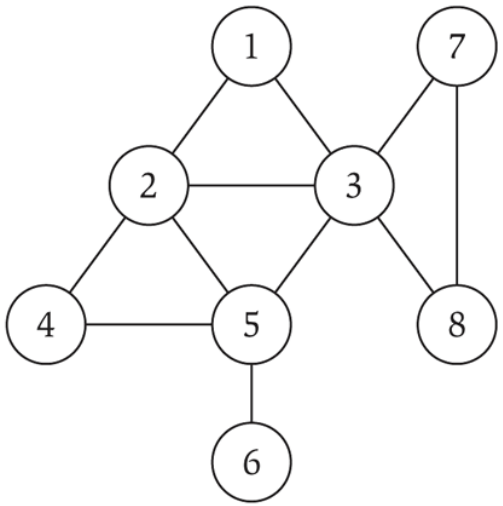
Grafi

Progettazione di Algoritmi a.a. 2015-16
Matricole congrue a 1
Docente: Annalisa De Bonis

Grafi non direzionati

- Grafi non direzionati. $G = (V, E)$
  - V = insieme nodi.
  - E = insieme archi.
  - Esprime le relazioni tra coppie di oggetti.
  - Parametri del grafo: $n = |V|$, $m = |E|$.



V = { 1, 2, 3, 4, 5, 6, 7, 8 }

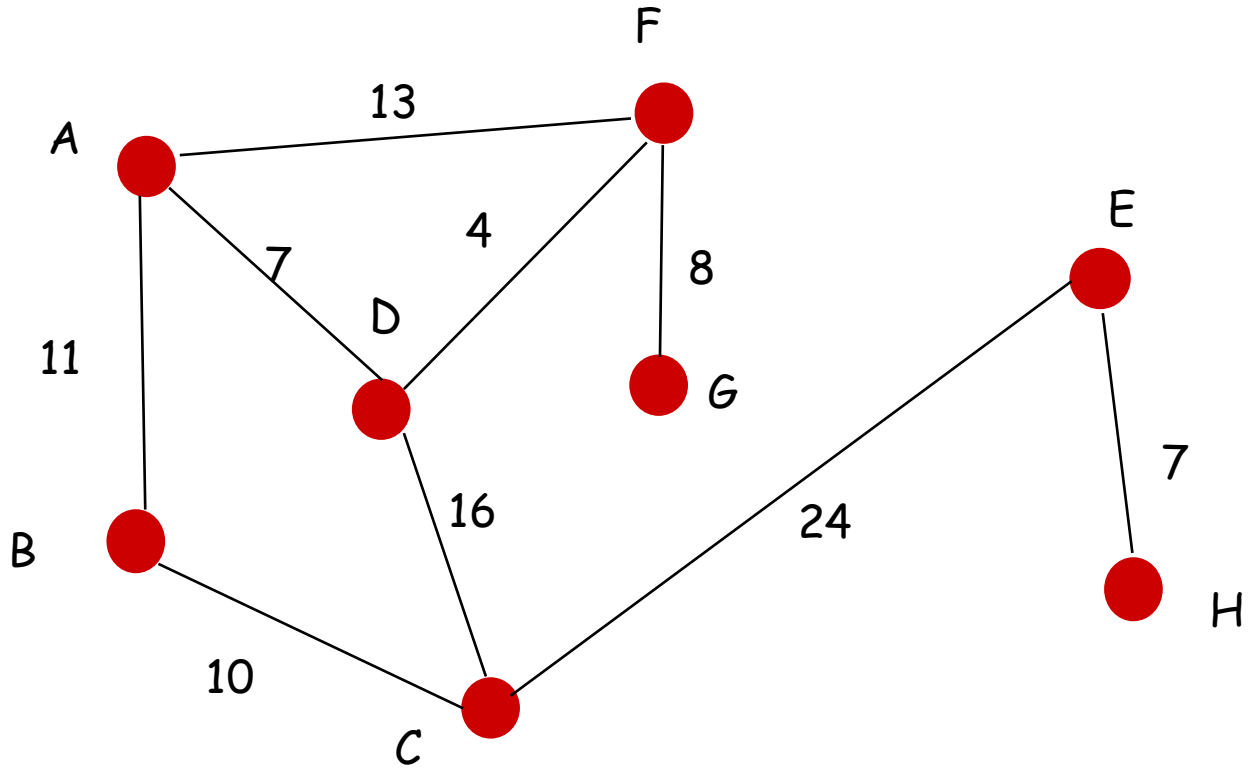
E = { 1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6 }

n = 8

m = 11

Esempio di applicazione

- Archi: strade (a doppio senso di circolazione)
- Nodi: intersezioni tra strade
- Pesi archi: lunghezza in km



Terminologia

- Consideriamo due nodi di un grafo *G* connessi dall'arco e = (u,v)

- Si dice che
  - u e v sono adiacenti
  - l'arco (u,v) incide sui vertici u e v
  - u è un nodo vicino di v
  - v è un nodo vicino di u

- Dato un vertice u di un grafo *G*
  - Grado di u = numero archi incidenti su u
    - è indicato con deg(u)

Numero di archi di un grafo *G*

- m = numero di archi di *G*; n= numero di nodi di *G*

1. La somma di tutti i gradi dei nodi di *G* è 2m

$$\sum_{u \in V} \deg(u) = 2m$$

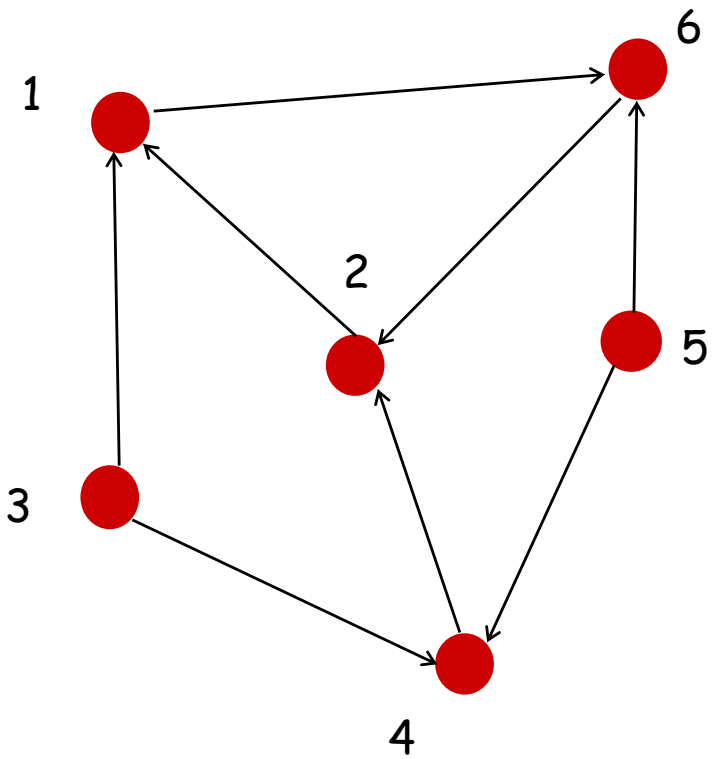
- Dim. Ciascun arco incide su due vertice e quindi viene contato due volte nella sommatoria in alto.
  - L'arco (u,v) è contato sia in deg(u) che in deg(v)

2. Il numero m di archi di un grafo *G* **non direzionato** è **al più** n(n-1)/2

- Dim. Il numero di coppie non ordinate distinte che si possono formare con n nodi è n(n-1)/2
  - Posso scegliere il primo nodo dell'arco in n nodi e il secondo in modo che sia diverso dal primo nodo, cioè n-1 modi. Dimezzo in quanto l'arco (u,v) è uguale all'arco (v,u)

Grafi direzionati

- Gli archi hanno una direzione
  - L'arco (u,v) è diverso dall'arco (v,u)
  - Si dice che l'arco e=(u,v) lascia u ed entra in v



V = { 1, 2, 3, 4, 5, 6 }

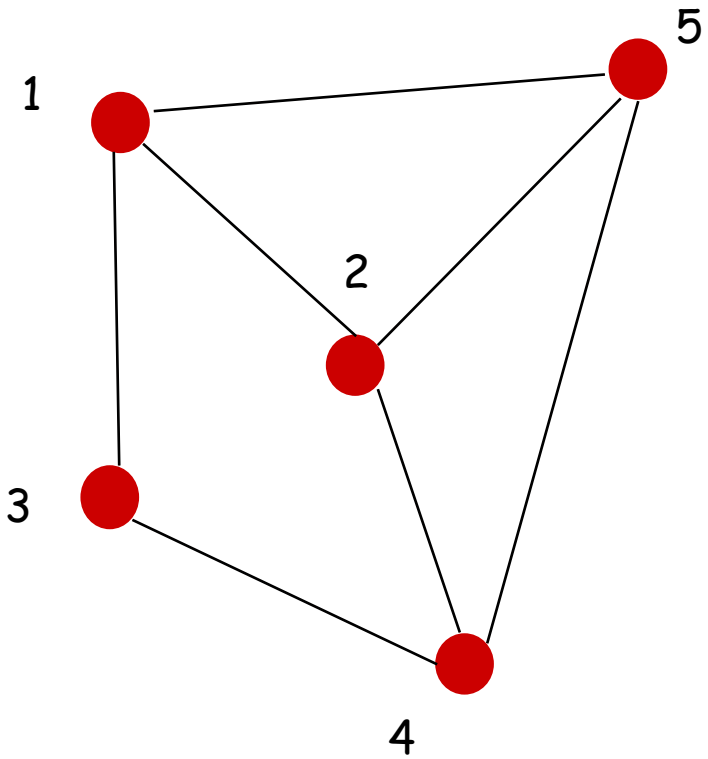
E = {1-6, 2-1, 3-1, 3-4, 4-2, 5-4, 5-6, 6-2 }

