Grafi (III parte)

Progettazione di Algoritmi a.a. 2023-24

Matricole congrue a 1

Docente: Annalisa De Bonis

Implementazione di DFS mediante uno stack

```
DFS(s):
1.   Poni Explored[s] = true ed Explored[v] = false per tutti gli altri nodi
2.   Inizializza  S con uno stack contenente s
3.   While S non è vuoto
4.       Metti in u  il nodo  al top di S
5.       If  c'e` un arco (u, v) incidente su u non ancora esaminato then
6.           If Explored[v] = false then
7.              Poni Explored[v] = true
8.              Inserisci v al top di S
9.           Endif
10.      Else  // tutti gli archi incidenti su u sono stati esaminati
11.          Rimuovi il top di S
12.      Endif
13.  Endwhile
```

- Per implementare la linea 6 in modo efficiente possiamo mantenere per ogni vertice u un puntatore al nodo della lista di adiacenza di u corrispondente al prossimo arco (u,v) da scandire.
- Si noti che un nodo u rimane nello stack fino a che non vengono scanditi tutti gli archi incidenti su u.

Analisi di DFS implementata mediante uno stack

Assumiamo G rappresentato con liste di adiacenza

DFS(s):

```
1.   Poni Explored[s] = true ed Explored[v] = false per tutti gli altri nodi   O(n)
2.   Inizializza  S con uno stack contenente s   O(1)
3.   While S non è vuoto
4.       Metti in u  il nodo  al top di S
5.       If  c'e` un arco (u, v) incidente su u non ancora esaminato then
6.          If Explored[v] = false then
7.             Poni Explored[v] = true
8.             Inserisci v al top di S
9.          Endif
10.      Else  // tutti gli archi incidenti suu sono stati esaminati
11.         Rimuovi il top di S
12.      Endif
13.  Endwhile
```

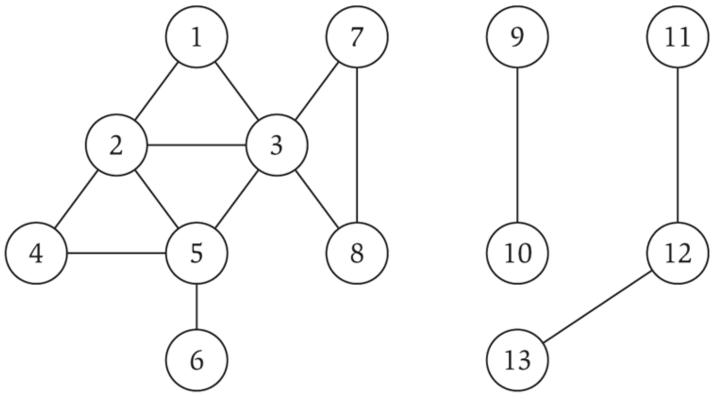
O(n+m)

- Analisi linee 3-13: Il while viene iterato deg(v)+1 volte per ogni nodo v inserito in S: ogni volta che v viene a trovarsi al top dello stack viene esaminato uno dei suoi archi non ancora esaminati oppure se non esiste un tale arco, v viene rimosso dallo stack. Vengono quindi effettuate deg(v) iterazioni del while prima di quella in cui v viene rimosso dallo stack → in totale il while e` iterato un numero di volte pari al piu` a $\sum_{v \in V}(deg(v)+1) = \sum_{v \in V} deg(v) + \sum_{v \in V} 1 \leq 2m+n$
- Se manteniamo traccia del prossimo arco da scandire (vedi slide precedente), la linea 6 richiede tempo O(1). Di conseguenza il corpo del while richiede O(1) per ogni iterazione →tempo totale per tutte le iterazioni O(2m+n)=O(m+n).

