Le applicazioni degli algoritmi di visita dei grafi

Gianpiero Cabodi e Paolo Camurati

Dip. Automatica e Informatica

Politecnico di Torino

Rilevazione di cicli

Un grafo è aciclico se e solo se in una visita in profondità non si incontrano archi etichettati B.

Componenti connesse

In un grafo non orientato rappresentato come lista delle adiacenze:

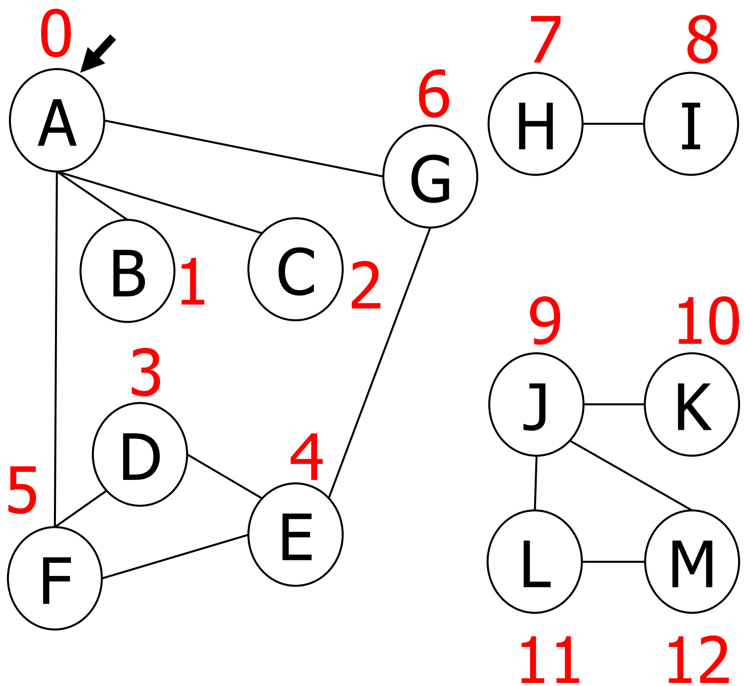
- ogni albero della foresta della DFS è una componente connessa
- `cc[v]` è un array locale a `GRAPHcc` che memorizza un intero che identifica ciascuna componente connessa. I vertici fungono da indici dell'array

wrapper

Esempio

in.txt

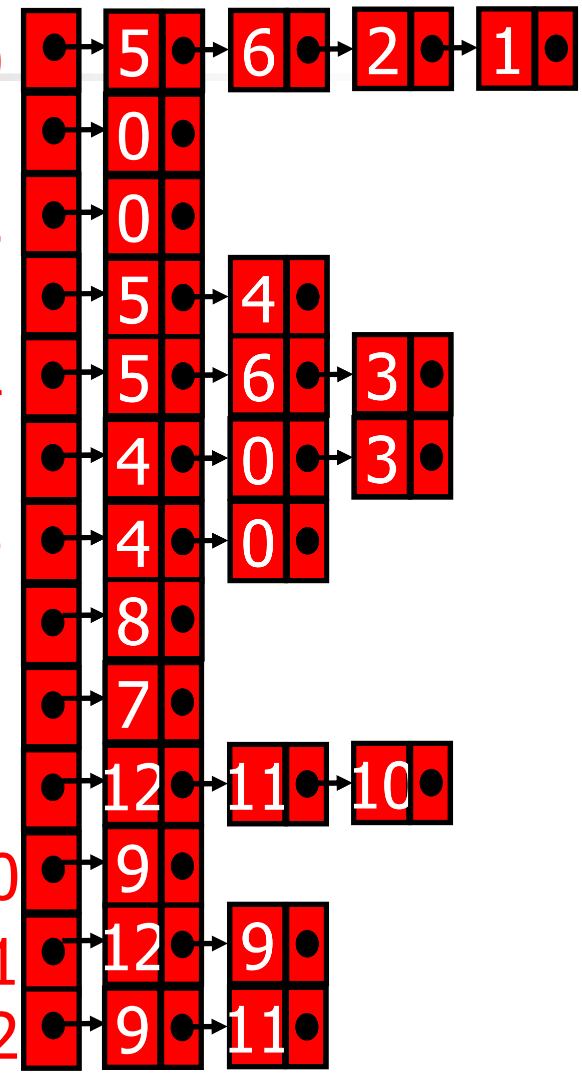
A B
A C
D E
D F
A G
H I
J K
J L
L M
G E
A F
F E
J M



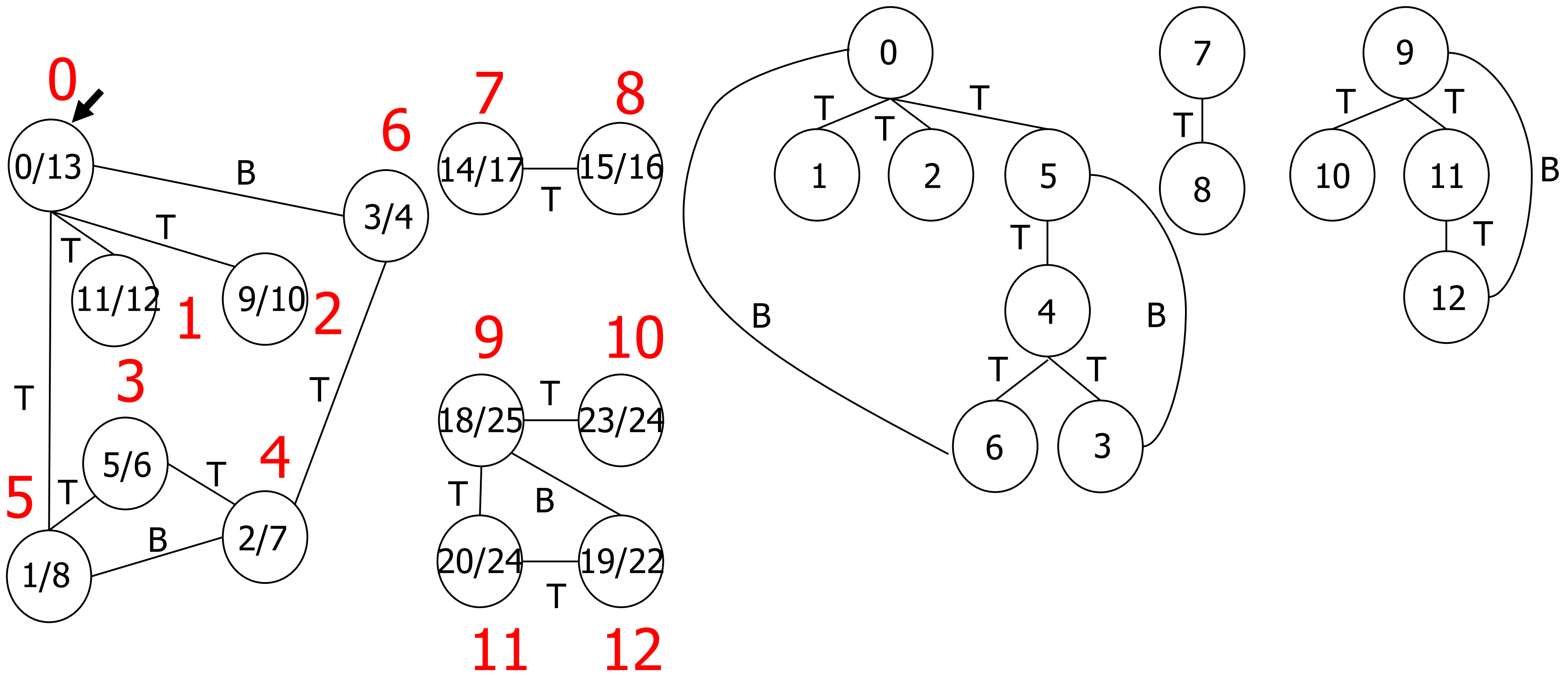
ST

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J
10	K
11	L
12	M

Lista delle adiacenze



0	5	6	2	1
1	0			
2	0			
3	5	4		
4	5	6	3	
5	4	0	3	
6	4	0		
7	8			
8	7			
9	12	11	10	
10	9			
11	12	9		
12	9	11		



