Grafi (II parte)

Progettazione di Algoritmi a.a. 2018-19

Matricole congrue a 1

Docente: Annalisa De Bonis

34

Depth first search (visita in profondità)

- La visita in profondità riproduce il comportamento di una persona che esplora un labirinto di camere interconnesse
  - La persona parte dalla prima camera (nodo s) e si sposta in una delle camere accessibili dalla prima (nodo adiacente ad s), di lì si sposta in una delle camere accessibili dalla seconda camera visitate e così via fino a che raggiunge una camera da cui non è possibile accedere a nessuna altra camera non ancora visitata. A questo punto torna nella camera precedentemente visitata e di lì prova a raggiungere nuove camere.

35

Depth first search (visita in profondità)

- La visita DFS parte dalla sorgente s e si spinge in profondità fino a che non è più possibile raggiungere nuovi nodi.
  - La visita parte da s, segue uno degli archi uscenti da s ed esplora il vertice v a cui porta l'arco.
  - Una volta in v, se c'è un arco uscente da v che porta in un vertice w non ancora esplorato allora l'algoritmo esplora w
  - Uno volta in w segue uno degli archi uscenti da w e così via fino a che non arriva in un nodo del quale sono già stati esplorati tutti i vicini.
  - A questo punto l'algoritmo fa backtrack (torna indietro) fino a che torna in un vertice a partire dal quale può visitare un vertice non ancora esplorato in precedenza.

36

Depth first search: pseudocodice

```
DFS(u):
  Mark u as "Explored" and add u to R
  For each edge (u, v) incident to u
    If v is not marked "Explored" then
      Recursively invoke DFS(v)
    Endif
  Endfor
```

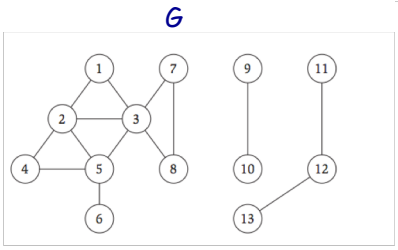
R = insieme dei vertici raggiunti

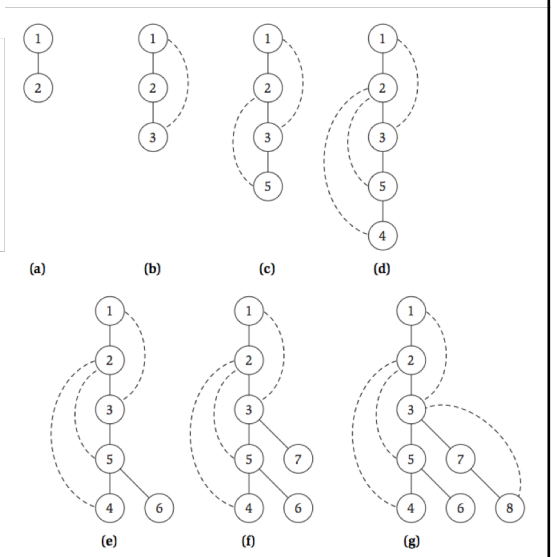
Analisi:
Ciascuna visita ricorsiva richiede tempo O(1) per marcare u e aggiungerlo ad R e tempo O(deg(u)) per eseguire il for.
Se inizialmente invochiamo DFS su un nodo s, allora DFS viene invocata ricorsivamente su tutti i nodi raggiungibili a partire da s. Il costo totale è quindi al più

$$\sum_{u \in V} O(1) + \sum_{u \in V} O(\deg(u)) = O(n) + O(m) = O(n+m)$$

37

Esempio di esecuzione di DFS





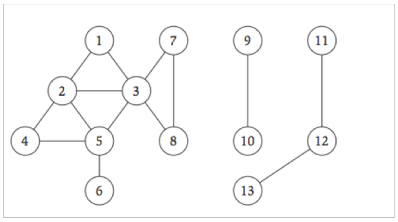
38

Depth First Search Tree (Albero DFS)

