

Grafi

(Parte 1)

I GRAFI sono utilizzati d. b. to per
rappresentare **NETWORKED DATA**, cioè
dati in cu. l'informazione è costituita da

(a) SINGOLI ELEMENTI

(b) RELAZIONI TRA (COPPIE DI) ELEMENTI

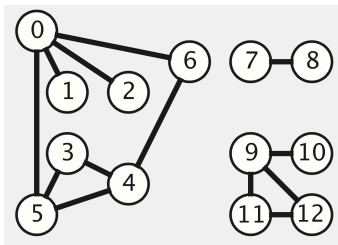
Definizione di Grafo

Definizione

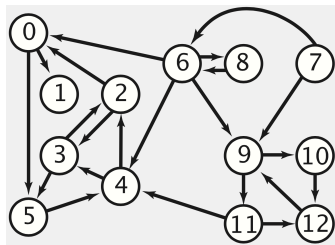
Grafo $G = (V, E)$:

- $V \equiv$ insieme di **vertici** (o **nodì**)
- $E \equiv$ collezione di **archi** (coppie di vertici).

Il grafo si dice **diretto** se ogni arco $(u, v) \in E$ è una coppia ordinata ($u \rightarrow v$), altrimenti si dice **non diretto** ($u - v$)



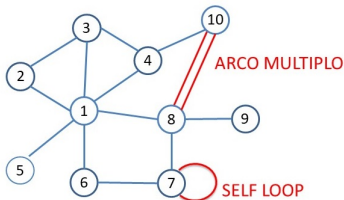
Grafo *non diretto*



Grafo *diretto*

Osservazioni sulla definizione

- La definizione ammette la presenza di **archi multipli** tra due vertici (per questo E è una *collezione* e non un *insieme*), e di archi (u, u) (**self loop**).



- Un **grafo semplice** è un grafo senza archi multipli (in questo caso E è un *insieme*) e senza self loop
- Per alcune applicazioni agli archi (e a volte ai vertici) sono associati dei **pesi**.

Esempi di grafi reali: RETI SOCIALI



FACEBOOK

Vertici $\equiv \{ \text{Prof. l.} \}$

Arch. $\equiv \{ \text{Amicizie} \}$

Grafo diretto? NO

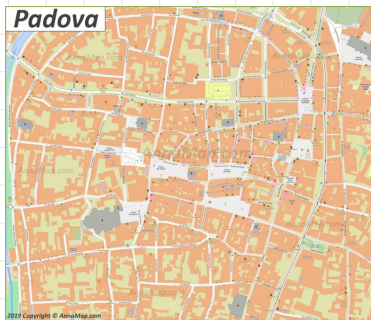
INSTAGRAM

Vertici $\equiv \{ \text{Prof. l.} \}$

Arch. $\equiv \{ (u, v) : u \text{ knows } v \}$

Grafo diretto? SI

Esempi di grafi reali: RETI STRADALI



Vertici $\equiv$ {Incroc.}

Archi $\equiv$ {segmenti di strade tra incroci}

Grafo diretto? Sì, se consideriamo i sensi unici

Altri esempi di grafi reali

- Internet
- Web
- Reti di comunicazione per supercomputer
- Reti peer-to-peer
- Reti wireless, reti di sensori
- Protein-protein network

Grafi Semplici e Non Diretti

Terminologia

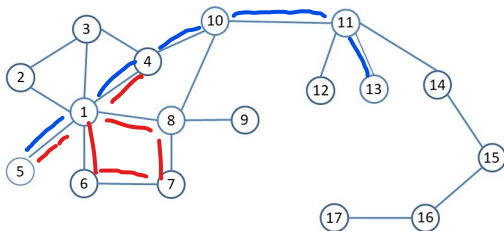
- **Inglese:** per nodi/vertici si usa **nodes/vertices** ([GTG14] usa vertices), per archi si usa **edges/arcs** ([GTG14] usa edges).

Di solito si usa edges per grafi non diretti e arcs per grafi diretti.

- Dato un arco $e = (u, v) \in E$, si dice che e è **incidente** su u e v e che u e v sono **adiacenti**.
- I **vicini** di un vertice v sono tutti i vertici u tali che $(v, u) \in E$.
- Il **grado** di un vertice $v \in V$ ($\text{degree}(v)$) è il numero di archi di E incidenti su v .

Concetti fondamentali: cammino

Grafo $G = (V, E)$:



Cammino (path): $u_1, u_2, \dots, u_k$ con $(u_i, u_{i+1}) \in E$ per $1 \leq i < k$.

La **lunghezza del cammino** è il numero di archi (u_i, u_{i+1}) , ovvero $k - 1$. Nel caso di archi pesati, la lunghezza è la somma dei pesi degli archi (u_i, u_{i+1}) .

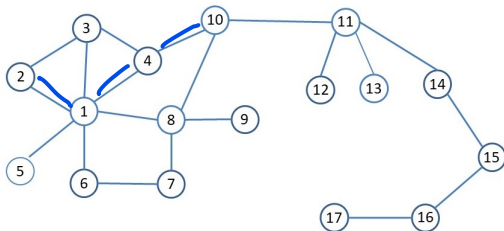
Oss.: il cammino si dice **semplice** se non ha vertici ripetuti. Di solito si omette l'aggettivo semplice anche se si parla di cammino semplice.

Esempio di cammino semplice: 5 1 4 10 11 13

Esempio di cammino non semplice: 5 1 6 7 8 1 4

Concetti fondamentali: cammino minimo e distanza

Grafo $G = (V, E)$:

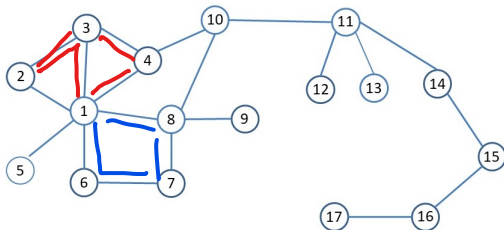


Dati due vertici $x, y \in V$, il **cammino minimo (shortest path)** tra x e y , se ne esiste uno, è un cammino $x = u_1, u_2, \dots, u_k = y$ di lunghezza minima. La sua lunghezza è detta **distanza** $d(x, y)$ tra x e y . Se non esiste alcun cammino tra x e y , per convenzione si considera $d(x, y) = +\infty$.

Cammino minimo da $x = 2$ a $y = 10$: 2 1 4 10

Concetti fondamentali: ciclo

Grafo $G = (V, E)$:



Ciclo (cycle): cammino $u_1, u_2, \dots, u_k$ con $u_1 = u_k$.

La **lunghezza del ciclo** è quella del cammino $u_1, u_2, \dots, u_k$. ($k - 1$)

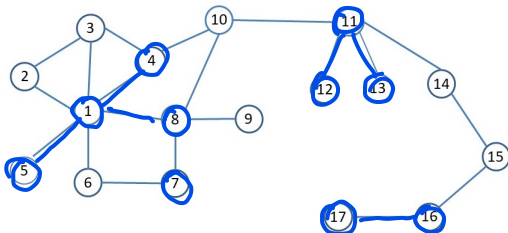
Oss.: il ciclo si dice **semplice** se non ha vertici ripetuti tranne gli estremi. Di solito si omette l'aggettivo semplice anche se si parla di ciclo semplice.

Ciclo semplice: 1 6 7 8 1

Ciclo non semplice: 2 3 1 4 3 2

Concetti fondamentali: sottografo

Grafo $G = (V, E)$:

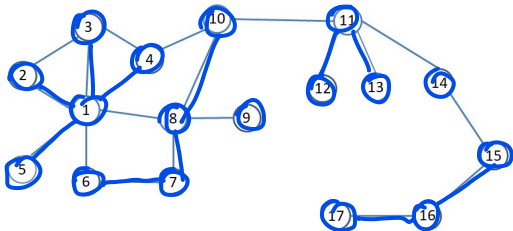


Sottografo (subgraph): $G' = (V', E')$ con $V' \subseteq V$, $E' \subseteq E$, e tale che gli archi di E' incidono solo su V' .

Esempio: vertici e archi blu

Concetti fondamentali: sottografo di copertura

Grafo $G = (V, E)$:

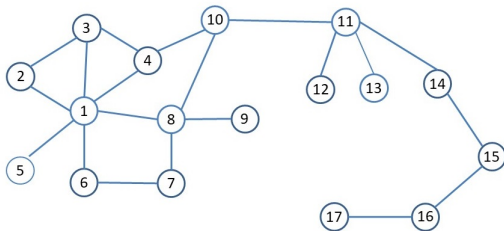


Sottografo di copertura (spanning subgraph): sottografo $G' = (V', E')$ con $V' = V$.

Esempio: vertici e archi blu

Concetti fondamentali: grafo connesso

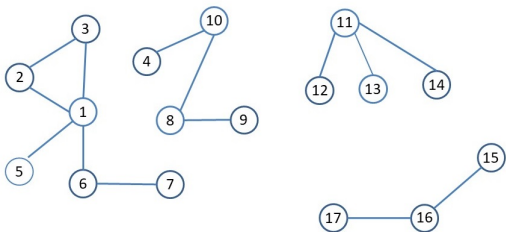
Grafo $G = (V, E)$:



Un grafo $G = (V, E)$ si dice **connesso (connected)** se per ogni $u, v \in V$ esiste un cammino che inizia in u e termina in v .

Concetti fondamentali: grafo disconnesso

Grafo $G = (V, E)$:



Un grafo $G = (V, E)$ si dice **disconnesso (disconnected)** se non è connesso, ovvero esistono due vertici $u, v \in V$ per i quali non esiste un cammino che inizia in u e termina in v .

