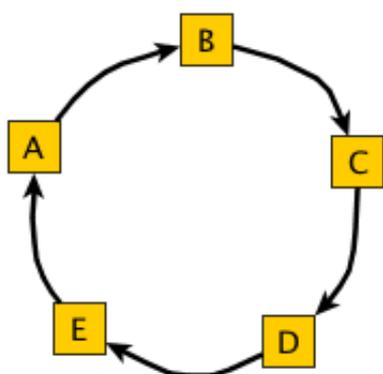


# Progettazione di algoritmi

## Discussione dell'esercizio [vincoli]

Prima di tutto rappresentiamo il problema con un grafo  $G$ : i nodi sono le  $n$  lavorazioni  $L_1, L_2, \dots, L_n$ , e tra due nodi  $L_h, L_k$  c'è un arco  $(L_h, L_k)$  se nella lista dei vincoli di  $L_k$  c'è  $L_h$ , cioè l'arco  $L_h \rightarrow L_k$  significa che  $L_h$  deve essere eseguita prima di  $L_k$ . Quindi il grafo  $G$  è diretto. Intuitivamente, un ordine fattibile di esecuzione delle lavorazioni esiste se e solo se  $G$  non ha cicli (diretti). Infatti se  $G$  ha un ciclo è impossibile rispettare tutti i vincoli dati dagli archi del ciclo.



Come nella figura qui sopra:  $A$  deve essere eseguita prima di  $B$  che deve essere eseguita prima di  $C$  che deve essere eseguita prima di  $D$  che deve essere eseguita prima di  $E$  che deve essere eseguita prima di  $A$ !

Assumiamo invece che  $G$  sia aciclico. Per prima cosa osserviamo che vale la seguente proprietà:

**Se un grafo è aciclico, esiste almeno un nodo che non ha archi entranti.**

Se così non fosse potremmo partire da un nodo  $v_1$  e prendere un suo arco entrante che esce da un nodo  $v_2$ , poi prendere un arco entrante in  $v_2$  e uscente da un nodo  $v_3$  diverso dai primi due perché il grafo è aciclico, anche  $v_3$  ha un arco entrante che deve uscire, per le stesse ragioni, da un quarto nodo distinto  $v_4$  e procedendo in questo modo si arriverebbe all' $n$ -esimo nodo ma quest'ultimo ha un arco entrante che necessariamente dovrebbe uscire da uno dei nodi già considerati chiudendo un ciclo che non può esistere, contraddizione.

Grazie a questa proprietà possiamo costruire un ordine come segue. Come primo nodo scegliamo un nodo  $v_1$  senza archi entranti (cioè, una lavorazione senza vincoli). Eliminando  $v_1$  dal grafo rimaniamo con un grafo aciclico e da questo scegliamo un nodo  $v_2$  senza archi entranti (cioè, una lavorazione o senza vincoli o che aveva come unico vincolo  $v_1$ ).

Eliminiamo  $v_2$  e otteniamo un grafo ancora aciclico e da questo scegliamo  $v_3$  un nodo senza archi entranti (cioè, una lavorazione o senza vincoli o che aveva come vincoli solo  $v_1$  o  $v_2$ ).

Possiamo procedere in questo modo fino all'ultimo nodo.

```
ORD(G: grafo aciclico)
  L <- lista vuota
  WHILE c'è almeno un nodo in G DO
    trova un nodo v senza archi entranti e rimuovilo da G
    L.append(v)
  RETURN L
```

Chiaramente quest'ordine rispetta tutti i vincoli perché non ci possono essere archi da  $v_h$  a  $v_k$  con  $k < h$ , (cioè, archi con orientazione contraria all'ordinamento).

Un grafo diretto aciclico è anche chiamato brevemente *DAG* (da *Directed Acyclic Graph*). Un ordine dei nodi di un DAG che rispetta l'orientazione degli archi (cioè, se c'è un arco  $(u, v)$  allora  $u$  deve venire prima di  $v$  nell'ordine), si chiama **ordine topologico** (*topological sort*). L'algoritmo precedente trova sempre un ordine topologico di un DAG ma l'implementazione diretta di tale algoritmo è piuttosto inefficiente. Infatti, richiede  $n$  passate e in ognuna deve trovare un nodo senza archi entranti e rimuoverlo. Ogni passata può richiedere tempo  $O(n + m)$  e quindi il tempo totale potrebbe essere  $O(n(n + m))$ . Non è proprio banale rendere questo algoritmo efficiente. Fra poco vedremo un altro algoritmo basato sulla DFS che è efficiente. Ma prima dobbiamo vedere le proprietà della DFS su grafi diretti.