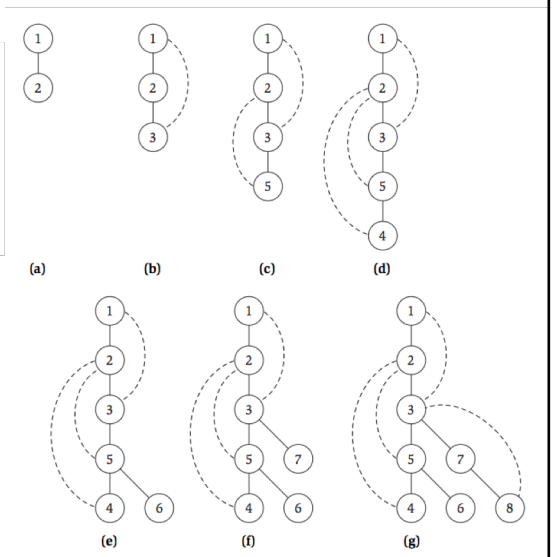
- Proprietà. L'algoritmo DFS produce un albero che ha come radice la sorgente s e come nodi tutti i nodi del grafo raggiungibili da s.
- L'albero si ottiene in questo modo:
  - Consideriamo il momento in cui viene invocata DFS(v)
  - Ciò avviene durante l'esecuzione di DFS(u) per un certo nodo u. In particolare durante l'esame dell'arco (u,v) nella chiamata DFS(u).
  - In questo momento, aggiungiamo l'arco (u,v) e il nodo v all'albero

Esempio di albero DFS

G



Gli archi non tratteggiati fanno parte del DFS tree.



40

Albero DFS

- Proprietà 1. Per una data chiamata ricorsiva DFS(u), tutti i nodi che vengono etichettati come "Esplorati" tra l'inizio e la fine della chiamata DFS(u), sono discendenti di u nell'albero DFS.

- Proprietà 2. Sia T un albero DFS e siano x e y due nodi di T collegati dall'arco (x,y) in G. Si ha che x e y sono l'uno antenato dell'altro in T.

Dim. Proprietà 2

- **Caso (x,y) e` in T.** In questo caso la proprietà e` ovviamente soddisfatta.
- **Caso (x,y) non e` in T.** Supponiamo senza perdere di generalità che DFS(x) venga invocata prima di DFS(y). Ciò vuol dire che quando viene invocata DFS(x), y non è ancora etichettato come "Esplorato".
- La chiamata DFS(x) esamina l'arco (x,y) e per ipotesi non inserisce (x,y) in T. Ciò si verifica solo se y è già stato etichettato come "Esplorato". Siccome y non era etichettato come "Esplorato" all'inizio di DFS(x) vuol dire è stato esplorato tra l'inizio e la fine della chiamata DFS(x). La proprietà 1 implica che y è discendente di x.

