POLITECNICO
DI TORINO

Dipartimento
di Automatica e Informatica

# Le applicazioni degli algoritmi di visita dei grafi

Gianpiero Cabodi e Paolo Camurati

# Rilevazione di cicli

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

Non basta però per identificare I cicli!

# Componenti connesse

In un grafo non orientato:

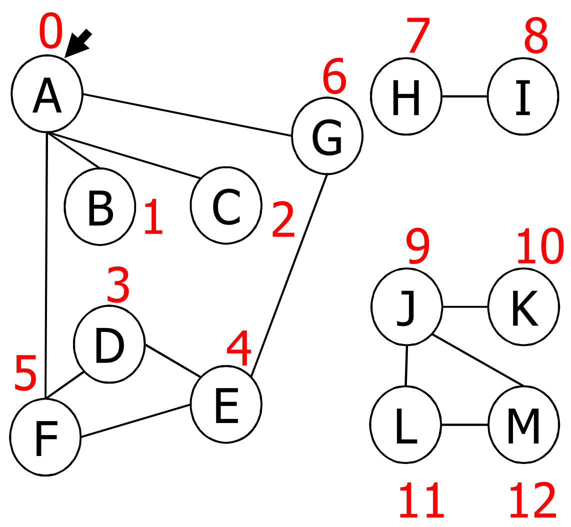
- ogni albero della foresta della DFS è una componente connessa
- `cc[v]` è un array locale a `GRAPHcc` che memorizza un intero che identifica ciascuna componente connessa. I vertici fungono da indici dell’array

