Cheatsheet APA

Quack

Questo file non è esaustivo e potrebbe contenere errori.
Ringrazio Fagadau Daniel per i suoi appunti.

Sommario

PD

Teoria

Dimostrazione Sottostruttura Ottima LCS: Siano $X = < x_1, ..., x_m >$ e $Y = < y_1, ..., y_n >$ due sequenze e sia $Z = < z_1, ..., z_k >$ una LCS di X e Y.

1. Se $x_m = y_n$ allora $x_m = y_n = z_k$ e $Z_{k-1} = LCS(X_{m-1}, Y_{n-1})$:
   - **Per assurdo**: $x_m \neq z_k$, allora esiste $Z' = Z + < x_m >$ tale che Z' è sottosequenza comune di X e Y e $|Z'| > |Z|$ e quindi Z non può essere la LCS
   - **Per assurdo**: $Z_{k-1} \neq LCS(X_{m-1}, Y_{n-1})$, allora esiste Z'_{k-1} sottosequenza comune a X_{m-1} e Y_{n-1} tale che $|Z'_{k-1}| > |Z_k|$
2. Se $x_m \neq y_n$ allora $z_k \neq x_m$ implica che $Z = LCS(X_{m-1}, Y_n)$
   - **Per assurdo**: $Z \neq LCS(X_{m-1}, Y_n)$ allora esiste Z' sottosequenza comune a X_{m-1} e Y_n tale che $|Z'| > k$. Z non può essere LCS
3. Se $x_m \neq y_n$ allora $z_k \neq y_n$ implica che $Z = LCS(X_m, Y_{n-1})$
   - **Per assurdo**: $Z \neq LCS(X_m, Y_{n-1})$ allora esiste Z' sottosequenza comune a X_m e Y_{n-1} tale che $|Z'| > k$. Z non può essere LCS

Dimostrazione Sottostruttura Ottima LIS: Sia $X =< x_1, ..., x_m >$ una sequenza e sia X_i un suo prefisso di lunghezza i. Sia Z_i una tra le più lunghe sottosequenze crescenti di X_i e che termina con x_i. Allora vale che $Z^i = Z^* | x_i$, $Z^* \epsilon W_i$, $|Z^*| = max\{|W| \; tc \; W \epsilon W_i\}$ dove W_i è l'insieme di tutte le sottosequenze crescenti di X_j e che finiscono con x_j a cui accodabile x_i.
Per assurdo: $Z^i \neq Z^* | x_i$, allora vale $Z^i = Z' | x_i$ e $|Z'| > |Z^*|$ dove Z' è una sottosequenza crescente di prefisso inferiore a X_i. Sia z' l'ultimo elemento di Z', vale che $z' < x_i$. Sia $h < i$ il più grande indice tale che $x_h = z'$. Si ottiene che $Z' \epsilon W_i$ ma ciò porta ad una contraddizione: $|Z'| > |Z^*|$ è contro l'ipotesi $|Z^*| = max\{|W| \; tc \; W \epsilon W_i\}$

Dimostrazione Sottostruttura Ottima LICS: Siano X sequenza di m interi e X_i un suo prefisso, e Y sequenza di n interi e Y_j suo prefisso. Sia Z^{ij} una LICS di X_i e Y_j e che termina con x_i e y_j. Allora vale che $Z^{ij} = Z^* | x_i$, $Z^* \epsilon W_{ij}$, $|Z^*| = max\{|W| \; tc \; W \epsilon W_{ij}\}$ dove W_{ij} è l'insieme di tutte le sottosequenze crescenti comuni a x_h e y_k a cui è accodabile x_i.
Per assurdo: $Z^{ij} \neq Z^* | x_i$, allora vale $Z^{ij} = Z' | x_i$ e $|Z'| > |Z^*|$ dove Z' è una sottosequenza comune crescente di prefisso inferiore a X_i e Y_j. Sia z' l'ultimo elemento di Z', vale che

$z' < x_i$. Siano $r < i$ e $s < j$ i più grandi indici tali che $x_r = y_s = z'$. Si ottiene che $Z' \epsilon W_{ij}$ ma ciò porta ad una contraddizione: $|Z'| > |Z^*|$ è contro l'ipotesi $|Z^*| = max\{|W| \, tc \, W \epsilon W_{ij}\}$

Dimostrazione Sottostruttura Ottima WIS: Sia $i \epsilon \{2, ..., n\}$ (sottoproblemi del passo ricorsivo). Assumo di aver già risolto i sottoproblemi più piccoli: $i - 1, i - 2, ..., 2, 1$, ossia di conoscere $S_{i-1}, S_{i-2}, ..., S_2, S_1$. Allora vale l'equazione di ricorrenza:
$S_i = \{S_{i-1} \; se \; OPT_{i-1} \geq OPT_{P(i)} + v_i\} \; S_i = \{S_{P(i)} \cup \{i\} \; altrimenti\}$

- $i \notin S_i$. Per assurdo S_{i-1} non è la soluzione del problema i-esimo. In particolare $S_i \neq S_{i-1}$ e $v(S_i) > v(S_{i-1})$. $S_i \subseteq \{1, ..., i - 1\}$ e $COMP(S_i) = true$. Allora S_{i-1} non è soluzione del problema i-1-esimo.
- $i \epsilon S_i$. Per assurdo $S_{P(i)} \cup \{i\}$ non è la soluzione del problema i-esimo. In particolare $S_i \neq S_{P(i)} \cup \{i\}$ e $v(S_i) > v(S_{P(i)}) + v_i$. $S_i = S' \cup \{i\}$ e $COMP(S_i) = true$. Allora $S' \subseteq \{1, ..., p(i)\}$ e $COMP(S') = true$. Posso riscrivere come $v(S') + v_i > v(S_{P(i)}) + v_i$ ovvero $v(S') > v(S_{P(i)})$ e quindi $S_{P(i)}$ non può essere soluzione del problema P(i)-esimo