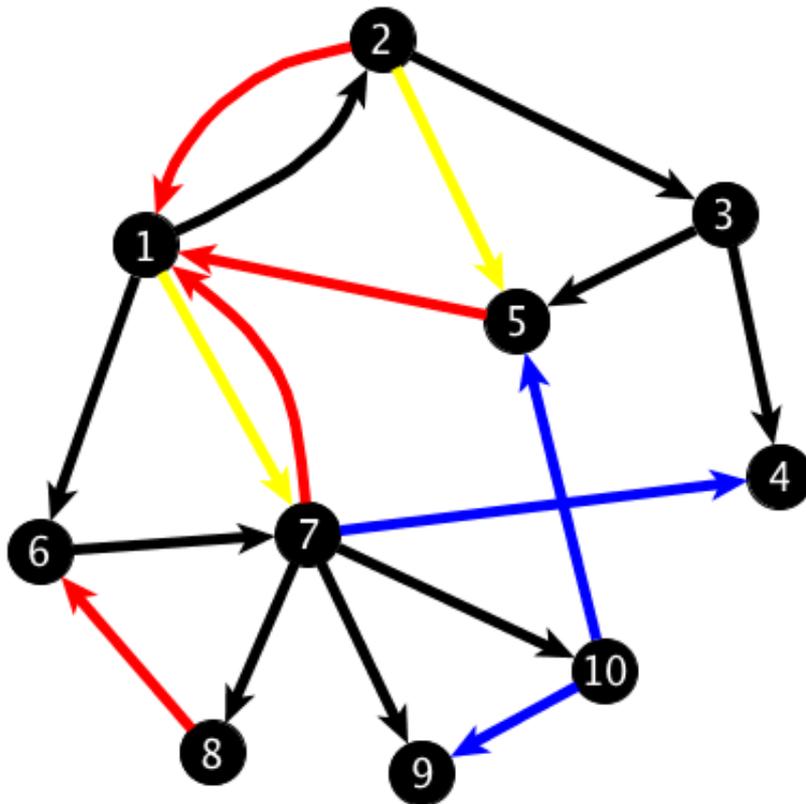
## Classificazione degli archi in una DFS

Se un grafo  $G$  non è diretto, abbiamo visto che dopo una DFS un arco di  $G$  può essere solo di due tipi: o è un arco dell'albero della DFS o è un arco all'indietro, cioè un arco da un nodo verso un antenato di quel nodo nell'albero della DFS. Ma cosa possiamo dire se il grafo è diretto? Consideriamo un qualsiasi arco  $(x, y)$  non appartenente all'albero della DFS. Quello che abbiamo dedotto circa gli intervalli di visita vale anche per grafi diretti. Quindi sono possibili solamente i seguenti casi.

- Gli intervalli di  $x$  e  $y$  sono disgiunti: in questo caso non può essere  $t(x) < t(y)$  perché l'arco  $(x, y)$  avrebbe forzato la visita di  $y$  durante la visita da  $x$  e i due intervalli non sarebbe stati disgiunti. Però può tranquillamente essere  $t(y) < t(x)$ , cioè l'arco  $(x, y)$  è tra due nodi che non hanno rapporti di discendenza e va da quello più giovane a quello più vecchio. Un tale tipo di arco (che non può esistere in grafi non diretti) è detto **arco di attraversamento** (in inglese *cross edge*).
- L'intervallo di  $x$  è contenuto nell'intervallo di  $y$ : l'arco va da un nodo  $x$  a un suo antenato  $y$  e quindi è un **arco all'indietro**.
- L'intervallo di  $x$  contiene l'intervallo di  $y$ : l'arco va da un nodo  $x$  a un suo discendente  $y$ . Questo vuol dire che il nodo  $y$  è stato visitato durante la DFS da  $x$  ma seguendo un cammino diverso dal semplice arco  $(x, y)$ . Questo tipo di arco è detto **arco in avanti** (in inglese *forward edge*).

Nella figura qui sotto i nodi sono numerati in ordine di visita, gli archi neri sono quelli

dell'albero della DFS, quelli rossi sono archi all'indietro, quelli gialli sono archi in avanti e quelli blu sono archi di attraversamento.



## Ordine topologico

Effettuiamo una DFS su un DAG  $G$ . Se la DFS da  $v$  termina dopo la DFS da  $w$ , siamo certi che non ci può essere un arco da  $w$  a  $v$ . Infatti, se ci fosse sarebbe un arco all'indietro ma in un DAG non essendoci cicli non ci possono essere archi all'indietro. Allora possiamo ottenere un ordinamento topologico di un DAG semplicemente ordinando i nodi per tempi di fine visita decrescenti. Quindi possiamo costruire l'ordine inserendo un nodo  $v$  in testa alla lista che mantiene l'ordine non appena la DFS da  $v$  termina.

```

ORDTOP(G: DAG)
  L <- lista vuota
  VIS: array di bool, inizializzato a false
  FOR ogni nodo v di G DO
    IF NOT VIS[v] THEN
      DFS_ORD(G, v, VIS, L)
  RETURN L

DFS_ORD(G: DAG, v: nodo, VIS: array, L: lista)
  VIS[v] <- true
  FOR ogni adiacente w di v DO
    IF NOT VIS[w] THEN
      DFS_ORD(G, w, VIS, L)
  L.add_head(v)

```

Chiaramente l'algoritmo ha la stessa complessità della DFS e quindi  $O(n + m)$ .

