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.

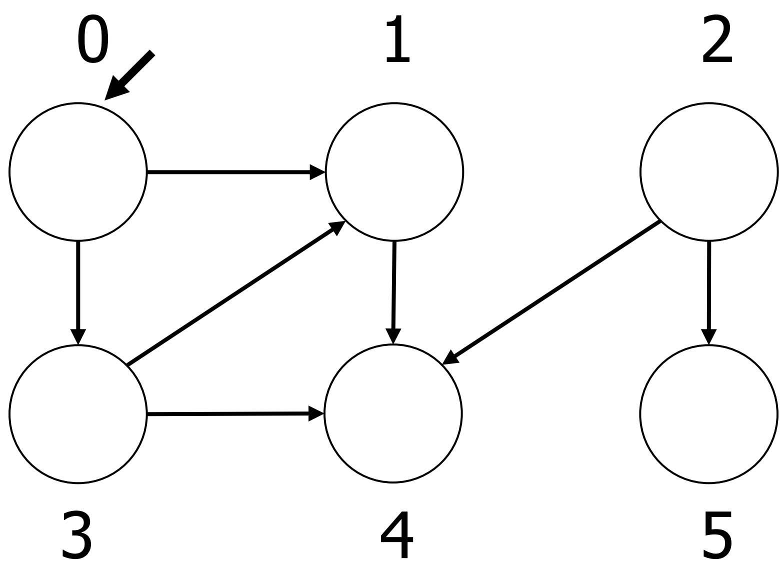
Per ogni vertice si memorizza:

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

Esempio

`time` = -1

`st`



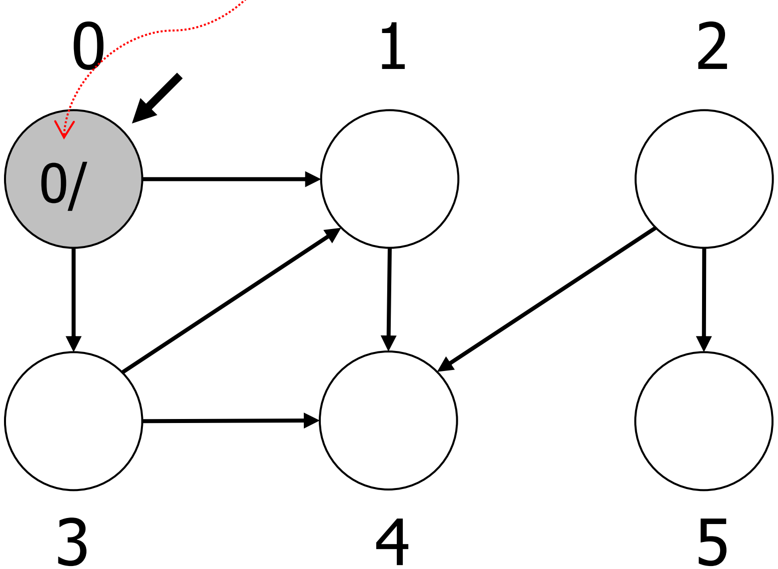
`st`	-1	-1	-1	-1	-1	-1
	0	1	2	3	4	5

time = 0

st

0

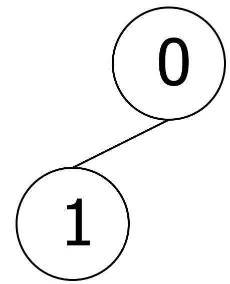
pre[i]/post[i]

