

STRUTTURA DEL CORSO

Si suddivide in due parti:

- paradigma di progettazione di algoritmi (problemi **TRATTABILI**)
- Problemi computazionalmente difficili (**INTRATTABILI**)

I problemi visti ad ASD sono tutti (o quasi) trattabili.

I problemi **TRATTABILI** portano generalmente a **SOLUZIONE ESTATE**.

Le principali tecniche di progettazione sono:

- **PROGRAMMAZIONE DINAMICA**
- **TECNICHE GREEDY**

Non sono efficaci su problemi **INTRATTABILI**

PROBLEMI FACILI

Dato un problema, se esiste un algoritmo che lo risolve in tempo almeno **POLINOMIALE** sono detti **FACILI**

PROBLEMI INTRATTABILI

I problemi **INTRATTABILI** sono risolvibili in tempo almeno **ESPOENZIALE**

Per risolvere questi, se input reale, sfruttano 2 tipi di algoritmi:

● ALGORITMI DI APPROXIMAZIONE

● APPROCCI EURISTICI

SOLUZIONE DI UN PROBLEMA

Puoi essere decomposta in 4 fasi:

① FORMALIZZAZIONE

- Quali sono i **PARAMETRI** e qual è la loro forma
- Che forma ha la **SOLUZIONE** (albero, sequenza...)

② DEFINIZIONE DI UNA SOLUZIONE ALGORITMICA

③ VALIDAZIONE (CORRETEZZA)

Questa si può portare avanti in parallelo alla fase 2

④ STUDIO DELLE CARATTERISTICHE COMPUTAZIONALI

Qui sceglio le strutture dati ottimali

L'implementazione segue queste 4 fasi

PROGRAMMAZIONE DINAMICA

la metodologia che usiamo per progettare algoritmi che sfruttano la

PROGRAMMAZIONE DINAMICA è un'estensione del **DIVIDE ET**

IMPERA.

Il paradigma divide et impera sfrutta la decomposizione **RICORSIVA** (sottoproblemi della stessa natura), questo tipo di paradigma può

generare una SOVRAPPOSIZIONE DEI SOTTOPROBLEMI

OSS

Il paradigma di DECOMPOSIZIONE (non riconosciuta) scinde il problema in sottoproblemi che hanno natura differente quindi non incontra problemi di sovrapposizione

SOVRAPPOSIZIONE DEI SOTTOPROBLEMI

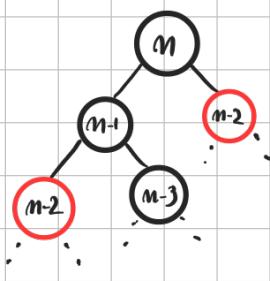
l'algoritmo di quick sort **NON** incontra questo problema perché il suo albero di ricchezza è ben pianto e per come è definito non abbiamo alcuna sovrapposizione (che implica una ridondanza)

Questo non vale per il calcolo della sequenza di Fibonacci

ANALISI DELL'ALGORITMO DI FIBONACCI

Scriviamo l'equazione e l'albero di ricchezza

$$F(n) = \begin{cases} 1 & n=1, n=2 \\ F(n-1) + F(n-2) & \text{altri casi} \end{cases}$$



Si puo' facilmente notare che abbiamo 2 istanze identiche con input $n-2$, e se continuano ci saranno altre ripetizioni, il nostro algoritmo le calcolera' ugualmente.

Se vogliamo scrivere un'equazione di ricorrenza del tempo di esecuzione sarà:

$$T_F(n) = \begin{cases} \Theta(1) & n=1, n=2 \\ T_F(n-1) + T_F(n-2) + \Theta(1) & \text{altrimenti} \end{cases}$$

Questo è uguale alla funzione di Fibonacci, che calcola le foglie dell'albero di ricorrenza.

Quindi $T_F = \Omega(F_{10}(n))$

Sappiamo che $F_{10}(n) = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right] \underset{n \rightarrow \infty}{\approx} \left(\frac{1+\sqrt{5}}{2} \right)^n$

Quindi $T_{F10}(n) = \Omega \left(\left(\frac{1+\sqrt{5}}{2} \right)^n \right)$

Questo tempo puo' essere ridotto a "tempo lineare" grazie alla

PROGRAMMAZIONE DINAMICA

OSS

I problemi diversi per il calcolo di $F_{IB}(n)$ sono $n-1$ ($F_{IB}(1), \dots, F_{IB}(n-1)$)

Possiamo "sviluppare" le soluzioni dei sotto problemi in un array F di lunghezza n dove $F[i] = F_{IB}(i)$

ALGORITMO BASE FIBONACCI

Implementiamo l'equazione di ricchezza

$FIB(i)$

$ris = 0$

IF $i=0$ OR $i=2$ THEN

$ris = 1$

ELSE

$ris = FIB(i-1) + FIB(i-2)$

RETURN ris

Come abbiamo visto la complessità di quest'algoritmo
è esponenziale $(\mathcal{O}((\frac{1+\sqrt{5}}{2})^n))$

Implementiamo ora l'"array dei sottoproblemi"

ALGORITMO OTIMIZZATO CON ARRAY

L'array avrà in ogni cella -1 se non è stata ancora calcolata la soluzione, altrimenti: $F[i] = F_{IB}(i)$

Questa è una proprietà **INVARIANTE** dell'array.

Inizializziamo l'array con $F(0) = 0$ e $F(i) = -1 \quad \forall i \quad 0 < i \leq m$

$FIB(i)$

IF $F[i] = -1$ THEN

se non è mai stato visitato

IF $i=1$ OR $i=2$ THEN

calcola la funzione

$nis = 1$

ELSE

$nis = FIB(i-1) + FIB(i-2)$

estrazione a tempo costante

$F[i] = nis$

aggiornamento dell'array

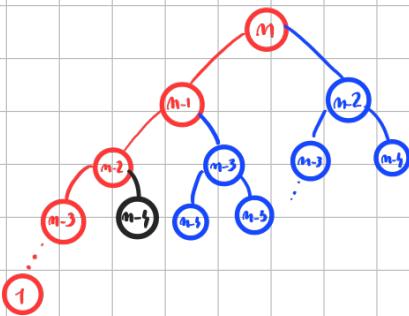
ELSE

$nis = F(i)$

RETURN nis

SCHEMATIZZAZIONE DEL PROBLEMA

Ricordiamo che l'albero di ricchezze e'



L'array inizializzato e':

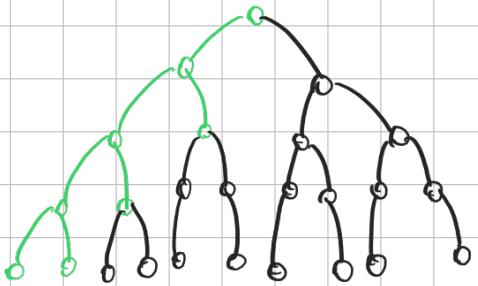
	0	1	2	m
F	0	-1	-1	-1

L'algoritmo costruisce ricorsivamente il percorso **rosso** fino al caso base ($F_{10}(1)$) assegnando $F[1]=1$ a tempo costante

Chiudendo le chiamate in risulta, effettueremo chiamate di sovrapposizione (blu) che, essendo già calcolate, richiederanno solo un accesso a tempo costante a $F[i]$ (perché $F[i] \neq -1$)

In definitiva, le uniche chiamate che verranno calcolate saranno quelle indicate in verde nel prossimo schema.

Il resto sono tutte sovrapposizioni.



OTTIMIZZAZIONE DINAMICA

Si puo' notare che $F[i] = F[i-1] + F[i-2]$

E' intuitivo quindi effettuare chiamate ricorsive se e solo se $F[i-1]$ e $F[i-2]$ sono già stati calcolati.

Questo dev' essere garantito dall'algoritmo che deve riempire in modo crescente l'array F .

FIBONACCI DINAMICO

L'algoritmo riempie l'array iterativamente partendo da $F[0]$

$\text{FIB}(m)$

$$F[0]=0 \quad \Theta(1)$$

$$F[1]=F[2]=1 \quad \Theta(1)$$

FOR $i=3$ TO m DO

$$F[i]=F[i-1]+F[i-2]$$

RETURN $F[m]$

}

$\Theta(m)$

$$T(m)=\Theta(m)$$

$\Theta(1)$

METODO DI OTTIMIZZAZIONE

La procedura di ottimizzazione che abbiamo utilizzato e'

EQUAZIONE → ALGO → ALGO BINAMICO

Tuttavia di solito la parte piu' complessa viene prima

DEFINIZIONE DEL PROBLEMA → ^{PROBLEM SOLVING} EQUAZIONE

Abbiamo "saltato" questo passaggio conoscendo già l'equazione di Fibonacci.

Non ci concentreremo su questa parte del processo di ottimizzazione.

PROBLEMA DEL TAGLIO OTTIMO DEL TRONCO

Abbiamo un tronco di lunghezza L .

Vogliamo produrre pezzi di legno, immaginiamo di avere un'unità di taglio. Possiamo quindi tagliare il tronco in tagli unitari o tagli di lunghezza $2 \cdot 1, 3 \cdot 1$ etc... (con 1 unità)

Questi tagli hanno costi distinti, diciamo che

$$\text{lunghezza} \xrightarrow{1 \leq d \leq L} c_d \xleftarrow{\text{costo}}$$

Vogliamo maximizzare il guadagno.

Per esempio, se tagliamo il tronco in pezzi unitari avremo

$$c(L) = \sum_{i=1}^L c_i$$

Se il tronco ha lunghezza 7 e dividiamo in due pezzi 4+3

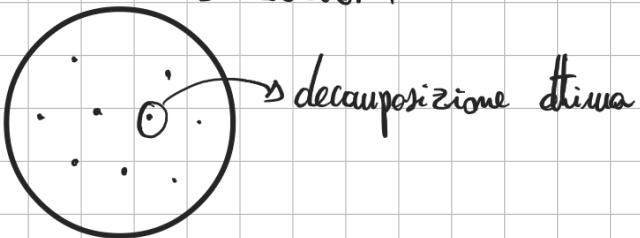
$$c'(L) = c_3 + c_4$$

Potremmo avere però che $c' \neq c$ per via dei singoli costi, noi vogliamo trovare il migliore finito L

ANALISI DEL PROBLEMA

Dobbiamo anzitutto capire quante sono le possibili decomposizioni fissato L

SOLUZIONI



Potremo (come faremo con gli algoritmi di ordinamento) generare tutte le soluzioni (decomposizioni) e trovare poi l'ottimale.

Se l'insieme delle soluzioni è finito, l'algoritmo termina senza problemi

Il costo di quest'algoritmo di **FORZA BRUTA** sarà dunque tanto quante sono le soluzioni. Se queste crescono esponenzialmente, il costo di questo calcolo crescerà esponenzialmente.

RAPPRESENTAZIONE DELLE SOLUZIONI

Possiamo immaginare il tronco come una sequenza di $L-1$ bit



Se l' i -esimo bit è 0 significa che **NON** effettuiamo un taglio in quel punto, altrimenti sarà 1 ovviamente.

Es.

Un tronco di lunghezza 7 tagliato 4+3 sarà così rappresentato.



0 0 0 1 0 0

(l'ultimo non serve)

Questo significa che ogni soluzione è rappresentata dalle sequenze di bit.

Ci sono 2^{L-1} possibili sequenze di $L-1$ bit.

Quindi l'algoritmo di forza bruta ha costo $\mathcal{O}(2^{L-1})$

Ottimizziamo la ricerca della soluzione ottimale grazie alla

DECOMPOSIZIONE IN SOTOPROBLEMI

SOLUZIONI AMMISSIBILI

Dobbiamo anzitutto capire che forme hanno le soluzioni.

Sia $P(L)$ il nostro problema, una soluzione avrà una forma generale

$$SOL(L) = \{l_1, l_2, \dots, l_k\}$$

Dove gli l_i sono i tagli rappresentati come numeri (lunghezza del tronco) e $SOL(l)$ è quindi una sequenza.

In particolare

$$\forall i \in [k] \quad 1 \leq l_i \leq L \quad \text{e} \quad \sum_{i=1}^k l_i = L$$

Questi sono i nostri **VINCOLI DI AMMISSIBILITÀ** sulla soluzione.

Questa è la definizione di una **SOLUZIONE AMMISIBILE** per $P(L)$

Si noti che dipendono strettamente dall'input L .

DECOMPOSIZIONE RICORSIVA DEL PROBLEMA

la decomposizione più semplice possibile è del tipo $\underline{l_1; (l_2, \dots, l_k)}$

avendo effettuando un taglio e isolandolo dal resto del tronco.

Se $\underline{l_1 \dots l_k}$ è soluzione di $P(L)$, $\underline{l_2 \dots l_k}$ potrebbe essere soluzione ammissibile di $P(L - l_1)$

Quindi il problema $P(L)$ potrebbe essere dato da $P(L - l_1) \cdot l_1$

$$SOL(L) = \underbrace{\underline{l_1 \dots l_k}}_{P(L)} = \underline{l_1} \cdot \underbrace{SOL(L - l_1)}_{P(L - l_1)}$$

Vogliamo dimostrare ciò.

OSS

Il nostro obiettivo è trovare un modo qualsiasi valido per ridurre il problema in sottoproblemi elementari che però portino a una soluzione valida.

PROPRIETA' ①

$SOL(L) = l_1 \cdot l_2 \cdot \dots \cdot l_k$ è ammmissibile per $P(L) \Rightarrow$

$\Rightarrow SOL(L - l_1) = l_2 \cdot \dots \cdot l_k$ è soluzione ammmissibile per $P(L - l_1)$

OSS

Non basta questa proprietà a dimostrare la nostra ipotesi perché stiamo banalmente dicendo che la struttura della soluzione resta valida, tuttavia non ci dà informazioni rispetto il problema principale. Dimostreremo una seconda proprietà che ci dà appunto informazioni in questo verso.

DIMOSTRAZIONE ①

Verifichiamo che siano verificati i vincoli di ammissibilità:

$$\bullet \sum_{i=1}^k l_i = L$$

$$L = l_1 + \sum_{i=2}^k l_i \quad (\text{per ipotesi}) \Rightarrow \sum_{i=2}^k l_i = L - l_1 = l_2 \dots l_k$$

Quindi essendo il membro sinistro vero per ipotesi, $l_2 \dots l_k$ rispetta il vincolo per $P(L - l_1)$

- $1 \leq l_i \leq L - l_1 \quad \forall i: 1 \leq i \leq k$

Poniamo $l_1 \leq L - l_1$ perché il nostro problema attuale non considera l_1 .

① $1 \leq l_2 \dots l_k$ per ipotesi

② Abbiamo dimostrato che $\sum_{i=2}^k l_i = L - l_1$, ma quindi non puo' esistere $l_i \geq L - l_1$ perch' $l_i > 0$ per cause li abbiamo scelti.