	0	1	2	3	4	5	6	7	8	9	10	11	12
cc	0	0	0	0	0	0	0	1	1	2	2	2	2

```c
void dfsRcc(Graph G, int v, int id, int *cc) {
  link t;
  cc[v] = id;
  for (t = G->adj[v]; t != G->z; t = t->next)
    if (cc[t->v] == -1)
      dfsRcc(G, t->v, id, cc);
}
int GRAPHcc(Graph G) {
  int v, id = 0, *cc;
  cc = malloc(G->V * sizeof(int));
  for (v = 0; v < G->V; v++) cc[v] = -1;
  for (v = 0; v < G->V; v++)
    if (cc[v] == -1) dfsRcc(G, v, id++, cc);
  printf("Connected component(s) \n");
  for (v = 0; v < G->V; v++)
    printf("node %s in cc %d\n",STretrieve(G->tab,v),cc[v]);
  return id;
}
```

Connettività

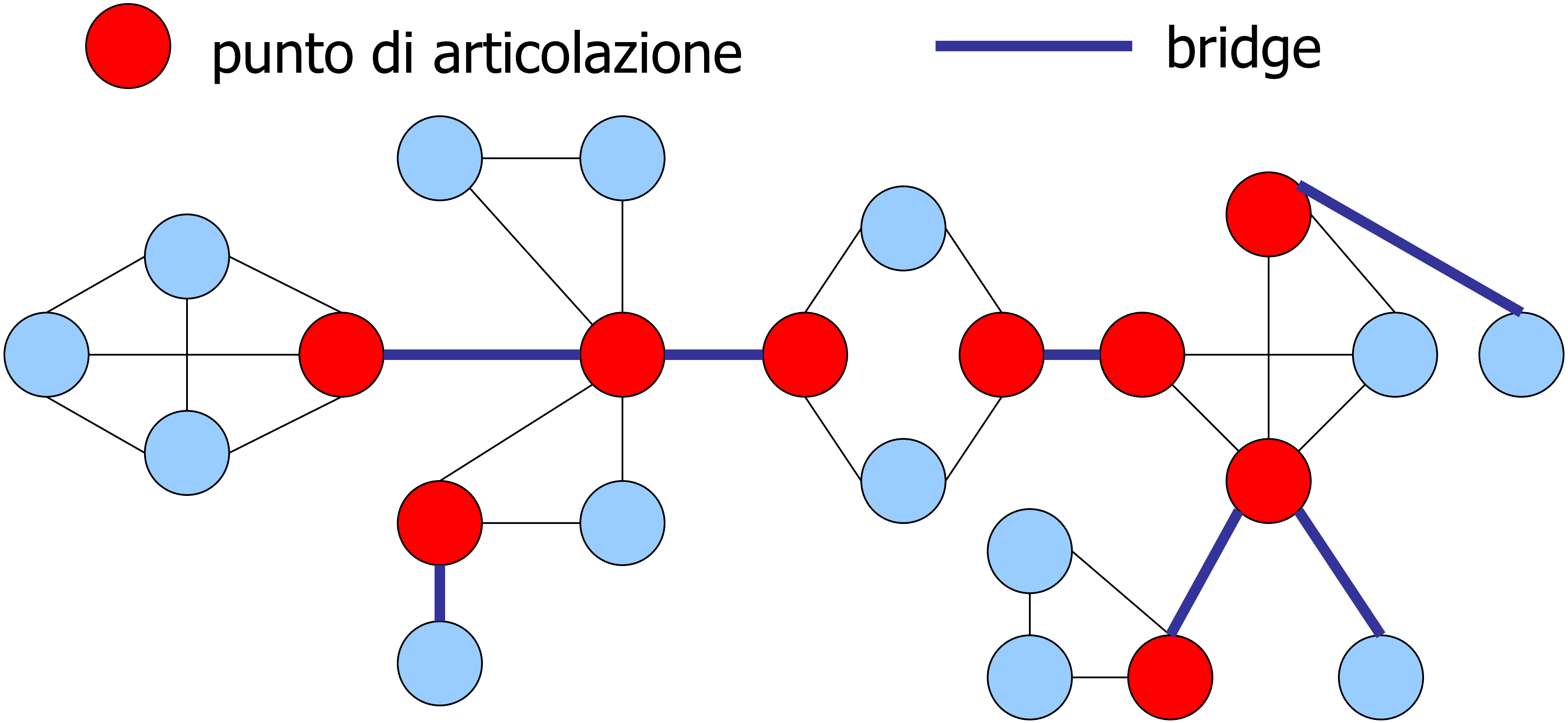
Dato un grafo non orientato e connesso, determinare se perde la proprietà di connessione a seguito della rimozione di:

- un arco
- un nodo.

Ponte (bridge): arco la cui rimozione disconnette il grafo.

Punto di articolazione: vertice la cui rimozione disconnette il grafo. Rimuovendo il vertice si rimuovono anche gli archi su di esso insistenti.

Esempio



Punto di articolazione

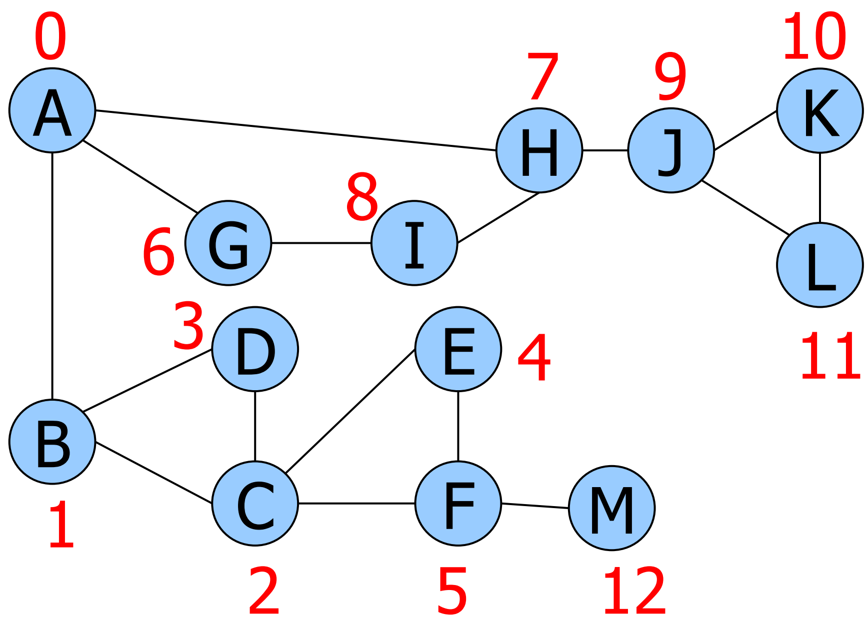
Dato un grafo non orientato G, dato l'albero G_π della visita in profondità,

- la radice di G_π è un punto di articolazione di G se e solo se ha almeno due figli
- ogni altro vertice v è un punto di articolazione di G se e solo se v ha un figlio s tale che non vi è alcun arco B da s o da un suo discendente a un antenato proprio di v.

Esempio

in.txt

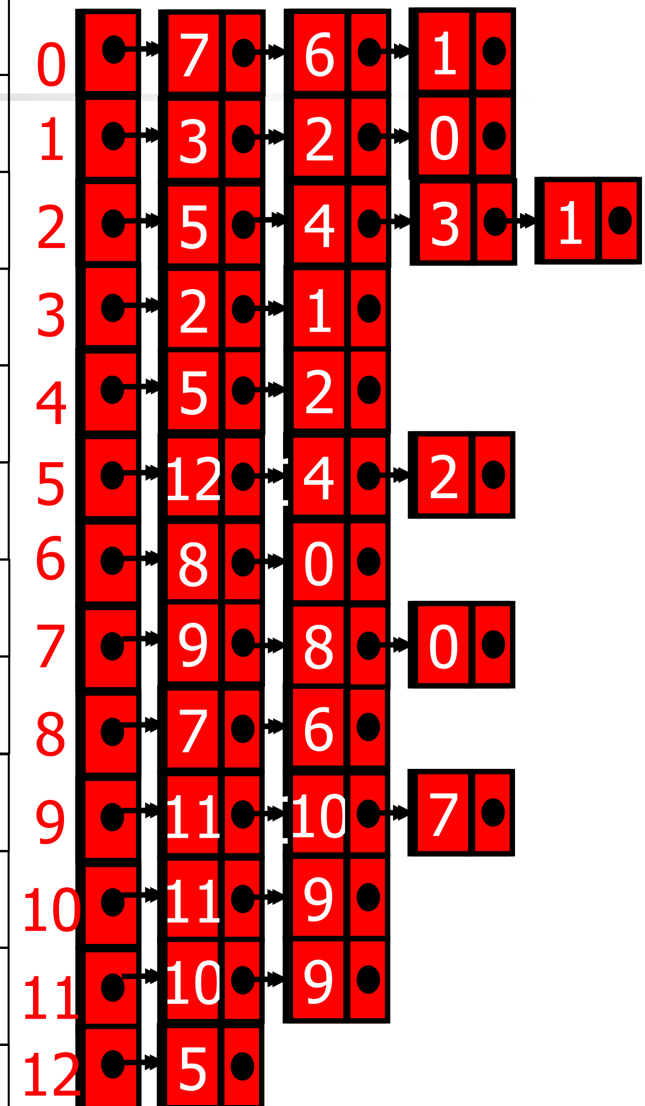
A B
B C
B D
C D
C E
C F
E F
A G
A H
G I
H I
H J
J K
J L
K L
F M