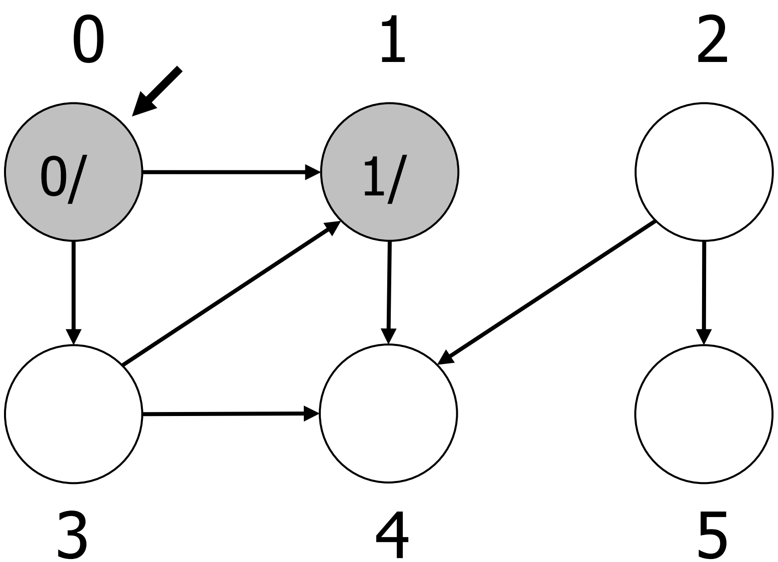


st	0	-1	-1	-1	-1	-1
	0	1	2	3	4	5

`time` = 1

`st`

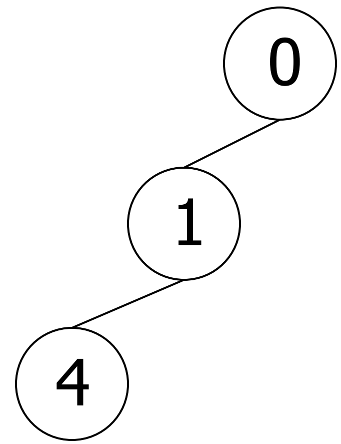


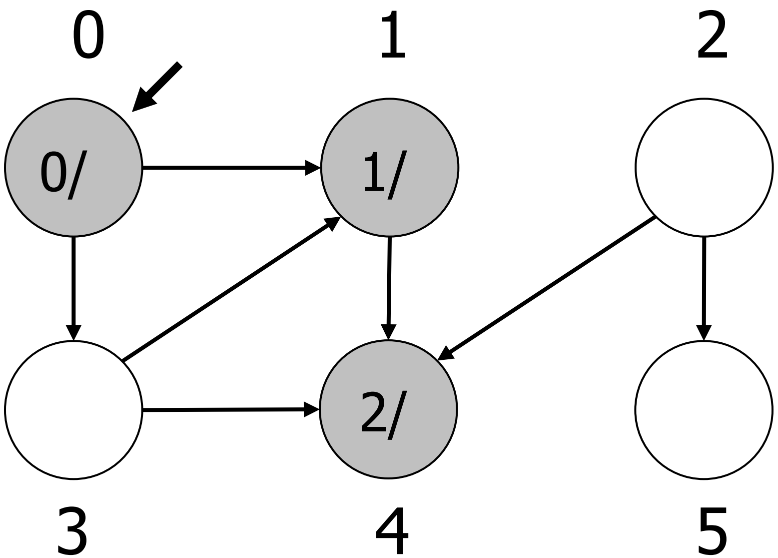


st	0	1	2	3	4	5
	0	0	-1	-1	-1	-1

time = 2

st

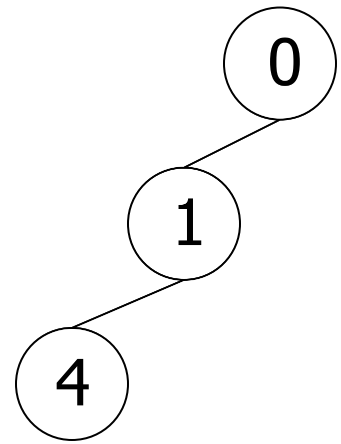


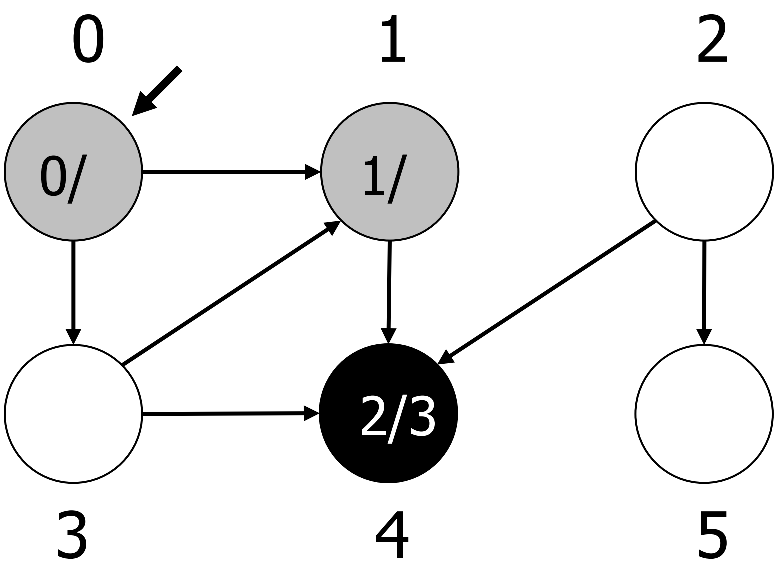


	0	1	2	3	4	5
st	0	0	-1	-1	1	-1

`time` = 3

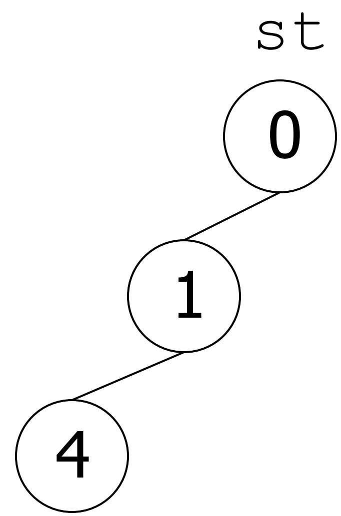
`st`



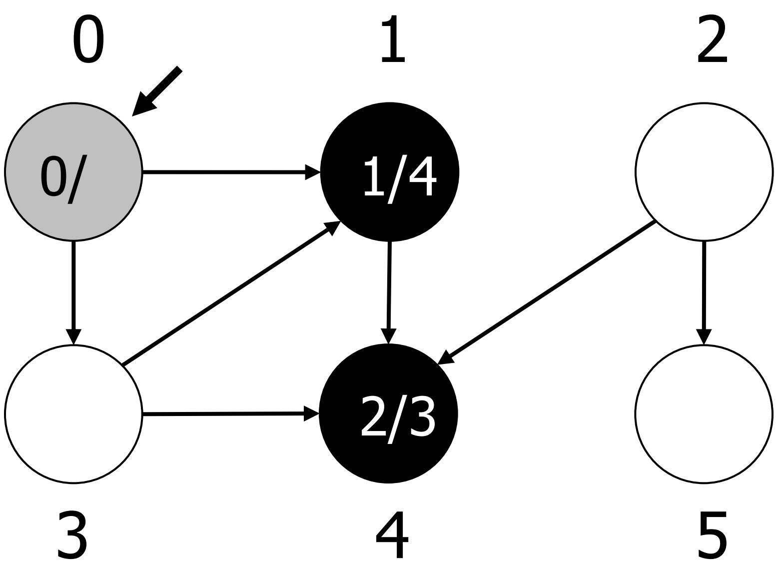


	0	1	2	3	4	5
`st`	0	0	-1	-1	1	-1

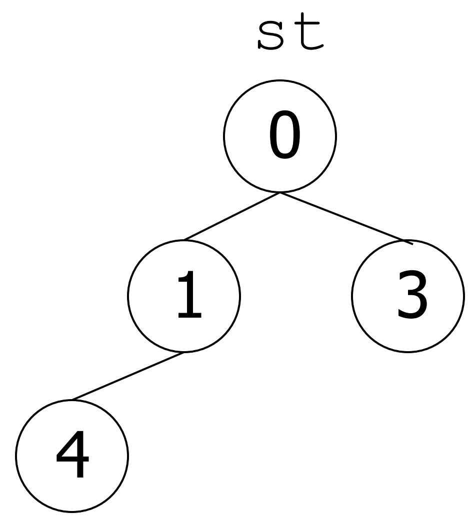
time = 4

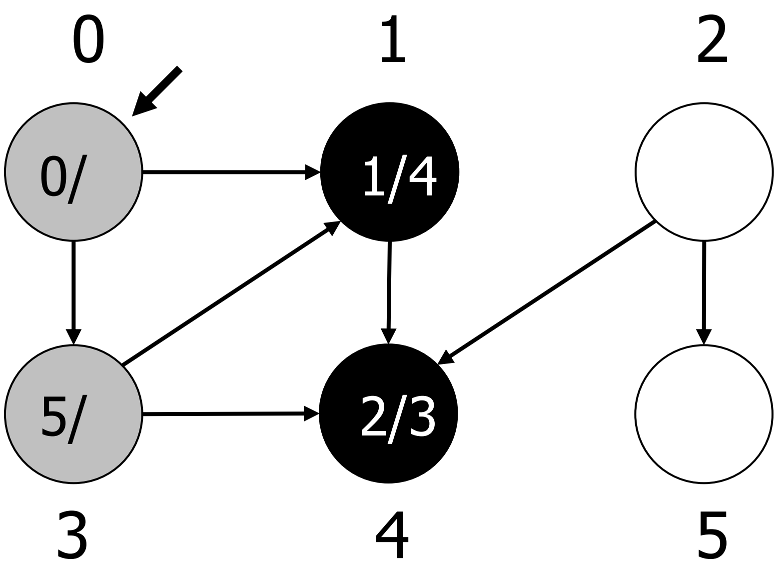


st	0	0	-1	-1	1	-1
	0	1	2	3	4	5



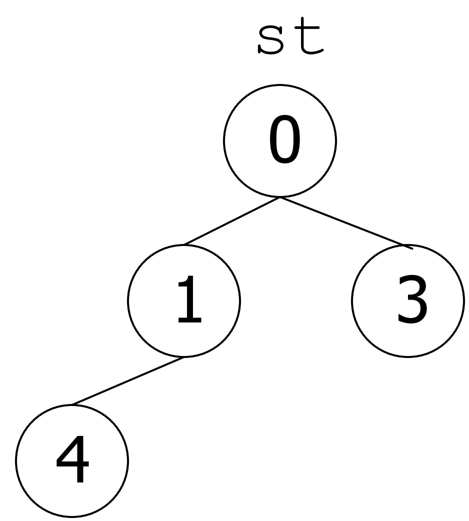
time = 5

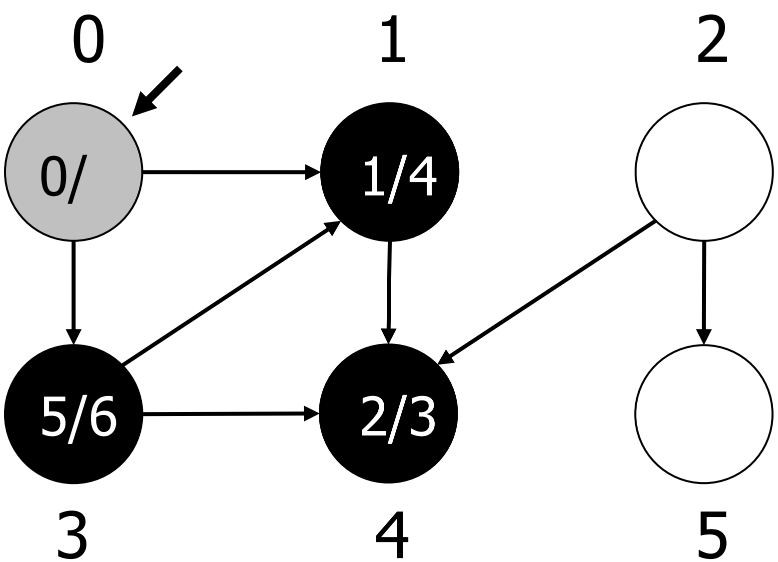




st	0	0	-1	0	1	-1
	0	1	2	3	4	5

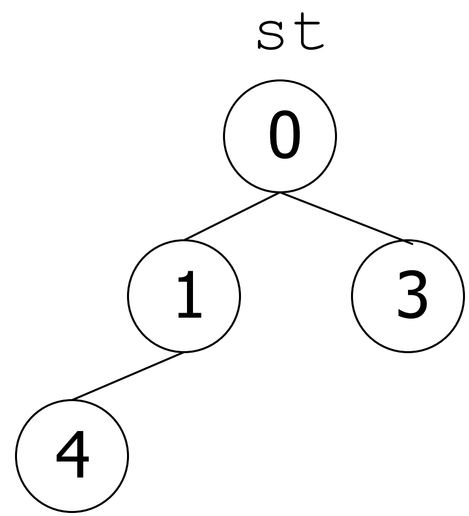
time = 6

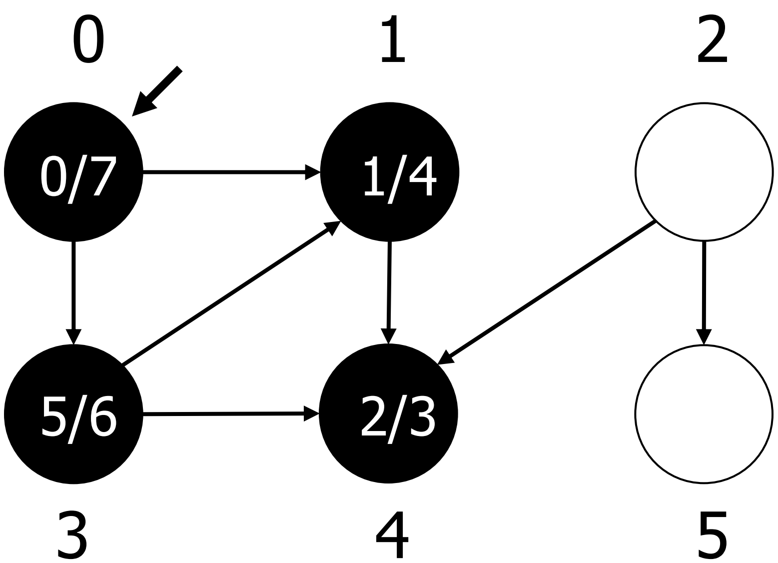




	0	1	2	3	4	5
st	0	0	-1	0	1	-1

`time` = 7

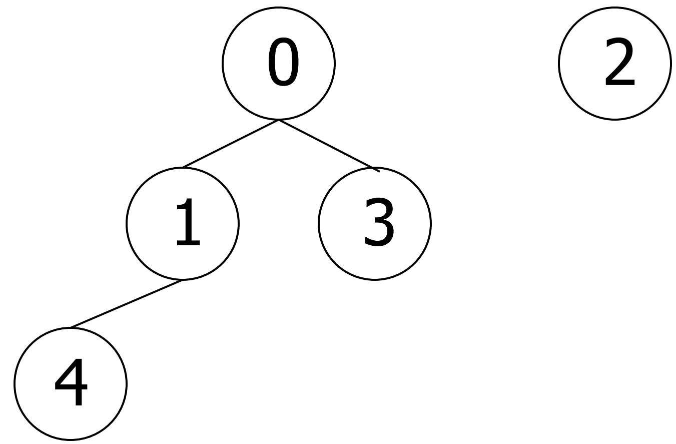


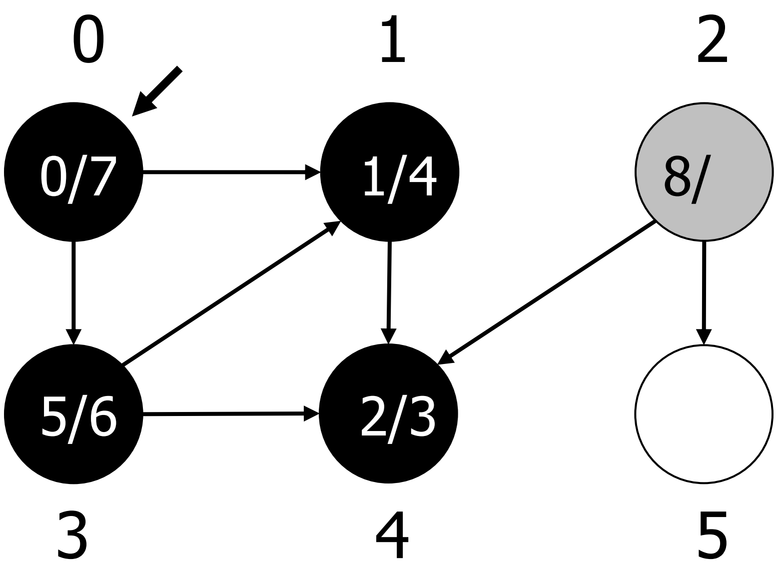


	0	1	2	3	4	5
`st`	0	0	-1	0	1	-1

time = 8

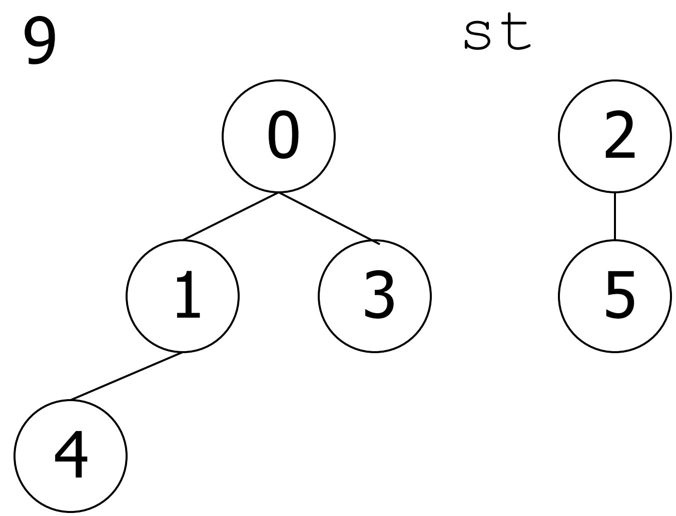
st

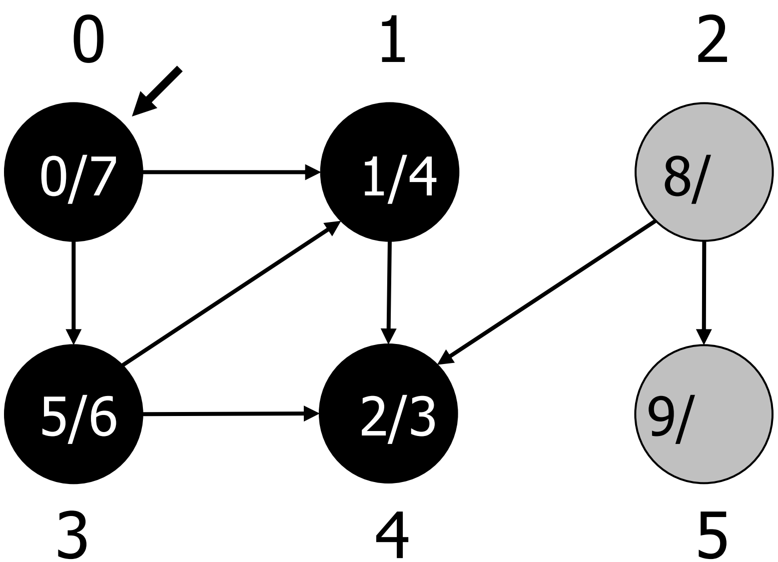




	0	1	2	3	4	5
st	0	0	2	0	1	-1

time = 9

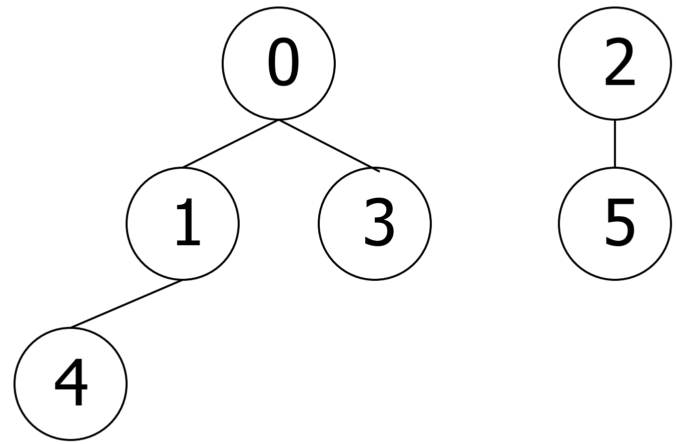


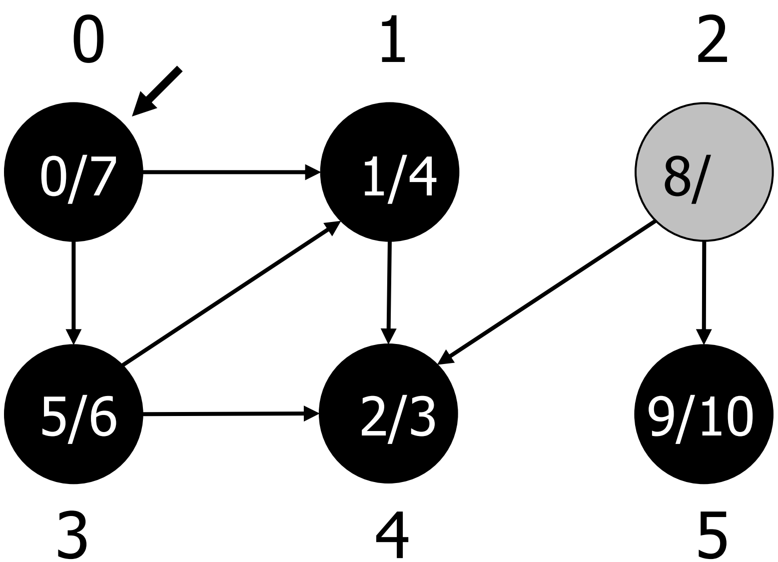


	0	1	2	3	4	5
st	0	0	2	0	1	2

time = 10

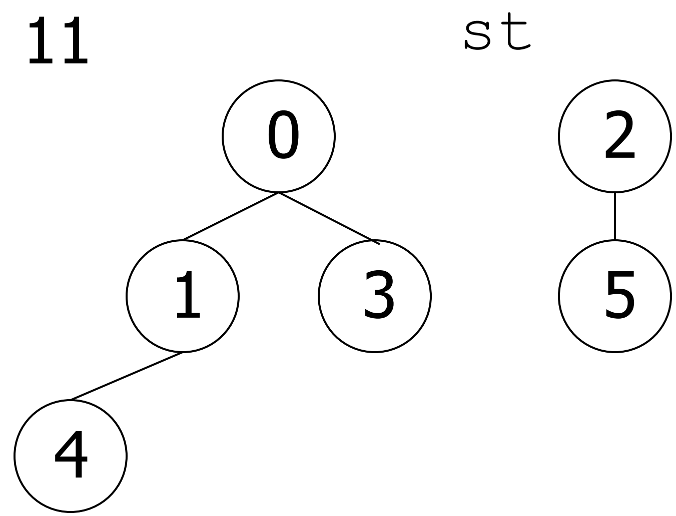
st

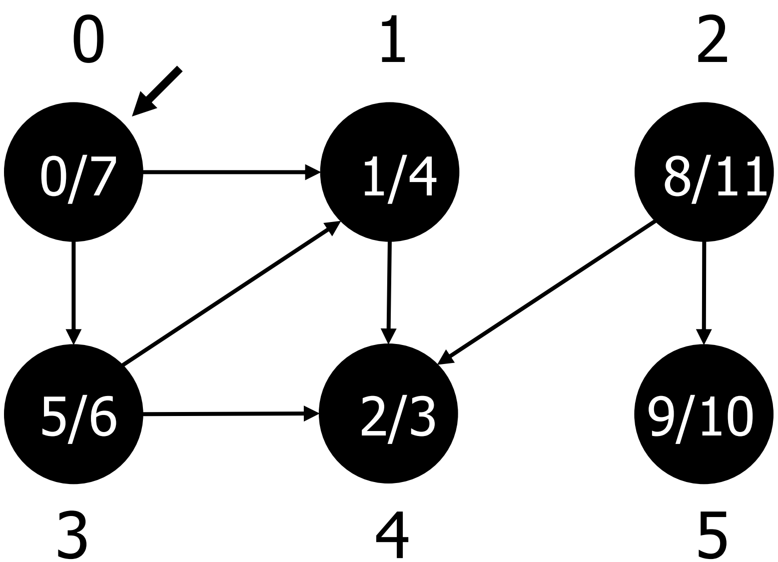




	0	1	2	3	4	5
st	0	0	2	0	1	2

time = 11





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

Classificazione degli archi

Grafo orientato:

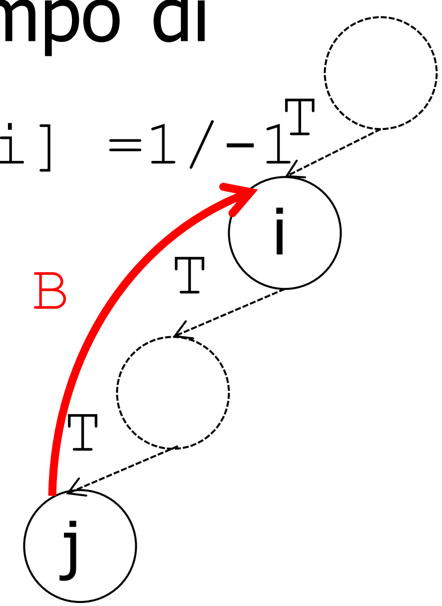
- T: archi dell'albero della visita in profondità
- B: connettono un vertice j ad un suo antenato i nell'albero:

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

  Equivale a testare se

  `post[i] == -1`

`pre[i]/post[i] =1/-1`

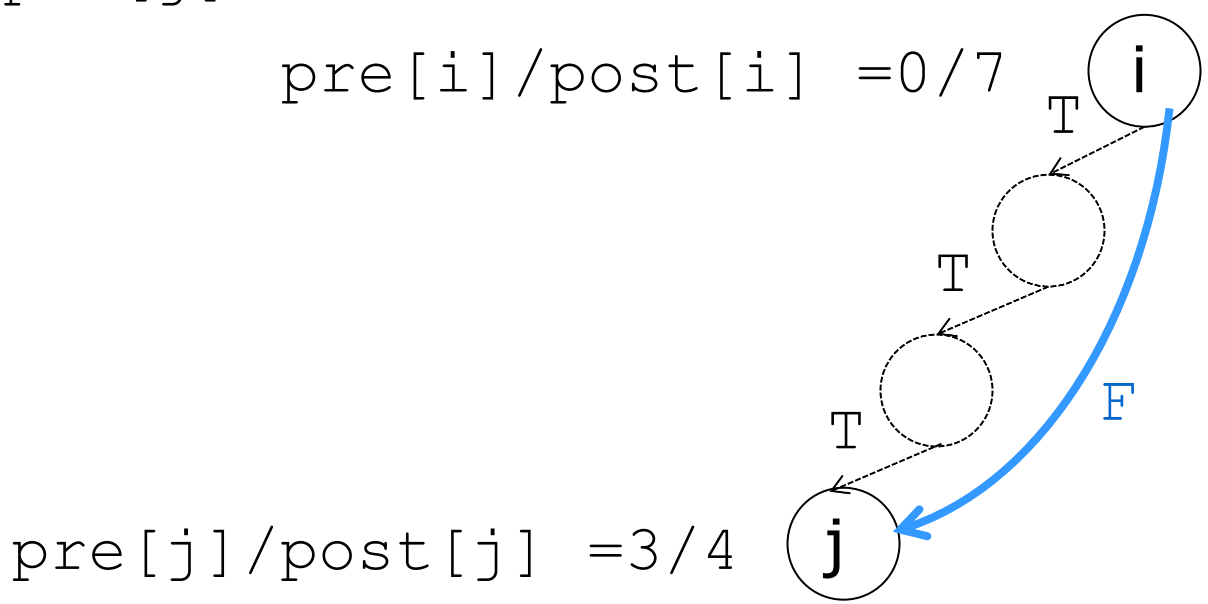


`pre[j]/post[j] =3/4`

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

  tempo di scoperta di i < tempo di scoperta di j

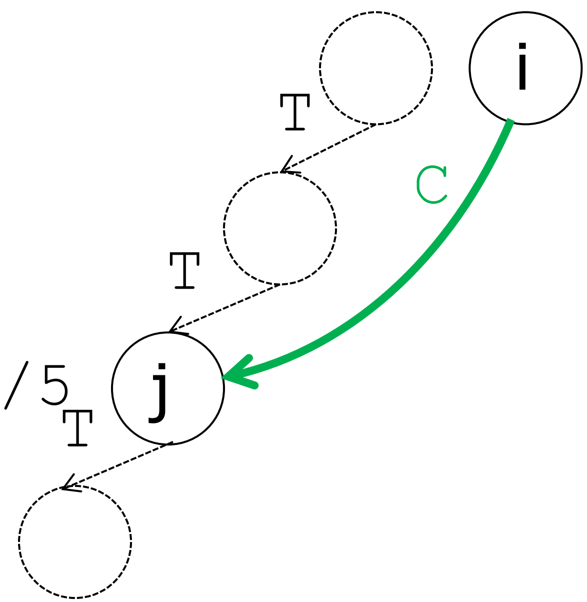
  `pre[i] < pre[j]`



- C: archi rimanenti, per cui
  tempo di scoperta di i > tempo di scoperta di j

  `pre[i] > pre[j]`

`pre[i]/post[i] =8/-1`



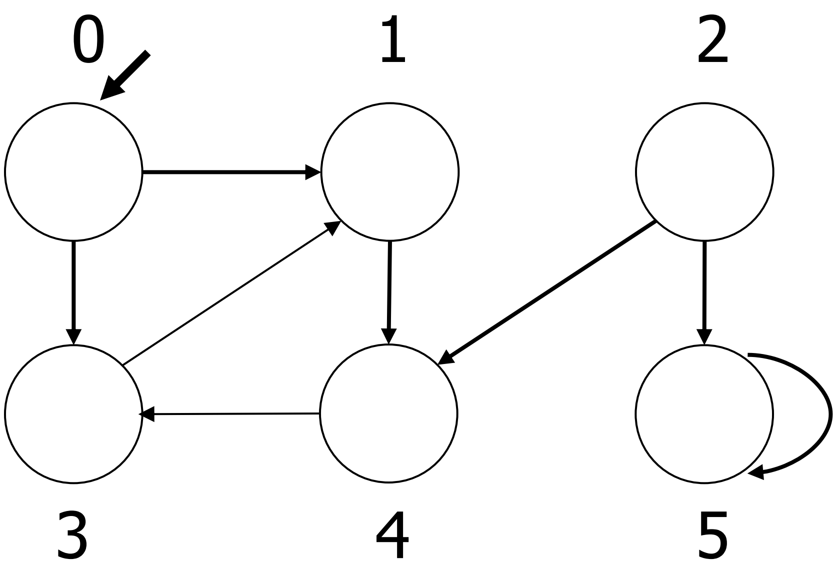
`pre[j]/post[j] =2/5`

Grafo non orientato: solo archi T e B.