Quindi: ① + ② = $1 \leq l_2 \dots l_k \leq L - l_1$

Abbiamo dunque dimostrato che $SOL(L) = l_1, l_2 \dots l_k \Rightarrow SOL(L - l_1) = l_2 \dots l_k$

Questo vale per ogni problema ovviamente.

Dobbiamo dimostrare ora "l'altro verso" per poter essere sicuri di poter risolvere il problema in maniera ricorsiva.

Ricordiamo che le condizioni di ommissibilità delle soluzioni per $P(l)$: $S(l) = l_1, l_2, \dots, l_k$ con

$$\left. \begin{array}{l} \textcircled{1} \forall 1 \leq i \leq k \quad l_i > 0 \\ \textcircled{2} \sum_{i=1}^k l_i = L \end{array} \right\} \begin{array}{l} 1 \leq l_i \leq L \\ \forall i \in \{1, \dots, k\} \end{array}$$

E abbiamo dimostrato che, fissata una decomposizione :

Se $S(l)$ è ammmissibile per $P(l) \Rightarrow$

$\Rightarrow S(l - l_1)$ è ammmissibile per $P(l - l_1)$

Dovremo dimostrare tuttavia l'inverso di questo teorema:

PROPRIETÀ ②

Se

- $S(l) = \underline{l_1, l_2, \dots, l_k}$ (nel nostro caso) è ammmissibile per $P(l)$

- $S'(l - l_1) = \underline{l'_1, l'_2, \dots, l'_z}$ ammmissibile per $P(l - l_1)$

allora $\underline{l_1, l'_2, l'_3, \dots, l'_z}$ è ammmissibile per $P(l)$

DIMOSTRAZIONE ②

Le nostre ipotesi sono le seguenti

$$\textcircled{1} \quad l'_1 + l'_2 + \dots + l'_z = L - l_1$$

$$\textcircled{2} \quad \forall_{1 \leq i \leq z} l'_i > 0$$

Dimostriamo i vincoli di ammissibilità:

- $\forall 1 \leq i \leq z \quad l_i > 0$

ma questo è banalmente verificato perché $l_i > 0$ (e soluzione ammessa) e abbiamo l'ipotesi $\textcircled{2}$

- $l_1 + l_2' + l_3' + \dots + l_z' = L$

È banalmente verificata perché

$$l_1 + l_2' + l_3' + \dots + l_z' = L - l_1 + l_1 \rightarrow l_1 + l_2' + l_3' + \dots + l_z' = L$$

che è proprio il vincolo $\textcircled{2}$

Sappiamo quindi che trovata una decomposizione ben posta del problema, la soluzione di questo sotto problema è soluzione del problema. Tuttavia non vogliamo la soluzione **OTTIMA**

SOLUZIONE OTTIMA

Definiamo la soluzione **OTTIMA** per $S(L) = l_1 \dots l_k$ come

$$C(S(L)) = \sum_{i=1}^k c(l_i)$$

sia massima tra le soluzioni ammissibili.

Dobbiamo capire se anche la funzione di costo è calcolabile analogamente alle soluzioni.

OSS

Indichiamo con $C(x)$ il costo della soluzione x ,
escremo $c(x)$ per indicare il costo unitario del pezzo x

TEOREMA ③

Se $S(l)$ è ottima per $P(l)$ $\Rightarrow S(l-l_1)$ è ottima per $P(l-l_1)$

DI MOSTRAZIONE

Per assurdo sia $S(l)$ ottima ma **NON** $S(l-l_1)$ (per $P(l-l_1)$)

Esiste quindi una soluzione $S'(l-l_1)$ che è ommissibile per $P(l-l_1)$ e $C(S'(l-l_1)) > C(S(l-l_1))$ (è ottima).

Per la proprietà ② $S'(l) = l_1 \cdot S'(l-l_1)$ è ommissibile per $P(l)$.

Abbiamo quindi 2 soluzioni ommissibili per $P(l)$ (ma è l'ipotesi).

Confrontiamo i costi di queste due

$$\bullet C(S(l)) = \sum_{i=2}^k c(l_i) + c(l_1) = C(S(l-l_1)) + c(l_1)$$

$$\bullet C(s'(l)) = \underline{c(l_1)} + \underline{c(s'(l-l_1))}$$

La differenza dei costi è determinata dai costi di $\underline{c(s'(l-l_1))}$ e $\underline{c(s(l-l_1))}$

Noi sappiamo che $\underline{c(s'(l-l_1))} > \underline{c(s(l-l_1))}$, il che implica
 $c(s'(l)) > c(s(l))$

ASSURDO!

Abbiamo trovato una soluzione "migliore" di quella ottima

Quello che possiamo dedurre è che la soluzione ottima contiene soluzioni ottime per alcuni sottoproblemi

Se conoscessimo l'oggetto l_1 , che induce il problema $P(l-l_1)$ con soluzione ottima potremmo conoscere la soluzione ottima di $P(l)$. Dobbiamo quindi cercare l_1 , che ottimizza $P(l-l_1)$.

Sappiamo che $l \leq l_1 < L$

$S(l)$ ottima, per quanto dimostrato, potrebbe essere:

$$l_1=1 \quad S_1(l) = 1 + S(l-1)$$

$$l_2=2 \quad S_2(l) = 2 + S(l-2)$$

soluzioni ottime

$$S_1(l) = l + S(0)$$

La soluzione ottima è $S_2(l)$ tale che

$$C(S_2(l)) \geq C(S_j(l)) \quad \forall 1 \leq j \leq L \quad (\text{max})$$

Studiamo la funzione di costo della soluzione ottima.

FUNZIONE DI COSTO OTTIMA

$$C(l) = \begin{cases} 0 & l=0 \\ \max_{1 \leq z \leq L} \{C(z) + C(l-z)\} & l>0 \end{cases}$$

La correttezza di quest'equazione è ben definita dall'ultima dimostrazione effettuata.

Scriviamo l'algoritmo che calcola $c(l)$.

ALGORITMO DI CALCOLO

Quest'algoritmo è la banale traduzione dell'equazione di ricurrenza come fatto nell'equazione di Fibonacci:

COSTO OTTIMO(l)

RIS = 0

IF $l \neq 0$

FOR $z=1$ TO L DO

$x = \text{COSTO OTTIMO}(L-z)$

IF $c(z)+x > \text{RIS}$

$\text{RIS} = c(z) + x$

Questo fissa calcola, al variazione di z , il massimo tra le selezioni.

aggiorna il max

RETURN RIS

Vediamo il costo di quest' algoritmo

$$T(L) = \begin{cases} \Theta(1) & L=0 \\ \Theta(1) + \sum_{z=1}^L (T(L-z) + \Theta(1)) & L>0 \end{cases}$$

rimuoviamo $\Theta(1)$ per semplificare

Studiamo il caso induuttivo notando che al varire di z L diminuisce costantemente. Leggerlo "al contrario" equivale a

$$T(L) = 1 + \sum_{z=0}^{L-1} T(z)$$

Scarponiamo il primo elemento otteneendo

$$T(L) = 1 + T(L-1) + \sum_{z=0}^{L-2} T(z)$$

ma mai sappiamo che $T(L-1) = 1 + \sum_{z=0}^{L-2} T(z)$

Sostituendo lo secondo nella prima per ottenerne

$$T(L) = 2T(L-1).$$

Possiamo riformulare l'equazione come segue:

$$T(L) = \begin{cases} 1 & L=0 \\ 2T(L-1) & L>0 \end{cases}$$

L'albero di ricchezza è un albero pieno di altezza L .

Si noti che, come per Fibonacci, solo le foglie (caso base) appaiono costi.

Quindi il costo è dato dal numero di foglie che è benvenuto 2^L .

In definitiva $T(L) = \Omega(2^L)$

Il costo **NON** è dato da un problema intrinseco del problema, bensì dell'algoritmo, che calcola moltissime volte istante sovrapposte dei nostri $L+1$ problemi distinti, in particolare risolve $2^{L+1} - 1$ problemi (numero di modi).

Implementiamo quindi una soluzione dinamica

SOLUZIONE DINAMICA

Abbiamo il nostro arco di lunghezza L dove salvare i costi ottimi le celle conteniamo:

$$C(L) = \begin{cases} 0 & L=0 \\ \max_{1 \leq z \leq L} \{C(z) + C(L-z)\} & L > 0 \end{cases}$$



$$C(x) = \max_{1 \leq z \leq x} \{C(z) + C(x-z)\}$$

Per ogni elemento x si riconoscono tutti i valori precedenti per essere calcolato il suo costo (per via di $C(x-z)$).

COSTO_OTTIMO_DINAMICO (L)

$$C[0] = 0$$

FOR $x = 1$ TO L DO

$$\text{COSTO} = -1$$

FOR $z = 1$ TO x DO

$$y = C(z) + C[x-z]$$

IF $y > \text{COSTO}$

$$\text{COSTO} = y$$

$$C[x] = \text{COSTO}$$

RETURN C

$$T(L) = \sum_{x=1}^L \sum_{z=1}^x \Theta(1) = \Theta\left(\sum_{x=1}^L \sum_{z=1}^x 1\right) = \Theta\left(\sum_{x=1}^L x\right) \approx \Theta(L^2)$$

↖ somma dei primi L numeri

ALGORITMO DI SOLUZIONE OTTIMA

Abbiamo trovato il costo ottimo, non la soluzione ottima.

Quello che però possiamo fare è salvare tutte le prime scelte (per ogni istanza) che massimizzano il costo



$$S[0] = l_1$$

$$S[L-l_1] = l_2$$

$$S[L-l_1 - l_2] = l_3$$

Quindi l'array S solva il max delle istanze, l'array S solva l'argmax delle istanze

Per come è costruito S , la soluzione sarà la stampa di S dall'ultimo elemento fino al primo (potremmo implementare S come una queue).

L'algoritmo finale sarà

TAGLIO OTTIMO (L)

$C[0] = 0$

$S[0] = -1$

FOR $x=0$ TO L DO

$\text{COSTO} = -1$

$S = 0$

Anay dei primi tagli che ottimizzano le sottosequenze

parametro che salva l'angolo x

FOR $z=1$ TO x DO

$y = C[z] + C[x-z]$

IF $y > \text{COSTO}$

$\text{COSTO} = y$

$S = z$

z ottimizza C

$C[x] = \text{COSTO}$

$S[x] = S$

$\text{SOL}(L, S)$

l'algoritmo $\text{SOL}(L, S)$ e' la codifica di

$$\text{SOL}(L) = L \cdot S(L-L_1)$$

$SOL(L, S)$

IF $L > 0$

PRINT $S[L]$

$SOL(L - S[L], S)$

}

$O(L)$

$L - S[L]$ per "soltare" all'istanza corretta

OSS

Si noti che alcune soluzioni non verranno mai usate a seguito
del primo taglio del sottoproblema attuale, visto che il decremento
è $L - l_1$, non $L - 1$

l' algoritmo definitivo per il taglio ottimo del tronco ha complessità $\Theta(l^2)$.

Averemo scelto come decomposizione ricorsiva della soluzione:

$$SOL(l) = l_1 \underline{l_2 \dots l_K} = l_1 \cdot SOL(\underline{l - l_1})$$

Non è l'unica decomposizione ammessa e ogni decomposizione ha determinate proprietà.

Se sceglieremo le soluzioni:

$$S_1 = \{2 \mid 4 \mid 3\}$$

$$S_2 = 2 \mid 4 \mid 3$$

$$S_3 = 1 \mid 2 \mid 3 \mid 4$$

Sono bandieramente equivalenti perché producono lo stesso costo, c'è quindi una ridondanza che può influire sulla ricerca della soluzione ottima.

RIDUZIONE DEL CAMPO DI RICERCA DELLE SOLUZIONI

Proviamo a rappresentare tutte le soluzioni equivalenti da una sola soluzione, così da ridurre lo spazio di ricerca.

Prendiamo S_3 come rappresentante che è monotona non decrescente.
 le soluzioni di una "stessa classe" rispettano i vincoli di base per
 essere soluzioni + i vincoli propri della classe.

Per S_3 avremo (tenendo presente la forma delle soluzioni):

$$\textcircled{1} \quad 1 \leq l_i \leq L \quad \forall 1 \leq i \leq k$$

$$\textcircled{2} \quad \sum_{i=1}^k l_i = L$$

$$\textcircled{3} \quad l_i \leq l_{i+1} \quad \forall 1 \leq i \leq k \quad (\text{monotonia})$$

Possiamo riformulare i vincoli notando che il $\textcircled{3}$ permette di rilassare l' $\textcircled{1}$ (se il primo è il più piccolo e tutti sono maggiori del primo,
 allora basta che $l_i \geq 1$ per rispettare $\textcircled{1}$)

$$\textcircled{1} \quad l_1 \geq 1$$

$$\textcircled{2} \quad l_i \leq l_{i+1} \quad \forall 1 \leq i \leq k$$

$$\textcircled{3} \quad \sum_{i=1}^k l_i = L$$

OSS

Aggiungendo il vincolo $\textcircled{3}$ la decomposizione ricorsiva **NON** è più
 valida

DIMOSTRAZIONE

La prima proprietà dimostrata era

$$S(l) = l_1 l_2 \dots l_k \Rightarrow S(l-l_1) = l_2 \dots l_k$$

Seguendo la dimostrazione originale non avremo problemi perché la monotonicità è preservata

$$1 \leq l_1 \leq l_2 \leq \dots \leq l_k$$



③ è dato da $\sum_{i=2}^k l_i = l - l_1$

Tuttavia avremo problemi col dimostrare il teorema della sottostruzione ottima (la soluzione ottimale contiene la sol. ottimale per i sottoproblemi) perché il teorema ② (inverso di ①) non è verificato.

Sceglieremo una soluzione ammissibile casuale per $P(l-l_1)$

$$S(l-l_1) = l'_2 \dots l'_k$$

Dobbiamo ricostruire $S(l)$.

Sappiamo per ipotesi che

$$\bullet l'_2 \geq 1$$

$$\bullet l'_i \leq l'_{i+1} \quad \forall 1 \leq i \leq k$$

Prendiamo un qualsiasi l_i che rispetti i vincoli di $P(l)$ per costruire $S(l)$ ovvero che sia ≥ 1 .

$l_1 l'_2 \dots l'_k$ non sappiamo se è ammissibile perché non abbiamo

informazioni sul rapporto tra l_1 e l_i

Quindi il teorema **NON** è verificato.

La nostra decomposizione deve garantire sempre i vincoli

CORREZIONE DELLA SOLUZIONE PER S_3

Il sottoproblema quando fa le sue scelte non tiene conto dei volati precedenti, non garantendo la monotonia.

