Progettazione di Algoritmi **Anno Accademico 2023–2024**

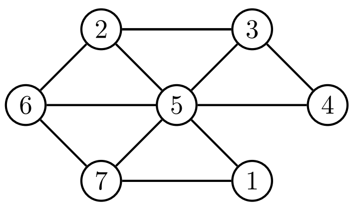
Note per la Lezione 24

Ugo Vaccaro

Parliamo di <u>Cicli in grafi.</u> Ricordiamo che un ciclo in un grafo $G = (V, E)$ è una sequenza di nodi $v_1, v_2, \ldots, v_n$ tale che

$$(v_i, v_{i+1}) \in E \text{ per ogni } i = 1, \ldots, n-1 \text{ e } v_1 = v_n$$

Esempio: il grafo non diretto di sotto

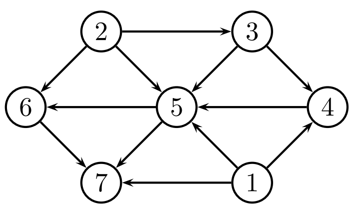


G

contiene vari cicli.

Un grafo G non diretto senza cicli ha una struttura molto semplice: o è un albero (se G è connesso) oppure ogni sua componente connessa è un albero (in tal caso G viene chiamato *una foresta*). La questione è molto più complessa nel caso di grafi diretti

Esempio: il grafo di seguito non è nè un albero, nè ha cicli!



G

Definizione: Un grafo diretto senza cicli viene chiamato in gergo DAG (*directed acyclic graph*)

Strutture di tipo DAG hanno molte applicazioni. Potremmo avere, ad esempio, "compiti" $c_1, c_2, \ldots, c_n$ da eseguire con la condizione che per certe coppie (c_i, c_j) di tali compiti è necessario che il compito c_i debba essere eseguito *prima* di c_j

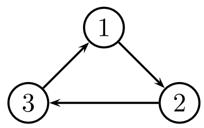
Esempi:
- Propedeuticità tra corsi: il corso c_i deve essere seguito prima di c_j
- Compilazione di moduli: il modulo c_i deve essere compilato prima di c_j
- Esecuzione in pipeline di job: il job c_i deve essere eseguito prima di c_j

$$\vdots$$

Possiamo rappresentare tali vincoli sull'ordine di esecuzione dei compiti mediante un grafo diretto $G = (V, E)$ in cui

- $V = \{c_1, \ldots, c_n\}$
- $(c_i, c_j) \in E$ se il compito c_i deve essere eseguito prima del compito c_j

E che c'entrano i DAG? Supponiamo che il grafo $G = (V, E)$ che rappresenta i vincoli sull'ordine di esecuzione dei compiti **non** sia un DAG. Ciò vuol dire che in G esiste un ciclo diretto, ad esempio:



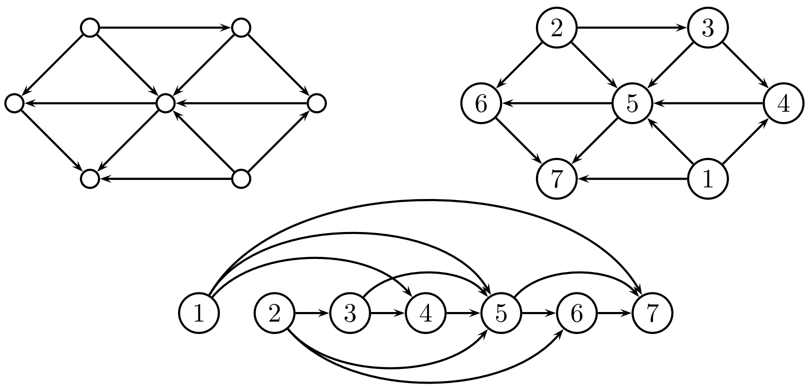
Dal punto di vista dei vincoli sull'esecuzione dei compiti, ciòvorrebbe dire che il compito 1 deve essere eseguito prima del compito 2, il compito 2 deve essere esguito prima di 3, *ma il compito 3 deve essere eseguito prima del compito 1!*, il che è ovviamente impossibile.

