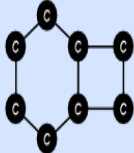
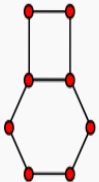
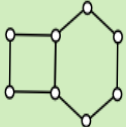



# Algoritmi e strutture dati

Grafi: visita in ampiezza e problemi collegati

CHEMISTRY	SOCIAL NETWORKS	BIOLOGY	MATH
 <p>BENZOCYCLOBUTADIENE</p> <p>● CARBON ATOMS — <math>\sigma</math>-ELECTRON BONDS</p>	<p>spikedmath.com © 2011</p>  <p>● INDIVIDUALS — FRIENDSHIPS</p>	 <p>PPI (SUB)NETWORK OF A SIMPLE ORGANISM</p> <p>○ PROTEINS — INTERACTIONS</p>	<p>THEY LOOK THE SAME TO ME.</p> <p>LET'S CALL IT A GRAPH.</p> 

"MATHEMATICS IS THE ART OF GIVING THE SAME NAME TO DIFFERENT THINGS."  
JULES HENRI POINCARÉ (1854-1912)

# Menú di questa lezione

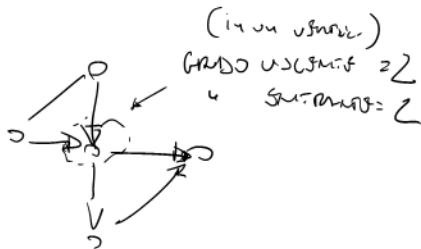
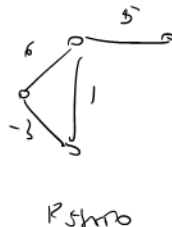
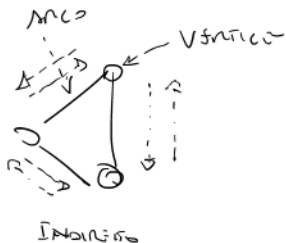
In questa lezione introduciamo il concetto di grafo, e vediamo il primo algoritmo di visita e le sue applicazioni.

# Grafi: introduzione

Un **grafo** è una tripla  $G = (V, E, W)$  composta da un insieme  $V$  di **vertici**, un insieme  $E \subseteq V \times V$  di **archi**, e una funzione  $W : E \rightarrow \mathbb{R}$  che assegna un peso ad ogni arco. Il grafo  $G$  è **indiretto** se vale sia  $(u, v) \in E \Leftrightarrow (v, u) \in E$  che  $W(u, v) = W(v, u)$ , e **diretto** altrimenti. Quando da un vertice  $u$  possiamo raggiungere un vertice  $v$  usiamo il simbolo  $u \rightsquigarrow v$ . Il **grado entrante** di un **nodo** di un grafo è il numero di archi che lo raggiungono, ed il **grado uscente** il numero di archi che lo lasciano. Il grafo  $G$  è **pesato** se  $W$  non è costante, e **non pesato** altrimenti (spesso, in questo caso, usiamo la notazione  $G = (V, E)$ ). Nel corso dello studio degli algoritmi elementari e non elementari su grafi, faremo riferimento ad eventuali ipotesi aggiuntive (per esempio: questo algoritmo è definito per grafi diretti, non pesati, e aciclici). In assenza di specifiche, si intende che un certo algoritmo o problema ha senso per tutti i casi.

Chiamiamo **sparso** un grafo tale che  $|E| \ll |V|^2$ , e lo definiamo **denso** altrimenti. Chiaramente  $|E|$  può, al massimo, arrivare a  $|V|^2$ . Ci sono due modi standard di rappresentare un grafo: con **liste di adiacenza** o con una **matrice di adiacenza**. In entrambi i casi, è conveniente pensare che ogni vertice  $v \in V$  sia identificabile con un numero naturale da 1 a  $|V|$ ; nello pseudo codice faremo questa ipotesi, ma, allo stesso tempo, ci riferiremo ai vertici come oggetti.