Esempio

`time` = -1

`st`

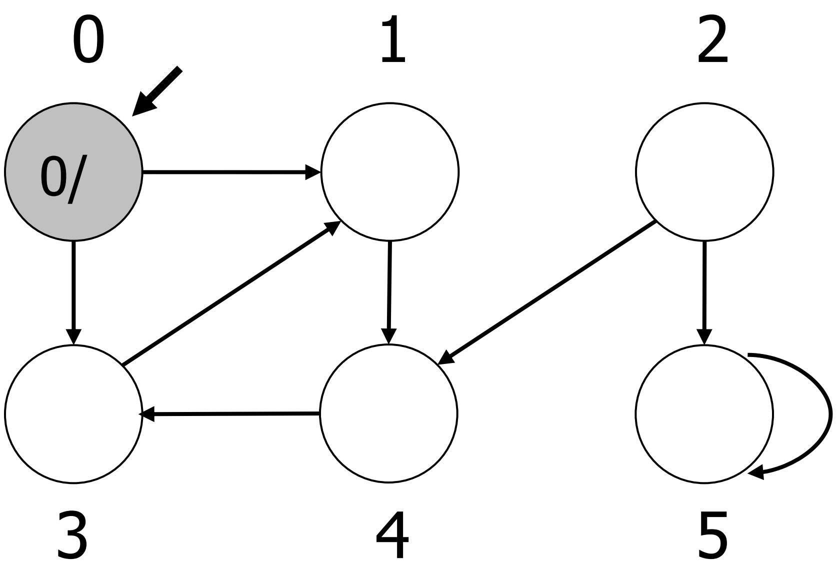


`st`	-1	-1	-1	-1	-1	-1
	0	1	2	3	4	5

time = 0

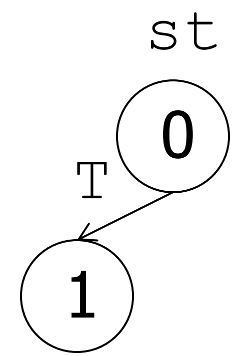
st

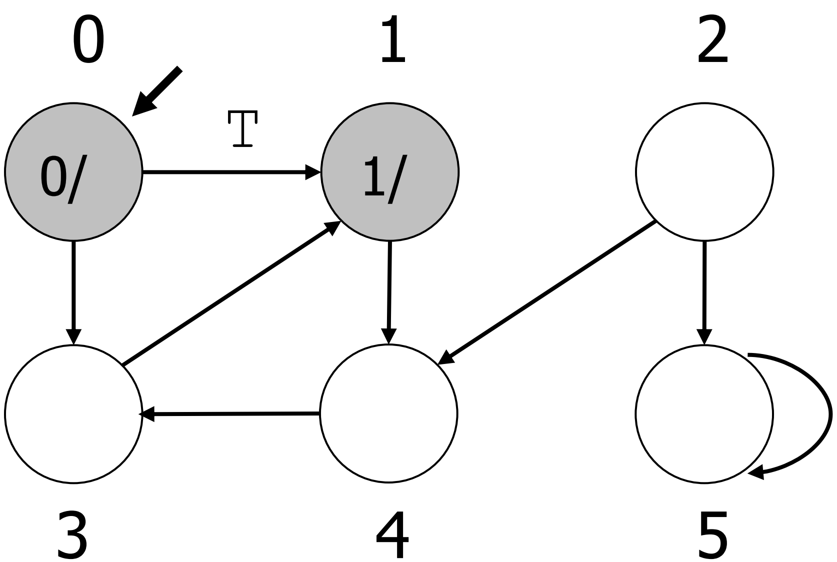
0



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

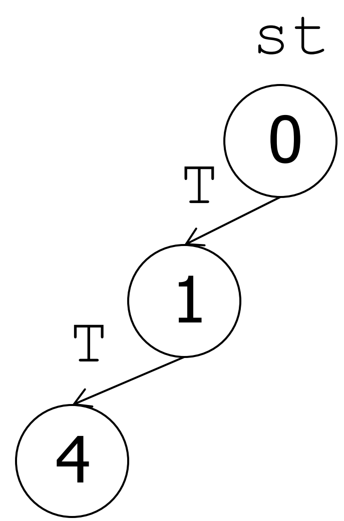
time = 1

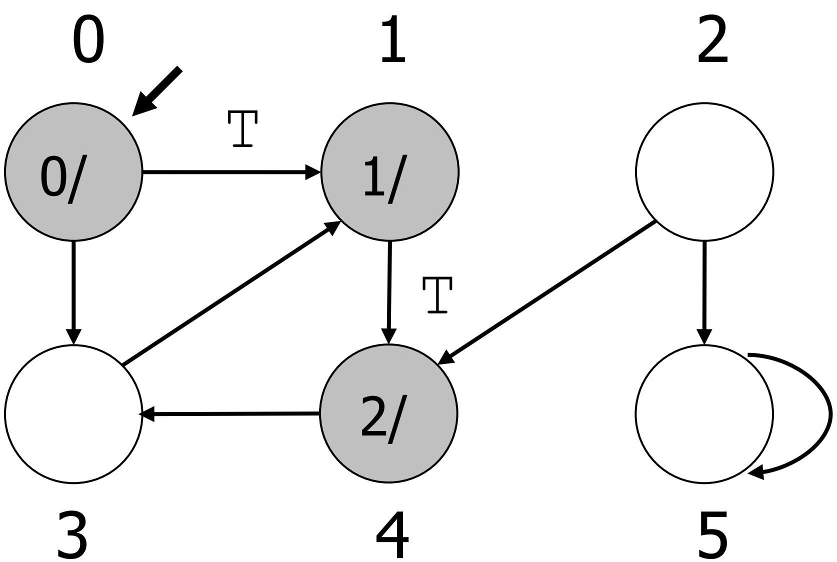




st	0	0	-1	-1	-1	-1
	0	1	2	3	4	5

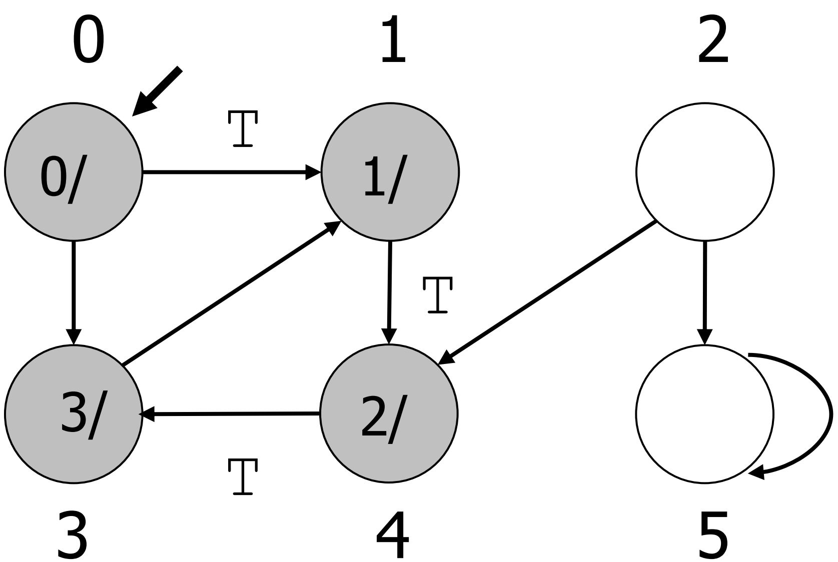
`time` = 2

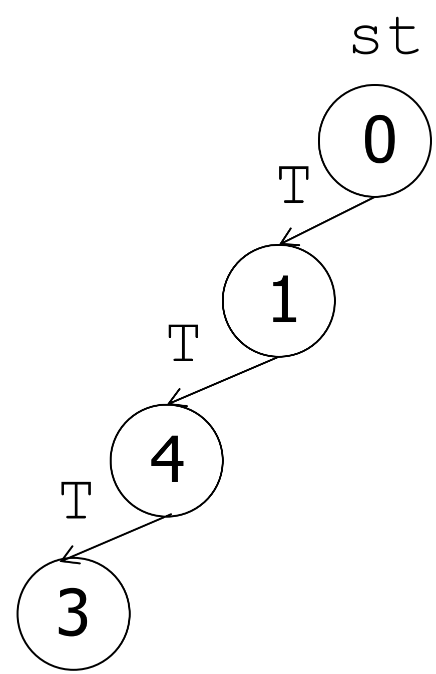




st	0	0	-1	-1	1	-1
	0	1	2	3	4	5

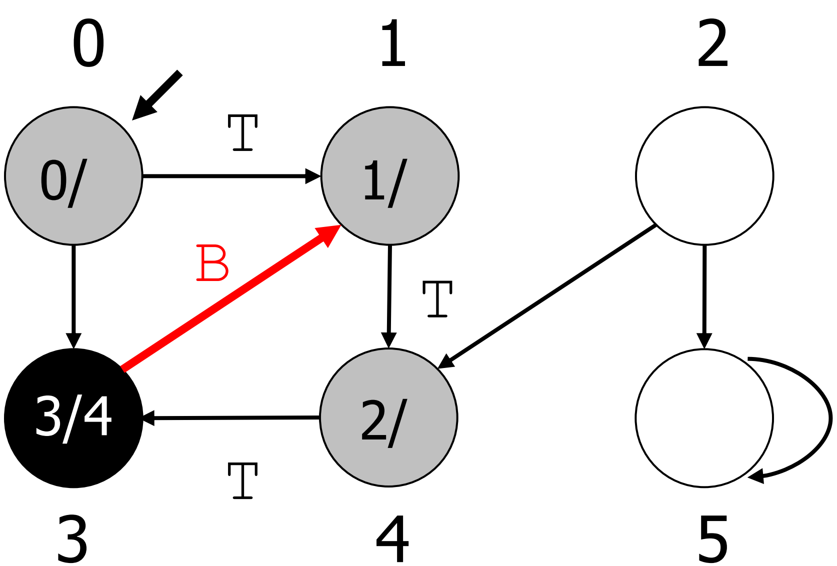
time = 3

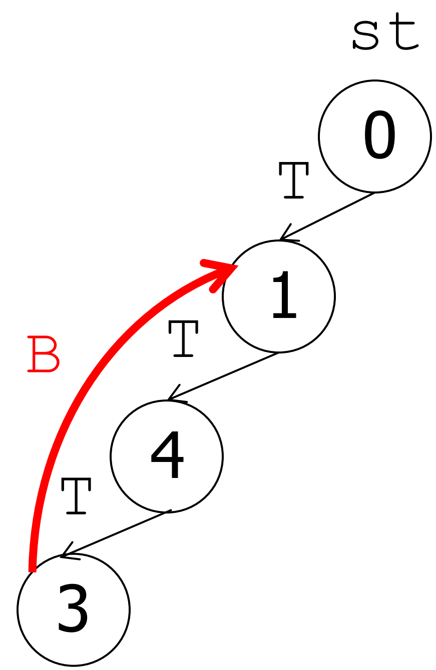




	0	1	2	3	4	5
st	0	0	-1	4	1	-1

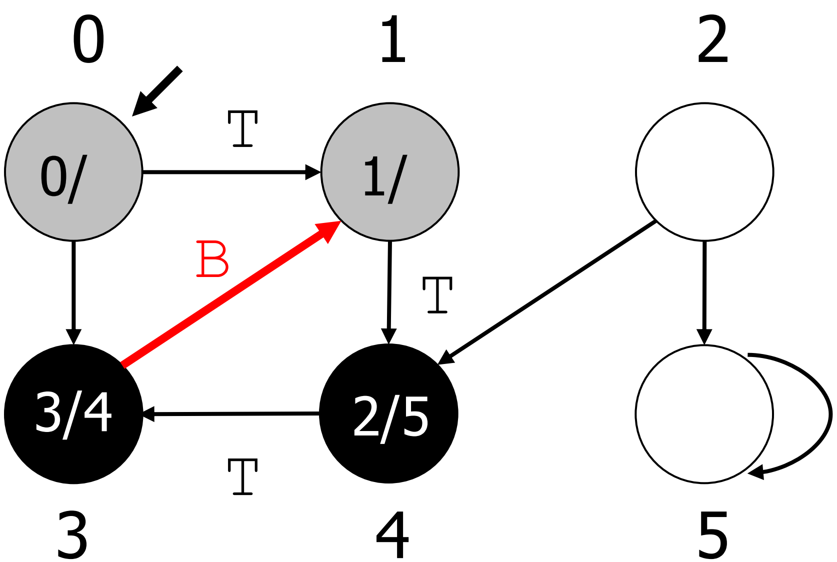
time = 4

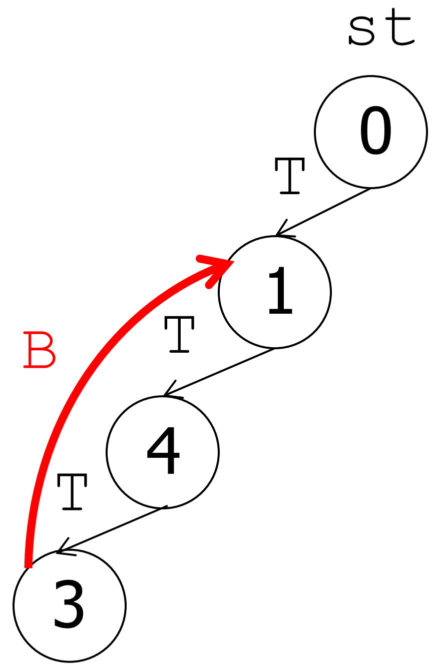




	0	1	2	3	4	5
st	0	0	-1	4	1	-1

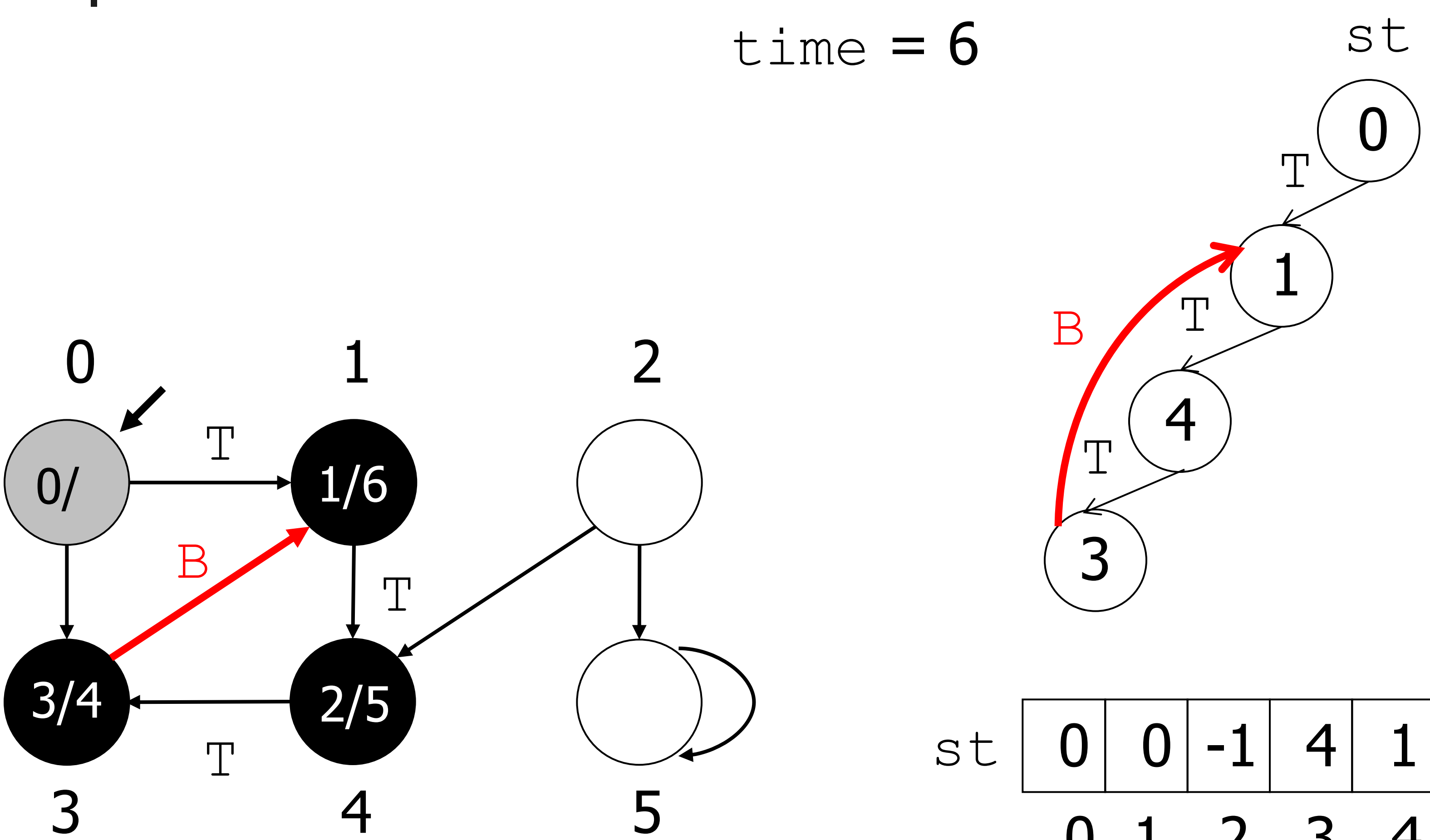
time = 5





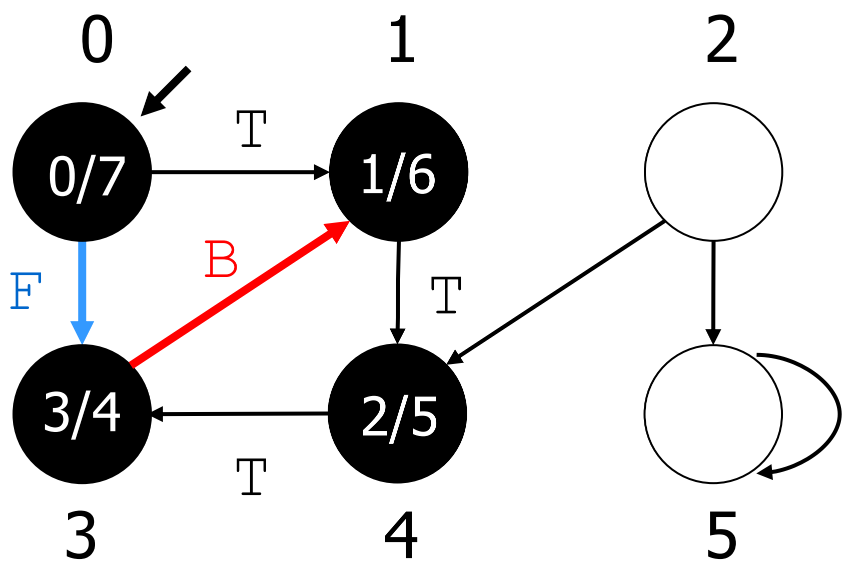
st	0	0	-1	4	1	-1
	0	1	2	3	4	5

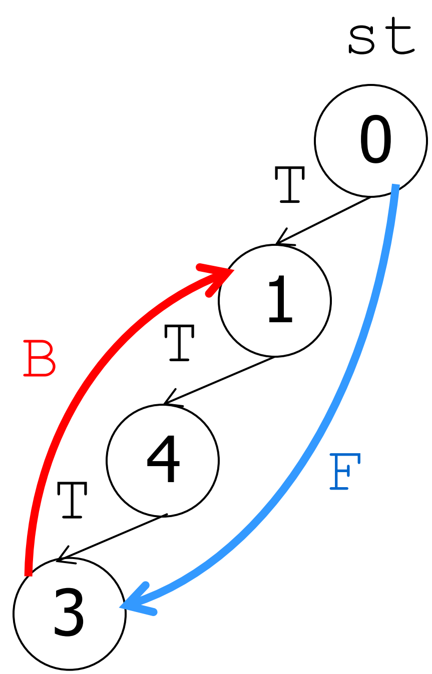
time = 6



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

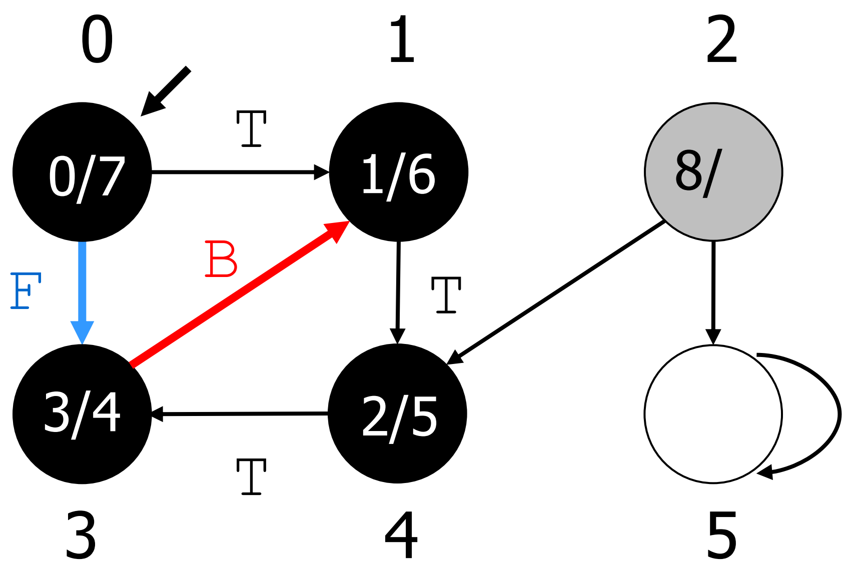
`time = 7`

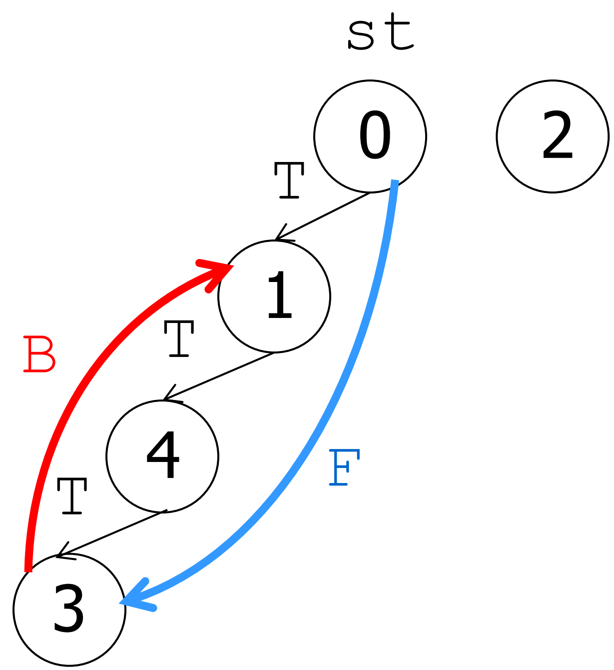




st	0	0	-1	4	1	-1
	0	1	2	3	4	5

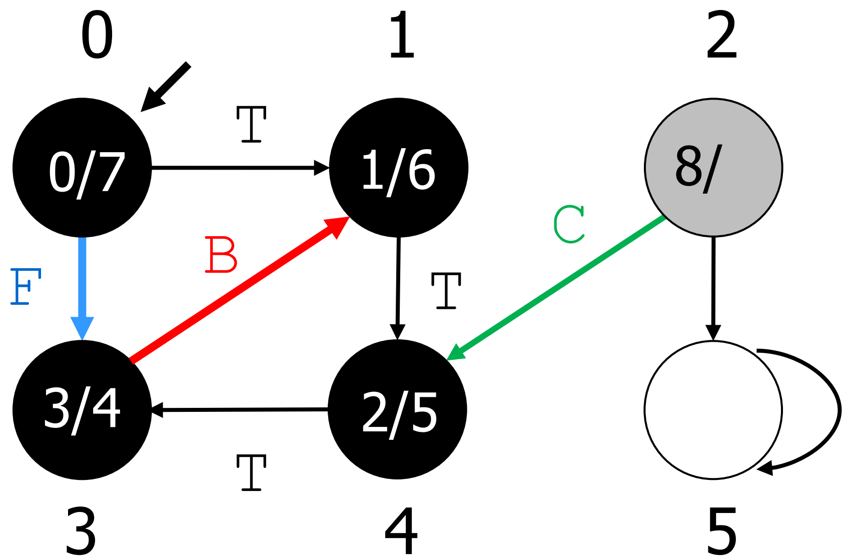
time = 8