$S(l)$ deve comunicare a $S(l-l_i)$ la lunghezza del tronco rimanente e il limite inferiore per la lunghezza dei pezzi.

Aggiungiamo un nuovo parametro alla soluzione:

$$S(l, 1) = \underbrace{l_1, l_2, \dots, l_k}_{P(l)} = l_1 S(l - l_1, l_1)$$

Si noti che per $P(l, 1)$

$$S(l, 1)$$

$$\textcircled{1} \quad l_1 \geq 1$$

$$\textcircled{2} \quad l_i \leq l_{i+1} \quad \forall 1 \leq i \leq k$$

$$\textcircled{3} \quad \sum_{i=1}^k l_i = l$$

Per i sottoproblemi ora si aggiungeranno due parametri evidenziati.

Possiamo ora dimostrare il Teorema 2

DIMOSTRAZIONE

Per ipotesi $S(L-l_1, l_1) = l_2' \dots l_k'$ rispetterà

$$\textcircled{1} \quad l_2' \geq l_1$$

$$\textcircled{2} \quad l_i' \leq l_{i+1}' \quad \forall 1 \leq i \leq k$$

$$\textcircled{3} \quad \sum_{i=2}^k l_i' = L - l_1$$

Verifichiamo che qualsiasi l_i ammesso sia ammesso sfruttando i vincoli di $P(L-l_1)$

$$l_1, l_2' \dots l_k'$$

$$\textcircled{1} \quad l_1 \geq 1$$

$$\textcircled{2} \quad l_i \leq l_{i+1} \quad \forall 1 \leq i \leq k \quad \wedge \quad \underbrace{l_1 \leq l_2'}_{\textcircled{1}}$$

$$\textcircled{3} \quad \textcircled{3} + l_1 = L$$

Quindi il Teorema 2 è verificato

OSS

Di solito viene invalidato il teorema 2 se le soluzioni non sono ben definite

Grazie a ciò che abbiamo dimostrato, il teorema ③ è ora verificato

DI MOSTRAZIONE

Sia $S(l, l) = l_1, \underline{l_2 \dots l_k}$ ottima per $P(l, l) \Rightarrow l_2 \dots l_k$ è ottima per $P(l - l_1, l_1)$

Per assurdo sia $l_2 \dots l_k$ non ottima per $P(l - l_1, l_1)$.

Eステ quindi $S'(l - l_1, l_1) = l'_1 \dots \underline{l'_k}$ ammmissibile |

$$(S'(l - l_1, l_1)) > C(S(l - l_1, l_1))$$

Per il teorema 2 $S'(l, l) = l_1, \underline{l'_2 \dots l'_k}$ è ammmissibile per $P(l, l)$

$S'(l, l)$ e $S(l, l)$ si distinguono a partire dal secondo taglio

Noi sappiamo però che $(S'(l - l_1, l_1)) > C(S(l - l_1, l_1))$, quindi

$(S'(l, l) > S(l, l))$ ASSURDO! perché $S(l, l)$ è ottimale per ipotesi

ALGORITMO DI CALCOLO PER S_3

Scriviamo l'algoritmo ricorsivo, partendo dalla funzione che descrive il problema ricorsivamente ricordando che

$$(S(l, 1)) = c(l_1) + (S(l - l_1, l_1))$$

Se provassimo a definire la funzione con due parametri avremmo:

$$C(l, x) = \begin{cases} 0 & l=0 \\ \max_{\substack{x \leq l \\ y \leq l}} \{c(y) + C(l-y, y)\} & l>0 \end{cases}$$

max lunghezza

Questa definizione è molto poco efficiente, anzitutto per la sovrapposizione dei problemi:
le istanze saranno diverse al variare di $l \cdot x$.

Quindi le istanze diverse sono $l \cdot x$

l può assumere $L+1$ valori, x ha L valori, quindi i sottoproblemi distinti sono $(L+1)L = L^2 + L$.

L'algoritmo finale che vediamo aveva tempo L^2 perché avevamo $L+1$ sottoproblemi (non avevamo il secondo parametro) e il calcolo di ciascuno di questi dipendeva da un numero lineare di sottoproblemi.

Adesso abbiamo un numero di sottoproblemi quadratico per un tempo di calcolo lineare, avendo un tempo cubico.

Questo problema e' intrinseco della decomposizione.

DECOMPOSIZIONE ALTERNATIVA

Sia $S(L,1) = l_1, l_2, \dots, l_k$ una soluzione per $P(L,1)$

Possiamo vedere questa come una soluzione per due problemi distinti:

$$\begin{cases} S(L,1) = 1 \cdot S(L-l_1, 1) & \text{se } l_1 = 1 \\ S(L,1) = S(L,2) & \text{se } l_1 > 1 \end{cases}$$

Se l_1 e' il minimo valore, il sottoproblema ammetterà ancora $l_2=1$ come taglio.

Se $l_1 > 1$, l_1, l_2, \dots, l_k e' soluzione di $P(L,2)$ perch \circ l_1 rispetta il vincolo e la sequenza e' monotona.

In questo secondo caso il taglio **NON** e' effettuato, bens \circ viene delegato al prossimo sottoproblema.

Ridimostriamo i 3 Teoremi in maniera coincisa per entrambe le possibili decomposizioni

DIMOSTRAZIONE

● TEOREMA ①

► $l_1 = l$

(Già dimostrato in precedenza (situazione analoga))

► $l_1 > l$

E' banalmente $SOL(l, 2)$ perché l_1 già rispetta il vincolo su 2, per il resto la soluzione è monotona.

● TEOREMA ②

► $l_1 = l$

Prendendo una qualsiasi $SOL(l-1, 1)$ (che è corretta solo se $l_1 = 1$) la possiamo comporre con $l_1 = 1$ (per forza) e avremo una $SOL(l, 1)$ che verifica i vincoli di ommissibilità (1 è maggiore o uguale di 1, $l_2 \geq 1$ per ipotesi etc..)

► $l_1 > l$

Certamente una $SOL(l, 2)$ è anche $SOL(l, 1)$ perché la somma da 1, 2 ≥ 1 etc

● TEOREMA ③

Ricordiamo che il costo è dato in termini di somma che preserva le proprietà ottime in decomposizione e vale il teorema ② che garantisce la validità della decomposizione.

► $l_1 = 1$

La dimostrazione è analoga a quanto dimostrato prima con unico caso $l_1 = 1$.

Se $1 \cdot l_2 \dots l_k$ è ottima per $P(l, 1)$, $l_2 \dots l_k$ è ottima per $P(l-1, 2)$ altrimenti esisterebbe $l'_2 \dots l'_k$ ottimale ma allora $1 \cdot l'_2 \dots l'_k$ sarebbe ottimale **ASSURDO!**

► $l_1 > 1$

I costi delle due soluzioni sono identici, se $l_1, l_2 \dots l_k$ è ottima per $S(l, 1)$ non può non essere ottima per $S(l, 2)$.

Se per esempio non fosse così ci sarebbe una soluzione migliore per $S(l, 2)$ ma questa stessa è una soluzione ommissibile per $S(l, 1)$ e quindi abbiamo l'assurdo (prendendo sempre $l_1 > 1$ ovviamente)

COSTO DELLA DECOMPOSIZIONE

La funzione di costo sarà

$$C(l, x) = \begin{cases} 0 & l=0 \vee x>l \quad \text{serve per limitare questa soluzione} \\ \max \left\{ c(x) + C(l-x, x), C(l, x+1) \right\} & l>0 \wedge x \le l \end{cases}$$

$\underbrace{\phantom{\max \left\{ c(x) + C(l-x, x), C(l, x+1) \right\}}}_{l=1} \quad \underbrace{\phantom{\max \left\{ c(x) + C(l-x, x), C(l, x+1) \right\}}}_{l>1}$

Il calcolo adesso non ha un numero lineare di scelte
(come il calcolo del min) bensì ne ha solo 2

Quindi grazie alla programmazione dinamica, se solviamo prima
le due scelte, dopo ci basterà prendere a tempo costante il max.

Avendo due parametri utilizzeremo una matrice bidimensionale,
riportando il tempo a quadratico (per il calcolo)

ALGORITMO DI CALCOLO

Costruiamo la matrice, nelle colonne poniamo il primo parametro
(L) e sulle righe il secondo (x).

Tutte le celle in cui $x>l$ saranno 0 (vedi l'equazione)
quindi tutta la diagonale principale varrà 0.

$\ell \rightarrow$	1	2	$\bar{\ell}$		L	L+1
X	0	0			0	0
$\downarrow 1$						

\bar{x}

L 

e' il nostro
problema $P(L,1)$

Prendiamo $C[\bar{\ell}, \bar{x}]$ non caso base e calcoliamo il costo ottimo.

Il contenuto dipende da (vedendo l'equazione) $C[\bar{\ell} - \bar{x}, \bar{x}]$ e $C[\bar{\ell}, \bar{x} + 1]$.

$C[\bar{\ell} - \bar{x}, \bar{x}]$ si trova nella stessa colonna, in qualche riga superiore (parte tratteggiata) mentre invece $C[\bar{\ell}, \bar{x} + 1]$ è l'adiacente destro.

Per avere già disponibili i contenuti di queste due celle dobbiamo calcolare la matrice scorrendo sia le righe in ordine crescente e le colonne in ordine decrescente, escludendo il triangolo superiore che verrà inizializzato a 0.

Supponiamo $c(x)$ l'array dei costi

COSTO OTTIMO (l, l_c)

FOR $l=0$ TO N

Inizializziamo il triangolo superiore

FOR $x=l+1$ TO $L+1$

$$C[l, x] = 0$$

Generiamo il triangolo inferiore
mostra desiderato

FOR $l=1$ TO L

la riga 0 e' composta da soli 0

FOR $x=l$ TO 1

$$y = c(x) + C[l-x, x] \quad \text{caso } l>1$$

IF $y > C[l, x+1]$ prendiamo il max

$$\max = y$$

ELSE

$$\max = C[l, x+1]$$

$$C[l, x] = \max$$

L'algoritmo ha un costo $\Theta(l^2)$

La differenza principale tra quest'algoritmo e l'altro non e'
nel tempo bensì nello spazio occupato, che prima era lineare
(array) ed ora e' quadratico (matrice quadrata).

In questo caso specifico ridurre il campo di ricerca non ci ha aiutato realmente, questo **NON** vale sempre ovviamente.

La complicazione è data dal vino che non necessario (per le soluzioni) che abbiamo aggiunto (la monotonia)

Vi ponderato quanto si è disposti a pagare per ottimizzare il tempo o lo spazio di un algoritmo (sacrificando l'altro componente), che dipende dal guadagno e dalle esigenze

L'algoritmo che ricostruisce la sequenza sarà:

$SOL(l, x, C, c)$

IF $l > 0$ AND $x < l$ caso ricorsivo

IF $C(x) + C([l-x, x]) > C([l, x+1])$

$SOL(l-x, x)$

PRINT x

ELSE

$SOL(l, x+1)$

Il costo è lineare su L.

SCHEBULAZIONE DI INTERVALLI PESATI

Astnrae il problema di avere una serie di tasks che hanno un costo e un tempo d'esecuzione.

La nostra risorsa (esecutore) puo' eseguire una task alla volta.

DEFINIZIONE DEL PROBLEMA

Gli elementi a nostra disposizione sono:

- ATTIVITA'

$$T = \{t_1, \dots, t_n\}$$

- DURATA DELLE ATTIVITA'

$s_i \rightarrow f_i$ $\forall i \in M$, sono fissati nel tempo

- FUNZIONE DI COSTO

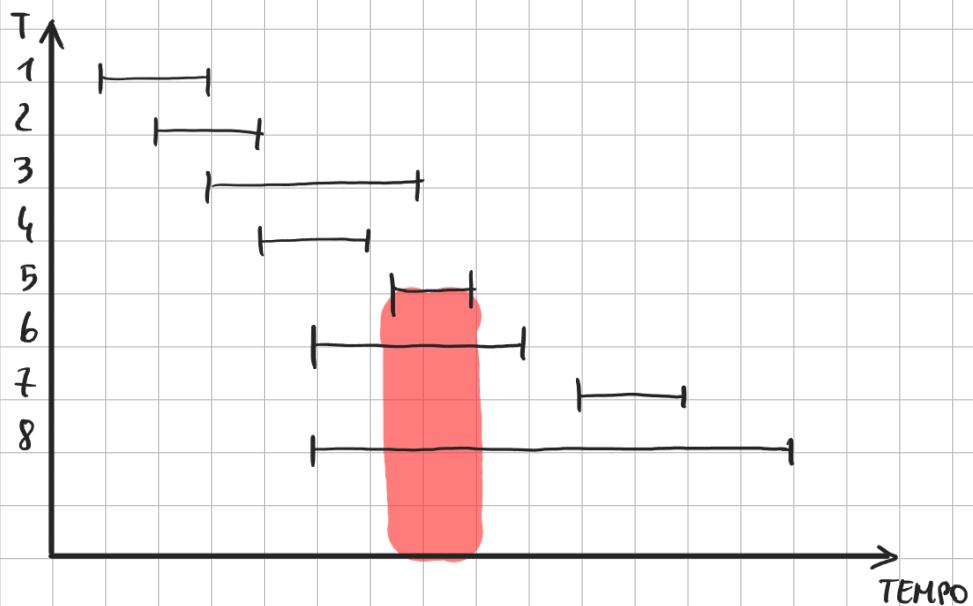
$c: T \rightarrow D$ De ordinato (esistono max e min)

Il nostro obiettivo e' calcolare l'insieme o sottinsieme
di attivita' COMPATIBILI che ne **HASSI HIZZA** la somma
dei costi

ATTIVITA' COMPATIBILI

Due attivita' sono **COMPATIBILI** se l'intersezione delle loro durate è vuota.

E.s.



$S_1 = \{3, 7, 1\}$ e' una soluzione ammessa (le attivita' sono tutte compatibili tra loro)

$S_2 = \{\underline{8}, \underline{1}, \underline{4}, \underline{5}\}$ NON e' ammessa perche' 5 e 8 si sovrappongono

SOLUZIONE AMMISSIBILE

$S \subseteq T$ e' soluzione **AMMISSIBILE** se

$$\forall x, y \in S \quad x \neq y \Rightarrow \left. \begin{array}{l} \textcircled{1} \quad s_x > f_y \\ \textcircled{2} \quad s_y > f_x \end{array} \right\} \text{COMPATIBILITÀ}$$

Per brevità possiamo definire la condizione di ammissibilità

come:

$$\forall x, y \quad x \neq y \Rightarrow x \cap y = \emptyset$$

Il **COSTO** di una soluzione ammissibile e' formalmente

$$C(S) = \sum_{x \in S} c(x)$$

OSS

Se ci sono due attività tali che $s_1 = s_2 \wedge f_1 = f_2$ possiamo escludere formalmente quella che produce il costo maggiore.