Componente connessa

- Componente connessa. Sottoinsieme di vertici tale per ciascuna coppia di vertici u e v esiste un percorso tra u e v

- Componente connessa contenente s. Formata da tutti i nodi raggiungibili da s

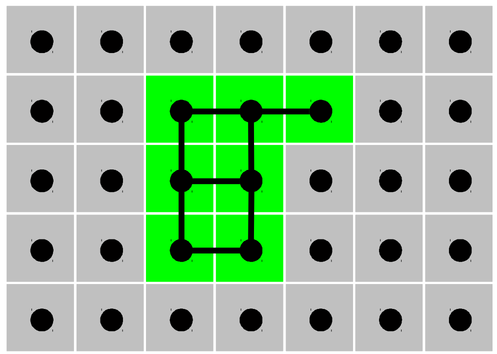


- Componente connessa contenente il nodo 1 è { 1, 2, 3, 4, 5, 6, 7, 8 }.

Flood Fill

- Flood fill. Data un'immagine, cambia il colore dell'area di pixel vicini di colore verde lime in blu.
  - Nodo: pixel.
  - Arco: due pixel vicini di colore verde lime.
  - Area di pixel vicini di colore verde lime: componente connessa di nodi associati a pixel verde lime.

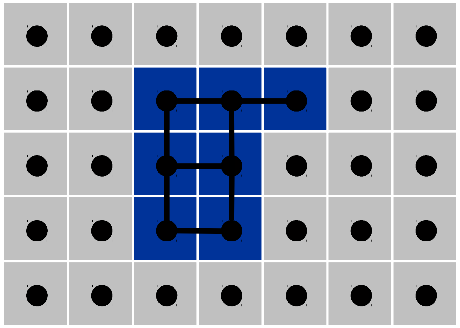
Ricolora l'area verde lime di blue



Flood Fill

- Flood fill. Data un'immagine, cambia il colore dell'area di pixel vicini di colore verde lime in blu.
  - Nodo: pixel.
  - Arco: due pixel vicini di colore verde lime.
  - Area di pixel vicini: componente connessa di pixel di colore verde lime.

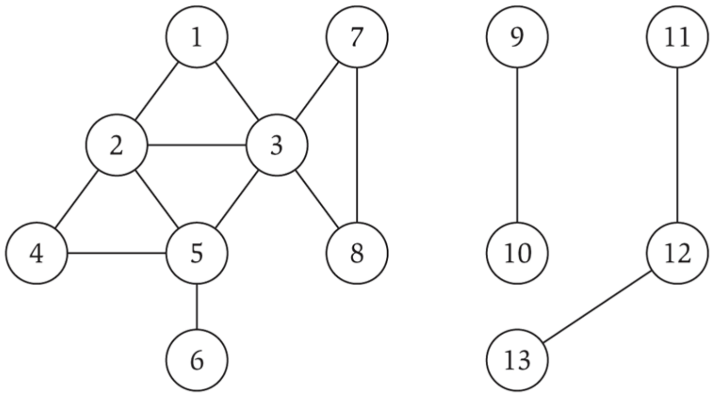
Ricolora l'area verde lime di blue



Componente connessa

- Componente connessa contenente s. Trova tutti i nodi raggiungibili da s
  - Come trovarla. Esegui BFS o DFS utilizzando s come sorgente

- Insieme di tutte le componenti connesse. Trova tutte le componenti connesse
  - Come trovarlo. Fino a quando ci sono nodi che non sono stati scoperti (esplorati), scegli uno di questi nodi ed esegui BFS (o DFS) su u utilizzando questo nodo come sorgente



Esempio: il grafo sottostante ha tre componenti connesse

Insieme di tutte componenti connesse

- Teorema. Per ogni due nodi s e t di un grafo, le loro componenti connesse o sono uguali o disgiunte
- Dim.
- Caso 1. Esiste un percorso tra s e t. In questo caso ogni nodo u raggiungibile da s è anche raggiungibile da t (basta andare da t ad s e da s ad u) e ogni ogni nodo u raggiungibile da t è anche raggiungibile da s (basta andare da s ad t e da t ad u). Ne consegue che un nodo u è nella componente connessa di s **se e solo** se è anche in quella di t e quindi le componenti connesse di s e t sono uguali.
- Caso 2. Non esiste un percorso tra s e t. In questo caso non può esserci un nodo che appartiene sia alla componente connessa di s che a quella di t. Se esistesse un tale nodo v questo sarebbe raggiungibile sia da s che da t e quindi potremmo andare da s a v e poi da v ad t. Ciò contraddice l'ipotesi che non c'è un percorso tra s e t.