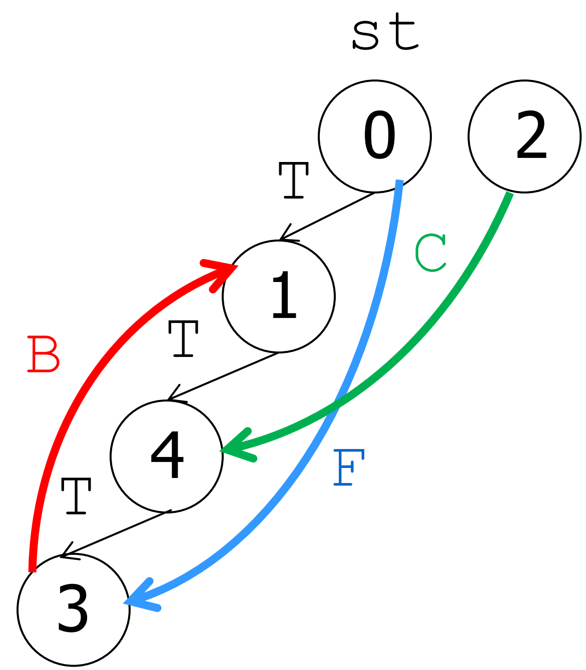




st	0	0	2	4	1	-1
	0	1	2	3	4	5

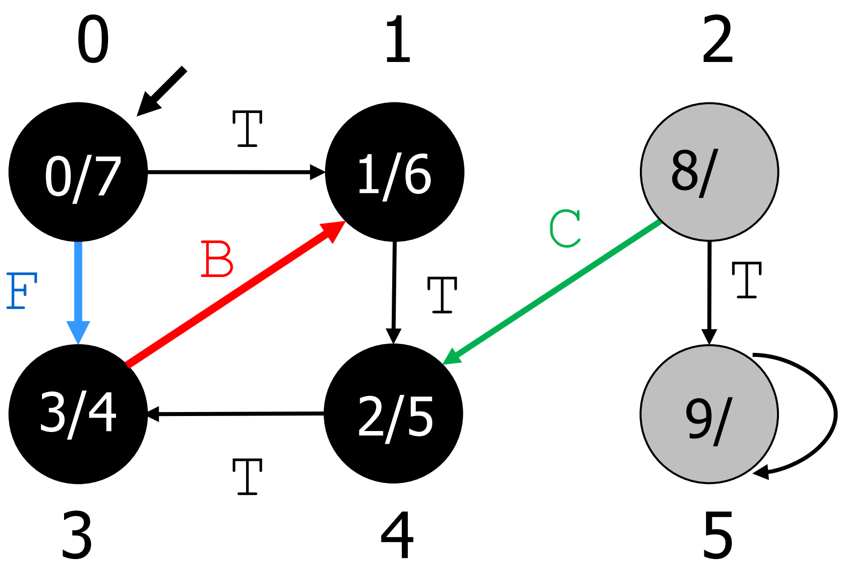
time = 8

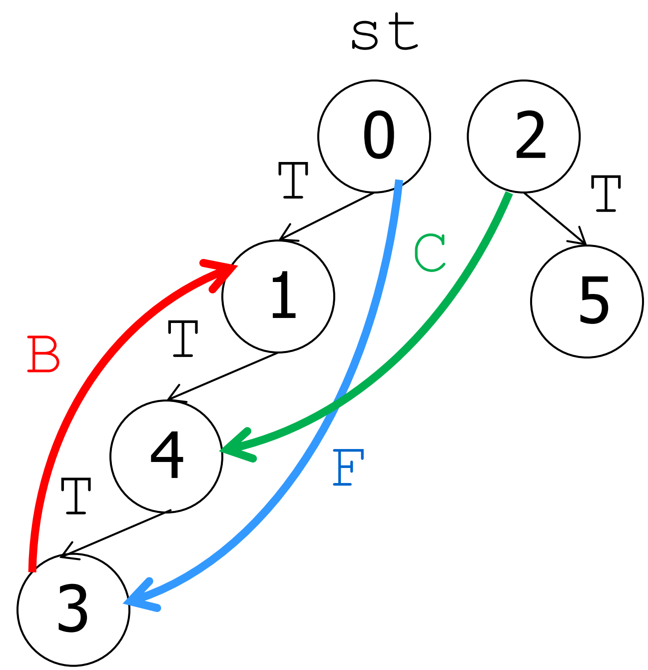




st	0	0	2	4	1	-1
	0	1	2	3	4	5

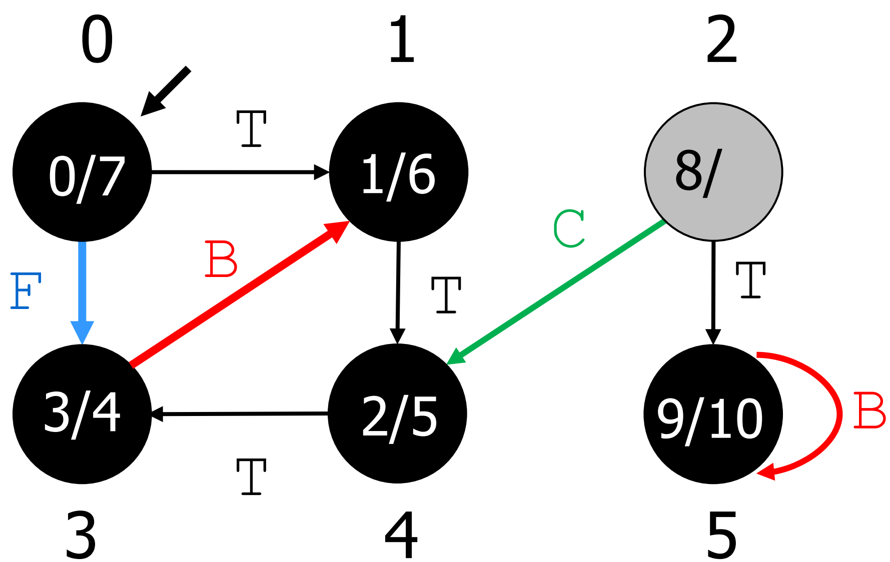
`time = 9`

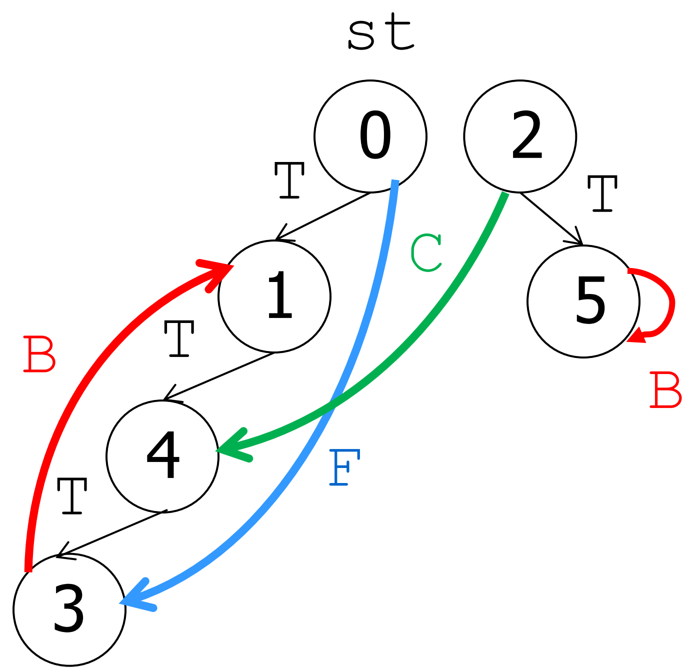




	0	1	2	3	4	5
st	0	0	2	4	1	2

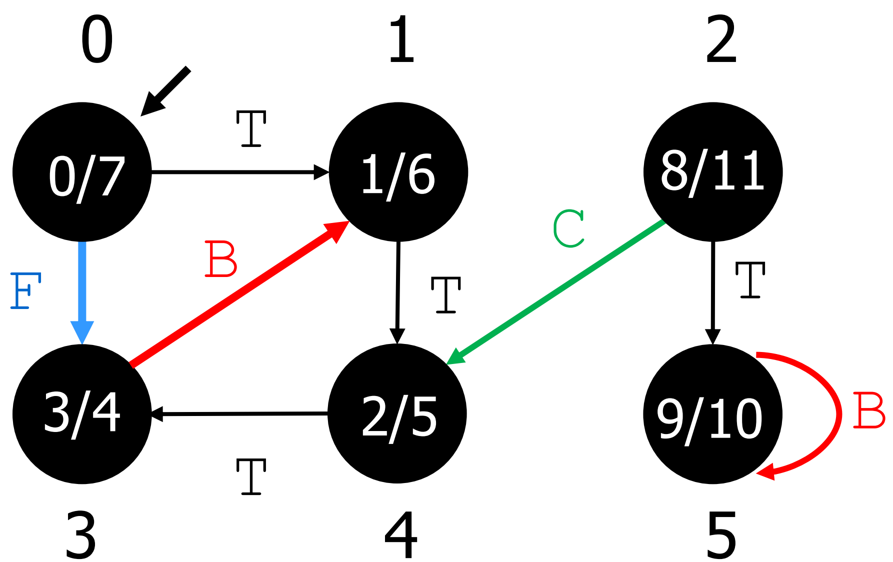
time = 10

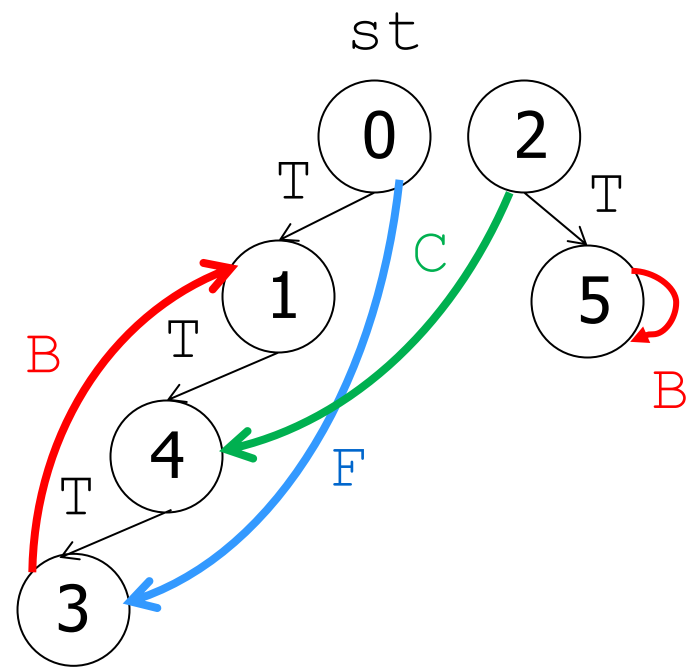




st	0	0	2	4	1	2
	0	1	2	3	4	5

time = 11





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

Algoritmo

- `GRAPHdfs`: procedura che visita tutti i vertici di un grafo, richiamando la procedura ricorsiva `dfsR`. Termina quando tutti i vertici sono neri.
- `dfsR`: procedura che visita in profondità a partire da un vertice `v` identificato fittiziamente come `Edge(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 come lista delle adiacenze
- vettore `pre` dei tempi di scoperta (numerazione in preordine dei vertici)
- vettore `post` dei tempi di completamento (numerazione in postordine dei vertici)
- vettore `st` dei padri per la costruzione della foresta degli alberi di visita in profondità
- contatore `time` per tempi di scoperta/fine elaborazione.

