

Grafi

(Parte 2)

Cammini minimi su grafi pesati

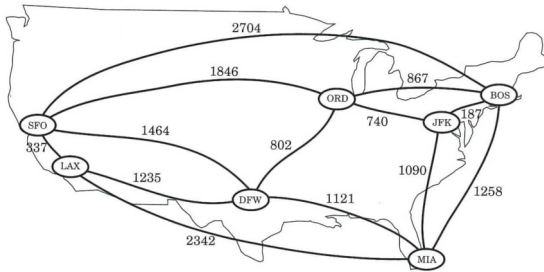
Algoritmo di Dijkstra

Cammini minimi su grafi pesati

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

- V è l'insieme dei vertici;
- E è l'insieme degli archi:
- $w : E \rightarrow \mathbb{R}$ funzione che associa un peso reale a ciascun arco $e \in E$. Può essere rappresentato con un campo **e.weight**.

Esempio (da [GTG14]):



Rete dei principali aeroporti americani con distanze in km

Cammini minimi su grafi pesati

Si ricordi che:

- La **lunghezza di un cammino** $u_1, u_2, \dots, u_k$ è la somma dei pesi degli archi del cammino, ovvero

$$\sum_{i=1}^{k-1} w(u_i, u_{i+1}).$$

Oss.: Se $w(e) = 1$ per ogni $e \in E$, la lunghezza di un cammino è pari al numero di archi (come in un grafo non pesato).

- La **distanza** $d(u, v)$ tra i due vertici $u, v \in V$ è la **minima lunghezza di un cammino da u a v** , e il cammino che la realizza si chiama **cammino minimo (shortest path) da u a v**

Cammini minimi su grafi pesati

Proposizione

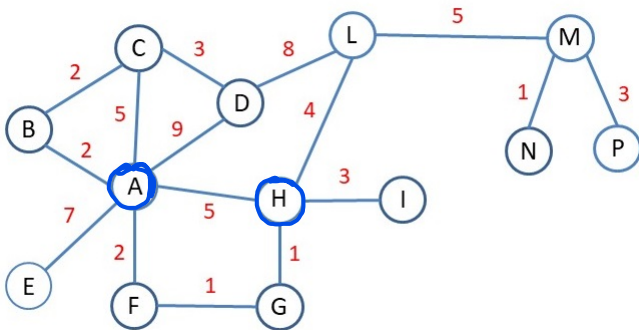
Se $u_1, u_2, \dots, u_k$ è un cammino minimo da u_1 a u_k , allora $u_i, u_{i+1}, \dots, u_j$ è un cammino minimo da u_i a u_j , per ogni $1 \leq i < j \leq k$.

DM. $u_1 - u_2 - \dots - u_i - u_{i+1} - \dots - u_j - \dots - u_k$
 $w_{ij} = \text{lunghezza di } u_i u_{i+1} \dots u_j$

PER ASSURDO : se esistesse un cammino da u_i a u_j di lunghezza $< w_{ij}$, sostituendo tale cammino al segmento $u_i u_{i+1} \dots u_j$ nel cammino da u_1 a u_k si otterrebbe un cammino da u_1 a u_k

di lunghezza minore, che è impossibile
dato che si è assunto che il cammino
da u_1 a u_k fosse minimo. $\square$

Esempio



Cammino minimo da A a H è: A, F, G, H
di lunghezza 4

Cammini minimi su grafi pesati

Come calcoliamo **distanze** e **cammini minimi** in un grafo pesato?

Definizione del problema

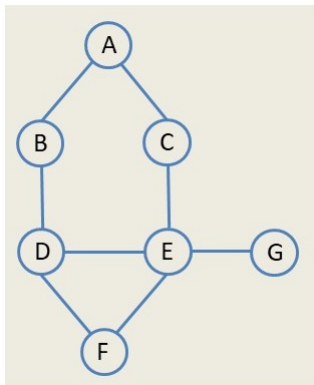
Dato un grafo pesato $G = (V, E, w)$, non diretto, e un vertice $s \in V$, il problema dei **Single-Source Shortest Paths (SSSP)** richiede di determinare **tutte le distanze** tra s e gli altri vertici di V e i relativi **cammini minimi**, opportunamente rappresentati.

Ipotesi: Assumiamo che G non contenga cicli di peso negativo, altrimenti per coppie di vertici nella stessa componente connessa del ciclo esisterebbero cammini di lunghezza negativa arbitrariamente piccola.

Osservazioni:

- Per i vertici non appartenenti alla componente connessa di s , la distanza da s sarà $+\infty$ e il relativo cammino minimo sarà vuoto.
- Se $w(e) = 1$ per ogni $e \in E$, la BFS risolve il problema dei SSSP in tempo $\Theta(|V| + |E|)$.

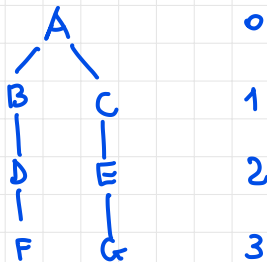
Grafo non pesato



Cammini minimi da $s = A$

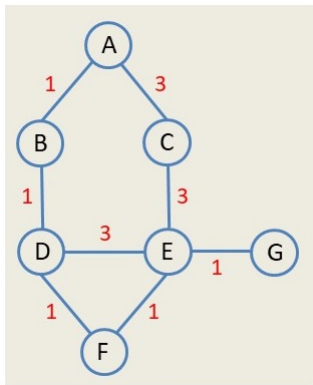
Albero dei
Discovery Edge

distanza



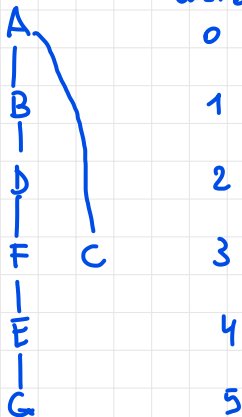
la BFS scopre i vertici in ordine di distanza
dalla sorgente s (A in questo caso)

Grafo pesato



Cammini minimi da $s = 1$

distanze



I cammini minimi da vert. sorgente
nel seguente ordine temporale: A B D C F E G

che è diverso dall'ordine in cui sono
scritti se il graf non è diretto

Algoritmo di Dijkstra per SSSP

- Algoritmo fondamentale sviluppato nel 1956 da E. Dijkstra.
- Richiede che i **pesi siano non negativi**.
- Generalizza lo schema della BFS al caso pesato. A partire da un vertice **sorgente $s \in V$** :
 - Esegue una serie di iterazioni che fanno crescere progressivamente la **cloud di vertici le cui distanze da s e i relativi cammini minimi sono stati identificati**;
 - La **cloud è inizializzata con $\{s\}$** ;
 - Per ogni **$v \in V$ mantiene la distanza corrente di v da s** , relativa a cammini interni alla cloud (tranne, eventualmente v);
 - In ciascuna iterazione viene **aggiunto alla cloud il vertice esterno alla cloud più vicino a s** ;
 - I **vertici esterni alla cloud sono mantenuti in una Priority Queue Q** in base alla loro distanza da s .

Dettagli implementativi

Per ogni vertice $v \in V$ usiamo i seguenti campi:

- $v.D$: memorizza la distanza corrente da s ($D[v]$ in [GTG14]);
- $v.parent$: memorizza il predecessore di v nel cammino ~~minimo~~ corrente da s a v (che quindi risulta definito dai campi `parent` dei vertici coinvolti).

Per la Priority Queue Q , assumiamo che:

- Un vertice $v \in V$ è rappresentato da una entry $(v.D, v)$, con chiave $v.D$.
- È disponibile un metodo $Q.decreaseKey(v.D, v)$ che, se Q contiene una entry (x, v) per il vertice v , con $x > v.D$, sostituisce la chiave di tale entry con $v.D$.

Algoritmo di Dijkstra: pseudocodice

Algoritmo ShortestPaths(G, s)

Input: grafo non diretto $G = (V, E, w)$ con pesi non negativi sugli archi, e sorgente $s \in V$

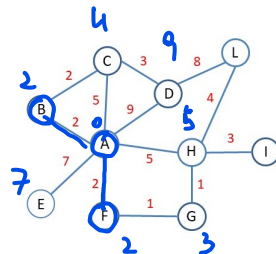
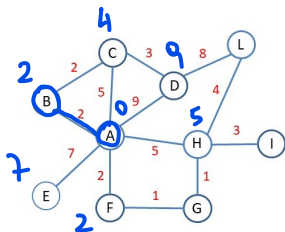
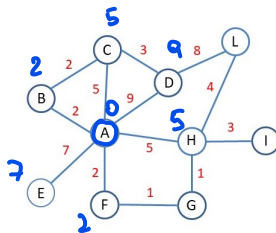
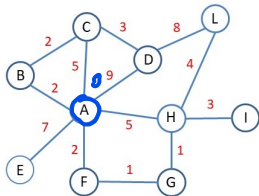
Output: distanze e cammini minimi da s per ogni $v \in V$, rappresentati dai campi $v.D$ e $v.parent$

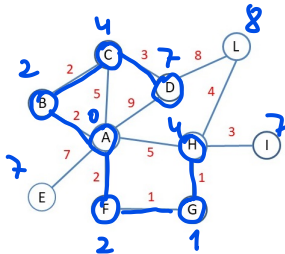
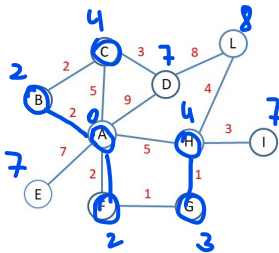
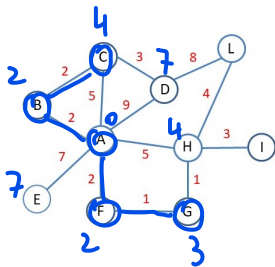
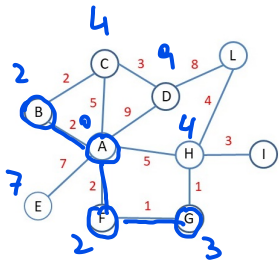
Algoritmo di Dijkstra: pseudocodice