Implementazione di DFS mediante uno stack

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

Analisi di DFS implementata mediante uno stack

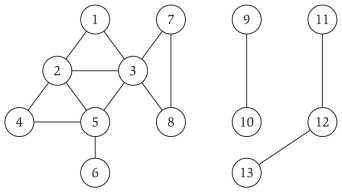
```
1.  DFS(s):
2.  Poni Explored[s] = true ed Explored[v] = false per tutti gli altri nodi
3.  Inizializza  S con uno stack contenente s
4.  While S non è vuoto
5.      Metti in u  il nodo  al top di S
6.      If  c'e`  un arco (u, v) incidente su u non ancora esaminato then
7.          If Explored[v]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

O(n+m)

- Linea 1: O(n)
- Linea 2: O(1)
- Il while viene iterato O(deg(v)) volte per ogni nodo v esplorato e di conseguenza inserito in S. Quindi in totale il while e` iterato un numero di volte pari alla somma dei gradi dei nodi esplorati che e` al più
  - 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(m).

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

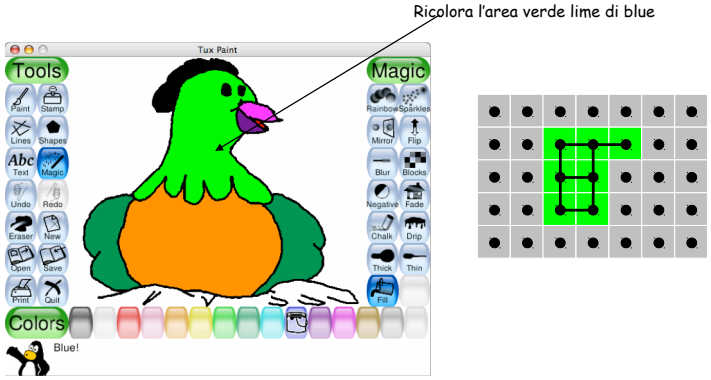


- Componente connessa contenente il nodo 1 è { 1, 2, 3, 4, 5, 6, 7, 8 }.

44

Flood Fill

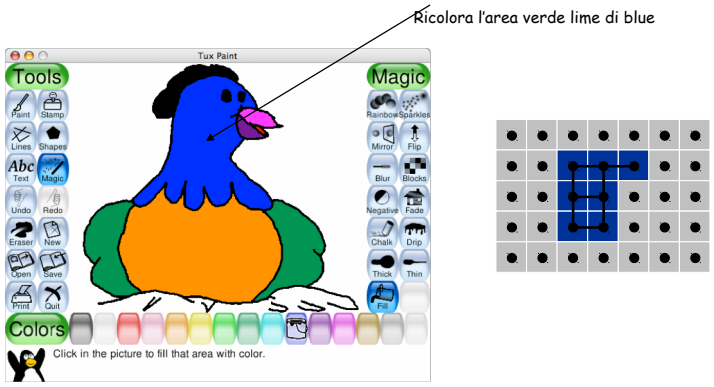
- Flood fill. Data un'immagine, cambia il colore dell'area di pixel vicini di colore verde lime in blu.
  - Nodo: pixel.
  - Arco: due pixel vicini di colore verde lime.
  - Area di pixel vicini: componente connessa di pixel di colore verde lime.



45

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.

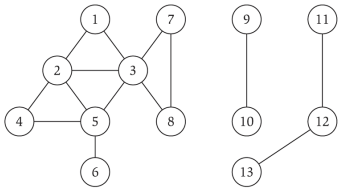


46

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

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 se 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

- Teorema. Per ogni due nodi s e t di un grafo, le loro componenti connesse o sono uguali o disgiunte
- Dim.

- Supponiamo che le componenti di s e t non siano né uguali né disgiunte

I. Non disgiunte → Esiste u diverso da s e da t nell'intersezione delle due componenti

II. Diverse → Esiste v che appartiene **solo** ad una delle due. Assumiamo s.p.d.g. (senza perdere di generalità) che v appartenga alla componente di s.

Dalla I posso andare da t a u e da u ad s. Dalla II posso andare da s a v. Di conseguenza posso andare da t a v passando per u. Deduciamo quindi che t e v sono nella stessa componente connessa contraddicendo l'ipotesi che v fosse solo nella componente di s.

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
```

- 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`

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

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

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

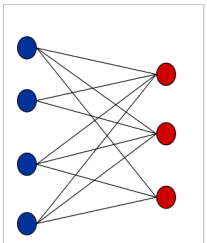
- e il tempo totale di esecuzione dell'algoritmo che scopre le componenti connesse e` $O(n)+O(n+m)=O(n+m)$

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

Grafi bipartiti

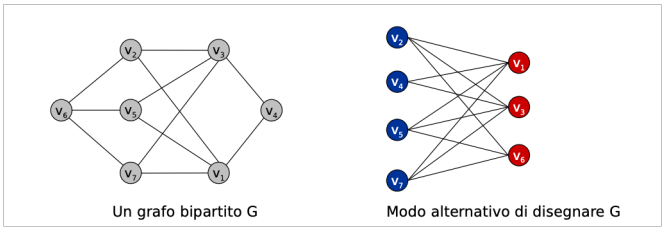
- Def. Un grafo non direzionato è **bipartito** se l'insieme di nodi può essere partizionato in due sottoinsiemi X e Y tali che ciascun arco del grafo ha una delle due estremità in X e l'altra in Y
  - Possiamo colorare i nodi con due colori (ad esempio, rosso e blu) in modo tale che ogni arco ha un'estremita rossa e l'altra blu.
- Applicazioni.
  - Matrimoni stabili: uomini = rosso, donna = blu.
  - Scheduling: macchine = rosso, job = blu.