Grafo orientato

```
void dfsR(Graph G, Edge e) {
  link t; int v, w = e.w; Edge x;
  if (e.v != e.w)
    printf("edge (%d, %d) is tree \n", e.v, e.w) ;
  st[e.w] = e.v;
  pre[w] = time++;
  for (t = G->adj[w]; t != NULL; t = t->next)
    if (pre[t->v] == -1)
      dfsR(G, EDGE(w, t->v));
    else {
      v = t->v;
      x = EDGE(w, v);
      if (post[v] == -1)
        printf("edge (%d, %d) is back \n", x.v, x.w);
      else
        if(pre[v]>pre[w])
          printf("edge (%d, %d) is forward \n", x.v, x.w);
        else
          printf("edge (%d, %d) is cross \n", x.v, x.w);
    }
  post[w] = time++;
}
```

Grafo non orientato

```
void dfsR(Graph G, Edge e) {
  link t; int v, w = e.w; Edge x;
  if (e.v != e.w)
    printf("edge (%d, %d) is tree \n", e.v, e.w) ;
  st[e.w] = e.v;
  pre[w] = time++;
  for (t = G->adj[w]; t != NULL; t = t->next)
    if (pre[t->v] == -1)
      dfsR(G, EDGE(w, t->v));
    else {
      v = t->v;
      x = EDGE(w, v);
      if (pre[w] < pre[v])
        printf("edge (%d, %d) is back \n", x.v, x.w) ;
    }
  post[w] = time++;
}
```

```
void GRAPHdfs(Graph G) {
  int v;
  time = 0;
  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, EDGE(v,v));
  printf("discovery/endprocessing time labels n");
  for (v=0; v < G->V; v++)
    printf("vertex %d : %d/%d \n", v, pre[v], post[v]);
  printf("resulting DFS tree \n");
    for (v=0; v < G->V; v++)
      printf("parent of vertex %d is vertex %d\n“,v,st[v]);
}
```

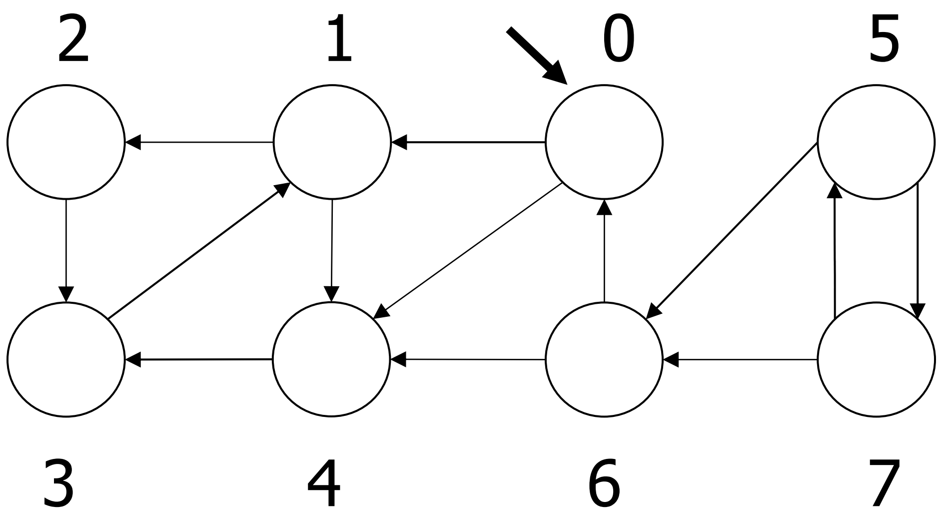
Complessità (lista adiacenze)

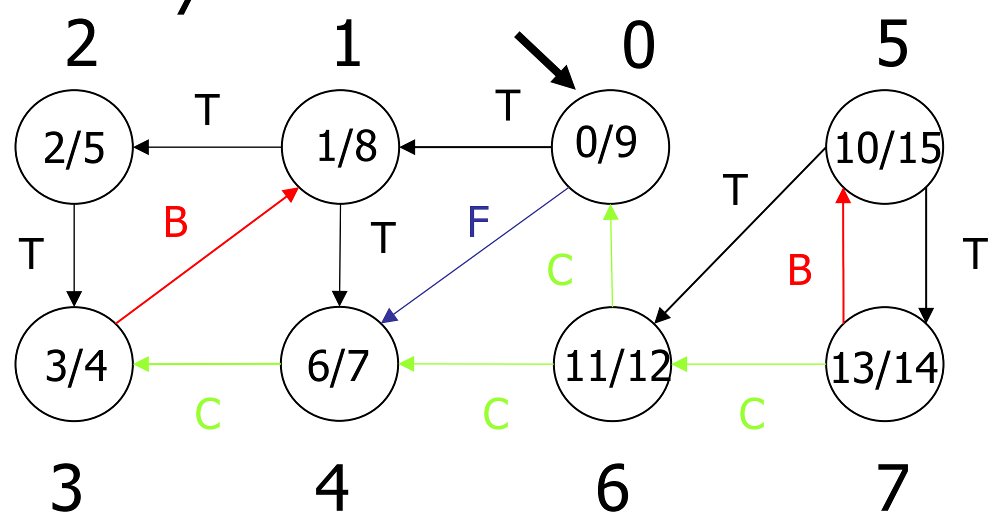
$\Theta(|V|)$

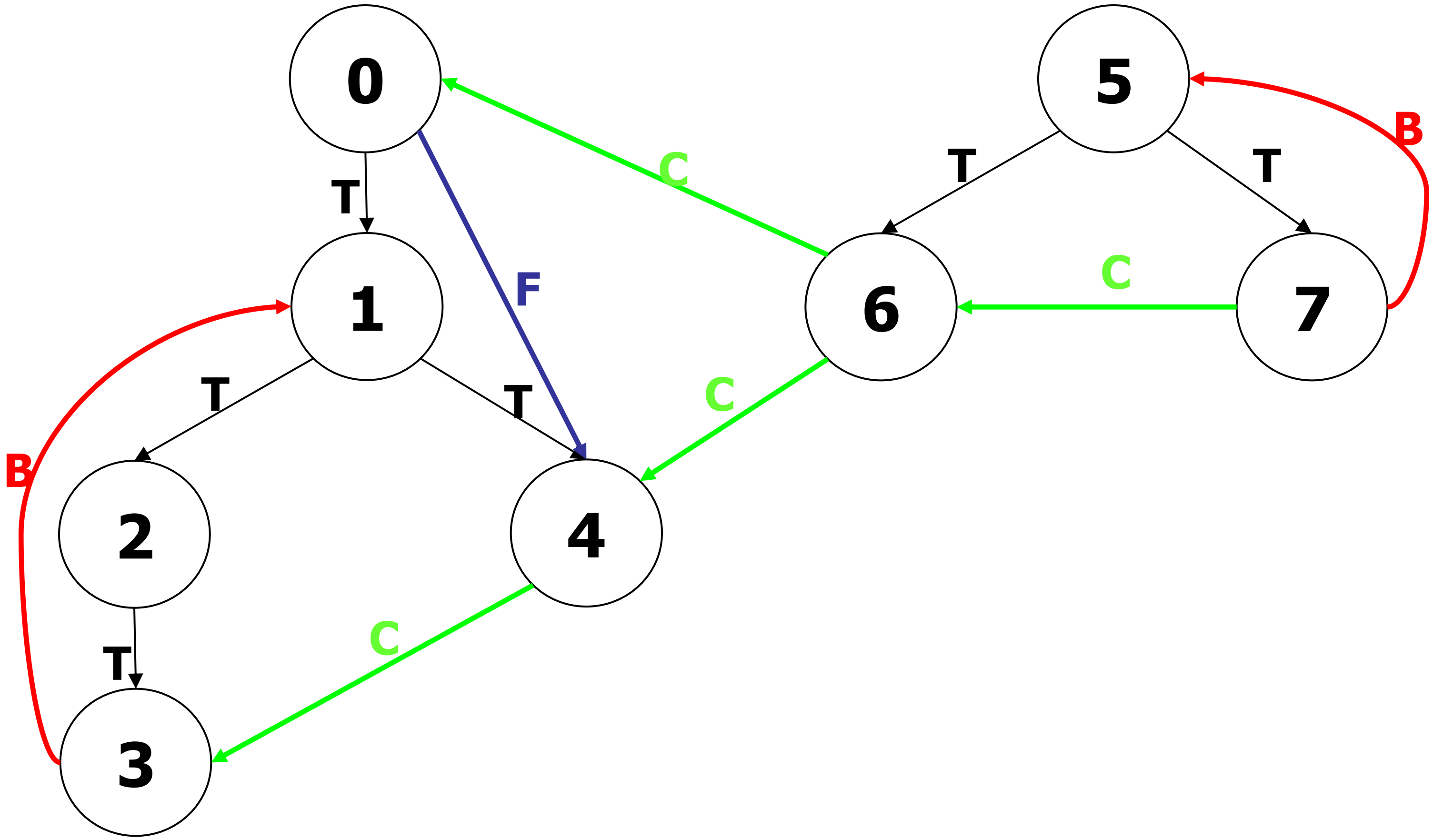
$\Theta(|E|)$

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

Esempio

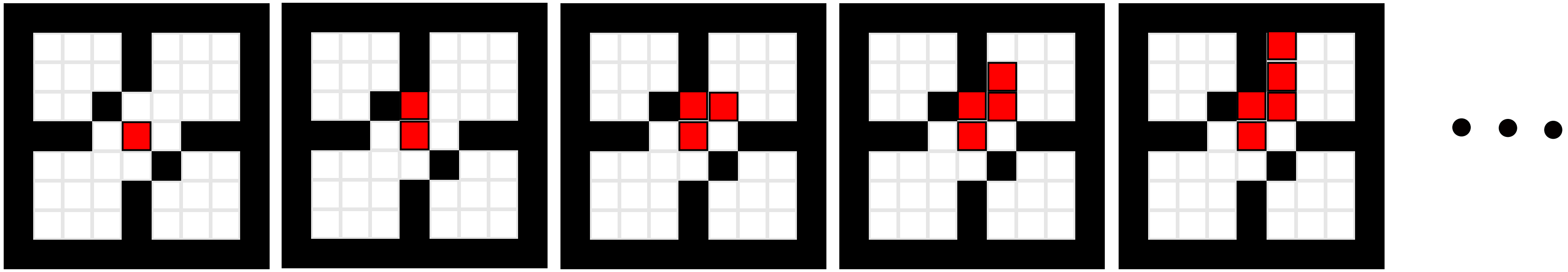

2
1
0
5
3
4
6
7


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


0
5
T
C
B
T
T
1
F
6
C
7
T
T
C
B
2
4
C
T
3

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.

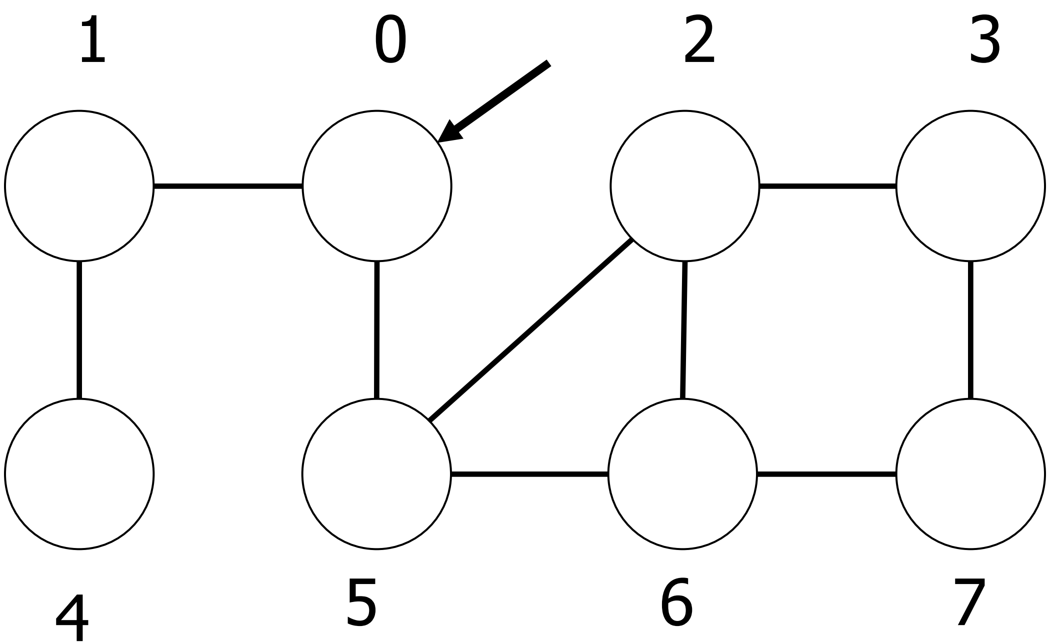
Dato vertice u:

- `st[u]`: padre di u nell'albero della visita.