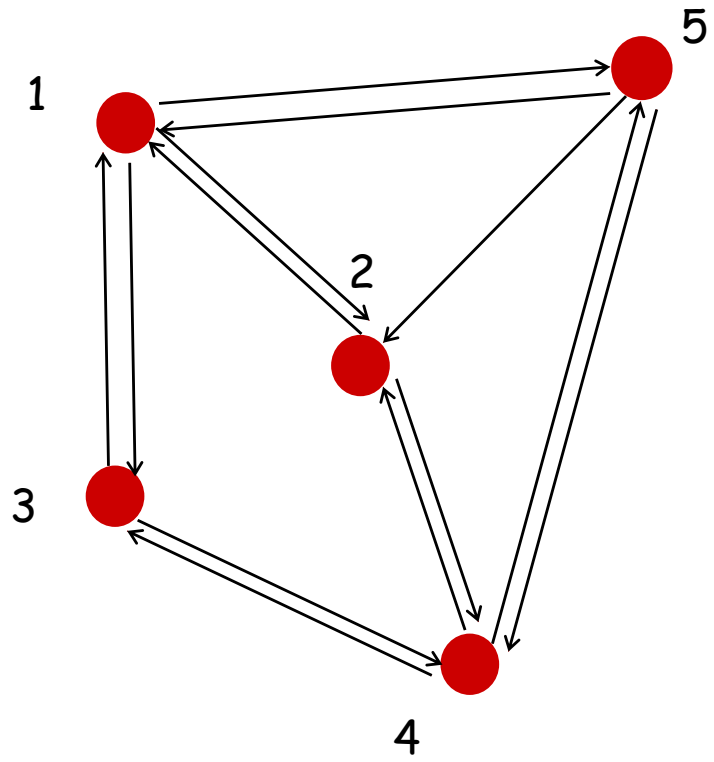
n = 6

m = 8

Grafi direzionati

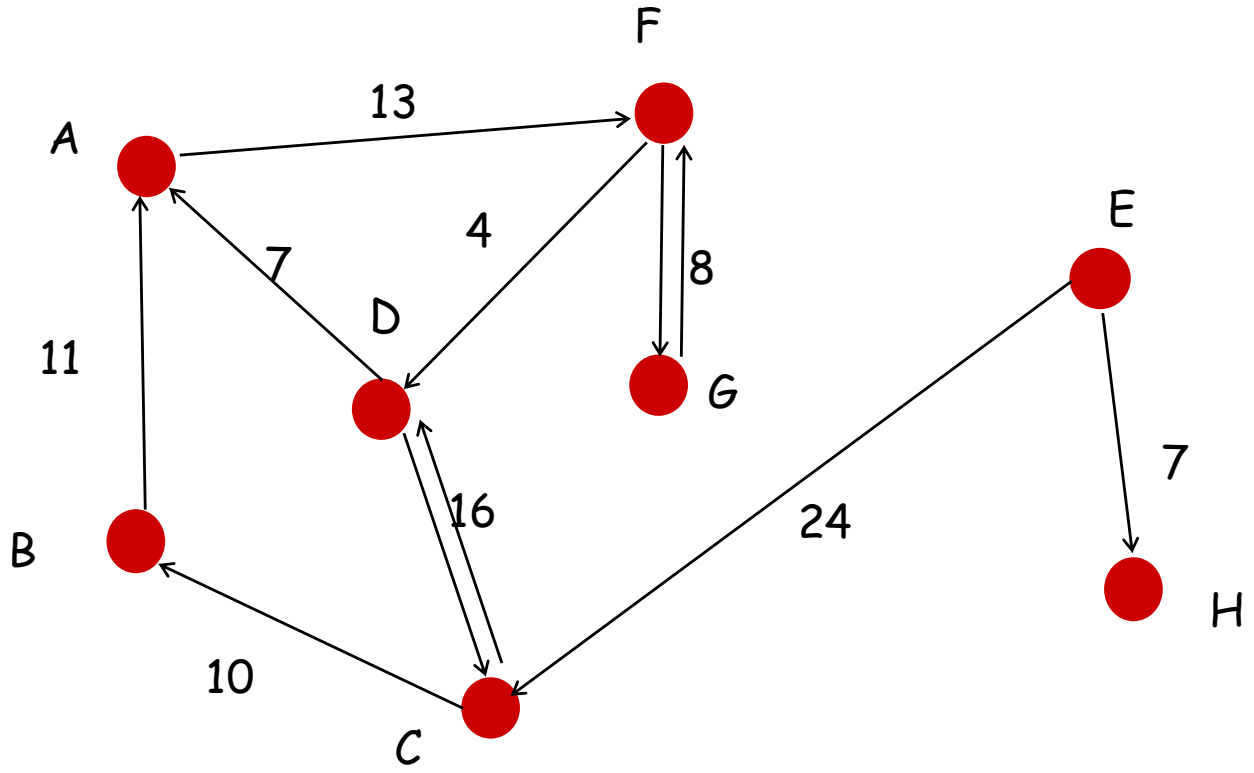
- Grafi non direzionati $G = (V, E)$ possono essere visti come un caso particolare degli archi direzionati in cui per ogni arco (u,v) c'è l'arco di direzione opposta (v,u)





Esempio di applicazione

- Archi: strade (a senso unico di circolazione)
- Nodi: intersezioni tra strade
- Pesi archi: lunghezza in km



Numero di archi di un grafo G direzionato

- m = numero di archi di G; n= numero di nodi di G

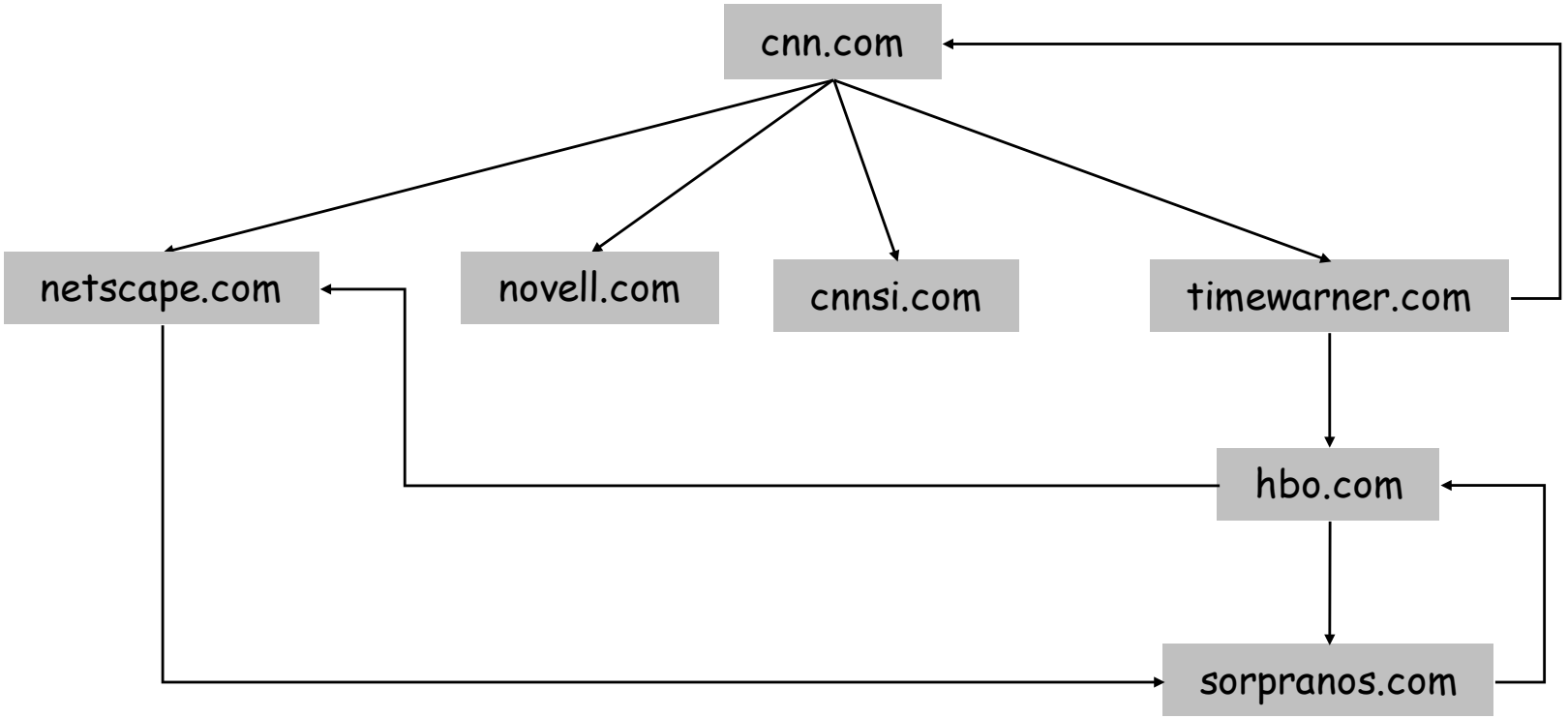
- Il numero m di archi di G è **al più** n^2
- Dim. Il numero di coppie ordinate distinte che si possono formare con n nodi è n^2
  - Posso scegliere il primo nodo dell'arco in n nodi e il secondo in altri n modi (se ammettiamo archi con entrambe le estremità uguali.

Alcune applicazioni dei grafi

Grafo	*Nodi*	*Archi*
trasporto	intersezioni di strade	strade
trasporto	aeroporti	voli diretti
comunicazione	computer	cavi di fibra ottica
World Wide Web	web page	hyperlink
rete sociale	persone	relazioni
catena del cibo	specie	predatore-preda
scheduling	task	vincoli di precedenza
circuiti	gate	wire

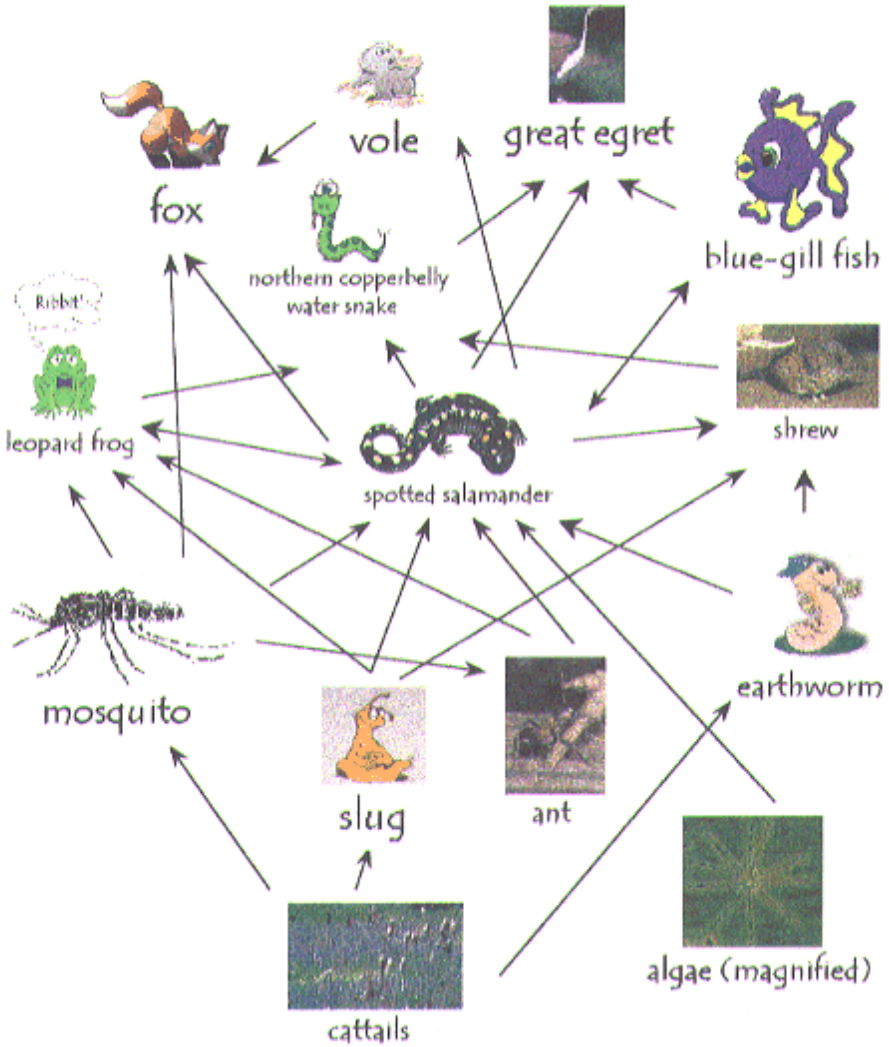
World Wide Web

- Web graph.
  - Nodo: pagina web.
  - Edge: hyperlink da una pagina all'altra.



Ecological Food Web

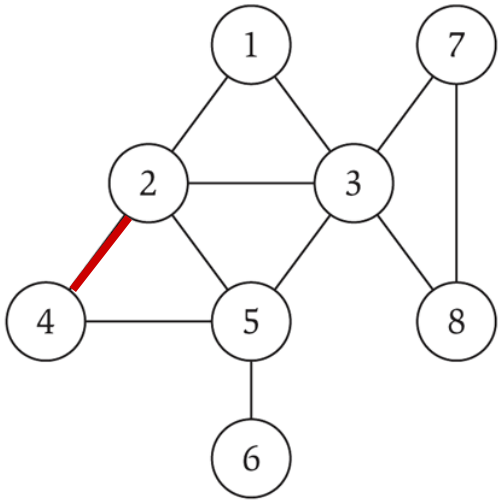
- Food web graph.
  - Nodo = specie.
  - Arco = dalla preda al predatore



Reference: http://www.twingroves.dètrict96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.giff

Rappresentazione di grafi: Matrice di Adiacenza

- Matrice di adiacenza. Matrice nxn con $A_{uv} = 1$ se (u, v) è an arco.
  - Due rappresentazioni di ciascun arco.
  - Spazio proporizionale a n^2.
  - Controllare se (u, v) è un arco richiede tempo $\Theta(1)$.
  - Identificare gli archi incidenti su un nodo u richiede $\Theta(n)$
  - Identificare tutti gli archi richiede tempo $\Theta(n^2)$.

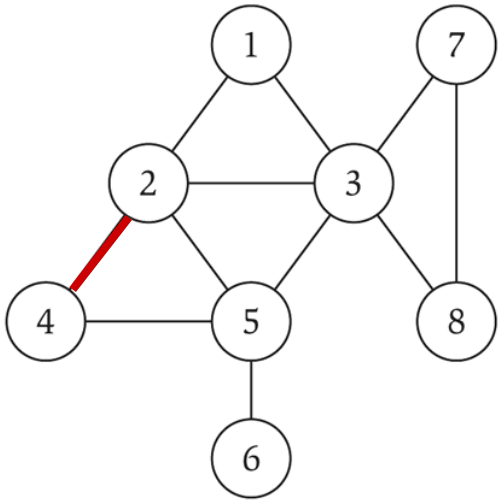


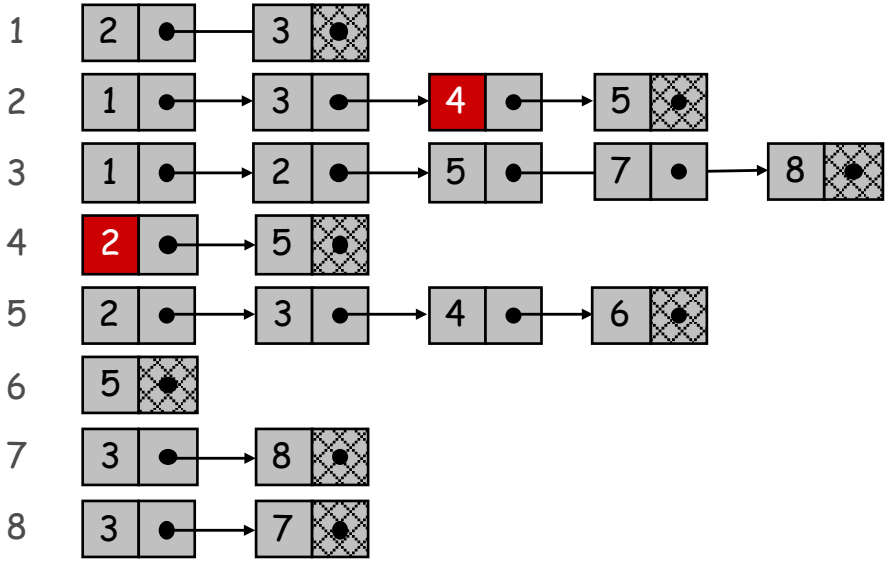
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	1	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Rappresentazione di un grafo: liste di adiacenza

- Liste di adiacenza. Array di liste in cui ogni lista è associata ad un nodo.
  - Ad ogni arco corrisponde un elemento della lista.
  - Spazio proporzionale a m + n.
  - Controllare se (u, v) è un arco richiede tempo O(deg(u)).
  - Individuare tutti gli archi richiede tempo $\Theta(m + n)$.

