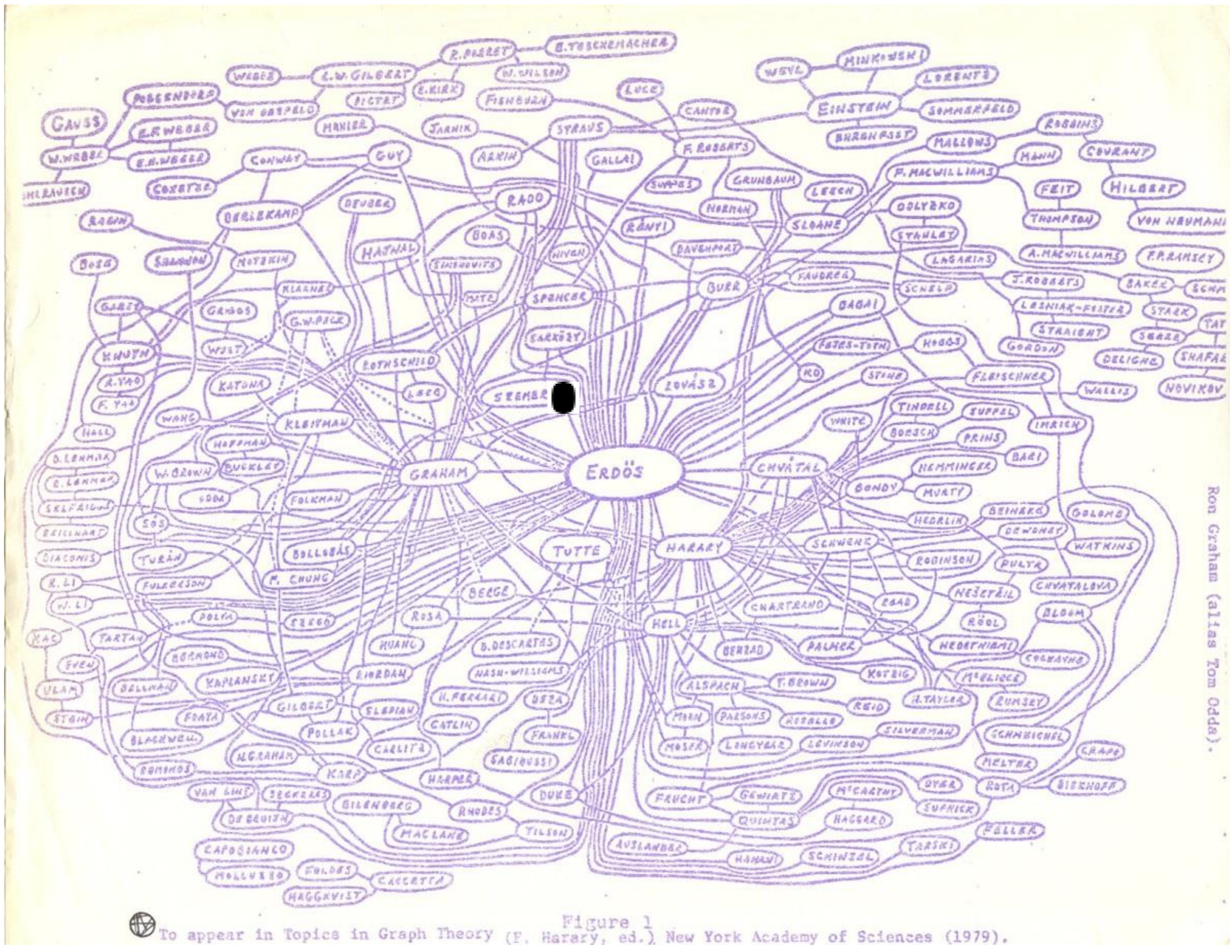
Cammino minimo da 0 a 3:
0, 5, 2, 3
lunghezza = 3



Applicazione: i numeri di Erdős



- Paul Erdős (1913-1996): matematico ungherese «itinerante»: pubblicazioni con moltissimi coautori
- Grafo non orientato:
  - vertici: matematici
  - arco: unisce 2 matematici che hanno una pubblicazione in comune
- numero di Erdős: distanza minima di ogni matematico da Erdős
- BFS



http://www.oakland.edu/upload/images/Erdos%20Number%20Project/cgraph.jpg

Sedgewick, Wayne, Algorithms Part I & II, www.coursera.org

Riferimenti

- Visita in profondità:
  - Sedgewick Part 5 18.2, 18.3, 18.4
  - Cormen 23.3
- Visita in ampiezza:
  - Sedgewick Part 5 18.7
  - Cormen 23.2
- Numero di Erdős:
  - Bertossi 9.5.2