Excludiamo quindi questo caso.

① DECOMPOSIZIONE RICORSIVA DEL PROBLEMA

Dato un problema $P(T, c)$, la decomposizione punta a semplificare il problema e quindi ridurre $T \circ c$.

La nostra soluzione sarà:

$$S(T, c) = \{k_1, \dots, k_z\} \text{ tale che}$$

$$\textcircled{1} \quad \{k_1, \dots, k_z\} \in T$$

$$\textcircled{2} \quad \forall x, y \in S(T, c)$$

$$x \neq y \Rightarrow x \cap y = \emptyset$$

Si può notare che c non compare, e' in qualche modo immutabile e non caratterizza l'ammissibilità, quindi possiamo ripetere il problema come

$$P(T) \rightarrow S(T)$$

Potremo decomporre la soluzione come già facciamo, ovvero isolando il "primo" elemento e campionando.

$$S(T) = \{k_1, k_2, \dots, k_z\} = \{k_1\} \cup S(T')$$

$$\text{Con } T' = T \setminus \{k_1\} \quad \text{quindi } P(T) \rightarrow P(T \setminus \{k_1\})$$

induce

Noi dobbiamo dimostrare sempre i due teoremi (poi il terzo).

Il Teorema ① e' verificato, il Teorema ② **No**

DIMOSTRAZIONE

Il Teorema ② ci dice che prendendo una soluzione arbitraria per $P(T \setminus \{k_1\})$ e componevolo k_1 , avremo una soluzione ammessa per $P(T)$.

$$S(T \setminus \{k_1\}) = k'_2, \dots, k'_e \quad | \quad ① k'_2, \dots, k'_e \in T \setminus \{k_1\}$$

$$② \forall 2 \leq i < l \quad \forall i < j \leq l \quad (\text{due elementi distinti})$$

$$k_i \cap k_j = \emptyset$$

k_1, k'_2, \dots, k'_e (con $\{k_1\}$ ammesso per $S(T)$) dev' essere $S(T)$.

- Il vincolo ① e' banalmente verificato.
- Il vincolo ② **Now** e' garantito perch' il sottoproblema non sa cosa garantire rispetto al problema principale (compatibilita') perch' l'input e' $T \setminus \{k_1\}$ ma non conosce k_1 .

DECOMPOSIZIONE RICORSIVA DEL PROBLEMA

Dobbiamo ridurre l'insieme da "passare" al sotto problema, ovvero

T' dev'essere il sottinsieme di T dei compatibili con $\{k_1\}$

$$T' = \{t \in T \mid t \cap k_1 = \emptyset\}$$

In questo modo i tre teoremi sono verificati

DI MOSTRAZIONE

- ① E' verificato perché sia $S(T) = \{k_1, k_2, \dots, k_e\}$ per definizione sono tutti compatibili tra loro, rimuovendo k_1 quindi i vincoli di ammissibilità saranno ancora validi
- ② Sia $S(T') = k'_1, \dots, k'_e$, il primo vincolo e' banale ed ora anche il secondo e' verificato per le proprietà di T'
- ③ la dimostrazione del teorema e' sempre la stessa, per ostendo sfruttando il teorema ②

QUALITA' DELLA DECOMPOSIZIONE

L'algoritmo di calcolo di questa decomposizione presenta
some apposizioni, basta pensare al seguente esempio:

Sia $P(\{1,2,3\})$ potremo avere:

Scegliendo prima 2 ovvero $P(\{1,3\})$ scegliendo poi 1 $P(\{3\})$

Scegliendo prima 1 ovvero $P(\{2,3\})$ scegliendo poi 2 $P(\{3\})$

la programmazione dinamica ci permette di avere un'ottimizzazione
coerente con il numero di sottoproblemi distinti.

In questo caso l'input di un sottoproblema è un insieme,
quindi non dovreemo solvere **TUTTI** i sottinsiemi di T
(istanze distinte) ovvero $2^{|T|}$, quindi avremo tempo e spazio
esponenziale.

La programmazione dinamica qui non puo' nulla, quindi
(se il problema non e' computazionalmente complesso) dobbiamo
cercare un'altra decomposizione.

OSS

la decomposizione ricorsiva con input un insieme posteriore

SEMPRE a una complessità esponenziale.

DECOMPOSIZIONE ORDINATA

Spesso torna utile imporre un ordine sull'insieme, in questo caso ordiniamo rispetto ai tempi, rappresentando l'insieme come una sequenza ordinata.

$$T = \langle t_1, t_2, \dots, t_m \rangle \quad \forall 1 \leq i \leq m \quad f_{t_i} < f_{t_{i+1}}$$

Questa sequenza può essere generata in tempo d'ordinamento dell'input, quindi $\Theta(n \lg m)$

La soluzione sarà a sua volta ordinata, si noti che non ci serve più come input T , bensì l'ultima attività

I possibili sotto problemi di $P(m)$ sono $m-1$.

$$S(m) = \langle K_1, \dots, K_2 \rangle \mid \left. \begin{array}{l} \textcircled{1} \quad \forall 1 \leq i \leq 2 \quad f_{K_i} < f_{K_{i+1}} \\ \textcircled{2} \quad K_i \in \{t_1, \dots, t_m\} \end{array} \right\}$$

VINCOLI DI AMMISSIBILITÀ

condizione di compatibilità → $\textcircled{3} \quad \forall 1 \leq i \leq 2 \quad \forall 1 \leq j \leq 2 \quad K_i \cap K_j = \emptyset$

la condizione di compatibilità può essere ripetuta sfruttando la monotonia della sequenza.

$$\textcircled{3} \quad \forall_{1 \leq i < z} \quad h_{k_i} < s_{k_{i+1}}$$

Decomporremo ricorsivamente nel seguente modo:

$$S(m) = \langle k_1, \dots, k_{z-1}, k_z \rangle$$

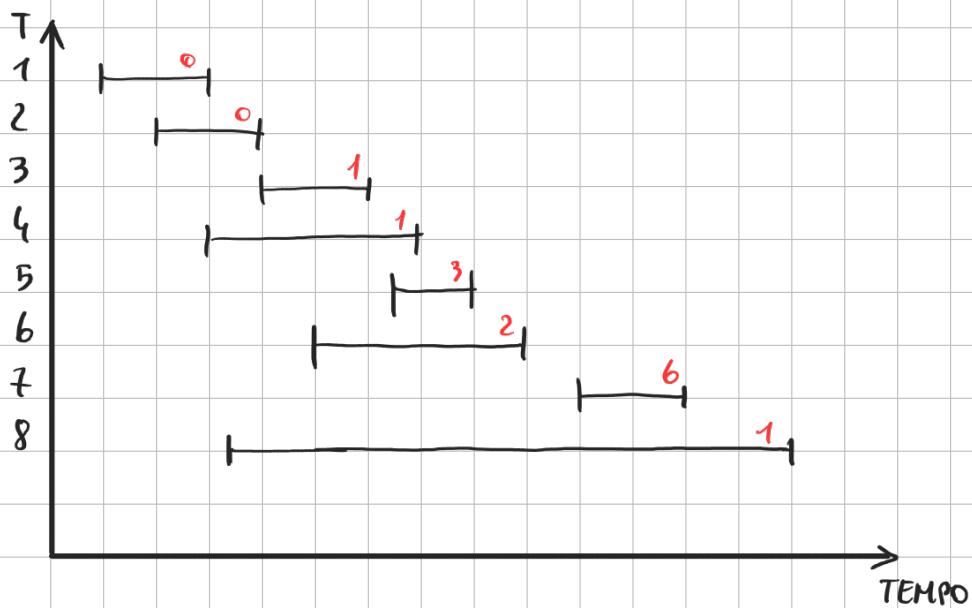
$$S(m') = \langle k_1, \dots, k_z \rangle \quad \text{teniamo tutti prima di } k_z \text{ per definizione}$$

$|S(m')| < |S(m)|$ perché ci dev'essere almeno k_z mancante quindi $m' < k_z$ ma non è detto che $m' = z-1$ per il teorema \textcircled{2}.
(potrebbe semplicemente non essere compatibile).

m' dev'essere l'ultima attività compatibile con k_z .

$$t_{m'} \cap K_z = \emptyset$$

Es.



Il problema iniziale è $P(8)$

$S(8)$ potrebbe essere $\langle \dots, 7 \rangle$, $m' = 6$ perché è il max compatibile

Se $S(8)$ fosse $\langle \dots, 8 \rangle$ m' sarebbe 1

Se $S(8)$ fosse $\langle \dots, 6 \rangle$ m' sarebbe 2

Chiamiamo $l: \{1, \dots, m\} \rightarrow \{0, \dots, m-1\}$ la funzione che fissato un intervallo restituisce m' .

$l(i) = \max \text{ indice } i \mid t_{l(i)} \text{ e' compatibile con } t_i$

DECOMPOSIZIONE FINALE

Sia data la funzione l , la decomposizione sarà:

$$S(m) = \langle k_1, \dots, k_z \rangle$$

$$\rightarrow k_z = m \Rightarrow S(m) = S(l(m)) \cdot m$$

$$\rightarrow k_z < m \Rightarrow S(m) = S(m-1)$$

Adesso il numero di sottoproblemi è polinomiale

RICAPITOLAZIONE

Dato T input, consideriamo il risultato dell' ordinamento sui tempi di terminazione (con costo $\Theta(n \lg n)$) su T .

$$T = \langle t_1, t_2, \dots, t_n \rangle \mid f_{t_i} < f_{t_{i+1}} \quad \forall 1 \leq i < n$$

Se $P(m)$ è il nostro problema iniziale, $P(i)$ sarà il problema di ottenere la soluzione ottima della sequenza $\langle t_1, \dots, t_i \rangle$

La soluzione al problema sarà una sequenza di indici si fatta.

$$S(n) = \langle i_1, i_2, \dots, i_k \rangle \mid \begin{array}{l} \textcircled{1} \quad 0 < i_1 < i_2 < \dots < i_k \leq n \\ \textcircled{2} \quad f_{i_j} < f_{i_{j+1}} \quad \forall 1 \leq j < k \end{array} \quad \begin{array}{l} \text{ORDINAMENTO} \\ \text{COMPATIBILITÀ} \end{array}$$

La decomposizione alla quale eravamo arrivati poste le precedenti ipotesi è la seguente:

Il problema $P(m)$ sceglie l'ultima task e passa al sotto problema $P(?)$. La sequenza passata in input a questo sotto problema dev'essere soluzione per la decomposizione ricorsiva.

Questa sequenza sarà (sia i_1, i_2, \dots, i_k soluzione di $P(n)$)

$$\textcircled{1} \quad i_1, i_2, \dots, i_{k-1} \quad \text{se } i_k = n$$

$$\textcircled{2} \quad i_1, i_2, \dots, i_k \quad \text{se } i_k \neq n \Rightarrow \text{induce } P(n-1)$$

Il caso ① sceglie come oggetto della soluzione l'ultimo disponibile. Non possiamo sapere che problema involge perché non conosciamo i valori precedenti a n .

Il caso ② è il suo complemento quindi il problema indotto dalla sua maa scelta è per forza $P(n-1)$.

Dobbiamo definire $P(?)$ (sottoproblema indotto dal caso ①).

Avevamo definito $l(i)$ funzione che restituisce la massima task compatibile con i . In questo modo possiamo passare al sottoproblema proprio $l(m)$, quindi $\textcircled{1} \Rightarrow P(l(m))$

Dimostriamo l'ammissibilità della decomposizione grazie ai 3 teoremi in entrambi i casi

DI MOSTRAZIONE ①

$S(m) = i_1 \dots i_k$ ammissibile per $P(m) \Rightarrow$

$\Rightarrow \textcircled{1} i_k = m \Rightarrow i_1 \dots i_{k-1}$ ammissibile per $P(l(m))$

$\textcircled{2} i_k \neq m \Rightarrow i_1 \dots i_k$ ammissibile per $P(n-1)$

E' banale in entrambi i casi:

② la sequenza $i_1 \dots i_k$ è ordinata e compatibile per ipotesi.

Se $i_k = m$ allora $i_k \leq m-1$.

Il primo vincolo di ammissibilità è sicuramente garantito quindi.

Il secondo è verificato per ipotesi.

① Abbassiamo la sequenza $i_1 \dots i_{k-1}$

1) $i_{k-1} < i_k = m$ e per ammissibilità di $P(m)$ $f_{i_{k-1}} < s_m$.

Tutti i precedenti di i_{k-1} terminano prima e quindi sono compatibili rispetto a m

$l(m)$ è da allora il max indice di un task compatibile con m ,

quindi varia per forza $i_{k-1} \leq l(m)$, quindi il primo vincolo è verificato

2) Il secondo vincolo è garantito per ipotesi (la sequenza già era compatibile)

DIMOSTRAZIONE ②

I due casi che vogliamo dimostrare sono:

① $\langle i'_1 \dots i'_z \rangle$ omm. per $P(l(m)) \Rightarrow \langle i'_1 \dots i'_{z'm} \rangle$ omm. per $P(m)$

② $\langle i'_1 \dots i'_z \rangle$ omm. per $P(m-1) \Rightarrow \langle i'_1 \dots i'_{z'} \rangle$ omm. per $P(m)$

② $i_2' \leq m-1$ per ipotesi, rispetta i vincoli di ommissibilità sono mantenuti banalmente, quindi è verificato

① I vincoli di ommissibilità ci dicono che:

$$1) 0 < i_1' < \dots < \underline{i_2' \leq l(m)}$$

$l(m) \neq m$ (m non è compatibile con se stesso), in particolare

$$\underline{l(m) < m} \quad e \quad f_{l(m)} < S_m$$

$$0 < i_1' < \dots < \underline{i_2' < m \leq m}$$

2) dobbiamo garantire l'ommissibilità tra i_2' e m , ma l'abbiamo appena dimostrato per transitività

DIMOSTRAZIONE ③

$$S(m) = \bar{i}_1, \bar{i}_2, \dots, \bar{i}_K, \quad C(S(m)) = \sum_{j=1}^K c(\bar{i}_j) \quad (\text{soluzione ottima})$$

$S(m)$ ottima per $P(m) \Rightarrow$ ① $\bar{i}_K = m \Rightarrow S(l(m)) = \bar{i}_1, \dots, \bar{i}_{K-1}$ ottima per $P(l(m))$

② $\bar{i}_K \neq m \Rightarrow S(m) = \bar{i}_1, \dots, \bar{i}_K$ ottima per $P(m-1)$

② P.a. i_1, \dots, i_K non ottima per $P(m-1)$.

$$\exists S'(m-1) = \bar{i}_1', \dots, \bar{i}_2' \text{ amm. per } P(m-1) \quad | \quad \underline{C(S'(m-1)) > C(S(m-1)) = C(S(m))}$$

$S'(m-1) = S'(m)$ (per ipotesi) ma $C(S'(m)) > C(S(m))$

ASSURDO!