Quindi se il grafo diretto che descrive i vincoli sull'ordine di esecuzione dei compiti **non** è un DAG allora non vi è alcun modo di esguire tutti i compiti rispettando i vincoli sul loro ordine di esecuzione relativo.

Nascono quindi due interessanti problemi algoritmici:

1. Dato un grafo diretto $G = (V, E)$ scoprire se esso è un DAG
2. Dato un DAG descrivente i vincoli sull'ordine di esecuzione di certi compiti, determinare una numerazione $n(v_1), n(v_2), \ldots, n(v_n)$ dei compiti in modo tale che per ogni arco diretto (v_i, v_j) abbiamo che $n(v_i) < n(v_j)$ (ovvero seguendo l'ordine naturale della numerazione il compito v_i viene correttamente eseguito prima del compito v_j)

Nell'esempio di sotto, il grafo G a sinistra è un DAG, a destra compare all'interno di ciascun nodo la numerazione $n(\cdot)$ di cui abbiamo parlato, e di sotto un modo alternativo di rappresentare lo stesso grafo G.

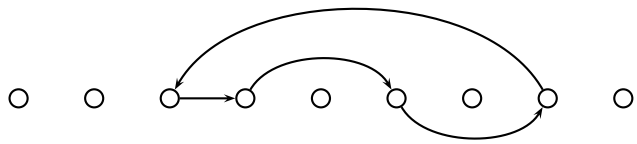


La numerazione $n(v)$ prende il nome di Ordinamento Topologico. Detto in altri termini, un ordinamento topologico di un grafo diretto G è una assegnazione di numeri ai vertici in modo tale che gli archi di G vanno solo da vertici con "numeri inferiori" a vertici con "numeri superiori".

'E possibile almeno trovare un ordinamento topologico per *ogni* grafo diretto? No.

Fatto 3. Se G ammette un ordinamento topologico allora G è necessariamente senza cicli.

Proviamolo. Ricordiamo innanzitutto che se G ammette un ordinamento topologico allora gli archi di G vanno solo da vertici con numeri $n(\cdot)$ bassi a vertici con numeri $n(\cdot)$ alti. Se in G esistesse un ciclo, esso deve per forza partire da un nodo e ritornarvi, tipo così:



il che contraddice chiaramente il fatto che G abbia un ordinamento topologico, visto che c'è un arco che và da un vertice con $n(\cdot)$ *alto* ad un vertice con numero $n(\cdot)$ *basso*.

È possibile trovare un ordinamento topologico per *ogni* grafo diretto aciclico? Si.

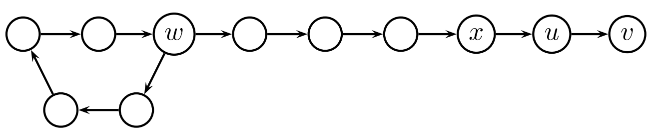
Proviamo innanzitutto il seguente utile risultato.

Fatto 4. Se G è un DAG allora ha un vertice senza archi entranti.

Prova: Supponiamo (per assurdo) che ogni vertice di G abbia un arco entrante. Consideriamo un generico vertice v e seguiamo *all'indietro* gli archi che entrano in v. Poichè v ha almeno un arco (u, v) entrante in esso possiamo andare in u. Poichè u ha almeno un arco (x, u) entrante in esso possiamo andare in x, e così via per sempre...

Ma il grafo è fatto da un numero finito di vertici, quindi prima o poi ripassiamo due volte in uno stesso vertice w. La sequenza di vertici tra due successive visite di w è chiaramente un ciclo, contraddicendo il fatto che G sia un DAG

La figura di sotto rappresenta in maniera grafica il ragionamento di sopra fatto.



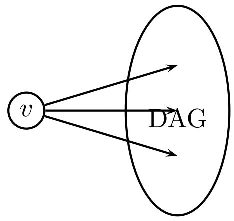
Possiamo qundi provare il risultato che cercavamo.

Fatto 5: Se $G = (V, E)$ è un DAG allora esso ha un ordinamento topologico.

Prova per induzione su $k = |V|$. Se $k = 1$ non v'è nulla da provare.