Dimostrazione Sottostruttura Ottima Hateville: Sia $i \epsilon \{2, ..., n\}$ (sottoproblemi del passo ricorsivo). Assumo di aver già risolto i sottoproblemi più piccoli: $i - 1, i - 2, ..., 2, 1$, ossia di conoscere $S_{i-1}, S_{i-2}, ..., S_2, S_1$. Allora vale l'equazione di ricorrenza:
$S_i = \{S_{i-1} \; se \; OPT_{i-1} \geq OPT_{i-2} + d_i\} \; S_i = \{S_{i-2} \cup \{i\} \; altrimenti\}$
$i \notin S_i$. Per assurdo S_{i-1} non è la soluzione del problema i-esimo. In particolare $S_i \neq S_{i-1}$ e $D(S_i) > D(S_{i-1})$. $S_i \subseteq X_{i-1}$ e $COMP(S_i) = true$. Allora S_{i-1} non è soluzione del problema i-1-esimo

- $i \epsilon S_i$. Per assurdo $S_{i-2} \cup \{i\}$ non è la soluzione del problema i-esimo. In particolare $S_i \neq S_{i-2} \cup \{i\}$ e $D(S_i) > D(S_{i-2}) + d_i$. $S_i = S' \cup \{i\}$ e $COMP(S_i) = true$. Allora $S' \subseteq X_{i-2}$ e $COMP(S') = true$. Posso riscrivere come $D(S') + d_i > D(S_{i-2}) + d_i$ ovvero $D(S') > D(S_{i-2})$ e quindi S_{i-2} non può essere soluzione del problema i-2-esimo

Pratica

Tutti gli esercizi di PD del tipo "Date due sequenze trovare sequenza comune crescente/decrescente/alternante con qualche vincolo" sono facilmente risolvibili in questo modo:

1. **Introduzione Problema Ausiliario**: riscrivere il sottoproblema di dimensione (i,j) ed aggiungere alla fine "e che termina con x_m e y_n se questi coincidono". Esempio:
    a. Sottoproblema P' LICS: date due sequenze X e Y , rispettivamente di m ed n numeri interi, si determini la lunghezza di una tra le più lunghe sottosequenze crescenti comuni al prefisso X_i e al prefisso Y_j e che termina con x_m e y_n se questi coincidono
2. **Calcolo Coefficiente**: è la lunghezza del sottoproblema del problema ausiliario + specificare lunghezza prefissi. Esempio:

    a. Coefficiente P': c_{ij} = lunghezza di una tra le più lunghe sottosequenze crescenti comuni a X_i e a Y_j, con $i \in \{1, ..., m\}$ e $j \in \{1, ..., n\}$ e che termina con x_m e y_n se questi coincidono
3. **Caso Base**: $c_{ij} = 0 \; se \; x_i \neq y_j$
4. **Passo Ricorsivo**: $c_{ij} = 1 + max\{c_{hk} \mid 1 \leq h < i, 1 \leq k < j \mid condizioni\}$ dove in condizioni inseriamo la condizione del problema. Esempi:
    a. LCS Crescente: $x_h < x_i$
    b. LCS Decrescente: $x_h > x_i$
    c. LCS Alternante: $x_h \neq x_i$
    d. LCS Pari/Dispari: $x_h \; mod \; 2 \neq x_i \; mod \; 2$
    e. LCS Alterna Qualcosa (es. Colore): $f(x_h) \neq f(x_i)$
    f. LCS No 2 cons: $x_h != qualcosa \vee x_i != qualcosa$
5. **Valore Ottimo**: $c_{mn} = max\{c_{ij} \mid 1 \leq i \leq m, 1 \leq j \leq n\}$
6. **Algoritmo DP**:

```
Unset
procedure L-CS(X, Y)
max = 0
for i=1 to m
	for j=1 to n
		if x[i] != y[j]
			C[i,j] = 0
		else
			tmp = 0
			for h = 1 to i -1
				for k = 1 to j - 1
					if condizione AND C[h,k] > tmp
						tmp = C[h,k]
						H[i,j] = (h,k)
			C[i,j] = tmp + 1
			if C[i,j] > max
				max = C[i,j]
return max
```

7. **Algoritmo Ricostruzione**:

```
Unset
procedure print-L-CS(i,j)
	if H[i,j] != (0,0)
		print-L-CS(H[i,j])
	append x[i]
```

Osservazione: Per altri esercizi PD non risolvibili in questo modo, come LCS, LIS, Knapsack ed altri, consiglio di guardare il file del Prof. Dennunzio.

FW

Teoria

Dato un grafo orientato e pesato G= (V,E) con W matrice dei pesi, vogliamo calcolare il peso del cammino minimo da i a j, per ogni coppia di vertici (i,j). Algoritmo di Programmazione Dinamica.

```
Unset
Floyd-Warshall(V,E,W)
D[0] = W
Π[0] = matrix (n x n) of NIL values
for i = 1 to n
       for j = 1 to n
              if i != j and w[i,j] != ∞
                     Π[0][i,j] = i
for k = 1 to n
       for i = 1 to n
              for j = 1 to n
                     D[k][i,j] = D[k-1][i,j]
                     Π[k][i,j] = Π[k-1][i,j]
                     if i != k and k != j
                            if D[k][i,j] > D[k-1][i,k] + D[k-1][k,j]
                                   D[k][i,j] = D[k-1][i,k] + D[k-1][k,j]
                                   Π[k][i,j] = Π[k-1][k,j]
```

Tempo di Esecuzione FW: $\Theta(n^3)$

Costruzione cammini minimi: un possibile metodo consiste nel costruire la matrici D dei pesi dei cammini minimi e da questa ricavare la matrice dei predecessori Π:

- Quando k = 0, un cammino minimo da i a j non ha alcun vertice intermedio, quindi il predecessore di k è o NIL se i=j o arco infinito oppure è i.
- Per k ≥ 1, se prendiamo i cammini i ↠ j e k ↠ j, dove $k != j$, allora il predecessore di j che scegliamo è uguale al predecessore di j che avevamo scelto in un cammino minimo da k con tutti i vertici in $\{1, ..., k-1\}$. Altrimenti, scegliamo lo stesso predecessore di j che avevamo scelto in un cammino minimo da i con tutti i vertici intermedi in $\{1, ..., k-1\}$.