# ESempi di Grafi



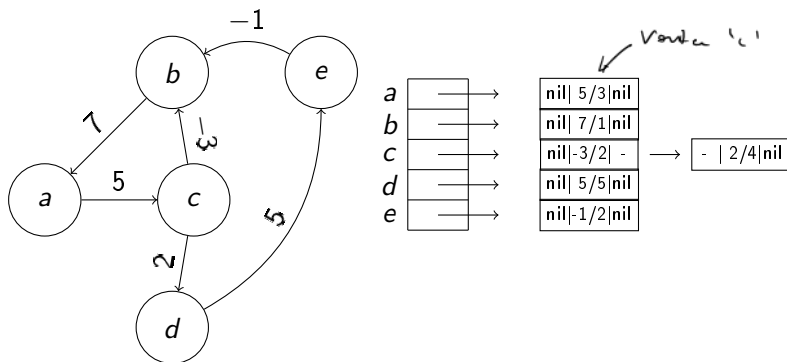
grafo  
di  
grado 3



grafo di  
grado 3

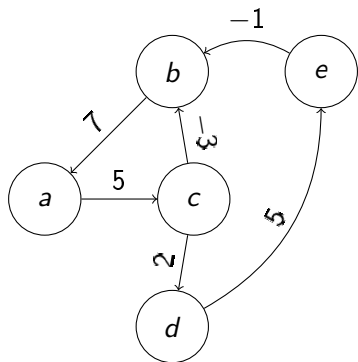
# Grafi: introduzione

Nel caso della rappresentazione a liste di adiacenza, usiamo un array  $Adj[1, \dots, |V|]$  dove ogni elemento punta a una lista (come in una tabella hash con conflitti risolti via chaining, ma non è la stessa cosa!). Per ogni vertice  $v$ , la lista puntata da  $Adj[v]$  contiene la chiave  $u$  se e solo se  $(v, u) \in E$ , e nel caso più generale, il nodo della lista contiene anche il peso dell'arco. Si preferisce questa rappresentazione per grafi sparsi.



# Grafi: introduzione

Nel caso della matrice di adiacenza, nella sua versione piú generale, usiamo una matrice  $W$  che contiene sia l'informazione sul peso di ogni arco sia quella sulla sua esistenza. Chiaramente è una matrice quadrata di lato  $|V|$ , e si preferisce questo metodo per grafi densi. Invece di  $W[i,j]$ , si usa la scrittura  $W_{ij}$ .



	a	b	c	d	e
a			5		
b	7				
c		-3		2	
d					5
e		-1			

↑  
de 'i' e 'j'

# Grafi: introduzione

I grafi (e gli algoritmi su di essi) sono pensati per essere **statici**. Infatti non abbiamo detto nulla circa l'aggiunta o l'eliminazione di un nodo da un grafo (e nessuno degli algoritmi che vedremo parlano o usano queste operazioni). Questo giustifica le due rappresentazioni viste sopra. Ma, come ci comportiamo per un grafo dinamico? Esistono almeno due soluzioni per la rappresentazione in questo caso. La prima prevede nodi come oggetti (strutture) che includono puntatori ai nodi (che rappresentano gli archi). Questa soluzione è complessa quando il grafo è pesato, e quando è ignoto il grado uscente massimo. La seconda prevede che sia i nodi, sia gli archi siano oggetti, entrambi inseriti in liste doppiamente collegate. Per quanto riguarda lo studio degli algoritmi su grafi, ci riferiremo solamente alle prime due versioni. Per noi, allora, un grafo è una struttura **statica, non basata sull'ordinamento**, e rappresentata in maniera **sparsa o compatta** a seconda del caso.



Per quanto riguarda il trattamento degli algoritmi su grafi usiamo la classica notazione  $u.att$  per indicare un attributo  $att$  associato con un vertice  $v$ . L'implementazione di algoritmi specifici può richiedere un trattamento più specializzato che dipende da caso a caso. Per quanto riguarda la complessità degli algoritmi su grafi, dobbiamo tenere conto di entrambe le componenti  $E$  e  $V$ . Tipicamente, un algoritmo **lineare** nel caso peggiore avrà complessità  $\Theta(|V| + |E|)$  o  $O(|V| + |E|)$ . In questa maniera possiamo poi riportarci a casi particolari di grafi sparsi o densi.