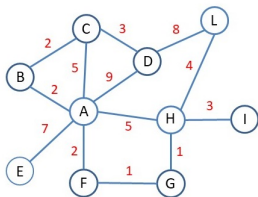
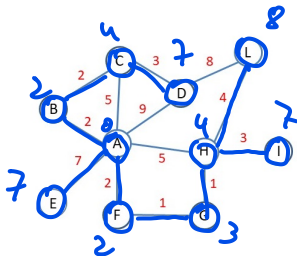
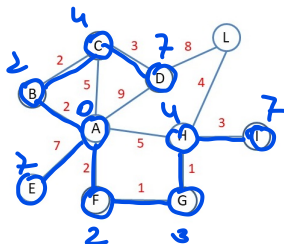
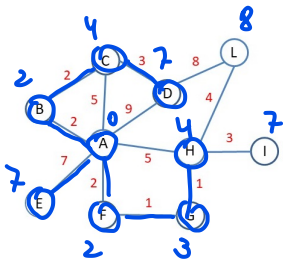
```
s.D  $\leftarrow$  0;
s.parent  $\leftarrow$  null;
forall  $v \in V - \{s\}$  do
    v.D  $\leftarrow$   $+\infty$ ;
    v.parent  $\leftarrow$  null
Q  $\leftarrow$  Priority Queue contenente una entry (v.D, v) per ogni  $v \in V$ ;
while (!Q.isEmpty()) do
    (u.D, u)  $\leftarrow$  Q.removeMin();
    forall ((u, v)  $\in$  G.incidentEdges(u)) do
        /* Edge relaxation */
        if (u.D + w(u, v) < v.D) then
            v.D  $\leftarrow$  u.D + w(u, v);
            v.parent  $\leftarrow$  u;
            Q.decreaseKey(v.D, v)
```

Esempio

$S = A$







Correttezza di ShortestPaths

Sia $G = (V, E, w)$ un grafo non diretto con pesi non negativi sugli archi e sia $s \in V$ un vertice arbitrario.

Lemma 1

Alla fine di ciascuna iterazione del ciclo **while** di ShortestPaths(G, s), vale il seguente invariante. Per ogni $v \in V$:

- se $v.D = +\infty$, allora $v.parent = \text{null}$.
- se $v.D < +\infty$, il cammino ottenuto risalendo da v di “parent in parent” è un cammino da s a v di lunghezza $v.D$.

Lemma 2

Durante l'esecuzione di ShortestPaths(G, s), quando un vertice u è estratto da Q , il campo $u.D$ è uguale alla distanza $d(s, u)$ di u da s .

Correttezza di ShortestPaths

Teorema

L'algoritmo ShortestPaths(G, s) è corretto.

DIMOSTRAZIONE del TEOREMA. Assumiamo veri i 2 lemmi

- * Per costruzione, tutti i vertici di V sono inseriti in Q e, prima o poi, ogni vertice sarà estratto
- * Quando un vertice $v \in V$ è estratto da Q si ha che
 - $V.D$ è la lunghezza del cammino minimo da s a v (del lemma 2)
 - Se $V.D < +\infty$, risalendo di parent in parent

s. ottiene un cammino da s a v di
lunghezza $V.D \Rightarrow$ cammino minimo
(dal lemma 1)

- Se $V.D = +\infty$, $v.parent = null$ (dal lemma 1)
ed è corretto dato che non esiste un cammino
da s a v □

DIMOSTRAZIONE del LEMMA 1

All'inizio del while l'invariante è garantito dalle inizializzazioni:

$$\left. \begin{array}{ll} S.D = 0 & V.D = +\infty \\ S.parent = \text{null} & V.parent = \text{null} \end{array} \right\} \forall v \in V - \{s\}$$

ESERCIZIO: Dimostrare che se l'invariante vale alla fine di una generica iterazione, le edge relaxation eseguite nella successiva iterazione lo rendono vero alla fine di essa.

La correttezza del lemma deriva immediatamente dall'esercizio e dalla osservazione che lo precede $\square$

DIMOSTRAZIONE del LEMMA 2. Assumiamo vero il lemma 1.

PER ASSURDO: Supponiamo che il lemma 2 non sia vero

e sia z il PRIMO VERTICE ESTRATTO da Q con

$z.D \neq d(s, z)$. Deve valere per forza che

* $z \neq s$ (dato che s è estratto per primo, con $s.D = 0$)

* $z.D > d(s, z)$ dato che il lemma 1 assicura

che esiste un cammino da s a z di lunghezza $z.D$

Sia P un cammino minimo da s a z

$$P \equiv \underset{s}{\overset{s}{0}} \text{-----} \underset{z}{\overset{z}{0}}$$

* **CASO1**: Ogni vertice $v \neq z$ in P viene estratto da

Q prima di z . Sia x il predecessore di z in P

Allora quando x viene estratto da Q , $x.D = d(s, x)$

e dopo la edge relaxation eseguita sull'arco (x, z)

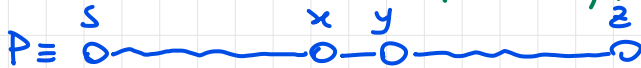
deve valere che

$$z.D \leq x.D + w(x, z) = d(s, x) + w(x, z) = d(s, z)$$

ma questo controbale l'ipotesi che $z.D > d(s, z)$

* **CASO 2.** Sia y il primo vertice di P che è ancora in Q quando z viene estratto da Q e sia x il suo predecessore in P . (Chiaramente y deve essere $\neq s$)

N.B. Il segmento da s a y è minimo, perché P è minimo



Quindi: x è estratto da Q prima di z e, data la scelta di z , quando x è estratto si ha che

$x.D = d(s, x)$. Inoltre, dopo le edge relaxation eseguite su (x, y) deve valere che

$$y.D \leq x.D + w(x, y) = d(s, x) + w(x, y) = d(s, y)$$

Adesso, dato che i pesi degli archi sono tutti ≥ 0

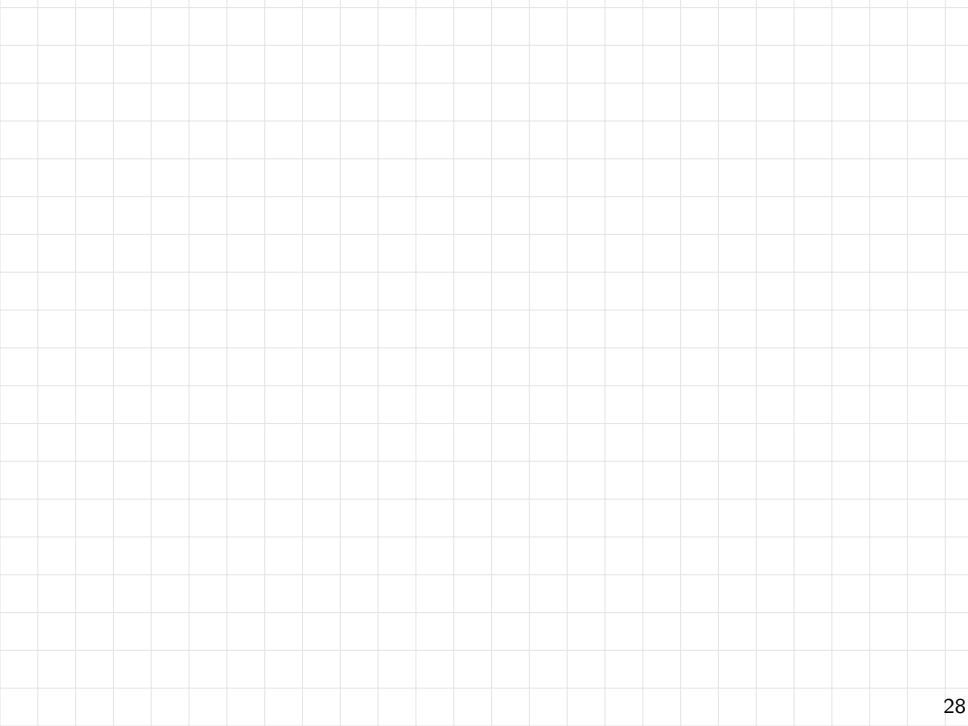
(Hp. dell'algoritmo) deve essere $d(s, y) \leq d(s, z)$

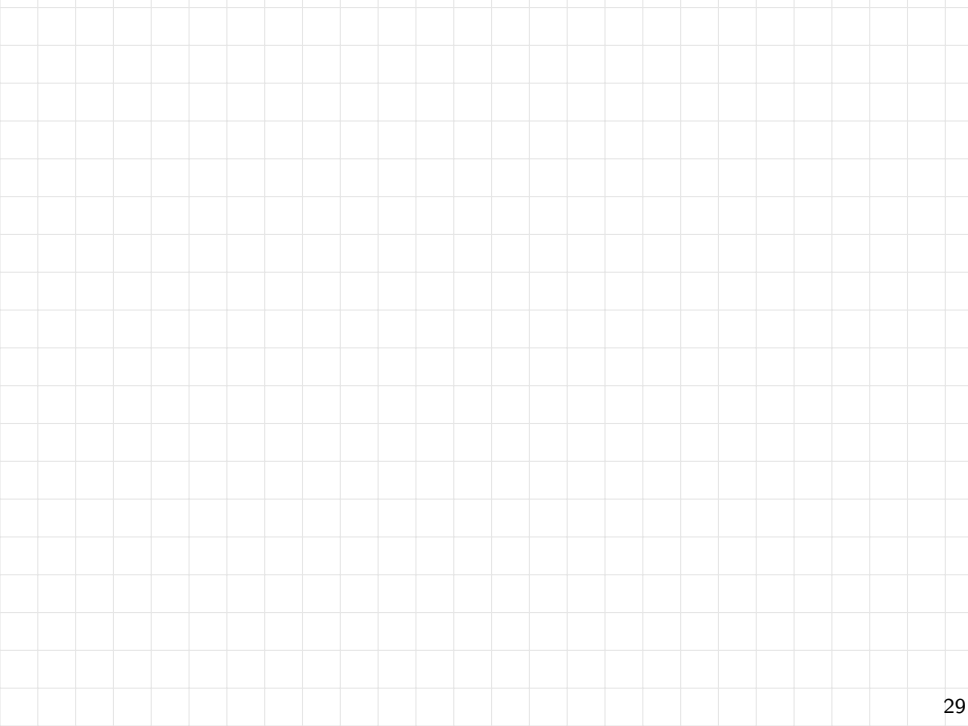
Quindi, a questo punto sappiamo che quando z viene estratto da Q

$$* z.D > d(s, z)$$

$$* y \in Q \text{ con } y.D \leq d(s, y) \leq d(s, z)$$

Ma questa è impossibile perché l'entry (z.d,z)
non sarebbe quella di estremo di Q (cioè quella
con chiave minima) □





Complessità di ShortestPaths

La complessità di ShortestPaths dipende dalla implementazione scelta per la Priority Queue Q .