Nel graf in figura non esiste alcun cammino tra 7 e 8

Concetti fondamentali: componenti connesse

Le **componenti connesse** (**connected component**) di un grafo $G = (V, E)$ sono i *sottografi connessi massimali* di G , ovvero la famiglia di sottografi $G_i = (V_i, E_i)$, con $1 \leq i \leq k$, tali che

- $G_i = (V_i, E_i)$ è connesso, per $1 \leq i \leq k$
- $V = V_1 \cup V_2 \cup \dots \cup V_k$ (partizione: $V_i \cap V_j = \emptyset, \forall i \neq j$)
- $E = E_1 \cup E_2 \cup \dots \cup E_k$ (partizione: $E_i \cap E_j = \emptyset, \forall i \neq j$)
- $\forall i \neq j$, non esistono archi in E tra V_i e V_j

Osservazioni:

- G connesso $\Leftrightarrow k = 1$.
- la partizione di G in componenti connesse è univoca.

Esempio

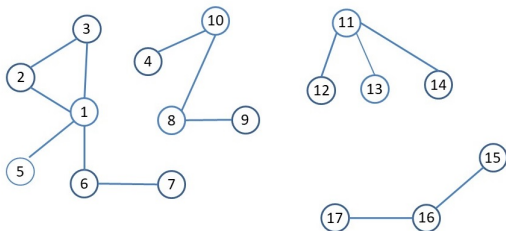


Figure: Grafo $G=(V,E)$

Nella figura le componenti connesse sono 4 con

- $V_1 = \{1, 2, 3, 5, 6, 7\}$ e $E_1 = \{(1, 2), (1, 3), (1, 5), (1, 6), (2, 3), (6, 7)\}$
- $V_2 = \{4, 8, 9, 10\}$ e $E_2 = \{(4, 10), (8, 10), (8, 9)\}$
- $V_3 = \{11, 12, 13, 14\}$ e $E_3 = \{(11, 12), (11, 13), (11, 14)\}$
- $V_4 = \{15, 16, 17\}$ e $E_4 = \{(15, 16), (16, 17)\}$

Concetti fondamentali: albero

Un **albero radicato** (**rooted tree**) è un grafo $G = (V, E)$ tale che

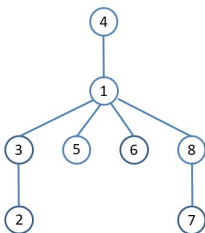
- Esiste un vertice *radice* $r \in V$
- Per ogni $u \in V$, con $u \neq r$, esiste un unico padre $p(u) \in V$, e si ha che $E = \{(u, p(u)) : u \in V, u \neq r\}$
- Per ogni $u \in V$ andando di padre in padre si raggiunge r

Un **albero libero** (**free tree**) è un grafo $G = (V, E)$ connesso e senza cicli

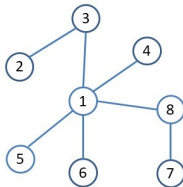
Una **foresta** (**forest**) è un grafo $G = (V, E)$ senza cicli, ovvero un insieme di alberi liberi disgiunti.

Oss.: Si noti che un albero è anche una foresta ma non vale il viceversa

Esempio



Rooted Tree

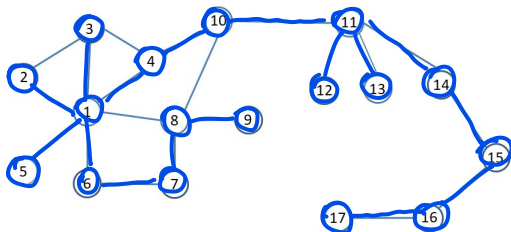


Free Tree

Osservazioni

- Il concetto di albero radicato è lo stesso studiato a inizio corso.
- I concetti di albero radicato e albero libero sono *equivalenti*: ogni albero radicato è un grafo connesso senza cicli, e ogni albero libero, può essere visto come albero radicato scegliendo un vertice *arbitrario* come radice e determinando di conseguenza le relazioni padre-figlio.

Concetti fondamentali: spanning tree

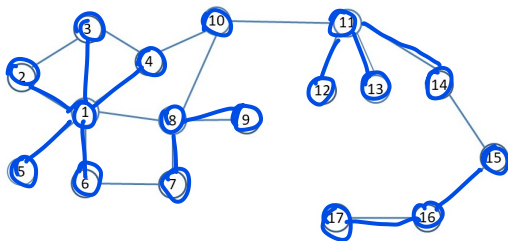


Uno **spanning tree** di un grafo G è uno spanning subgraph connesso e senza cicli (free tree).

N.B.: uno spanning tree esiste solo se G è connesso.

Esempio: vertici e archi blu

Concetti fondamentali: spanning forest



Una **spanning forest** di un grafo G è uno spanning subgraph senza cicli.

Esempio: vertici e archi blu

Primitive Importanti

- **Traversal:** Esplorazione sistematica del grafo (es.: crawling)
- **Connettività:** Verifica se il grafo è connesso (es.: reti wireless)
- **Identificazione delle componenti connesse:** (es.: reti wireless)
- **Ricerca di cammini minimi:** (es.: navigatore)
- **Ricerca di minimum spanning tree:** (es.: broadcast efficiente)
- **Stima della distanza media/massima:** (es.: social network)

Esercizio

La **teoria dei 6 gradi di separazione** fu formulata per la prima volta nel 1929 dallo scrittore ungherese Frigyes Karinthy nel racconto omonimo pubblicato nel volume Catene.

- Trovare l'enunciato di tale teoria.
- Definire un problema computazionale su grafo tale che la sua risoluzione possa servire a confermare o confutare la teoria, usando Facebook come insieme di dati di input.

Proprietà dei Grafi

Proposizione (Prop. 14.8, 14.10 e 14.11 in [GTG14])

Sia $G = (V, E)$ un grafo non diretto e semplice con $|V| = n$ vertici e $|E| = m$ archi. Valgono le seguenti proprietà:

- 1 $\sum_{v \in V} \text{degree}(v) = 2m$
- 2 $m \leq \binom{n}{2}$, e quindi $m \in O(n^2)$
- 3 Se G è un albero (rooted o free), allora $m = n - 1$
- 4 Se G è connesso, allora $m \geq n - 1$
- 5 Se G è senza cicli (cioè una foresta) $m \leq n - 1$

Dimostrazione

- ① Ogni arco (u, v) è contato ESATTAMENTE 2 volte in $\text{deg}_\text{in}(u)$ e $\text{deg}_\text{in}(v)$
- ② Essendo G semplice, c'è al più un arco fra ogni coppia (u, v) con $u \neq v$, $u, v \in V$
 $\Rightarrow m \leq \binom{n}{2} = n(n-1)/2 \in O(n^2)$
- ③ Usando l'equivalenza tra rooted e free tree è sufficiente provare la proprietà per i rooted tree
E in un rooted tree c'è ESATTAMENTE un arco

$(u, p(u))$ per ogni vertice $u \neq r \Rightarrow n-1$ archi.

④ Consideriamo il seguente while

while $(\exists \text{ un ciclo } C \text{ in } G)$ do {

rimuovi un arco da C

}

Ad ogni iterazione G resta rimane connesso

e alla fine del while non avrà cicli, e

quindi, dopo l'eliminazione di alcuni archi

seguono rimasti $n-1$ archi.

$\Rightarrow$ inizialmente doveva valere che $m \geq n-1$

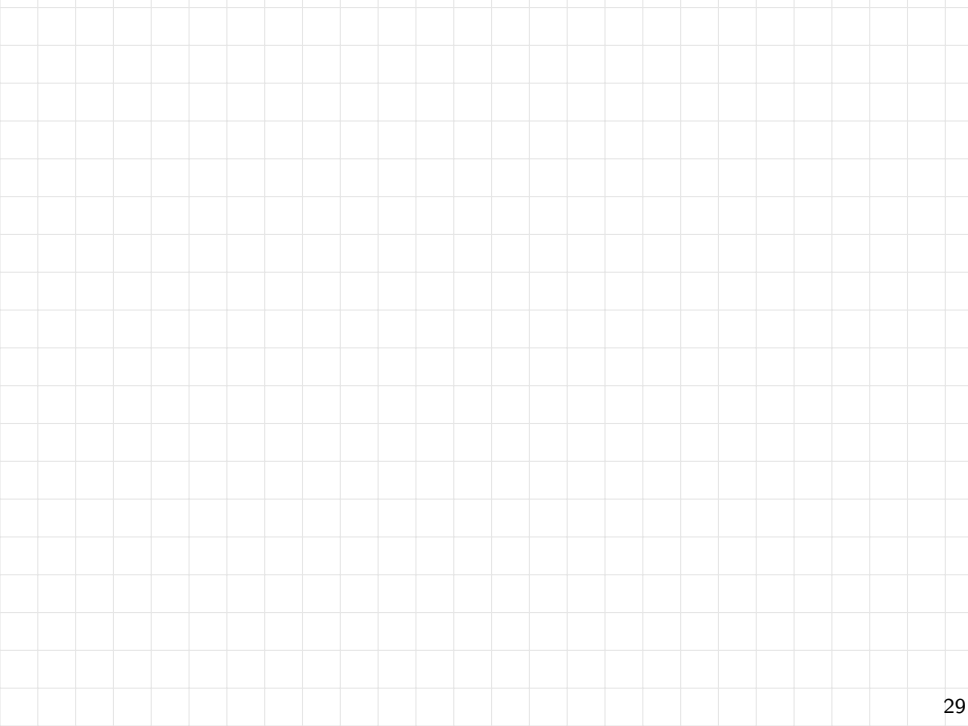
⑤ Sia $k \geq 1$ il numero di componenti connesse di G : $G_i = (V_i, E_i) \quad i=1..k$