# Grafi: introduzione

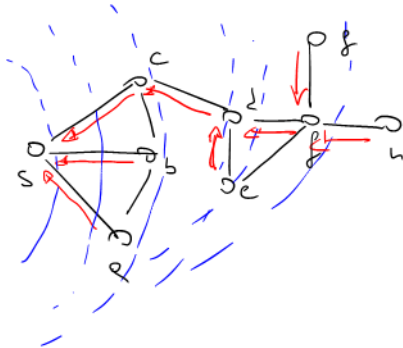
Le applicazioni dei grafi sono innumerevoli, perchè un grafo può essere usato per rappresentare molti aspetti della realtà. Un grafo può essere la rappresentazione di una rete sociale, dove i nodi sono le persone e gli archi le amicizie o i legami; la rappresentazione di un programma, dove i nodi sono funzioni e gli archi sono le dipendenze (esempio: ho bisogno di conoscere  $f()$  per compilare  $g()$ ); o la rappresentazione di una mappa, dove i nodi sono i luoghi e gli archi sono le strade che portano da un luogo all'altro. Poi, esistono applicazioni più complesse, per esempio in intelligenza artificiale, che usano grafi ad esempio per rappresentare testi dai quali si vuole estrarre informazione.

## Grafi: visita in ampiezza

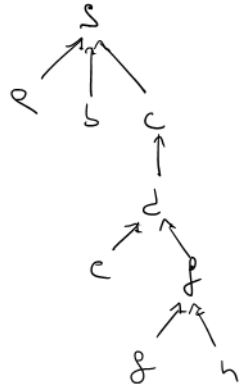
Dato un grafo  $G = (V, E)$  diretto o indiretto, non pesato, ed un vertice particolare chiamato **sorgente**  $s \in V$ , vogliamo sapere quanti archi sono necessari a raggiungere qualunque altro vertice (raggiungibile) da  $s$ . A questo fine, utilizzeremo una visita di  $G$  chiamata **in ampiezza**.

*BreadthFirstSearch* esplora sistematicamente gli archi di  $G$ , e cerca di **scoprire** nuovi vertici raggiungibili da quelli già conosciuti. Computa la distanza minima in termini di numero di archi che esiste tra  $s$  ed ogni vertice scoperto e produce un **albero di visita in ampiezza** che contiene tutti i vertici raggiungibili. Sebbene questa visita e questo problema funziona sia per grafi pesati che non pesati, non ha tanto senso utilizzarla nel primo caso, già che i pesi non giocano nessun ruolo. Il problema di stabilire il peso del camminino minimo tra  $s$  ed un vertice qualsiasi in un grafo pesato lo risolveremo più avanti. Allo stesso modo, viene naturale applicare la visita in ampiezza a grafi indiretti, sebbene quest'ipotesi non sia necessaria.

క్రింది "Graph" లో,  
 VISIT in APPROX



ALSO VISIT  
 in APPROX



# Grafi: visita in ampiezza

La visita in ampiezza è stata introdotta da Konrad Zuse nel 1945, e poi re-inventata da Edward Moore nel 1959.



## Grafi: visita in ampiezza

Utilizzeremo i colori bianco, grigio e nero per colorare i vertici mentre sono scoperti. Tutti sono bianchi all'inizio: quando un nuovo vertice è scoperto, diventa grigio, e quando anche tutti i vertici ad esso adiacente sono stati scoperti, diventa nero (ed il suo ruolo termina). Solo  $s$  è colorato di grigio all'inizio. Durante la scoperta,  $s$  diventa la radice dell'albero di visita in ampiezza. Alla scoperta di un nuovo vertice  $v$  (da bianco diventa grigio) a partire da un vertice già scoperto  $u$ , l'arco  $(u, v)$  diventa parte dell'albero (virtualmente), e  $u$  viene marcato come **predecessore** di  $v$  nell'albero stesso. Un vertice bianco viene scoperto (e colorato di grigio) al massimo una volta: quindi l'albero risultante è ben definito. Per convenienza, assumeremo che  $G$  sia rappresentato con liste di adiacenza. Nel codice,  $Q$  è una coda con politica FIFO.

# Grafi: visita in ampiezza

\* occhi  
per seguire  $u$  e  $s$

proc **BreadthFirstSearch** ( $G, s$ )

```
for ( $u \in G.V \setminus \{s\}$ )  
  {  
     $u.color = WHITE$   
     $u.d = \infty$   
     $u.\pi = nil$   
     $s.color = GREY$   
     $s.d = 0$   
     $s.\pi = nil$   
     $Q = \emptyset$   
    Enqueue( $Q, s$ )  
    while ( $Q \neq \emptyset$ )  
      {  
         $u = Dequeue(Q)$   
        for ( $v \in G.Adj[u]$ )  
          {  
            if ( $v.color = WHITE$ )  
              then  
                {  
                   $v.color = GRAY$   
                   $v.d = u.d + 1$   
                   $v.\pi = u$   
                  Enqueue( $Q, v$ )  
                }  
             $u.color = BLACK$   
          }  
      }
```

