Tecnica Greedy

Introduzione

Consiste nel calcolo della soluzione ottima attraverso sequenze di scelte localmente ottime.

✅ Facile da scrivere ed efficiente

✅ Scelta del locale ottimo non dipende dalle scelte successive

✅ Scelta riduce sottoproblemi da risolvere

Algoritmi greedy

- Soluzione ottima
- Top-Down
- Pochi sottoproblemi da risolvere
- + efficiente e semplice
- - applicabile

Programmazione Dinamica

- Soluzione e valore ottimo
- Bottom-Up
- Tanti sottoproblemi da risolvere
- - efficiente e + complicato
- + applicabile

Sistema di indipendenza

Se nella coppia (S,F) dove:
- S è il numero finito {
$s_1, ..., s_n$ } di elementi
- F è sottoinsieme dell'insieme delle parti di S
Vale la proprietà:
Preso
$A, B \in F$, allora qualsiasi $B \subseteq A$ appartiene a F

Come capire se un problema accetta un algoritmo greedy?

Matroide

💡 Struttura combinatoria a cui è associato un algoritmo greedy

Sistema di indipendenza dove per qualsiasi $A, B \in F$ tale che $|B| > |A|$ allora esiste almeno un elemento $b \in B - A$ tale che $\{b\} \cup A \in F$

Matroide Grafico

💡 Dato un grafo G=(V,E) non orientato e connesso, $M_g = (S, F)$ con S insieme E degli archi e F tutti i sottoinsiemi di S aciclici è il matroide grafico di G

M_G è un matroide:

1. $A \in F,\ B \subseteq A \Rightarrow B \in F$
2. $\forall A,\ B \in F\ t.c. |B| = |A| + 1 \Rightarrow \exists\, b \in B - A\ \ t.c.\ \{b\} \cup A \in F$

   Questa viene chiamata proprietà dallo scambio, dimostriamola:

   Siano $A, B \in F$ tali che $|B| = |A| + 1$ dove:

   $G_A = (V, A) \rightarrow$ foresta di $|V| - |A|$ alberi

   $G_B = (V, B) \rightarrow$ foresta di $|V| - |A|$ alberi
   Quest'ultimo avrà un albero in meno rispetto al primo.

   $\Rightarrow$ in G_B esiste un arco (u, v) che connette due vertici u e v che in G_A stanno in due alberi diversi

Qualche definizione

- Estensione: $s \in S$ è l'estensione di $A \in F$ se $A\ \cup \{s\} \in F$
- Massimale: $A \in F$ è massimale se non esistono estensioni
- Matroide pesato: al matroide viene associata una funzione peso $W : S \rightarrow R^+$

Teorema di Rado

💡 La coppia (S, F) è un matroide se e solo se per ogni funzione peso W, l'algoritmo standard greedy fornisce la soluzione ottima (sottoinsieme indipendente di peso massimo)

MST (Minimum Spanning Tree)

Introduzione

INPUT

Grafo connesso non orientato pesato $G = (V, E)$ con $W : E \to R^+$ tale che $W(u, v)$ è il peso dell'arco (u, v)

OUTPUT

$T \subseteq E$ aciclico tale che:

1. $\forall\ v \in V,\ \exists\ (u, v) \in T$
2. $W(T) = \sum_{(u,v)\ \in\ T} W(u, v)$ è minimo

$G_T = (V, T) \to$ MST

🚨 In poche parole…
Devo avere ogni nodo collegato almeno da un arco, in modo tale che la somma di tutti questi archi sia la più piccola possibile

Qualche Definizione

- Taglio: Partizione di V in due insiemi V' e V-V'
- Arco attraversa taglio: arco $(u, v) \in E \;\; t.c. \, u \in V' \;\wedge\; v \in V' - V$
- Taglio che rispetta l'insieme: un taglio rispetta un insieme $A \subseteq E$ se nessun arco di A attraversa il taglio
- Arco leggero: arco di peso minimo che attraversa il taglio

Teorema dell'arco sicuro

Dati:

- un grafo connesso non orientato e pesato $G = (V, E)$
- un sottoinsieme A dell'insieme T di archi di un MST
- un qualsiasi taglio $(S, V - S)$ che rispetti A
- un arco leggero (u, v) del taglio

Allora l'arco leggero (u, v) è sicuro per A, ovvero $A \cup \{(u, v)\} \subseteq T$

NOTA! (u, v) è un arco sicuro per A se quell'arco appartiene al MST

Dimostrazione

IPOTESI: Esiste almeno un MST $T \subseteq E \;\; t.c \;\; A \subseteq T$

TESI: trovare MST $T' \subseteq E \;\; t.c \;\; A \cup \{(u, v)\} \subseteq T$

Visto che $(S, V - S)$ rispetta A e (u, v) attraversa il taglio, allora $(u, v) \notin A$.