Consideriamo due possibili implementazioni: **Heap** e **Doubly-linked list non ordinata**.

Lemma

Se $G = (V, E, w)$ ha n vertici, i metodi invocati da ShortestPaths(G, s) su Q possono essere implementati con le seguenti complessità:

Operazione	Heap	Doubly-Linked List
(A) Costruzione iniziale	$\Theta(n)$	$\Theta(n)$
(B) $Q.removeMin()$	$\Theta(\log n)$	$\Theta(n)$
(C) $Q.decreaseKey(v.D, v)$	$\Theta(\log n)$	$\Theta(1)$

DIMOSTRAZIONE del LEMMA

Ipotesi Implementative

- * L'oggetto che rappresenta un vertice $v \in V$ ha un link all'entry $(v.d, v)$ in Q (il link è null se l'entry è già stata estratta da Q)
- * l'entry $(v.d, v)$ ha come valore un riferimento all'oggetto che rappresenta v



Operazioni A, B : le complessità seguono direttamente da quanto visto a lezione sulle Priority Queue

Operazione C : Con le ipotesi implementative fatte, da v accesso alle entry $(v.d, v)$ in tempo $\Theta(1)$.

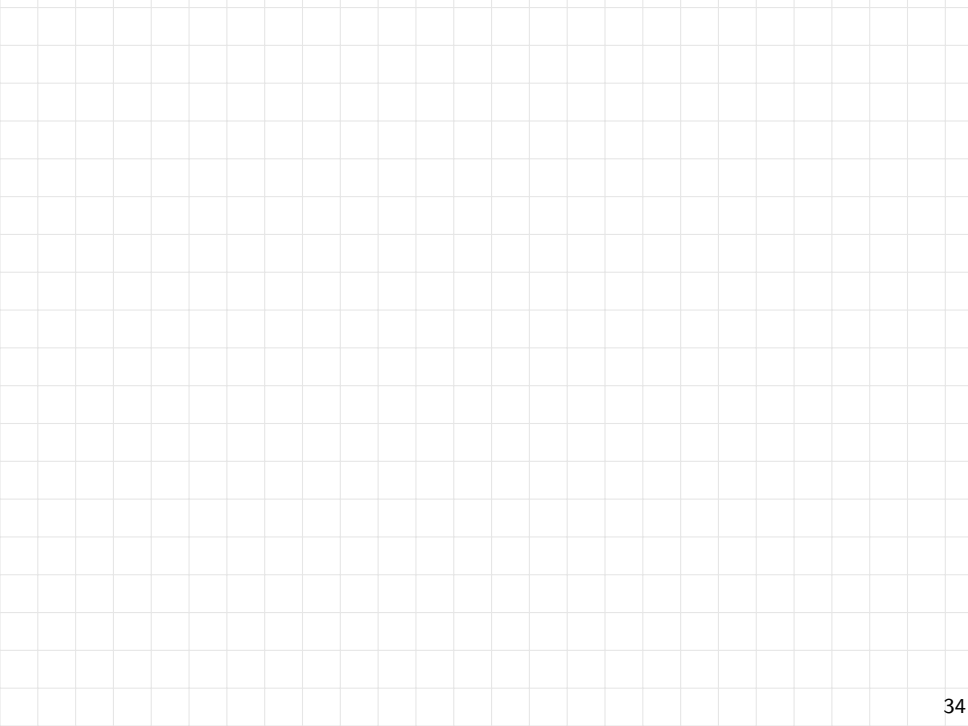
Roi :

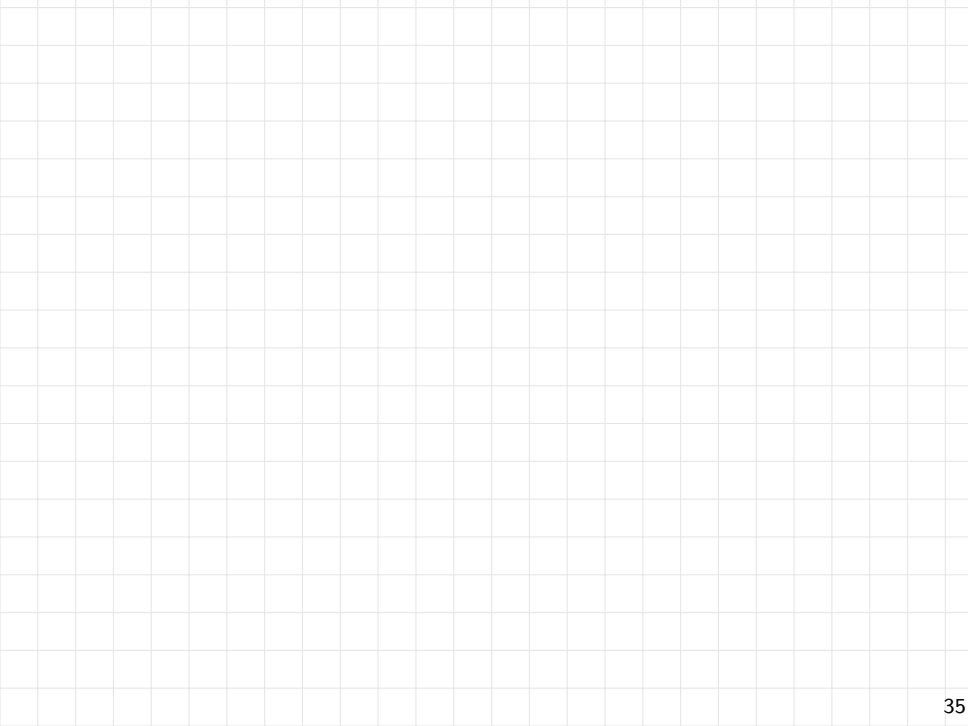
Se $Q \equiv \text{HEAP}$: aggiorni $v.d$ al valore minore ed esegui un up-heap bubbling dal nodo associato a $(v.d, v) \Rightarrow \text{Complessità} \in \Theta(\log |Q|)$

e quindi $\Theta(\log n)$, dato che in ogni momento $|Q| \leq n$

Se $Q \equiv$ LISTA NON ORDINATA : aggiungo semplicemente
v.d $\Rightarrow$ complessità $\in \Theta(1)$ $\square$

N.B. Tutte le operazioni: Considerate viscoso a
mantenere i link tra le rappresentazioni dei vertici
e le entry a essi associate in Q senza aggancio
della complessità asintotica





Teorema

Sia $G = (V, E, w)$ un grafo non diretto con n vertici ed m archi e con pesi non negativi sugli archi. Per ogni $s \in V$ la complessità di $\text{ShortestPaths}(G, s)$ è $O(\min\{n^2, (n + m) \log n\})$.

DIMOSTRAZIONE

Si osserva che ogni arco (u, v) in ESATTAMENTE 2 edge relaxation e il costo di una edge relaxation è AL PIÙ il costo di una decrease key in Q. Quindi la complessità di Shortest Path è dominata dalle seguenti operazioni:

OPERAZIONE	HEAP	LISTA NON ORDINATA
* Inizializzazione di Q	$\Theta(n)$	$\Theta(n)$
* Ciclo while (in totale)		
- n removeU	$\Theta(n \log n)$	$\Theta(n^2)$
- 2m edge relaxation	$\Theta(m \log n)$	$\Theta(m)$

⇒ COMPLESSITÀ FINALE

HEAP: $\Theta((n+m) \log n)$

LISTA NON ORDINATA: $\Theta(n^2)$

Anche l'enunciato indica come complessità

$$\Theta(\min\{n^2, (n+m)\log n\}) \quad ?$$

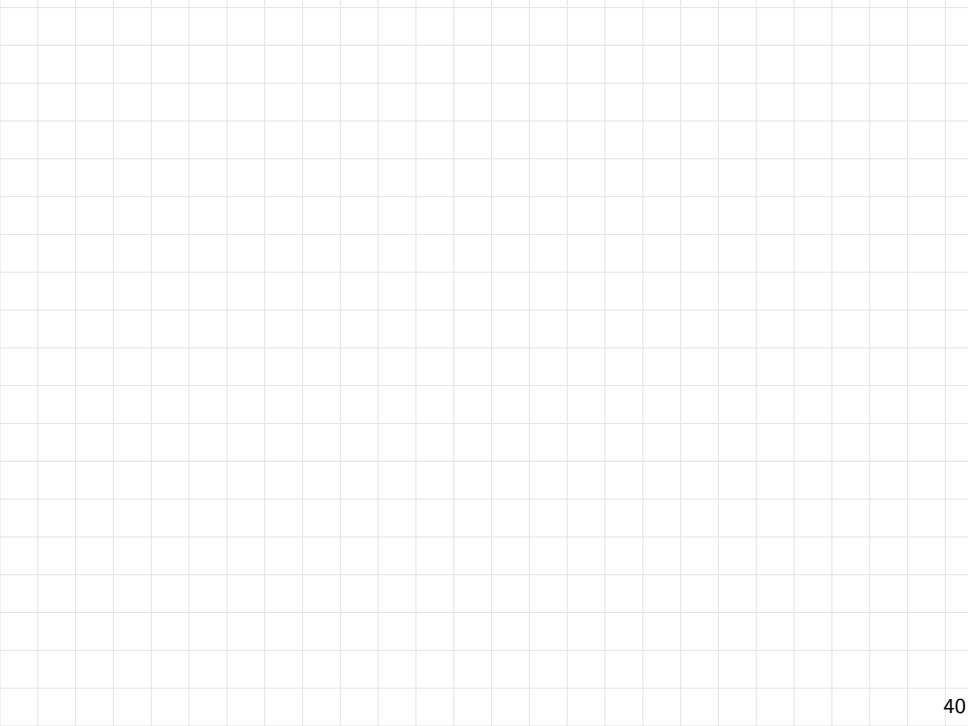
Dato che n e m , se non noti in input, possono essere calcolati facilmente in tempo $\Theta(n+m)$,

Una volta che sono conosciuti scegli l'implementazione migliore

$$* \quad m \in O(n^2/\log n) \Rightarrow \text{HEAP}$$

$$* \quad m \in \Omega(n^2/\log n) \Rightarrow \text{LISTA NON ORDINATA}$$

Osservazione: Usando un FIBONACCI HEAP in Q
la complessità scende a $\Theta(m + n \log n)$
che è avvicina alla complessità della BFS
che risolve il problema nel caso non pesato



Esercizio

Sia $G = (V, E, w)$ un grafo non diretto con n vertici ed m archi, e con pesi non negativi sugli archi.