Degree (grado)= numero di vicini di u



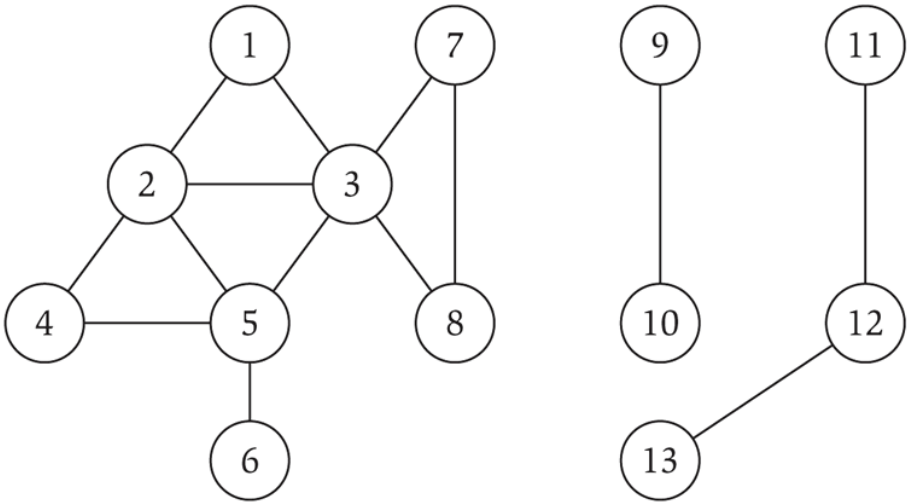


Percorsi e connettività

- Def. Un percorso in un grafo non direzionato $G = (V, E)$ è una sequenza P di nodi $v_1, v_2, \ldots, v_{k-1}, v_k$ con la proprietà che ciascuna coppia di vertici consecutivi v_i, v_{i+1} è unita da un arco in E.

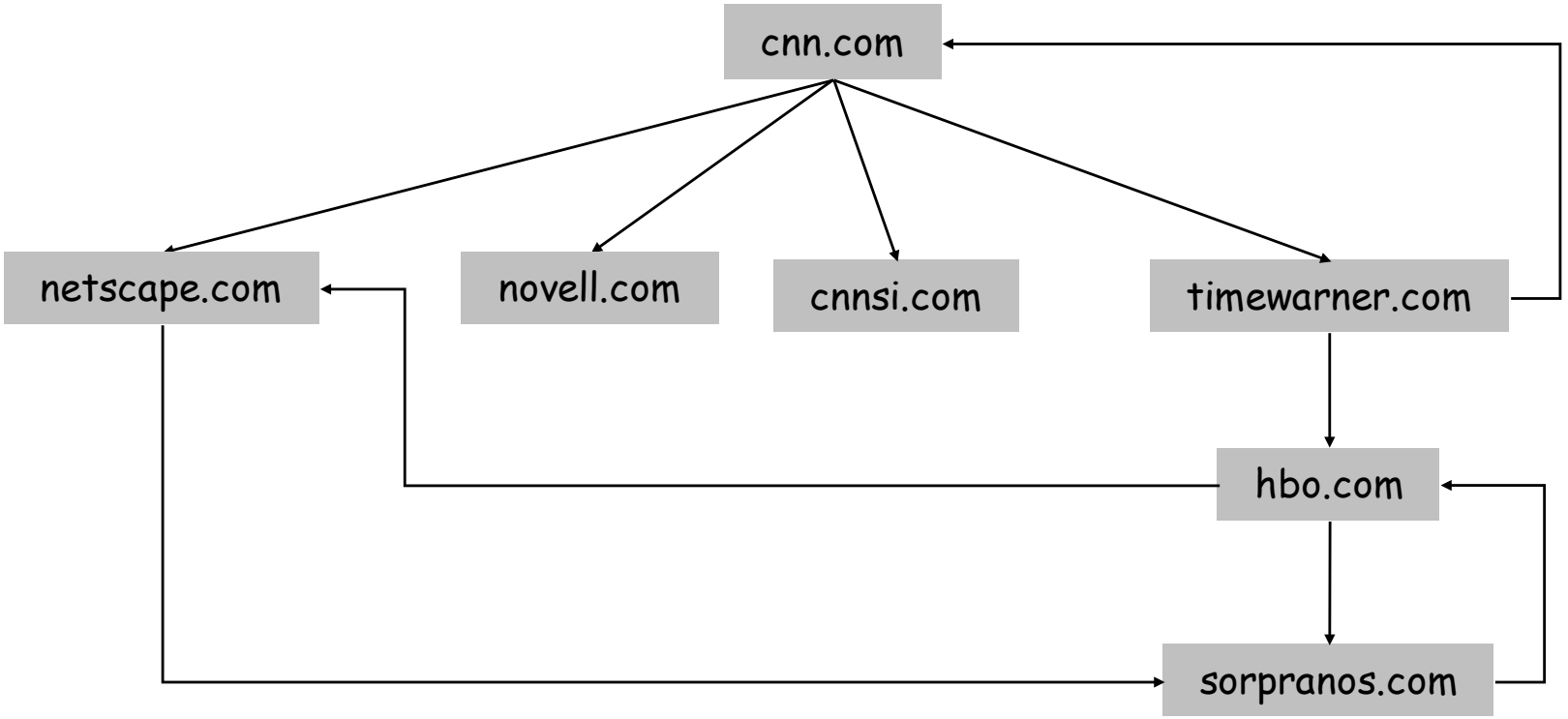
- Def. Un percorso è semplice se tutti i nodi sono distinti.

- Def. Un grafo non direzionato è connesso se per ogni coppia di nodi u e v, esiste un percorso tra u e v.



Applicazione del concetto di percorso

- Esempi:
  - Web graph. Voglio capire se è possibile, partendo da una pagina web è seguendo gli hyperlink nelle pagine via via attraversate, arrivare ad una determinata pagina



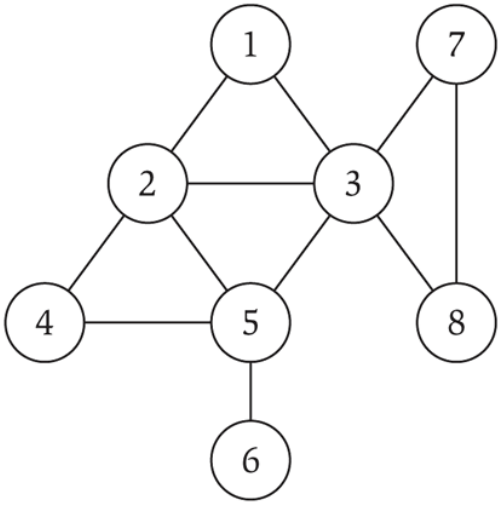
Applicazione del concetto di percorso

- In alcuni casi può essere interessante scoprire il percorso più corto tra due nodi

- Esempio:
  - Grafo : rete di trasporti dove i nodi sono gli aeroporti e gli archi i collegamenti diretti tra aeroporti.
  - Voglio arrivare da Napoli a New York facendo il minimo numero di scali.
-

Cicli

- Def. Un ciclo è un percorso $v_1, v_2, \ldots, v_{k-1}, v_k$ in cui $v_1 = v_k$, $k > 2$, e u primi k-1 nodi sono tutti distinti tra di loro



ciclo C = 1-2-4-5-3-1

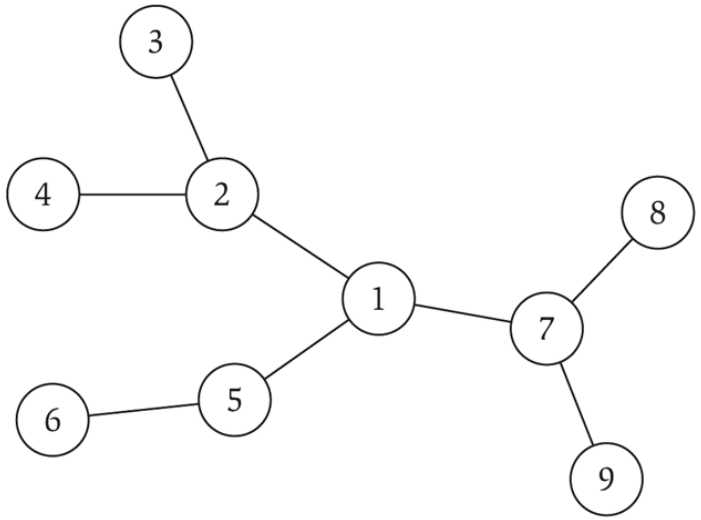
Alberi

- Def. Un grafo non direzionato è un albero (tree) se è connesso e non contiene cicli

- Teorema. Sia G un grafo direzionato con n nodi. Ogni due delle seguenti affermazioni implica la restante affermazione.
  - 1 e 2 ⟹ 3 ; 1 e 3 ⟹ 2 ; 2 e 3 ⟹ 1

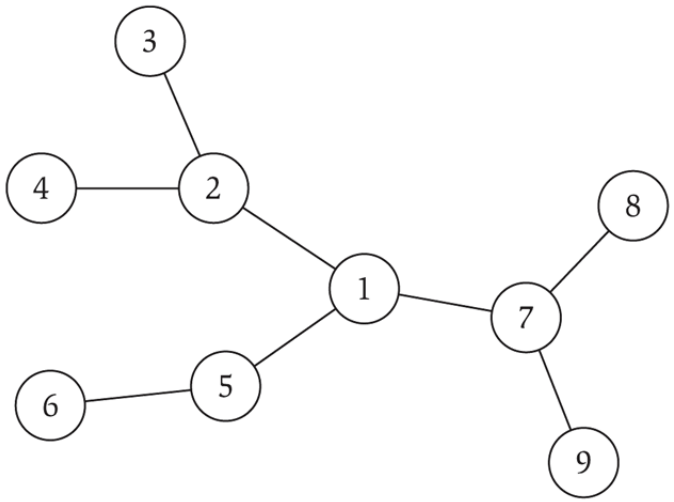
1. G è connesso.
2. G non contiene clicli.
3. G ha n-1 archi.