Insieme di tutte componenti connesse

- Il teorema precedente implica che le componenti connesse di un grafo sono a due a due disgiunte.

- Algoritmo per trovare l'insieme di tutte le componenti connesse

```
AllComponents(G)
 Per ogni nodo u di G setta discovered[u]=false
 For each node u of G
    If Discovered[u] = false
      BFS(u)
  Endif
 Endfor
```

- BFS modificata in modo tale che nella fase di inizializzazione non vengano settati a False le entrate dell'array Discovered

- Al posto della BFS possiamo usare la DFS e al posto dell'array Discovered l'array Explored

Insieme di tutte componenti connesse: analisi

- Indichiamo con k il numero di componenti connesse
- Indichiamo con n_i e con m_i rispettivamente il numero di nodi e di archi della componente i-esima
- L'esecuzione della visita BFS o DFS sulla componente i-esima richiede tempo $O(n_i+m_i)$
- Il tempo totale richiesto da tutte le visite BFS o DFS e`

$$\sum_{i=1}^{k} O(n_i+m_i)=O\left(\sum_{i=1}^{k}(n_i+m_i)\right)$$

- Poiche' le componenti sono a due a due disgiunte, si ha che

$$\sum_{i=1}^{k}(n_i+m_i)=n+m$$

- e il tempo totale di esecuzione dell'algoritmo che scopre le componenti connesse e` O(n)+O(n+m)=O(n+m)

Insieme di tutte componenti connesse: alcune considerazioni

- Se l'algoritmo utilizza BFS allora BFS deve essere modificata in modo che non resetti a false ogni volta i campi discovered.
- E` possibile modificare AllComponents in modo che assegni a ciascun nodo la componente di cui fa parte. A questo scopo usiamo:
  - contatore delle componenti.
  - array Component t.c. Component[u] = j se u appartiene alla componente j-esima.

  - Esercizio: modificare lo pseudocodice dell'algoritmo AllComponents in modo che assegni a ciascun nodo la componente di cui fa parte. Ricordatevi che occorre modificare anche l'algoritmo di visita invocato da AllComponents.

Visita di grafi direzionati

- Raggiungibilità con direzione. Dato un nodo s, trova tutti i nodi raggiungibili da s.

- Il problema del più corto percorso diretto da s a t. Dati due nodi s e t, qual è la lunghezza del percorso più corto da s a t?

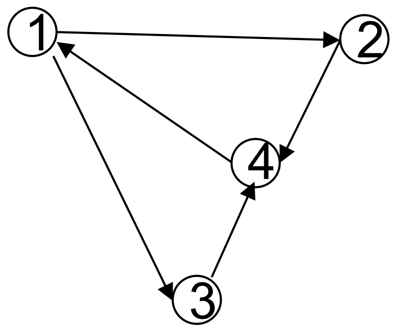
- Visita di un grafo. Le visite BFS e DFS si estendono naturalmente ai grafi direzionati.
  - Quando si esaminano gli archi incidenti su un certo vertice u, si considerano solo quelli uscenti da u.

- Web crawler. Comincia dalla pagina web s. Trova tutte le pagine raggiungibili a partire da s, sia direttamente che indirettamente.

Connettività forte

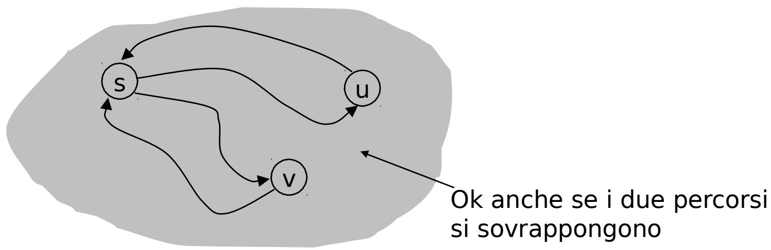
- Def. I nodi u e v sono mutualmente raggiungibili se c'è un percorso da u a v e anche un percorso da v a u.