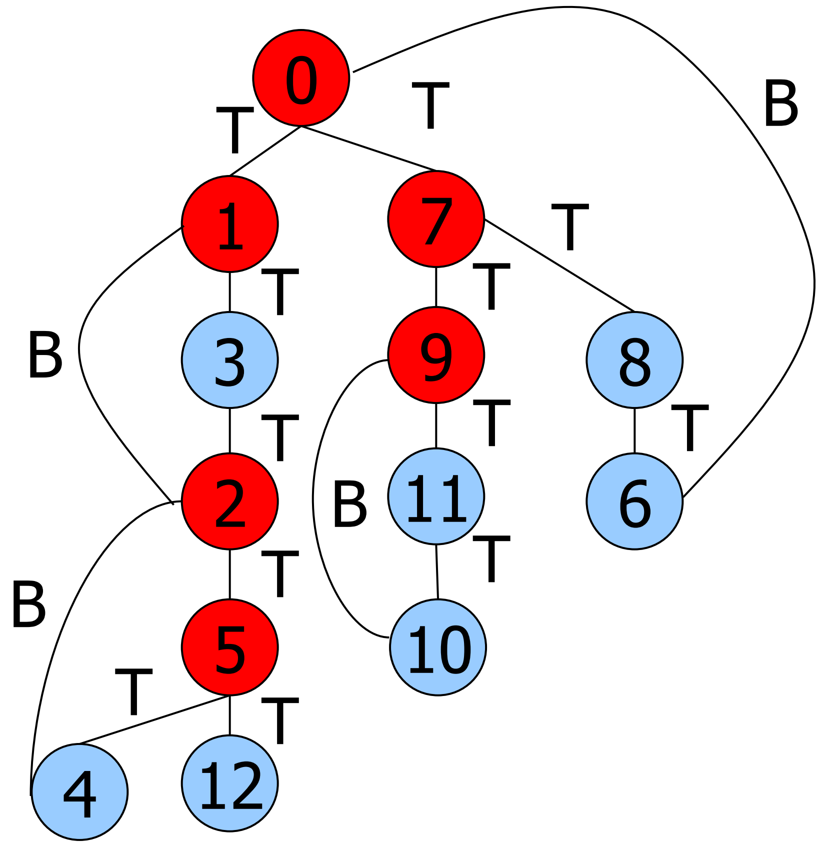


ST

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J
10	K
11	L
12	M

Lista delle adiacenze




0
T
T
B
1
7
T
T
T
B
3
9
8
T
T
T
2
B
11
6
T
B
T
5
10
T
T
4
12

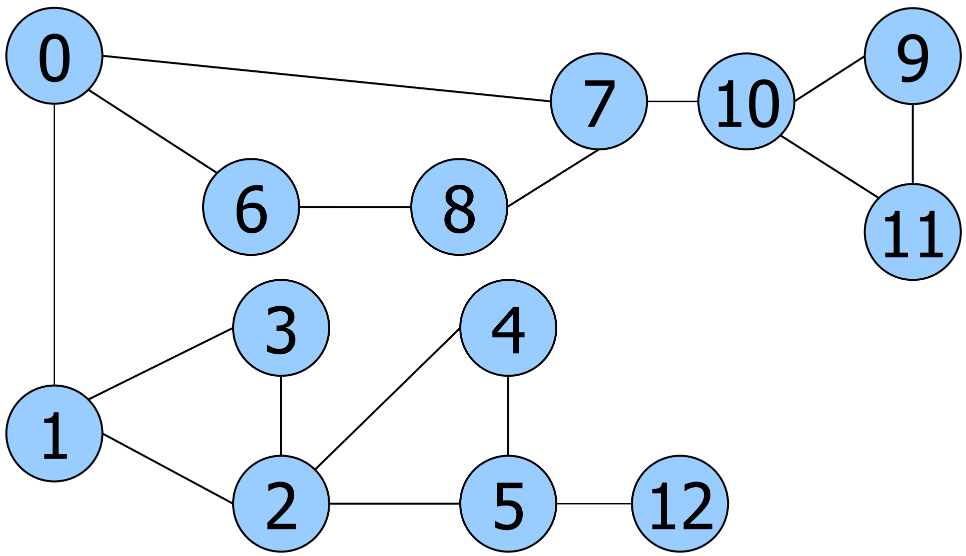
Bridge

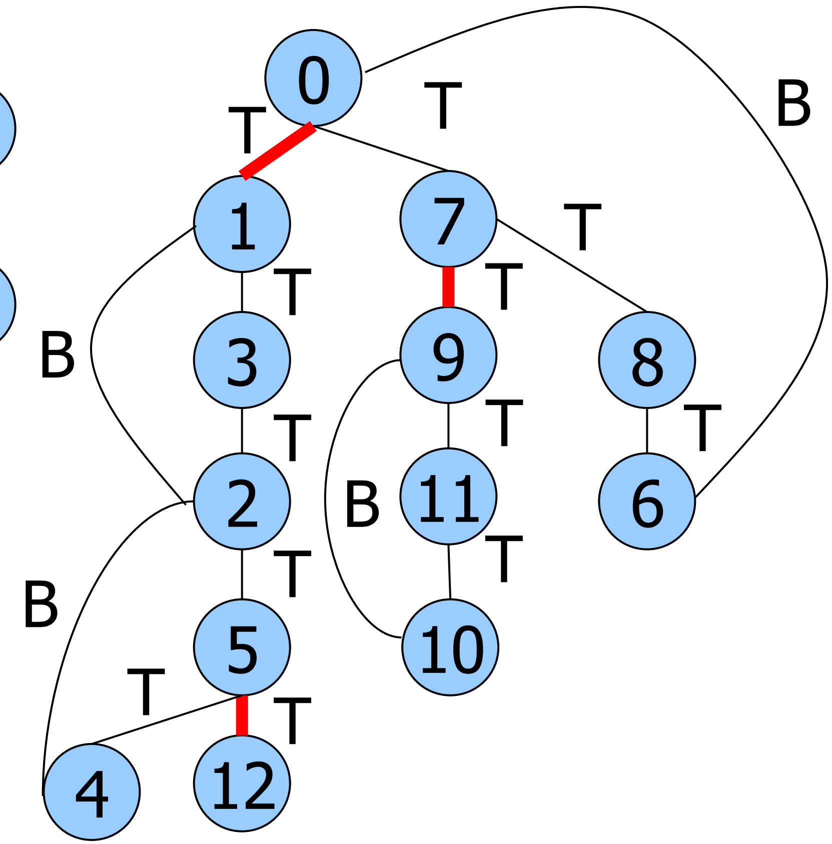
Un arco (v,w) Back non può essere un ponte (i vertici v e w sono anche connessi da un cammino nell’albero della visita DFS).

Un arco (v,w) Tree è un ponte se e solo se non esistono archi Back che connettono un discendente di w a un antenato di v nell’albero della visita DFS.

Algoritmo banale: rimuovere gli archi uno alla volta e verificare se il grafo rimane connesso.

Esempio





Bridge: (A, B), (F, M), (H, J)

Directed Acyclic Graph (DAG)

DAG: modelli impliciti per ordini parziali utilizzati nei problemi di scheduling.