} albero

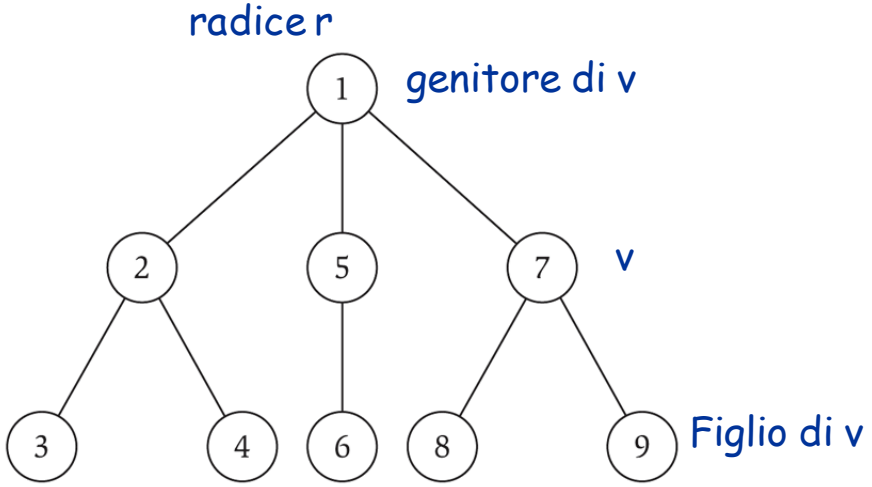


Alberi con radice

- Albero con radice. Dato un albero T, si sceglie un nodo radice r e si considerano gli archi di T come orientati a partire da r
- Dato un nodo v di T si dice
  - Genitore di v: il nodo che precede v lungo il percorso da r a v
  - Antenato: un qualsiasi nodo w lungo il percorso che va da r a v (v viene detto discendente di w)
- Foglia: nodo senza discendenti



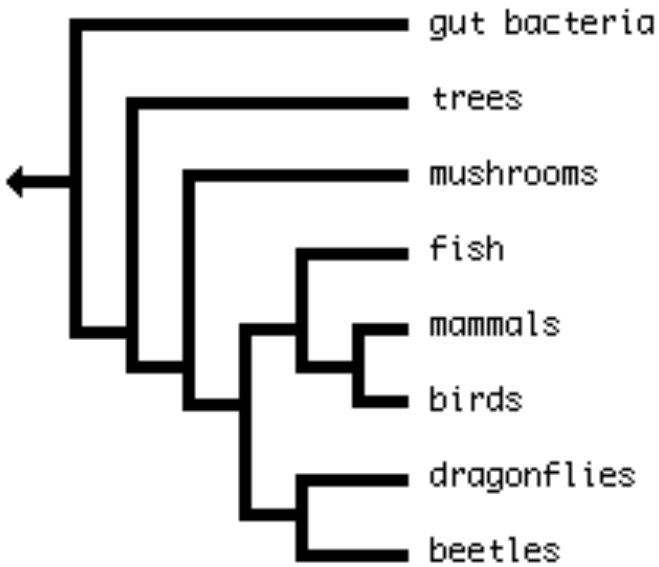
Un albero



Lo stesso albero con radice 1

Importanza degli alberi: rappresentano strutture gerarchiche

- Alberi filogenetici. Descrivono la storia evolutiva delle specie animali.

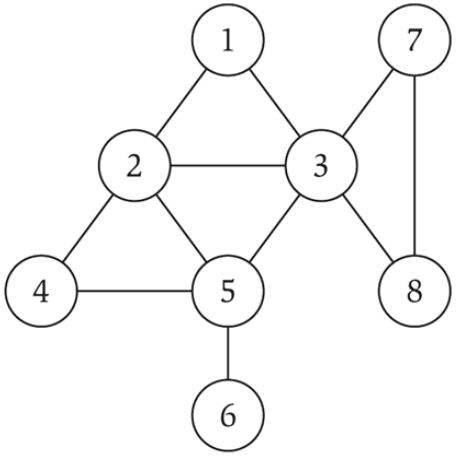


La filogenesi afferma l'esistenza di una specie ancestrale che diede origine a mammiferi e uccelli ma non alle altre specie rappresentate nell'albero (cioè, mammiferi e uccelli condividono un antenato che non è comune ad altre specie nell'albero). La filogenesi afferma inoltre che tutti gli animali discendono da un antenato non condiviso con i funghi, gli alberi e i batteri, e così via.

Problemi su grafi

- Problema della connettività tra s e t. Dati due nodi s e t, esiste un percorso tra s e t?

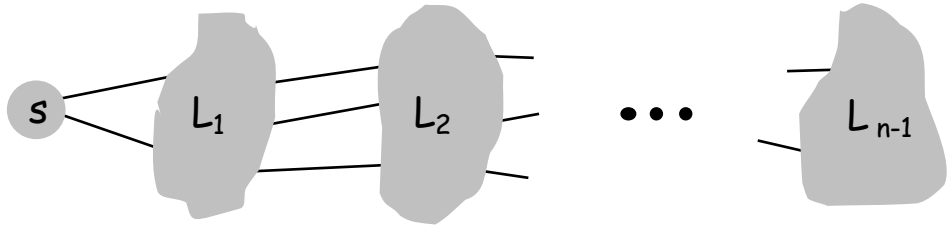
- Problema del percorso più corto tra s e t. Dati due nodi s e t, qual è la lunghezza del percorso più corto tra s e t?



- Applicazioni.
  - Attraversamento di un labirinto.
  - Erdős number.
  - Minimo numero di dispositivi che devono essere attraversati dai dati in una rete di comunicazione per andare dalla sorgente alla destinazione

Breadth First Search (visita in ampiezza)

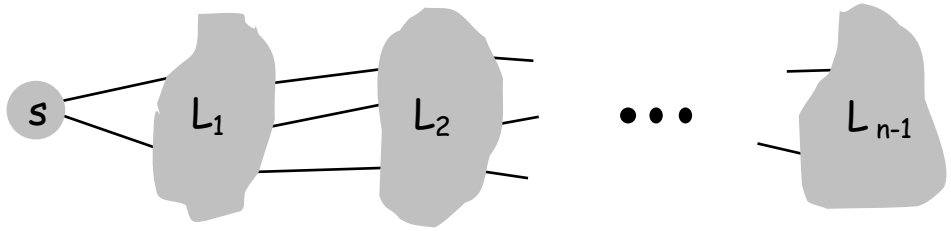
- BFS. Explora il grafo a partire da una sorgente s muovendosi in tutte le possibile direzioni e visitando i nodi livello per livello (N.B.: il libro li chiama layer e cioè strati).



- Algoritmo BFS.
  - $L_0 = \{ s \}$.
  - L_1 = tutti i vicini di s.
  - L_2 = tutti i nodi che non appartengono a L_0 or L_1, e che sono uniti da un arco ad un nodo in L_1.
  - L_{i+1} = tutti i nodi che non appartengono agli strati precedenti e che sono uniti da un arco ad un nodo in L_i.

Breadth First Search

- Teorema. Per ogni i, L_i consiste di tutti i nodi a distanza i da s. C'è un percorso da s a t se e solo t appare in qualche livello.



L_1: livello dei nodi a distanza 1 da s
L_2: livello dei nodi a distanza 2 da s
...
L_{n-1}: livello dei nodi a distanza n-1 da s

Breadth First Search

Pseudocodice

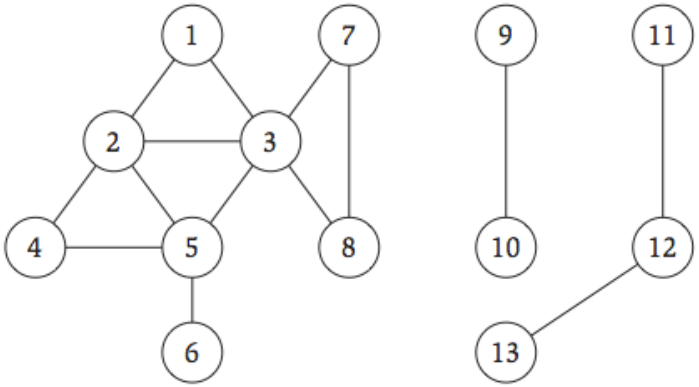
1. BFS(s)
2. L_0={ s }
3 **For**(i=0;i≤n-2;i++)
4. L_{i+1}= ∅ ;
5. **For each** nodo u in L_i
6. **For each** nodo v adiacente ad u
7. **if(** v non appartiene ad $L_0,\dots,L_i$)
8. $L_{i+1}=L_{i+1} \cup$ { v }
9. **EndIf**
10. **Endfor**
11. **Endfor**

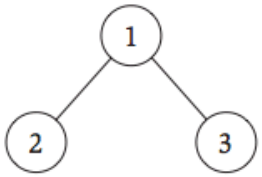
queste due linee sono eseguite $\sum_{u \in V} deg(u)$ volte

Occorre un modo per capire se un nodo è gia stato visitato in precedenza. Il tempo di esecuzione dipende dal modo scelto, da come è implementato il grafo e da come sono rappresentati gli insiemi L_i che rappresentano i livelli

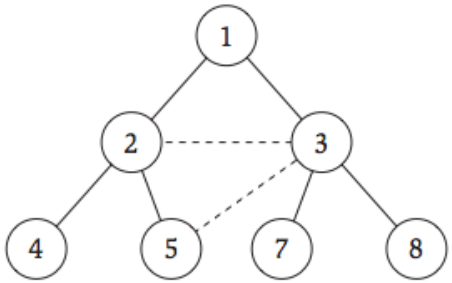
Esempio di esecuzione di BFS

G

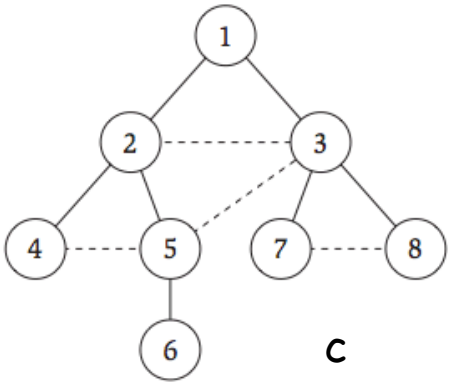




a



b



c

L_0={1}

a. L_1={2,3}

b. L_2={4,5,7,8}

c. L_3={6}

Breadth First Search Tree (Albero BFS)

- Proprietà. L'algoritmo BFS 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 un vertice v viene scoperto, e cioè il momento in cui visitato per la prima volta.
  - Ciò avviene durante l'esame dei vertici adiacenti ad un certo vertice u di un certo livello L_i (linea 6).
  - In questo momento, oltre ad aggiungere v al livello L_{i+1} (linea 8), aggiungiamo l'arco (u,v) e il nodo v all'albero

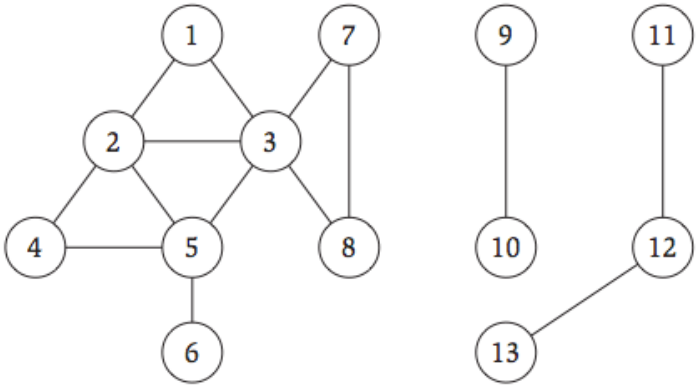
Implementazione di BFS

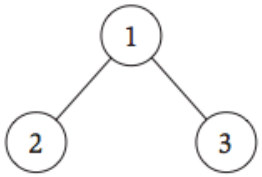
- Ciascun insieme L_i è rappresentato da una lista L[i]
- Usiamo un array di valori booleani Discovered per associare a ciascun nodo il valore vero o falso a seconda che sia già stato scoperto o meno
- Durante l'algoritmo costruiamo anche l'albero BFS

```
1.  BFS(s):
2.  Poni Discovered[s] = true e Discovered[v] = false per tutti gli altri v
3.  Inizializza L[0] in modo che contenga solo s
4.  Poni il contatore dei livelli i = 0
5.  Inizializza il BFS tree T con un albero vuoto
6.  While L[i] non è vuota                          //L[i] è vuota se non ci sono
7.      Inizializza L[i+1] con una lista vuota      //nodi raggiungibili da L[i-1]
8.      For each u ∈ L[i]
9.      Considera ogni arco (u, v) incidente su u
10.       If Discovered[v] = false then
11.           Poni Discovered[v] = true
12.           Inserisci v in L[i+1]
13.           Aggiungi l'arco (u, v) all'albero T
14.     Endif
15.   Endfor
16.  i=i+1
17. Endwhile
```

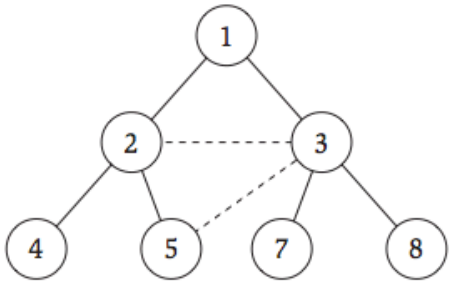
Esempio di esecuzione di BFS

G

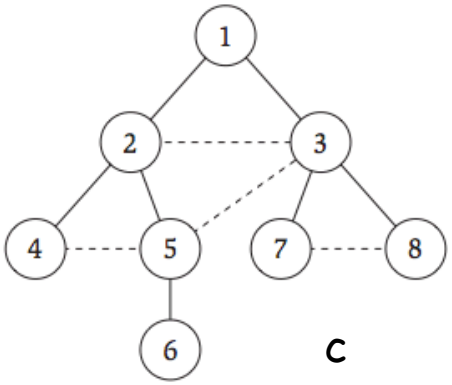




a



b



c

L_0={1}

a. L_1={2,3} aggiunge gli archi (1,2) e (1,3) all'albero

b. L_2={4,5,7,8} aggiunge gli archi (2,4), (2,5), (3,7), (3,8) all'albero

c. L_3={6} aggiunge l'arco (5,6) all'albero

Analisi di BFS nel caso in cui il grafo è rappresentato con liste di adiacenza

1. BFS(s):
2. Poni Discovered[*s*] = true e Discovered[*v*] = false per tutti gli altri *v*
3. Inizializza *L*[0] in modo che contenga solo *s*
4. Poni il contatore dei livelli *i* = 0
5. Inizializza il BFS tree *T* con un albero vuoto
6. **While** *L*[*i*] non è vuota
7. Inizializza *L*[*i*+1] con una lista vuota
8. **For each** $u \in L[i]$
9. Considera ogni arco (*u*, *v*) incidente su *u*
10. **If** Discovered[*v*] = false **then**
11. Poni Discovered[*v*] = true
12. Inserisci v in L[i+1]
13. Aggiungi l'arco (*u*, *v*) all'albero *T*
14. *Endif*
15. *Endfor*
16. *i=i+1*
17. *Endwhile*

O(n)

O(1)

Al più n-1 volte

O(n)

Sul totale di tutte le iterazioni del while, al più n volte

Sul totale di tutte le iterazioni dei cicli più esterni, al più 2m volte.