Predecessore

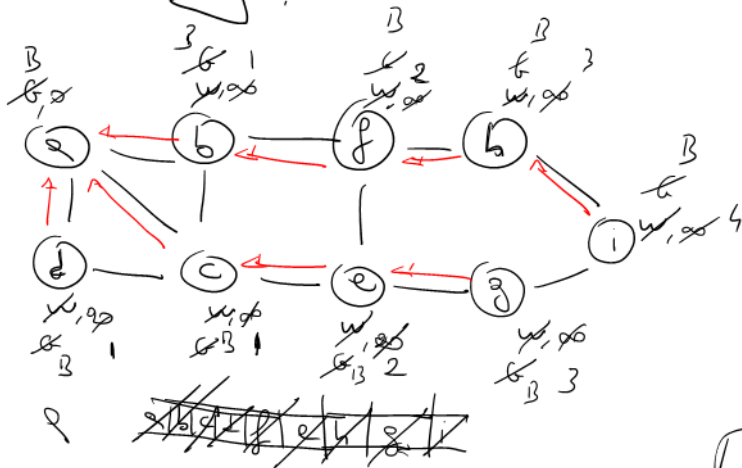
predecessore

Test  $\Delta$

Test  $\Delta$   $v$   
come u e v  
white

Visita v  
e padre

SS Snelzo  $\sqrt{12}$  :

$$\text{BFS}(G, q)$$


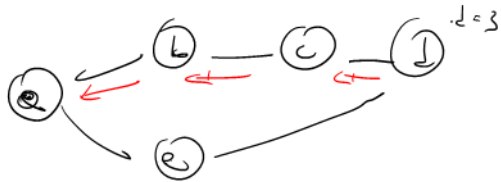
A



Tutti i nodi ad eccezione di  $s$  sono inizializzati con il colore bianco, la loro distanza a infinito, e il loro padre nell'albero di visita a nullo. L'uso della coda è essenziale: i vertici vengono inseriti nella coda e gli adiacenti ai vertici vengono scoperti nell'ordine di inserimento: si vede qui come la visita è effettivamente in ampiezza. L'elemento  $v.\pi$ , per ogni  $v$ , è chiamato **puntatore** (al **padre** di  $v$  nell'albero di visita in ampiezza). Ma non dobbiamo lasciarci ingannare: non è un puntatore nel senso classico del termine, bensì contiene il nome (il vertice) che è il padre di  $v$  nell'albero di visita.

Definiamo la **distanza piú corta** di  $v$  dalla sorgente  $s$  (denotata con  $\delta(s, v)$ ) come il **numero minimo di archi che sono necessari per raggiungere  $v$  da  $s$** . Le sue proprietà sono il fatto che essa sia zero tra un vertice e se stesso (riflessività,  $\delta(s, s) = 0$ ), che sia infinito da  $s$  a  $v$  quando il secondo è irraggiungibile dal primo ( $\delta(s, v) = \infty$ ), che sia una distanza (disuguaglianza triangolare, cioè per ogni coppia di vertici  $v, u$  tali che esiste un arco  $(u, v)$ , succede che  $\delta(s, v) \leq \delta(s, u) + 1$ ). Si osservi che la disuguaglianza triangolare vale anche se uno tra  $u, v$ , o entrambi, sono irraggiungibili da  $s$ .

ESEMPIO DI VISITA NON CONNESSA



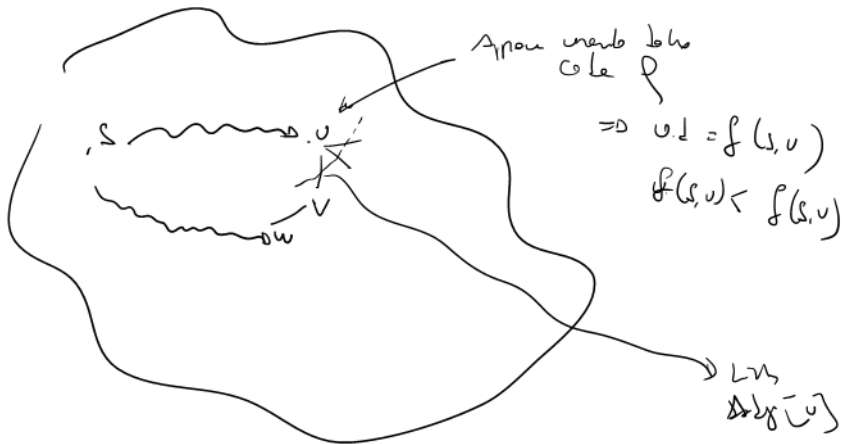
## Correttezza e complessità di *BreadthFirstSearch*