Caratterizzazione cammini minimi: La caratterizzazione della struttura di cammino minimo di Floyd-Warshall si base sul concetto di vertice intermedio. Dato un cammino $p = \{v_1, ..., v_l\}$ si considera vertice intermedio qualunque vertice dell'insieme $\{v_2, ..., v_{l-1}\}$ Consideriamo un sottoinsieme $\{1, ..., k\}$ di vertici. Per una coppia di vertici qualsiasi i, j ∈ V , consideriamo tutti i cammini i cui vertici intermedi sono tutti in $\{1, ..., k\}$ e sia p un cammino minimo fra di essi.

- Se k non è vertice intermedio di p, allora tutti i vertici intermedi di p sono nell'insieme $\{1, ..., k-1\}$. Quindi, un cammino minimo dal i a j con tutti i vertici

intermedi in $\{1, \ldots, k-1\}$ è anche un cammino minimo da i a j con tutti i vertici intermedi in $\{1, \ldots, k\}$.

- Se k è un vertice intermedio di p allora spezziamo p in p1:i → k e p2:k → j.
    - p1 è un cammino minimo da i a k con tutti i vertici nell'insieme $\{1, \ldots, k\}$. Poiché k non è vertice intermedio di p1, allora p1 è un cammino minimo da i a k con tutti i vertici intermedi in $\{1, \ldots, k-1\}$.
    - Vale lo stesso ragionamento per p2

Chiusura transitiva di un grafo orientato G=(N,A) è un grafo orientato $G_+ = (N, A_+)$, tale che un arco (i,j) è in A_+ se e solo se esiste un cammino da i a j in G. Un modo per calcolare la chiusura transitiva di un grafo consiste nell'assegnare 1 a ogni arco in E e nell'eseguire l'algoritmo di Floyd-Warshall. Se esiste un cammino dal vertice i al vertice j, si ha $d_{ij} < n$, altrimenti $d_{ij} = \infty$.

Chiusura transitiva in FW: si possono modificare le equazioni di ricorrenza di Floyd-Warshall in questo modo:

- si sostituisce a min un ∨
- si sostituisce a + un ∧

Le variabili assumono valori True o False, in base a se esiste o meno il cammino minimo.

Pratica

Tutti gli esercizi che non richiedono un problema ausiliario sono risolvibili nel seguente modo:

1. **Definizione Coefficiente**:
    a. Richiede cammino minimo: $d^{(k,\ldots)}(i,j)$ **è il peso** di un cammino minimo dal vertice i al vertice j, i cui vertici intermedi appartengono all'insieme $\{1, \ldots, k\}$e che **condizione**.
    b. Richiede esistenza cammino minimo:$d^{(k,\ldots)}(i,j)$ **vale True** di un cammino minimo dal vertice i al vertice j, i cui vertici intermedi appartengono all'insieme $\{1, \ldots, k\}$e che **condizione**, altrimenti False.
2. **Caso Base**: Il passo più difficile. Si ha per k = 0, ovvero quando abbiamo solo un nodo (i = j) o due nodi (i !=j). Bisogna innanzitutto vedere se è necessario mantenere una nuova variabile per tenere conto di qualcosa (numero vertici-archi cammino, pari-dispari, massimo numero vertici-archi che soddisfano una condizione ecc..). Bisogna controllare se la condizione è =, ≥ o ≤ , bisogna controllare se richiede il controllo su archi o sui vertici. Purtroppo non so come sintetizzare.
3. **Passo Ricorsivo:** Si ha per k > 0. Quasi sempre basta comportarsi così:
    a. $k \notin cammino\ minimo$: $d^{(k,\ldots)}(i,j) = d^{(k-1,\ldots)}(i,j) = e_0$
    b. $k \in cammino\ minimo$:
    $d^{(k,\ldots)}(i,j) = min\{ d^{(k-1,\ldots)}(i,k) + d^{(k-1,\ldots)}(k,j)\}\ tc\ controllo = e1$, dove il controllo è sulle altre variabili, esempio:
        i. $d^{(k,h)}(i,j) = min\{ d^{(k-1,h1)}(i,k) + d^{(k-1,h2)}(k,j)\}\ tc\ h1 + h2 = h = e_1$ il seguente codice controlla che la somma dei due cammini sia esattamente quella del cammino originale.

   c. $d^{(k,\dots)}(i,j) = min\{e_0, e_1, \dots, e_n\}$
   d. Se richiede esistenza cammino: sostituisci il + con un AND e sostituisci il min con V.
4. Valore Ottimo: $d^{(n)}(i,j)$

Gli esercizi che richiedono il problema ausiliario sono tendenzialmente quelli in cui si chiede una condizione tra archi adiacenti. Dato che bisogna controllare l'ultimo arco prima k e il primo arco dopo k ma non abbiamo a disposizione tale informazioni, si introducono le variabili (c,d) che rappresentano il primo arco e l'ultimo arco del grafo. Di seguito un esempio:

Esistenza di cammini senza archi consecutivi rossi:

1. **Definizione Coefficiente**: $d^{(k)}(i, j, a, b)$ vale True se esiste un cammino minimo dal vertice i al vertice j, con vertici intermedi appartenenti all'insieme $\{1, \dots, k\}$, senza due archi consecutivi di colore rosso, con il primo arco di colore a, e l'ultimo arco di colore b, False altrimenti.
2. **Caso Base**:
   $k = 0,\ d^{(0)}(i, j, a, b) = True\ se\ i \neq j \wedge (i, j) \in E \wedge a = b = col(i,j),\ False\ altrimenti$
3. **Passo Ricorsivo**: $k > 0$:
   a. $k \notin cammino\ minimo$: $d^{(k)}(i, j, a, b) = \{d^{(k-1)}(i, j, a, b)\} = e_0$
   b. $k \in cammino\ minimo$:
      $d^{(k)}(i, j, a, b) = \{d^{(k-1)}(i, k, a, c) \wedge d^{(k-1)}(k, j, d, b)\ tc\ c \neq R \vee d \neq R\} = e_1$
   c. In conclusione: $d^{(k)}(i, j, a, b) = e_0 \vee e_1$
4. **Valore Ottimo**: $d^{(n)}(i,j)$

