A taste of di-graphs

corso di laurea in **Matematica**

Informatica Generale, Lezione **24**

Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

Connessioni & visite

Nei grafi orientati, la nozione di **raggiungibilità non è simmetrica**. Ciò implica, ad es., che una visita radicata in un certo nodo s non scopre tutti i nodi "attaccati" a s, ma **solo quelli raggiungibili da s**.

Nei grafi diretti sono rilevanti diverse nozioni di **connessione**. Un grafo $G = (V, E)$ è:

- **semi-connesso** se per ogni $u, v \in V$, $u \rightsquigarrow v$ **oppure** $v \rightsquigarrow u$.
- **fortemente connesso** se per ogni $u, v \in V$, $u \rightsquigarrow v$ **e** $v \rightsquigarrow u$.

Un grafo è fortemente connesso **se è un unico grande "ciclo"** (con eventuali corde). Come determinare se G è fortemente connesso?

È sufficiente scegliere un nodo s a caso, fare una visita e vedere se tutti i nodi sono raggiungibili. Dopodiché è necessario vedere se s sia esso stesso raggiungibile da tutti. Come fare?

Si fa una visita sempre radicata in s sul grafo trasposto (o rovesciato) G^T in cui ho invertito l'orientamento di tutti gli archi.

◗ **Esercizio**: calcolare G^T in $\Theta(m + n)$ se G è rappresentato con liste di adiacenza. Farlo anche con matrici di adiacenza e liste di archi.

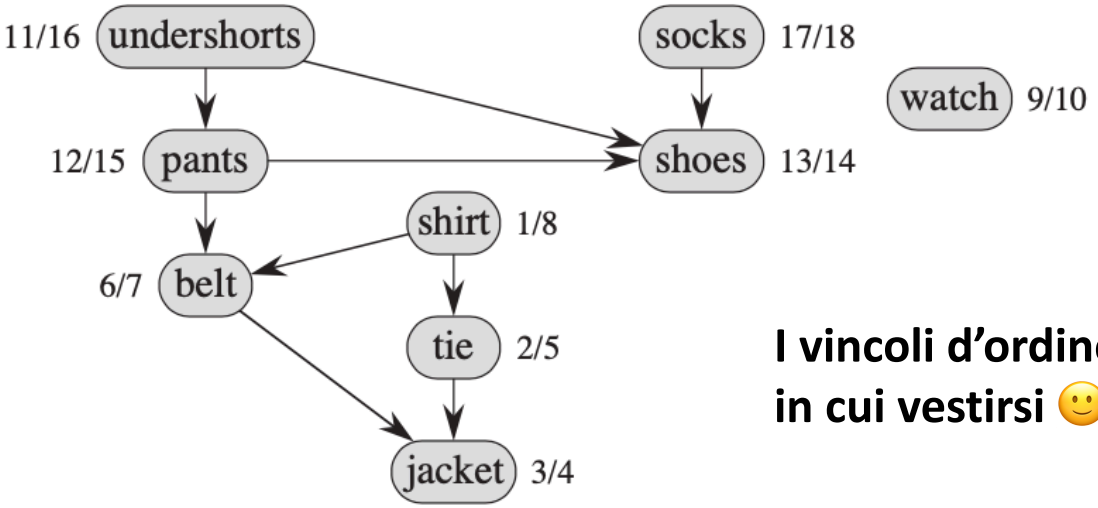
Grafi Diretti Aciclici (DAGs)

Una famiglia molto importante di grafi diretti sono i **grafi diretti aciclici** (noti anche come **DAG**, **D**irected **A**cyclic **G**raphs).

I DAG sono una "specie di albero" in cui però i cammini possono "riconfluire" e con (eventualmente) più radici.

Una evidente situazione in cui emergono, è una **sequenza di azioni** che sono **parzialmente ordinate** (ad esempio: **un processo produttivo**, potrebbe essere utile per **determinare il tempo** di **completamento**, rispettando **i vincoli di precedenza**).

Ecco un simpatico esempio dal Cormen:



I vincoli d'ordine in cui vestirsi 🙂

Grafi Diretti: altre osservazioni

Anche problemi relativamente semplici, nei diretti si **complicano leggermente**: ad esempio **determinare** se un grafo **contiene un ciclo**.

Nei grafi **non diretti**, una qualsiasi visita (BFS o DFS) permette di determinare l'esistenza di un ciclo **non appena si scopre un arco non dell'albero di visita** (sia di attraversamento nella BFS che all'indietro nella DFS). Questo si individua quando si incontra un nodo già marcato (o con distanza già calcolata, o con tempo di entrata già calcolato etc.)

Ciò **non è vero nei grafi diretti**. Ad esempio, la versione non orientata di un DAG può essere ciclica (per esempio il DAG della slide precedente)

⏵ **Esercizio**: Determinare la condizione per cui un arco all'indietro in una DFS garantisca l'esistenza di un ciclo.