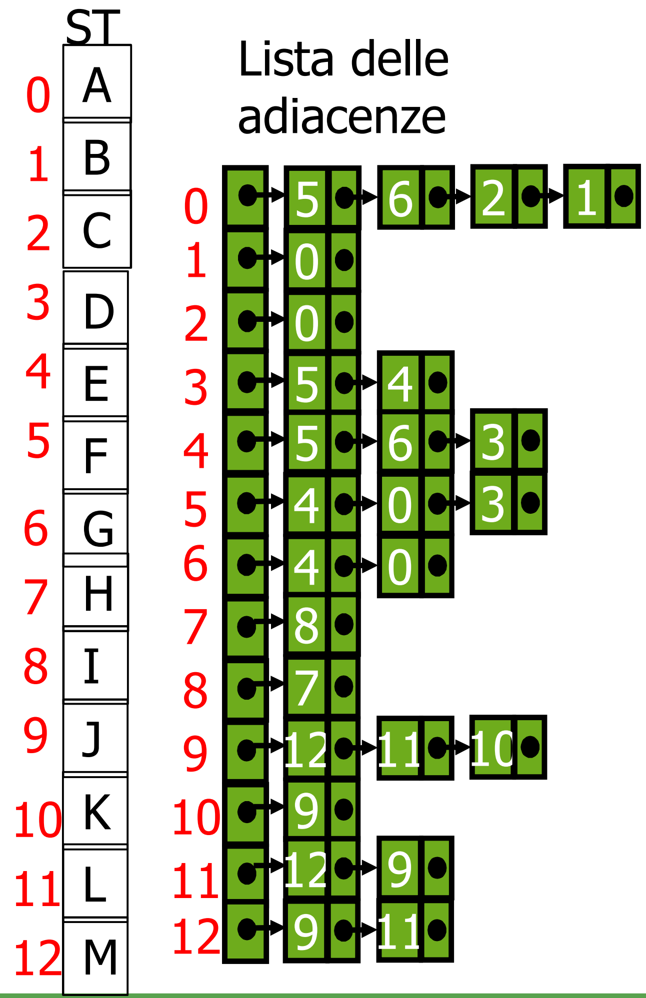
wrapper

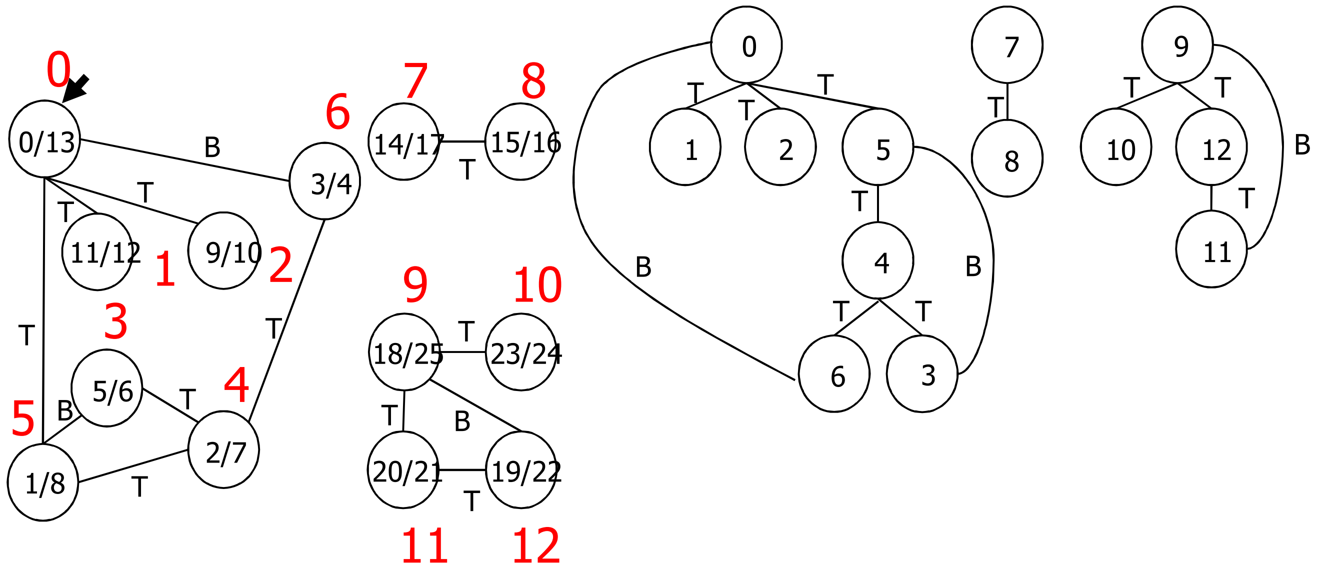
# Esempio

Lista archi

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






| | 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->ladj[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",STsearchByIndex(G->tab,v),cc[v]);
  return id;
}
```

# Connettività

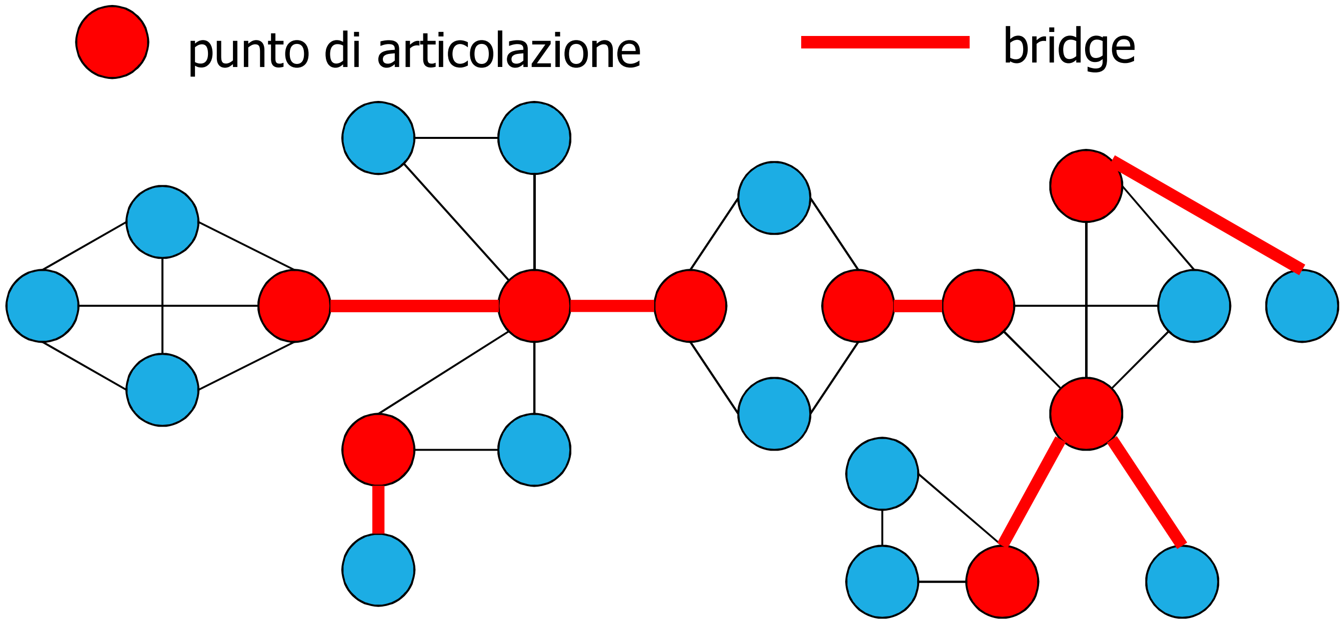
Dato un grafo non orientato e connesso, determinare se perde la proprietà di connessione a seguito della rimozione di:

- 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

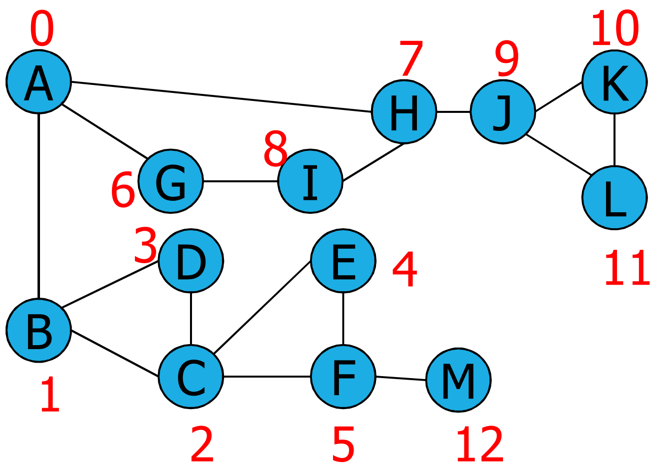
Dato un grafo non orientato e connesso G, dato l'albero $G_\pi$ della visita in profondità,