O(m)

For each più esterno in totale viene eseguito al più n volte in quanto ogni nodo appartiene ad una sola lista L_i

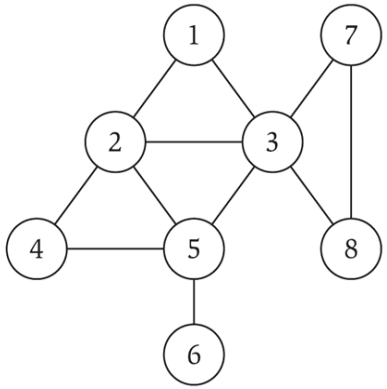
For each più interno in totale viene eseguito $\sum_{i=0}^{n} \sum_{u \in L[i]} \deg(u) \leq \sum_{u \in V} \deg(u) \leq 2m$ volte

Algoritmo è O(n+m)

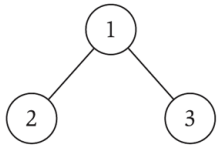
Breadth First Search Tree

- Proprietà. Sia T un albero BFS di G = (V, E), e sia (x, y) un arco di G. I livelli di x e y differiscono di al più di 1.

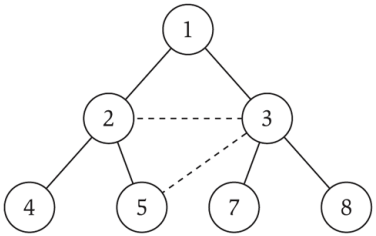
- Dim. Sia L_i il livello di x ed L_j quello di y. Supponiamo senza perdere di generalità che x venga scoperto prima di y cioè che $i \leq j$. Se i=j la proprietà è banalmente soddisfatta. Supponiamo quindi che $i<j$ e consideriamo il momento in cui l'algoritmo esamina gli archi incidenti su x. Tra gli archi esaminati c'è anche (x,y) e poichè y non è stato ancora scoperto (altrimenti $j \leq i$), y viene inserito in L_{i+1} per cui j=i+1.



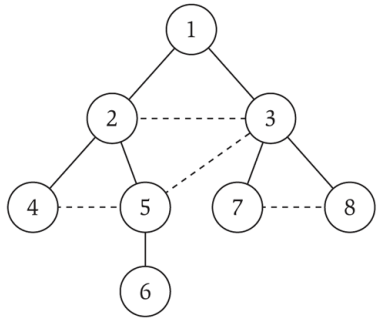
G



(a)



(b)



(c)

Implementazione di BFS con coda FIFO

- L'algoritmo BFS si presta ad essere implementate con un coda

- Ogni volta che viene scoperto un nodo u, il nodo u viene inserito nella coda
- Vengono esaminati gli archi incidenti sul nodo al front della coda

BFS(s)

1. Inizializza Q con una coda vuota
2. Poni Discovered[*s*] = true e Discovered[*v*] = false per tutti gli altri *v*
3. Inserisci s in coda a Q con una enqueue
4. While(Q non è vuota)
5. estrai il front di Q con una deque e ponilo in u
6. For each arco (u,v) incidente su u
7. If(Discovered[v]=false)
8. Poni Discovered[v]= true
9. aggiungi v in coda a Q con una enqueue
10. aggiungi (u,v) al BFS tree
11. Endif
12. Endwhile
13. Endwhile

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 vistata. A questo punto torna nella camera precedentemente visitata e di lì prova a raggiungere nuove camere.

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, l'algoritmo segue uno degli archi uscenti da v. Se l'arco 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.

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
```

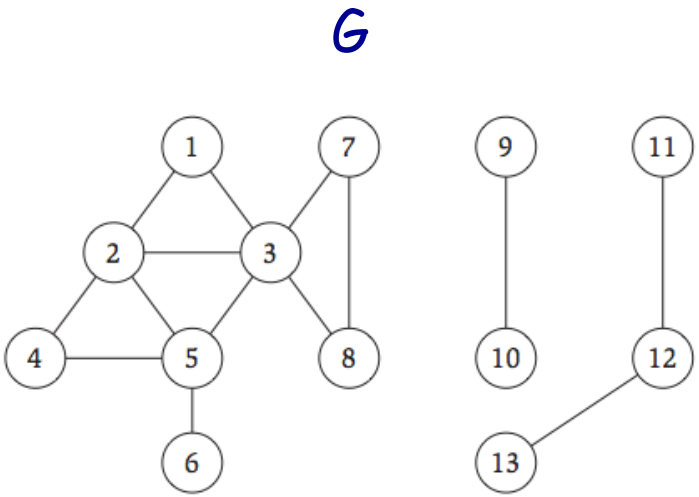
R = insieme dei vertici raggiunti

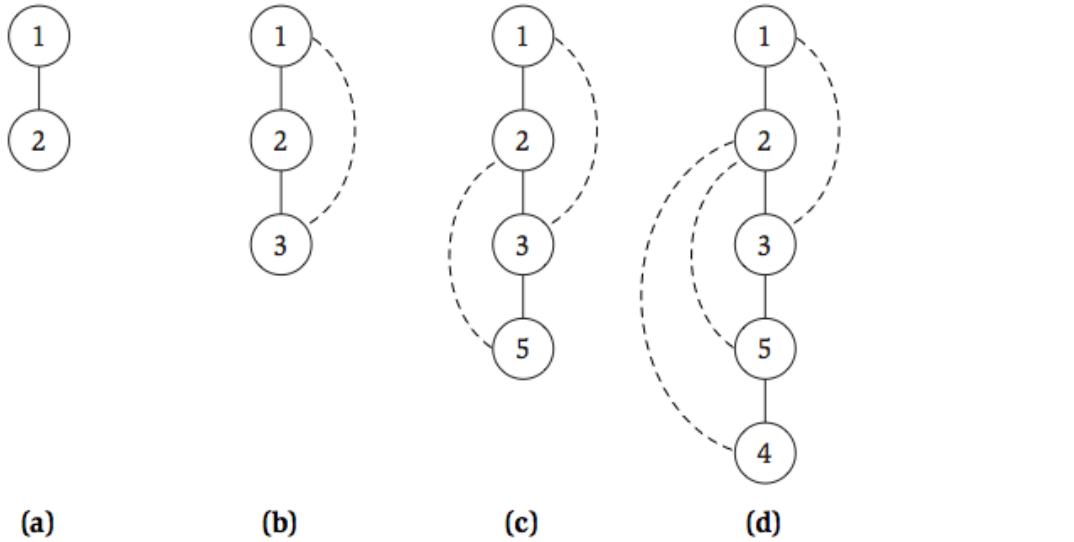
Analisi:

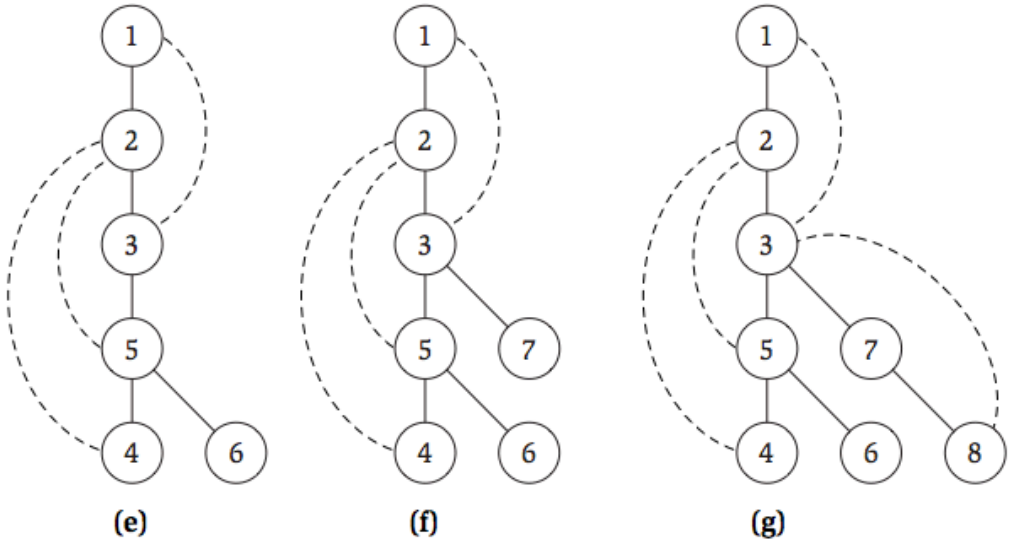
Ciascuna visita ricorsiva richiede tempo O(1) per marcare u e aggiungerlo ad R e tempo O(deg(u)) per eseguire il for.

Se inizialmente invochiamo DFS su un nodo s, allora DFS viene invocata ricorsimante su tutti i nodi raggiungibili a partire da s. Il costo totale è quindi $\sum_{u \in V} O(1) + \sum_{u \in V} O(\deg(u)) = O(n) + O(m) = O(n+m)$

Esempio di esecuzione di DFS







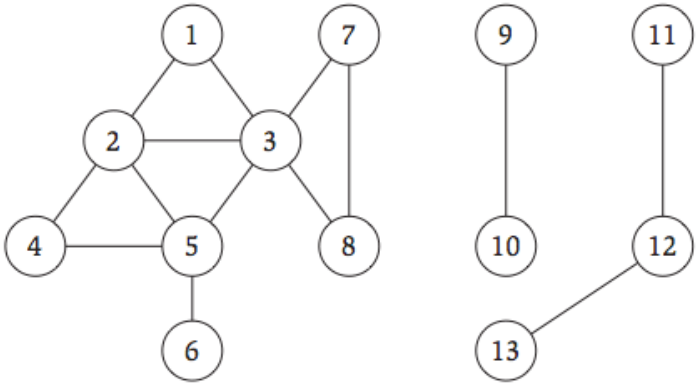
Depth First Search Tree (Albero DFS)

- 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.

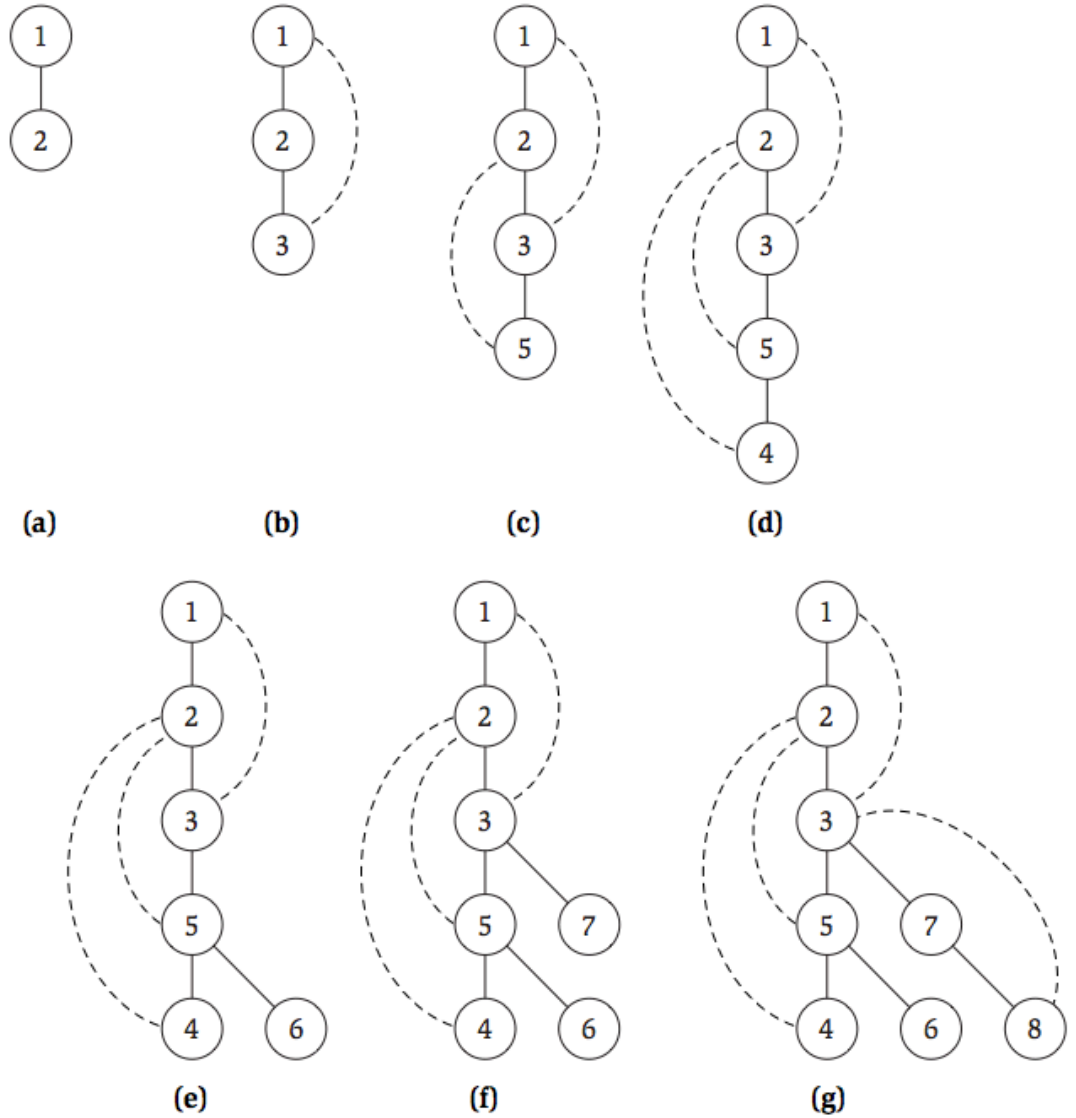
- L'albero si ottiene in questo modo:
- Consideriamo il momento in cui viene invocata DFS(v)
  - Ciò avviene durante l'esecuzione di DFS(u) per un certo nodo u. In particolare durante l'esame dell'arco (u,v) nella chiamata DFS(u).
  - In questo momento, aggiungiamo l'arco (u,v) e il nodo v all'albero

Esempio di albero DFS

G



Gli archi non tratteggiati fanno parte del DFS tree.



Albero DFS

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

- Dim. 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 precedente implica che y è discendente di x.

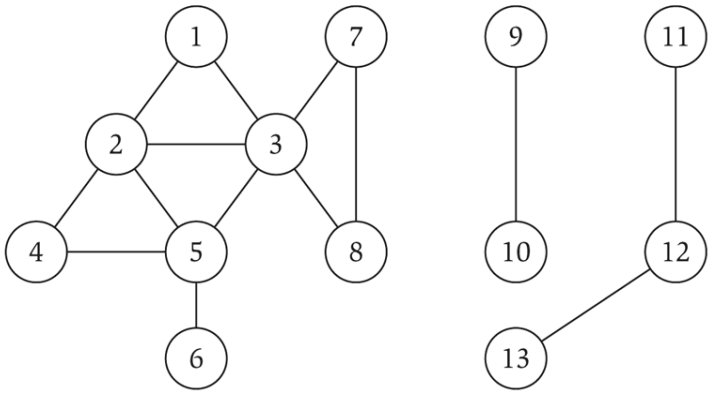
Implementazione di DFS mediante uno stack