BFS

Teoria

```
Unset
BFS(G, s) {
for all u in V - {s}
       u.col = WHITE
       u.d = ∞
       u. = NIL
s.col = GREY
s.d = 0
s.π = NIL
enqueue(Q, s)
while Q ≠ Ø
       u = dequeue(Q)
       for all v Adj[u]
              if v.col == WHITE
                     v.col = GREY
                     v.d = u.d + 1
                     v.π = u
                     enqueue(Q, v)
       u.col = BLACK
}
```

Tempo di Esecuzione: l'inizializzazione richiede $O(V)$, Le operazioni di enqueue e dequeue richiedono tempo costante e sono eseguite una volta per ogni vertice, quindi $O(V)$ in totale. La scansione delle liste di adiacenza avviene una volta per ogni vertice. La lunghezza totale è $\Theta(E)$ quindi il tempo è $O(E)$. In totale il tempo è: $O(V + E)$

Colori Vertici: I vertici di un grafo possono assumere durante l'esecuzione di una BFS, tre colori:

1. bianco: il nodo non è ancora stato scoperto
2. grigio: il nodo è stato scoperto ma la sua lista di adiacenza non è stata esplorata del tutto
3. nero: il nodo è stato scoperto e la lista di adianceza è stata completamente esplorata

Osservazione: Se mi serve sapere solo i nodi raggiungibili dalla sorgente, ovviamente non ho bisogno dei **colori** o dei **predecessori**. Queste sono informazioni che ci servono per ulteriori implementazioni dell'algoritmo.

Il **grafo dei predecessori**, è definito formalmente come $G_\pi = (V_\pi, E_\pi)$ dove:

- $V_\pi = \{v \in V: v.\pi \neq NIL\} \cup \{s\}$
- $E_\pi = \{(v.\pi, v) : v \in V - \{s\}\}$

Il grado dei predecessori contiene quindi tutti i vertici raggiungibili dal vertice sorgente e, per ognuno di essi, il cammino minimo dal vertice sorgente ad esso.

Pratica

Contare vertici raggiungibili: Basta che chiamo BFS, e scorro tutti i vertici, per ogni vertice che ha distanza diversa da infinito incremento il contatore.

```
Unset
BFS(G=(V, x)
n = 0
for all v in V
	if v.d != ∞
		n++
Return n
```

Stabilire se un grafo non orientato è albero: sapendo che è non orientato, per essere albero manca aciclico e connesso:

- connesso: come prima, BFS visita solo i nodi raggiungibili, quindi se trovo almeno un nodo che ha colore bianco, allora non è connesso
- aciclico: presa la lista di adiacenza di un nodo, se trovo un vertice grigio, vuol dire che l'ho visitato in precedenza e quindi ho ciclo.

```
Unset
isConnected(G) {
s = Random(V)
BFS(G, s)
for all v in V
	if v.d == ∞
		return FALSE
return TRUE
}
```

```
Unset
...
for all v Adj[u]
	...
	if v.col == GREY
		return FALSE
	return TRUE
```

Variante grafo non orientato è albero: sapendo che il grafo è connesso, questo sarà un

albero se $|E| = |V| - 1$. Possiamo allora usare una variabile che conta gli archi e viene incrementata per ogni vertice della lista di adiacenza, mentre il numero di vertici viene fornito da “Adj.lenght”. Bisogna dividere e/2 perchè grafo è non orientato e quindi conta 2 volte.

```
Unset
...
for all v Adj[u]
	ne++
	...
if isConnected(G) AND ne/2 == Adj.length -1
	return TRUE
return FALSE
```

Stabilire se cc è albero: dobbiamo solo contare il numero di vertici e archi in quanto sappiamo già che la componente è connessa. In questo caso però il numero di vertici non è dato da Adj.length, ma bisogna usare una variabile che viene incrementata ogni volta che effettuo una dequeue.

```
Unset
while Q ≠ Ø
	u = dequeue(Q)
	nv++
	for all v Adj[u]
		ne++
		...
if ne/2 == nv - 1
	return TRUE
return FALSE
```

Modifica BFS tale che considera nodi a distanza minore uguale a k: semplicemente faccio la enqueue dei soli vertici a distanza minore di k

```
Unset
for all v Adj[u]
	if v.d == ∞
		v.d = u.d + 1
		if v.d < k
			enqueue(Q, v)
```

Stabilire se cc siano grafi completi:

- se il grafo è non orientato deve valere la seguente proprietà: $|E| = \frac{(n-1)\cdot n}{2}$.
- se il grafo è orientato, deve valere la seguente proprietà: $|E| = n \cdot (n-1)$

Mi comporto allo stesso modo del problema dell'albero e cambio il controllo.

```
Unset
while Q ≠ Ø
        u = dequeue(Q)
        nv++
        for all v Adj[u]
                ne++
                ...
if "controllo numeri archi"
        return TRUE
return FALSE
```

DFS

Teoria

```
Unset
DFS(G)
for all v in V
	v.col = WHITE
	v.π = NIL
time = 0
for all v in V
	if v.col == WHITE
		DFS_Visit(G, v)

DFS_Visit(G, u)
time++
u.d = time
u.col = GREY
for all w in Adj[u]
	if w.col == WHITE
		w.π = u
		DFS_Visit(G, w)
u.col = BLACK
time++
u.f = time
```

Tempi di Esecuzione: la procedura DFS, avviene per ogni vertice, quindi $\Theta(V)$. La procedura DFS-VISIT è chiamata per ogni vertice, e il ciclo interno viene eseguito tante volte quanto la lunghezza della lista di adiacenza, ovvero $\Theta(E)$. In totale quindi $\Theta(V + E)$

Osservazione: Il numero di chiamate a DFS-VISIT da DFS corrisponde al numero di componenti connesse di un grafo non orientato. Questo perchè, per come è strutturata una DFS, DFS-VISIT viene chiamata da DFS in modo da formare un albero DF disgiunto dagli altri delle foresta DF

Il **grafo dei predecessori** prodotto da una visita DF è definito formalmente come $G_\pi = (V, E_\pi)$ dove:

- $E_\pi = \{(v.\pi, v) : v \in V \ \wedge \ v.\pi \neq NIL\}$

Il sottografo è una foresta DF composta da vari alberi DF. Gli archi in E_π sono archi dell'albero.