- 1 Progettare una modifica di $\text{ShortestPaths}(G, s)$ in modo che la complessità sia $O(n + m_s \log n_s)$, dove n_s è il numero di vertici e m_s il numero di archi nella componente connessa di s (C_s).
- 2 Dire se l'algoritmo modificato calcola le distanze corrette anche per i vertici non in C_s .

1 Se G non è connesso



L'algoritmo presentato prima estrae da Q

prima tutti i vertici di C_s (con $V.D < +\infty$) e
alla fine tutti quelli di $G - C_s$ con $V.D = +\infty$
e $v.parent = null \Rightarrow$ È inutile inserire in Q
i vertici di $G - C_s$. Per evitare questo si
possono fare le seguenti modifiche a Shortest Paths
* Inizializzazione dei campi D e $parent$ ESATTAMENTE
come prima

* Q viene inizializzata solo con la entry $(s.D, s)$
dove $s.D = 0$

* Nel momento in cui si fa la edge-relaxation
 su un arco (u,v) che arco incidente su u
 e si trova $v.D = +\infty$ e $u.D + w(u,v) < +\infty$
 allora si imposta $v.D = u.D + w(u,v)$ e si
 INSERISCE la entry $(v.D, v)$ in Q
 In questo modo Q contiene solo vertici
 di C_S , e, in ogni momento, $|Q| \leq n_S$
 Esercizio: scrivere lo pseudocode

Complexità

- * Inizializzazione dei tempi D e parent: $\Theta(n)$
- * Inizializzazione di Q: $\Theta(1)$
- * Ciclo while
 - n_s iterazioni (una per ogni vertice di G_s)
 - l'iterazione che estrae $u \in G_s$ farà
 - o $\deg(u)$ edge relaxation
 - o in ciascuna di queste edge-relaxation al più 1 insert o 1 decreasekey in Q

per l'altra operazione

$\Rightarrow$ Costo di questa iterazione è
 $\Theta(\text{degree}(u) \cdot \log n_s)$

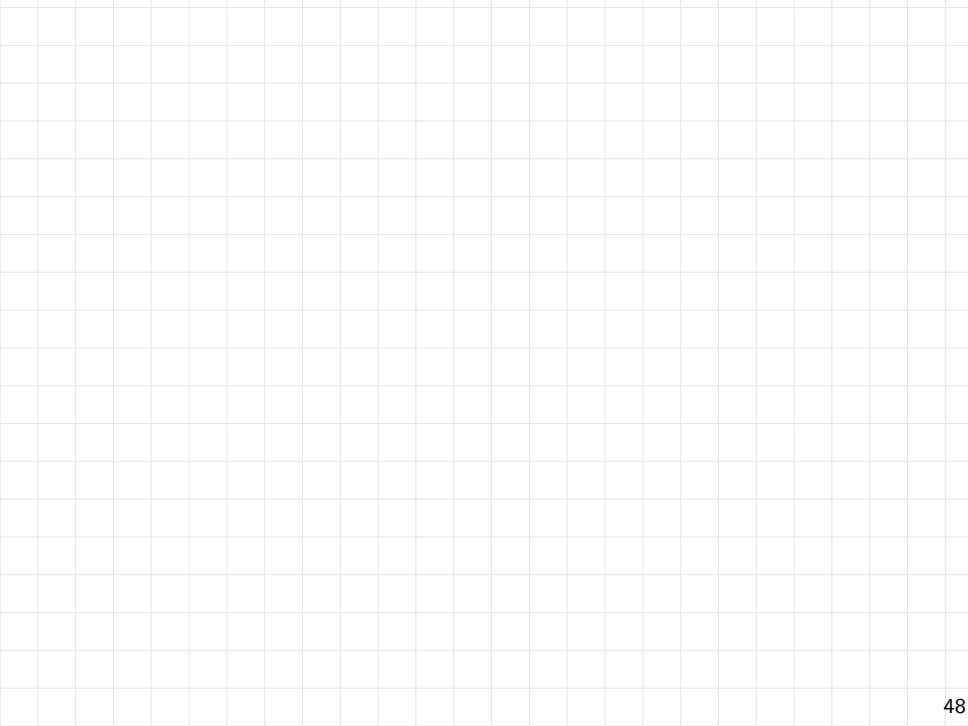
$\rightarrow$ implementare Q
come heap

$\Rightarrow$ Complessità totale

$$O\left(n + \sum_{u \in C_s} \text{degree}(u) \cdot \log n_s\right)$$

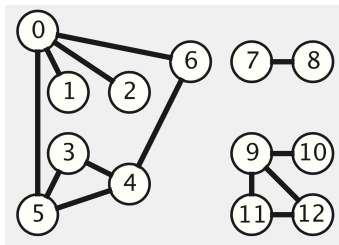
$$= O(n + m_s \log n_s)$$

② Per tutti i vertici di G alle fine
faremo i comp. D e parent
correttamente impostati.

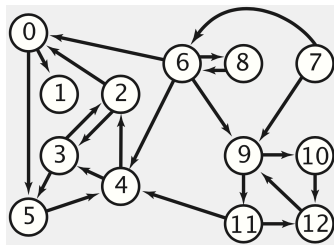


Grafi semplici e diretti

Grafo non diretto vs grafo diretto



Grafo non diretto



Grafo diretto

La differenza tra grafo diretto e non diretto è che **in un grafo diretto** $G = (V, E)$ ogni arco $e = (u, v) \in E$ è una **coppia ordinata** di vertici

Terminologia

Dato un vertice v :

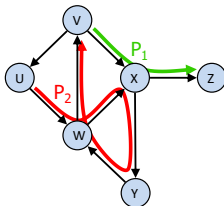
- Gli archi (v, u) si dicono **uscenti da v**
- Gli archi (u, v) si dicono **entranti in v**
- **Grado uscente** di $v \in V$ ($\text{outdegree}(v)$) = numero archi uscenti da v
- **Grado entrante** di $v \in V$ ($\text{indegree}(v)$) = numero archi entranti in v

Per un grafo diretto vale che $G = (V, E)$:

$$\sum_{v \in V} \text{outdegree}(v) = \sum_{v \in V} \text{indegree}(v) = |E|$$

Giustificazione: Ogni arco contribuisce **ESATTAMENTE 1** all'outdegree di un vertice, e 1 all'indegree di un altro vertice

Cammino, ciclo e distanza in un grafo diretto



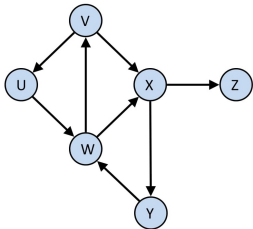
Le definizioni di **cammino** e **ciclo** sono analoghe quelle per i grafi non diretti, ma seguono la direzionalità degli archi! La definizione di **lunghezza** di un cammino/ciclo rimane la stessa.

Esempi:

- Cammini: **P1** (semplice), **P2** (non semplice)
- Ciclo: $x - y - w - x$

Dati $u, v \in V$, la **distanza** $d(u, v)$ è definita come la lunghezza minima di un cammino diretto da u a v .

Vertici raggiungibili

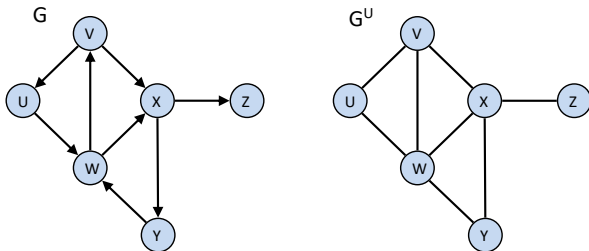


Dato $v \in V$, l'insieme di **vertici raggiungibili** da v ($\text{reachable}(v)$) consiste di tutti i vertici $u \in V$ tali che esiste un cammino diretto da v a u .

Esempi:

- $\text{reachable}(x) = \{y, z, w, v, u\}$
- $\text{reachable}(z) = \emptyset$

Versione non diretta di un Grafo diretto

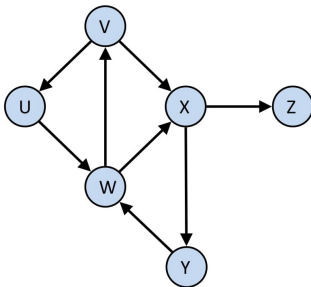


Dato un grafo diretto $G = (V, E)$, la sua **versione non diretta** G^U si ottiene ignorando la direzione di ogni arco $(u, v) \in E$.

Grafo Fortemente e Debolmente Connesso

$G = (V, E)$ si dice **fortemente connesso** (*strongly connected*) se per ogni coppia ordinata $u, v \in V$ esiste un cammino diretto da u a v in G .

$G = (V, E)$ si dice **debolmente connesso** (*weakly connected*) se G^U (versione non diretta di G) è connesso



Il graf nell'esempio

- * non è FORTEMENTE CONNESSO, perché da z non posso raggiungere nessun altro vertice
- * è DEBOLMENTE CONNESSO perché G^u è connesso

Rappresentazione degli archi: liste di adiacenza

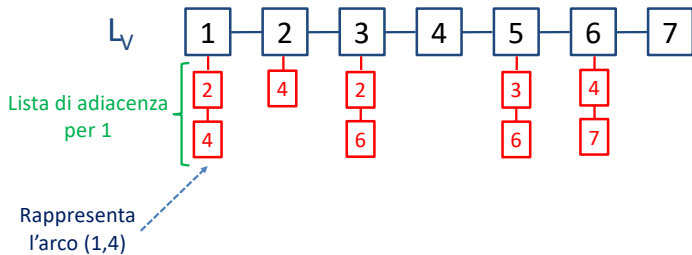
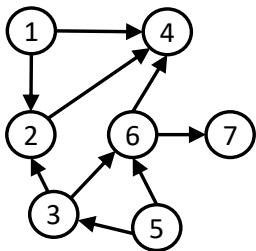
Come per i grafi non diretti, è spesso conveniente utilizzare le **liste di adiacenza** per rappresentare gli archi di un grafo diretto.