- Def. Un grafo in cui ogni ogni coppia di nodi è mutualmente raggiungibile si dice fortemente connesso



Connettività forte

- **Lemma**. Sia s un qualsiasi nodo di un grafo direzionato G. G è fortemente connesso se e solo se ogni nodo è raggiungibile da s ed s è raggiungibile da ogni nodo.
- **Dim.** $\Rightarrow$ Segue dalla definizione.
- **Dim.** $\Leftarrow$ Un percorso da u a v si ottiene concatenando il percorso da u ad s con il percorso da s a v. Un percorso da v ad u si ottiene concatenando il percorso da v ad s con il percorso da s ad u.



Algoritmo per la connettività forte

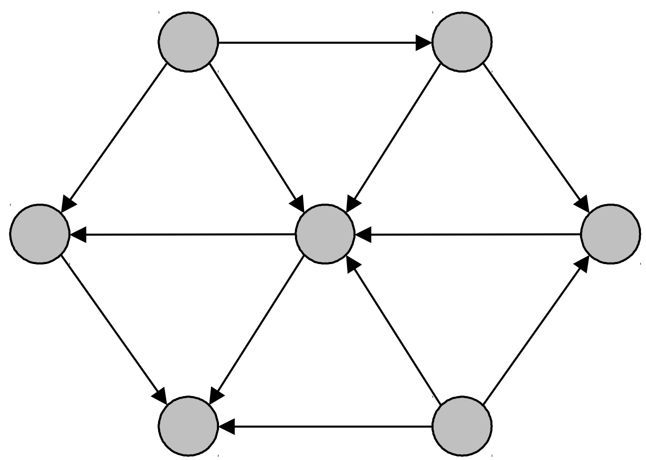
Teorema. Si può determinare se G è fortemente connesso in tempo $O(m + n)$.

Dim.

- Prendi un qualsiasi nodo s.
- Esegui la BFS con sorgente s in G.
- Crea il grafo G^{rev} invertendo la direzione di ogni arco in G
- Esegui la BFS con sorgente s in G^{rev}.
- Restituisci true se e solo se tutti i nodi di G vengono raggiunti in entrambe le esecuzioni della BFS.
- La correttezza segue dal lemma precedente.
  - La prima esecuzione trova i percorsi da s a tutti gli altri nodi
  - La seconda esecuzione trova i percorsi da tutti gli altri nodi ad s perchè avendo invertito gli archi un percorso da s a u è di fatto un percorso da u ad s nel grafo di partenza.

Grafi direzionati aciclici (DAG)

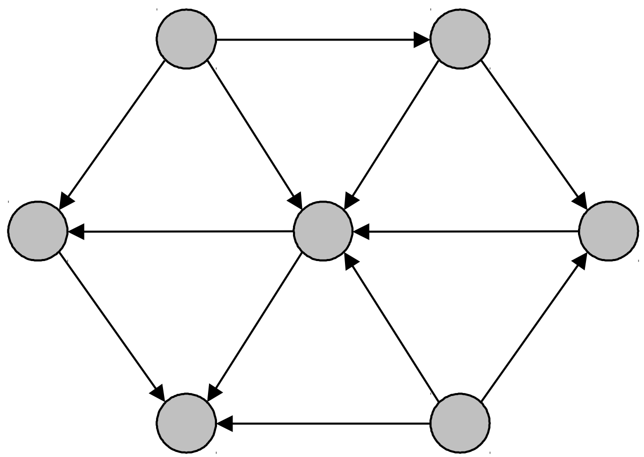
- Def. Un DAG è un grafo direzionato che non contiene cicli direzionati
- Possono essere usati per esprimere vincoli di precedenza o dipendenza: l'arco (v_i, v_j) indica che v_i deve precedere v_j o che v_j dipende da v_i
- Infatti generalmente i grafi usati per esprimere i suddetti vincoli sono privi di clicli
- Esempio. Vincoli di precedenza: grafo delle propedeuticità degli esami



Un DAG G

Grafi direzionati aciclici (DAG)

- Applicazioni
  - Propedeuticita`: il corso v_i deve essere superato prima di sostenere l'esame del corso v_j .
  - Compilazione: il modulo v_i deve essere compilato prima del modulo v_j .
  - Pipeline dell'esecuzione di job: l'output del job v_i serve per determinare l'input del job v_j
  - Pianificazione dello sviluppo di un software: alcuni moduli devono essere scritti prima di altri.