Un grafo bipartito

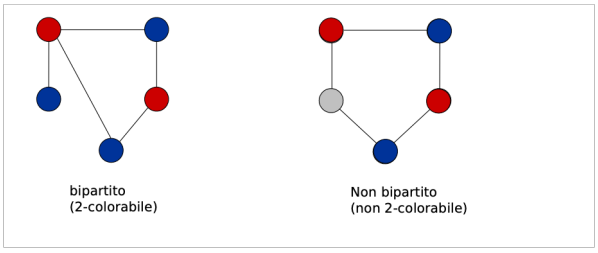
Testare se un grafo è bipartito

- Testare se un grafo è bipartito. Dato un grafo G, vogliamo scoprire se è bipartito.
  - Molti problemi su grafi diventano:
    - Più facili se il grafo sottostante è bipartito (matching: sottoinsieme di archi tali che non hanno estremità in comune)
    - Trattabili se il grafo è bipartito (max insieme indipendente)



Grafi bipartiti

- Lemma. Se un grafo G è bipartito, non può contenere un ciclo dispari (formato da un numero dispari di archi)
- Dim. Non è possibile colorare di rosso e blu i nodi su un ciclo dispari in modo che ogni arco abbia le estremità di diverso colore.



Breadth First Search Tree

- Proprietà. Si consideri un'esecuzione di BFS su $G = (V, E)$, e sia (x, y) un arco di G. I livelli di x e y differiscono di al più di 1.

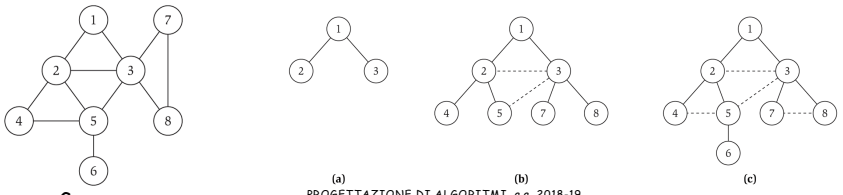
- Dim. Sia L_i il livello di x ed L_j quello di y. Supponiamo senza perdere di generalità che x venga scoperto prima di y cioè che **$i \leq j$**. Consideriamo il momento in cui l'algoritmo esamina gli archi incidenti su x.

- Caso 1. Il nodo y è stato già scoperto:

Siccome per ipotesi y viene scoperto dopo x allora sicuramente y viene inserito o nel livello i dopo x (se adiacente a qualche nodo nel livello $i-1$) o nel livello $i+1$ (se adiacente a qualche nodo del livello i esaminato nel **For each** prima di x). Quindi in questo caso $j = i$ o $j = i+1$.

- Caso 2. Il nodo y non è stato ancora scoperto:

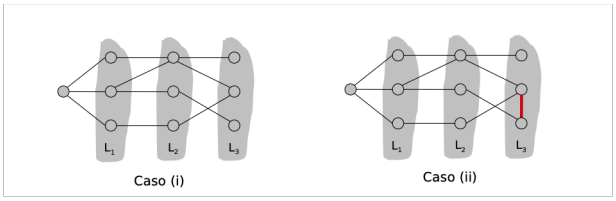
Siccome tra gli archi incidenti su x c'è anche (x,y) allora y viene inserito in questo momento in L_{i+1} . Quindi in questo caso $j = i+1$.



G (a) (b) (c)

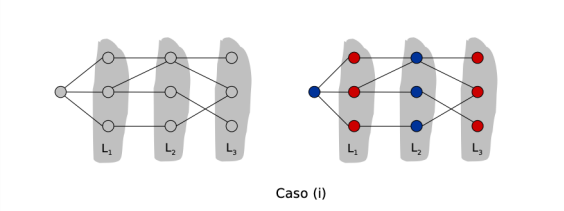
Grafi bipartiti

- Osservazione. Sia G un grafo connesso e siano $L_0, \ldots, L_k$ i livelli prodotti da un'esecuzione di BFS a partire dal nodo s. Può avvenire o che si verifichi la (i) o la (ii)
  - (i) Nessun arco di G collega due nodi sullo stesso livello
  - (ii) Un arco di G collega due nodi sullo stesso