Classificazione Archi: definiamo quattro tipi di archi in un grafo orientato:

- Arco Tree: sono gli archi nella foresta DF. Sono gli archi tale che la destinazione ha colore White.
- Archi Backward: sono gli archi che collegano un vertice u ad un suo antenato v in un albero DF. Sono gli archi tali che la destinazione ha colore Grey.

- Archi Forward: sono gli archi che collegano un vertice u ad un discendente v in un albero DF. Sono gli archi tali che la destinazione ha colore Black e u.d < v.d
- Archi trasversali: tutti gli altri archi, ovvero quelli tali che la destinazione ha colore Black e u.d ≥ v.d

Teorema delle parentesi: Dopo una visita in profondità, uno solo dei tre casi seguenti si può verificare per due vertici u e v:

1. [d[u],f[u]] contiene [d[v],f[v]] ovvero v è discendente di u in un albero della visita
2. [d[v],f[v]] contiene [d[u],f[u]] ovvero u è discendente di v in un albero della visita
3. [d[u],f[u]] e [d[v],f[v]] sono disgiunti ovvero u e v non sono discendenti l'uno dell'altro in un albero della visita

Dimostrazione Teorema parentesi:

1. Caso 1: d[u] < d[v]
    a. d[v] < f[u]
        i. verranno ispezionati tutti gli archi uscenti da v prima di riprendere l'ispezione degli archi uscenti da u
        ii. v è discendente di u in un albero della visita
        iii. f[v] < f[u]
        iv. [d[u], f[u]] contiene [d[v], f[v]]
    b. f[u] < d[v]
        i. sicuramente d[u] < f[u] e d[v] < f[v] allora d[u] < f[u] < d[v] < f[v}
        ii. nessuno dei due vertici è stato scoperto mentre l'altro era grigio
        iii. nessuno dei due vertici è discendente dell'altro nello stesso albero della visita
        iv. [d[u],f[u]] e [d[v],f[v]] sono disgiunti
2. Caso 2: d[u] > d[v] si ripete scambiando i ruoli dei due vertici

Ordinamento Topologico: avviene su un DAG, ovvero un grafo orientato aciclico. Rappresenta un ordinamento lineare dei suoi vertici, tale per cui se nel grafo esiste un arco (u,v) allora u comparirà prima di v nell'ordinamento. Per ottenere un ordinamento topologico possiamo sfruttare DFS:

```
Unset
procedure topological-sort(G)
DFS(G)
una volta completata ispezione vertice, inserire vertice in testa a lista concatenata
ritornare la lista concatenata
```

Pratica

Contare i vertici: nell'iterazione, quando setto un nuovo vertice a grey, incremento la variabile che salva il numero di vertici.

```
Unset
DFS_Visit(G, u)
time++
u.d = time
u.col = GREY
nv++
```

Contare le cc di un grafo: Ogni volta che effettuiamo la visita, stiamo esplorando una nuova CC, quindi la incremento prima di effettuare una nuova procedura.

```
Unset
for all v in V
	if v.col == WHITE
		cc++
		DFS_Visit(G, v)
```

Contare quante cc sono alberi: come prima essendo non orientati e connessi basta che siano aciclico. Possiamo anche, invece di controllare che sia ciclico, usare la proprietà di vertici ed archi. P

```
Unset
DFS(G)
...
for all v in V
	if v.col == WHITE
		Acyclic = TRUE
		DFS_Visit(G, v)
		if ne/2 = nv -1
			nTree++

DFS_Visit(G,u)
nv++
...
for all w in Adj[u]
	ne++
	...
```

Contare quante cc sono grafi completi: stesso principio di prima ma cambia il controllo.

```
Unset
DFS(G)
...
for all v in V
	if v.col == WHITE
		Acyclic = TRUE
		DFS_Visit(G, v)
		if ne/2 == ((nv - 1) * nv)\2
			nComplete++

DFS_Visit(G,u)
nv++
...
for all w in Adj[u]
	ne++
	...
```

Etichettare archi grafo orientato: se il nodo è bianco è un T, se il nodo è grigio è un B, se il nodo di partenza è stato scoperto prima del nodo di arrivo è F, altrimenti C.

```
Unset
for all w in Adj[u]
	if w.col == WHITE
		print 'Tree-Edge'
		w.π = u
		DFS_Visit(G, w)
	else if w.col == GREY
		print 'Back-Edge'
	else if u.d < w.d
		print 'Forward-Edge'
	else
		print 'Cross-Edge'
```

Osservazione: se il grafo è non orientato, nello scorrimento non considero il parent di u, ed etichetto solo con T e B.