$S'(m)$ è $S(m)$ oppure genera un assurdo.

① P.a.

$S(m) = i_1 \dots i_k$ ottima per $P(m)$ ma

$S(l_m) = i_1 \dots i_{k-1}$ non ottima per $P(l_m)$

$\exists \underline{S'(l_m)} = \underline{i'_1 \dots i'_z} \quad | \quad \underline{C(S'(l_m)) > C(S(l_m))}$

$S'(m) = i'_1 \dots i'_z m$ è omm. per $P(m)$ (per il Teorema ②)

$$C(S'(m)) = C(S'(l_m)) + c(m)$$

$$\Rightarrow C(S'(m)) > C(S(m))$$

$$C(S(m)) = \underbrace{C(S(l_m))}_{\text{ammisibile per il}} + c(m)$$

Teorema ②

ASSURDO!

FUNZIONE DI CALCOLO DEL COSTO DELLE SOLUZIONI

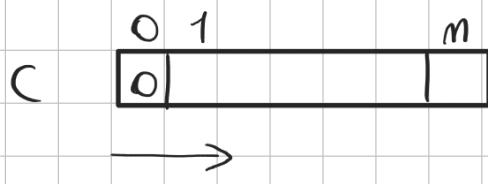
$$C(m) = \begin{cases} 0 & m=0 \\ \max \left\{ C(l_m) + c(m), C(m-1) \right\} & m \neq 0 \end{cases}$$

(1) (2)

E' facile notare che $C(l_m)$ presenta tutti i sottoproblemi di $C(m-1)$, quindi l'algoritmo naïf di calcolo dell'equazione sarebbe estremamente inefficiente, passiamo all'implementazione dinamica.

ALGORITMO DINAMICO DI CALCOLO

Istanziamo un array C di lunghezza $m+1$ che contiene i costi ottimi di tutte le istanze notando che $C[0] = 0$



Dobbiamo banalmente riempire in ordine crescente (per via di $C(m-1)$ nell'equazione)

Scriviamo gli algoritmi necessari per il calcolo della soluzione ottima

● COSTO OTTIMO (n)

$$C[0] = 0$$

FOR $i=1$ TO n DO

$$\text{CASO_1} = C[\ell(i)] + C(i)$$

$$\text{CASO_2} = C[i-1]$$

IF $\text{CASO_1} > \text{CASO_2}$

$$C[i] = \text{CASO_1}$$

ELSE

$$C[i] = \text{CASO_2}$$

● SOL-OTTIMA (i, c)

IF $i > 0$

i alla prima chiamata sarà m

caso induuttivo

IF $C[i] = C[i-1]$

CASO ② (non mettiamo i nella soluzione)

SOL-OTTIMA ($i-1, c$)

ELSE

CASO ① (inseriamo i nella soluzione)

SOL-OTTIMA ($\ell[i], c$)

PRINT (i)

Il costo di COSTO OTTIMO e' $\Theta(n)$.

Il costo di SOL OTTIMA sembrerebbe essere $O(n)$ (caso in cui sceglieremo sempre il caso ②).

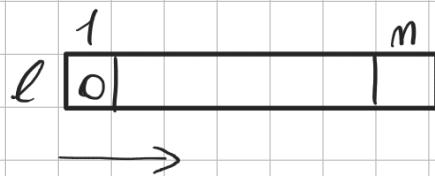
Abbiamo tuttavia considerato che le task siano ordinate e

che $l(i)$ sia calcolata a tempo costante (non e' cosi')

L'ordinamento costerà $O(n \lg n)$, la funzione $l(i)$ può essere salvata in un array con accesso costante, vediamo l'algoritmo che lo calcola e il costo di questo

CALCOLO DELLA FUNZIONE $l(i)$

L'array ha lunghezza n e $l[1] = 0$.



$l(i)$ considera gli elementi da 1 a $i-1$.

Si supponga T sequenza ordinata delle task, $l(i)$ può essere calcolato con una semplice ricerca binaria

$$q = \frac{1+(i-1)}{2}$$

A diagram illustrating a binary search process on a sequence T . The sequence is represented by a horizontal line with indices 1 , q , and $i-1$ marked. Below the line, NO is under index 1 and SI is under index $i-1$. A bracket above the line spans from index 1 to index q , and another bracket spans from index q to index $i-1$.

Se t_q e' compatibile con t_i , significa che da $i+1$ a $q-1$ sono tutti compatibili e non vogliamo il max, cerchiamolo a destra (estremi $q+1, i-1$) altrimenti cerchiamo analogamente a sinistra.

● CALCOLO $l(m)$

$$i = 1$$

$$j = m - 1$$

WHILE ($i \leq j$)

la ricerca non e' terminata.

$$q = \left\lfloor \frac{i+j}{2} \right\rfloor$$

IF $f(q) < S(m)$

t_q e' ammmissibile, cerco a destra

$$i = q + 1$$

ELSE

cerco a sinistra

$$j = q - 1$$

RETURN j

Adesso che abbiamo l' algoritmo che calcola $l(i)$, vediamo l' algoritmo che riempie completamente l' array

● RIEMPI-l(m)

$$l[1] = 0$$

FOR $i=2$ TO m DO

$$l[i] = \text{CALCOLO-}l(i)$$

CALCOLO-l costa $O(\lg m)$, RIEMPI-l chiama m volte CALCOLO-l, quindi il costo totale per risolvere la schedulazione di intervalli pesati è:

$$T_{\text{SIP}}(m) = \underbrace{\Theta(n)}_{\text{COSTO OTTIMO}} + \underbrace{\Theta(n)}_{\text{SOL-OTTIMA}} + \underbrace{\Theta(n \lg n)}_{\text{ORDINAMENTO}} + \underbrace{\Theta(n \lg n)}_{\text{RIEMPI-}l} = O(n \lg n)$$

DIMOSTRAZIONE DI CORRETTEZZA DI CALCOLA-e

① Dobbiamo dimostrare che al termine dell'algoritmo $j = l(i)$

la proprietà iniziale dell'algoritmo è che $i \leq q \leq j$.

Il ciclo termina quando $i > j$.

Le operazioni possibili ad ogni iterazione sono $j = q - 1$

oppure $i = q + 1$, ma j è sempre maggiore di q ,

quindi usciremo solo quando varrà $i - 1 = j$ perché:

Supponendo l'intervallo con almeno due elementi, avremo due così di terminazione

- Se $q < j$

Se viene eseguito $i = q + 1$ non terminiamo ($i = j$)

Se viene eseguito $j = q - 1$, se $q = i$ abbiamo terminato

($j = i - 1$) se $q > i$ non avremo ancora terminato ($j > q - 1$)

- Se $i = j$

$i = q = j$, all'ultima iterazione indifferentemente dalla scelta varrà $j = i - 1$

Chiamo questo punto, per dimostrare la correttezza dell'

algoritmo dimostriamo la validità della seguente proprietà
invariante del ciclo:

$$l(i) \in [i-1, j]$$

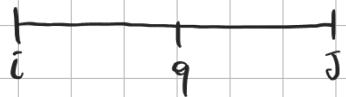
② Dimostrazione per induzione:

● Caso base (prima iterazione)

Alla prima iterazione abbiamo $i=1$ e $j=i-1$.

$l(i)$ per definizione restituisce, se esiste, una task compatibile
compresa tra 1 e $i-1$, altrimenti restituisce 0 ($i-1=0$).

● Caso induttivo (k iterazioni)



Essendo $i \leq j$ c'è almeno un elemento, q sarà il mediano

Abbiamo due casi:

$$\blacktriangleright f(q) \geq s_{(i)}$$

tutte le tasks $\geq q$ sono incompatibili e viene impostato $j=q-1$.

$l(i)$ quelli saranno per forza compresi tra 1 e $q-1 (=j)$

$$\blacktriangleright f(q) < s_{(i)}$$

q è compatibile (di conseguenza sicuramente anche tutti quelli compresi fra l e q), potrebbe esser qualche compatibile $> q$ e quindi $l(i) > q$. Vediamo entrambi i casi.

1) $l(i) > q$

L'algoritmo associa $i = q + 1$ e cerca $l(i)$ tra $q + 1$ e j , quindi la proprietà è verificata.

2) $l(i) = q$

L'algoritmo associa $i = q + 1$ e cerca $l(i)$ tra $q + 1$ e j , che però non appartiene a quest'intervallo.

Tuttavia $l(i) = q = i - 1$.

Dopo questo caso l'algoritmo sceglierà sempre $f(q) \geq S(i)$ (le task sono tutte incompatibili) eseguendo sempre $j = q - 1$ fino ad uscire.

Abbiamo dimostrato al punto ① che quando esce abbiamo sempre $j = i - 1$ ma $i = q + 1 = l(i) + 1 \Rightarrow j = l(i)$ e viene restituito.

Si osservi che vale analogamente per $l(i) = 0$.

HASSINA SOTTOSEQUENZA COMUNE

Siamo date due sequenze di caratteri "x" e "y" di lunghezze potenzialmente diverse.

$$X = x_1 x_2 \dots x_m$$

$$Y = y_1 y_2 \dots y_n$$

La massimizzazione è definita sulla lunghezza della sequenza.

La sottosequenza può non essere contigua dove però mantenere l'ordine.

E.s.

$$X = x_1 x_2 x_3 x_4 x_5 x_6$$

$x' = x_1 x_3 x_5 x_6$ è sottosequenza di X

$$X = \underline{AB} \underline{AC} \underline{DE} \underline{AA} \underline{B}$$

$$Y = \underline{CAD} \underline{BDA} \underline{ECAB}$$

$$\underline{Z}_1 = \underline{AB} \underline{D} \underline{A} \underline{A} \underline{B} \quad \text{lunghezza: 6}$$

$$\underline{Z}_2 = \underline{AB} \underline{D} \underline{E} \underline{A} \underline{B} \quad \text{lunghezza: 6}$$

le massime sottosequenze comuni possono essere multiple

MATCHING PARZIALE

Il **MATCHING PARZIALE** è una funzione parziale $X \rightarrow Y$

(potrebbe non essere definita per alcuni elementi di X) tale che

$$(x_i, y_j) \in H \Rightarrow x_i = y_j$$

CONDIZIONI DI AMMISSIBILITÀ

date due sequenze X di lunghezza n e Y di lunghezza m ,
la sequenza Z di lunghezza $k \leq \min\{n, m\}$ è ammessa per $P(X, Y)$ se

$$\exists H: X \rightarrow Y = (i_1, j_1) \dots (i_k, j_k) \mid$$

a) $1 \leq i_1 \leq i_2 \leq \dots \leq i_k \leq n$ b) $1 \leq j_1 \leq j_2 \leq \dots \leq j_k \leq m$ c) $z_e = x_{i_e} = y_{j_e} \quad \forall 1 \leq e \leq k$

$\left. \begin{array}{l} \\ \\ \end{array} \right\}$ VINCOLO DI MONOTONICITÀ

E.

$$X = \underline{AB} \underline{AC} \underline{DE} \underline{AA} \underline{B}$$

$$Y = \underline{CA} \underline{D} \underline{BD} \underline{A} \underline{EC} \underline{AB}$$

$$H = \{(x_1, j_1), (x_2, j_2), (x_3, j_3), (x_4, j_4), (x_5, j_5), (x_6, j_6)\}$$

$$SOL(X, Y) = \{(1, 2), (2, 4), (5, 5), (7, 6), (8, 9), (9, 10)\}$$

A B D A A B

OSS

- Immaginiamo che X e Y sono variabili globali, il problema $P(X, Y)$ puo' essere visto come $P(m, m)$
- La funzione di costo sara' definita basolmente sulla lunghezza della sottosequenza.

DECOMPOSIZIONE

Dato $P(m, m)$, sia $S(n, m) = z_1 \dots z_{n-1} z_n$ ammisible

Vogliamo decomporre scegliendo localmente l'ultimo elemento e delegando ai sottoproblemi il resto della sottosequenza, avremo tre casi per z_n scelta locale:

① $z_n = X_m = Y_m$

Gli ultimi due caratteri sono uguali

la sottosequenza $z_1 \dots z_{n-1}$ e' ammisible per $S(n-1, m-1)$, quindi

$$S(n, m) = S(n-1, m-1) \cdot (X_m, Y_m)$$

② $X_m \neq Y_m$

Gli ultimi 2 caratteri sono diversi.

Potremmo tuttavia avere un matching tra un carattere interno

ad una stringa che matcha l'ultimo carattere dell'altra stringa

Svilupperemo 2 casi:

a) $z_k \neq x_m \Rightarrow S(n, m) = S(n-1, m)$

Se l'ultimo elemento della soluzione non è l'ultimo carattere di x , possiamo escluderlo dal sottoproblema e delegare la soluzione a quest'ultimo

b) $z_k \neq y_m \Rightarrow S(n, m) = S(n, m-1)$

In ognuno di questi casi, essendo diversi x_m e y_m , dovranno di questi due **NON** sarà l'ultimo elemento della soluzione

Dimostriamo la validità della decomposizione (3 teoremi)

DIMOSTRAZIONE ①

Assumendo che $z = z_1 \dots z_{k-1}, z_k$ sia ammmissibile per $P(n, m)$, vogliamo dimostrare che:

① Se $z_k = x_m = y_m \Rightarrow z_1 \dots z_{k-1} = S(n-1, m-1)$ ammmissibile per $P(n-1, m-1)$

② Se $y_m \neq x_m$ a) $z_k \neq x_m \Rightarrow S(n, m)$ è ammmissibile per $P(n-1, m)$

b) $z_k \neq y_m \Rightarrow S(n, m)$ è ammmissibile per $P(n, m-1)$

① Per ipotesi $\exists H = (i_1, j_1) \dots (i_{k-1}, j_{k-1}) (i_k, j_k)$

• $i_k \leq n$ $j_k \leq m$

• Gli indici sono ordinati

• $z_e = x_{i_e} = y_{j_e}$

$i_{k-1} < i_k$ per la monotonicità.

$i_k \leq m \Rightarrow i_{k-1} < m \Rightarrow i_{k-1} \leq m-1$

$j_k \leq m \Rightarrow j_{k-1} < m \Rightarrow j_{k-1} \leq m-1$

Questo dimostra esattamente la validità del vincolo di monotonicità

per $P(n-1, m-1)$ (il vincolo di matching è verificato per ipotesi).