## Cicli

Se il grafo non è diretto (e connesso), la presenza di un qualsiasi arco all'indietro indica l'esistenza di un ciclo. E se non ci sono archi all'indietro il grafo è aciclico perchè coincide con l'albero della DFS.

Lo stesso vale per grafi diretti, cioè il grafo ha un ciclo se e solo se c'è almeno un arco all'indietro. Per vederlo supponiamo che  $u_0, u_1, \dots, u_k$  sia un ciclo (orientato e semplice) di  $G$ , quindi  $u_0 = u_k$ . Supponiamo senza perdita di generalità che  $u_0$  sia il primo nodo del ciclo ad essere visitato dalla DFS. Siccome tutti gli altri nodi del ciclo saranno visitati durante la DFS da  $u_0$ , l'arco  $(u_{k-1}, u_0)$  è necessariamente un arco all'indietro. Dobbiamo ora far vedere che se c'è un arco all'indietro, c'è un ciclo. Sia  $(w, v)$  un arco all'indietro. Siccome  $v$  è un antenato di  $w$ , nell'albero della DFS c'è un cammino (orientato e semplice) che va da  $v$  a  $w$  e quindi l'arco  $(w, v)$  chiude tale cammino in un ciclo.

Vediamo nel dettaglio come determinare se un grafo  $G$  ha dei cicli e in caso affermativo trovare uno di questi. Nel caso di grafi non diretti abbiamo detto che è sufficiente trovare un arco all'indietro. Però nel cercare un arco all'indietro da un certo nodo  $v$  bisogna prestare attenzione a non confonderlo con l'arco che collega  $v$  al genitore (cioè l'arco che ha permesso di scoprire  $v$ ). Per recuperare il ciclo dopo averlo trovato possiamo memorizzare, durante la DFS, l'albero di visita. Per rappresentare un albero radicato (o una arborescenza) risulta spesso utile una semplice struttura chiamata **vettore dei padri** che consiste in un array di  $n$  elementi  $P$  tale che  $P[v]$  contiene il padre del nodo  $v$  nell'albero e se  $u$  è la radice  $P[u] = u$ . Assumiamo che i nodi siano identificati dagli interi da 1 a  $n$ .

```

P: vettore dei padri inizializzato con 0

DFS_CYCLE(G: grafo, v: nodo, u: node, P: vettore dei padri)
  P[v] <- -u /* il valore negativo indica che la visita è iniziata ma non
terminata */
  FOR ogni adiacente w di v DO
    IF P[w] = 0 THEN
      z <- DFS_CYCLE(G, w, v, P)
      IF z <> 0 THEN /* un ciclo è già stato trovato */
        P[v] <- -P[v]
        RETURN z
      ELSE IF P[w] < 0 AND w <> u THEN /* trovato ciclo */
        P[w] <- 0
        P[v] <- u
        RETURN v
  P[v] = u /* visita da u terminata */
  RETURN 0 /* senza aver trovato un ciclo */

```

La chiamata sarà  $DFS\_CYCLE(G, u, u, P)$ , dove  $u$  è un qualsiasi nodo. Se ritorna  $0$  significa che non ci sono cicli. Altrimenti ritorna l'indice  $v$  dell'ultimo nodo del ciclo rispetto all'albero della DFS, mentre il primo nodo  $w$  è marcato con  $P[w] = 0$ . La lista dei nodi del ciclo si ottiene con il seguente codice:

```

L <- lista vuota
w <- v /* v è il nodo ritornato da DFS_CYCLE */
DO
  L.append(w)
  w <- P[w]
WHILE w > 0

```

Si osservi che l'algoritmo  $DFS\_CYCLE$  non costa più della DFS. Inoltre, nel caso di un grafo non diretto, può essere molto più efficiente perché termina sempre in  $O(n)$ . Infatti, se il grafo è aciclico la DFS stessa impiega  $O(n)$  perché il grafo è un albero che ha solamente  $n - 1$  archi. Se invece il grafo ha almeno un ciclo, l'algoritmo termina non appena trova un arco all'indietro. Al più saranno visitati tutti gli  $n - 2$  archi dell'albero della DFS prima di incontrare un tale arco (dato che un qualsiasi arco o appartiene all'albero o è un arco all'indietro). Quindi il costo dell'algoritmo è  $O(n)$ .

## Esercizio [gradi]

Il primo algoritmo  $ORD$  che abbiamo visto per l'ordinamento topologico può essere implementato in modo che abbia complessità  $O(n + m)$ . Assumendo che il DAG sia rappresentato tramite liste di adiacenza (cioè, per ogni nodo è data la lista degli adiacenti uscenti), descrivere una implementazione così efficiente dell'algoritmo.