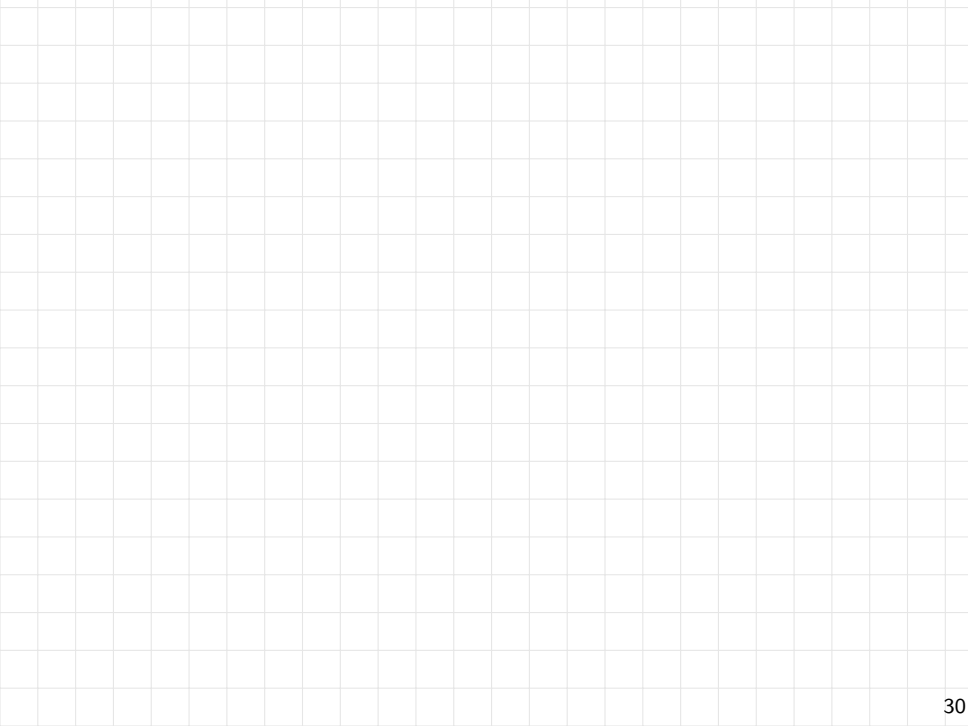
$\Rightarrow \forall i \quad G_i$ è un graf connesso e senza cicli

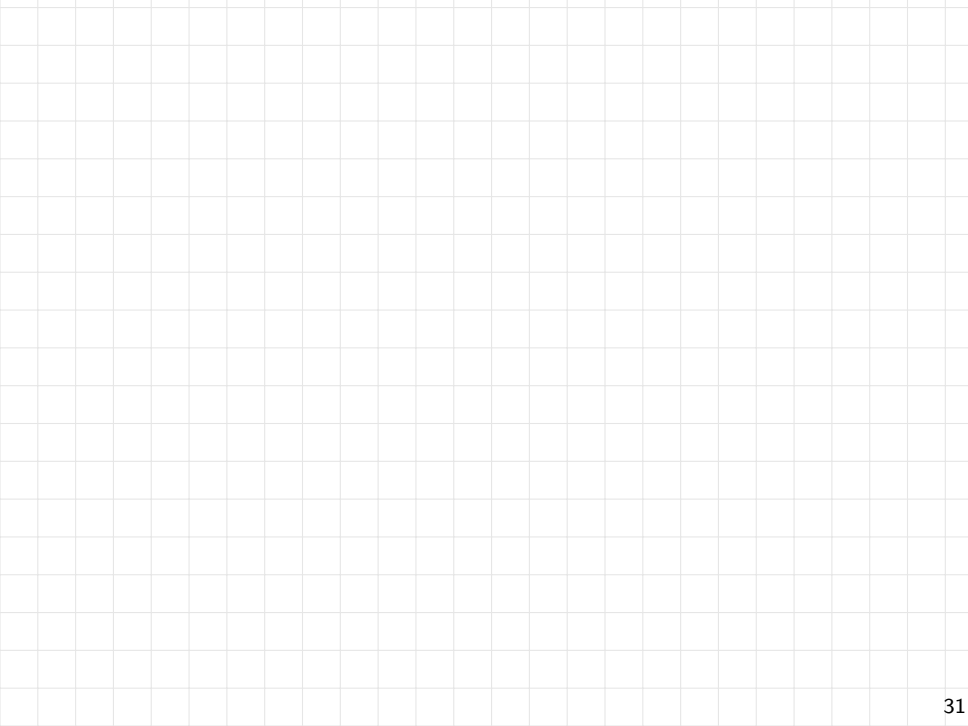
$\Rightarrow$ se $n_i = |V_i| \quad m_i = |E_i|$ allora

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

ed usando $k \geq 1$ si ha che $m = n - k \leq n - 1 \quad \square$







Rappresentazione di Grafi: strutture di base

Sia $G = (V, E)$ un grafo con n vertici ed m archi. (A volte, per comodità notazionale, rappresentiamo i vertici con gli interi da 1 a n .)

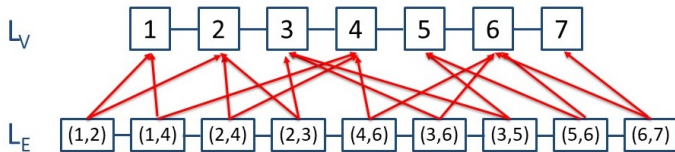
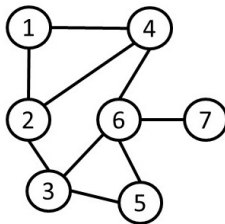
La rappresentazione più semplice per G fa uso delle seguenti

STRUTTURE di BASE

- **Lista di vertici L_V :** ogni nodo della lista contiene tutte le informazioni rilevanti per un vertice $v \in V$ distinto. Se si rappresentano i vertici con gli interi, L_V può essere un array.
- **Lista di archi L_E :** ogni nodo della lista contiene tutte le informazioni rilevanti per un arco $e = (u, v) \in E$ distinto, includendo tra esse i puntatori a u e v .

Osservazione: A seconda del problema da risolvere, ciascun vertice/arco può essere arricchito da campi (ID, peso, etichette varie, ecc.) strumentali all'algoritmo scelto.

Esempio



Rappresentazione degli archi: liste di adiacenza

Rappresentare gli archi tramite L_E rende poco efficienti gli algoritmi che devono esplorare i vicini di un vertice o che devono avere accesso diretto agli archi. Vediamo quindi altre rappresentazioni da usare in alternativa o in aggiunta a L_E .

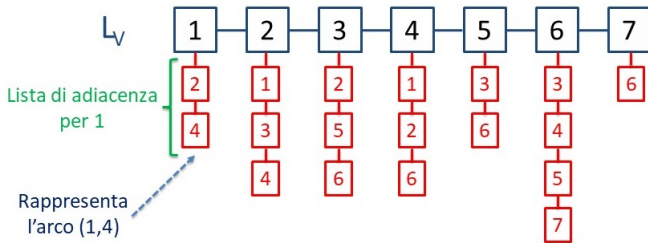
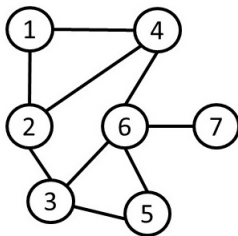
LISTE di ADIACENZA (Adjacency List): Per ogni vertice $v \in V$ si ha una lista $\mathcal{I}(v)$ di puntatori agli archi (elementi di E) incidenti su v .

Osservazione: la rappresentazione tramite liste di adiacenza, è quella più utilizzata per i seguenti motivi:

- Permette di rappresentare G in spazio lineare nella taglia del grafo, cioè $\Theta(n + m)$.
- Consente l'accesso sequenziale ai vicini di un vertice v , in tempo lineare nel grado di v .

N.B.: se non specificato diversamente noi useremo la rappresentazione con liste di adiacenza ma, se serve, assumeremo di avere a disposizione anche L_E .

Esempio di liste di adiacenza



Spazio occupato da L_v è $\Theta(n)$

Spazio occupato dalle liste di adiacenza è

$$\Theta\left(\sum_{v \in V} \text{degree}(v)\right) \in \Theta(m)$$

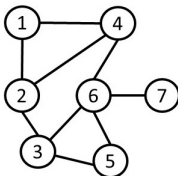
Oss. Questa analisi assume che ogni nodo di L_v e ogni nodo di una lista di adiacenza richieda spazio costante

Rappresentazione di archi: matrice di adiacenza

MATRICE di ADIACENZA (Adjacency Matrix) A : matrice $n \times n$ dove righe e colonne sono in corrispondenza 1-1 con i vertici, e

$$A[i_1, i_2] \doteq \begin{cases} \text{null} & \text{se } (i_1, i_2) \notin E \\ \text{puntatore a } e = (i_1, i_2) \in E & \text{se tale arco esiste} \end{cases}$$

Dato che il grafo è non diretto, rappresentano lo stesso arco



	1	2	3	4	5	6	7
1		(1,2)		(1,4)			
2	(2,1)		(2,3)	(2,4)			
3		(3,2)			(3,5)	(3,6)	
4	(4,1)	(4,2)				(4,6)	
5			(5,3)			(5,6)	
6			(6,3)	(6,4)	(6,5)		(6,7)
7						(7,6)	

Esempio di matrice di adiacenza

Osservazioni:

- La matrice di adiacenza consente l'**accesso in tempo costante a un arco e alle sue informazioni**, ma richiede una rappresentazione dei vertici tramite interi ($V = \{1, 2, \dots, n\}$).
- La matrice di adiacenza **occupa spazio $\Theta(n^2)$** che può risultare superlineare nella taglia del grafo. Per questo motivo è una rappresentazione utilizzata soprattutto nel caso di grafi densi (con numero di archi quadratico nei vertici) o per grafi con pochi vertici.