**Correttezza e terminazione** di *BreadthFirstSearch*. Mostriamo che dopo l'esecuzione, per ogni  $v$  raggiungibile da  $s$ ,  $v.d = \delta(s, v)$ , e se  $v \neq s$ , almeno uno dei percorsi più brevi da  $s$  a  $v$  si ottiene da uno dei percorsi più brevi da  $s$  a  $v.\pi$  con l'arco  $(v.\pi, v)$  (questo implica che ogni vertice è stato scoperto). Per dimostrarlo, usiamo la seguente **invariante** in due parti: primo, se un vertice  $v$  entra in  $Q$  allora  $v.d = \delta(s, v)$ , e, secondo, quando un vertice  $v$  esce dalla coda  $\delta(s, v) < \delta(s, w)$  per ogni  $w$  ancora nella coda. L'induzione è sul numero di operazioni *Enqueue*.

## Correttezza e complessità di *BreadthFirstSearch*

- **Caso base:**  $v = s$ . Chiaramente all'entrata in  $Q$   $s.d = 0 = \delta(s, s)$ , e all'uscita  $\delta(s, s) = 0$  è minima.
- **Caso induttivo.** Consideriamo  $v$  scoperto durante l'esplorazione degli archi di  $u$ . Supponiamo, per assurdo, che  $v$  sia il primo vertice che entra nella coda con  $v.d \neq \delta(s, v)$ . Siccome  $v$  è entrato, questo è accaduto perché il vertice  $u$  è appena uscito, ed è stato fatto l'assegnamento  $v.d = u.d + 1$ . Se, nell'ipotesi assurda,  $v.d < \delta(s, v)$ , allora esiste  $w$  che è ancora in  $Q$  tale che  $\delta(s, w) < \delta(s, u)$ ; ma questo contraddice l'ipotesi induttiva che  $u$  avesse il minimo  $\delta$  all'uscita. Se, nell'ipotesi assurda,  $v.d > \delta(s, v)$ , allora esiste  $w$  con  $\delta(s, w) < \delta(s, u)$ , già uscito dalla coda, ed esiste l'arco  $(w, v)$ , ma questo contraddice l'ipotesi che  $v$  sia stato scoperto da  $u$  e la disuguaglianza triangolare. Dunque  $v.d = \delta(s, v)$ . Chiaramente questo implica anche che, all'uscita,  $\delta(s, v)$  è minimo.

CMO INDUCTION BUT DIR. DI CONSTRUCTION



## Correttezza e complessità di *BreadthFirstSearch*

La **complessità** della visita è semplice da calcolare: infatti, è facile osservare che un nodo entra nella coda al massimo una volta, poiché nessuno colora di bianco alcun nodo (dopo l'inizializzazione). Per ogni nodo che è entrato nella coda, si analizzano i suoi adiacenti. Ma il totale degli archi è comunque  $|E|$  (grazie alla rappresentazione con liste di adiacenza). Questa analisi è anche chiamata **aggregata**, ed è usata spesso negli algoritmi su grafi. E' una alternativa alle analisi che abbiamo fatto finora, che risponde all'esigenza di calcolare la complessità di un algoritmo che si esegue su una struttura la cui topologia non è descrivibile in maniera semplice. Possiamo fare ciò solo quando gli algoritmi sono sufficientemente semplici, perchè dobbiamo avere l'intuizione sul numero totale di passi che si eseguono senza contarli esplicitamente. Con questa analisi, se il grafo è connesso, allora la complessità è  $\Theta(|V| + |E|)$  in tutti i casi. Se, invece, il grafo è sconnesso, allora è  $O(|V| + |E|)$ , perchè una parte degli archi potrebbe non essere mai vista. La funzione  $|V| + |E|$  **diventa**  $|V|^2$  quando il grafo è denso. Quindi, nel caso peggiore (grafo connesso e denso) è  $\Theta(|V|^2)$ , ma normalmente si accetta  $\Theta(|V| + |E|)$ .

La visita in ampiezza è la più semplice tra le visite e uno degli algoritmi più semplici che vediamo sui grafi. Come si vede dagli esercizi, però, è particolarmente versatile e utile nella pratica.