Esempio

Q

st

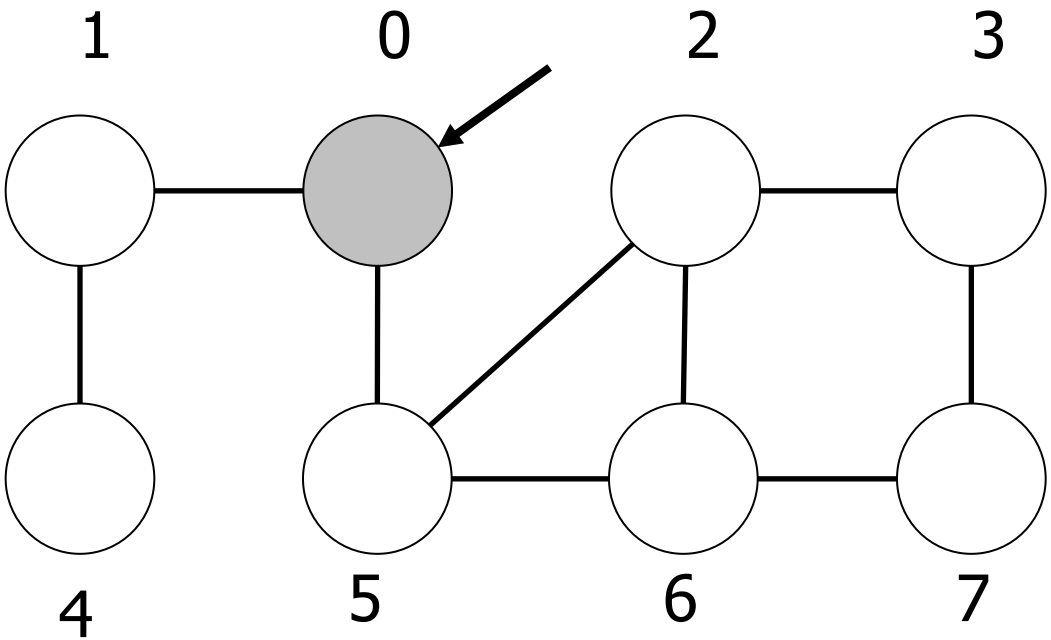


st

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

Q 0

st 0



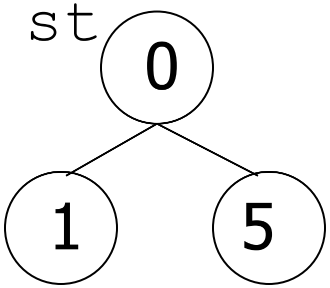
st

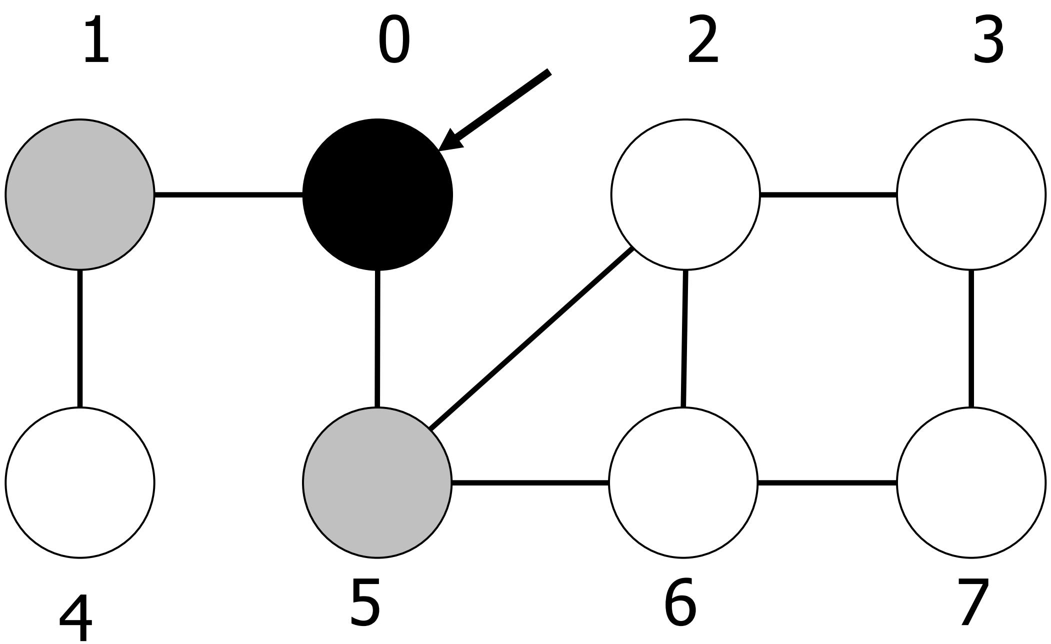
0	1	2	3	4	5	6	7
0	-1	-1	-1	-1	-1	-1	-1

Q

1	5

st



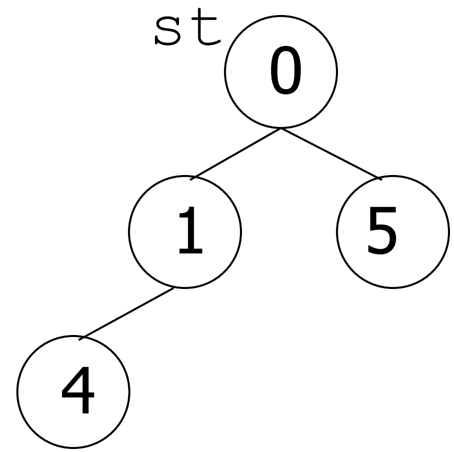


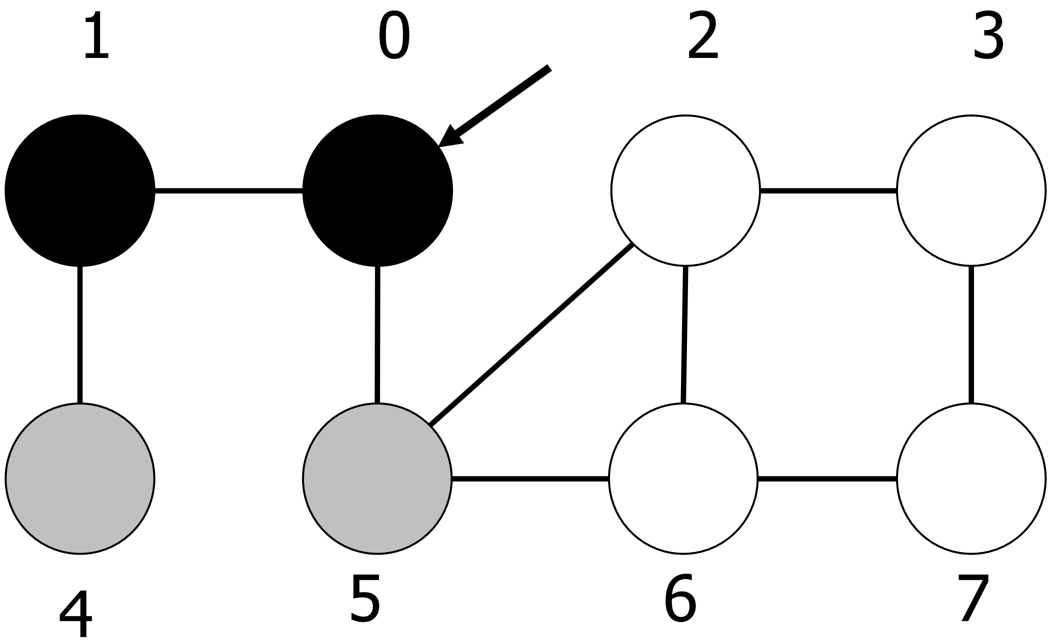
st

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

Q

5	4





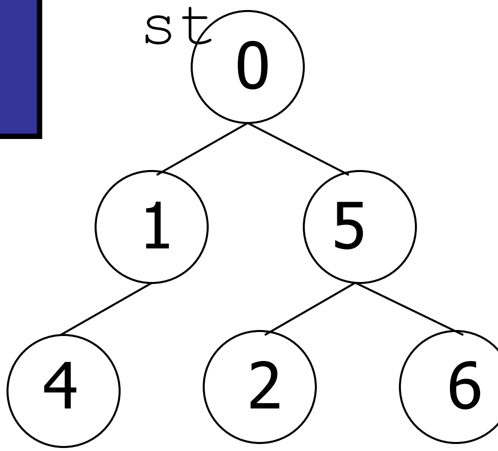
st

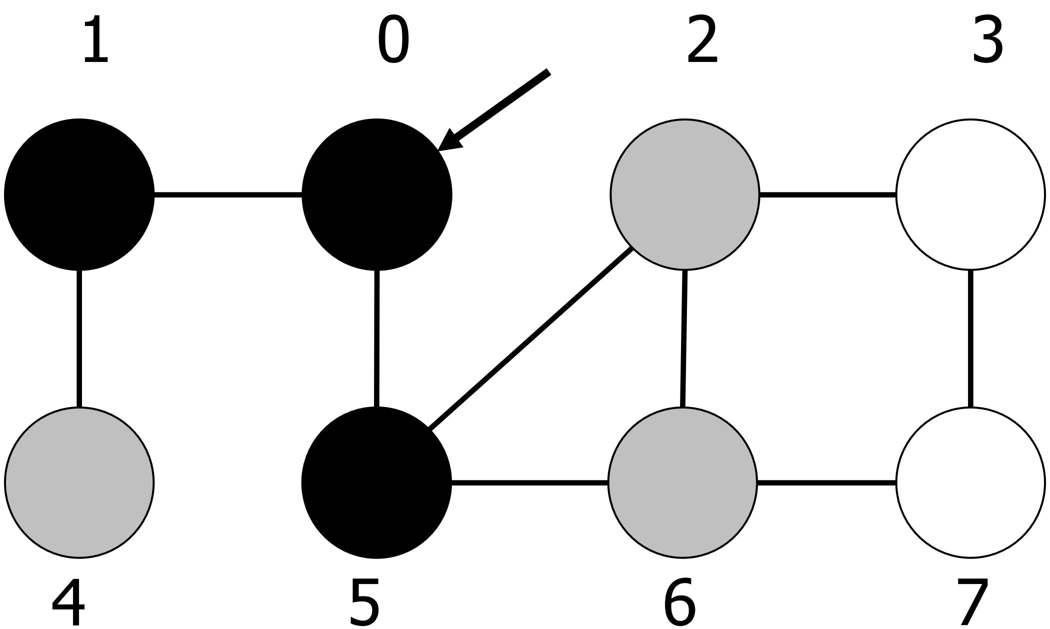
0	0	-1	-1	1	0	-1	-1
0	1	2	3	4	5	6	7

Q

4	2	6

st



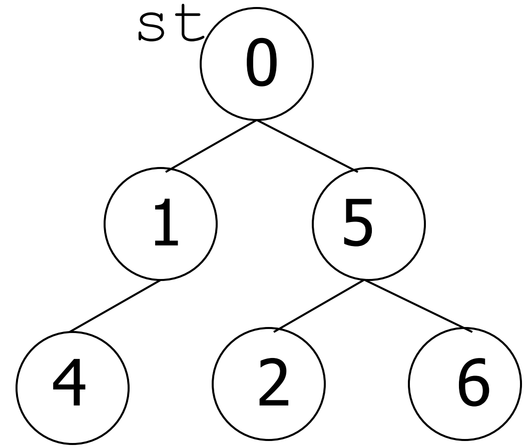


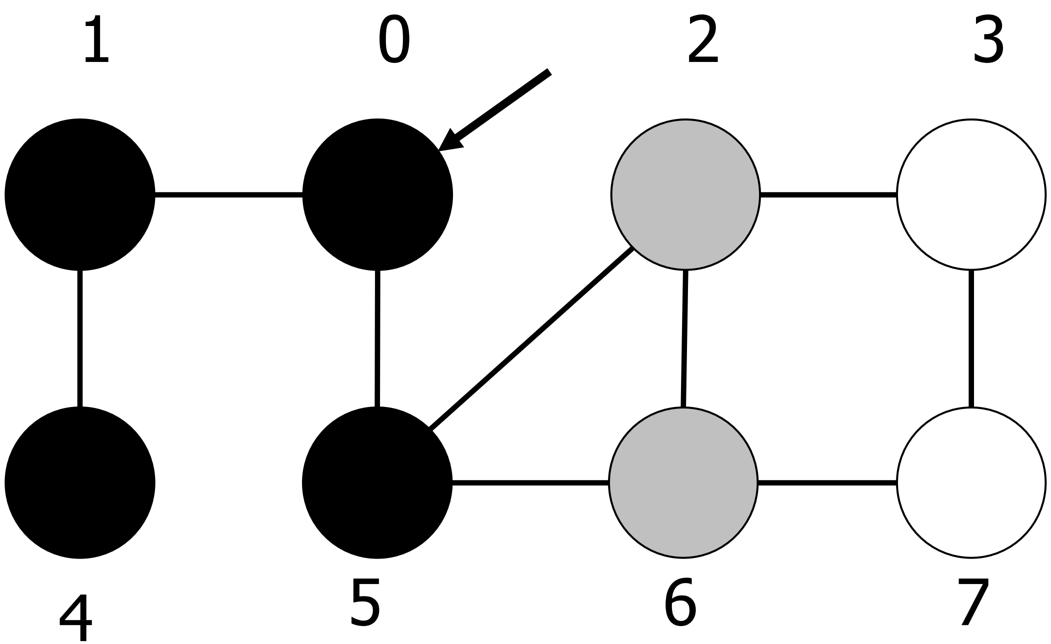
st

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

Q		
	2	6

st



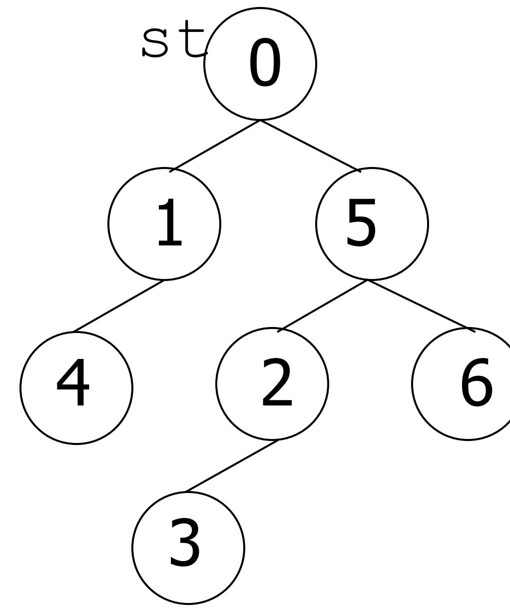


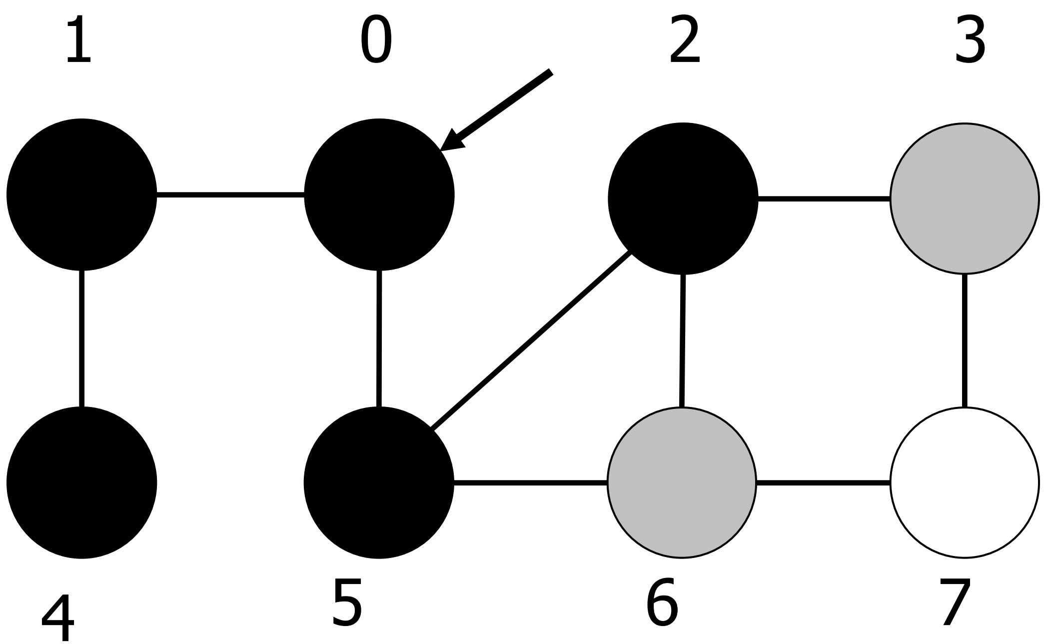
st

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

Q

6	3



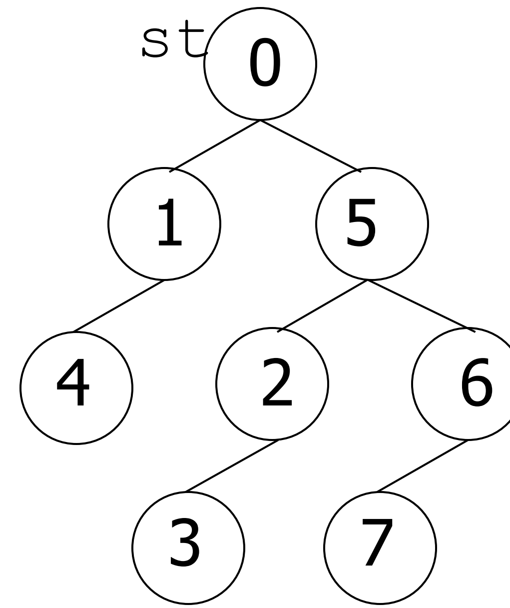


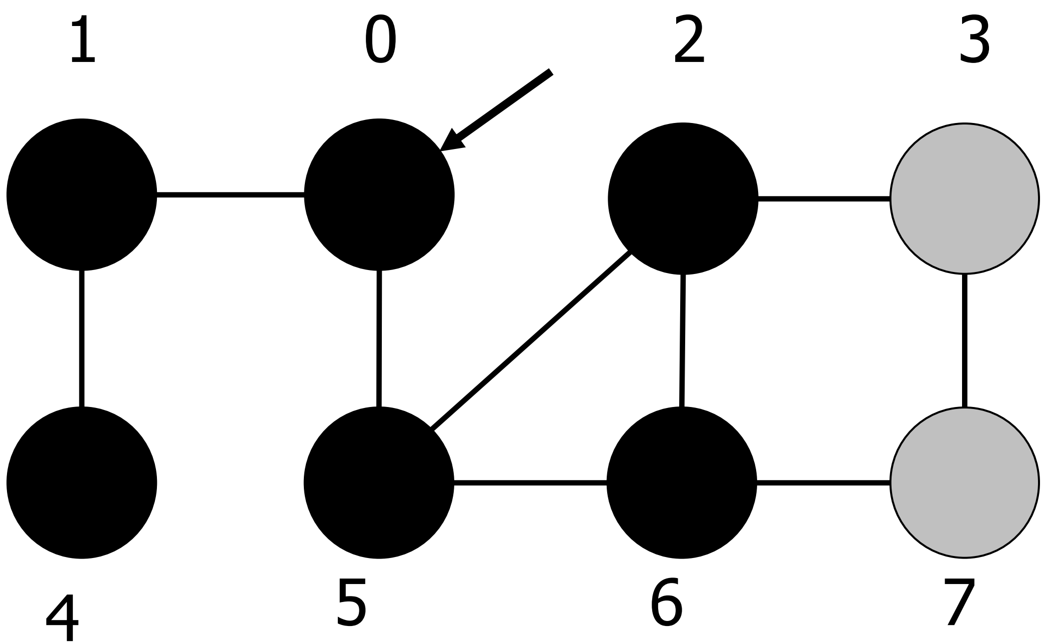
st

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

Q

3	7

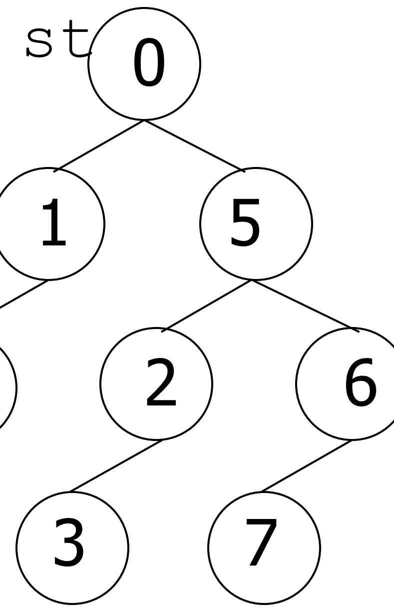


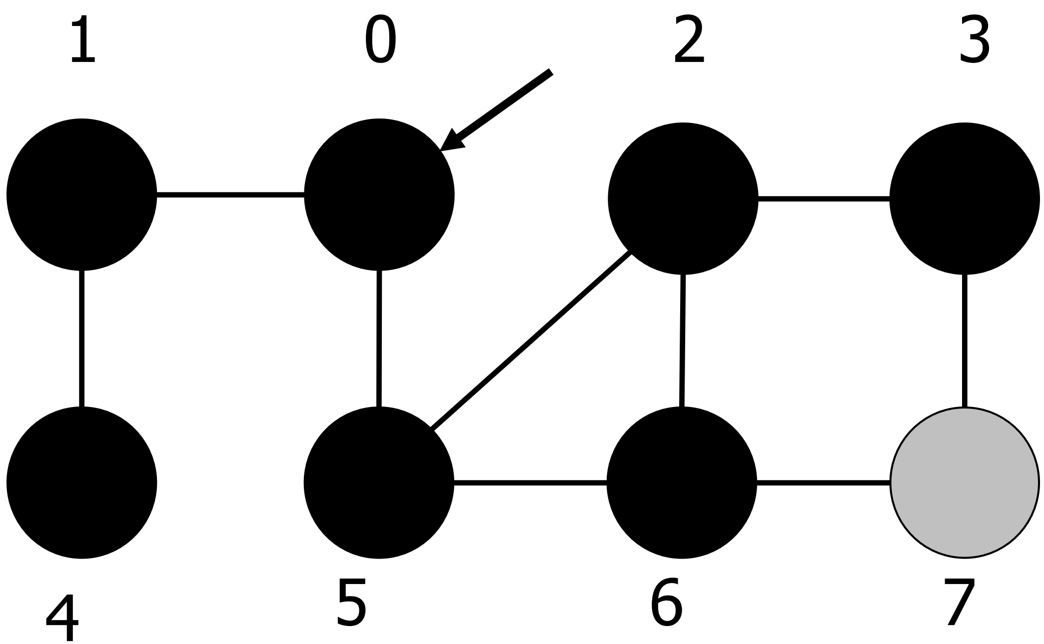


st

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

Q 7

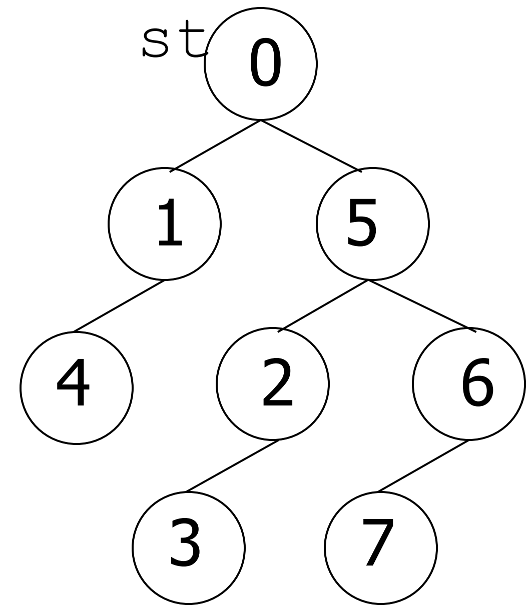


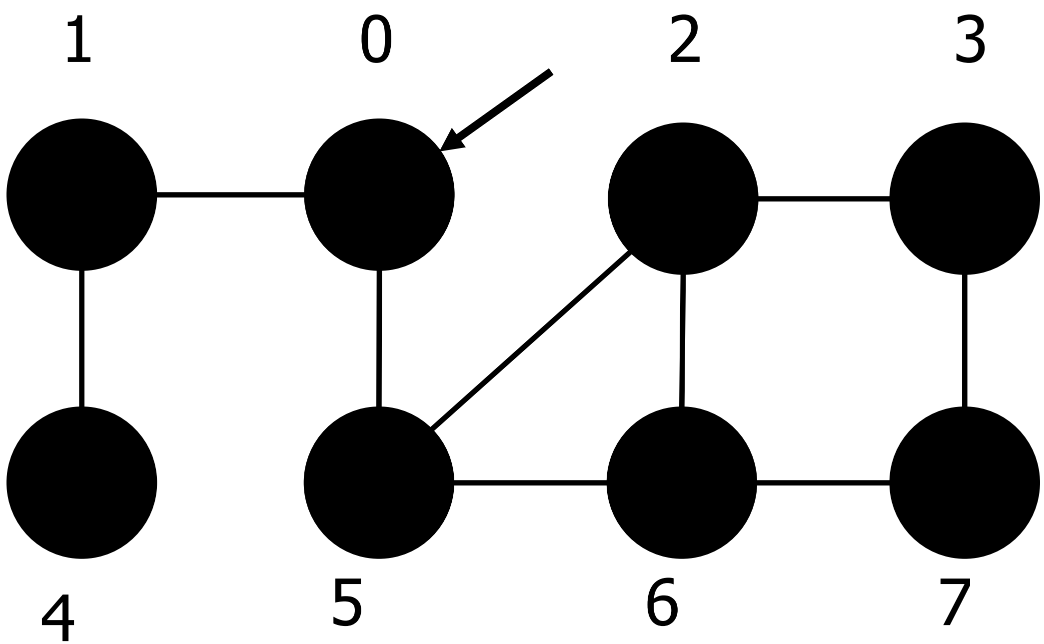


st

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

Strutture dati

Strutture dati:

- grafo come matrice delle adiacenze
- coda `Q` dei vertici grigi
- vettore `st` dei padri nell'albero di visita in ampiezza
- vettore `pre` dei tempi di scoperta dei vertici
- contatore `time` del tempo.

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

```
void bfs(Graph G, Edge e)
{
  int v, w;
  Q q = QUEUEinit();