LISTE di ADIACENZA (Adjacency List): Per ogni vertice $v \in V$ si ha una lista $out(v)$ di puntatori agli archi $(v, u) \in E$ uscenti da v .

Esercizio

Sia G un grafo diretto rappresentato tramite liste di adiacenza. Progettare un algoritmo che, in tempo lineare nella taglia del grafo, salva in ogni nodo $v \in V$ due campi $v.outdeg$ e $v.indeg$ uguali, rispettivamente, a $outdegree(v)$ e $indegree(v)$.

Esempio di liste di adiacenza



Visita di un grafo diretto

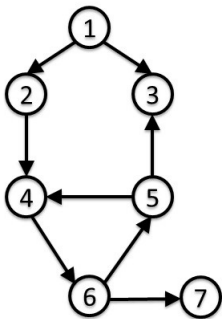
BFS e DFS si applicano a grafi diretti, con alcune piccole modifiche:

- Adesso `incidentEdges( $v$ )` restituisce un iteratore non a tutti gli archi incidenti, ma solo agli **archi uscenti da v** .
- Nella BFS un arco $e = (u, v)$ può essere etichettato come DISCOVERY EDGE o ALTRO
- Nella DFS un arco $e = (u, v)$ esaminato durante **DFS(G, u)** e non etichettato come DISCOVERY, può essere etichettato come:
 - **BACK EDGE**: se v è antenato di u nell'albero definito dai DISCOVERY EDGE
 - **ALTRO**: negli altri casi. (Il libro di testo distingue anche tra FORWARD EDGE, se v è discendente di u , e CROSS EDGE altrimenti.)

Oss.: A differenza del caso non diretto, non tutti gli archi che non sono DISCOVERY EDGE sono BACK EDGE.

Visita di un grafo diretto

Nei prossimi lucidi vediamo l'esecuzione della BFS e della DFS sul seguente grafo diretto a partire dal vertice $s = 1$



Liste di adiacenza (ordine crescente)

1: 2 3

2: 4

3:

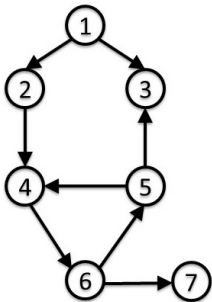
4: 6

5: 3 4

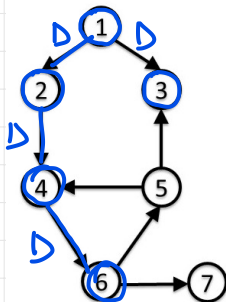
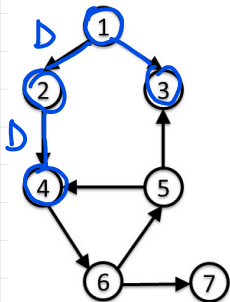
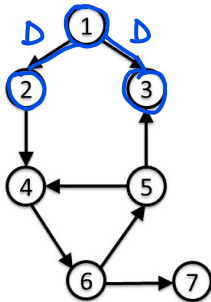
6: 5 7

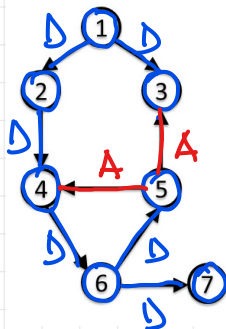
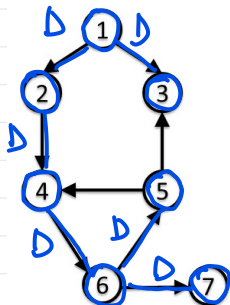
7:

$S = 1$



BFS





RISULTATO

LIVELLI

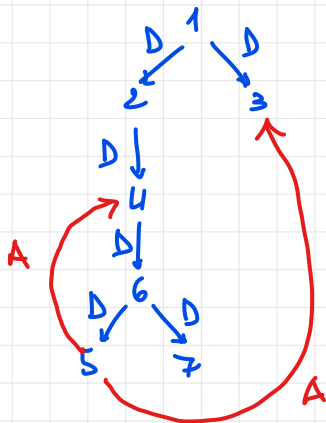
0

1

2

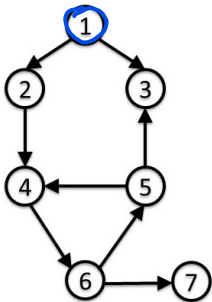
3

4

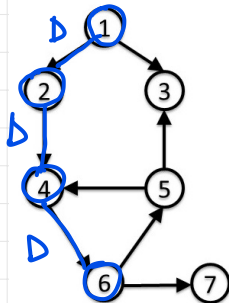
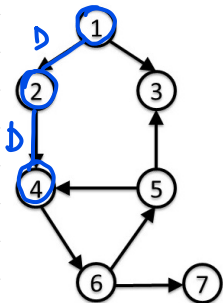
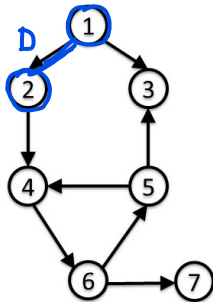


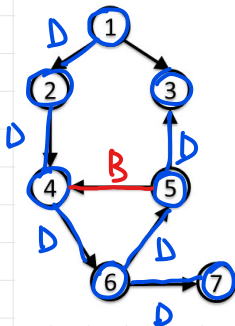
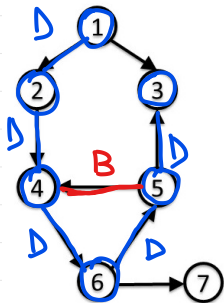
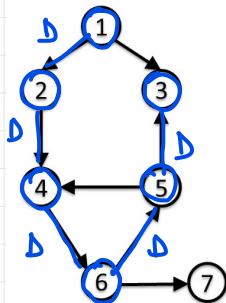
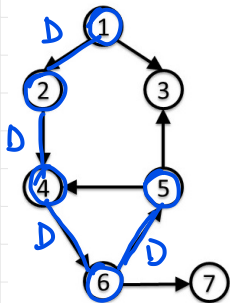
In blu ci sono i
DISCOVERY EDGE che
formano uno Spanning
tree dei vertici
raggiungibili da s

$S=1$

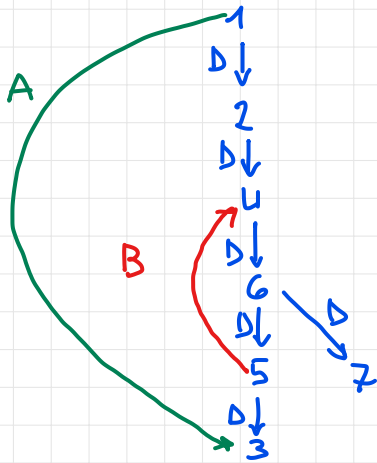


DFS





Alle fine tutte le chiamate si dividono e quella
su $S=1$ prima di chiamare etichetta (1,3) viene
ALTRO. Il risultato finale è



Esercizio

Modificare lo pseudocodice $\text{DFS}(G, s)$ in modo che ogni arco esaminato e non etichettato come DISCOVERY EDGE, sia etichettato correttamente come BACK EDGE o ALTRO, in base alla definizione data prima.

Sia (v, w) un arco esaminato da $\text{DFS}(G, v)$
e tale per cui $w.D = 1 \Rightarrow$ l'arco non può essere
etichettato come DISCOVERY EDGE. Allora i seguenti
vedono che esso è un BACK EDGE se
Quando si esamina (v, w) le chiamate
 $\text{DFS}(G, w)$ si iniziate ma non condurre.



Per identificare le sottosequenze
tutti i BACK EDGE è
sufficiente operare così:

$\forall v \in V$ aggiungiamo un flag
 booleano $V.ON$ inizializzato a
 0 per tutti e impostato a 1
 all'inizio delle chiamate $DFS(G, v)$
 e restituito a 0 alle fine delle chiamate