Una **via di mezzo tra liste di adiacenza e matrice di adiacenza** è costituita dalle **mappe di adiacenza**: per ogni vertice v gli archi incidenti su v sono memorizzati in una mappa. Usando una **tabella hash** per ogni mappa, si può quindi avere accesso a un qualsiasi arco in *tempo medio* $O(1)$, utilizzando globalmente uno spazio lineare nella taglia del grafo.

Implementazione Java

Le seguenti librerie software in Java supportano la gestione e l'utilizzo di grafi nelle applicazioni.

- JGraphT (<http://jgrapht.org>)
- JUNG - Java Universal Network/Graph Framework (<http://jung.sourceforge.net>)

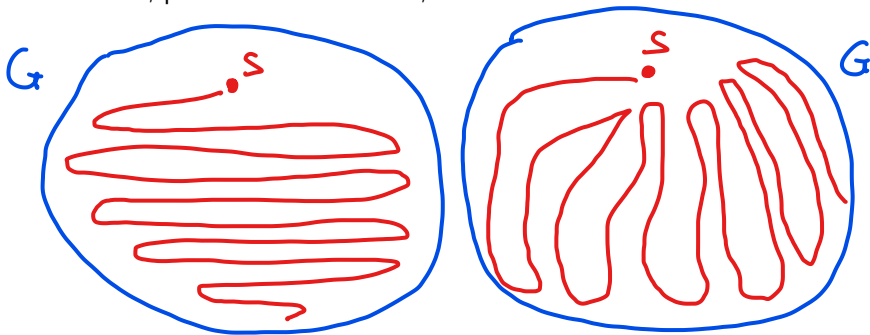
VISITE

(Graph Traversal)

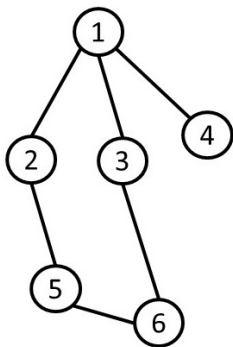
Graph Traversal di $G = (V, E)$

Procedura sistematica per **esplorare** G a partire da un vertice s visitando tutti i vertici.

- **Breadth-First Search** (BFS): dopo la visita di un vertice si visitano tutti i vicini prima di passare ai vicini dei vicini.
- **Depth-First Search** (DFS): dopo la visita di un vertice si visita un vicino, poi un vicino del vicino, ecc.



Esempio



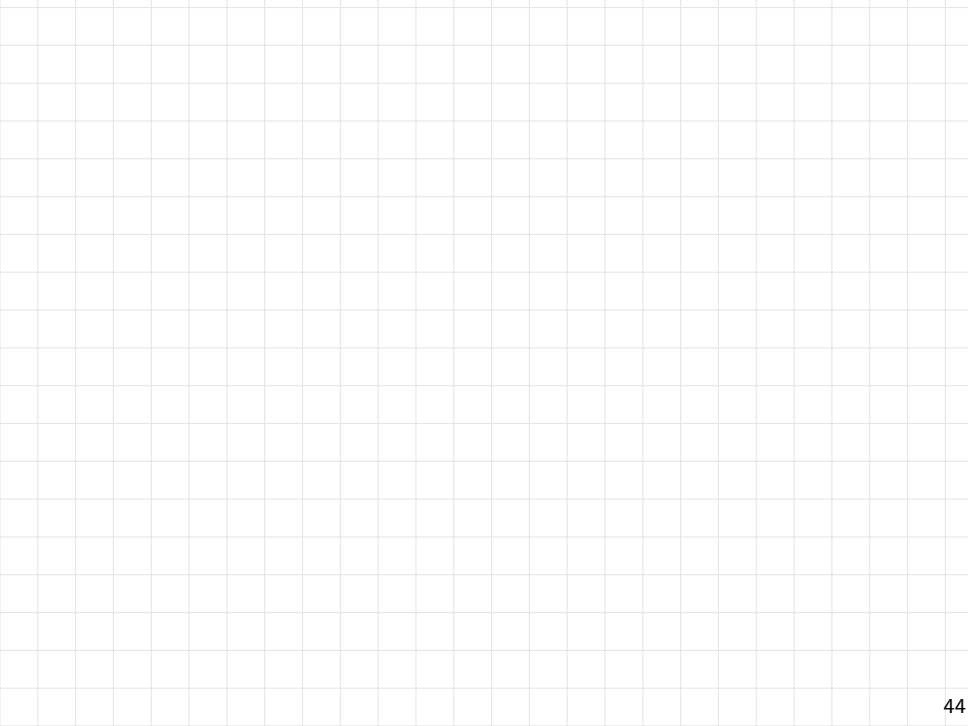
ORDINE di VISITA DEI VERTICI

BFS: 1 → 2 → 3 → 4 → 5 → 6

DFS: 1 → 2 → 5 → 6 → 3 → 4

Osservazioni

- * le liste sono dei DESIGN PATTERN ALGORITMICI che possono essere utilizzate (opportunosamente edettate) per risolvere diversi problemi.
- * la sistematicità dello stile è fondamentale e strumentale per la soluzione EFFICIENTE dei problemi, che, in molti casi, non si potrebbe ottenere con una generica selezione di vertici e archi



Breadth-First Search: notazioni e definizioni

Sia $G = (V; E)$ un grafo semplice, non diretto e non pesato:

- Dato $s \in V$, $C_s \subseteq G$ denota la **componente connessa di G contenente s** , costituita da tutti i vertici raggiungibili con cammini da s , e dagli archi su essi incidenti.
- Ricordiamo che dati due vertici $x, y \in V$ nella stessa componente connessa, la loro **distanza $d(x, y)$** è definita come la **minima lunghezza di un cammino da x a y** . Se i vertici sono in componenti connesse distinte, si assume $d(x, y) = +\infty$.

Breadth-First Search: ipotesi implementative

Assunzioni sull'implementazione di $G = (V; E)$:

- Per ogni vertice $v \in V$ esiste un campo $v.ID$ che vale 1, se v è stato *visitato*, e vale 0 altrimenti.
- Per ogni arco $e \in E$ esiste un campo $e.label$ che memorizza una opportuna *etichetta* oppure vale *null* se e non ha ancora etichetta
- Per ogni $v \in V$, $incidentEdges(v)$ restituisce un iteratore agli archi incidenti su v (e quindi alla sua lista di adiacenza), che possono essere enumerati (ad es., con un ciclo **forall**) in tempo $\Theta(1)$ ciascuno
- Per ogni arco $e = (v, w) \in E$, $opposite(v, e)$ restituisce w in tempo $\Theta(1)$.

Breadth-First Search: Algoritmo BFS

Algoritmo iterativo che, a partire da un vertice s ,

- visita tutti i vertici di C_s , (impostando a 1 i loro ID)
- etichetta ciascun arco di C_s come DISCOVERY o CROSS EDGE,
- partiziona i vertici di C_s in livelli L_i in base alla loro distanza i da s .

Algoritmo BFS(G, s)

Input: grafo $G = (V, E)$, $s \in V$, ogni vertice v di C_s con $v.ID = 0$, e ogni arco e di C_s con $e.label = null$.

Output: visita ogni vertice di C_s ed etichetta ogni arco di C_s come DISCOVERY EDGE o CROSS EDGE.

impostando a 1 i loro ID

Breadth-First Search: Algoritmo BFS

visita il vertice s e imposta $s.ID \leftarrow 1$;

$L_0 \leftarrow$ lista contenente s ; $i \leftarrow 0$;

while ($!L_i.isEmpty()$) **do**

crea una lista di vertici L_{i+1} vuota;

forall $v \in L_i$ **do**

forall $e \in G.incidentEdges(v)$ **do**

if ($e.label = null$) **then**

$w \leftarrow G.opposite(v, e)$;

if ($w.ID = 0$) **then**

$e.label \leftarrow$ DISCOVERY EDGE;

visita il vertice w e imposta $w.ID \leftarrow 1$;

inserisci w in L_{i+1} ;

else $e.label \leftarrow$ CROSS EDGE;

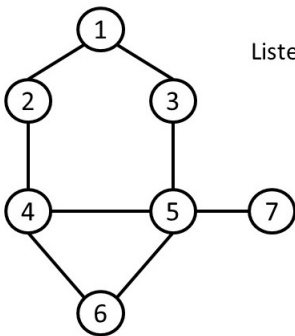
$i \leftarrow i + 1$;

return;

l'iterazione ESATINA v

Orizzonte che
 v SCOPRE w

Esempio



Liste di adiacenza (ordine crescente)

```
1: 2 3
2: 1 4
3: 1 5
4: 2 5 6
5: 3 4 6 7
6: 4 5
7: 5
```

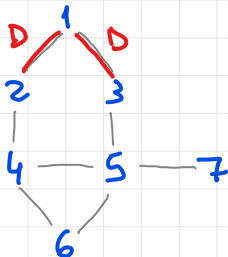
$s =$ 

Inizializzazione:

Iterazione 0

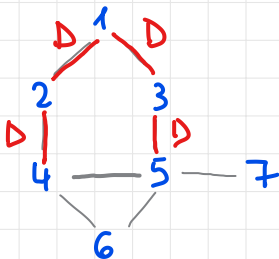
$$L_0 = 1$$

$$L_1 = 2 \ 3$$

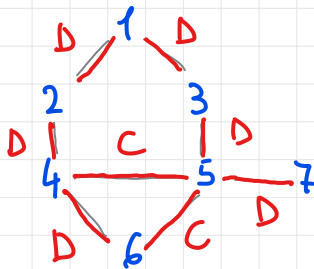


Iterazione 1

$$L_2 = 4 \ 5$$



Iterazione 2



$$L_3 = 6 \ 7$$

Iterazione 3

$$L_4 = \emptyset$$

Non viene eseguita l'iterazione 4 perché
 $L_4 = \emptyset$ (condizione di uscita del while)

Correttezza e proprietà di BFS

Proposizione (simile a Prop. 14.16 [GTG14])