Abbiamo quindi 2 casi:

Se $(u, v) \in T$

Allora $A \cup \{(u, v)\} \subseteq T$ il quale è una MST

Se $(u, v) \notin T$

Dal momento che T è connesso, allora esisterà un cammino p che va da u a v. Visto che (u, v) attraversa il taglio, allora significa che si trovano da due parti opposte rispetto a quest'ultimo. Esiste allora almeno un arco (x, y) di p che attraversa il taglio.

Sia $T' = (T \setminus \{(x, y\} \cup \{(u, v)\})$:

> Sappiamo che $A \subseteq T$ e che $(x, y) \notin A$ visto che attraversa il taglio, allora $A \subseteq T \setminus \{(x, y\}$
>
> A maggior ragione $A \subseteq (T \setminus \{(x, y)\}) \cup \{(u, v)\} = T' \;\Rightarrow A \cup \{(u, v)\} \subseteq T'$.

Verificando il peso di T':

$$w(T') = w(T) - w(x,y) + w(u,v)$$

Dal momento che (u,v) è un arco leggero del taglio attraversato anche da (x,y) allora $w(x,y) \geq w(u,v) \Rightarrow \; w(T') \leq w(T)$

Essendo T un MST, allora lo è anche T', il quale contiene $A \cup \{(u,v)\}$.

Corollario

$A \subseteq T$ è tale che $G_A = (V, A)$ è una foresta con $|V| - |A|$ alberi.

Sia $C = (V_C, A_C)$, con $V_C \subseteq V \; e \; A_C \subseteq A$, una componente connessa di G_A.

$\Rightarrow (V_C, V - V_C)$ è sicuramente un taglio che rispetta A

$\Rightarrow$ un arco leggero di $(V_C, V - V_C)$ è un arco sicuro per A

🚨 In poche parole…
Per trovare un nuovo arco sicuro da aggiungere ad A:
- Considero una delle componenti C della foresta
- Trovo arco leggero che collega un vertice in C con uno non in C

Algo Generico

1. Inizializza un insieme A vuoto
2. Aggiunge ad ogni passo un arco (u,v) tale che $A \cup \{(u,v)\}$ è un sottoinsieme dell'insieme T degli archi di MST
3. Algoritmo termina quando $A = T$, ovvero $G_A = (V, T) \Rightarrow MST$

Algoritmo GENERIC-MST

GENERIC-MST (G,W)

$A \leftarrow \emptyset$

WHILE $|V| - |A| > 1$

trova arco (u,v) sicuro per A

$A = A \cup \{(u,v)\}$

RETURN A

Cosa fa?

1. A rimane aciclico durante le iterazioni
2. $G_A = (V, A)$ ad ogni iterazione è una foresta di $|V| - |A|$ alberi

3. All’inizio, G_A contiene $|V|$ alberi (singoli vertici)
4. Ogni iterazione riduce di 1 il numero di alberi e l’arco sicuro collega sempre componenti distinte di G_A
5. Quando arriva ad un solo albero l’algoritmo termina (ovvero tutti i vertici sono collegati)
6. Il numero di iterazioni è parti a $|V| - 1$

Algoritmo di Kruskal

Algoritmo per trovare MST, tramite l’ordinamento degli archi in ordine crescente di costo e successivamente analizzandoli singolarmente, inserendo l'arco nella soluzione se non forma cicli con gli archi precedentemente selezionati (ovvero connette due componenti diverse di G_A.

Algoritmo definitivo

KRUSKAL-MST (G=(V,E), W)

$A \leftarrow \emptyset$

$E \leftarrow \langle e_1, ..., e_n \rangle$ ordinati per peso non decrescente

FOREACH $v \in V$

MAKE_SET (v)

FOR i from 1 to n

$(u, v) \leftarrow e_i$

IF FIND_SET (u) $\neq$ FIND_SET(v)

$A = \{(u, v)\} \cup A$

UNION (u,v)

RETURN A

Tempo di calcolo

Sapendo che:

- $|E| \geq |V| - 1$
- $\alpha \leq log|V| \rightarrow \alpha \leq log|E|$

L’ordinamento ha tempo $O(|E|log|E|)$, FOREACH invece $O(|V|)$ e infine il FOR complessivamente è $O(|E|\alpha)$. Sommando troviamo:

$$O(|E|log|E| + (|V| + |E|)\alpha) \rightarrow O(|E|log|E|)$$

Algoritmo di Prim

Funzionamento di base

1. Sceglie vertice arbitrario r (componente C all’inizio composta quindi solo da vertice r)

2. Trova l'arco di peso minimo che connette r ad un altro vertice v (entra così anche v in C)
3. Trovo arco di peso minimo che connette un vertice in C ad un vertice v non in C (anche questo entra in C)
4. Ripeto il passo 3
5. Termina quando C comprende tutti i vertici del grafo e quindi coincide con il MST

Proprietà dell'algoritmo

Ad ogni passo:

1. Il sottoinsieme A degli archi di MST aggiunti fanno parte della componente C. La foresta è composta quindi da:
   - $C = (V_C, A)$
   - $|V - V_C|$ componenti di vertici singoli (non ancora inseriti)
2. Il taglio $(V_C, V - V_C)$ rispetta l'insieme A
3. L'arco sicuro è l'arco leggero (di peso minimo) che connette un vertice in C con uno non in C.

Elementi utili

Coda Q

Coda di min-priority che contiene tutti i vertici che non appartengono a C (quindi all'inizio tutti), permettendo di estrarre un vertice v tale che (u, v) è l'arco leggero (peso minimo) che collega un vertice $u \in C$ con un vertice $v \notin C$.

Campi dei vertici

Ad ogni vertice v sono associati due campi:

- **v.key** → minimo valore del peso degli archi (u, v) incidenti in v tale che $u \in C$.
- $v.\pi$ → indica un vertice u tale che (u, v) è l'arco di peso minimo di v.key

🚨 All'inizio $v.key = \infty$ e $v.\pi = NIL$ per tutti i vertici tranne per il primo, il quale scelto in modo arbitrario e ha $r.key = 0$.

Ad ogni passo…

1. Viene estratto da Q il vertice u con il minor valore del campo key:
   - l'arco $(u.\pi, u)$ è un nuovo arco di MST
   - u è un vertice che si aggiunge alla componente C
2. Per ogni vertice v adiacente a u, se v è in Q e $v.key > W(u, v)$, vengono aggiornati:
   - $v.key$ al valore $W(u, v)$
   - $v.\pi$ al valore u

3. Algo termina quando Q vuota

Algoritmo

PRIM-MST (G,W,r)

FOREACH $v \in V$

$v.key \leftarrow \infty$

$v.\pi \leftarrow NIL$

$r.key \leftarrow 0$

Aggiungi tutti i vertici di V alla coda Q

WHILE $Q \neq \emptyset$

$u \leftarrow$ *estrai vertice da Q*

FOREACH $v \in adj(u)$

IF $v \in Q$ **AND** $W(u,v) < v.key$

$v.key \leftarrow W(u,v)$

$v.\pi \leftarrow u$

Tempo di calcolo

L'inizializzazione dei valori e l'aggiunta dei vertici a Q hanno tempo lineare $O(|V|)$, anche WHILE $O(|V|)$, l'estrazione del vertice da Q $O(log|V|)$, l'ultimo FOREACH invece $O(|E|)$ e infine l'assegnazione del valore da $W(u,v)$ $O(log|V|)$. Possiamo quindi calcolare il tempo totale:

$$O(|V|) + O(|V|log|V|) + O(|E|log|V|) \rightarrow O(|E|log|E|)$$

Algoritmo di Dijkstra

Introduzione

INPUT

Dato un grafo $G = (V, E, W)$ orientato e pesato:

- $W : E \to R^+$ tale che $W(i, j) = w_{ij}$ = peso dell'arco (i, j)
- Un vertice $s \in V$ chiamato vertice sorgente

OUTPUT

Per ogni $v \in V$ diverso da s, trovare il cammino di peso minimo che inizia in s e termina in v.

Sottostruttura ottima del cammino minimo

Se il cammino $P = \langle v_1, ..., v_{k-1}, v_k \rangle$ è minimo, allora sono minimo anche tutti i sottocammini:

$P_{ij} = \langle v_i, ..., v_j \rangle$ per $1 \leq i < j \leq k$.

Scomposizione del peso

La scomposizione del peso del cammino minimo $\delta(s, v)$ dalla sorgente s al vertice v:

$$\delta(s, v) = \delta(s, u) + W(u, v)$$

Dove:

- u è il predecessore di v nel cammino minimo da s a v.