Grafi Bipartiti

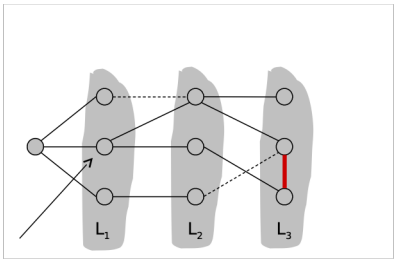
- Nel caso (i) il grafo è bipartito.
- Dim. Tutti gli archi collegano nodi in livelli consecutivi (per la proprietà sulla distanza dei nodi adiacenti nel BFS tree). Quindi se coloro i livelli di indice dispari di rosso e quelli di indice pari di blu, ho che le estremità di ogni arco sono di colore diverso.



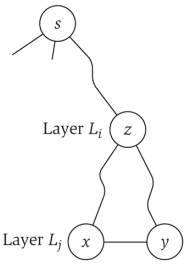
Caso (i)

Grafi Bipartiti

Nel caso (ii) il grafo non è bipartito.
Dim. Il grafo contiene un ciclo dispari: supponiamo che esiste l'arco (x,y) tra i vertici x e y di L_j. Indichiamo con z l'antenato comune a x e y nell'albero BFS che si trova più vicino a x e y. Sia L_i il livello in cui si trova z. Possiamo ottenere un ciclo dispari del grafo prendendo il percorso seguito dalla BFS da z a x (j-i archi), quello da z a y (j-i archi) e l'arco (x,y). In totale il ciclo contiene 2(j-i)+1 archi.



Antenato comune più vicino (lowest common ancestor)



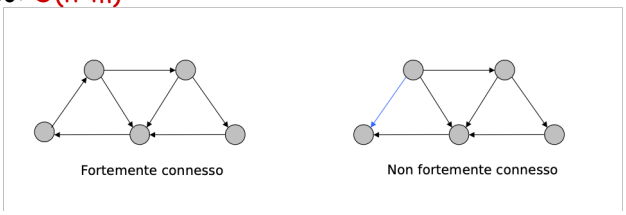
Caso (ii)

Algoritmo che usa BFS per determinare se un grafo è bipartito

Modifichiamo BFS come segue:

- Usiamo un array Color per assegnare i colori ai nodi
- Ogni volta che aggiungiamo un nodo v alla lista L[i+1] poniamo Color[v] uguale a rosso se i+1 è pari e uguale a blu altrimenti
- Alla fine esaminiamo tutti gli archi per vedere se c'è ne è uno con le estremità dello stesso colore . Se c'è concludiamo che G non è bipartito (perche'?); altrimenti concludiamo che G è bipartito (perche'?).

- Tempo: $O(n+m)$

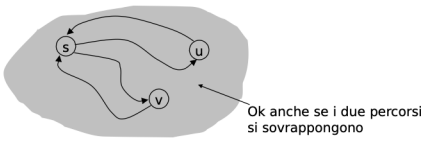


Visita di grafi direzionati

- 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 visitw 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.

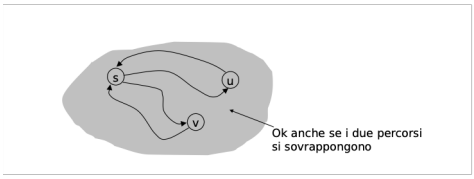
Connettività forte

- 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 è fortemente connesso se ogni coppia di nodi è mutualmente raggiungibile



Connettività forte

- Lemma. Sia s un qualsiasi nodo di 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

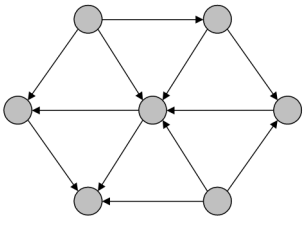
Teorema. Si può determinare se G è fortemente connesso in tempo O(m + n).