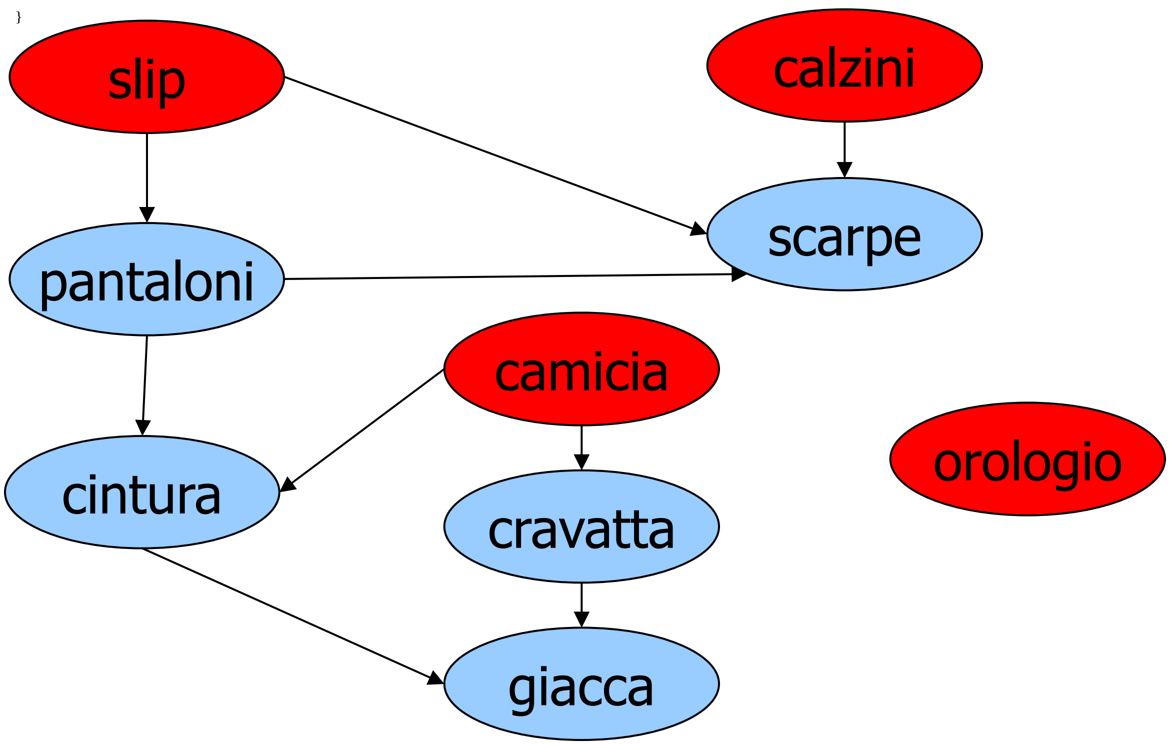
Scheduling:

- dati compiti (tasks) e vincoli di precedenza (constraints)
- come programmare i compiti in modo che siano tutti svolti rispettando le precedenze.

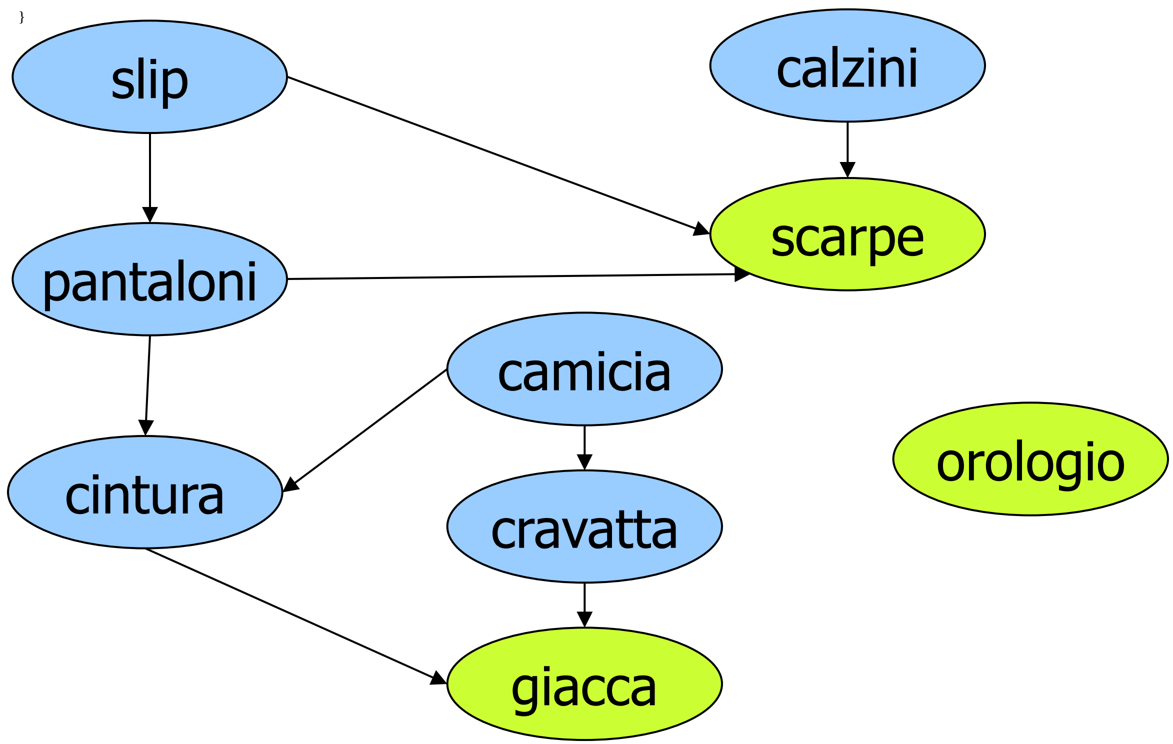
Nei DAG esistono 2 particolari classi di nodi:

- i nodi sorgente («source») che hanno in-degree=0
- i nodi pozzo o scolo («sink») che hanno out-degree=0.

Esempio



In rosso i nodi sorgente



In verde i nodi pozzo

Ordinamento topologico (inverso): riordino dei vertici secondo una linea orizzontale, per cui se esiste l'arco (u, v) il vertice u compare a SX (DX) di v e gli archi vanno tutti da SX (DX) a DX (SX).

I tempi di fine elaborazione `ts[v]` della visita DFS danno un ordinamento topologico inverso del DAG.

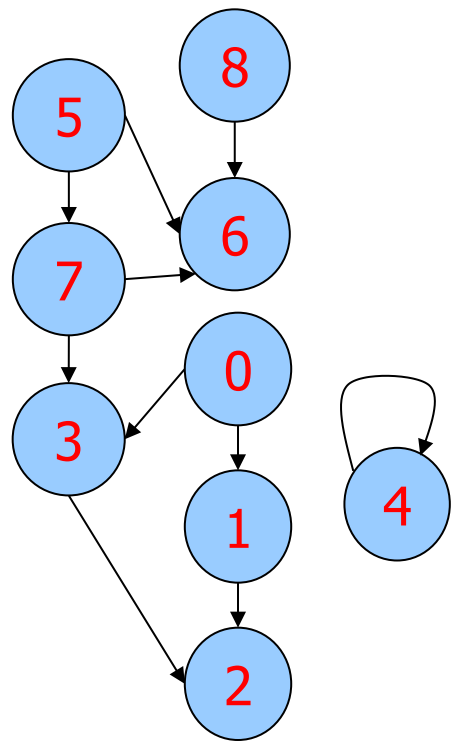
Esempio: ord. topologico inverso

in.txt

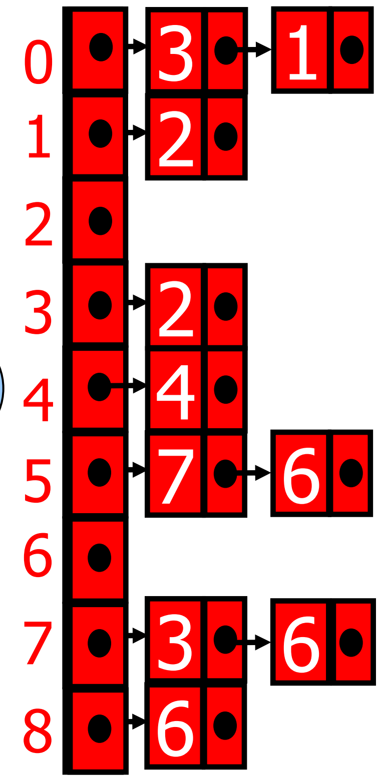
camicia cravatta
cravatta giacca
camicia cintura
cintura giacca
orologio orologio
slip scarpe
slip pantaloni
calzini scarpe
pantaloni sca
pantaloni cin