```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
       Set Explored[u] = true
       For each edge (u, v) incident to u
         Add v to the stack S
       Endfor
    Endif
  Endwhile
```

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

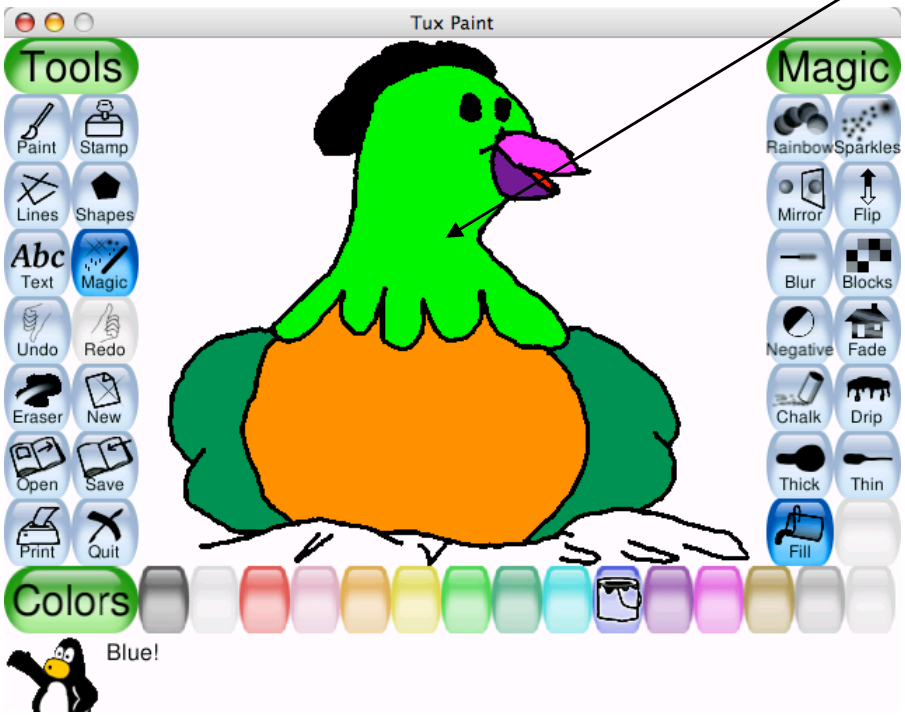


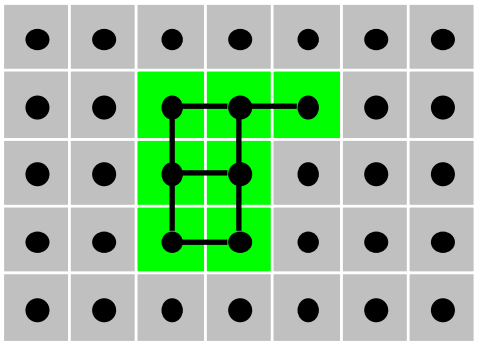
- Componente connessa contenente il nodo 1 è { 1, 2, 3, 4, 5, 6, 7, 8 }.

Flood Fill

- **Flood fill.** Data un'area di pixel di colore verde lime in 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

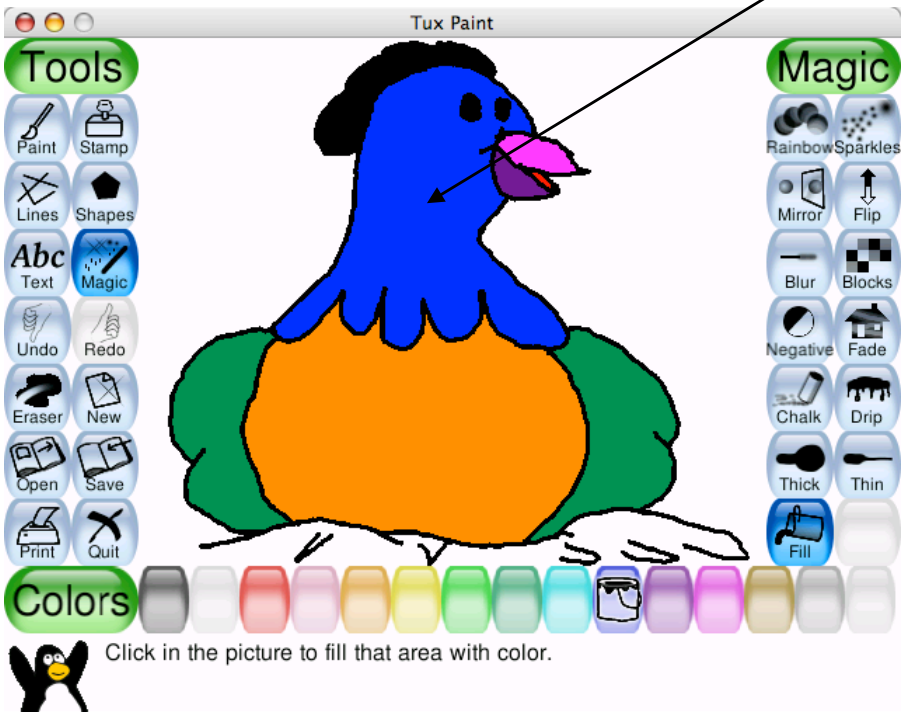


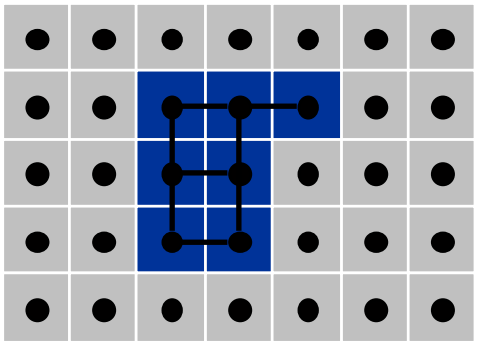


Flood Fill

- **Flood fill.** Data un'area di pixel di colore verde lime in 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.

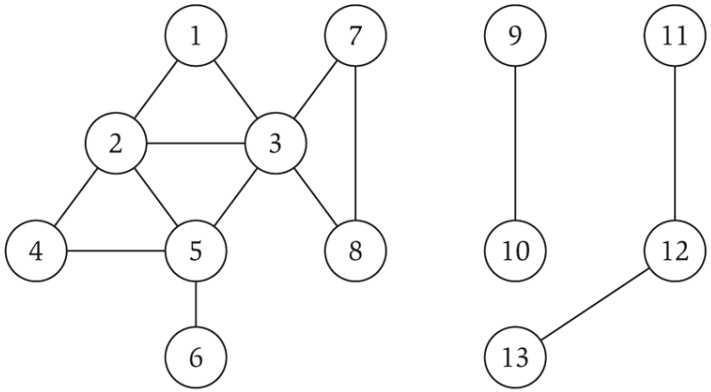
Ricolora l'area verde lime di blue





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
- 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

- 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)
- For each node u of G
- If Discovered[u]= false
- BFS(u)
- Endif
- Endfor

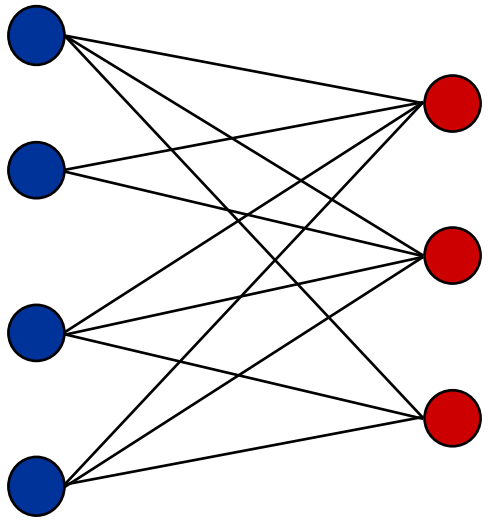
O(n+m)

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

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.

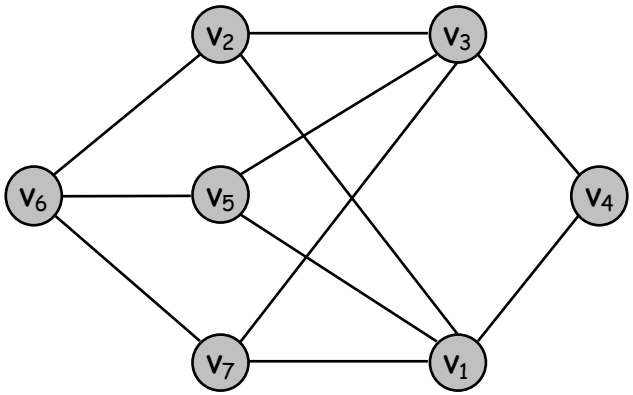
- Applicazioni.
  - Matrimoni stabili: uomini = rosso, donna = blu.
  - Scheduling: macchine = rosso, job = blu.



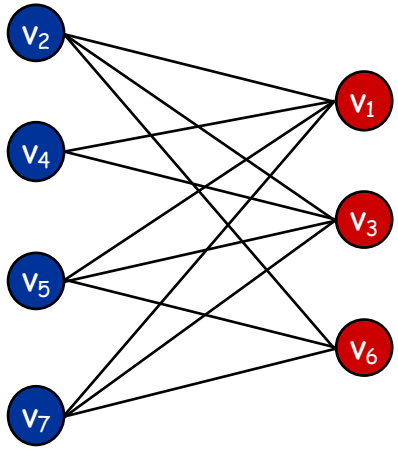
Un grafo bipartito

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)



Un grafo bipartito G

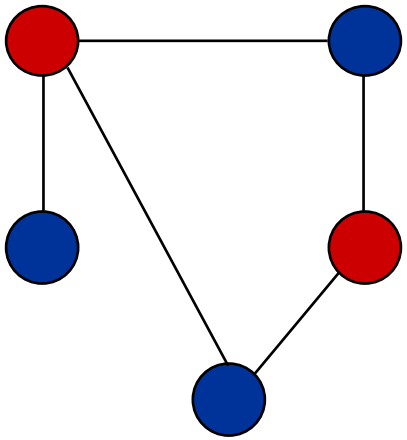


Modo alternativo di disegnare G

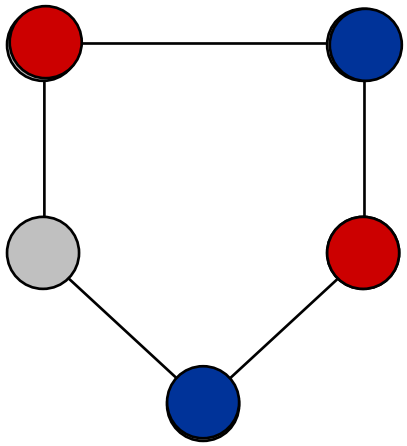
Grafi bipartiti

- 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.



bipartito
(2-colorabile)



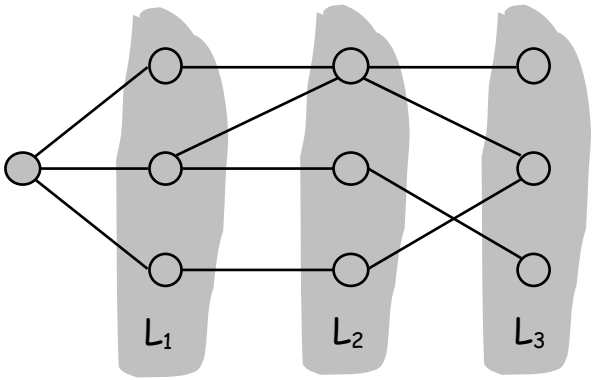
Non bipartito
(non 2-colorabile)

Grafi bipartiti

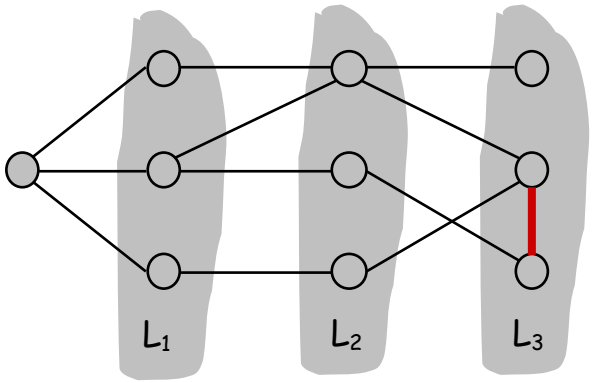
- Osservazione. Sia G un grafo connesso e siano $L_0, \ldots, L_k$ i livelli prodotti da un'esecuzione di BFS a partire dal nodo s. Può avvenire o che si verifichi la (i) o la (ii)

(i) Nessun arco di G collega due nodi sullo stesso livello

(ii) Un arco di G collega due nodi sullo stesso



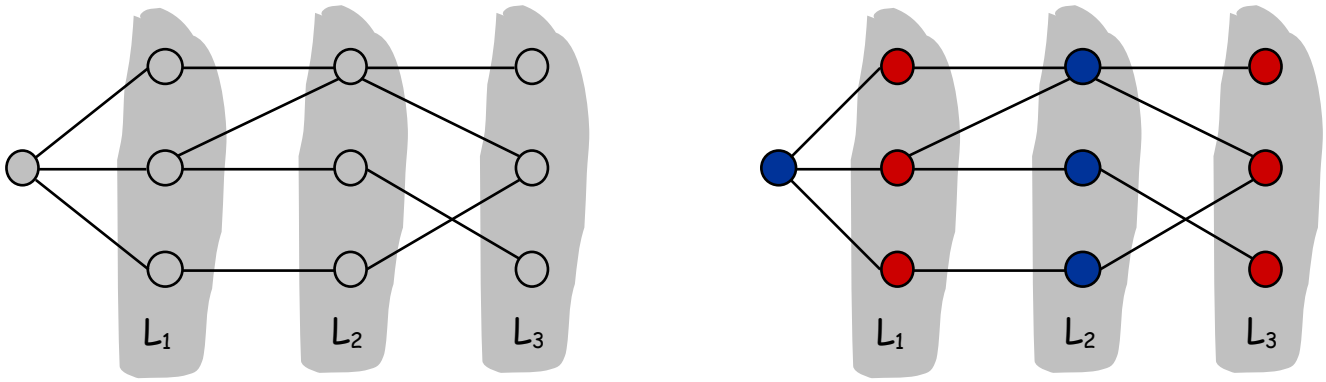
Case (i)



Case (ii)

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 pari di rosso e quelli di indice dispari di blu, ho che le estremità di ogni arco sono di colore diverso.

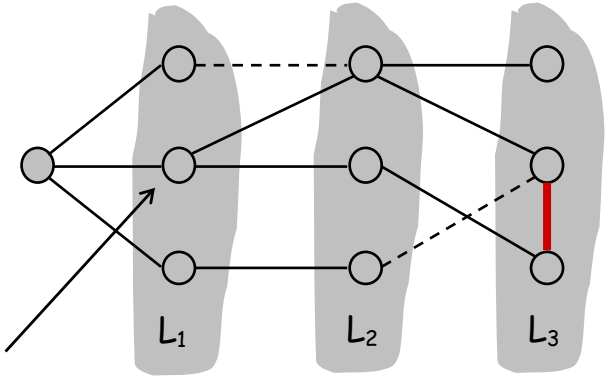


Caso (i)

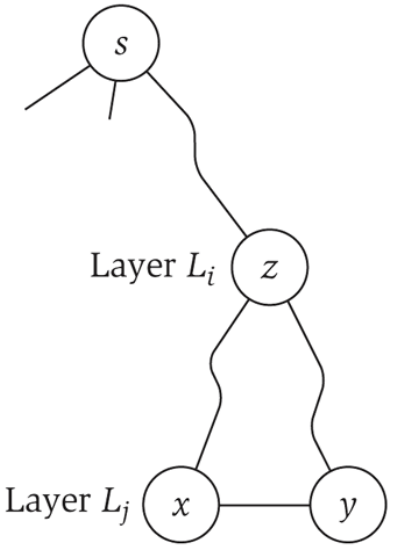
Grafi Bipartiti

Nel caso (ii) il grafo non è bipartito.

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

- 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?

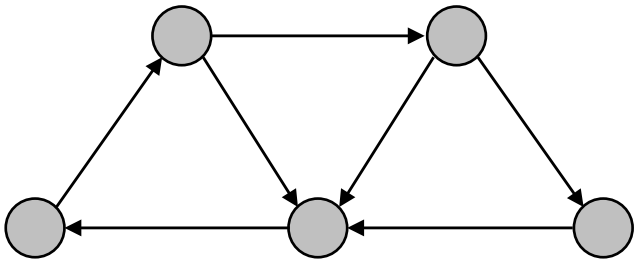
- Visita di un grafo. La BFS si estende naturalmente ai grafi direzionati.
  - Quando si esaminano gli archi incidenti su un certo vertice u, si considerano solo quelli uscenti da u.

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

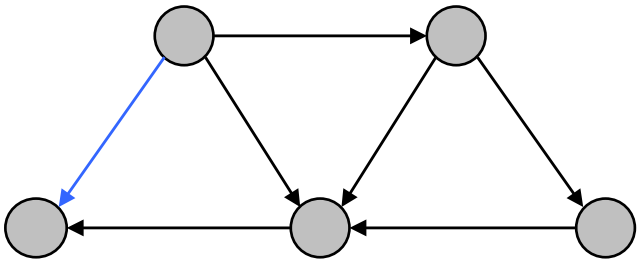
Connettività forte

- 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



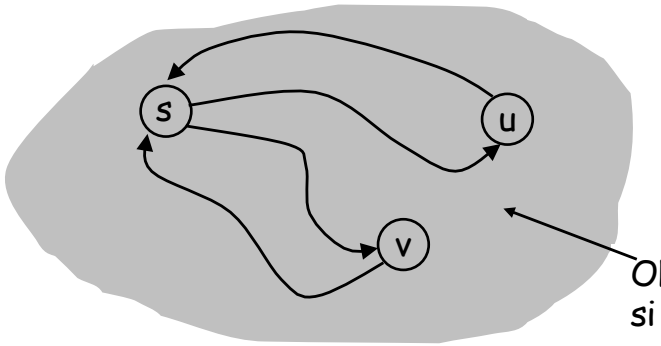
Fortemente connesso



Non fortemente connessog

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.



Ok anche se i due percorsi si sovrappongono

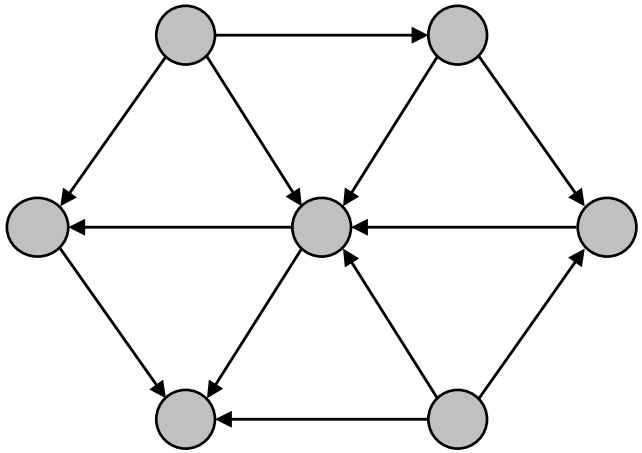
Algoritmo per la connettività forte

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

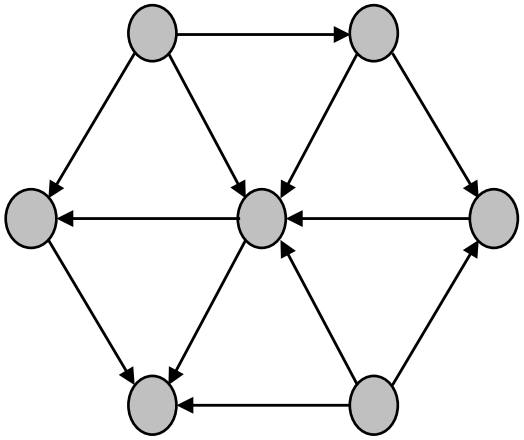


Un DAG G

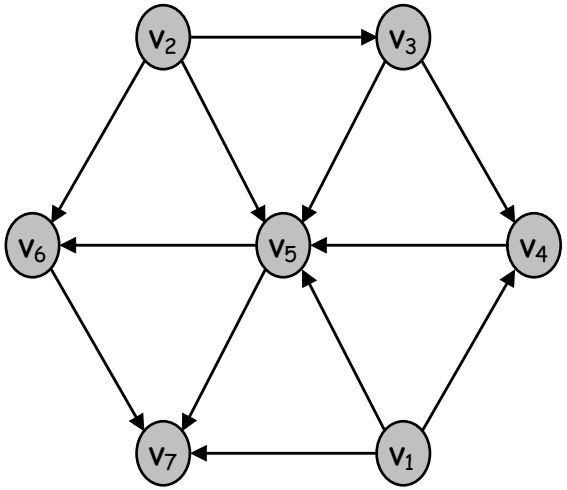
Ordine topologico