- $\delta(s, v)$ è il peso del cammino minimo da s a v
- $\delta(s, u)$ è il peso del cammino minimo da s a u
- $W(u, v)$ è il peso dell'arco (u, v)

Limite superiore

Dato un qualsiasi arco (u, v) si ha:

$$\delta(s, v) \leq \delta(s, u) + W(u, v)$$

Dove:

- $\delta(s, v)$ è il peso del cammino minimo da s a v
- $\delta(s, u)$ è il peso del cammino minimo da s a u
- $W(u, v)$ è il peso dell'arco (u, v), questo arco può non appartenere al cammino minimo.

Tecnica del rilassamento

Aggiungo ad ogni vertice v due attributi:

- $v.d$ → limite superiore per $\delta(s, v)$
- $v.\pi$ → vertice u tale che $(u, v) \in E$

Questa tecnica viene eseguita per ogni arco (u, v) del grafo una sola volta.

Prima dell'esecuzione

- $v.d = \infty$ per ogni vertice v diverso dalla sorgente s
- $v.\pi = NIL$ per ogni vertice v
- $s.d = 0$ per la sorgente s

Durante l'esecuzione

Se $v.d > u.d + W(u, v)$ allora:

1. $v.d = u.d + W(u, v)$
2. $v.\pi = u$

Dopo l'esecuzione

- $v.d = \delta(s, v)$ ovvero il peso del cammino minimo da s a v
- $v.\pi = u$ ovvero il predecessore di v nel cammino minimo da s a v

Algoritmo RELAX

RELAX (u ,v ,W)

IF $v.d > u.d + W(u, v)$

$v.d \leftarrow u.d + W(u, v)$

$v.\pi \leftarrow u$

Main topic

Coda Q

Troviamo una coda Q di min-priority che contiene tutti i vertici che non hanno raggiunto il valore $\delta(s, v)$ nel proprio campo $v.d$, quindi ad ogni passo:

1. Viene estratto un vertice u da Q quando $u.d = \delta(s, u)$
2. Viene eseguito il rilassamento di ogni arco (u, v) uscente da u.
3. Dopo che Q si svuota l'algoritmo termina, e tramite il valore del campo dei predecessori si può ricostruire il cammino minimo dalla sorgente s ad un determinato vertice v.

Algoritmo

DIJKSTRA (G, W, s)

  Inizialize-Single-Source (G,s)

  $S \leftarrow \emptyset$

  Aggiungi tutti i vertici di V alla coda Q

  WHILE $Q \neq \emptyset$

    $u \leftarrow$ *estrai vertice da Q*

    $S \leftarrow S \cup \{u\}$

    FOREACH $v \in adj(u)$

      RELAX (u, v, W)

RELAX (u ,v ,W)

  IF $v.d > u.d + W(u, v)$

    $v.d \leftarrow u.d + W(u, v)$

    $v.\pi \leftarrow u$

Inizialize-Single-Source (G ,s)

  FOREACH $v \in V$

    $v.d \leftarrow \infty$

    $v.\pi \leftarrow NIL$

  $s.d \leftarrow 0$

Prova di correttezza

Teorema

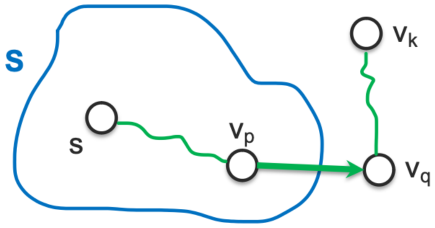
Sia $\langle v_1 = s, v_2, ..., v_k, v_{k-1}, ..., v_n \rangle$ la sequenza di vertici estratti da Q in un'esecuzione. Quando il vertice v_k viene estratto, allora $v_k.d = \delta(s, v_k)$.

Lemma

Se $v.d = \delta(s, v)$ a qualche passo dell'esecuzione, allora sicuramente $v.d$ rimarrà uguale a $\delta(s, v)$ per il resto dell'esecuzione.

Dimostrazione per assurdo

Assumiamo che $v_k.d < \delta(v_k)$, allora il cammino minimo dalla sorgente a v_k deve coinvolgere vertici del set $V - R$ (dove R contiene tutti i vertici eliminato dalla coda Q). Prendiamo in considerazione quindi un vertice v_q di questo set che fa parte del cammino fino a v_k e il suo predecessore v_p.