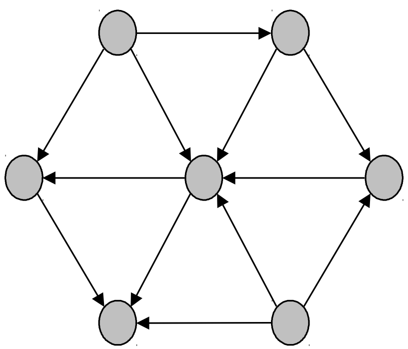


Un DAG G

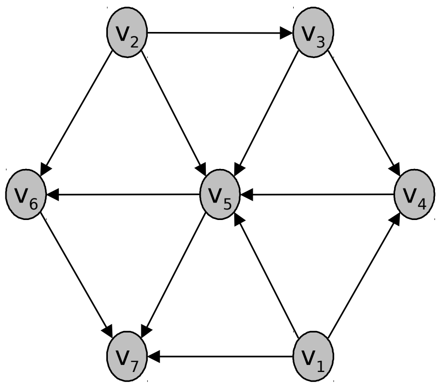
Ordine topologico

- Def. Un ordinamento topologico di un grafo direzionato $G = (V, E)$ è un etichettatura dei suoi nodi $v_1, v_2, \ldots, v_n$ tale che se G contiene l'arco (v_i, v_j) si ha $i < j$. Detto in un altro modo, un ordinamento topologico di G è un ordinamento dei nodi di G tale che se c'è l'arco (u,w) in G, allora il vertice u precede il vertice w nell'ordinamento (tutti gli archi puntano in avanti nell'ordinamento).

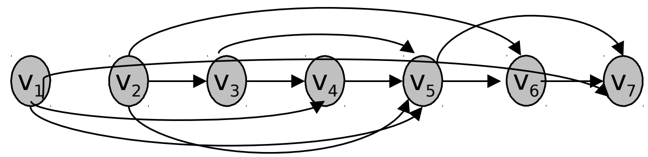
- Esempio. Nel caso in cui un grafo direzionato G rappresenti le propedeuticità degli esami, un ordinamento topologico indica un possibile ordine in cui gli esami possono essere sostenuti dallo studente.



Un DAG G



Un ordinamento topologico di G



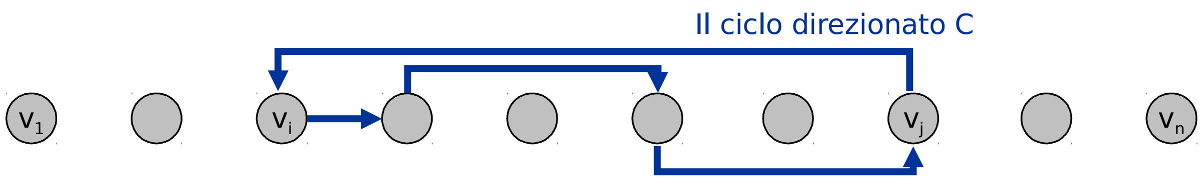
Un modo diverso di ridisegnare G in modo da evidenziare l'ordinamento topologico di G

DAG e ordinamento topologico

Lemma. Se un grafo direzionato G ha un ordinamento topologico allora G è un DAG.

Dim. (per assurdo)

- Supponiamo che G sia un grafo direzionato e che abbia un ordinamento $v_1, \ldots, v_n$. Supponiamo per assurdo che G non sia un DAG ovvero che abbia un ciclo direzionato C. Vediamo cosa accade.
- Consideriamo i nodi che appartengono a C e tra questi sia v_i quello con indice più piccolo e sia v_j il vertice che precede v_i nel ciclo C. Ciò ovviamente implica che (v_j, v_i) è un arco.
- Siccome (v_j, v_i) è un arco e $v_1, \ldots, v_n$ è un ordinamento topologico allora, deve essere $j < i$.
- $j < i$ è impossibile in quanto abbiamo scelto v_i come il vertice di indice più piccolo in C e di conseguenza vale $i < j$. Siamo arrivati ad un assurdo. Cioè una contraddizione al fatto che G contiene un ciclo.