- 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).

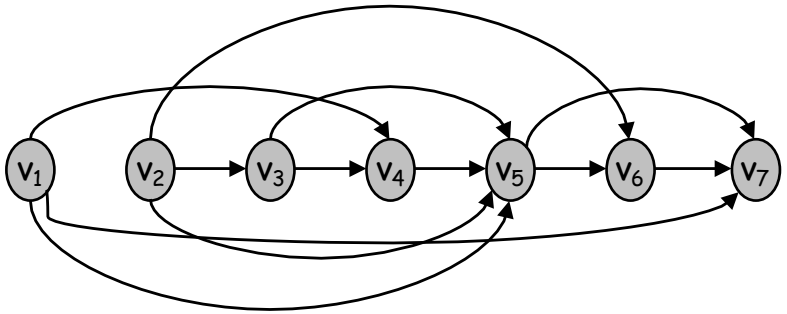
- Esempio. Nel caso in cui un grafo direzionato G rappresenti le propedeuticità degli esami, un ordinamento topologico indica un possibile ordine in cui gli esami possono essere sostenuti dallo studente.



Un DAG G



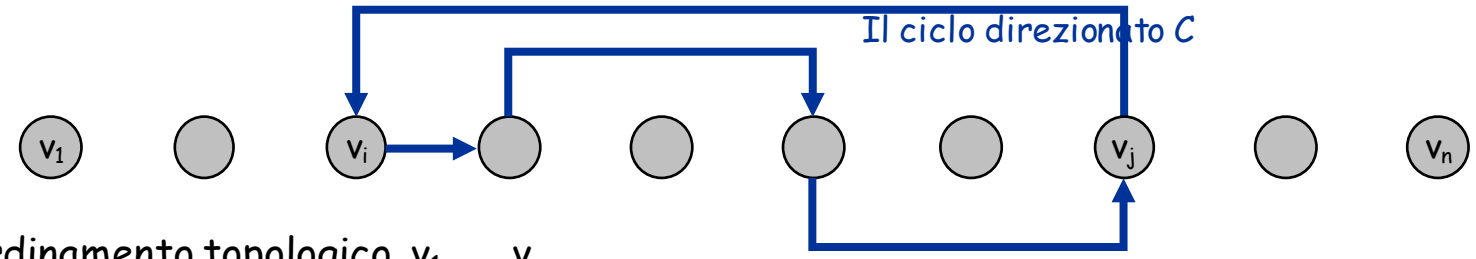
Un ordinamento topologico di G



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

DAG e ordinamento topologico

- Lemma. Se 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.
  - Per come abbiamo scelto i, abbiamo $i < j$.
  - D'altra parte, siccome (v_j, v_i) è un arco e $v_1, \ldots, v_n$ è un ordinamento topologico allora, deve essere $j < i$, che è una contraddizione al fatto che G contiene un ciclo.



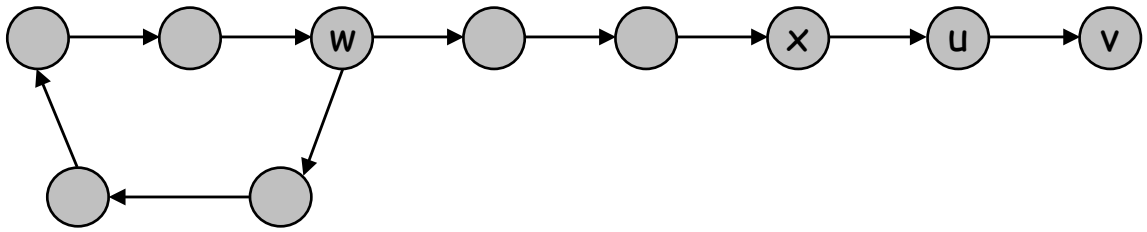
L'ordinamento topologico $v_1, \ldots, v_n$

DAG e ordinamento topologico

- Abbiamo visto che se G ha un ordinamento topologico allora G è un DAG.
- Domanda. 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

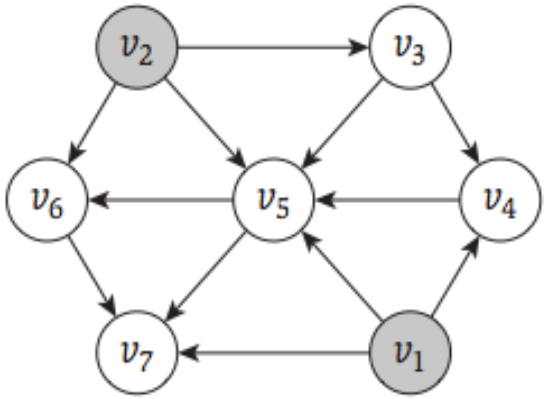
- Lemma. Se G è un DAG, G ha un ordinamento topologico.
- Dim. (induzione su n)
  - Caso base: vero banalmente se n = 1.
  - Passo induttivo: supponiamo asserto del lemma vero per DAG con n≥1 nodi
  - Dato un DAG con n+1 > 1 nodi, prendiamo un nodo v senza archi entranti (abbiamo dimostrato che un tale 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 quello che si ottiene è un ordinamento topologico (tutti gli archi puntano in avanti).

Algoritmo per l'ordinamento topologico

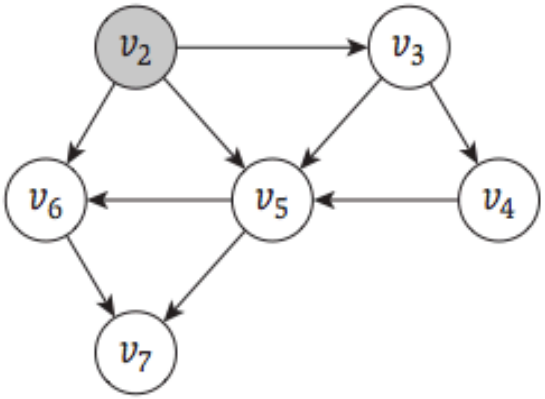
- La dimostrazione per induzione che abbiamo appena visto suggerisce un algorimo ricorsivo per trovare l' ordinamento topologico di un DAG.

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

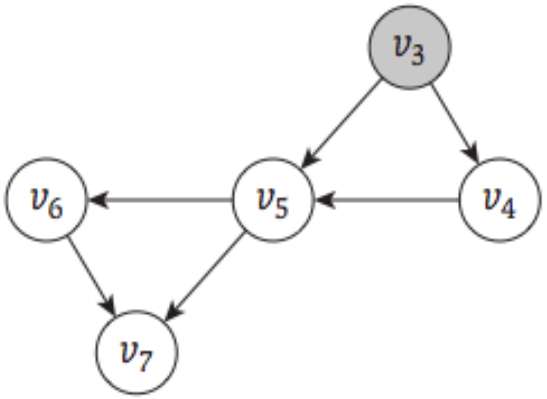
Algoritmo per l'ordinamento topologico



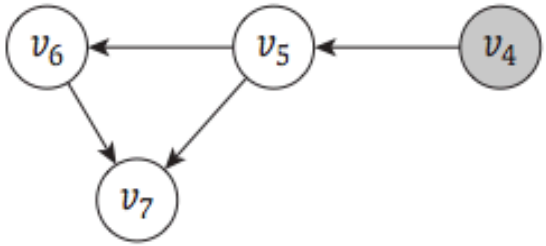
(a)



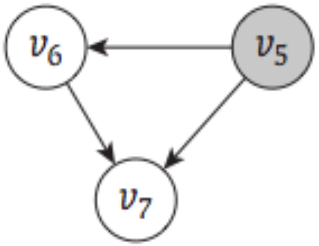
(b)



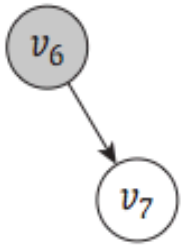
(c)



(d)



(e)



(f)

Algoritmo per l'ordinamento topologico : analisi dell'algoritmo

- Trovare un nodo senza archi entranti richiede $O(n)$
- Cancellare un nodo v da G richiede tempo proporzionale al numero di archi uscenti da v che è **al più** deg(v)=O(n)
- Se consideriamo tutte le n chiamate ricorsive il tempo è $O(n^2)$

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 $O(n+dev(v))=O(n)$, 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

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

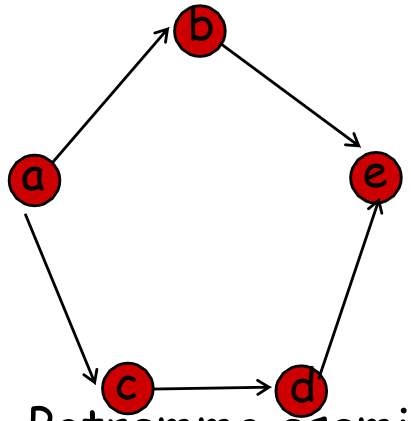
- Teorema. L'algoritmo trova l'ordinamento topologico di un DAG in tempo $O(m + n)$.
- Dim.
  - Inizializzazione. Richiede tempo $O(m + n)$ in quanto
    - Inizialmente tutti i nodi sono attivi per cui S consiste dei nodi di G senza archi entranti➔ basta scandire tutti i nodi una sola volta per inizializzare S.
  - 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.
  - Aggiornamento. Per trovare il nodo v da cancellare basta prendere un nodo da S. Per cancellare v occorre
    1. Cancellare v da S. Tempo $O(1)$
    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

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

Esercizio 1 su grafi

- Fornire tutti gli ordinamenti topologici del grafo sottostante



- Soluzione. Potremmo esaminare le 5!= 120 possibili permutazioni….
- Ragioniamo: il primo nodo dell'odinamento 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

Esercizio 2 su Grafi