  QUEUEput(q, e);
  while (!QUEUEempty(q))
    if (pre[(e = QUEUEget(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)
            QUEUEput(q,EDGE(e.w, v));
      }
}
```

```
void GRAPHbfs(Graph G) {
  int v;
  time = 0;
  for (v=0; v < G->V; v++) {
    pre[v] = -1;
    st[v] = -1;
  }

  bfs(G, EDGE(0,0));

  printf("Resulting BFS tree \n");
  for (v=0; v < G->V; v++)
    printf("parent %d is %d\n", v, st[v]);

}
```

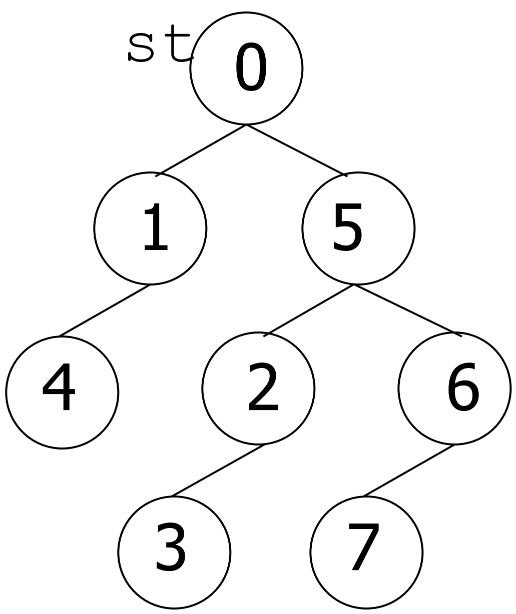
Complessità

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

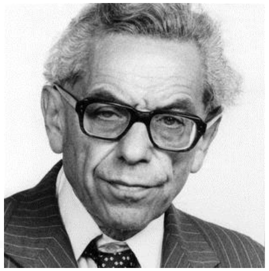
Proprietà

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

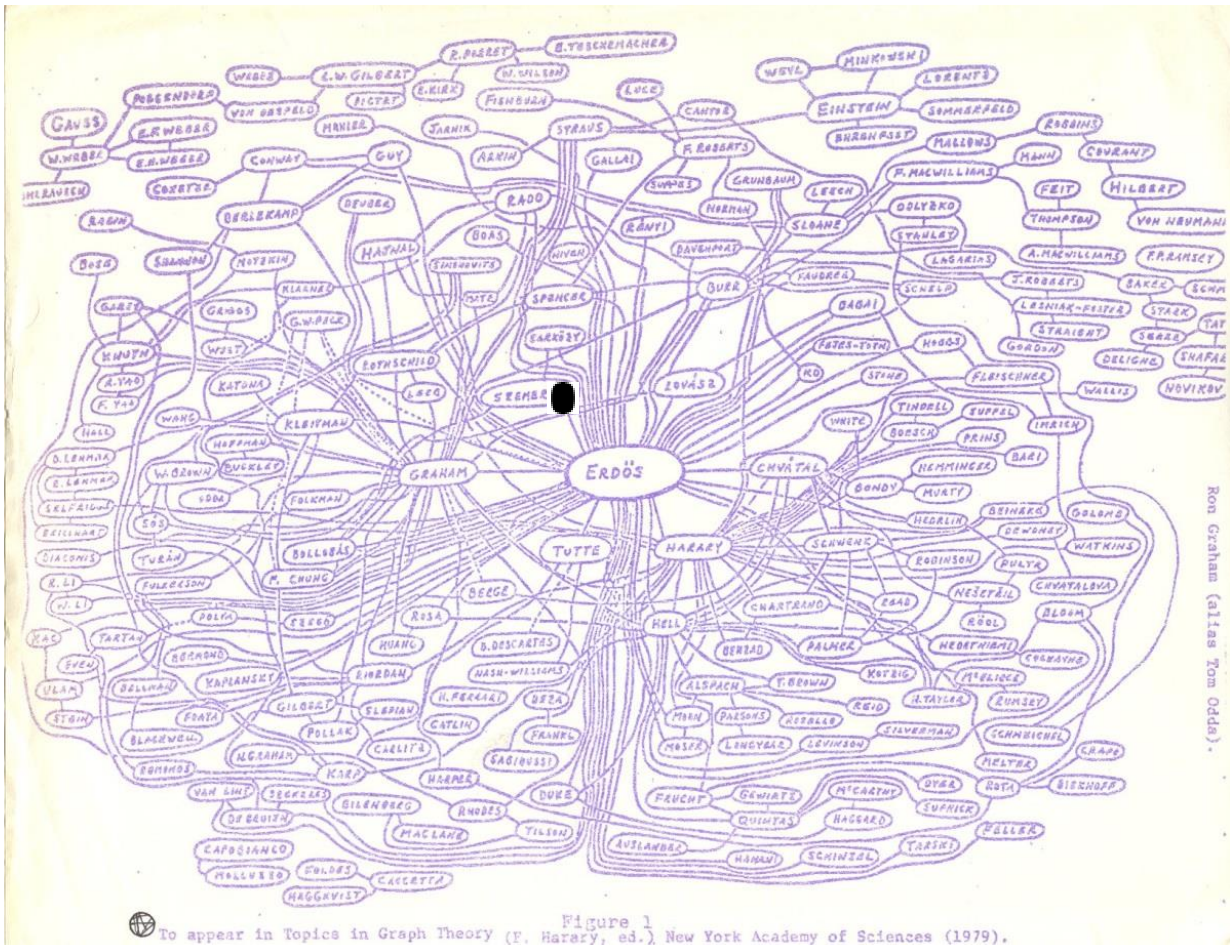
Cammino minimo da 0 a 3:
0, 5, 2, 3
lunghezza = 3



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

Caro Babbo Natale,

per questo Natale non siamo interessati a regali materiali, ma ti chiediamo qualcosa di veramente importante:

come prima cosa ti chiediamo di farci superare (con un bel voto) l'esame più temuto ALGORITMI E PROGRAMMAZIONE!

Nel tempo che ti resta, se fosse possibile, vorremmo superare anche tutto il resto:

ELETTROTECNICA;

ANALISI II,

FISICA II.

E poiché quest'anno siamo stati tanto buoni, ci aspettiamo che le nostre richieste siano esaudite.

Con tanto

…portante:

Come prima cosa ti chiediamo di farci superare (con un bel voto) l'esame più temuto ALGORITMI E PROGRAMMAZIONE!

Nel tempo che ti resta, se fosse possibile, vorremmo superare anche tutto il resto:

ELETTROTECNICA;

ANALISI II;

FISICA II.

E poiché quest'anno siamo stati tanto buoni, ci aspettiamo che le nostre richieste siano esaudite.

Con tanto affetto e speranza,

[redacted]

Studia [redacted]