Alla fine dell'esecuzione di $\text{BFS}(G, s)$ si ha che:

- 1 tutti i vertici di C_s sono visitati e tutti gli archi di C_s sono etichettati come DISCOVERY o CROSS EDGE;
- 2 i DISCOVERY EDGE formano uno spanning tree T di C_s radicato in s e chiamato *BFS tree*;
- 3 $\forall v \in L_i$ il cammino in T da s a v ha i archi e $i = d(s, v)$ (distanza tra s e v);
- 4 se $(u, v) \in E$ e $(u, v) \notin T$ ($\Rightarrow (u, v)$ è un CROSS EDGE) gli indici dei livelli di u e v differiscono al più di 1



Osservazione: in virtù della Proprietà 2, in [GTG14] i DISCOVERY EDGE sono anche chiamati **TREE EDGE**

Dimostrazione

① le prove si articolano come segue

* Il vertice s è visitato all'inizio

* $\forall v \in C_s, \exists$ un cammino da s a v

$$s = u_0 - u_1 - u_2 - \dots - u_k = v$$

Una semplice induzione dimostra che prima

o poi $\text{BFS}(G, s)$ visita $u_i, \forall 0 \leq i \leq k$, e quindi

prima o poi $\text{BFS}(G, s)$ visita tutti i vertici di C_s

* Sia (u, v) un generico arco di C_s , e quindi

u e v sono vertici di C_s . Per quanto detto sopra

prima o poi u e v saranno visitati da $BFS(G, s)$
e l'arco (u, v) sarà etichettato come DISCOVERY
o CROSS EDGE quando sarà considerato come arco
incidente su primo dei 2 vertici, u e v , esaminato

② Per come è progettata $BFS(G, s)$, per ogni vertice
 $w \in G$, con $w \neq s$, esiste ESATTAMENTE un
DISCOVERY EDGE (v, w) etichettato mentre si
esamina v . Se definiamo **v PADRE di w**
avremo che

- * I DISCOVERY EDGE definiscono relazioni padre-figlio
- * $\forall w \in C_s, w \neq s$, esiste un unico padre
- * $\forall w \in C_s$ risalendo di padre in padre, lungo DISCOVERY EDGE si arriva a s .

$\Rightarrow$ I DISCOVERY EDGE definiscono un rooted tree con radice s , che è uno Spanning tree di C_s

③ Sia $v \in L_i$. La prova si articola così

* Risalendo da v di padre in padre lungo

DISCOVERY EDGE si ottiene un cammino da s
a v di lunghezza i

* Supponiamo PER ASSURDO che esista un
cammino da s a v di lunghezza $t < i$
(quindi più corto). Sia esso

$$s = u_0 - u_1 - \dots - u_t = v$$

Per come è progettato $\text{BFS}(G, s)$ deve valere

che $u_0 \in L_0$

$u_1 \in L_1$

$u_2 \in L_j$ per un qualche $j \leq 2$

$\vdots$

$u_h \in L_j$ per un qualche $j \leq h$

$\vdots$

$v = u_t \in L_j$ per un qualche $j \leq t < i$

che è impossibile dato che abbiamo

ipotizzato $v \in L_i$

④ Sia (u, v) un CROSS EDGE con $u, v \in C_j$
e $u \in L_i$ e $v \in L_j$. Assumiamo che
 ~~u venga eliminato prima di v~~
(altrimenti li scambiamo). Eliminiamo
 ~~u~~ se guarderemo l'arco (u, v)

* Se $v.ID = 1 \Rightarrow v \in L_i$ o $v \in L_{i+1}$

* Se $v.ID = 0 \Rightarrow v$ è inserito in L_{i+1}

$$\Rightarrow j = i + 1$$

$$\Rightarrow |j - i| \leq 1$$

□

OSSERVAZIONE IMPORTANTE: Dal punto ④ della precedente proposizione e dal funzionamento dell'algoritmo si evince immediatamente che alla fine dell'esecuzione di $\text{BFS}(G, s)$, dato un qualsiasi arco (u, v) di G_s , $\text{DISCOVERY}_u \neq \text{DISCOVERY}_v$, i due vertici u, v sono in uno stesso livello L_i , o in due livelli adiacenti L_i e L_{i+1} .

Complessità di BFS(G, s)

Teorema

La complessità di BFS(G, s) è $\Theta(m_s)$, dove m_s è il numero di archi in C_s .

N.B.: se $G = (V, E)$ è connesso, la complessità di BFS(G, s) è $\Theta(|E|)$.

Assumiamo che il costo di una visita di un vertice sia $\Theta(1)$

* $\forall v \in C_s$ viene eseguita ESATTAMENTE una iterazione del for all esterno
il cui costo è $\Theta(1 + \text{degree}(v))$

$\Rightarrow$ Complessità è $\Theta\left(\sum_{v \text{ vertice di } C_s} (1 + \text{degree}(v))\right)$

$$e \sum_{v \text{ vertice di } C_S} (1 + \deg(v)) = \Theta(m_S + n_S) \\ = \Theta(m_S)$$

dove $n_S = \# \text{ vertic. di } C_S$ e

$m_S \geq n_S - 1$ dato che C_S è un
 sottografo connesso.

□

OSSERVAZIONE IMPORTANTE: nel caso in cui la
 "visita" di un vertice abbia un costo non costante,
 ad $\Theta(k_v)$ per il vertice v , la complessità

di BFS(G, s) diventa $\Theta(m_s + \sum_{v \in G_s} t_v)$

Visita di tutto il grafo

Si noti che l'algoritmo $\text{BFS}(G, s)$ visita solo la componente connessa di s .

Il seguente *design pattern* permette la **visita di tutto il grafo**, nel caso esso non sia connesso. Esso suppone di poter enumerare vertici e archi (ad es. tramite iteratori delle liste L_V e L_E) generando il prossimo vertice/arco in tempo costante.

```
forall  $v \in V$  do  $v.\text{ID} \leftarrow 0$ ;  
forall  $e \in E$  do  $e.\text{label} \leftarrow \text{null}$ ;  
forall  $v \in V$  do  
  | if ( $v.\text{ID} = 0$ ) then  $\text{BFS}(G, v)$  ;
```

Osservazione: In [GTG14] un simile design pattern è presentato usando la DFS (Algoritmo `DFSComplete`)

Analisi

Sia $G = (V, E)$ con $|V| = n$ e $|E| = m$

Supponiamo che G abbia k componenti connesse

$G_1, G_2, \dots, G_k$ con $k \geq 1$

Sia $m_i = \# \text{ archi in } G_i$ $1 \leq i \leq k$

$$\Rightarrow \sum_{i=1}^k m_i = m$$

La complessità della visita di tutto il graf
è ottenuta sommando i seguenti contributi

* Primi 2 cicli for all: $\Theta(n+m)$

* Terzo ciclo for all

- Selezione di L_v : $\Theta(n)$

- ESATTAMENTE una BFS per ogni G_i :

$$\Theta\left(\sum_{i=1}^k m_i\right) = \Theta(m)$$

$\Rightarrow$ Complessità totale: $\Theta(n+m)$

Oss. Se G non è connesso potremmo avere $n > m$
quindi è necessario tenere n e m nell'ordine
di grandezza

Applicazioni della BFS: connettività

Input: Grafo $G = (V, E)$.

Output: Numero delle componenti connesse di G e vertici di ciascuna componente etichettati con ID distinti.

Oss.: Se il numero restituito è 1, allora G è connesso.

IDEA: visita tutto G e quando si visita le insieme CC, si impostano gli ID dei suoi vertici a i (contatore globale)

BFS(G, s, i): variante di BFS(G, s) in cui invece di impostare gli ID dei vertici visitati a 1, si impostano a i

Algo: true

for all $v \in V$ do $v.id \leftarrow 0$;

for all $e \in E$ do $e.label \leftarrow null$;

$i \leftarrow 0$;

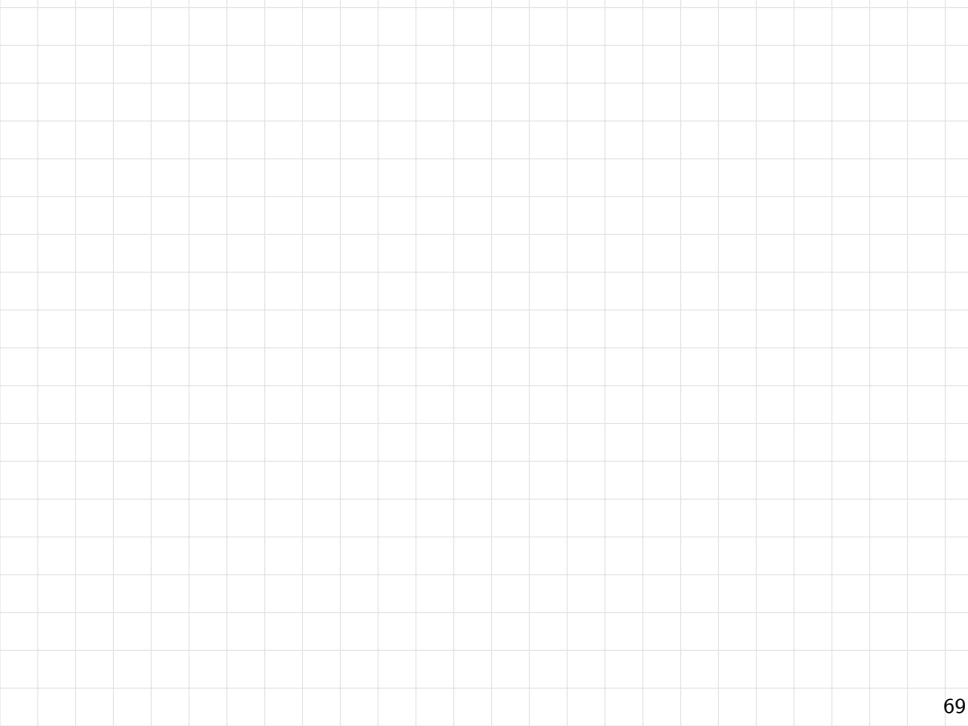
for all $v \in V$ do {

if ($v.id = 0$) then { $i \leftarrow i+1$; $BFS(G, v, i)$ }

return i

Complessità: $\Theta(n+m)$

- * Stessa struttura della visita completa di G
- * le modifiche alla BFS non ne alterano la complessità



Applicazioni della BFS: spanning tree

Input: Grafo $G = (V, E)$ connesso.