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)

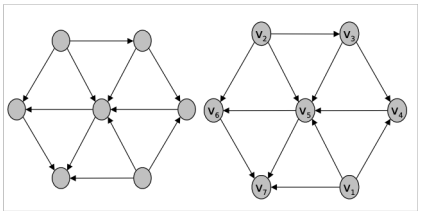
- Def. Un DAG è un grafo direzionato che non contiene cicli direzionati
- Possono essere usati per esprimere vincoli di precedenza o dipendenza: l'arco (v_i, v_j) indica che v_i deve precedere v_j o che v_j dipende da v_i
- Infatti generalmente i grafi usati per esprimere i suddetti vincoli sono privi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



Un DAG G

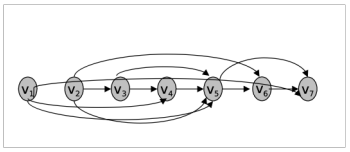
Ordine topologico

- Def. Un ordinamento topologico di un grafo direzionato $G = (V, E)$ è una etichettatura dei suoi nodi $v_1, v_2, \ldots, v_n$ tale che per ogni arco (v_i, v_j) si ha $i < j$. Detto in un altro modo, 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



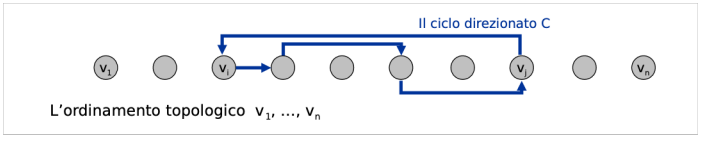
Un modo diverso di ridisegnare G in modo da evidenziare l'ordinamento topologico di G

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 $j < i$.
- $j < i$. e` impossibile in quanto abbiamo scelto i minore di j e quindi siamo arrivati ad un assurdo. Cio` è una contraddizione al fatto che G contiene un ciclo.



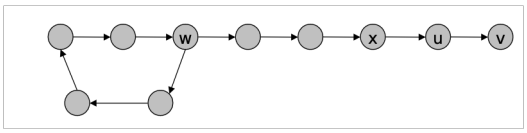
DAG e ordinamento topologico

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

DAG e ordinamento topologico

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 (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)

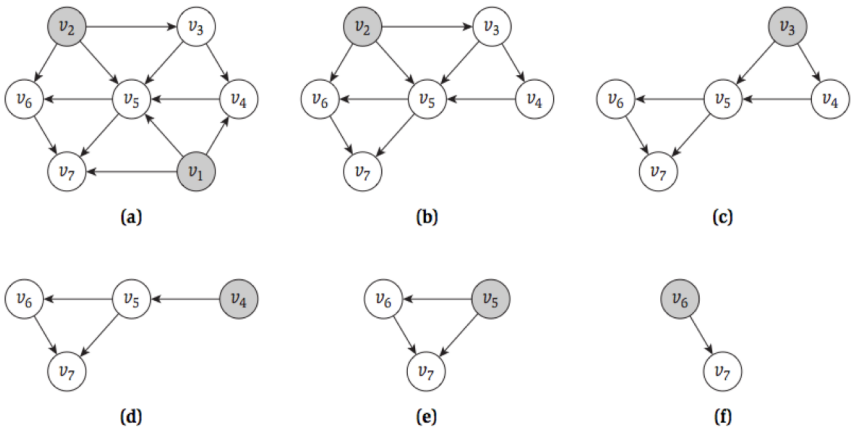
- Caso base: vero banalmente se n = 1.
- Passo induttivo: supponiamo asserto del lemma vero per DAG con n≥1 nodi
- Dato un DAG con n+1 > 1 nodi, prendiamo un nodo v senza archi entranti (abbiamo dimostrato che un tale nodo deve esistere).
- G - { v } è un DAG, in quanto cancellare un nodo non introduce 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 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.

```
To compute a topological ordering of G:
Find a node v with no incoming edges and order it first
Delete v from G
Recursively compute a topological ordering of G−{v}
  and append this order after v
```

Algoritmo per l'ordinamento topologico



Algoritmo per l'ordinamento topologico : analisi dell'algoritmo

1) Trovare un nodo senza archi entranti richiede $O(n)$
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 $O(n^2+m)=O(n^2)$

```
To compute a topological ordering of G:
Find a node v with no incoming edges and order it first
Delete v from G
Recursively compute a topological ordering of G−{v}
  and append this order after v
```