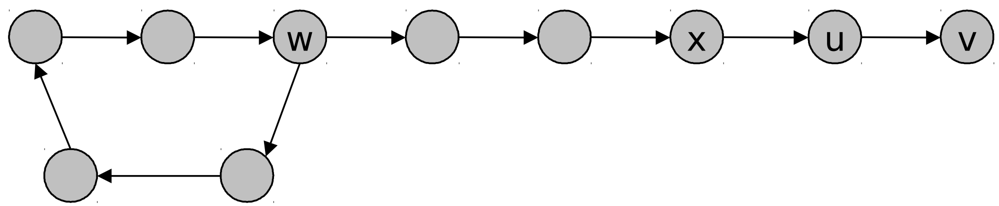
L'ordinamento topologico $v_1, \ldots, v_n$

DAG e ordinamento topologico

- Abbiamo visto che se G ha un ordinamento topologico allora G è un DAG.
- Domanda. é vera anche l'implicazione inversa? Cioè dato un DAG, è sempre possibile trovare un suo ordinamento topologico?
- E se sì, come trovarlo?

Lemma. Se G è un DAG allora G ha un nodo senza archi entranti
Dim. (per assurdo)

- Supponiamo che G sia un DAG e che ogni nodo di G abbia almeno un arco entrante. Vediamo cosa succede.
- Prendiamo un qualsiasi nodo v e cominciamo a seguire gli archi in senso contrario alla loro direzione a partire da v. Possiamo farlo perchè ogni nodo ha un arco entrante: v ha un arco entrante (u,v), il nodo u ha un arco entrante (x,u) e così via.
- Possiamo continuare in questo modo per quante volte vogliamo. Immaginiamo di farlo per n o più volte. Così facendo attraversiamo a ritroso almeno n archi e di conseguenza passiamo per almeno n+1 vertici. Ciò vuol dire che c’è un vertice w che viene incontrato almeno due volte e quindi deve esistere un ciclo direzionato C che comincia e finisce in w



DAG e ordinamento topologico

Lemma. Se G è un DAG, G ha un ordinamento topologico.

Dim. (induzione su n)

- Caso base: vero banalmente se n = 1.
- Passo induttivo: supponiamo asserto del lemma vero per DAG con n≥1 nodi
- Dato un DAG con n+1 > 1 nodi, prendiamo un nodo v senza archi entranti (abbiamo dimostrato che un tale nodo deve esistere).
- G - { v } è un DAG, in quanto cancellare un nodo non introduce clicli nel grafo.
- Poiché G - { v } è un DAG con n nodi allora, per ipotesi induttiva, G - { v } ha un ordinamento topologico.
- Consideriamo l'ordinamento dei nodi di G che si ottiene mettendo v all'inizio dell'ordinamento e aggiungendo gli altri nodi nell'ordine in cui appaiono nell'ordinamento topologico di G - { v }.
- Siccome v non ha archi entranti tutti i suoi archi sono archi uscenti e ovviamente puntano verso nodi di G-{v}. Quello che si ottiene è un ordinamento topologico (tutti gli archi puntano in avanti).