- la radice di $G_\pi$ 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

Lista archi

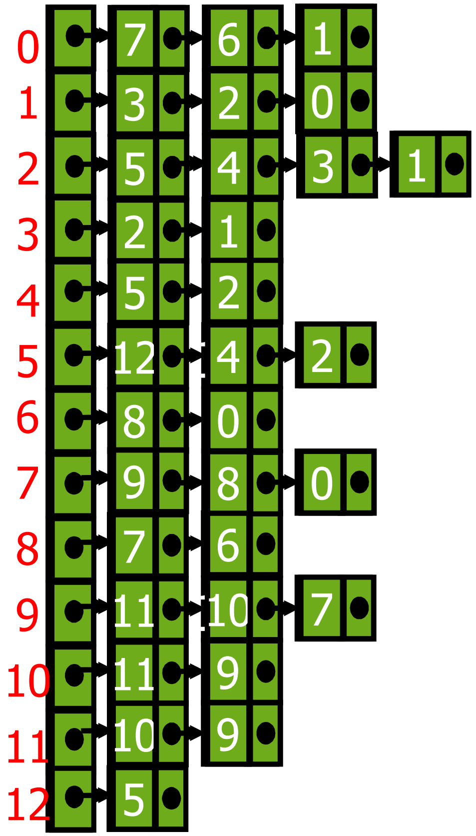
A B
B C
B D
C D
C E
C F
E F
A G
A H
G I
H I
H J
J K
J L
K L
F M

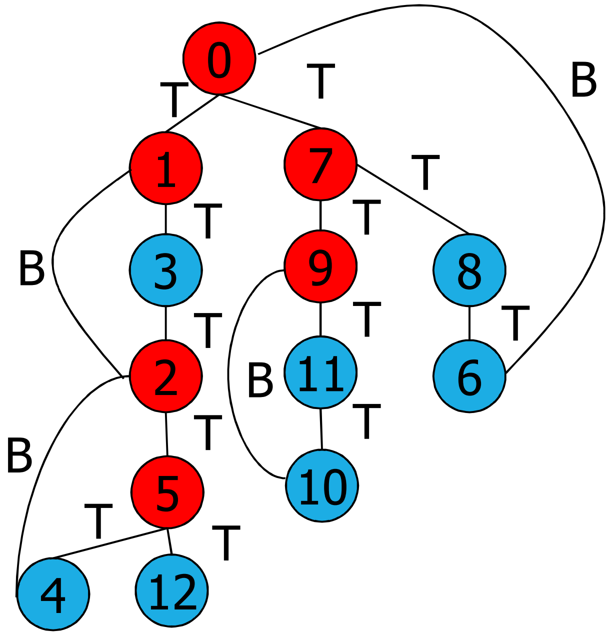



ST

| | |
|---|---|
| 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
11
6
B
T
T
B
5
10
T
T
4
12

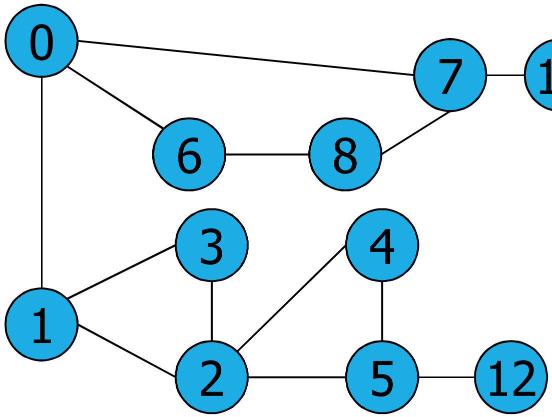
# Bridge

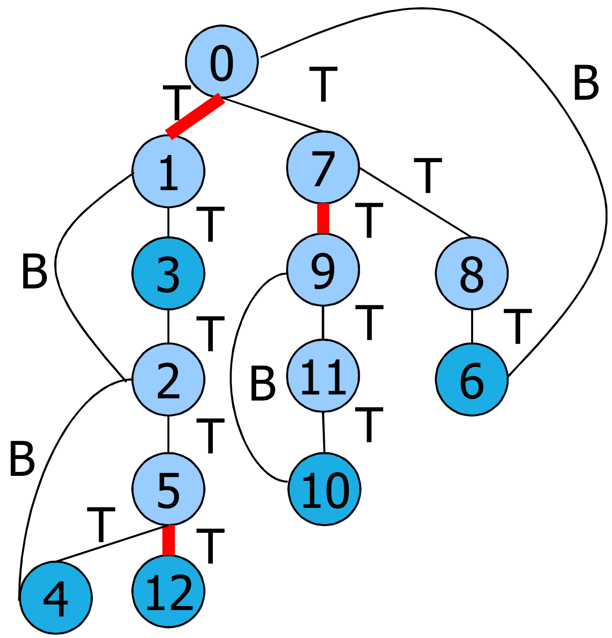
Un arco (v,w) Back non può essere un ponte (i vertici v e w sono anche connessi da un cammino nell'albero della visita DFS).

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.

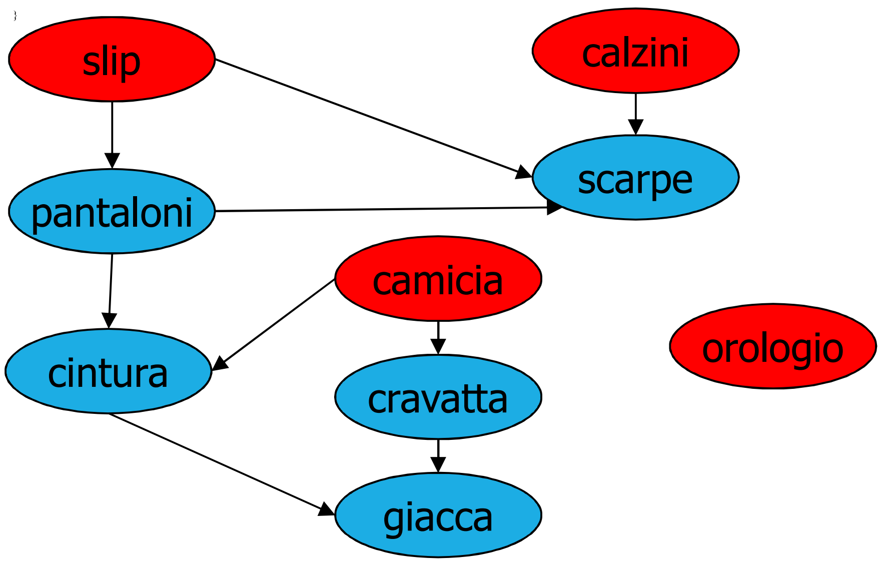
Scheduling:

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

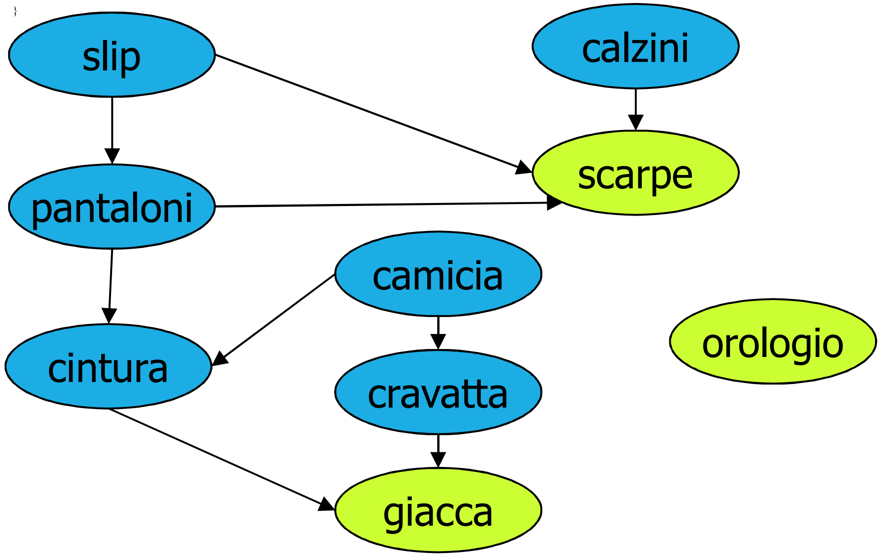
Nei DAG esistono 2 particolari classi di nodi:

- 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



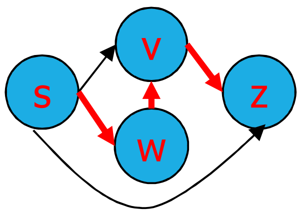

In verde i nodi pozzo

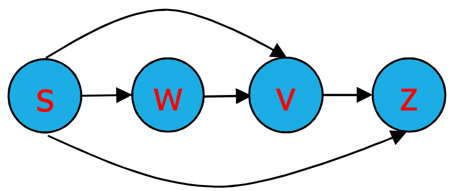
Ordinamento topologico: riordino dei vertici secondo una linea orizzontale, per cui se esiste l'arco (u, v) il vertice u compare a SX di v e gli archi vanno tutti da SX a DX.

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

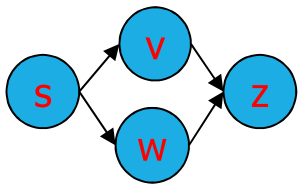
# Unicità dell'ordinamento topologico

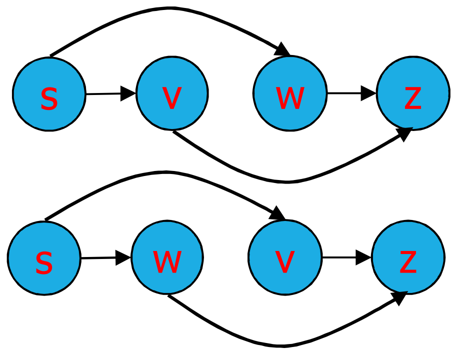