Output: Spanning tree di G (rappresentato come lista di archi).

IDEA: Sfruttare il fatto che i DISCOVERY EDGE
formano una spanning tree delle componenti
connesse del vertice da cui si parte
 $BFS(G, s, L)$ con $L = \emptyset$ liste vuote
modifica di $BFS(G, s)$ che
* inserisce una copia di ogni DISCOVERY EDGE
in L

* Alla fine restituisce L

Algoritmo

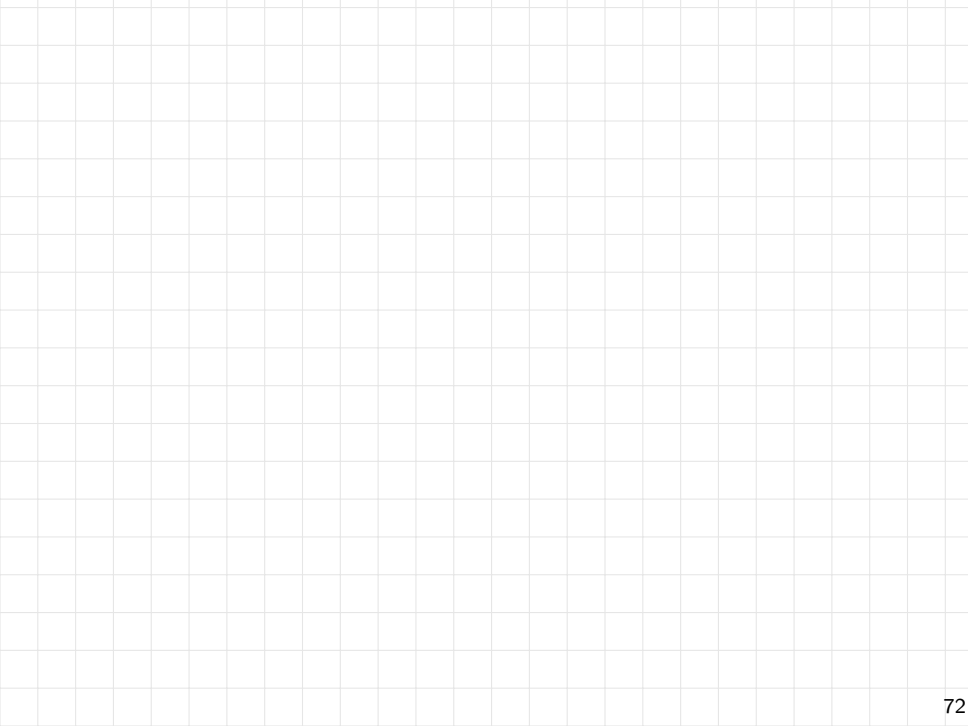
$L \leftarrow$ lista vuota; $s \leftarrow$ vertice arbitrario di V

return BFS(G, s, L)

Complessità: $\Theta(m)$

* G è connesso e quindi la BFS costa $\Theta(m)$

* la modifica alla BFS non ne altera la complessità



Applicazioni della BFS: cammini minimi

Input: Grafo $G = (V, E)$ e due vertici $s, t \in V$.

Output: Cammino di lunghezza minima da s in t in G (rappresentato come lista di archi), se esiste, o null se non esiste alcun cammino.

Modifichiamo le BFS come segue

* $\forall v \in V$ uso 2 campi aggiuntivi:

$v.parent$, $v.edge$

* la visita di un vertice w scoperto da v

consiste nell'impostare $w.parent \leftarrow v$

e $w.edge \leftarrow (v, w)$. Se $w = s$, entrambi

i campi sono null

for all $v \in V$ do $\{v.ID \leftarrow 0; v.parent \leftarrow null; v.edge \leftarrow null\}$

for all $e \in E$ do $e.label \leftarrow null$

BFS(G, s) // can be modified like this

if ($t.ID = 0$) then return null;

$L \leftarrow$ list vnodes; $w \leftarrow t$;

while ($w \neq s$) do {

 appending $w.edge$ to L ;

$w \leftarrow w.parent$;

}

return L

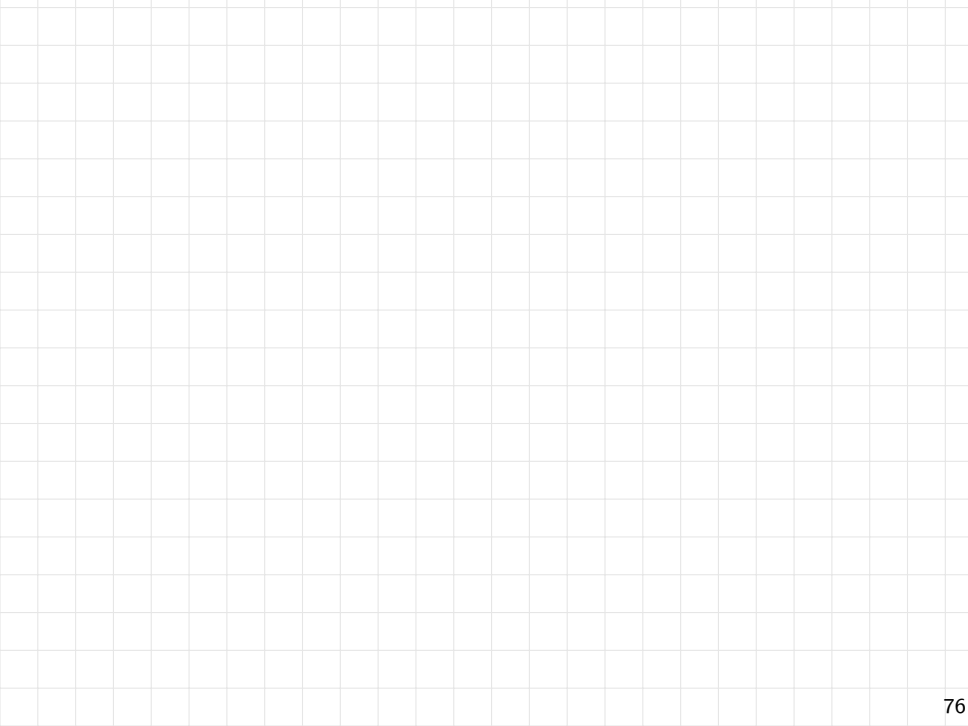
Complexity: $\Theta(n+m)$

* Prim's 2nd full: $\Theta(n+m)$

* BFS: $\mathcal{O}(m)$

* Cycle while: $\mathcal{O}(n)$

$\Rightarrow$ Total $\Theta(n+m)$



Applicazioni della BFS: ciclicità

Input: Grafo $G = (V, E)$.

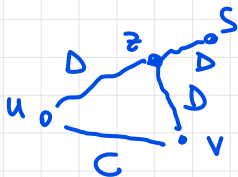
Output: Un ciclo in G (rappresentato come lista di archi), se esiste, o null se non esiste.

IDEA : visitare tutto G invocando BFS su
ciascuna componente connessa

* Se nessun arco è stato etichettato CROSS EDGE
return null, perché i DISCOVERY EDGE non
formano cicli

* Se c'è un CROSS EDGE c'è sicuramente

un ciclo. Sia (u, v) il CROSS EDGE
 etichettato tale durante $\text{BFS}(G, s)$

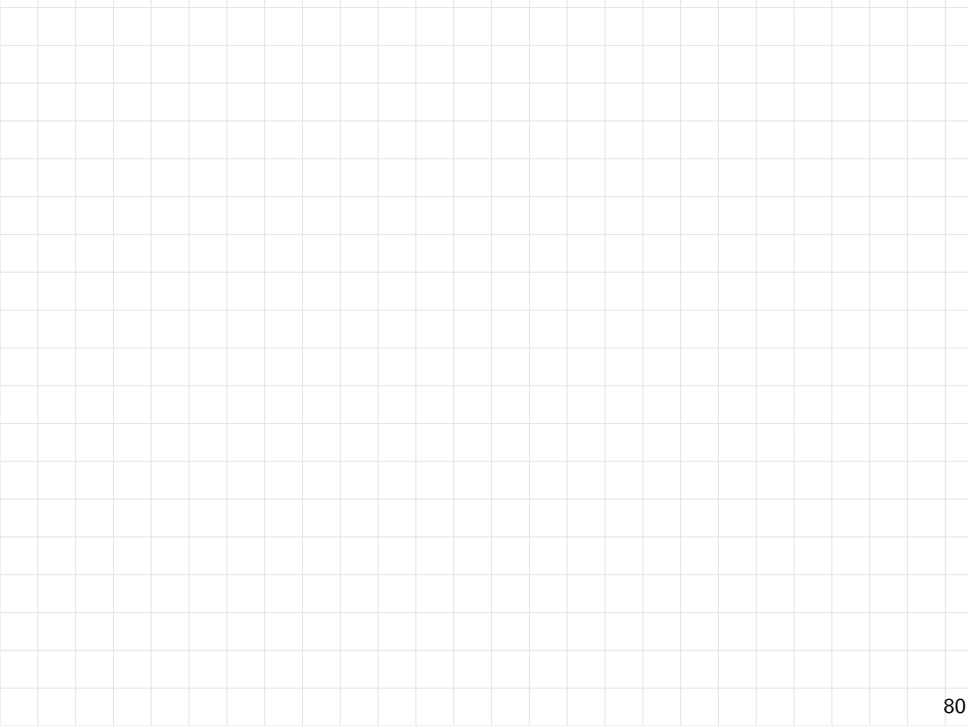


$u \in L_i$, $v \in L_i \text{ o } L_{i+1}$

- Se $v \in L_{i+1}$ inserisco nel ciclo (u, v) e $(v, v.\text{parent})$ e poi ripeto in parallelo fino a z aggiungendo al ciclo i DISCOVERY EDGE in contatto.