Dato un DAG G con $k > 1$ vertici, troviamo in G un vertice v senza archi entranti (sappiamo che un tale vertice esiste). Il grafo $G - \{v\}$ è ancora un DAG (se tolgo vertici non posso creare cicli se prima non c'erano). Per ipotesi induttiva $G - \{v\}$ ha un ordinamento topologico. Assegniamo a v un numero $n(\cdot)$ *minore* di tutti i numeri dell'ordinamento topologico di $G - \{v\}$. La numerazione così ottenuta è chiaramente un valido ordinamento topologico per l'intero grafo G in quanto v non ha archi entranti.

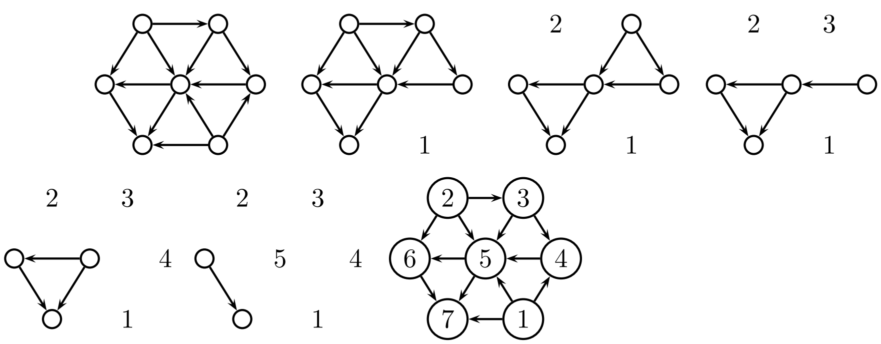
La figura di seguito illustra l'idea dell'induzione:



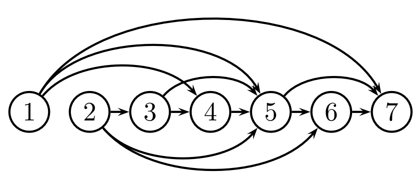
L'algoritmo per il calcolo dell'ordinamento topologico di un DAG G è presentato di seguito (dove la variabile i è da intendersi come variabile globale).

`TopSort`(G)
$i \leftarrow 1$
`Trova un nodo` v `senza archi entranti e assegna` $n(v) \leftarrow i$
`Cancella` v `da` G
$i \leftarrow i + 1$
`Ricorsivamente calcola un ordinamento topologico di` $G - \{v\}$

Vediamo un esempio di esecuzione dell'algoritmo.



Un altro equivalente modo per rappresentare lo stesso grafo è il seguente:



Ricordiamo ora che in un grafo diretto $G = (V, E)$ i nodi $u, v \in V$ sono mutualmente raggiungibili se e solo se esiste un cammino diretto da u e v ed un cammino diretto da v ad u. Il grafo diretto G è fortemente connesso se e solo se *ogni coppia* dei suoi nodi è mutualmente raggiungibile, ovvero se per ogni sua coppia di nodi u, v, esiste un cammino diretto da u e v ed un cammino diretto da v ad u. Nella lezione scorsa abbiamo osservato che vale la seguente proprietà:

Fatto 1. Sia s un qualsiasi nodo di G. G è fortemente connesso $\Leftrightarrow$ ogni nodo in G è raggiungibile da s ed inoltre s è raggiungibile da un qualsiasi nodo di G.

Come si fà a sapere se un grafo $G = (V, E)$ è fortemente connesso? Il seguente algoritmo fornisce una risposta.

```
AlgoritmoFC(G)
1.  Scegli un nodo arbitrario s in G
2.  Esegui BFS(s) in G (o DFS(s) se vi piace di più)
3.  Esegui BFS(s) in G^R
4.  Ritorna True se e solo se tutti i nodi di G sono stati raggiunti in entrambe le
esecuzioni di BFS
```

La BFS(s) in **2.** ci dice se *tutti* i nodi di G sono raggiungibili da s, la BFS(s) in **3.** ci dice se s è raggiungibile da *tutti* i nodi di G. Il Fatto 1 precedente ci assicura che G è fortemente connesso se e solo se entrambe le condizioni precedenti sono vere. La complessità dell'algoritmo è chiaramente $\Theta(n+m)$, dove come al solito $n = |V|, m = |E|$.