Se esiste un cammino hamiltoniano orientato, l'ordinamento topologico è unico. Tutte le coppie di vertici consecutivi sono connesse da archi.



Se $\nexists$ cammino hamiltoniamo orientato $\Rightarrow$ l'ordinamento topologico non è unico




Ogni DAG ha quindi sempre almeno un ordinamento topologico.
Data la rappresentazione del grafo, il codice calcolerà un solo ordinamento topologico.

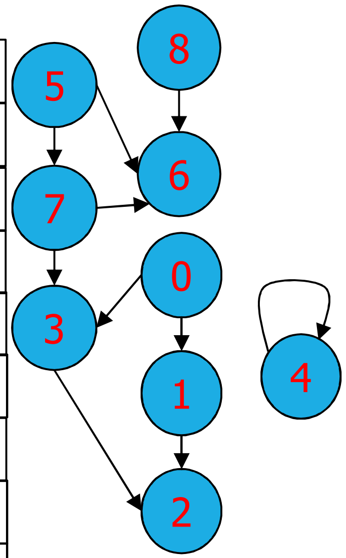
# Esempio: ordinamento topologico inverso

Lista archi

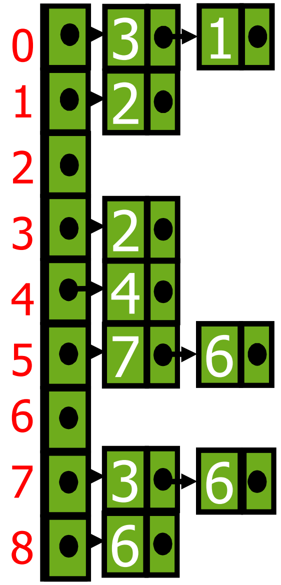
camicia cravatta
cravatta giacca
camicia cintura
cintura giacca
orologio orologio
slip scarpe
slip pantalo
calzini scar
pantaloni s
pantaloni

ST

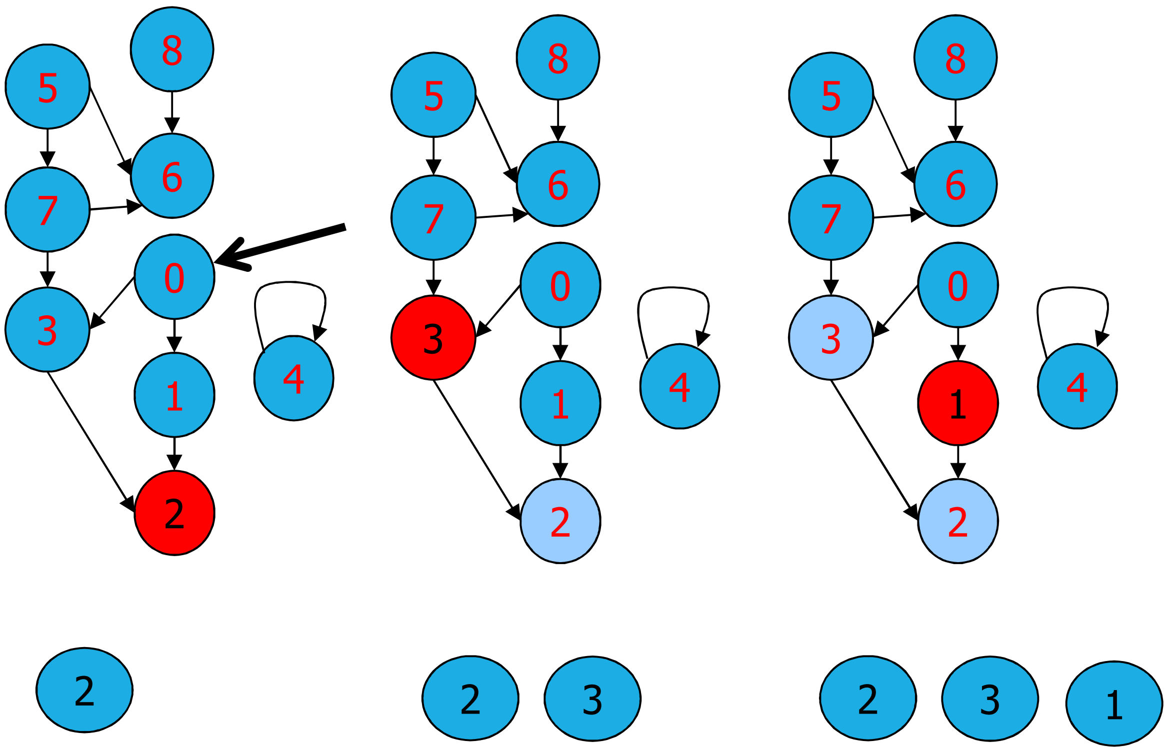
| | |
|---|---|
| 0 | camicia |
| 1 | cravatta |
| 2 | giacca |