ST

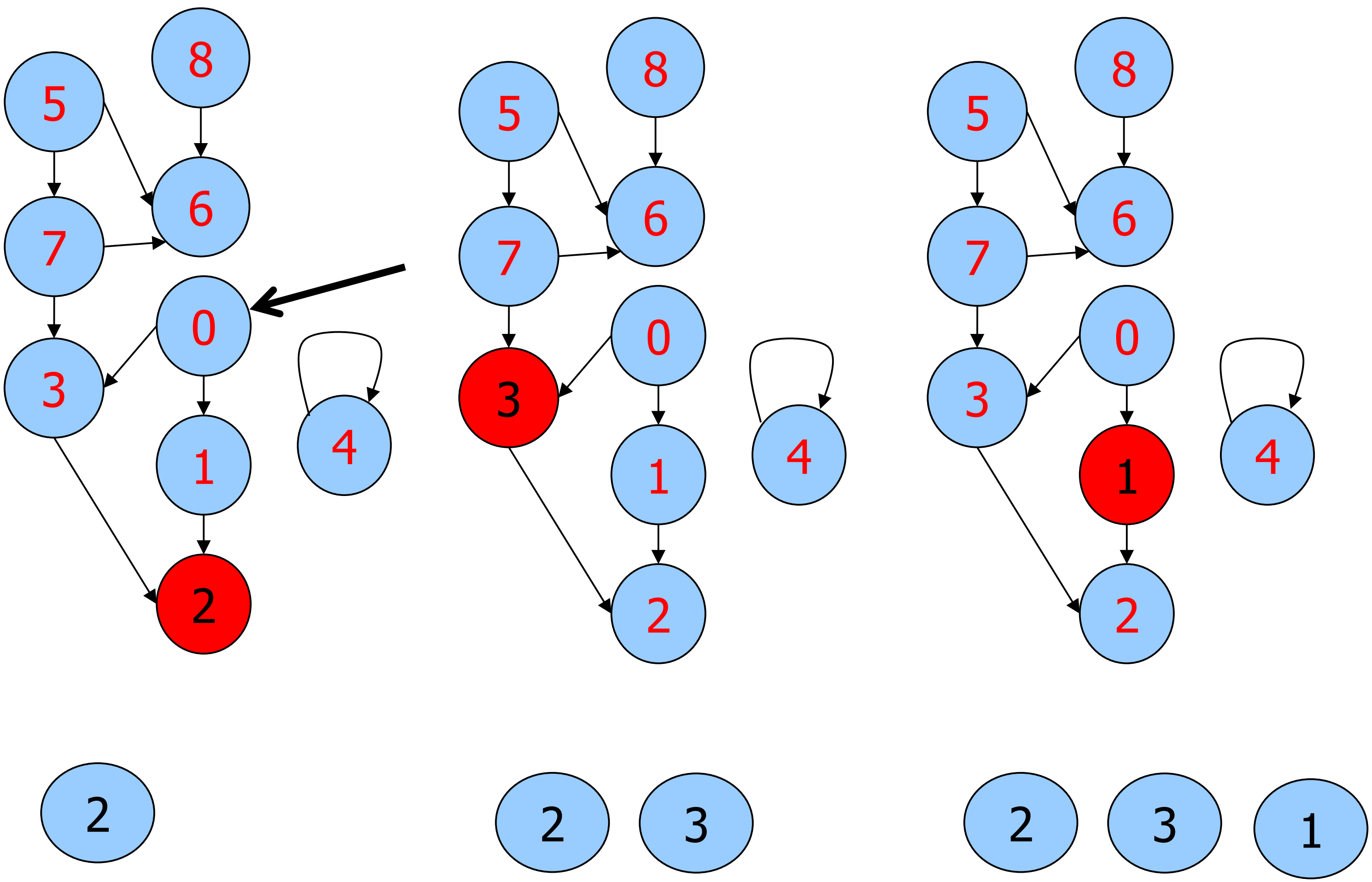
0	camicia
1	cravatta
2	giacca
3	cintura
4	orologio
5	slip
6	scarpe
7	pantaloni
8	calzini