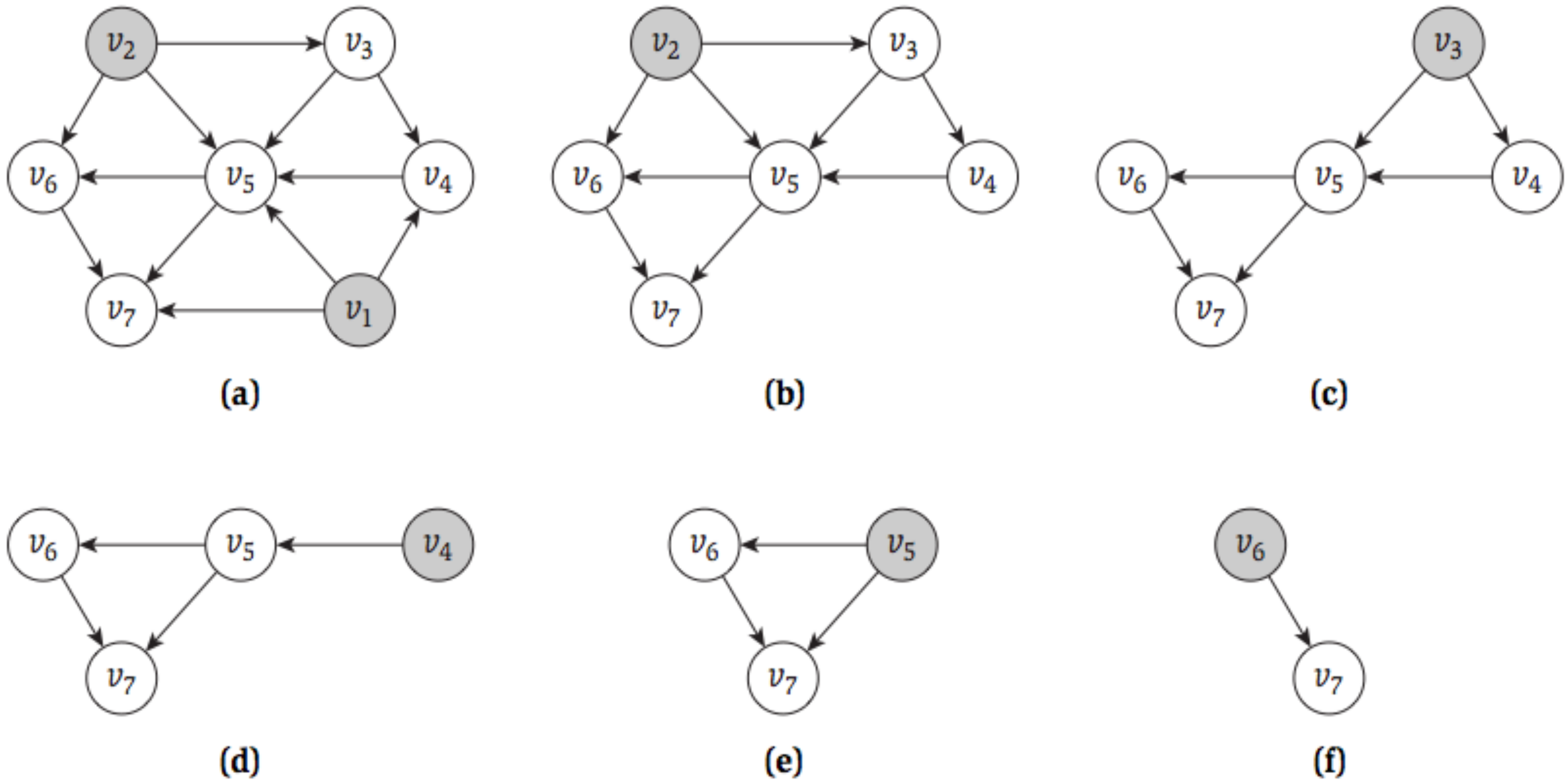
Algoritmo per l'ordinamento topologico

- La dimostrazione per induzione che abbiamo appena visto suggerisce un algoritmo ricorsivo per trovare l' ordinamento topologico di un DAG.

G: DAG

```
TopologicalOrder(G)
    if esiste nodo v senza archi entranti
        cancella v da G in modo da ottenere G-{v}
        L=TopologicalOrder(G-{v})
        aggiungi v all'inizio di L
        return L
    endif
    else     //siccome G è un DAG, l'else è eseguito solo se G vuoto
        return lista vuota
```

Algoritmo per l'ordinamento topologico



Algoritmo per l'ordinamento topologico : analisi dell'algoritmo

1) Trovare un nodo senza archi entranti nell'if richiede $O(n)$ se per ogni nodo viene memorizzato il numero di archi entranti
2) Cancellare un nodo v da G richiede tempo proporzionale al numero di archi uscenti da v che è **al più** deg(v)

- Se consideriamo tutte le n chiamate ricorsive il tempo è $O(n^2)$ per 1) e $O(m)$ per 2). Quindi il tempo di esecuzione è $O(n^2+m)=O(n^2)$

```
TopologicalOrder(G)
    if esiste nodo v senza archi entranti
        cancella v da G in modo da ottenere G-{v}
        L=TopologicalOrder(G-{v})
        aggiungi v all'inizio di L
        return L
    endif
    else
        return lista vuota
```