| 3 | cintura |
| 4 | orologio |
| 5 | slip |
| 6 | scarpe |
| 7 | pantaloni |
| 8 | calzini |

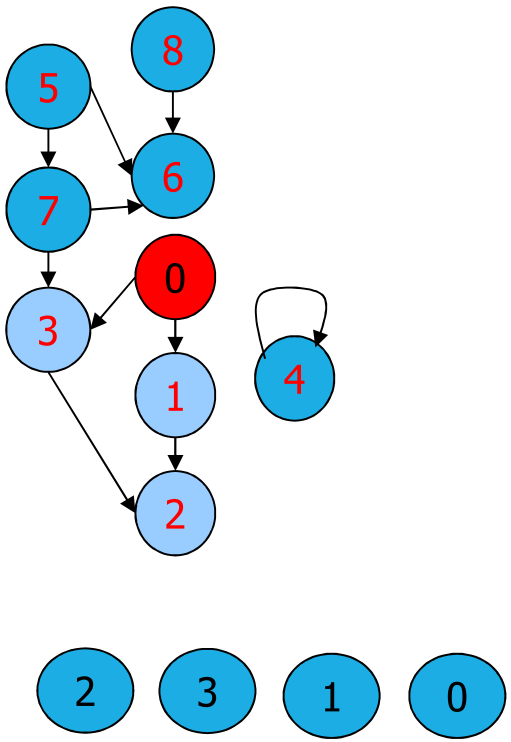


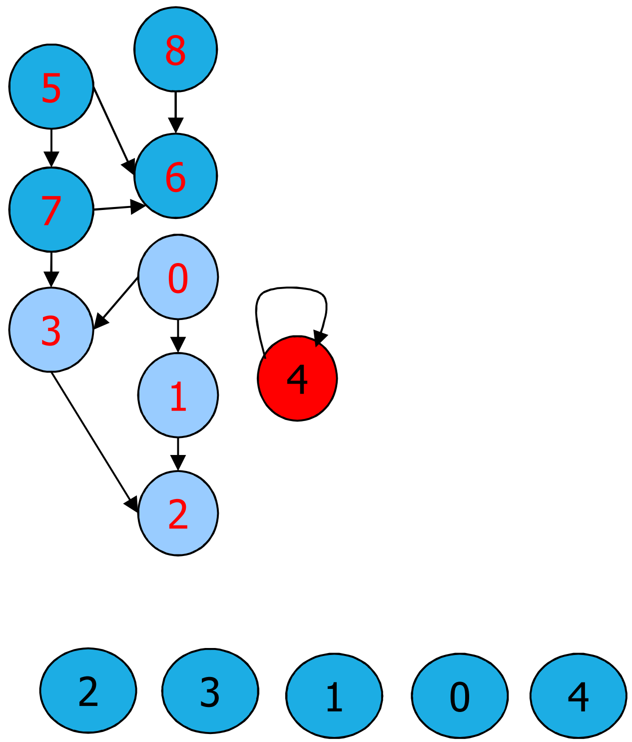

Lista delle adiacenze

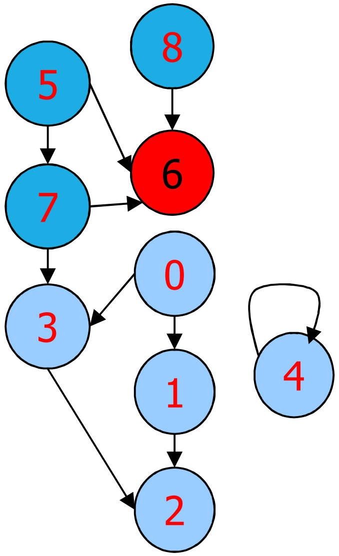



arco fittizio necessario per poter creare il nodo

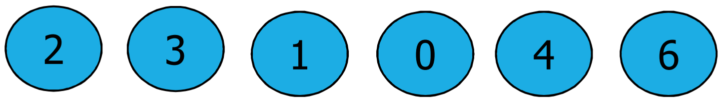

5
8
7
6
0
3
1
4
2
2
5
8
7
6
0
3
1
4
2
2
3
5
8
7
6
0
3
1
4
2
2
3
1


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

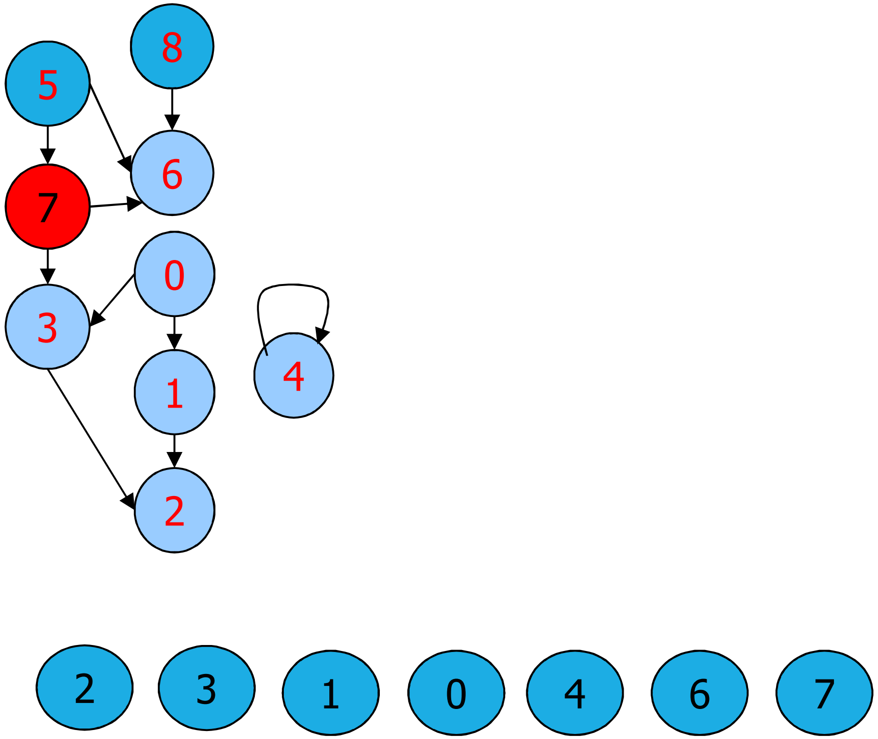

5
8
6
7
0
3
1
4
2
2
3
1
0
4

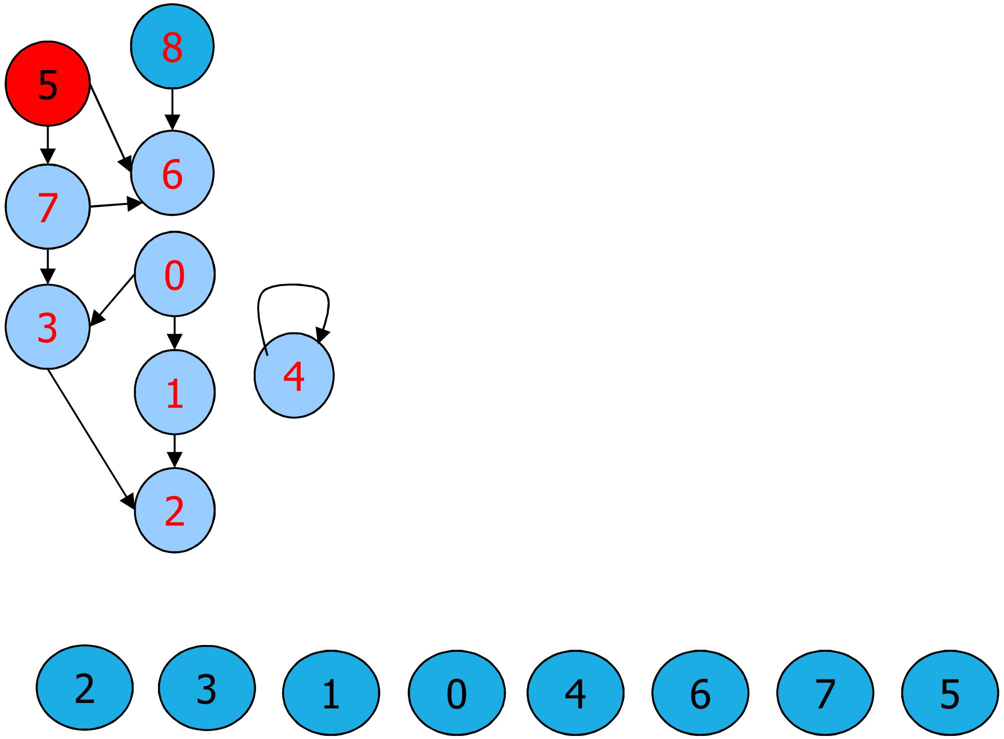


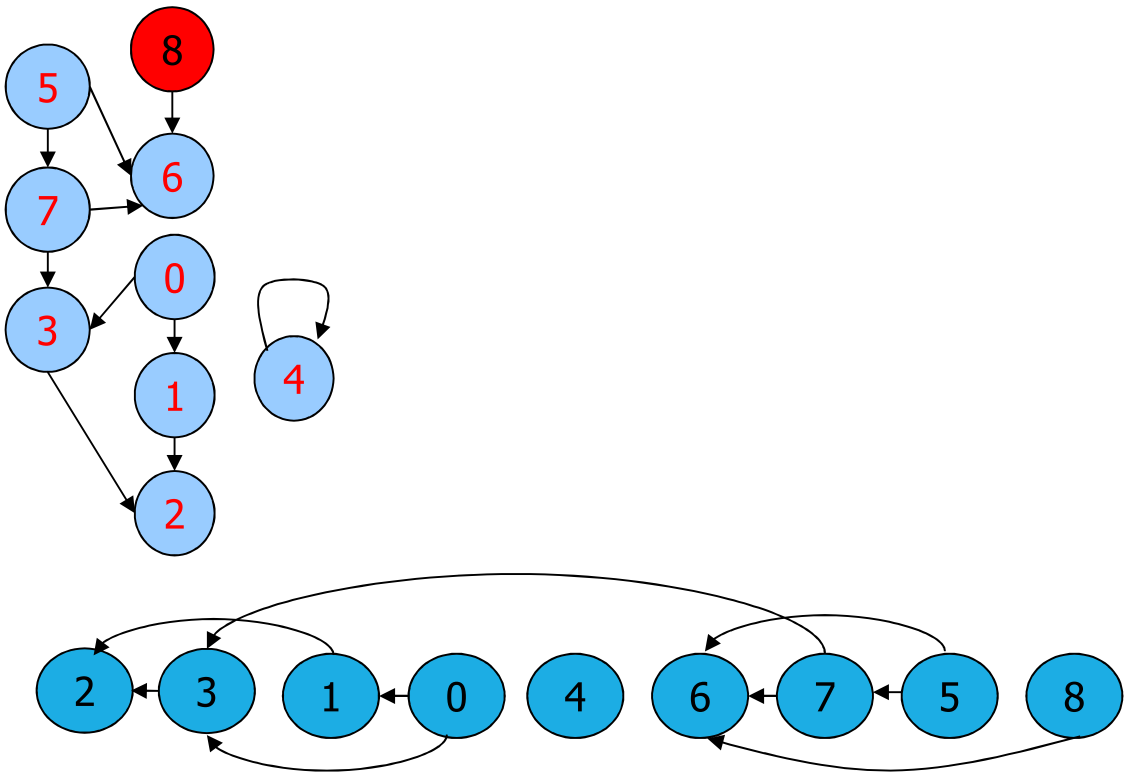
8
5
6
7
0
3
4
1
2

2
3
1
0
4
6


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


5
8
6
7
0
3
1
4
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

## Struttura dati

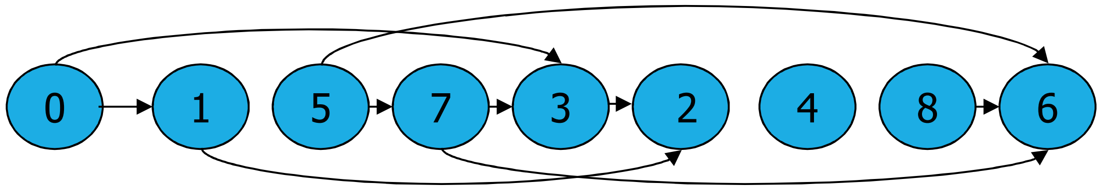
- DAG come ADT di I classe
- rappresentazione come lista delle adiacenze
- vettori dove per ciascun vertice:
  - vettore `pre[i]` per registrare se il vertice è già stato scoperto o meno
  - vettore `ts[i]` dove per ciascun tempo si registra quale vertice è stato completato a quel tempo
- contatore `time` 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->ladj[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;
  /* allocazione di pre e ts */
  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 ", STsearchByIndex(D->tab, ts[v]));
  printf("\n");
}
```