② a) $z_k \neq x_m$

x_m non può compiere in $z_1 \dots z_{k-1}$ per la monotonicità

(dovebbero esistere altri caratteri nella soluzione uguali a quelli successivi a x_m , che però è l'ultimo carattere)

Se quindi x_m non compone nella soluzione ammessa, rinnovando dal problema la soluzione sarà ancora ammessa

Analogamente vale per 2b

DIMOSTRAZIONE ②

① Per ipotesi $z_1 \dots z_e$ ammmissibile per $P(n-1, m-1)$ e $x_m = y_m$

Vogliamo dimostrare che $z_1 \dots z_e z'$ è omn. per $P(n, m)$ con $z' = x_m = y_m$

Per ipotesi sappiamo che (vinci di ommissibilità)

Sei $M = (i_1, j_1) \dots (i_e, j_e)$

a) $1 \leq i_1 \leq i_2 \leq \dots \leq i_e \leq n-1$

b) $1 \leq j_1 \leq j_2 \leq \dots \leq j_e \leq m-1$

c) $z_s = x_{i_s} = y_{i_s} \quad \forall 1 \leq s \leq e$

Aggiungendo ad $M(n, m) = z'$ continuano ad avere un matching parziale.

Il vincolo di monotonicità è garantito perché

$$\underline{i_e \leq n-1 < n} \quad \underline{j_e \leq m-1 < m}$$

② Intuitivamente, se la soluzione è ammmissibile per $P(n-1, m)$, sarà per forza ammmissibile per un problema che aggiunge qualcosa alla struttura. (la dimostrazione formale non l'ha fatta)

DIMOSTRAZIONE ③

Dimostriamo che se $z_1 \dots z_k$ è ottima per $P(n,m) \Rightarrow$

\Rightarrow ① $x_n = y_m \Rightarrow z_k = x_n = y_m \wedge z_1 \dots z_{k-1}$ ottima per $P(n-1,m-1)$

② $x_n \neq y_m \Rightarrow$ a) $z_k \neq x_n \quad z_1 \dots z_k$ ottima per $P(n-1,m)$

b) $z_k \neq y_m \quad z_1 \dots z_k$ ottima per $P(n,m-1)$

① P.a. $\underline{z_k \neq x_n} \wedge \underline{z_k \neq y_m}$

Per le proprietà di ommissibilità $\exists H = (i_1, j_1) \dots (i_k, j_k) \mid$

a) $i_1 < \dots < i_k \leq m$

b) $j_1 < \dots < j_k \leq n$

Per l'ipotesi d'assurdo, $i_k \neq n \wedge j_k \neq m$ quindi abbiamo

$i_k < n \wedge j_k < m$.

Se concateniamo $(n,m)=z$ a H abbiamo una soluzione

ammisibile per $P(n,m)$ perché $n \leq n$ e $m \leq m$.

Questa nuova sottosequenza è ottimale, **ASSURDO!**

Questo significa che la nostra soluzione ottima, se $x_n = y_m$

dove essere scelto **PER FORZA**

①.2 P.a. z_1, \dots, z_k ottima per $P(n, m)$ ma $\underline{z_1 \dots z_{k-1}}$ NON ottima per $P(n-1, m-1)$.

Per ipotesi $\exists z' = z_1 \dots z_e \mid e > k-1$.

Per ipotesi di ammissibilità $\exists H = (i'_1, j'_1) \dots (i'_e, j'_e)$

$$a) 1 \leq i'_1 \leq i'_2 \leq \dots \leq i'_e \leq n-1$$

$$b) 1 \leq j'_1 \leq j'_2 \leq \dots \leq j'_e \leq m-1$$

$$c) z'_j = x_{i'_j} = y_{i'_j} \quad \forall 1 \leq j \leq e$$

Per il teorema ②, possiamo concatenare a z' $z_{e+1} = x_m = y_m$.

La nuova soluzione di $P(n, m)$ $z'' = z'_1 \dots z'_e z_{e+1}$ è ammissibile perché rispetta tutti e 3 i vincoli riadattati:

$$a) 1 \leq i'_1 \leq i'_2 \leq \dots \leq i'_e \leq i_m \leq m$$

$$b) 1 \leq j'_1 \leq j'_2 \leq \dots \leq j'_e \leq j_m \leq m$$

$$c) z'_j = x_{i'_j} = y_{i'_j} \quad \forall 1 \leq j \leq e$$

z'' è ammissibile e ottima per $P(n, m)$ perché

$$C(z'_1 \dots z'_e) > C(z_1 \dots z_{k-1}), \text{ ASSURDO!}$$

② Se $x_m = y_m$ uno tra i due non e' nella soluzione.

I potizziamo $z_k \neq x_m$.

P. a. sia $z = z_1, \dots, z_K$ ottima per $P(m, m)$ ma non ottima per $P(m-1, m)$

$\exists z' = z'_1, \dots, z'_e$ ottima per $P(m-1, m)$

Ha se $z_K \neq x_m$ allora z' e' ottima anche per $P(m, m)$, ASSURDO!

FUNZIONE DI COSTO OTTIMO

Definiamo la funzione di costo ottimo per $P(i,j)$

$$C(i,j) = \begin{cases} 0 & i=0 \vee j=0 \\ 1 + C(i-1, j-1) & x_i = y_j \\ \max \left\{ C(i-1, j), C(i, j-1) \right\} & x_i \neq y_j \end{cases}$$

2a 2b

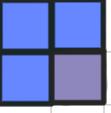
ALGORITMO DI CALCOLO DELLA FUNZIONE DI COSTO

L'implementazione naïf è come segue sempre esponenziale per la sovrapposizione.

Essendo $0 \leq i \leq m$ e $0 \leq j \leq m$, i problemi diversi sono $(m+1)(m+1) = \Theta(m \cdot m)$ problemi distinti (polinomiale).

Avevamo 2 parametri, aiutiamoci con una matrice

		0	1		J-1	J		m	
		0	0	0	-	-	-	-	0
		1	0						
		/							
		i-1	/						
		i	/						
		/							
		m	0						



Possiamo calcolare $C[i, j]$ (con $i \geq 0, j \geq 0$) potremo essere nel caso ① o ②. Per tanto dipende da $C[i-1, j]$, $C[i, j-1]$, $C[i-1, j-1]$.

Dobbiamo quindi riempire prima queste tre che significa riempire le celle in ordine crescente di riga e di colonna.

COSTOOTTIMO(m, m, X, Y)

FOR $J = 0$ TO M DO ④

$$C[0, j] = 0$$

FOR $i = 0$ TO N DO ⑤

$$C[i, 0] = 0$$

FOR $i = 1$ TO m DO

FOR $j = 1$ TO m DO

IF $X[i] = Y[j]$ ⑥

$$C[i, j] = 1 + C[i-1, j-1]$$

ELSE ⑦

IF $C[i-1, j] > C[i, j-1]$

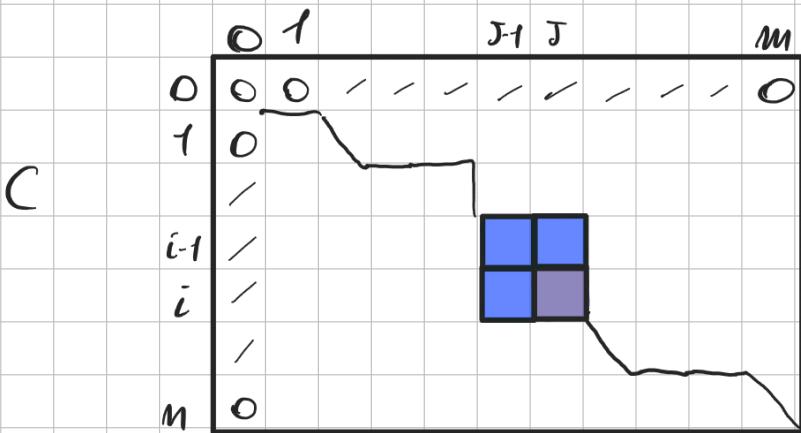
$$C[i, j] = C[i-1, j]$$

ELSE

$$C[i, j] = C[i, j-1]$$

Le operazioni sono a tempo costante, per i due passi l'esecuzione richiede tempo quadratico, che in questo caso è una proprietà intrinseca del problema.

CALCOLO DELLA SOLUZIONE OTTIMA



Nella posizione (i,j) scegliamo il carattere solo se possiamo
alla cella $(i-1, j-1)$ quindi navigando in diagonale la matrice,
altrimenti cerchiamo nelle celle $(i-1, j)$ e $(i, j-1)$ (spostamento di
1 in verticale o orizzontale)

SOLUZIONE OTTIMA (n, m, x, y)

$$\bar{C} = M$$

$$J = M$$

$$SOL = \emptyset$$

SOL e' uno stack

WHILE ($i > 0$ OR $j > 0$) DO caso di termine

IF $X[i] = Y[j]$ THEN

PUSH(SOL, $X[i]$)

$i--$

$j--$

ELSE

IF $C[i,j] = C[i-1,j]$

2a

$i--$

ELSE

2b

$j--$

RETURN SOL

Il costo di quest algoritmo e' $m+n+2$ perche'

il percorso da (n,m) a $(0,0)$ e' pari a $m+n+2$

(proprietà geometrica).

Il problema e' risolvibile in $\Theta(n \cdot m)$

OTTIMIZZAZIONE DI RICERCA IN ABR

Sono date n chiavi ordinate e distinte K_1, \dots, K_n e una sequenza P_1, \dots, P_n dove P_i è la probabilità che venga cercata la chiave K_i ($\sum_{i=1}^n P_i = 1$).

Vogliamo costruire l'ABR che minimizza il tempo medio di ricerca di una chiave.

SOLUZIONE AMMISSIBILE

Una soluzione S è ammessa per $P(K_1, \dots, K_n) \Leftrightarrow$

S è un ABR contenente tutte e sole le chiavi K_1, \dots, K_n

FUNZIONE DI COSTO

Dato un qualsiasi ABR, il tempo medio di accesso alle chiavi è dato dalla media dei tempi di tutti gli accessi, che dipende dalla profondità del nodo + 1 (consideriamo anche la radice).

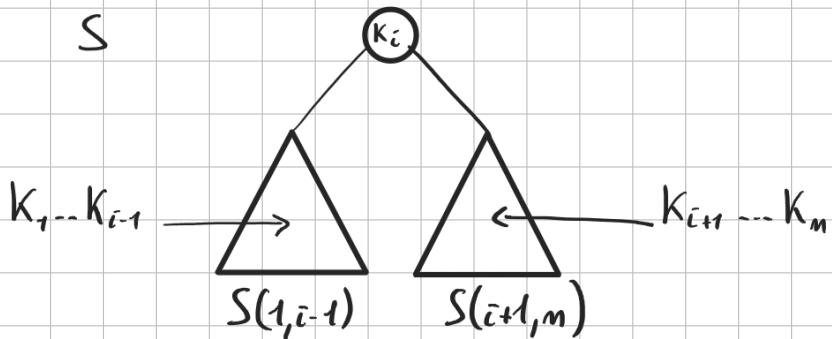
Possiamo formulare dato T un ABR, esprimendo il tempo di ricerca di K_j come $d_T(K_j) + 1$.

Il Tempo medio in T sarà $(T) = \sum_{i=1}^m (d_T(k_i) + 1) \cdot p_i$

DEFINIZIONE DI SOLUZIONE AMMISSIBILE

DATA IN INPUT $k_1 \leq \dots \leq k_m$, UN ABR S E' SOL. AMMISSIBILE

SE $S(1, m) = S(1, i-1) \cdot k_i \cdot S(i+1, m)$



Ogni problema, se sono gli elementi contigui, puo' essere espresso tramite gli estremi della sequenza.

Dimostriamo la validita' della scomposizione

DIMOSTRAZIONE ①

Il Teorema ① e' dimostrato perché ogni ABR ha come sottoalbero un ABR, quindi ogni sottoproblema di $S(1, m)$ sarà un ABR e conterrà tutti i nodi.

DIMOSTRAZIONE ②

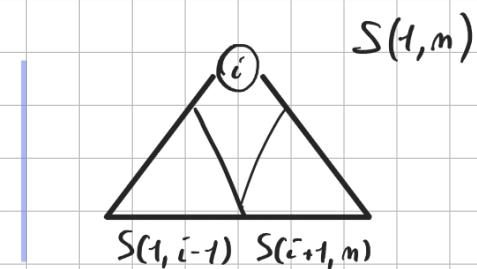
Sia $S(1, i-1)$ sol. ottim. di $P(1, i-1)$, sostituendo questo sottobelbo con un altro ABR $S'(1, i-1)$ e' ammmissibile poche' e' un ABR e contiene ogni elemento da 1 a $i-1$

Quindi il teorema ② e' verificato

DIMOSTRAZIONE ③

Sia $S(1, n)$ ottima per $P(1, n)$ allora $S(1, i-1)$ e $S(i+1, n)$ sono ottime per $P(1, i-1)$ e $P(i+1, n)$.

Ricordiamo la funzione di costo $C(S(1, n)) = \sum_{i=1}^n p_i (d_s(i) + 1)$



Non possiamo bandire la funzione di costo come
chiomata ricorsiva a $Sx + \text{costo radice} + \text{chiomata a } d_x$
perche' la profondita' di un nodo $\in Sdx$ e' diversa nell'
albero S o nel sottobelbo Sdx quindi il suo costo cambierà.

Definiamo $C(S(1, n))$ in funzione di $C(S(1, i-1))$ e $C(S(i+1, n))$.

$$C(S(1, n)) = \sum_{j=1}^n p_j \left(d_{S(1, n)}(j) + 1 \right) = \sum_{j=1}^n \left(p_j \cdot d_{S(1, n)}(j) + p_j \right) = \sum_{j=1}^n \left(p_j \cdot d_{S(1, n)}(j) \right) + \sum_{j=1}^n p_j$$

Scappiamo la prima sommatoria nei 3 contributi dell'albero
(sottoalbero sinistro, sottoalbero destro, radice).

$$= \sum_{j=1}^n p_j + \left[\sum_{j=1}^{i-1} \left(p_j \cdot d_{S(1, i-1)}(j) \right) + p_i \cdot d_{S(1, n)}(i) + \sum_{j=i+1}^n \left(p_j \cdot d_{S(i+1, n)}(j) \right) \right] =$$

$p_i \cdot d_{S(1, n)}(i) = 0$ perché i è la radice, quindi $d_{S(1, n)}(i) = 0$

(se osservi che il contributo della radice era dato del +1
della prima espressione, che abbiamo "fatto diventare" $\sum_{j=1}^n p_j$).

$$= \sum_{j=1}^n p_j + \left[\sum_{j=1}^{i-1} \left(p_j \cdot d_{S(1, n)}(j) \right) + \sum_{j=i+1}^n \left(p_j \cdot d_{S(1, n)}(j) \right) \right]$$

Per definizione le sommatorie in parentesi non rappresentano
le somme dei costi dei due sottoalberi perché anzitutto manca
il +1 e la profondità è considerata sull'albero $T(1, n)$:

$$C(S(1, i-1)) = \sum_{j=1}^{i-1} p_j (d_{S(1, i-1)}(j) + 1) \quad (\text{analogamente per } S(i+1, n))$$

Basta però osservare che $\forall j \quad d_{S(1,m)}(j) = d_{S(1,i-1)}(j) + 1$ radice
 (analogamente vale con $S(i+1,m)$).

ma quindi sostituendo avremo proprio l'espressione desiderata:

$$= \sum_{j=1}^m p_j + \left[\sum_{j=1}^{i-1} p_j (d_{S(i-1)}(j) + 1) + \sum_{j=i+1}^m p_j (d_{S(i+1,m)}(j) + 1) \right]$$

OSS

$\sum_{j=1}^m p_j = 1$ per definizione, ma potremmo anche avere altri $T(r,q)$

Definita la scomposizione correttamente, possiamo dimostrare
 il Teorema ③ come sempre:

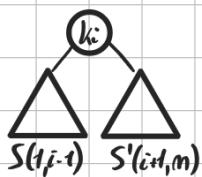
P. a.