Stabilire se un grafo è foresta con k alberi: come nel contare le cc di un grafo, ma chiamo la visita solo se è ancora aciclico e il numero di cc è < k.

```
Unset
DFS(G)
for all v V
v.col = WHITE
v. = NIL
```

```
DFS(G)
...
CC = 0
while all v in  V AND CC <  k AND Acyclic
       if v.col == WHITE AND CC < k
              DFS_Visit(G, v)
              CC++
       else if v.col == WHITE
              CC++
if CC == k AND Acyclic
       Return TRUE
else
       Return FALSE

DFS_Visit(G, u)
...
while all w Adj[u]  \ u.π  AND Acyclic
       if w.col == WHITE
              ...
       else
              Acyclic = FALSE
...
```

Greedy e Matroidi

Teoria

Una coppia (S, F), dove S è un insieme finito di elementi e F famiglia di sottoinsiemi di S, è definita **Sistema di Indipendenza** se preso $A \in F$, allora un qualsiasi $B \subseteq A, B \in F$

Un sistema di indipendenza è un **Matroide** se $\forall A, B \in F \; tc \; |B| = |A| + 1$ allora $\exists \, b \in B - A \; tc \; A \cup \{b\} \in F$

Dato un grafo G = (V, E) non orientato e connesso, $M_G = (S, F)$ è il **matroide grafico di G**, tale che S è l'insieme degli archi E, e F tutti i sottoinsiemi di S aciclici.

Dato un matroide M = (S, F), $s \in S$ è detto **estensione** di $A \in F$ se $A \cup \{s\} \in F$

Dato un matroide M = (S, F), $A \in F$ è detto **massimale** se non ha estensioni

Teorema di Rado: la coppia (S, F) è matroide sse per ogni funzione peso W, l'algoritmo greedy standard fornisce la soluzione ottima

```
Unset
Greedy_Standard(I)
S = Ø
<calcolo n parametri>
ordino per parametro
for i=1 to n
       if I[i] può essere aggiunto
              S = S U {I[i]}
return S
```

Tempo di Esecuzione: $O(n \log n)$

Osservazione: Il problema associato ad una coppia costituita da un SI e una funzione peso su esso è un insieme $M \in F$ tale che il suo peso sia massimo. L'algoritmo procede così:

```
Unset
Greedy(E,F,w)
S = Ø
Q = E
while(Q ≠ Ø)
       determina elemento m di peso massimo in Q
       Q = Q - {m}
       if S U {m} ∈ F
              S = S U {m}
```

```
return S
```

Dimostrazione Matroide Grafico è Matroide: M_G è un matroide:

1. $A \in F, B \subseteq A \Rightarrow B \in F$
   a. Se $A \in F$, allora anche $B \subseteq A$ sarà aciclico e appartiene quindi a F
2. $\forall A, B \in F \; tc \; |B| = |A| + 1$ allora $\exists \, b \in B - A \; tc \; A \cup \{b\} \in F$
   a. Sino $A, B \in F$ tali che $|B| = |A| + 1$ ovvero
      i. G_A = foresta di $|V| - |A|$ alberi
      ii. G_B = foresta di $|V| - |B|$ alberi
   b. G_B ha un albero in meno di G_A e quindi in G_B esiste arco (u,v) che connette due vertici u e v che in G_A stanno in due alberi diversi: $\{(u, v)\} \cup A \in F$

Pratica

S insieme dei primi 1000 interi positivi, F famiglia dei sottoinsiemi A di S tale che la somma dei numeri è multiplo di 3.

- (S, F) è Sistema di Indipendenza, ovvero $A \in F, B \subseteq A$ allora $B \in F$? **No**, controesempio A = {2, 4, 3}, B = {4, 3}
- Conclusione: **(S, F) non è matroide**

E insieme finito di vettori in V, F sottoinsieme di E formato dai vettori linearmente indipendenti.

- (S, F) è Sistema di Indipendenza? **Si**, se ho un insieme indipendente e ne prendo un sottoinsieme, saranno ancora tra loro indipendenti.
- (S, F) è Matroide, ovvero vale la proprietà di scambio $\forall A, B \in F \; tc \; |B| = |A| + 1$ allora $\exists \, b \in B - A \; tc \; A \cup \{b\} \in F$? **Si**, se prendo un sottoinsieme di n vettori linearmente indipendenti e uno di n+1, deve esistere un vettore da aggiungere al più piccolo che non fa parte di questo, in quanto altrimenti l'insieme più grande sarebbe ottenibile dai n vettori del gruppo minore e quindi linearmente dipendente.
- Conclusione: **(S, F) è matroide**

Knapsack

- (S, F) è Sistema di Indipendenza? **Si**, perchè se A non supera il limite, sicuramente un sottoinsieme non lo farà
- Per (S, F) vale la proprietà di scambio? No, controesempio:
  a. valore 10 peso 50
  b. valore 20 peso 30
  c. valore 15 peso 40
- con limite a 70 e A = {a} e B = {b, c} non posso aggiungere nessun elemento ad A
- Conclusione: **Knapsack non è matroide**, infatti abbiamo dimostrato che se si prende come funzione peso il valore, knapsack non fornisce la soluzione ottima

Insiemi Disgiunti

Una Struttura per Insiemi Disgiunti è una collezione $S = \langle S_1, S_2, ..., S_k \rangle$ di insiemi disgiunti. Ogni insieme è identificato da un rappresentante, uno tra i membri dell'insieme stesso. Le operazioni supportate sono:

- MakeSet(x): Crea un nuovo insieme con un singolo membro
- Union(x, y): Unisce i due insiemi contenenti x ed y
- FindSet(x): Ritorna il rappresentante dell'insieme in cui x è contenuto

Implementazioni: è possibile rappresentare una struttura dati per insieme disgiunti in due modi:

- Lista, in cui ogni elemento ha: valore, puntatore alla testa, puntatore a next, puntatore alla coda (solo il rappresentante). Complessità:
  - Make-set(x): O(1)
  - Union(x,y): O(n)
  - Find-set(x): O(1)
- Foresta, in cui gli elementi di ogni albero hanno solo un puntatore al proprio genitore. La radice contiene in più un puntatore al rappresentante. Complessità:
  - Make-set(x): O(1)
  - Union(x,y): O(1)
  - Find-set(x): O(n)

Euristica Unione Pesata (Liste): Ogni lista conterrà anche un attributo con la sua lunghezza, così che quando faccio la Union la lista più corta viene aggiunta a quella più lunga. Allora m operazioni vengono fatte in $O(m + n \cdot \log n)$: dato che un singolo elemento può aggiornare il suo rappresentante log n volte e che ci sono n elementi, arriviamo a tale tempo.