Algoritmo per l'ordinamento topologico : analisi dell'algoritmo

- Possiamo anche scrivere la relazione di ricorrenza

$$T(n) \leq \begin{cases} c & \text{per } n=1 \\ T(n-1)+c'n & \text{per } n>1 \end{cases}$$

Lavoro ad ogni chiamata ricorsiva è $O(n+deg(v))=O(n)$, dove v è il nodo rimosso da G

che ha soluzione $T(n)=O(n^2)$

Metodo iterativo

$T(n) \leq T(n-1)+c'n \leq T(n-2)+c'(n-1)+c'n \leq T(n-3)+c'(n-2)+c'(n-1)+c'n \leq \ldots \leq$

$T(1) + c'2+\ldots+ c'(n-1)+ c'n \leq c + c'2+\ldots+ c'(n-1)+ nc' = c+c'n(n+1)/2 - c' = O(n^2)$

Metodo di sostituzione. Ipotizziamo $T(n) \leq Cn^2$ per $n \geq n_0$, dove C ed n_0 sono costanti positive da determinare. Dimostriamo che la nostra intuizione e` corretta utilizzando l'induzione.

Base induzione: $T(1) \leq c \leq 1^2 C$ se $C \geq c$

Passo induttivo.

$T(n) \leq T(n-1)+c'n \leq C(n-1)^2+c'n = Cn^2+C-2Cn+c'n$ l'ultimo membro è $\leq Cn^2$ se $C-2Cn+c'n \leq 0$ e questa disuguaglianza vale se $C \geq c'n/(2n-1)$.

Siccome $c'n/(2n-1) \leq c'$ allora basta prendere $C \geq c'$

Affinche' valgano sia la base dell'induzione e il passo induttivo, basta quindi prendere $C=\max\{c,c'\}$ e $n_0=1$

Algoritmo per l'ordinamento topologico con informazioni aggiuntive

- Il bound $O(n^2)$ non è molto buono se il grafo è sparso, cioè se il numero di archi è molto più piccolo di n^2
- Possiamo ottenere un bound migliore?

- Per ottenere un bound migliore occorre usare un modo efficiente per individuare un nodo senza archi entranti ad ogni chiamata ricorsiva

- Si procede nel modo seguente:
- Un nodo si dice attivo se non è stato ancora cancellato
- Occorre mantenere le seguenti informazioni:
- per ciascun vertice attivo w
  - count[w] = numero di archi entranti in w provenienti da nodi attivi.
  - S = insieme dei nodi attivi che non hanno archi entranti provenienti da altri nodi attivi.

Algoritmo per l'ordinamento topologico con informazioni aggiuntive: analisi

Teorema. L'algoritmo trova l'ordinamento topologico di un DAG in tempo $O(m + n)$.
Dim.

- Inizializzazione. Richiede tempo $O(m + n)$ in quanto
  - I valori di count[w] vengono inizializzati scandendo tutti gli archi e incrementando count[w] per ogni arco entrante in w basta scandire tutti gli archi una sola volta → tempo $O(m)$
    - Se per ogni nodo viene memorizzato il numero di archi entranti → tempo $O(n)$
  - Inizialmente tutti i nodi sono attivi per cui S consiste dei nodi di G senza archi entranti ed è sufficiente esaminare count[w] per tutti i nodi w una sola volta per inizializzare S → tempo $O(n)$

- Aggiornamento. Per trovare il nodo v da cancellare basta prendere un nodo da S. Per cancellare v occorre
  1. Cancellare v da S e da G. Cancellarlo da G costa deg(v). Se S è rappresentata da una lista e se cancelliamo ogni volta da S il primo nodo della lista→ tempo $O(1)$ (anche in una lista a puntatori singoli)
  2. Per ogni arco (v,w), decrementare count[w] e se count[w] diventa uguale 0 aggiungere w a S → tempo $O(\deg(v))$.

I passi 1. e 2. vengono eseguiti una volta per ogni vertice → tutti gli aggiornamenti vengono fatti in $\sum_{u \in V} O(1) + \sum_{u \in V} O(\deg(u)) = O(n) + O(m) = O(n+m)$