DAG: Topological Sort

corso di laurea in **Matematica**

Informatica Generale, Lezione **24(a)**

Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

Ordine Topologico sui Nodi

Un **ordinamento totale** dei nodi di un grafo diretto G è un **ordine topologico** se **rispetta la relazione di discendenza**.

Formalmente, dato un grafo orientato $G = (V, E)$, diciamo che $(V, <)$ è un ordinamento topologico se $u \rightarrow v$ implica $u < v$.

Lemma: Se **esiste un ordinamento topologico** dei nodi di un grafo orientato G, **allora G è un DAG**.

Dim.: Assumiamo che G ammetta un ordinamento topologico, ma contenga un ciclo C.

Ordiniamo i nodi di G in sequenza $v_1 < v_2 < \ldots < v_n$ secondo l'ordine topologico e sia v_i il nodo di indice minimo in C.

Sia v_j il precedente di v_i in C. Questo implica che l'arco $v_j \rightarrow v_i$ è in C, ma $j > i$ (per la scelta di v_i nodo di indice minimo). Assurdo. □

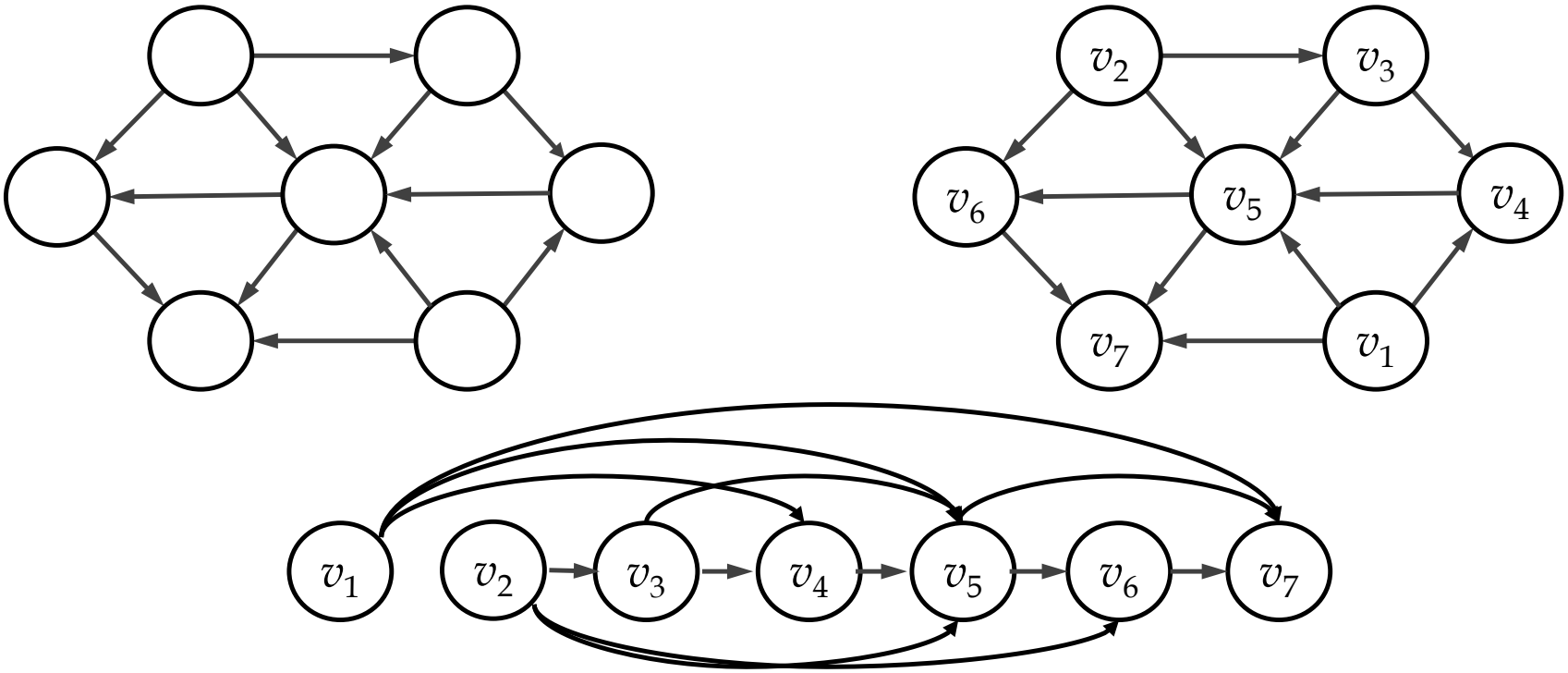
➧ **Esercizio**: dimostrate che un DAG con n nodi ha fino a $\binom{n}{2}$ archi.