Euristica Unione per Rango (Foreste): L'idea è di fare in modo di unire un albero 'corto' in subordine alla radice di uno lungo piuttosto che il contrario. Ad ogni nodo è associato un rango, ovvero il limite superiore per l'altezza del nodo, vale a dire il numero di archi del cammino più lungo fra sé ed una foglia.

Compressione dei Cammini: Visto che 'perdiamo tempo' a scorrere l'albero, possiamo modificarlo intanto per migliorare le findSet future.

```
Unset
findSet(x)
if x != P(x)
       P(x) = findSet(P(x))
Return P(x)
```

Ad ogni nodo viene assegnato come parent il rappresentante del proprio parent e questo permette di modificarlo per ottenere una struttura che migliorerà le findSet future

Tempi di Calcolo: l'unione per rango porta ad un tempo $O(m \cdot \log n)$. Se aggiungiamo la nuova findSet arriviamo ad un tempo $O(m \cdot \alpha(m, n))$, $\alpha \leq 4$ ovvero un tempo lineare.

MST

MST:

- Input: grafo connesso non orientato pesato G = (V, E) con funzione peso tale che W(u,v) è il peso dell'arco (u,v=
- Output: $T \subseteq E$ aciclico tale che:
    a. $\forall v \in V, \exists (u,v) \in T$

    b. $W(T) = \sum_{(u,v) \in T} W(u,v)$ è minimo.

Algoritmo Generico:

1. Inizializza un insieme A vuoto
2. Aggiunge ad ogni passo un arco (u,v) in modo tale che unito ad A è sottoinsieme dell'insieme T degli archi di MST.
3. L'algoritmo termina non appena A = T, ovvero G_A è MST.

```
Unset
GENERIC-MST(G,w)
A = ∅
while A != MST
      trova (u,v) arco sicuro per A
      A = A ∪ {(u, v)}
return A
```

Si dice **taglio** una partizione di un grafo non orientato (S, V − S).
Si dice che un **arco (u,v) attraversa un taglio** se una delle due estremità si trova in S e l'altra in V − S.
Si dice che un **taglio rispetta un sottoinsieme di archi** se nessun arco del sottoinsieme lo attraversa.
Si dice **arco leggero** per un taglio l'arco di peso minimo tra quelli che lo attraversano.

Teorema dell'arco sicuro: Sia

1. G = (V, E) un grafo connesso non orientato con una funzione peso w a valori reali definita in E.
2. Sia A un sottoinsieme di E che è contenuto in qualche MST per G
3. Sia (S, V − S) un taglio qualsiasi di G che rispetta A
4. Sia (u, v) un arco leggero che attraversa (S, V − S).

Allora, l'arco (u, v) è sicuro per A.

Dimostrazione arco sicuro: Sia T un MST che include A e non contiene arco leggero (u,v). Costruiamo $T' = A \cup \{(u,v)\}$. L'arco (u,v) forma un ciclo con gli archi del cammino p da u a v in T. Dato che u e v sono sui lati opposti del taglio (S, V - S) almeno un arco in T nel cammino p attraverso il taglio. Sia (x,y) quest'arco. L'arco (x,y) non è in A, perchè il taglio rispetta A. Dato che (x,y) è sul cammino da u a v in T, rimuovendolo rompe T in due componenti. Aggiungendo (u,v) le riconnettiamo formando un nuovo spanning tree $T' = (T - \{(x,y)\}) \cup \{(u,v)\}$. Dato che (u,v) è arco leggero che attraversa (S, V - S) e anche

(x,y) attraversa questo taglio, $w(u, v) \leq w(x, y)$ e quindi $w(T') \leq w(T)$. Ma dato che T è MST, allora anche T' deve esserlo. Rimane da dimostrare che (u,v) è veramente sicuro per A. Abbiamo

1. $A \subseteq T'$
2. $A \subseteq T$
3. $(x, y) \notin A$
4. $A \cup \{(u, v)\} \subseteq T'$

Dato che T' è MST allora (u,v) è sicuro per A.

Corollario: Sia

1. G = (V, E) connesso e non orientato con funzione peso w a valori reali in E.
2. Sia $A \subseteq E$ che include qualche MST per G
3. Sia $C = (V_C, E_C)$ una componente connessa di $G_A = (V, A)$.

Se (u,v) è un arco leggero che connette C a qualche altra componente in G_A, allora (u,v) è sicuro per A.

Dimostrazione: Il taglio $(V_C, V - V_C)$ rispetta A, e (u,v) è arco leggero per questo taglio, quindi è sicuro per A.

Kruskal

L'idea chiave dell'algoritmo di Kruskal è di selezionare gli archi più leggeri che non creano cicli, garantendo così che il MST risultante sia di peso minimo attraverso una approccio greedy. Come strutture, utilizza strutture per insiemi disgiunti.

```
Unset
KRUSKAL-MST(G=(V,E), W)
A = Ø
E = ⟨e1, ..., en⟩ ordinati per peso crescente
for each v in V
       makeSet(v)
for i = 1 to n
       (u,v) = ei
       if findSet(u) != findSet(v)
              A = A U {{u,v)}
              union(u,v)
return A
```

Tempi di Esecuzione: $O(|E| \, log|E|)$

- Ordinamento: $O(|E| \, log|E|)$ perchè ordiniamo |E| elementi
- Makeset: $O(|V|)$ perchè è operazione costante fatta su tutti i vertici
- Ciclo: $O(|E|\alpha)$ con $\alpha \leq log|E|$ perché effettuiamo operazioni sulla struttura della foresta per tutti gli archi

Kruskal:

- Comincia dal vertice dal peso minore

- Attraversa un nodo una sola volta
- Può gestire grafi non connessi
- Più efficiente per grafi sparsi

Prim

Nell'algoritmo di Prim, viene utilizzata una coda di priorità (o heap) per mantenere traccia degli archi e dei loro pesi. Ogni vertice ha due attributi:

- Chiave: rappresenta il peso minimo dell'arco che connette quel nodo all'MST parziale. Inizialmente vale infinito.
- Predecessore: tiene traccia del nodo nel MST parziale al quale è connesso con l'arco di peso minimo. Inizialmente vale NIL

Iterativamente, estrai il nodo con la chiave minima dalla coda di priorità, esamina i suoi archi adiacenti e aggiorna le chiavi e i predecessori se trovi archi con pesi minori. Continua finché tutti i nodi sono inclusi nell'MST parziale.

```
Unset
PRIM-MST(G,W,r)
foreach v in V
       v.key = ∞
       v.π = NIL
r.key = 0

Aggiungi tutti i vertici di V alla coda Q

while Q != Ø
       u = estrai vertice da Q
       foreach v ∈ adj(u)
              if v ∈ Q and W(u,v) < v.key
                     v.key = W(u,v)
                     v.π = u
```

Tempi di Esecuzione: $O(|E|\ log|E|)$

- Inizializzazione + Inserimento vertici in coda: $O(|V|)$ poichè inizializziamo ed inseriamo tutti i vertici una volta
- While: $O(|V|)$ poiché ogni vertice viene estratto dalla coda esattamente una volta.
- Estrai vertice: $O(log|V|)$ per la necessità di mantenere la struttura di heap e riordinare gli elementi dopo l'estrazione
- Foreach: $O(|E|)$ perchè dobbiamo scorrere tutta la lista di adiacenza

Prim:

- Comincia da qualsiasi vertice del grafo
- Attraversa un nodo più volte
- Grafo deve essere connesso
- Più efficiente per grafi densi

Dijkstra (CMSU)

Prova di Correttezza algoritmo di Dijkstra

Definiamo $\delta(v)$ come il cammino minimo di un vertice v dalla sorgente s.
Lemma: Se $d[v] = \delta(v)$ per ogni vertice v, ad ogni punto dell'algoritmo di Dijkstra, allora lo sarà per il resto dell'algoritmo.
Dimostrazione: Se $d[v] < \delta(v)$ allora la condizione della procedura Relax() fallirebbe sempre.
Teorema: Definiamo $< v_1 = s, v_2, ..., v_k >$ la sequenza di vertici estratti dalla coda Q dall'algoritmo di Dijkstra. Quando un vertice v_i viene estratto da Q, allora $d[v] = \delta(v)$
Dimostrazione: L'affermazione è valida nel caso base $v_1 = s$ in quanto $d[s] = \delta(s) = 0$ per definizione. Assumiamo sia vero per i primi $k - 1$ vertici, ovvero assumiamo che quando essi vengono rimossi $d[v_i] = \delta(v_i)$. Prendiamo il vertice v_k al momento della rimozione dalla lista Q. Avremo che, per come lavora l'algoritmo di Dijkstra, $d[v_k] \leq d[v_j]$, $j = k + 1, \ldots, n$. Osserviamo che se il cammino minimo dalla sorgente al vertice v_k consiste di soli vertici del set di vertici eliminati $R = \{v_1, \ldots, v_{k-1}\}$, allora $d[v_k] = \delta(v_k)$.
Dimostrazione per Assurdo: Assumiamo che $d[v_k] < \delta(v_k)$, allora il suo cammino minimo coinvolge vertici del set $V - R$. Consideriamo il primo vertice di questo insieme v_q nel cammino dalla sorgente a v_k. Definiamo v_p il vertice prima di v_q nel cammino. Quest'ultimo viene eliminato dalla coda Q quando tutti i suoi archi sono rilassati, compreso l'arco che porta a v_q, e di conseguenza $d[v_q] = \delta(v_q)$. Dato che non ci sono archi a costo zero, $\delta(v_q) < \delta(v_k)$ e quindi $d[v_q] < d[v_k]$. Ma questo significa che v_k non può essere stato scelto prima di v_q dall'algoritmo di Dijkstra, contraddicendo la scelta di v_k come vertice per cui $d[v_k] < \delta(v_k)$ quando eliminato da Q

Confronto tempi di calcolo Dijkstra e Floyd Warshall

IDijkstra calcola i cammini minimi partendo da un'unica sorgente, impiegando un tempo di $O(|V| \, log|V|)$. Floyd Warshall calcola invece i cammini minimi per ogni sorgente e verso ogni destinazione, impiegando $\theta(|V|^3)$. Se dovessimo modificare Dijkstra affinchè calcolasse tutt i percorsi, dovremmo eseguirlo per ogni vertice sorgente, ed otterremo come tempo $O(|V^2| \, log|V|)$, comunque meglio di Floyd-Warshall, ma con l'ipotesi di avere solo archi positivi. Possiamo allora concludere che:

- Meglio utilizzare Dijkstra per grafi poco sparsi, ovvero con molti meno archi rispetto a vertici, in quanto algoritmo Greedy ed usa coda.
- Meglio utilizzare Floyd Warshall per grafi molto sparsi, in quanto algoritmo di Dynamic Programming.

La **procedura di rilassamento** è un passo chiave per determinare i percorsi minimi. L'algoritmo di Dijkstra utilizza la procedura di rilassamento in ogni passo per determinare le distanze minime dai nodi di partenza a tutti gli altri nodi nel grafo. A ogni iterazione, si seleziona il nodo con la distanza minima, si rilassano gli archi adiacenti, e si aggiorna la

stima di distanza. Questo processo continua fino a quando tutti i nodi sono stati visitati. Nello specifico, il rilassamento, esamina se il percorso attraverso il nodo corrente è più breve del percorso precedentemente calcolato. Se è così, si aggiorna la stima di distanza per quel nodo con la somma delle distanze dal nodo di partenza al nodo selezionato e dall'arco tra il nodo selezionato e il nodo adiacente. In codice:

```
Unset
Relax(u,v, w)
if d(v) > d(u) + w(u, v)
       d(v) = d(u) + w(u,v)
       π(v) = u
```