- Se $v \in L_i$ fare la stessa cosa ottenendo l'iniziale aggiunta di $(v, v, parent)$ al Cibo

Dettaglio: esercizio (anche l'altro)



Nelle slide precedenti abbiamo dimostrato la seguente proposizione:

Proposizione

Dato $G = (V, E)$, con $|V| = n$ e $|E| = m$, i seguenti problemi possono essere risolti in tempo $O(m + n)$ usando la BFS:

- 1 testare se G è connesso;
- 2 trovare le componenti connesse di G ;
- 3 trovare uno spanning tree di G , se G è connesso;
- 4 trovare un cammino minimo tra 2 vertici s e t , se esiste;
- 5 trovare un ciclo, se esiste.

Esercizio

Sia $G = (V, E)$ un grafo con n vertici ed m archi. Progettare un algoritmo che conti le coppie di vertici $u, v \in V$, tali che $u \neq v$ ed esiste un cammino da u a v , analizzandone la complessità. Per avere punteggio pieno la complessità deve essere $O(n + m)$.

IDEA: Sia $G = G_1 \cup G_2 \cup \dots \cup G_k$ $k \geq 1$

G_i = i -esima componente connessa

n_i = # vertici di G_i

Il valore cercato è $\sum_{i=1}^k \binom{n_i}{2}$

in quanto u e v sono collegati da un cammino sse sono nella stessa G_i

e G_i contribuire $\binom{n_i}{2}$ coppie

Modifica della BFS: aggiungere un contatore C inizializzato a 0, incrementato di 1 ogni volta che si visita un vertice e restituito in output (la modifica non altera la complessità di BFS)

Algoritmo Reachable Pairs(G)

input $G = (V, E)$

output # coppie $u, v \in V, u \neq v$ collegate da un cammino

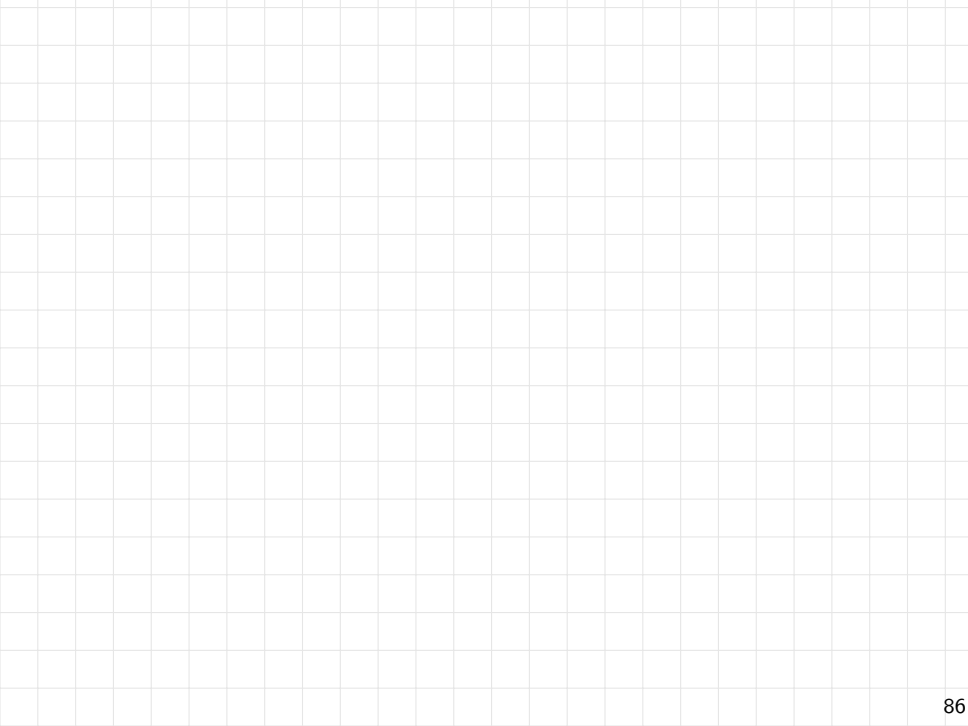
```

count  $\leftarrow$  0;
for all  $v \in V$  do  $v.id \leftarrow 0$ ;
for all  $e \in E$  do  $e.label \leftarrow null$ 
for all  $v \in V$  do {
    if ( $v.id = 0$ ) then {
         $x \leftarrow BFS(G, v)$  // # vertic. dalle
                           cc di v
        if ( $x > 1$ ) then count  $\leftarrow$  count +  $x(x-1)/2$ 
    }
}
return count

```


Complete: $\Theta(n+m)$

Stene avel r. delle n-ita complete dG



Esercizio

Sia G un grafo non diretto e connesso in cui ciascun vertice ha grado esattamente c , con $c > 2$ costante intera. Si consideri l'esecuzione di $\text{BFS}(G, s)$ a partire da un arbitrario vertice $s \in V$. Dimostrare per induzione su i che il livello L_i generato da $\text{BFS}(G, s)$ contiene $\leq c \cdot (c-1)^{i-1}$ vertici, per ogni $i \geq 0$.

BASE : $i = 0, 1$

$$* L_0 = \{s\} \Rightarrow |L_0| = 1 \stackrel{?}{\leq} c \cdot (c-1)^{i-1} = \frac{c}{c-1} > 1 \quad \text{OK}$$

$$+ L_1 = \{\text{vicini di } s\} \Rightarrow |L_1| = c \stackrel{?}{\leq} c \cdot (c-1)^{i-1} \quad \text{OK}$$

$\searrow = c$

PASSO INDUTTIVO : Fissare $i \geq 1$ arbitrario.

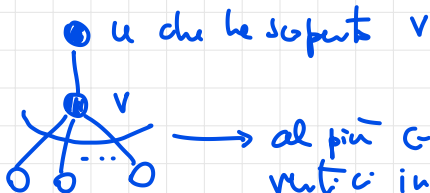
$$\text{Hp. Induttiva : } |L_j| \leq c(c-1)^{j-1} \quad \forall j: 0 \leq j \leq i$$

Considerare L_{i+1} $i+1 > 1$

L_{i-1}

L_i

L_{i+1}



$$\Rightarrow |L_{i+1}| \leq (c-1) \cdot |L_i|$$

$$\leq (c-1) \cdot c \cdot (c-1)^{i-1} \quad \text{per hp. induttive}$$

$$= c (c-1)^{i+1-1}$$

□

Osservazione indipendente dell'esercizio

Il numero totale di vertici in G è

$$n = |V| = \sum_{i=0}^h |L_i| \leq \sum_{i=0}^h c \cdot (c-1)^{i-1}$$

dove h è l'indice dell'ultimo livello

Un vinco su h (che è anche la distanza max
di un vertice da s) è che

$$\sum_{i=0}^h c(c-1)^{i-1} \geq n \quad (= \# \text{ vertice di } G)$$

Esercizio: far vedere che questo implica che
 $h \in \Omega(\log n)$

Esercizio

Sia $G = (V, E)$ un grafo con n vertici e m archi. Sviluppare un algoritmo che restituisce, se esiste, un vertice $v \in V$ da cui sono raggiungibili (con cammini) $\geq n/2$ altri vertici, e analizzarne la complessità. Se un tale vertice non esiste l'algoritmo restituisce null.

Esercizio C-14.49 [GTG14]

Un grafo $G = (V, E)$ si dice *bipartito* se l'insieme di vertici V può essere partizionato in due sottoinsiemi X e Y tali che ogni arco di E incide su un vertice di X e uno di Y . Progettare e analizzare un algoritmo efficiente che determini se un grafo non diretto G è bipartito.

Esercizio

Progettare e analizzare un algoritmo efficiente per risolvere il problema definito nell'esercizio della Slide 24

Depth-First Search: Algoritmo DFS

DFS è un algoritmo ricorsivo che, a partire da un vertice s ,

- visita tutti i vertici di C_s ,
- etichetta ciascun arco di C_s come DISCOVERY o BACK EDGE

Si usano le **stesse ipotesi implementative della BFS**.

Algoritmo DFS(G, v) (*prima invocazione: $v = s$*)

Input: grafo $G = (V, E)$, $v \in V$

Output: visita ogni vertice raggiungibile da v e non ancora visitato, ed etichetta ogni arco esaminato come **DISCOVERY** o **BACK EDGE**.

Osservazione: Nella **prima invocazione** ($v = s$), tutti i vertici v di C_s hanno $v.ID = 0$, e tutti gli archi e di C_s hanno $e.label = null$. In un'invocazione generica, alcuni vertici/archi sono già stati visitati/etichettati.