Quando v_p esce da Q, tutti i suoi archi vengono <u>rilassati</u>, compreso v_q dove troviamo quindi $v_q.d = \delta(v_q)$. Non avendo archi di peso nullo, allora $\delta(v_q) < \delta(v_k) \rightarrow v_q.d < v_k.d$.

Questo però può accadere solo se v_q viene scelto prima v_k dall'algoritmo, cosa che però è in contraddizione con la scelta v_k.

BFS (visita in ampiezza)

Introduzione

BFS (G,s) visita in ampiezza partendo dal vertice sorgente s, visitando man mano tutti gli adiacenti, calcolando così la distanza di ognuno dalla sorgente.

✅ Visita tutti e soli i vertici raggiungibili da s

✅ Ogni vertice visitato una sola volta

✅ Permette di stabilite la distanza da s di tutti i vertici raggiungibili

Struttura di base

Color

Vettore dei colori:

- Bianco: vertice non visitato
- Grigio: vertice visitato (ma adiacenti non tutti visitati)
- Nero: vertice e i suoi adiacenti visitati

D (distanza)

All’inizio

- $v.d = \infty$ per un qualsiasi vertice v
- $s.d = 0$ per la sorgente

Al termine

- $v.d = n$ per tutti i vertici v raggiungibili a distanza n

- $v.d = \infty$ per tutti i vertici v non raggiungibili
- $s.d = 0$ per la sorgente

π (predecessore)

All’inizio

- $v.d = NIL$ per ogni vertice, compresa la sorgente

Al termine

- $v.\pi = u$ per vertice v con predecessore u
- $v.\pi = NIL$ per tutti i vertici v non raggiungibili
- $s.\pi = NIL$ per la sorgente (non avendo predecessori)

Coda Q

Contiene solo i vertici di colore grigio, la visita termina quando Q è vuota

Operazioni

- head(Q), restituisce vertice in testa
- enqueue(Q, v), inserisce vertice v in testa (il quale è stato appena visitato e diventa grigio)
- dequeue(Q), elimina il vertice in testa (il quale è diventato nero)

Algoritmo

Procedura BFS (G, s)

    FOREACH $v \in V \setminus \{s\}$

        color[v]= W

        d[v]=∞

        π[v]=NIL

    color[s]= G

    d[s]=0

    π[s]=NIL

    $Q = \emptyset$

    ENQUEUE (Q, s)

    WHILE $Q \neq \emptyset$

        $v =$ HEAD(Q)

        FOREACH $u \in adj(v)$

            IF color[u] = W

color[u] = G

ENQUEUE (Q, v)

$d.u = d.v + 1$

$\pi[u] = v$

DEQUEUE(Q)

color[v]=B

Complessità in tempo

Sapendo che:

- Costo inizializzazione: $O(|V|)$
- Ogni lista di adiacenza ispezionata al più una volta, con costo: $O(|E|)$

Quindi in totale la complessità vale: $O(|V| + |E|)$

DFS (visita in profondità)

Introduzione

DFS (G) visita in profondità di un grafo G:

1. Sceglie arbitrariamente un vertice s come sorgente e visita s
2. Visita un adiacente a_1 di s, poi un adiacente a_2 di a_1, ecc ecc…
3. Quando raggiunge un vertice senza adiacenti, risale al predecessore e cerca un nuovo adiacente, nel caso risalendo fino a trovarne uno.
4. Quando risale fino ad s e non ha più nessun adiacente nuovo da visitare, si cerca una nuova sorgente e si riparte dal punto 2.
5. Tutto termina quando non ci sono più vertici disponibili ad essere selezionati come sorgenti.

Struttura di base

Color

Vettore di colori associati ai vertici:

- Bianco: vertice non ancora visitato
- Grigio: vertice visitato ma non ancora visitati tutti gli adiacenti
- Nero: vertice e adiacenti visitati

$\pi[v]$

Indica il predecessore di v nella visita

All'inizio

- $\pi[v] = NIL$

Al termine

- $\pi[s] = NIL$ se un vertice sorgente
- $\pi[v] = u$, indica u come predecessore di v nella visita

Vettore dei tempi

d[v]

Vettore dei tempi di scoperta, ovvero segna il tempo quando v passa da bianco a grigio.

$d[v] \in \{1, 2, ..., 2|V|\}$

f[v]

Vettore dei tempi di completamento, ovvero quando v passa da grigio a nero.