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

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

$\leq c+c'2+\ldots+nc' = c+c'n(n+1)/2 - c' = O(n^2)$

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: $T(1)) \leq c \leq 1^2C$ se $C \geq c$
Passo induttivo.
$T(n) \leq T(n-1)+c'n \leq C(n-1)^2+c'n = Cn^2+C-2Cn+c'n \leq Cn^2$ se $C \geq c'$
Basta 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 $O(m)$
  - Inizialmente tutti i nodi sono attivi per cui S consiste dei nodi di G senza archi entranti. E` sufficiente scandire tutti i nodi una sola volta per inizializzare S → tempo $O(n)$

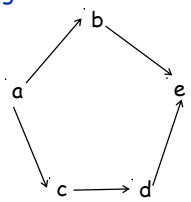
- Aggiornamento. Per trovare il nodo v da cancellare basta prendere un nodo da S. Per cancellare v occorre
  1. Cancellare v da S e da G. Cancellarlo da G costa deg(v). Possiamo rappresentare S mediante una lista. Se cancelliamo ogni volta da S il primo nodo della lista→tempo $O(1)$ (anche in una lista a puntatori singoli)
  2. Decrementare count[w] per ogni arco (v,w). Se count[w]=0 allora occorre 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 $\sum_{u \in V} O(1) + \sum_{u \in V} O(\deg(u)) = O(n) + O(m) = O(n+m)$

76

Esercizio

Fornire tutti gli ordinamenti topologici del grafo sottostante



Soluzione. Potremmo esaminare le 5!= 120 possibili permutazioni….
Ragioniamo: il primo nodo dell'ordinamento non deve avere archi entranti, l'ultimo non deve avere archi uscenti. Gli unici nodi che rispettivamente soddisfano questi requisiti sono a ed e. Quindi ogni ordinamento topologico deve cominciare con a e finire con e. In quanti modi possono essere sistemati gli altri nodi? Osserviamo che l'arco (c,d) implica che c precede d in quasiasi ordinamento topologico mentre b può trovarsi in una qualsiasi posizione tra a ed e. In totale, ci sono quindi 3 ordinamenti topologici
a b c d e , a c b d e, a c d b e

77

Esercizio 2 Cap 3

Fornire un algoritmo che, dato un grafo non direzionato G, scopre se G contiene cicli e in caso affermativo produce in output uno dei cicli. L'algoritmo deve avere tempo di esecuzione O(n+m)

Soluzione. Si esegua una visita BFS sul grafo. Se il grafo non è connesso si eseguono più visite, una per componente connessa. Se al termine gli alberi BFS contengono tutti gli archi allora G non contiene cicli. In caso contrario, c'è almeno un arco (x,y) che non fa parte degli alberi BFS. Consideriamo l'albero BFS T in cui si trovano x e y e sia z l'antenato comune più vicino a x e y (LCA di x e y). L'arco (x,y) insieme ai percorsi tra z e x e quello tra z e y forma un ciclo .

Come facciamo a trovare lo LCA di x e y in tempo O(n)?

Esercizio 3 Cap. 3

Modificare l'algoritmo per l'ordinamento topologico di un DAG in modo tale che se il grafo direzionato input non è un DAG l'algoritmo riporta in output un ciclo che fa parte del grafo.

Soluzione.

- Caso 1. All'inizio di ogni chiamata ricorsiva dell'algoritmo per l'ordinamento topologico, l'insieme S non è vuoto. In questo caso riusciamo ad ottenere un ordinamento topologico perché ogni nodo cancellato v non ha archi entranti che provengono dai nodi che sono ancora attivi e che quindi saranno posizionati nell'ordinamento dopo v. Il Lemma ci dice che se il grafo ha un ordinamento topologico allora il grafo è un DAG.
- Caso 2. All'inizio di una certa chiamata ricorsiva, S è vuoto. In questo caso il grafo formato dai nodi attivi non è un DAG per il lemma che dice che un DAG ha almeno un nodo senza archi entranti. Il ciclo è ottenuto percorrendo a ritroso gli archi a partire da un qualsiasi nodo attivo v fino a che non incontriamo uno stesso nodo w due volte.