Parliamo ora di componenti fortementi connesse di un grafo diretto $G = (V, E)$. In analogia con il caso di grafi non diretti, possiamo definire la *componente fortemente connessa* contenente il generico vertice s di un grafo diretto G come l'insieme dei nodi v tali che s e v sono mutualmente raggiungibili.

Sempre in perfetta analogia con il caso di grafi non diretti, possiamo mostrare il seguente fatto:

Fatto 2. Per ogni coppia di nodi u e v nel grafo diretto G, le loro componenti fortemente connesse o sono uguali o sono disgiunte (cioè non hanno alcun nodo in comune)

E se volessimo calcolare la componente fortementa connessa contenente un dato vertice s? L'algoritmo prima visto ci permette di calcolarla.

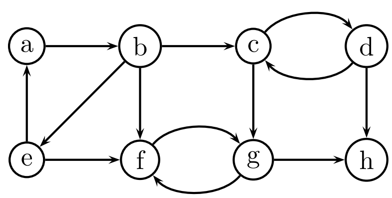
```
AlgoritmoCF(s, G)
1.  Esegui DFS(s) in G
2.  Esegui DFS(s) in G^R
3.  Ritorna l'insieme F dei nodi che appaiono sia nell'albero DFS prodotto da DFS(s) in G
    che DFS(s) in G^R
```

Il fatto che i nodi in F appaiono nell'albero DFS prodotto da DFS(s) in G ci assicura che i nodi in F sono raggiungibili a partire da s, mentre Il fatto che i nodi in F appaiono nell'albero DFS prodotto da DFS(s) in G^R ci assicura che da tutti i nodi in F è possibile raggiungere s, *ergo* F è la componente connessa di s.

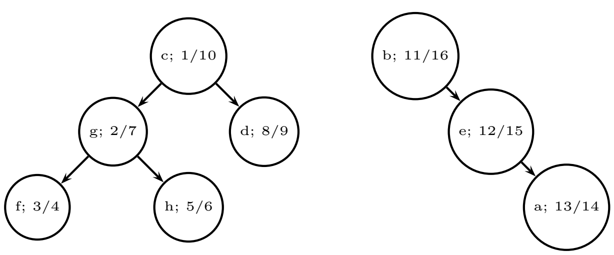
E se volessimo calcolare tutte le componenti fortemente connesse di un grafo $G = (V, E)$? Il seguente algoritmo lo fà:

```
AlgoritmoTutte_CF(G)
FOR ciascun vertice u ∈ V marca u ‘‘non Esplorato’’
FOR ciascun vertice u ∈ V
    IF (u ‘‘non Esplorato’’)
    THEN esegui DFS(u) e per u calcola l’istante i(u) in cui u viene visitato per la prima
      volta e l’istante f(u) in cui la ricorsione su u termina
FOR ciascun vertice u ∈ V marca u ‘‘non Esplorato’’
FOR ciascun vertice u ∈ V nell’ordine degli f(u) decrescenti
       IF (u ‘‘non Esplorato’’)
       THEN esegui DFS(u) nel grafo G^R
RETURN gli alberi prodotti dalla DFS nel grafo G^R come componenti fortementi connesse di G
```

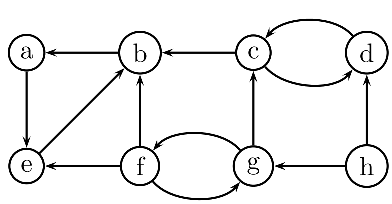
Vediamo un esempio di esecuzione dell’algoritmo sul grafo G di sotto.



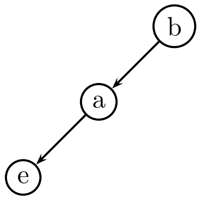
Supponiamo ora di eseguire le DFS a partire dal nodo c. Otterremmo gli alberi seguenti, in cui i numeri all’interno di ogni nodo u corrispondono ii valori $i(u)$ e $f(u)$.