$S(1,m)$ con radice k_i ottima per $P(1,m)$ e $S(i+1,m)$

ottima per $P(i+1,m)$ ma $S(1,i-1)$ non ottima per $P(1,i-1)$

Allora $\exists S'(1,i-1)$ ammissibile per $P(1,i-1)$ | C(S'(1,i-1)) < C(S(1,i-1)).

Costruiamo allora l'albero $S'(1,m)$



Per il teorema ② $S'(1, n)$ è ammesso per $P(1, n)$.

Il costo di $S'(1, n)$ è:

$$C(S'(1, n)) = \sum_{j=1}^n P_j + C(S(1, i-1)) + C(S'(i+1, n))$$

Il costo di $S(1, n)$ ricordiamo essere

$$C(S(1, n)) = \sum_{j=1}^n P_j + C(S(1, i-1)) + C(S(i+1, n))$$

Ha mai sappiamo volere $C(S'(1, i-1)) < C(S(1, i-1))$ quindi
 $S(1, n)$ NON è ottimo. ASSURDO!

Analogamente vale supponendo l'assurdo per $S(1, i-1)$.

FUNZIONE DI COSTO OTTIMO

Ogni problema sceglie una radice che induce 2 sottoproblemi (2 sottodberi).

$$C(p, r) = \begin{cases} \emptyset & p > r \\ \min_{p \leq i \leq r} \left\{ \sum_{j=i}^r p_j + C(p, i-1) + C(i+1, r) \right\} & \text{otherwise} \end{cases}$$

Non sappiamo localmente quale scelta induce i sottoproblemi migliori, vorremo provare tutte e scelta quella che minimizza il costo

ALGORITMO DI CALCOLO DEL COSTO OTTIMO

la sovrapposizione dei problemi avviene interamente al secondo caso della funzione, al variare di i .

Costruiamo una matrice perché il problema ha 2 variabili.

Si osservi che, sia $(1, n)$ l'input iniziale, allora

$$1 \leq p \leq n+1 \quad p-1 \leq r \leq n.$$

Abbriamo una matrice quadrata $(n+1) \times (n+1)$ dove le righe rappresentano il primo parametro e le colonne il secondo

	0	r	m
1	0		
p	0 0		
	0 0 0		
	0 0 0 0		
	0 0 0 0 0		
	0 0 0 0 0 0		
	0 0 0 0 0 0 0		

$\uparrow \rightarrow n+1$

Per calcolare le celle $[p, r]$, guardando la funzione di costo, dobbiamo conoscere gli elementi precedenti in riga per calcolare $C[p, r-1]$ e gli elementi successivi in colonna per calcolare $C[i+1, r]$.

Riempiamo la matrice partendo dall'ultima riga calcolando tutta la riga e così via.

Si noti che la diagonale inferiore ($p > r$) rappresenta solo così base.

Segue l'algoritmo.

● COSTO_OTTIMO(1, m, K, P)

```

FOR p=1 TO m+1 DO
  FOR r=0 TO p-1 DO
    C[p,r] = 0
  } INIT(c)
}
```

FOR $p=m$ DOWN TO 1

FOR $r=p$ TO m DO

SUM=0

FOR $k=p$ TO r DO

SUM=SUM + $P_{[k]}$

$$\left\{ \sum \sum_{j=i}^r P_j \right\}$$

HIN = max_rappresentabile

FOR $i=p$ TO r

calcolo del min

$$x = SUM + C[p, i-1] + C[i+1, r]$$

IF $x < HIN$

INDEX = i

chiave da mettere in radice

HIN = x

$$C[p, r] = HIN$$

matrice dei costi

$$S[p, r] = INDEX$$

matrice delle scelte locali
identica a C .

Al termine di quest' algoritmo ci interessa solo la matrice S contenente gli indici delle radici dei sottobeni.

ALGORITMO DI SOLUZIONE OTTIMA

Data S , poniamo come radice dell'albero soluzione, la chiave contenuta in $S[i, m]$ (sia questa i).

La radice avrà come figlio sinistro la chiave in $S[1, i-1]$ e come figlio destro la chiave in $S[i+1, m]$ e così via.

● SOLUZIONE-OTTIMA(p, r, S, K)

$x = \text{NIL}$

IF $p \leq r$

$x = \text{alloc-node}()$

$x \rightarrow \text{key} = K[S(p, r)]$

$x \rightarrow s_x = \text{SOLUZIONE-OTTIMA}(p, S(p, r)-1)$

$x \rightarrow d_x = \text{SOLUZIONE-OTTIMA}(S(p, r)+1, r)$

RETURN x

ANALISI DEI COSTI

SOLUZIONE-OTTIMA effettua $\Theta(n)$ chiamate ricorsive che eseguono operazioni a tempo costante.

COSTO OTTIMO riempie una matrice di grandezza circa n^2
ma il riempimento non è costante, bensì lineare su n
(calcolo del minimo).

Il costo dell'algoritmo è quindi $\Theta(n^3)$.

Tuttavia COSTO OTTIMO può essere ottimizzato applicando al calcolo
del minimo l'ottimizzazione applicata al calcolo della
sottosequenza massima visto ad ASD, quindi senza ricalcolo
ogni volta "la sequenza" (nel nostro caso SUM) bensì tenendone
traccia via via che si scorre la "sequenza"
(nel nostro caso FOR $i=p$ TO m DO)

COSTO OTTIMO ($1, m, K, P$)

INIT (c)

SUM = 0

FOR $p=m$ DOWN TO 1

FOR $i=p$ TO m DO

SUM = SUM + $P_{[i]}$

HIN = max rappresentabile

Si noti che la complessità resta cubica (per il calcolo del min) ma abbiamo ridotto di un fattore il calcolo

ZAINO Θ-1

L'input del problema è:

- Uno zaino di capacità K .
- O_1, O_2, \dots, O_m insieme di oggetti
- C_1, C_2, \dots, C_n costi degli oggetti
- P_1, P_2, \dots, P_m peso degli oggetti

Costo e peso dello zaino sono boudamente definiti per somma.

Il peso dello zaino dev'essere **INFERIORE** o uguale a K .

Una soluzione del problema $P(O)$ è un sottoinsieme di oggetti la cui somma dei pesi è inferiore o uguale a O .

Come già visto, l'insieme dei sottoproblemi di un problema che ha un input un insieme è esponenziale ($2^{|O|}$).

Usiamo O come sequenza ordinata secondo qualche criterio per risolvere il problema come nella schedulazione delle tasks individuando l'ultimo elemento.

Il problema $P(m, k)$ riduce (se sceglio come ultimo elemento l' n -esimo della sequenza O) $P(m-1, k - p_n)$ più di O_n scelto.

Se $P(m, k)$ non sceglie l' n -esimo induno $P(m-1, k)$.

SOLUZIONE AMMISSIBILE

Dato un problema $P(n, w)$ con input

- $O = O_1, \dots, O_n$ oggetti
- $P = p_1, \dots, p_n$ pesi
- $C = c_1, \dots, c_n$ costi degli oggetti
- w capacità dello zaino

Una sequenza $S(J, w) = i_1, \dots, i_k \subseteq O$ è soluzione ammissibile se:

- ① $1 \leq i_1 \leq i_2 \leq \dots \leq i_k \leq J$ (J è l'ultimo indice disponibile)
- ② $\sum_{\ell=1}^k p_{i_\ell} \leq w$ (w è il peso disponibile da occupare)

Possiamo vedere $S(J, w) = i_1, \dots, i_k$ come (decomposizioni):

$$③ S(k-1, w - p_k) \circ i_k$$

$$④ \begin{cases} S(J-1, w) & i_k < J \\ S(J-1, w - p_J) \circ J & i_k = J \end{cases} \quad \begin{array}{l} J \text{ non viene inserito} \\ J \text{ viene inserito} \end{array}$$

Entrambe rispettano tutti i vincoli, noi useremo la decomposizione ④.

La dimostrazione dei due teoremi è molto simile a quella del problema delle task (dimostriamo la seconda decomposizione).

DIMOSTRAZIONE ①

Sia $S(j, w) = i_1, \dots, i_k$ soluzione per $P(j, w)$.

- $i_k < j$

i_1, \dots, i_k sarà soluzione di $P(j-1, w)$ perché:

① ~~meno~~ $i_k < j \Rightarrow i_k \leq j-1$

② w non eccede w (avviamente)

- $i_k = j$

i_1, \dots, i_{k-1} è soluzione di $(j-1, w - p_j)$ perché

① $i_{k-1} < i_k$ ma $i_k = j$ quindi $i_{k-1} \leq j-1$

② $\sum_{\ell=1}^k p_{i_\ell} = w$, togliendo i_k avremo $w - p_k$

DIMOSTRAZIONE ②

- $i_k < j$

$S(j-1, w)$ è ammmissibile per $P(j, w)$ perché questo richiede
vincoli più leggeri ($j > j-1$ e $w=w$)

- $i_k = j$

$S(j-1, w - p_j) + j$ è ammmissibile per $P(j, w)$:

① sia $S(j-1, w - p_j) = i_1, \dots, i_{j-1}$

concatenando j avremo i_1, \dots, i_{j-1}, j che rispetta il primo
vincolo perché $j > j-1$, quindi è ancora crescente

② $\sum_{e=1}^j p_{i_e} \leq K - p_j$

Aggiungendo p_j (e quindi j) abbiamo proprio K

OSS

Dimostrare la validità della decomposizione ① è quasi
del tutto analogo alla dimostrazione del caso $i_k=j$.

Dimostrazione ③

Ricordiamo che $C(S(j, w)) = \sum_{e=1}^k c(i_e)$

• $i_k < j$

banche in quanto $S(j, w) = S(j-1, w)$.

Se p.a. non fosse ottima per $P(j-1, w)$, avremo la stessa
soluzione con lo stesso peso, non sarebbe ottima per $P(j, w)$ ASSURDO!

• $\bar{i}_k = j$

$i_1 \dots i_{j-1}$ dev'essere ottima per $P(j-1, w - p_j)$

P.a. $i_1 \dots i_{j-1}$ NON ottima per $P(j-1, w - p_j)$.

$$\exists S'(j-1, w - p_j) = \bar{i}'_1 \dots \bar{i}'_k \text{ amm. per } P(j-1, w - p_j) \quad \left| \sum_{e=1}^j c(\bar{i}'_e) > \sum_{e=1}^{k-1} c(\bar{i}_e) \right.$$

Per il teorema ② $S'(p, w) = \bar{i}'_1 \dots \bar{i}'_j$ e' amm. per $P(j, w)$.

$$C(S'(j, w)) = C(j) + \sum_{e=1}^j c(\bar{i}'_e)$$

$$C(S(j, w)) = C(j) + \sum_{e=1}^{k-1} c(\bar{i}_e)$$

Ma $\sum_{e=1}^j c(\bar{i}'_e) > \sum_{e=1}^{k-1} c(\bar{i}_e)$, quindi $S'(j, w)$ e' ottima per $P(j, w)$

ASSURDO!

FUNZIONE DI COSTO OTTIMA

Vediamo la funzione per entrambe le decomposizioni.

a)

$$C(j, w) = \begin{cases} 0 \\ \max_{\substack{i \leq k \leq j \\ p_k \leq w}} \left\{ C(k-1, w - p_k) + C(k) \right\} \end{cases} \quad \forall i \leq k \leq j \quad p_k > w$$

ci sono due problemi principali in questa decomposizione:
 non conoscendo l' i_K che minimizza il sottoproblema, li proviamo tutti e sceglieremo di conseguenza. Sorge però un problema, noi non abbiamo vincoli sul peso di O_{i_K} , che potenzialmente potrebbe essere maggiore di K , quindi dobbiamo porsi un vincolo in più sul max. Un secondo problema sorge nel momento in cui non ci sono oggetti che rispettano questo nuovo vincolo, perché $\max\{\emptyset\}$ non esiste, quindi uniamo questo caso a quello in cui $J=0$ e ottieniamo il caso base.

$$(b) \quad C(J, w) = \begin{cases} 0 & J=0 \\ C(J-1, w) & P_J > w \wedge J > 0 \\ \max \left\{ C(J-1, w), C(J-1, w-P_J) + C(J) \right\} & P_J \leq w \wedge J > 0 \\ & i_K < J \\ & i_K = J \end{cases}$$

Anche in questo caso sorge il problema di P_J , ci basta dire che se $P_J \leq w \wedge J > 0$ allora prendiamo il max tra la soluzione con J e quella senza J , ottenimenti se

$P_J > w \wedge J > 0$ scarto l'ipotesi $C(J-1, w-P_J) + C(J)$ (caso $i_K = J$),

quindi delego la scelta al sottoproblema. Altrimenti se $j=0$ abbiamo il nostro caso base.

OSS

Entrambe le funzioni generano $n \cdot w$ sottoproblemi, ma nella decomposizione (a), nel secondo caso per riempire una cella non banale, dobbiamo conoscere tutti i valori della chiamata $C(K-1, w - p_k)$ e poi confrontarla con $C(K)$. K varia tra 1 ed J , J varia tra 1 ed n quindi ogni problema (cella) viene riempita in tempo lineare su n e quindi la matrice in tempo $\Theta(n^2 \cdot w)$ (per colonna dell'indice del max).

La soluzione (b) nel suo terzo caso confronta il max di due sole celle, il che implica un tempo costante e quindi un tempo di riempimento della matrice $\Theta(nw)$.