Depth-First Search: Algoritmo DFS

visita il vertice v e imposta $v.ID \leftarrow 1$;

forall $e \in G.incidentEdges(v)$ do

 if ($e.label = null$) then

$w \leftarrow G.opposite(v, e)$;

 if ($w.ID = 0$) then

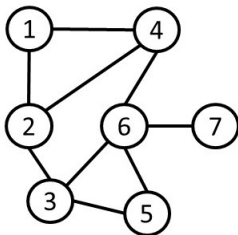
$e.label \leftarrow$ DISCOVERY EDGE;

 DFS(G, w);

 else $e.label \leftarrow$ BACK EDGE;

} v scopre w

Esempio



$s =$ 

Liste di adiacenza (ordine crescente)

1: 2 4

2: 1 3 4

3: 2 5 6

4: 1 2 6

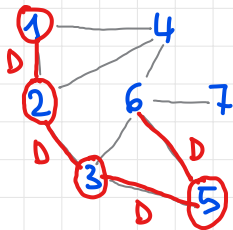
5: 3 6

6: 3 4 5 7

7: 6

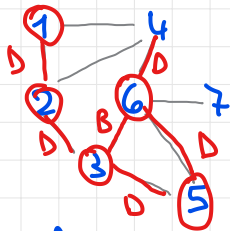
Etichette impostate e vertici
visitati da $\text{DFS}(G, i)$

con $i = 1, 2, 3, 5$

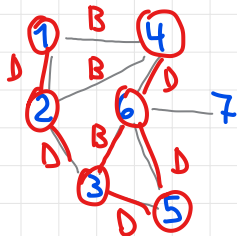


Etichette impostate e vertici
visitati da $\text{DFS}(G, 6)$

N.B. $\text{DFS}(G, 6)$ imposterà
anche l'etichetta di (6,7) ma solo
dopo l'esecuzione di $\text{DFS}(G, 4)$



Etichette impostate e vertici
visitati da DFS(G, u)

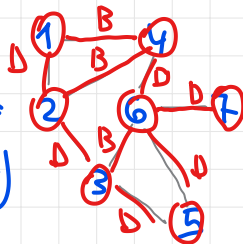


Case succede ora?

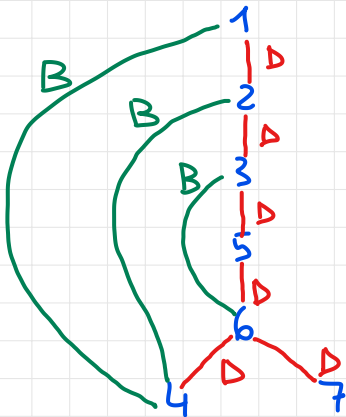
Dfs(G, u) termine l'execution
e il controllo ritorna a Dfs(G, v)

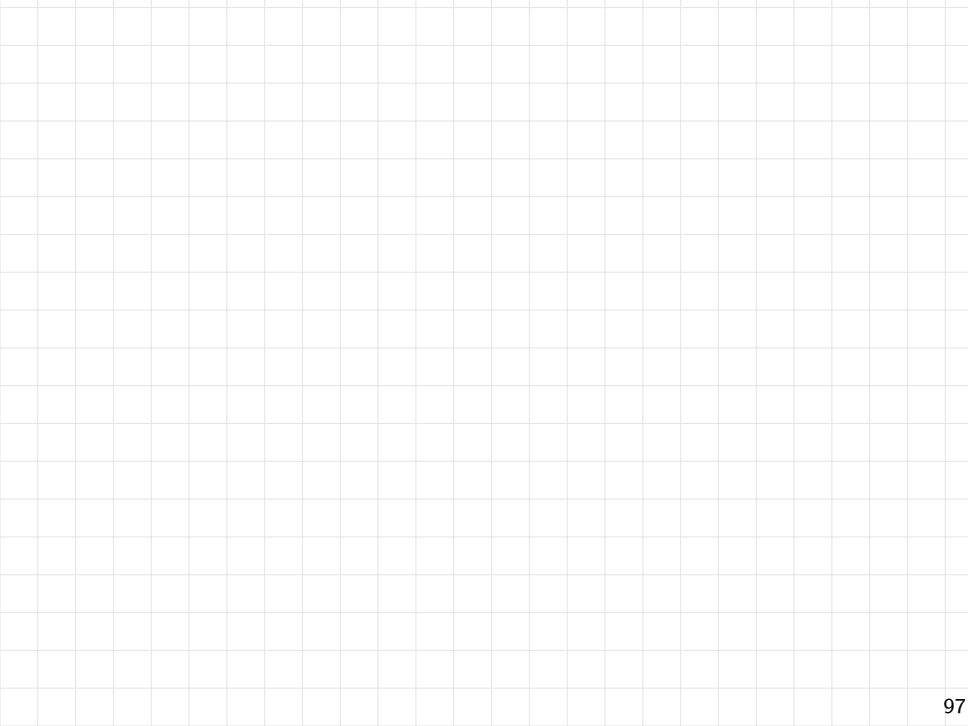
che imposta $(6,7)$ a D e chiave DFS $(G,7)$

Dopo $\text{DFS}(G, 7)$ terminano tutte le $\text{DFS}(G, i)$
aperte: $i = 6, 5, 3, 2, 1$



Per comprendere meglio il significato delle etichette BACK EDGE, disegniamo il graf in modo diverso





Analisi di DFS(G, s)

Proposizione (simile a Proposizione 14.12 di [GTG14])

Si supponga di eseguire DFS(G, s) e che, all'inizio, nessuno dei vertici/archi di C_s sia visitato/etichettato. A fine esecuzione si ha che:

- ① tutti i vertici di C_s sono visitati e tutti gli archi di C_s sono etichettati come DISCOVERY o BACK EDGE;
- ② i DISCOVERY EDGE formano uno spanning tree T di C_s radicato in s .

La dimostrazione è simile a quella fatta per le BFS e le seltenes

Complessità di DFS(G, s)

Teorema

Si supponga di eseguire DFS(G, s) e che, all'inizio, nessuno dei vertici/archi di C_s sia visitato/etichettato. La complessità di DFS(G, s) è $\Theta(m_s)$, dove m_s è il numero di archi in C_s .

N.B.: se $G = (V, E)$ è connesso, la complessità di DFS(G, s) è $\Theta(|E|)$.

Dim. L'albero delle ricorsione associato a DFS(G, s) è tale che

- * contiene esattamente 1 nodo per ogni vertice di C_s
- * il costo contribuito da un vertice $v \in C_s$

è $\Theta(\text{degree}(v))$ (assumendo che la visita di v richiede $\Theta(1)$ operazioni).

$$\Rightarrow \text{Complessità} \in \Theta\left(\sum_{v \in G_S} \text{degree}(v)\right) = \Theta(n_S)$$

Qu. Il primo punto deriva dal fatto che tutti i vertici di G_S saranno visitati (della proposizione precedente) e ogni vertice $v \in G_S$ può essere visitato solo 1 volta (per costruzione dell'algoritmo) e nelle chiamate $\text{DFS}(G, v)$

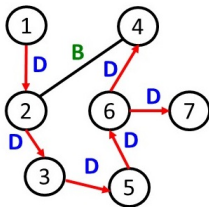
$\Rightarrow$ C. deve essere ESATTAMENTE UNA CHIAMATA
per ogni vertice $v \in G$

OSSERVAZIONE : Come osservato per la BFS, se
la visita di un vertice v costa $\Theta(t_v)$, non
costante, alla complessità si aggiunge un termine
additivo $\Theta\left(\sum_{v \in G} t_v\right)$

Esercizio C-14.37 [GTG14]

Si consideri l'esecuzione di $\text{DFS}(G, s)$, e sia T lo spanning tree di C_s , radicato in s , costituito dagli archi etichettati come DISCOVERY EDGE da tale esecuzione. Supponiamo che durante una delle varie chiamate ricorsive $\text{DFS}(G, v)$ si etichetti un arco (v, w) come BACK EDGE. Dimostrare che w è antenato di v in T .

Esempio:



- Gli archi rossi costituiscono lo spanning tree dei discovery edge
- Il verso delle frecce rappresenta le relazioni padre-figlio
- $(4,2)$ è marcato come BACK EDGE e 2 è antenato di 4

Proposizione (simile a Proposizione 14.14 di [GTG14])

Dato $G = (V, E)$, con $|V| = n$ e $|E| = m$, i seguenti problemi possono essere risolti in tempo $O(m + n)$ usando la DFS:

- 1 visitare tutto il grafo;
- 2 testare se G è connesso;
- 3 trovare le componenti connesse di G ;
- 4 trovare uno spanning tree di G , se G è connesso;
- 5 trovare un cammino tra 2 vertici s e t , se esiste (s - t reachability);
- 6 trovare un ciclo, se esiste.

La dimostrazione è lasciata come esercizio.

Esercizio

Sia $G = (V, E)$ un grafo non diretto con $k > 1$ componenti connesse. Progettare un algoritmo basato sulla DFS che aggiunga $k - 1$ archi a G per renderlo connesso, e analizzarne la complessità. Si assuma di poter aggiungere a E un arco $(u, v) \notin E$ in tempo costante invocando il metodo $G.addArc(u, v)$.

Esercizio



Si consideri un labirinto L che ha un unico punto di ingresso s e al cui interno è nascosto il terribile Minotauro. L'ing. Teseo deve entrare in L , trovare il Minotauro, ucciderlo, e uscire da L . Trovare un'opportuna rappresentazione del labirinto come grafo e far vedere come, sfruttando l'algoritmo DFS, l'ing. Teseo può compiere con successo la sua missione. Si assuma che il labirinto sia connesso, nel senso che ogni punto di esso sia raggiungibile da s .

BFS vs DFS

- * Su graf. non dirett., BFS e DFS hanno la stessa complessità e possono essere usati equivalentemente per: connettività, spanning tree, rilevamento di cicli, e reachability
 - * Ciascuna visita ha pro' degli aspetti distintivi che la rendono preferibile all'altra
- BFS: calcolo dei cammini minimi
- DFS:

- Più space-efficient perché richiede solo spazio proporzionale all'altezza dell'albero della ricorrenza (in aggiunta a quello per l'input)
- Può risultare efficiente per trovare un cammino da s a t , se t è molto lontano da s in G , senza necessariamente dover visitare tanti vertici
- Viene usata per trovare cicli nei graf.

di utti (lo vedremo più avanti)