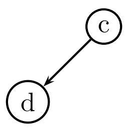
A questo punto calcoliamo G^R, invertendo la direzione degli archi in G, per ottenere



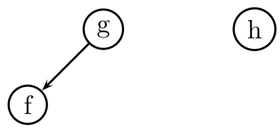
ed eseguiamo la DFS a partire dal nodo u con il numero $f(u)$ più grande di tutti. Tale numero è 16, e corrisponde al nodo b. Per cui otterremo il primo albero pari a:



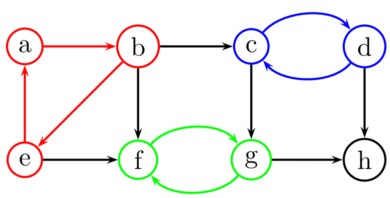
Usando l'ordine dei numeri $f(\cdot)$ decrescenti, il prossimo vertice da cui far partire la DFS è il vertice c. Per cui otterremo il secondo albero



ed infine, gli ultimi due alberi



I nodi del primo albero corrispondono alla componente connessa di G con i vertici in rosso, i nodi del secondo albero corrispondono alla componente connessa di G con i vertici in blu, i nodi del terzo albero corrispondono alla componente connessa di G con i vertici in verde, il nodo del quarto albero corrispondono alla componente connessa di G con i vertici in nero.



Diamo una prova del perchè l'algoritmo produce le componenti fortementi connesse (c.f.c.) del generico grafo input $G = (V, E)$. Siano $C_1, \ldots, C_k$ le c.f.c di G. Ricordiamo che, per definizione di c.f.c., queste sono disgiunte tra di loro e se si "esce" con un arco da una di esse, non è possibile rientrarvi.

Definiamo un nuovo grafo $G' = (V', E')$, dove $V' = \{C_1, \ldots, C_k\}$ e c'è un arco diretto (C_i, C_j) dal "vertice" C_i al "vertice" C_j se e solo se, *nel grafo* $G = (V, E)$ esiste un qualche arco della forma (u, v), dove $u \in C_i$ e $v \in C_j$.

Proviamo che $G' = (V', E')$ è un DAG, ovvero non ha cicli. Se per assurdo avesse un ciclo, ad esempio $(C_{i_1}, C_{i_2}), (C_{i_2}, C_{i_3}), \ldots, (C_{i_s}, C_{i_1})$, questo vorrebbe dire che, *nel grafo* $G = (V, E)$, *da ogni* vertice di C_{i_1} è possibile raggiungere il vertice ui_1 di C_{i_1} che è connesso ad un qualche vertice ui_2 di C_{i_2}, da cui poter raggungere ogni vertice di C_{i_2} (ricordiamo che i C_{i_j} sono componenti fortemente connesse di G e quindi da ogni vertice nella componente è possibile raggiungere ogni altro vertice della stessa componente mediante un cammino diretto). Iterando, *da ogni* vertice di C_{i_2} è possibile raggiungere il vertice v_{i_2} di C_{i_2} che è connesso ad un qualche vertice ui_3 di C_{i_3}, da cui poter raggungere ogni vertice di C_{i_3}. In questo modo, potremmo raggiungere ogni vertice di C_{i_s} da cui è possibile rggiungere ogni vertice di C_{i_1}. Cosa abbiamo ottenuto? Che da ogni vertice ogni C_{i_j} è possibile raggiungere ogni vertice di ogni altra componente C_{i_ℓ}. Detto in altri termini, tutti i vertici in $\cup_j C_{i_j}$

sono raggiungibili tra di loro, ovvero l'insieme dei vertici in $\cup_j C_{i_j}$ sono un'unica componente fortemente connessa, contro l'ipotesi!

Avendo provato che $G' = (V', E')$ è un DAG, sappiamo che esiste un C_{i_j} *senza archi entranti*, ed un C_{i_ℓ} senza archi uscenti (ricordiamocelo!)

Supponiamo ora di eseguire l'algoritmo `AlgoritmoTutte_CF`(G) sul grafo G. Per ogni c.f.c C_i, per $i = 1, \ldots, k$, definiamo $f(C_i) = \max\{f(u) : u \in C_i\}$ (i valori $f(\cdot)$ sono calcolati nell'algoritmo `AlgoritmoTutte_CF`(G). Proviamo ora che se esiste un arco da qualche nodo di una c.f.c C_i ad un nodo di un'altra c.f.c C_j, (ovvero esiste l'arco (C_i, C_j) in G') allora $f(C_i) > f(C_j)$. La prova è semplice. Distinguiamo due casi:

- Quando l'algoritmo `AlgoritmoTutte_CF`(G) visita per la *prima volta* qualche nodo u di C_i, *nessun* nodo di C_j è stato ancora visitato. Sotto questa ipotesi, la corrispondente ricerca DFS eseguita dall'algoritmo `AlgoritmoTutte_CF`(G) procederà ad esplorare tutti i nodi raggiungibili da u (in particolare, anche in nodi in C_j) e terminerà le ricorsioni dopo aver teminato le ricorsioni relative alle visite che iniziano dai nodi in C_j. Ciò comporta che il valore $f(u)$, che è pari all'istante in cui la ricorsione su u termina, *è più grande* dei corrispondenti valori $f(v)$, $\forall v \in C_j$. Ovvero, $f(C_i) > f(C_j)$.
- Il secondo caso è l'opposto, ovvero quando l'algoritmo `AlgoritmoTutte_CF`(G) visita per la *prima volta* qualche nodo v di C_j, *nessun* nodo di C_i è stato ancora visitato. Per trattare questo caso, ricordiamo che non vi è alcun percorso da nodi in C_j in nodi in C_i (ricordiamo: G' è un DAG, quindi dato che esiste l'arco (C_i, C_j), se fosse possibile da C_j ritornare a C_i si creerebbe un ciclo, in G'!). Visto che non vi sono percorsi da C_j a C_i, quando la ricorsione che parte da v è terminata non abbiamo ancora visitato nulla di C_i, per cui i tempi di fine $f(\cdot)$ dei nodi $u \in C_i$ saranno tutti maggiori dei tempi di fine di ogni nodo in C_j, ovvero $f(C_i) > f(C_j)$.

Come conseguenza di quello che abbiamo appena provato, otteniamo che la componente connessa C con $f(C)$ *massimo non ha archi entranti!*. Infatti, se avesse un arco entrante che parte da qualche altra c.f.c C', allora, per quanto appena provato, varrebbe $f(C') > f(C)$, contro la massimalità di $f(C)$.

Prima di provare la correttezza dell'algoritmo `AlgoritmoTutte_CF`(G), notiamo l'ovvio fatto che le c.f.c di G sono le stesse c.f.c di G^R. Infatti, se C è una c.f.c. di G, allora ciò vuol dire che $\forall u, v \in$ esiste un percorso diretto da u a v ed uno da v ad u. Chiaramente, lo stesso vale in G^R.

Ritorniamo ora all'algoritmo `AlgoritmoTutte_CF`(G). La seconda parte dell'algoritmo consiste nell'effettuare delle visite DFS in G^R, a partire dal nodo u che ha valore $f(u)$ massimo. Questo nodo appartiene sicuramente ad una componente connessa C di G che non ha archi entranti, per quanto prima osservato. Potrebbe avere archi uscenti, ma questi archi in G^R *sono entranti*. Questo vuol dire che quando eseguiamo DFS(u), scopriremo *tutti* i nodi in C in quanto, per definizione di c.f.c, tutti i nodi di C sono raggiungibili da u, e poi ci *fermiamo*, in quanto non è possibile (nel grafo G^R, uscire da C. Di conseguenza, il primo albero prodotto da `AlgoritmoTutte_CF`(G) corrisponde esattamemte ai nodi di C, corretta componente fortemente connessa di G.

Dopo aver visitato tutti i nodi di C l'algoritmo `AlgoritmoTutte_CF`(G) procede con una nuova DFS(v), dove è il nodo con valore $f(v)$ massimo non ancora visitato. Per quanto prima detto, esso appartiene ad una c.f.c D che non ha archi entranti tranne quelli che possono entrare da C, prima considerata. Tutti gli archi uscenti da D vengono, in G^R, trasformati in archi entranti, e gli eventuali archi entranti in D che parte da C vengono trasformati in archi uscenti. La visita DFS(v) esplorerà tutti i nodi di D (in quanto essi sono raggiungibili da v) e *non uscirà* da D in quanto gli unici archi che escono da D vanno verso nodi in C *che sono stati già visitati*. Pertanto, la visita termina in D, ed il secondo albero prodotto da `AlgoritmoTutte_CF`(G) corrisponde esattamemte ai nodi di D, corretta componente fortemente connessa di G, e così via...