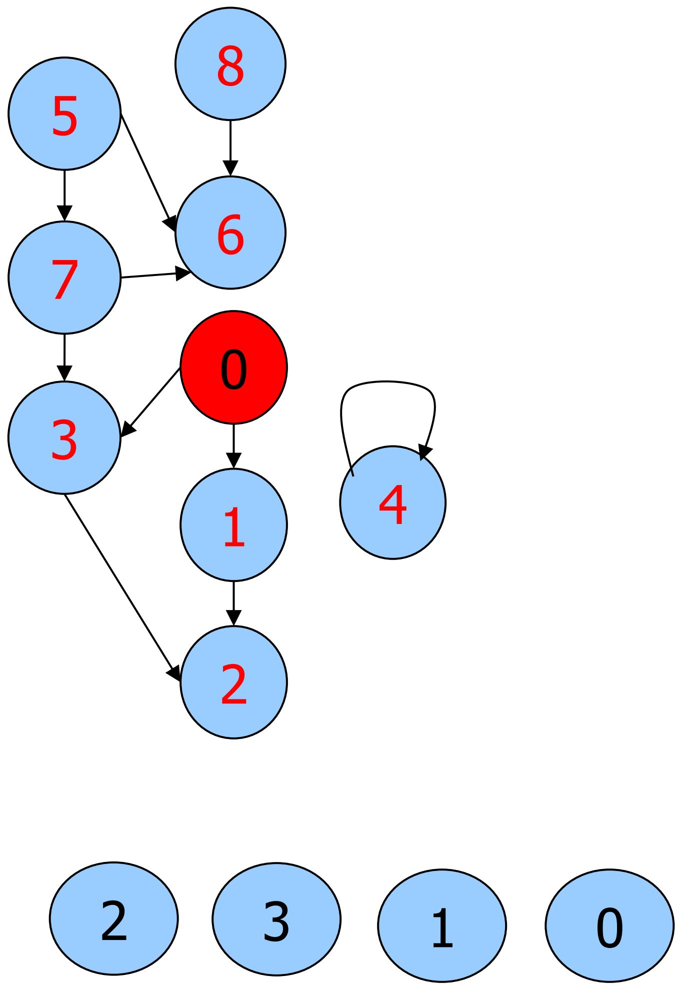


Lista delle adiacenze

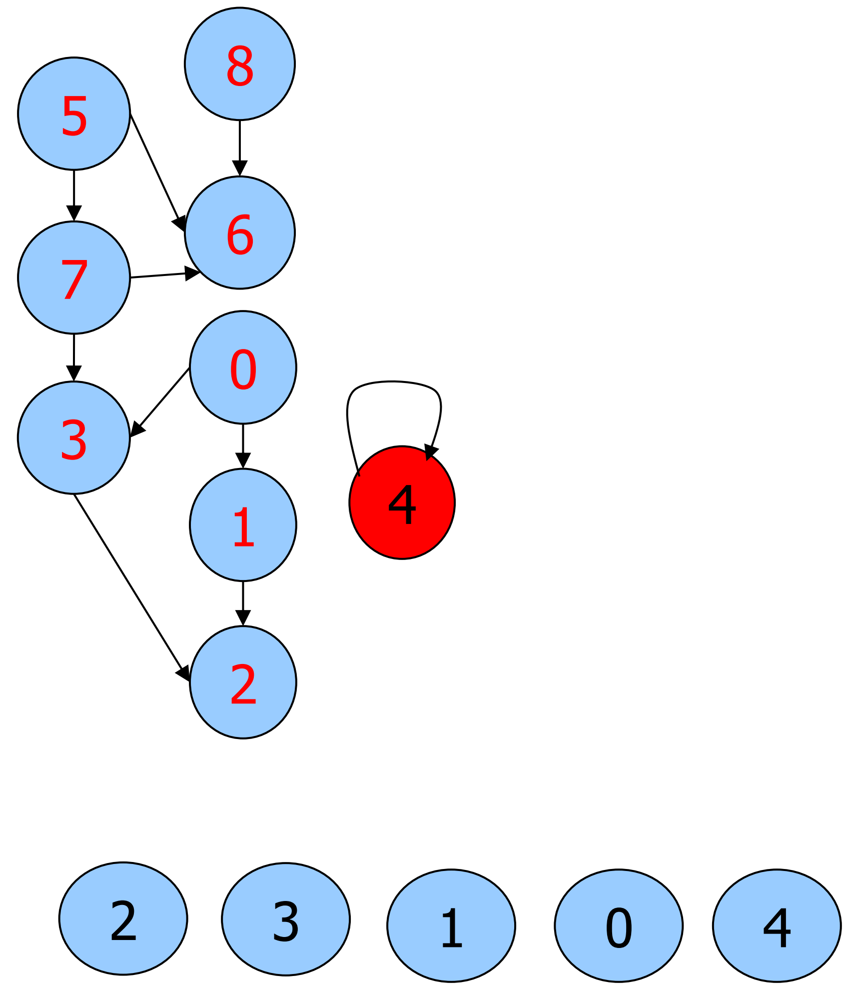


Arco fittizio necessario per poter creare il nodo

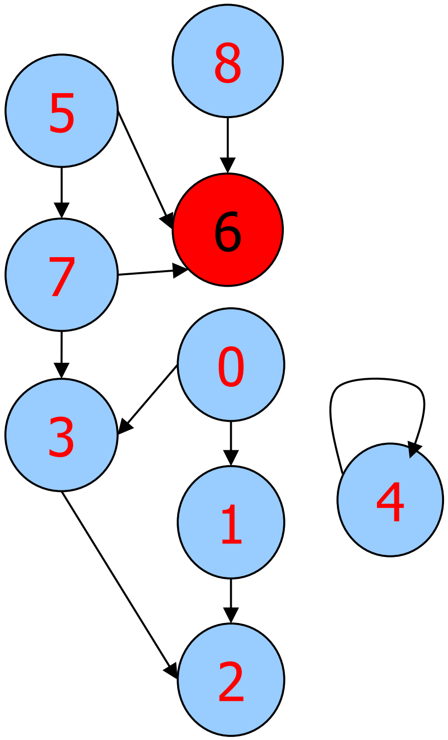

8
5
6
7
0
3
4
1
2
2
8
5
6
7
0
3
4
1
2
2
3
8
5
6
7
0
3
4
1
2
2
3
1

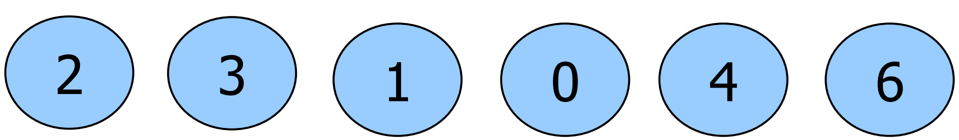


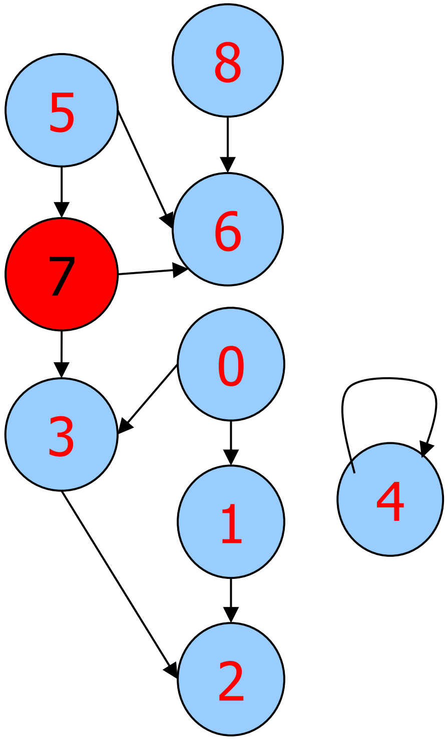
8
5
6
7
0
3
4
1
2
2 3 1 0

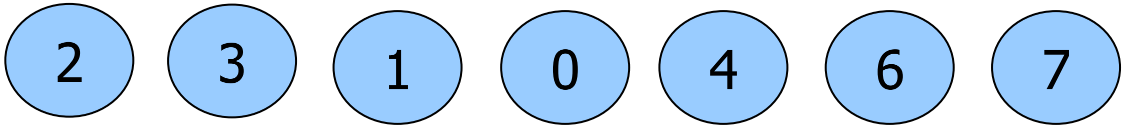


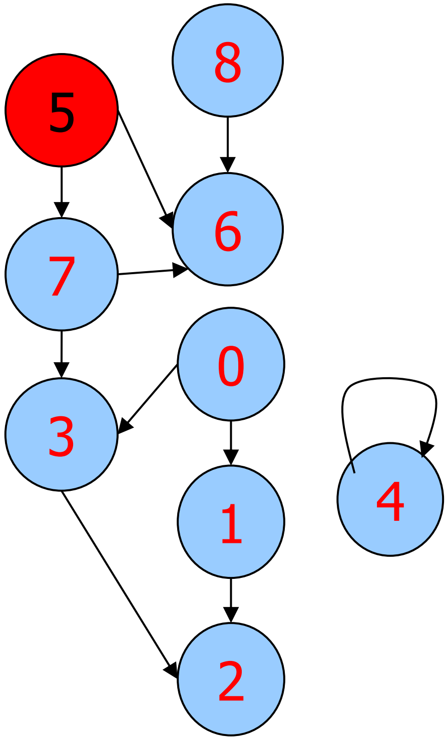
8
5
6
7
0
3
4
1
2
2 3 1 0 4

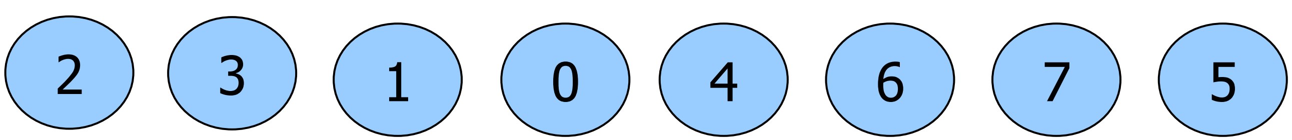

8
5
6
7
0
3
1
4
2

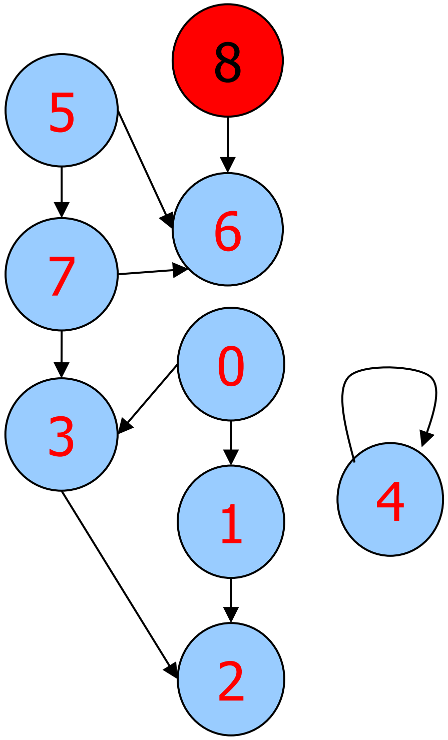

2 3 1 0 4 6

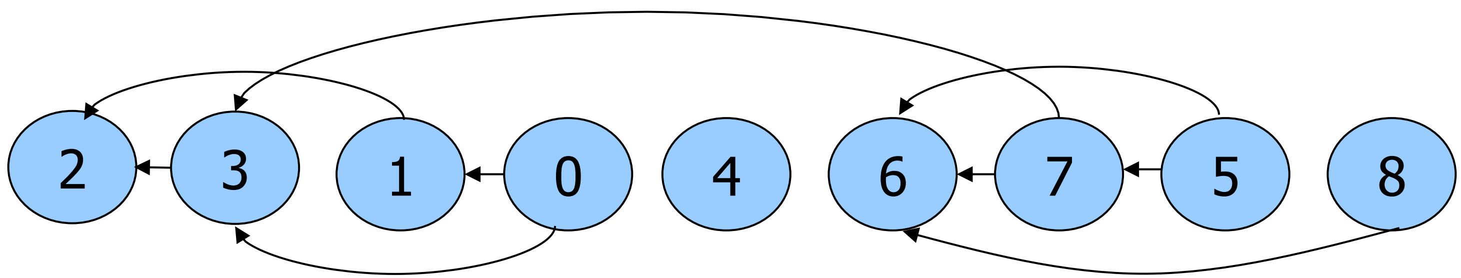

8
5
6
7
0
3
4
1
2


2
3
1
0
4
6
7


8
5
6
7
0
3
4
1
2


2 3 1 0 4 6 7 5


8
5
6
7
0
3
4
1
2


2
3
1
0
4
6
7
5
8

Strutture dati

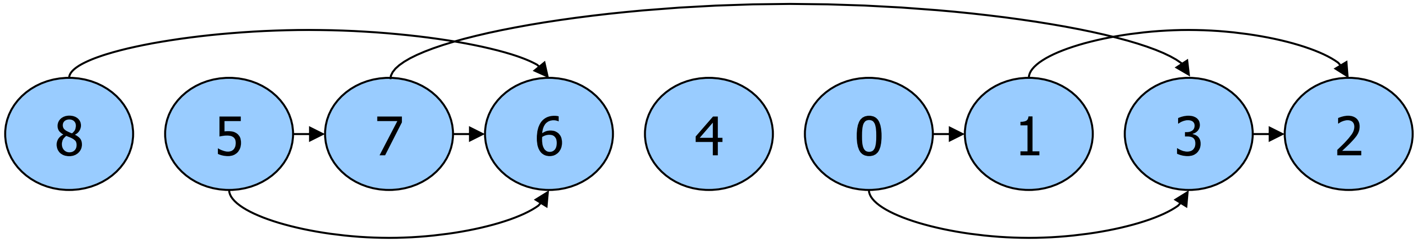
- DAG come ADT di I categoria
- rappresentazione come lista delle adiacenze
- vettori dove per ciascun vertice:
  - si registra il tempo di scoperta (numerazione in preordine dei vertici) `pre[i]`
  - vettore `ts[i]` dove per ciascun tempo si registra quale vertice è stato completato a quel tempo
- contatore `time` per tempi di completamento (avanza solo quando un vertice è completato, non scoperto)
- `time`, `*pre`, e `*ts` sono locali alla funzione `DAGrts` e passati by reference alla funzione ricorsiva `TSdfsR`.

wrapper

```
void TSdfsR(Dag D, int v, int *ts, int *pre, int *time) {
  link t; pre[v] = 0;
  for (t = D->adj[v]; t != D->z; t = t->next)
    if (pre[t->v] == -1)
      TSdfsR(D, t->v, ts, pre, time);
  ts[(*time)++] = v;
}
void DAGrts(Dag D) {
  int v, time = 0, *pre, *ts;
  pre = malloc(D->V*sizeof(int));
  ts = malloc(D->V*sizeof(int));