* Durante la chiamata $DFS(G, v)$ si etichetta (v, w) come BACK EDGE se

- $w.ID = 1$

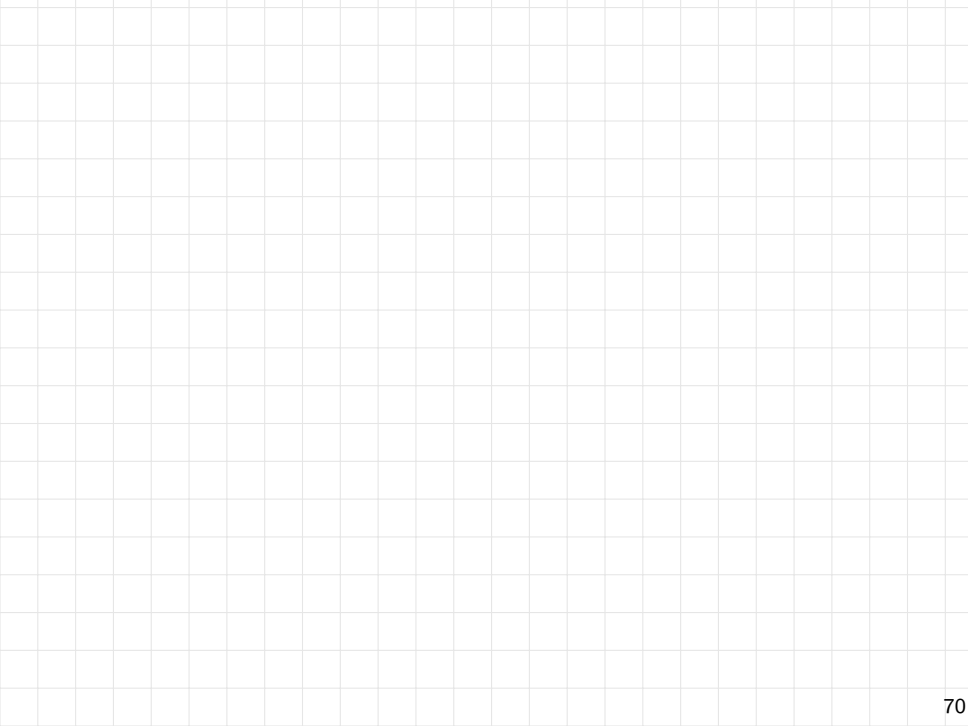
- $w.ON = 1$

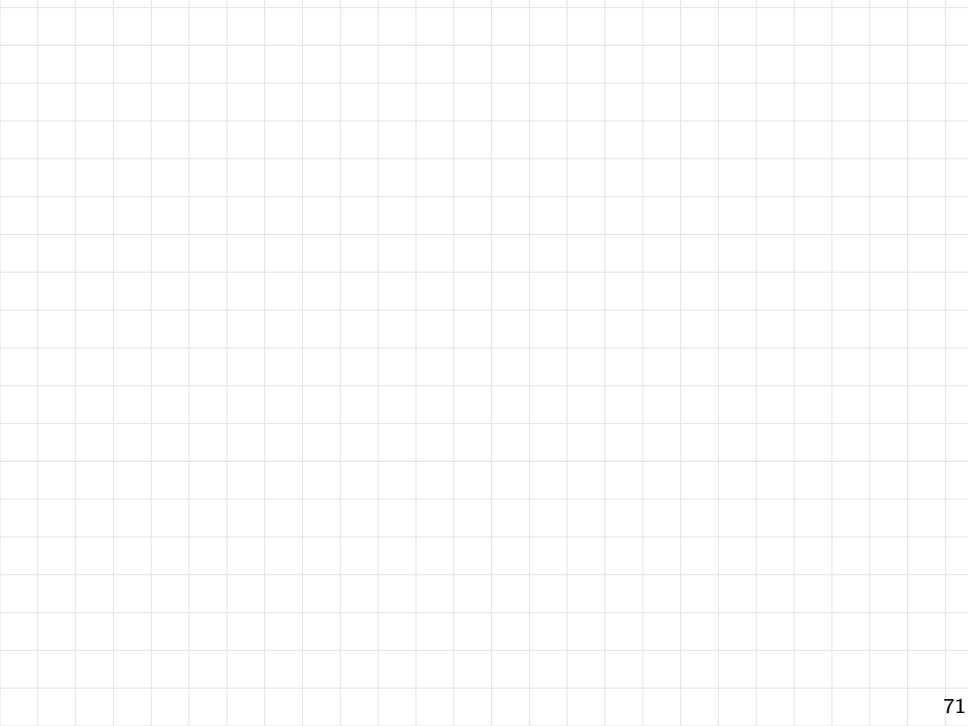
mentre si etichetta l'arco come ALTO se

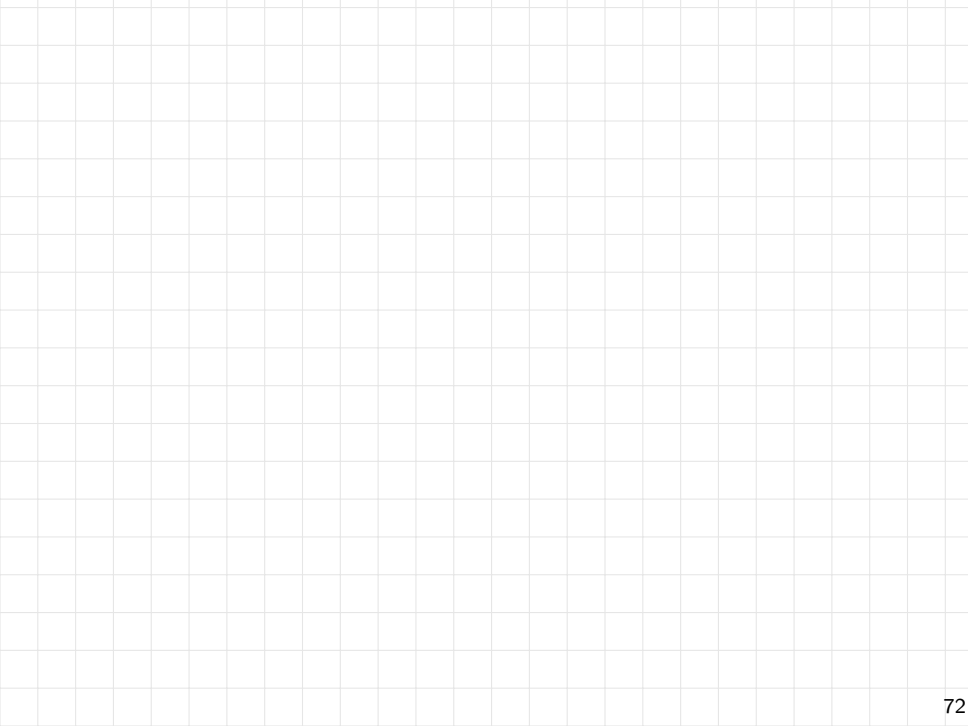
- $w.ID = 1$

- $w.ON = 0$

Oss. Queste modifiche alla DFS non ne
le complessità







Analisi di DFS e BFS

La proposizione nel lucido seguente (che non dimostriamo) stabilisce la complessità delle visite (BFS e DFS) su grafi diretti ed enuncia alcune loro proprietà e applicazioni importanti.

Sia $G = (V, E)$ un grafo diretto e sia $s \in V$. Si definisca

- $R_s = \text{reachable}(s)$, l'insieme di vertici raggiungibili da s , cioè i vertici v per i quali esiste un cammino diretto da s a v .
- $m_s = \sum_{v \in R_s} \text{outdegree}(v)$.

Analisi di DFS e BFS

Proposizione (derivata da Prop. 14.13, 14.15, 14.16, 14.17 di [GTG14])

Sia $\text{DFS}(G, s)$ che $\text{BFS}(G, s)$ hanno complessità $\Theta(m_s)$ e, al termine della loro esecuzione, si ha:

- 1 tutti i vertici di R_s e i loro archi uscenti sono stati visitati/etichettati;
- 2 i DISCOVERY EDGE formano uno spanning tree T di R_s radicato in s .

Inoltre permettono di risolvere i seguenti problemi:

- 1 ($\text{BFS} + \text{DFS}$) Determinare R_s (complessità $\Theta(m_s)$).
- 2 (DFS) Determinare se G è fortemente connesso (complessità $\Theta(|V| + |E|)$). *anche con BFS*
- 3 (DFS) Determinare un ciclo diretto in G , se esiste (complessità $\Theta(|V| + |E|)$).
- 4 (BFS) Trovare le distanze e i cammini minimi diretti da s a ogni $v \in R_s$ (complessità $\Theta(m_s)$).

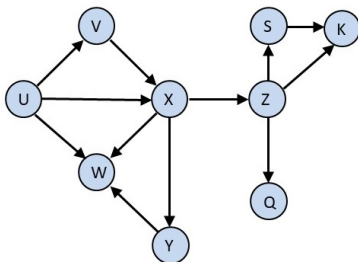
Esercizio

Sia $G = (V, E)$ un grafo diretto con n vertici e m archi. Dato un vertice $s \in V$ si definisca E_s l'insieme di archi uscenti da $\text{reachable}(s)$. È noto che esiste un ciclo diretto in E_s se e solo se $\text{DFS}(G, s)$ etichetta un arco come BACK EDGE. Usando questa proprietà, progettare e analizzare un algoritmo che dato $s \in V$ restituisce una lista L che contiene gli archi di un ciclo in E_s , se ne esiste uno, altrimenti è vuota.

Esercizio

Sia $G = (V, E)$ il grafo diretto di Instagram in cui i vertici rappresentano i profili e un arco (u, v) il fatto che il profilo u segue il profilo v . Progettare e analizzare un algoritmo che dato il profilo di un influencer $i \in V$ e un valore $k > 0$, restituisca il numero dei profili $u \in V$ tali che la distanza $d(u, i)$ del cammino minimo da u a i è $\leq k$. L'algoritmo deve avere complessità $O(n + m)$.

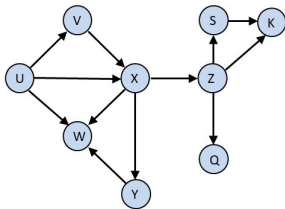
Grafi Diretti Aciclici



Un **Grafo Diretto Aciclico** (*Directed Acyclic Graph* o **DAG**) è un grafo diretto privo di cicli diretti.

Osservazione Ignorando l'orientamento degli archi, il grafo in figura ha dei cicli, ma non ha cicli diretti.

Ordinamento Topologico



Un **Ordinamento Topologico** per un DAG $G = (V, E)$ è un ordinamento dei vertici $v_1, v_2, \dots, v_n$ di V tale che per ogni arco $(v_i, v_j) \in E$ vale $i < j$.

L'ordinamento topologico del grafo in figura è: $u, v, x, y, w, z, q, s, k$.

APPLICAZIONE IMPORTANTE: Scheduling di task con dipendenze funzionali.

L'ordinamento topol. può essere usato per decidere l'ordine di esecuzione dei task

Un DAG può rappresentare un contesto di scheduling in cui si deve stabilire l'ordine (e a volte anche il tempo) in cui i task di un certo insieme devono essere eseguiti rispettando le dipendenze funzionali tra essi (ad.es. task i deve essere eseguito prima del task j)

$\Rightarrow \text{DAG} : G = (V, E) : V = \{\text{task}\}$

$E = \{\text{dipendenze funzionali}\}$

$(u, v) \in E \Rightarrow u$ deve essere eseguito prima di v

arco diretto

Algoritmo per Ordinamento Topologico

L'ordinamento topologico di un DAG $G = (V, E)$ può essere calcolato con un **algoritmo iterativo** che utilizza una **coda** L (inizialmente vuota) e **lista** S (inizializzata con i vertici con indegree 0).

In ciascuna iterazione, estrae un vertice v da S eseguendo le seguenti operazioni:

- aggiunge v in coda a L ;
- per ogni arco $e = (v, u)$ uscente da v , decrementa di 1 il conteggio degli archi entranti in u , e se diventa 0, inserisce u in S .

Alla fine: L contiene l'ordinamento topologico di V .

L'algoritmo assume di avere a disposizione un campo $u.indeg$, per ogni $u \in V$, che usa per tenere traccia del grado entrante di u .