Ordine Topologico: Esempio

Può non essere evidente che un grafo sia un DAG (fig. a **sinistra**).

Scriviamo i nodi indicizzati per evidenziare un possibile ordine topologico (fig. a **destra**).

Evidenziamo l'ordinamento topologico, disegnando il grafo "linearmente" (fig. **sotto**)



Proprietà dell'Ordine Topologico

Lemma: Un DAG G ha almeno un nodo senza archi entranti.

Dim.: Sempre per assurdo, assumiamo che non sia così. Prendiamo un nodo qualsiasi v_0: ha almeno un arco entrante da v_1. Anche v_1 ha un arco entrante da $v_2 \neq v_0$ (altrimenti avrei creato un ciclo).

Possiamo continuare questa costruzione con $v_{i+1} \notin \{v_1, \ldots, v_i\}$ per tutti i nodi $V = \{v_1, \ldots, v_n\}$ con v_n ancora con un arco entrante non già considerato. Ma per il principio dei buchi di piccionaia, v_{n+1} deve necessariamente essere in $\{v_1, \ldots, v_n\}$, creando un ciclo. □

Corollario: Se G è un DAG, allora ha un ordinamento topologico.

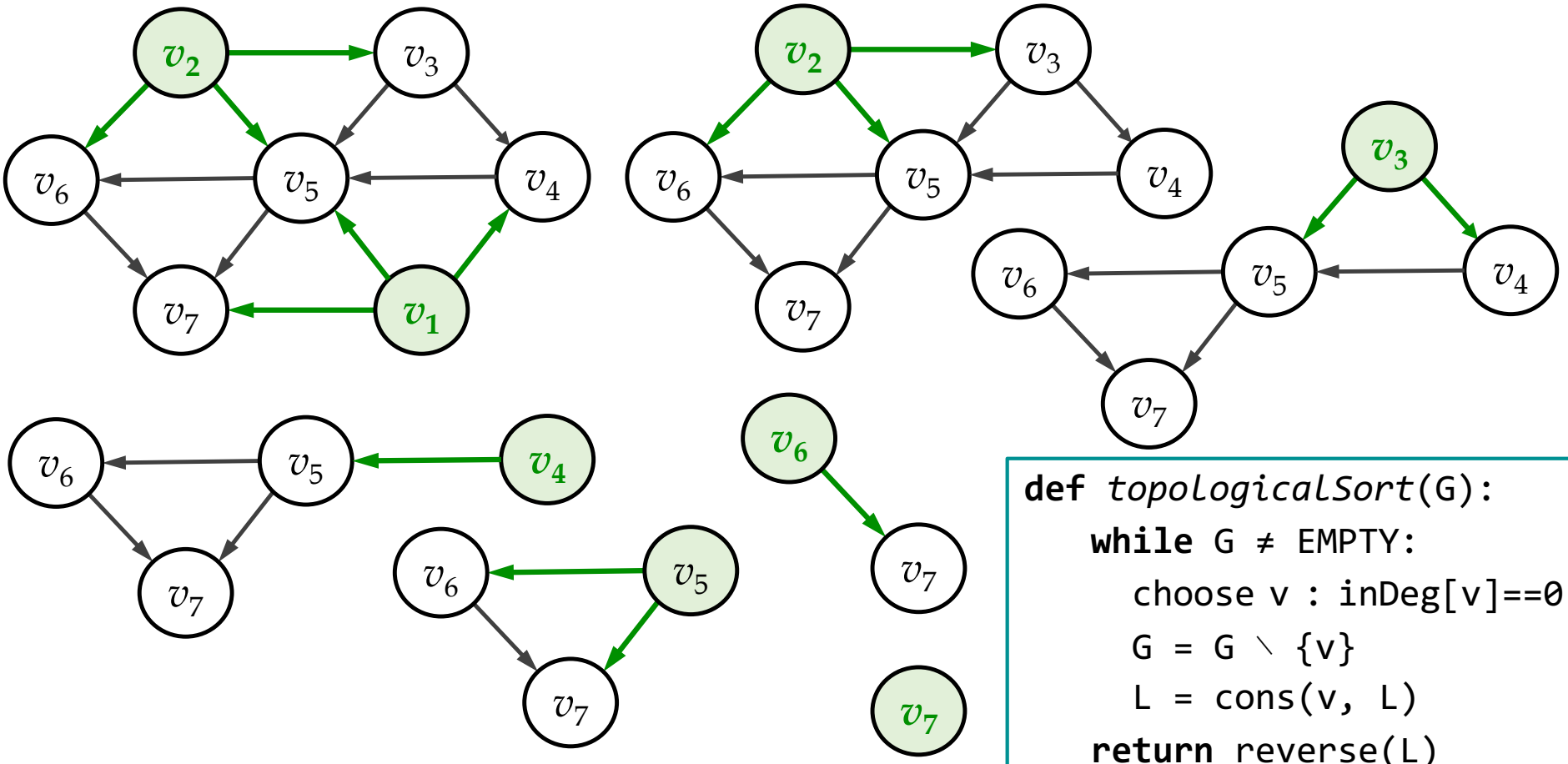
Dim.: In G esiste un **insieme non vuoto** di **nodi senza archi entranti** per il Lemma precedente. **Ne scelgo uno**, lo chiamo v_1, come elemento minimo di $<$.