  for (v=0; v < D->V; v++) { pre[v] = -1; ts[v] = -1; }
  for (v=0; v < D->V; v++)
    if (pre[v]== -1) TSdfsR(D, v, ts, pre, &time);
  printf("DAG nodes in reverse topological order \n");
  for (v=0; v < D->V; v++)
     printf("%s ", STretrieve(D->tab, ts[v]));
  printf("\n");
}
```

ordine topologico: con il DAG rappresentato da una matrice delle adiacenze, basta invertire i riferimenti riga-colonna:

```
void TSdfsR(Dag D, int v, int *ts, int *pre, int *time) {
  int w;
  pre[v] = 0;
  for (w  = 0; w < D->V; w++)
    if (D->adj[w][v]  != 0)
      if (pre[w] == -1)
        TSdfsR(D, w, ts);
  ts[(*time)++] = v;
}
```



Componenti fortemente connesse

Algoritmo di Kosaraju (anni ’80):

- trasporre il grafo
- eseguire DFS sul grafo trasposto, calcolando i tempi di scoperta e di fine elaborazione
- eseguire DFS sul grafo originale per tempi di fine elaborazione descrescenti
- gli alberi dell’ultima DFS sono le componenti fortemente connesse.

- Le SCC sono classi di equivalenza rispetto alla proprietà di mutua raggiungibilità
- Si può “estrarre” un grafo ridotto G’ considerando un vertice come rappresentativo di ogni classe
- Il grafo ridotto G’ è un DAG ed è detto “kernel DAG” del grafo G.

Grafo trasposto

Dato un grafo orientato G=(V, E), il suo grafo trasposto G^T =(V, E^T) è tale per cui

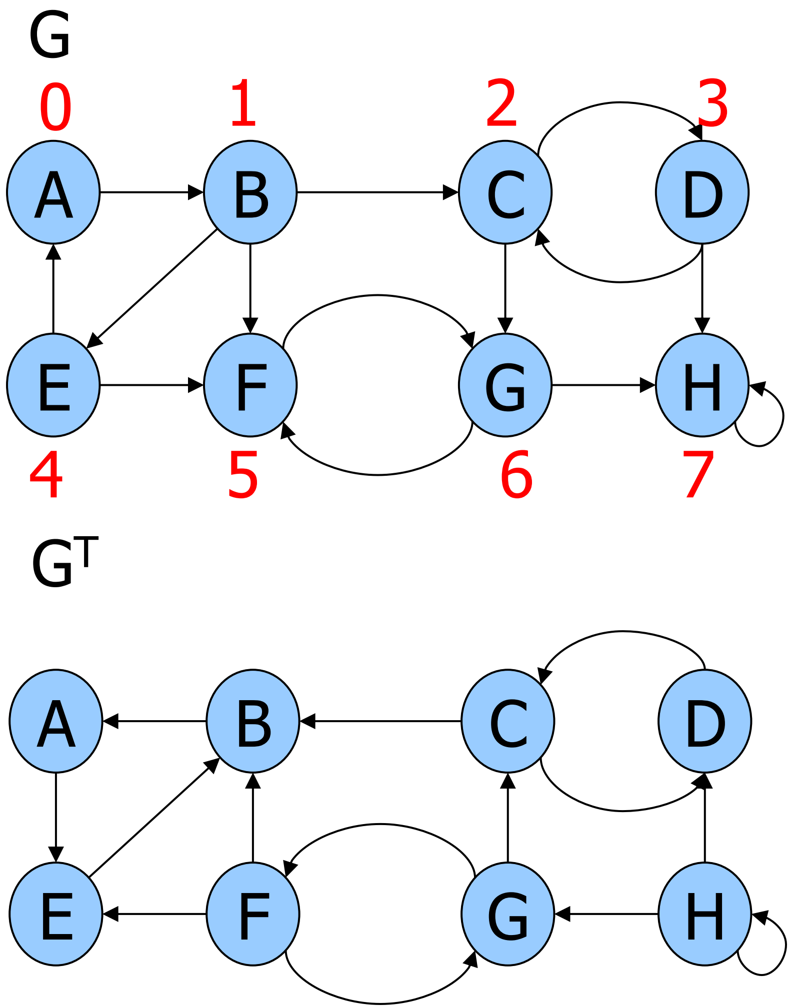
$$(u, v) \in E \Leftrightarrow (v,u) \in E^T.$$

```
Graph reverse(Graph G) {
  int v;
  link t;
  Graph R = GRAPHinit(G->V);
  for (v=0; v < G->V; v++)
    for (t= G->adj[v]; t != G->z; t = t->next)
      insertE(R, EDGEcreate(t->v, v));
  return R;
}
```

Esempio

in.txt

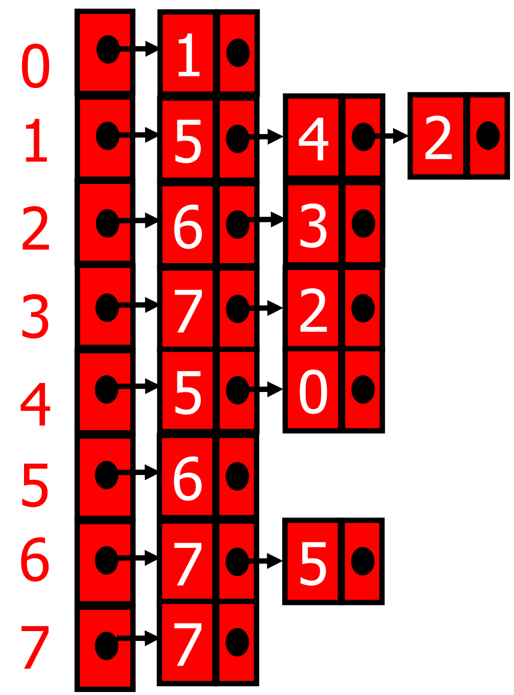
A B
B C
C D
D C
E A
E F
B E
B F
F G
G F
C G
G H
D H
H H