ordinamento topologico: con il DAG rappresentato da una matrice delle adiacenze, basta invertire i riferimenti riga-colonna (considerando gli archi incidenti):

```
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->madj[w][v]  != 0)
      if (pre[w] == -1)
        TSdfsR(D, w, ts, pre, time);
  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 decrescenti
- 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

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

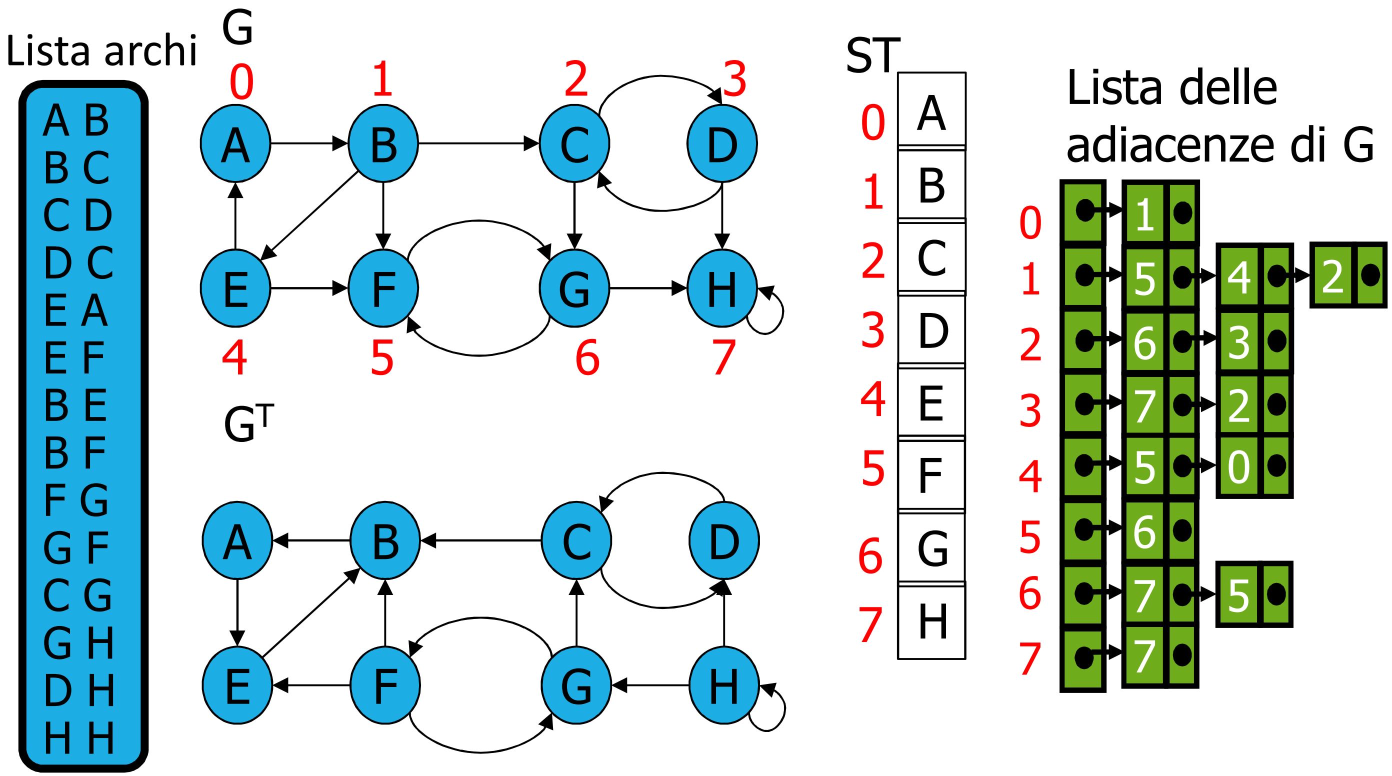
```
Graphreverse(Graph G) {
  int v;
  link t;
  Graph R = GRAPHinit(G->V);
  for (v=0; v < G->V; v++)
    for (t= G->ladj[v]; t != G->z; t = t->next)
      GRAPHinsertE(R, t->v, v);
  return R;
}
```

# Algoritmo e strutture dati

- `GRAPHscc` è il wrapper con i vettori e le variabili locali passati by reference alla funzione ricorsiva `SCCdfsR`.
- in `sccG[w]` per ogni vertice si memorizza un intero che
  - identifica la componente fortemente connessa cui esso appartiene
  - marca anche se il vertice è stato visitato dalla DFS
- `sccR[w]` serve per marcare i vertici visitati dalla DFS del grafo trasposto
- `time0` è il contatore del tempo che avanza solo quando di un vertice è terminata l'elaborazione (non serve il tempo di scoperta)

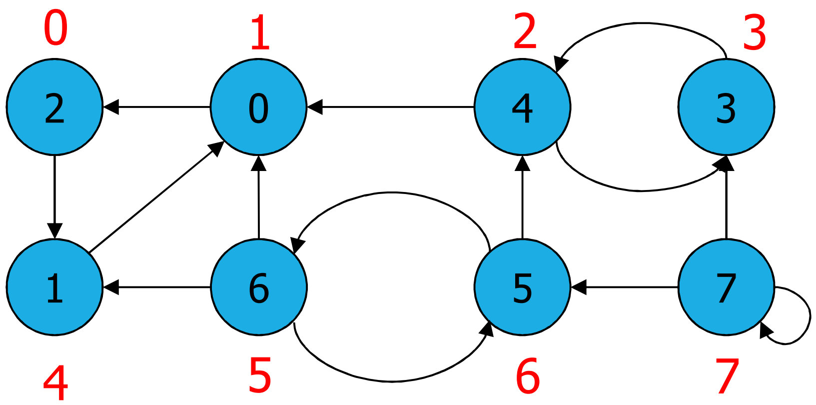
- `time1` è il contatore delle SCC
- `*postR` contiene per ogni valore del contatore di tempo `time0` quale vertice è stato terminato a quel tempo
- l'istruzione `post[(*time0)++]=w` registra che al tempo `(*time0)` è stato terminato `w`, quindi c'è un implicito ordinamento per tempi di completamento crescenti
- percorrendo in discesa il vettore `postR` si considerano i vertici in ordine di tempo di fine elaborazione decrescente senza bisogno di un algoritmo di ordinamento
- `*postG` viene introdotto soltanto per avere un'unica versione della funzione ricorsiva `SCCdfsR` utilizzabile sia sul grafo `G`, sia sul grafo trasposto $G^T$.

# Esempio



Lista archi
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
G
0 1 2 3
A B C D
E F G H
4 5 6 7
G$^T$
A B C D
E F G H
ST
0 A
1 B
2 C
3 D
4 E
5 F
6 G
7 H
Lista delle adiacenze di G
0 1
1 5 4 2
2 6 3
3 7 2
4 5 0
5 6
6 7 5
7 7

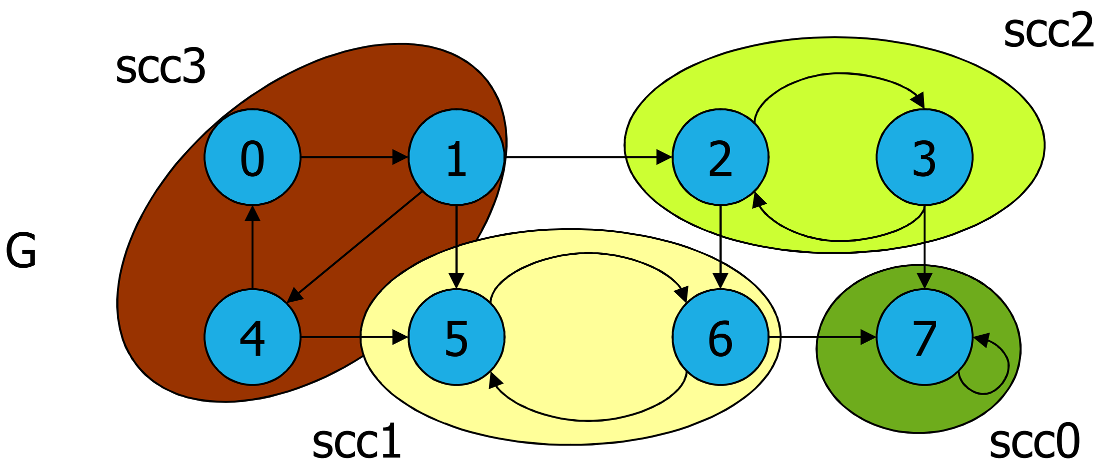
# Visita DFS del grafo trasposto $G^T$



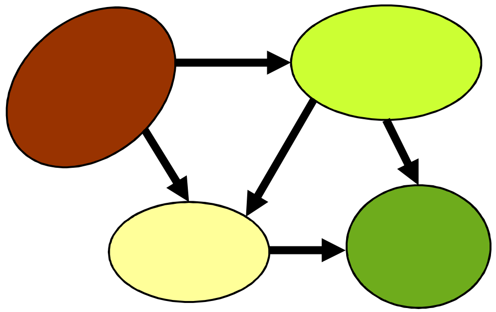

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| postR | 1 | 4 | 0 | 3 | 2 | 6 | 5 | 7 |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Visita DFS del grafo secondo tempi decrescenti di fine elaborazione del grafo trasposto $G^T$ (percorrimento in discesa di `postR`)

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



Kernel DAG



```
void SCCdfsR(Graph G,int w,int *scc,int *time0,int time1,int *post) {
  link t;
  scc[w] = time1;
  for (t = G->ladj[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 %s in scc %d\n",STsearchByIndex(G->tab,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

# Esercizi di teoria


- 10. Visite dei grafi e applicazioni
  - 10.4 Componenti connesse
  - 10.5 Componenti fortemente connesse
  - 10.6 Punti di articolazione
  - 10.7 Ordinamento topologico dei DAG