$f[v] \in \{1, 2, ..., 2|V|\}$ e $f[v] > d[v]$

Etichettatura degli archi

Arco d'albero (arco T)

v bianco e u grigio quando l'arco viene esplorato $\rightarrow$ u predecessore di v

Arco all'indietro (arco B)

v grigio quando l'arco viene esplorato $\rightarrow$ u non è predecessore di v ($d[v] < d[u]$)

Arco in avanti (arco F)

v nero e $d[u] < d[v]$ quando l'arco viene visitato $\rightarrow$ u non è predecessore di v

Arco trasversale (arco C)

v nero e $d[u] > d[v]$ quando l'arco viene visitato $\rightarrow u$ non è predecessore di v e non sono antenati.

Teorema delle parentesi

Dopo la visita in profondità, si possono verificare 3 casi con u e v:

1. $[d[u], f[u]]$ contiene $[d[v], f[v]] \rightarrow v$ discende da u nell'albero.
2. $[d[v], f[v]]$ contiene $[d[u], f[u]] \rightarrow u$ discende da v nell'albero.
3. $[d[v], f[v]]$ e $[d[u], f[u]]$ sono disgiunti $\rightarrow u$ e v non discendono l'uno dall'altro.

Dimostrazione Caso 1 (d[u]<d[v])

Due possibilità:

$d[v] < f[u]$

Vengono ispezionati tutti gli archi uscenti da v prima di riprendere l'ispezione degli archi uscenti da u. Quindi v discende da u $(f[v] < f[u])$

$\rightarrow [d[u], f[u]]$ contiene $[d[v], f[v]]$

$f[u] < d[v]$

Sicuramente $d[u] < f[u] < d[v] < f[v]$, quindi prima di visitare v ho già finito di visitare u e i suoi adiacenti, segue che nessuno discende dall'altro.

Dimostrazione Caso 2 (d[u]>d[v])

Semplicemente applico la dimostrazione precedente scambiando i ruoli di v e u.

Algoritmo

Procedura DFS (G)

FOREACH $v \in V$

color[v] = W

π[v]=NIL

d[v]=0

f[v]=0

time=0

FOREACH $v \in V$

IF color[v]=W

DFS_visit (G,v)

Procedura DFS_visit (G,u)

time= time + 1

d[u] = time

color[u] = G

FOREACH $v \in adj(u)$

IF color[v]=W

π[v]= u

DFS_visit (G, u)

time= time + 1

f[u]= time

color[u]=B

Complessità in tempo

- Costo inizializzaione: $O(|V|)$
- DFS_visit viene chiamata nel caso peggiore una volta per vertice: $O(|V|)$ chiamate
- In DFS_visit in totale il costo di ispezione delle liste di adiacenza è: $O(|E|)$

Quindi in totale la complessità vale: $O(|E| + |V|)$.

Algo con etichettatura archi

Per adattare l’algoritmo precedente, basta modificare DFS_visit, DFS possiamo lasciarlo invariato.

Procedura DFS_visit (G,u)

time = time + 1

d[u] = time

color[u]= G

FOREACH $v \in adj(u)$

IF color[v]= W

π[v]= u

DFS_visit(G, v)

$(u, v) \rightarrow$ “Arco T”

ELSE IF color[v]= G

$(u, v) \rightarrow$ “Arco B”

ELSE IF $d[u] < d[v]$

$(u, v) \rightarrow$ “Arco F”

ELSE

$(u, v) \rightarrow$ “Arco C”

color[u] = B

time= time + 1

f[u] = time

Ordinamento Topologico

Considerando un grafo G = (V,E) orientato aciclico, l’ordinamento topologico è un elenco dei vertici: $T = \langle v_1, ..., v_n \rangle$ tale che per ogni arco (v_i, v_j) si ha che v_i viene prima di v_j in T.

> 🚨 Nel cado del DFS l’ordinamento topologico coincide con la lista (pila) dei vertici disposta in ordine decrescente rispetto a $f[v]$

Algoritmo DFS con ordinamento topologico

Procedura DFS (G)

FOREACH $v \in V$

color[v] = W

π[v]=NIL

d[v]=0

f[v]=0

time=0

S = empty stack

FOREACH $v \in V$

IF color[v]=W

DFS_visit (G,v)

WHILE NOT isEmpty(s)

print top(S)

pop(S)

Procedura DFS_visit (G,u)

time= time + 1

d[u] = time

color[u] = G

FOREACH $v \in adj(u)$

IF color[v]=W

π[v]= u

DFS_visit (G, u)

time= time + 1

f[u]= time

color[u]=B

push (S, u)