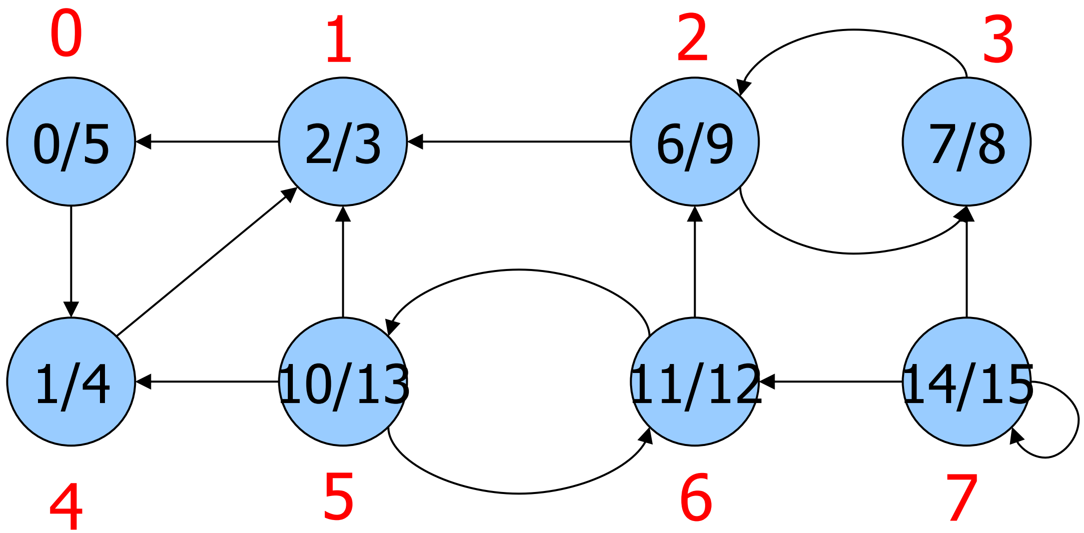
ST

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

Lista delle adiacenze di G

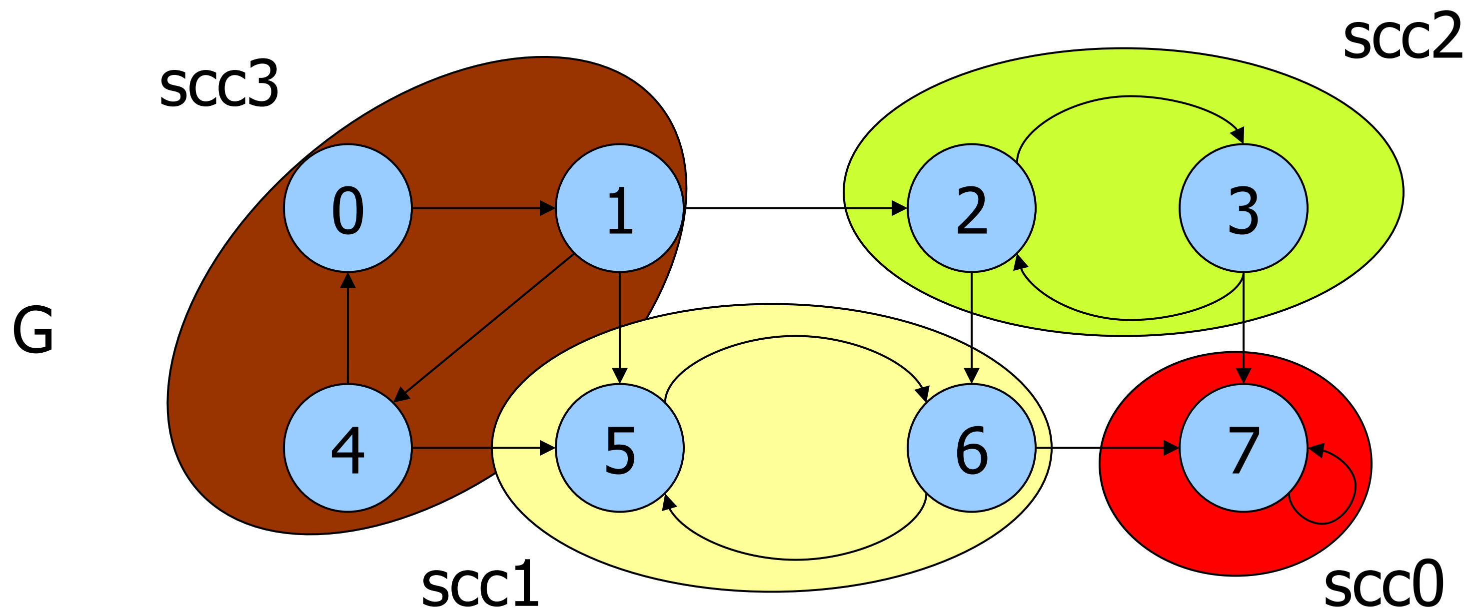


Visita DFS del grafo trasposto G^T (sono indicati gli usuali tempi di scoperta e di fine elaborazione della DFS). Il codice calcola solo i tempi di fine elaborazione.

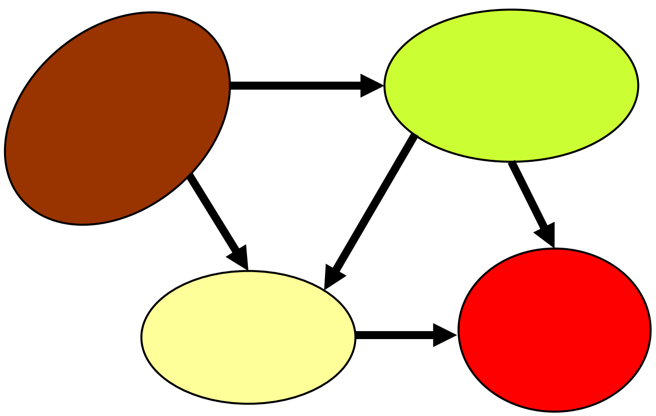


Visita DFS del grafo secondo tempi decrescenti di fine elaborazione del grafo trasposto G^T

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



- Kernel DAG.



Algoritmo e strutture dati

- `sccG[w]` e `sccR[w]` sono array locali a `GRAPHscc` che memorizzano un intero che identifica ciascuna componente fortemente connessa. I vertici fungono da indici dell'array
- `time0`, `time1`, `*postG` e `*postV` sono locali alla funzione `GRAPHscc` e passati by reference alla funzione ricorsiva `SCCdfsR`.

wrapper

- nella DFS del grafo trasposto il contatore del tempo `time0` avanza solo quando di un nodo è terminata l'elaborazione (non serve il tempo di scoperta). `time1` è il contatore delle SCC.

- l'istruzione `post[(*time0)++]=w` registra che al tempo (*`time0`) è stato terminato `w`, quindi c'è un implicito ordinamento per tempi di completamento crescenti. I nodi vengono presi per tempo di completamento decrescenti con il ciclo discendente da `G->V-1` a 0.

```
void SCCdfsR(Graph G, int w, int *scc, int *time0,
             int time1, int *post) {
  link t;
  scc[w] = time1;
  for (t = G->adj[w]; t != G->z; t = t->next)
    if (scc[t->v] == -1)
      SCCdfsR(G, t->v, scc, time0, time1, post);
  post[(*time0)++]= w;
}
int GRAPHscc(Graph G) {
  int v, time0 = 0, time1 = 0, *sccG, *sccR, *postG, *postR;
  Graph R = GRAPHreverse(G);

  sccG = malloc(G->V * sizeof(int));
  sccR = malloc(G->V * sizeof(int));
  postG = malloc(G->V * sizeof(int));
  postR = malloc(G->V * sizeof(int));
```

```
  for (v=0; v < G->V; v++) {
    sccG[v]=-1; sccR[v]=-1; postG[v]=-1; postR[v]=-1;
  }
  for (v=0; v < G->V; v++)
    if (sccR[v] == -1)
      SCCdfsR(R, v, sccR, &time0, time1, postR);
  time0 = 0; time1 = 0;
  for (v = G->V-1; v >= 0; v--)
    if (sccG[postR[v]]==-1){
      SCCdfsR(G,postR[v], sccG, &time0, time1, postG);
      time1++;
    }
  printf("strongly connected components \n");
  for (v = 0; v < G->V; v++)
    printf("node %d in scc %d \n", v, sccG[v]);
  return time1;
}
```

Riferimenti

- Componenti connesse:
  - Sedgewick Part 5 18.5
- Bridge e punti di articolazione:
  - Sedgewick Part 5 18.6
- DAG e ordinamento topologico dei DAG:
  - Sedgewick Part 5 19.5 e 19.6
  - Cormen 23.4
- Componenti fortemente connesse:
  - Sedgewick Part 5 19.8
  - Cormen 23.5