Algoritmo per Ordinamento Topologico

Algoritmo TopologicalSort(G)

Input: DAG $G = (V, E)$ rappresentato con liste di adiacenza

Output: Coda L con l'ordinamento topologico dei vertici di G

inizializza $u.indeg = indegree(u) \quad \forall u \in V$; // tempo $\Theta(n+m)$

$L \leftarrow$ coda vuota;

$S \leftarrow$ lista di vertici $v \in V$ con $v.indeg = 0$;

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

$v \leftarrow S.first().getElement()$;

$S.remove(S.first())$;

$L.addLast(v)$;

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

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

$u.indeg \leftarrow u.indeg - 1$;

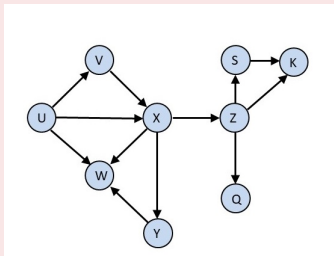
if ($u.indeg = 0$) **then** $S.addLast(u)$;

return L

$|V|=n$
 $|E|=m$

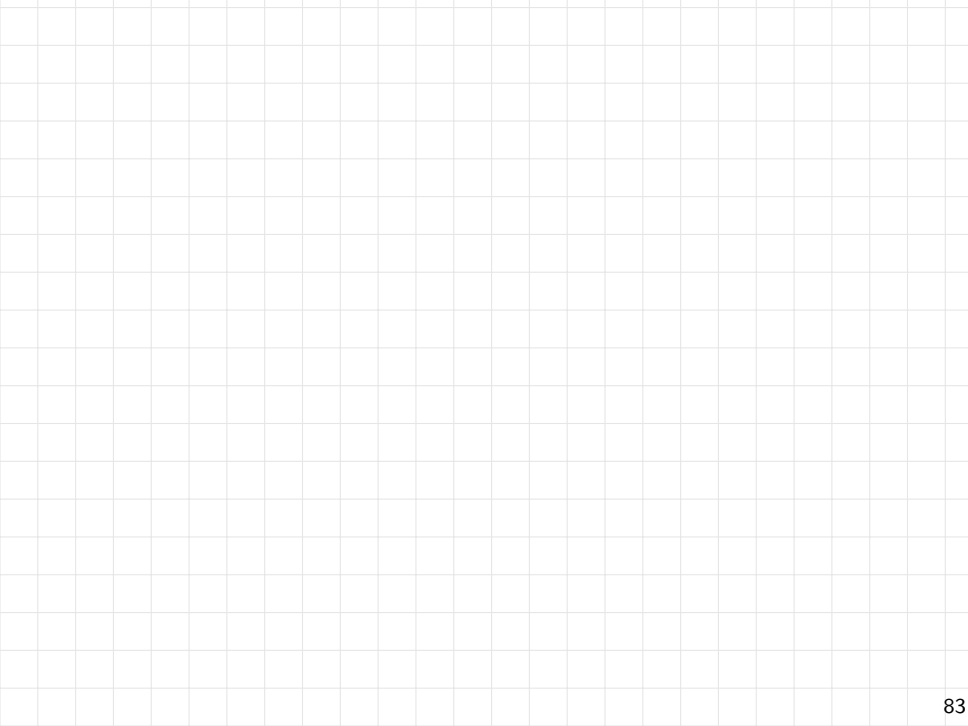
Esercizio

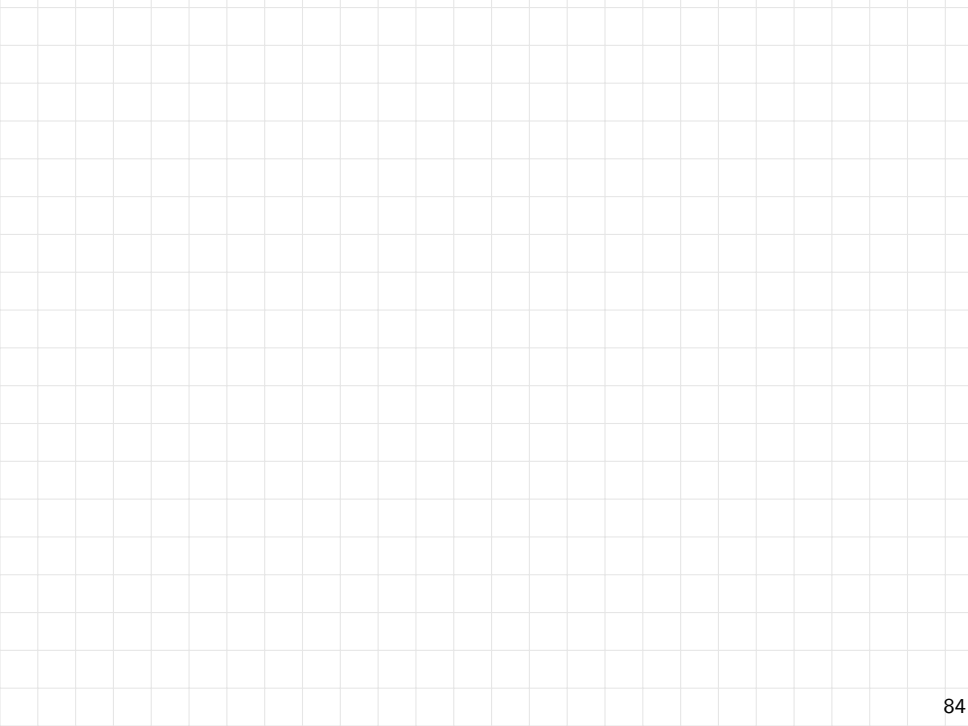
Mostrare l'esecuzione dell'algoritmo TopologicalSort sul grafo



Se cambiamo l'orientamento all'arco (z, k) , trasformandolo in (k, z) , il grafo non è più un DAG. Cosa succede se lo si dà in input a TopologicalSort?

S	L	indegree									
		u	v	x	y	w	s	z	k	Q	
u	$\emptyset$	0	1	2	1	3	1	1	2	1	
v	u	0	0	1	1	2	1	1	2	1	
x	uv	0	0	0	1	2	1	1	2	1	
yz	uvx	0	0	0	0	1	1	0	2	1	
zw	uvxy	0	0	0	0	0	1	0	2	1	
wsQ	uvxyz	0	0	0	0	0	0	0	1	0	
sQ	uvxyzw	0	0	0	0	0	0	0	1	0	
QK	uvxyzws	0	0	0	0	0	0	0	0	0	
k	uvxyzwsQ	0	0	0	0	0	0	0	0	0	
$\emptyset$	uvxyzwsQK	0	0	0	0	0	0	0	0	0	





Analisi di TopologicalSort

Proposizione

Sia $G = (V, E)$ un DAG con n vertici ed m archi.

TopologicalSort(G) restituisce correttamente un ordinamento topologico di V in tempo $\Theta(n + m)$.

DIMOSTRAZIONE : Sottiamo le prove di correttezza

Completezza :

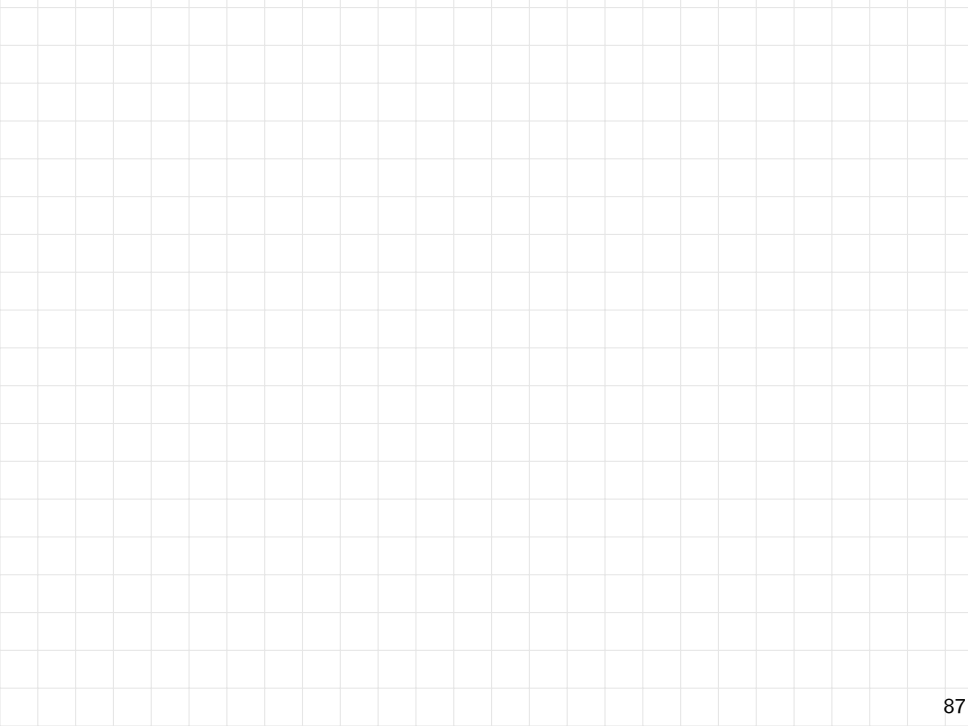
* Inizializzazione dei tempi u.indg : $\Theta(n+m)$
(vedi esercizio in una slide precedente)

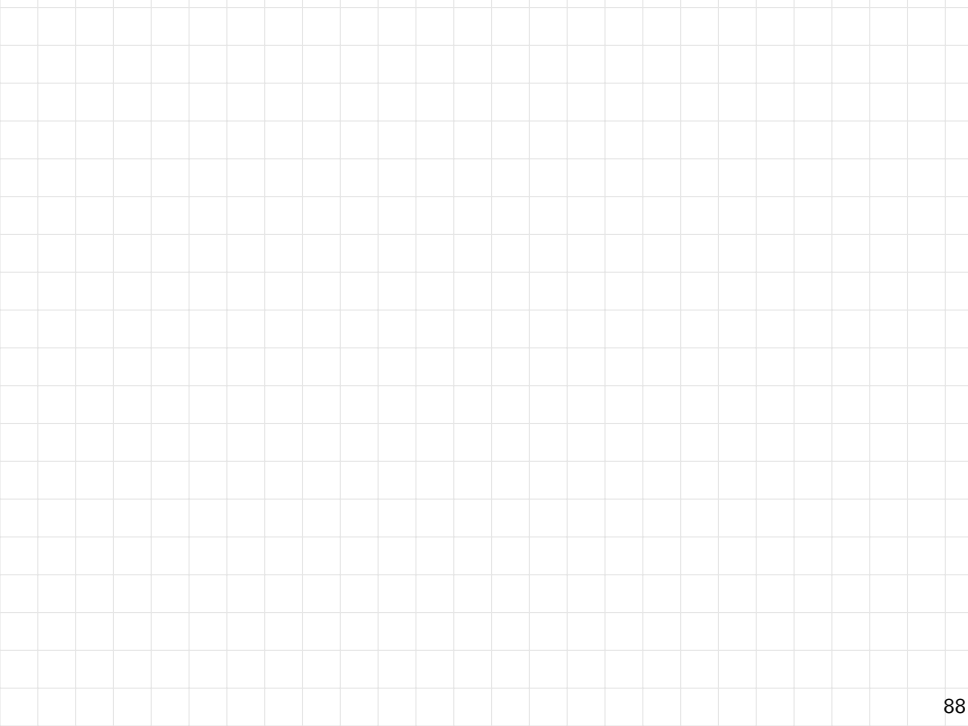
* Se l'elaborato è corretto, ogni vertice $v \in V$
sarà inserito ed estratto da S ESATTAMENTE