Rimuovo v_1 e il sottografo indotto $G' = (V \setminus \{v_1\}, E')$ è ancora un DAG. Scelgo v_2 tra i nodi senza archi entranti e **itero il procedimento** fino a esaurimento di tutti i nodi. L'ordine di scelta definisce un ordine topologico. □

Calcolare un Ordine Topologico

Il corollario precedente **definisce un semplice algoritmo** per **calcolare l'ordine topologico** di un DAG.

Notate che l'**ordinamento non è unico** (qui abbiamo la scelta tra v_1 e v_2 per scegliere il primo nodo, ma spesso ci sono più scelte).




```
def topologicalSort(G):
    while G ≠ EMPTY:
        choose v : inDeg[v]==0
        G = G \ {v}
        L = cons(v, L)
    return reverse(L)
```

Algoritmo per calcolare OT

Benchè l'algoritmo sia semplice e intuitivo, è interessante vedere come si possa implementarlo in tempo $\Theta(m + n)$.

In tempo $\Theta(m + n)$ possiamo calcolare **tutti i gradi entranti** in un vettore $inDeg[u]$ indicizzato sui nodi.

In tempo $\Theta(n)$ possiamo caricare una lista (o coda, o pila, **l'ordine non è rilevante** qui) di nodi con grado entrante 0.

Si esaminano gli archi del nodo scelto u, diminuisco di 1 $inDeg[v]$ per ogni arco $u \rightarrow v$. Se $inDeg[u]$ diventa 0, aggiungo v alla coda.

```
def topologicalSort(G):
    inDeg, Z, TS = initDeg(G), newQ(), NULL
    forall v ∈ V:
        if inDeg[v]==0: enqueue(Z, v)
    while not(isEmpty(Z)):
        u, TS = dequeue(Z), cons(TS, u)
        forall v ∈ adj(u):
            inDeg[v]=indeg[v]-1
            if inDeg[v]==0: enqueue(Z, v)
    return reverse(TS)
```

$\Theta(n)$

$\Theta(m+n)$

```
def initDeg(G):
    inDeg = allZero(n)
    forall u ∈ V:
        forall v ∈ adj(u):
            inDeg[v] = inDeg[v] + 1
    return inDeg
```

$\Theta(m+n)$

TS con una sola DFS

Benché l'algoritmo visto sia ottimo, essendo lineare nelle dimensioni del grafo, si può calcolare il **topological sort** con **un'unica DFS**.

Tale algoritmo è molto **istruttivo** per capire/ricordare le proprietà delle DFS su grafo diretto.

È sufficiente osservare che i nodi, **ordinanti rovesciati** rispetto ai **tempi di uscita** ***out***[v] di una DFS sono ordinati topologicamente.

Infatti, se ***out***[u] < ***out***[v] ci sono 2 possibilità (Teor. delle Parentesi):

- v **è un discendente di** u, quindi **deve essere** $u < v$
- v **è stato scoperto dopo** u (ma né u è discendente di v, né u è discendente di v) quindi **l'ordine è indifferente**.

```
def tsDfs(G, u, t, in, out, L):
    t = t + 1; in[u] = t
    forall v ∈ adj(u):
        if in[v] > 0:
            p[v] = u
            t, L = dfs(G, v, VIS, t, in, out)
    out[u], L = t+1, cons(u, L)
    return t+1, L
```

Cammini minimi su DAG

Se conosco le distanze dalla sorgente s a tutti i precedenti di u, allora posso calcolare $\delta(s, u)$ semplicemente **analizzando gli archi entranti**.

Morale: si possono calcolare i cammini minimi in un DAG **rilassando i nodi una sola volta**, nell'ordine in cui compaiono **nell'ordine topologico**.

La complessità in questo caso è quindi lineare, cioè $\Theta(n + m)$.

È interessante che, in questo caso, lo stesso algoritmo **può calcolare** il **costo massimo** (che corrisponde ai cammini critici in un processo)

```
def dagShortestPath(G, w, s):
    L = topologicalSort(G)
    d, p = allocaV(n), allocaV(n)
    initSingleSource(G, s, d, p)
    forall u ∈ L:
        forall v ∈ adj(u):
            relax(u, v, w)
```