Continua nella prossima slide

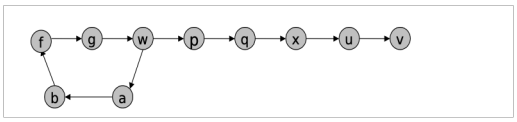
Esercizio 3 Cap. 3

- Basta quindi modificare l'algoritmo in modo che se all'inizio di una chiamata ricorsiva si ha che S è vuoto allora l'algoritmo sceglie un nodo attivo v e comincia a percorrere gli archi a ritroso a partire da v: si sceglie un arco (x,v) nella lista degli archi entranti in v, poi si sceglie un arco (y,x) nella lista degli archi entranti in x e così via.
- Ogni volta che viene attraversato un arco (p,q) a ritroso, il nodo p raggiunto viene inserito all'inizio di una lista a doppi puntatori ed etichettato come visitato.
- Se ad un certo punto si raggiunge un nodo w già etichettato come visitato, l'algoritmo interrompe questo percorso all'indietro e cancella dalla lista tutti i nodi a partire dalla fine della lista fino a che incontra per la prima volta w.
- I nodi restanti nella lista formano un ciclo direzionato che comincia e finisce in w.
- Tempo O(n+m) in quanto l'algoritmo per l'ordinamento ha costo O(n+m) e il costo aggiuntivo per trovare il ciclo è O(n).

Continua nella prossima slide

Esercizio 3 Cap. 3

Esempio di grafo con ciclo



Se cominciamo il cammino a ritroso a partire da v, la lista dei nodi attraversati è w a b f g w p q x u v (aggiungiamo ogni nodo attraversato all'inizio della lista). Non appena incontriamo la seconda occorrenza di w, ci fermiamo e cancelliamo gli ultimi 5 nodi della lista scandendo la lista a partire dalla fine. I nodi che rimangono nella lista formano il ciclo w a b f g w.

Esercizio 7 Cap. 3

Dimostrare o confutare la seguente affermazione:
Sia G un grafo non direzionato con un numero n pari di vertici e in cui ogni vertice ha grado almeno n/2. G è connesso.

Soluzione:

- L'affermazione è vera.
- Dimostrazione. Immaginiamo di eseguire la BFS su G a partire da un certo vertice u.
- Siccome u ha grado almeno n/2 allora nel livello L_1 ci saranno almeno n/2 vertici.
- Sia v un vertice diverso da u che non è in L_1
- Siccome v ha grado almeno n/2 e non è adiacente ad u (perche'?) allora almeno uno degli archi incidenti su u deve incidere su un vertice che si trova nel livello L_1.
  - Semplice argomento: se escludiamo u ed i nodi adiacenti ad u, rimangono al più n-1-n/2= n/2-1 altri nodi.
- Quindi ogni vertice v diverso da u e che non è adiacente ad u deve essere adiacente ad un nodo adiacente ad u.
- Quindi ogni nodo che non è nei primi due livelli sara inserito nella BFS nel terzo livello L_2 → G è connesso

Esercizio 6 Cap. 3

Sia G un grafo connesso tale che il DFS tree e il BFS tree di G sono uguali allo stesso albero T. Dimostrare che G=T (cioe` non ci sono archi di G che non sono inclusi in T).

Soluzione:

- Dimostrazione. Supponiamo per assurdo che esista un arco (x,y) di G che non e` in T.

1. Siccome T e` un BFS tree allora per la proprietà dimostrata per BFS si ha: (x,y) in G → i livelli di x e y in T differiscono al più di 1.
2. Siccome T e` anche un DFS tree allora per la proprietà dimostrata per DFS si ha: (x,y) in G → x e` discendente di y o y e` discendente di x
3. Siccome (x,y) non e` in T allora x e y non sono in relazione padre figlio.

- Dalla 2. sappiamo che x e y sono uno discendente dell'altro in T e quindi non possono essere sullo stesso livello. La 1. e la 2. insieme allora implicano che x e y sono in livelli consecutivi e sono uno discendente dell'altro. Ciò e` possibile solo se x e y sono in relazione padre figlio.
- Siamo arrivati a contraddire la 3 e quindi non e` possibile che esista un arco di G che non e` in T.