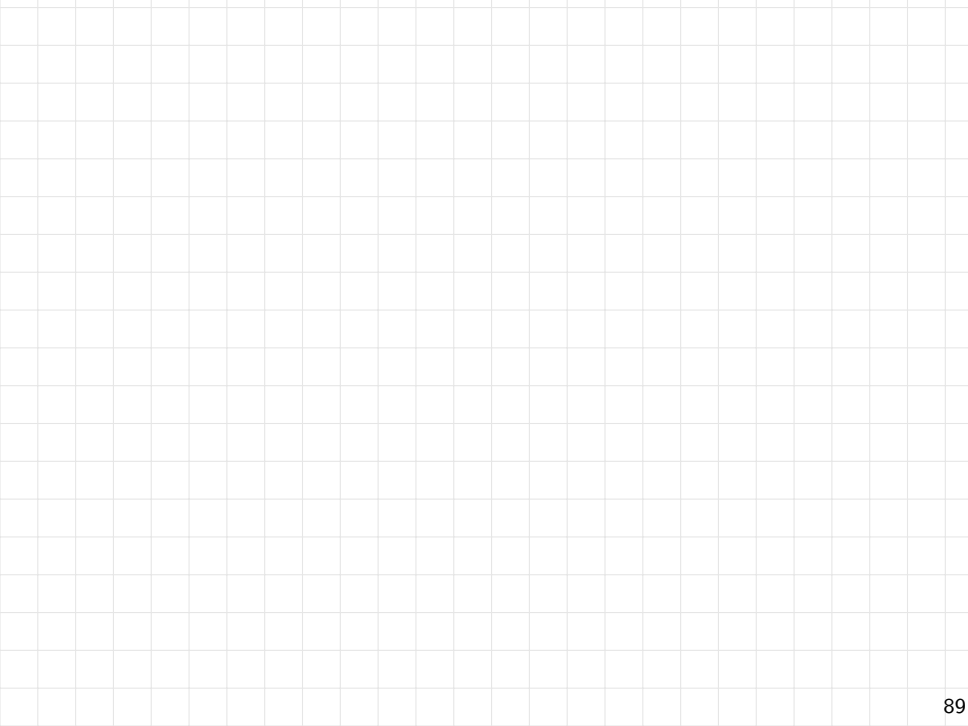
una volta $\Rightarrow$ n iterazioni del while
e l'iterazione che estrae un vertice v anche
a enumerare tutti gli archi uscenti da v
 $\Rightarrow$ complessità del while è

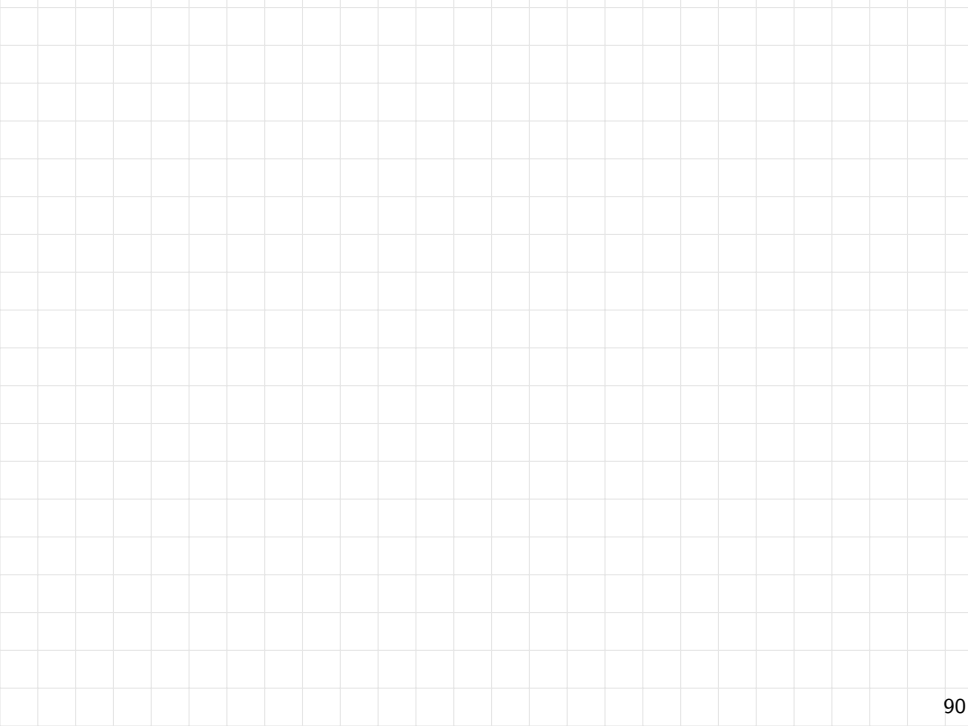
$$\Theta\left(n + \sum_{v \in V} \text{outdeg}(v)\right) = \Theta(n + m)$$

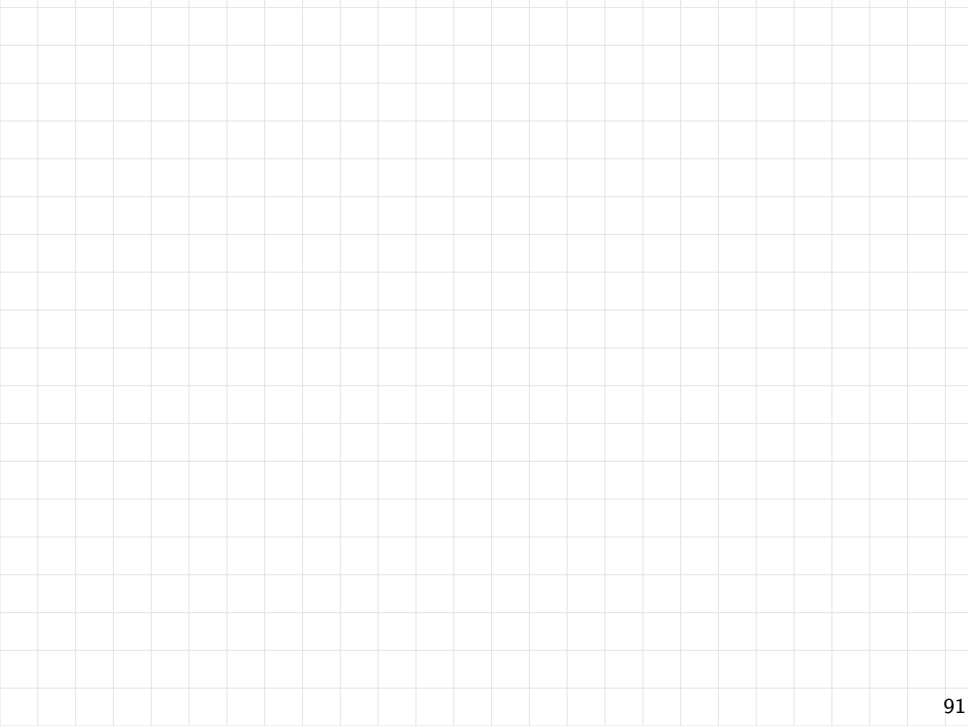
□











Esercizio

Modificare l'algoritmo `TopologicalSort( $G$ )` in modo che se il grafo di input non è un DAG restituisce `null`, altrimenti restituisce, come ora, l'ordinamento topologico dei suoi vertici.

Suggerimento: sfruttare l'intuizione ricavata dall'esercizio nel Lucido 81.

Esercizio

Dimostrare che se un grafo diretto $G = (V, E)$ ha un ciclo diretto, non può avere un ordinamento topologico.

Esercizio

Dimostrare che dato un grafo semplice non diretto $G = (V, E)$ è possibile dare un orientamento ai suoi archi in modo che esso diventi un DAG.

Esercizio

Sia G un DAG con n vertici ed m archi, i cui vertici rappresentano task e un arco (u, v) indica che v deve essere eseguito dopo u . Per ogni task v , un campo $v.length$ contiene la durata del task. Progettare un algoritmo che in tempo $O(n + m)$ assegna a ciascun task v un tempo di inizio compatibile con le informazioni contenute in G , memorizzandolo in un campo $v.start$.

Suggerimento: usare l'algoritmo $TopologicalSort(G)$ come primitiva.

Riepilogo sui Grafi

- Definizioni e terminologia sui grafi.
- Concetti fondamentali: cammino, ciclo, grafo connesso, subgraph, spanning subgraph, alberi radicati/liberi, spanning tree/forest.
- Relazione tra la somma dei degree e il numero di archi in un grafo.
- Relazioni tra numero di vertici e numero di archi in alberi, foreste e grafi connessi.
- Breadth-First Search:
 - Pseudocodice.
 - Proprietà e complessità.
 - Problemi computazionali risolvibili tramite BFS.
- Depth-First Search:
 - Pseudocodice.
 - Proprietà e complessità.
 - Problemi computazionali risolvibili tramite DFS.

- Grafi pesati
 - Cammini minimi su grafi pesati e loro proprietà.
 - Problema SSSP.
 - Algoritmo di Dijkstra: pseudocodice, correttezza e complessità.
- Grafi semplici e diretti.
 - Terminologia.
 - Nozione di grafo fortemente/debolmente connesso.
 - BFS e DFS su grafi diretti e loro applicazioni.
 - Directed Acyclic Graph (DAG).
 - Ordinamento Topologico di un DAG: definizione, applicazione, algoritmo e analisi.