1. Due robot, il primo parte dal punto a e deve arrivare nel punto c mentre il secondo parte dal punto b e deve arrivare nel punto d.
2. In ogni momento i due robot devono trovarsi a distanza maggiore di r per evitare interferenze tra i trasmettitori che utilizzano per comunicare con la stazione base.

- La pianta del piano in cui si muovono i robot può essere rappresentata da un grafo G in cui i nodi rappresentano i vari punti del piano e gli archi uniscono due punti adiacenti
- Se non vi fosse il punto 2., il problema consisterebbe semplicemente nell'assegnare a ciascun robot degli istanti in cui il robot si muove da una certa posizione ad una adiacente e l'altro sta fermo, in modo tale che alla fine il primo robot venga a trovarsi in c e il secondo in d.

Continua nella prossima slide

Esercizio 2 su Grafi

- Il punto 2 impone che le rispettive posizioni dei robot siano sempre a distanza maggiore di r
- Per poter risolvere il problema costruiamo un grafo H come segue:
- I nodi di H sono coppie (u,v), dove u è la posizione del primo robot e v quella del secondo robot
- Due nodi (u,v) e (u',v') di H sono collegati se u=u' e (v,v') è un arco in G oppure v=v' e (u,u') è un arco in G. Questo perché ad ogni passo, uno dei due robot si sposta in una posizione adiacente a quella in cui si trovava prima mentre l'altro robot rimane fermo.
- Un percorso in H dalla configurazione (a,b) alla configurazione (c,d) è una sequenza di passi che porta i robot nelle posizioni desiderate

Continua nella prossima slide

Esercizio 2 su Grafi

- Un percorso in H dalla configurazione (a,b) alla configurazione (c,d) però non rispetta necessariamente il vincolo 2. che impone che i due robot possono trovarsi nella configurazione (u,v) solo se la distanza tra u e v è maggiore di r.
- Per essere certi di trovare un percorso in H che soddisfi il vincolo 2., eliminiamo da H tutti i nodi (u,v) con u e v che si trovano a distanza al più r. Chiamiamo H' il grafo risultante.
- Una volta costruito H', eseguiamo una visita BFS o DFS a partire dal nodo sorgente (a,b). Se nel visitare H' raggiungiamo (c,d) allora vuol dire che è possibile far arrivare i due robot nei punti desiderati senza che vi sia mai interferenza tra i loro trasmettitori.

Continua nella prossima slide

Esercizio 2 su Grafi

- Analisi dell'algoritmo:
- La creazione di H richiede tempo $O(n^2)$ per creare i nodi e $O(n^3)$ per creare gli archi:
- Vi sono al più n^2 nodi
- Ogni nodo (u,v) ha un numero di archi uscenti della forma (u,v') pari al numero di archi uscenti da u in G e ha un numero di archi uscenti della forma (u',v) pari al numero di archi uscenti da v in G. Ne consegue che per ogni nodo (u,v) vi è un a numero di archi uscenti pari al più $\deg(u)+\deg(v) \leq n$. In totale vi sono quindi n^3 archi in H.
- Per cancellare da H i nodi (u,v) con u e v a distanza al più r, eseguiamo BFS(u) in G per ogni nodo u e creiamo la lista di tutti i nodi a distanza al più r da u. Per ogni nodo v in questa lista, rimuoviamo il nodo (u,v) da H. La cancellazione di ciascun nodo (u,v) richiede tempo O(n) perché ci sono al più n archi incidenti (entranti e uscenti) su (u,v). In totale la costruzione di H' a partire da H, richiede $O(n(n+m))=O(nm+n^2)$
- Il tempo per la visita di H' richiede $O(n^2+n^3)= O(n^3)$

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

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

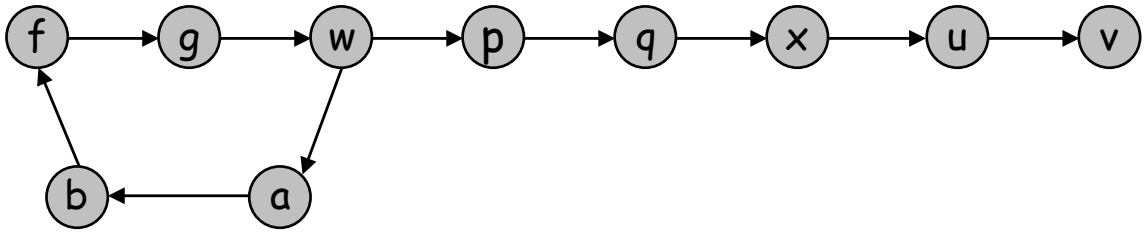
Esercizio 3 Cap. 3

- 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.