Sceglieremo perciò la soluzione (b)

ALGORITMO DI CALCOLO DELLA FUNZIONE DI COSTO OTTIMA

La prima riga è tutta 0 (poniamo j sulle righe e w sulle colonne), la chiamata $(j-1, w)$ ci obbliga a calcolare prima la riga precedente a quella corrente (quindi riempiremo le righe in ordine crescente) e siccome per $((j-1, w - p_j))$ $w - p_j$ potrebbe assumere qualsiasi valore, quindi dobbiamo ancora di conoscere tutta la colonna (l'ordine di riempimento di questa quindi non importa).

• COSTO-OTTIMO ($n, w, p, c, 0$)

FOR $w=0$ TO W DO

$C[0, w] = 0$

FOR $j=1$ TO n DO

FOR $w=0$ TO W DO

costo = $C[j-1, w]$

IF $p_j \leq w$

IF $\text{costo} < C[j-1, w - p_j] + c_{ij}$

costo = $C[j-1, w - p_j] + c_{ij}$

$C[j, w] = \text{costo}$

$((j-1, w))$ va sempre calcolato

controllo del vincolo di peso

calcolo del nuovo

$\Theta(n \cdot w)$

● SOL OTTIMA (n, o, p, c, C)

$J = n$

partiamo da $S(n, w)$

$w = W$

$SOL = \emptyset$

WHILE $J > 0$ DO

IF $p_{[j]} > w$

se la sulta $i_k=j$ non e' ammessa delega

$J--$

ELSE

dobbiamo scegliere se prendere j oppure no

IF $C_{[j-1, w]} = C_{[j, w]}$

$i_k < j$

$J--$

ELSE

$i_k = j$

$SOL = PUSH(SOL, j)$

$J--$

$w = w - p_{[j]}$

RETURN SOL

la soluzione al problema dello zaino viene calcolata in
tempo $\Theta(nw)$

RICERCA PERCORSO MIN IN GRAPPI PESATI

Dato $G = \langle V, w \rangle$ grafo pesato e due vertici $s, t \in V$, vogliamo trovare il percorso minimo tra s e t .

Ogni arco ha un peso ovvero w è una funzione $w: V \times V \rightarrow \mathbb{Z} \cup \{\infty\}$.

Una soluzione ammmissibile è un percorso tra s e t , quindi:

$$s v_1 \dots v_k t \mid \forall i \leq k \quad v_i \in V$$

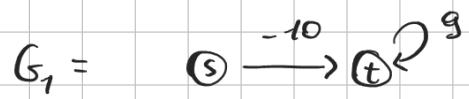
$$\forall i \leq k \quad w(v_i, v_{i+1}) < \infty$$

Il costo (peso) di un percorso π è

$$C(\pi) = \sum_{e=1}^{k-1} w(v_e, v_{e+1})$$

Per **MINIMO** intendiamo il percorso che minimizza il peso.

A differenza degli altri problemi, il percorso minimo potrebbe **NON** esistere:



In G_1 , il percorso minimo è s, t con peso $-10 + 9$ (prendendo una sola volta t). Concatenando t (s, t, t, \dots) il peso aumenta.

In G_2 poniamo avere $\pi = s, t$ con peso -10, potremo anche prendere $\pi' = s, t, s, t$ con peso -11 e così via.

In G_2 quindi **NON** esiste, dovuto alla presenza di un ciclo con un peso negativo.

OSS

Un grafo pesato privo di cicli a peso negativo ammette **SEMPRE** un percorso minimo.

Se sono presenti cicli a peso positivo potrebbe comunque esistere il percorso desiderato.

PROPRIETA' DEL PROBLEMA

E' importante notare che una soluzione e' già una sequenza ordinata (di vertici), quindi se $abcd$ e' un percorso, anche bc e' un percorso.

Questo ci permette di segmentare una soluzione ($ab\cdots cd$)

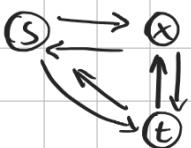
Quindi avendo $v_1, \dots, v_j, v_{j+1}, \dots, v_k$ soluzione di $P(v_1, v_k)$, abbiamo banalmente anche $S(v_1, v_j)$ e $S(v_{j+1}, v_k)$.

Potremmo adottare quindi una decomposizione del tipo

$$S(v_1, v_k) = v_1 v_2 \dots v_k = S(v_1, v_2) \cdot v_2 \dots v_k$$

Purtroppo questa decomposizione, per quanto valida, non e' efficace

E.s.



Sia il nostro problema $P(s, t)$.

Con la decomposizione ricorsiva avremo:

$$S(s, t) = (s, x) \cdot P(x, t) \quad \text{decomponiamo ora } P(x, t)$$

$$S(x, t) = (x, t)$$

Siamo tornati al punto di partenza, quindi la decomposizione non e' ben definita.

OSS

Se il grafo fosse aciclico, la decomposizione sarebbe valida.

PROPRIETA'

Se esiste un percorso minimo da s a t (vertici), allora esiste un percorso minimo semplice (in graphi pesati)

DI MOSTRAZIONE

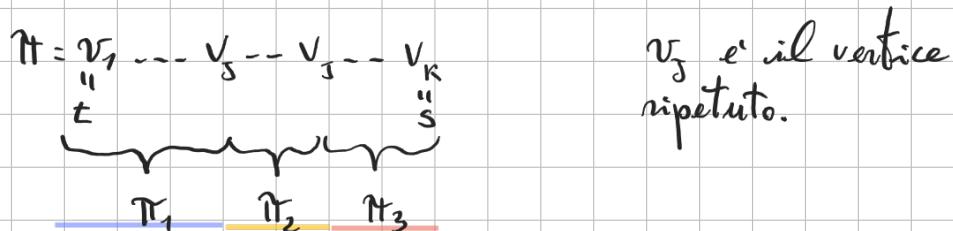
Vogliamo dimostrare che, sia π il percorso minimo da s a t , esiste π' tale che:

$$\textcircled{1} \quad C(\pi) = C(\pi')$$

$$\textcircled{2} \quad \pi' \text{ non presenta ripetizioni di vertici}$$

Se π è semplice, $\pi = \pi'$ (banale).

Sia π quindi non semplice :



$$C(\pi) = C(\pi_1) + C(\pi_2) + C(\pi_3) = \sum_{(v,v') \in \pi_1} w(v, v') + \sum_{(v,v') \in \pi_2} w(v, v') + \sum_{(v,v') \in \pi_3} w(v, v')$$

Sappiamo per ipotesi che non sono presenti cicli o peso negativo (altrimenti non avrebbe soluzione), questo implica che $C(\pi_2) \geq 0$. Quindi possiamo dire che:

$$C(\pi_1) + C(\pi_3) \leq C(\pi_1) + \underline{C(\pi_2)} + C(\pi_3)$$

E' facile notare che $\pi' = \pi_1 \cdot \pi_3$ e' un percorso da s a t
(perche' π parte e termina con lo stesso vertice che e' proprio sul
vertice v_3 che unisce π_1 e π_3) minimo.

Questo significa che basta ridursi ai percorsi semplici in quanto
i cicli sono tutti a peso ≥ 0 quindi non ha senso tenerne conto.

PROPRIETA'

Se π e' un percorso semplice allora

$$\#\text{archi}(\pi) \leq |V| - 1$$

CONSEGUENZE DELLA PROPRIETA'

Sfrutteremo il no di archi per la nostra decomposizione
sapendo che $|V|-1$ e' il limite superiore.

Sia $P(s, t, |V|-1)$ il problema iniziale:

$$P(s, t, |V|-1) = (s, x) \cdot P(x, t, |V|-2)$$

In questo modo, pur entrando in un ciclo, il numero di archi

a disposizione diminuisce fino a 0.

Possiamo rimuovere t come parametro caratterizzante in quanto sarà sempre presente.

Quindi il problema iniziale $P(s, |V|-1)$ può essere generalizzato come $P(v, j)$.

DECOMPOSIZIONE RICORSIVA

Ricordiamo che $S(v, j) = v_1 \dots v_j$ è **SOLUZIONE AMMISIBILE** per $P(s, |V|-1)$ se :

① $v_1 = s \wedge v_k = t$

② $v_1 \dots v_k$ è presente in G

③ $k-1 \leq j$ (il numero di nodi è sempre +1 numero di archi)

Decomponiamo $S(s, |V|-1) = v_1 v_2 \dots v_k$ come

① $S(s, |V|-1) = (v_1, v_2) \cdot S(v_2, |V|-2)$ se $k-1 = |V|-1$

② $S(s, |V|-1) = S(s, |V|-2)$ se $k-1 < |V|-1$

Se il numero di archi è massimo dobbiamo "rimuovere"

un arco per passare a un sottoproblema, altrimenti non ce n'è bisogno

Dimostriamo i 3 teoremi

DIMOSTRAZIONE ①

$$S(s, |V|-1) = v_1, v_2 \dots v_k \text{ amm. per } P(s, |V|-1)$$

$$\textcircled{1} \quad k-1 = |V|-1$$

Vogliamo dimostrare che $v_2 \dots v_k$ è amm. per $P(v_2, |V|-2)$.

Questo è ovvio perché sono rispettati i 3 vincoli di ommissibilità:

$$1) \quad v_2 = v_2$$

2) Rimuovendo l'arco (v_1, v_2) il percorso ottenuto sarà sicuramente presente nel grafo (sapendo che $v_1, v_2 \dots v_k$ è presente)

3) Rimuovendo v_1 la lunghezza del percorso sarà $k-1-1 = k-2$.

Se $k-1 \leq |V|-1$ per ipotesi allora $k-2 \leq |V|-2$

$$\textcircled{2} \quad k-1 < |V|-1$$

Vogliamo dimostrare che $v_2 \dots v_k$ è amm. per $P(v_2, |V|-2)$.

$$1) \quad v_1 = v_1$$

2) Vero per ipotesi

3) $k-1 < |V|-1 \Rightarrow k-1 \leq |V|-2$

DIMOSTRAZIONE ②

Sia $v_2' \dots v_n'$ ottim. per $P(v_2, |V|-2)$

① $r = |V|-2$

$s v_2' \dots v_n'$ ottim. per $P(s, |V|-1)$

$v_2' = t$ e $(s, v_2') \in E$ per ipotesi di ammissibilità,

concatenando s abbiamo un percorso da s a t di lunghezza

$r+1 = |V|-1$

② $r < |V|-2$

$r \leq |V|-1$ ed è un percorso in G da s a t per ipotesi

DIMOSTRAZIONE ③

Sia $S(s, |V|-1) = v_1, v_2 \dots v_k$ ottimo per $P(s, |V|-1)$

① $C(S(s, |V|-1)) = w(v_1, v_2) + C(S(v_2, |V|-2))$

P.o.a. sia $S(v_2, |V|-2)$ non ottima per $P(v_2, |V|-2) \Rightarrow$

$\Rightarrow \exists S'(v_2, |V|-2) \mid \underline{C(S'(v_2, |V|-2))} > \underline{C(S(v_2, |V|-2))}$

Esisterà allora $S'(v_1, |V|-1) = (v_1, v_2) \cdot S'(v_2, |V|-2)$ tale che

$$C(S(s, |V|-1)) = w(v_1, v_2) + C(S(v_2, |V|-2))$$

Siccome $C(S(v, |V|-2)) > C(S(v_2, |V|-2))$ allora

$$C(S(s, |V|-1)) > C(S(s, |V|-1)) . \text{ ASSURDO!}$$

② $C(S(s, |V|-1)) = C(S(s, |V|-2))$

Bonole con l'assurdo

FUNZIONE DI COSTO OTTIMA

$$C(v, j) = \begin{cases} 0 & \text{se } j=0 \wedge v=t \\ \infty & \text{se } j=0 \wedge v \neq t \\ \min \left\{ C(v, j-1), \min_{x \in A_{j-1}(v)} \left\{ w(v, x) + C(x, j-1) \right\} \right\} & \text{altrimenti} \end{cases}$$

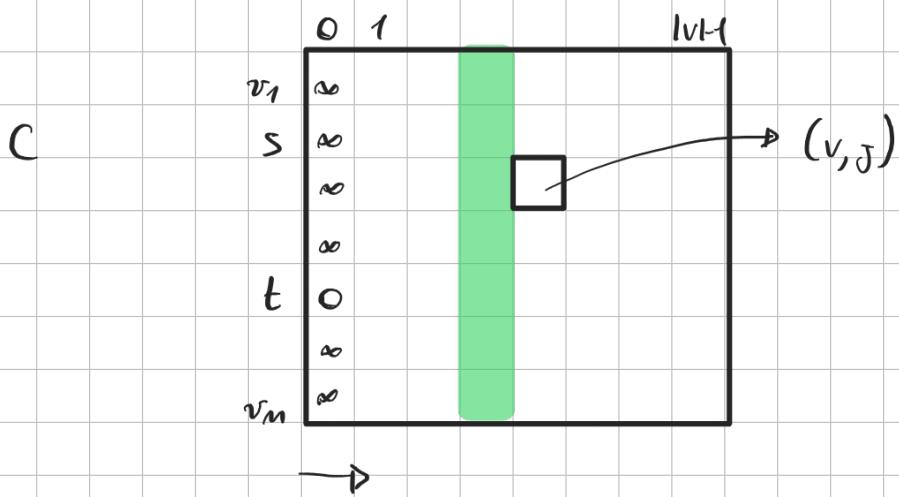
①

②

Non conosciamo il modo x che minimizza $w(v, x)$, quindi
dobbiamo provarle tutte e prendere il minimo

ALGORITMO DI COSTO OTTIMO (FLOYD-WARSHALL)

Istruziono la matrice dei costi avente sulle righe
i nodi e sulle colonne i nodi da 0 a $|V|-1$



La cella (v, j) dipende da $\underline{(v, j-1)}$ e tutti gli elementi antecedenti ad esso nella colonna precedente $\underline{(c(x, j-1))}$.

Dobbiamo quindi riempire in ordine crescente per colonna (tutta la colonna) la matrice.

Paralleamente costruiamo una matrice delle scelte sempre $|V| \times |V|-1$ dove, al riempimento di $C[v, j]$ solviamo la scelta nella cella $S[v, j]$

● COSTO-OTTIMO (G, s, t)

$$m = |V|-1$$

FOR EACH $v \in V$ DO

IF $v = t$ THEN CASI BASE

$$C[v, 0] = 0$$

ELSE

$$C[v, 0] = \infty$$

$$S[v, 0] = NIL$$

FOR $j=1$ TO m DO

FOREACH $v \in V$ DO

$$\left. \begin{array}{l} C[v, j] = C[v, j-1] \\ S[v, j] = NIL \end{array} \right\} \textcircled{1} \quad (\text{assumiamo sia il minimo inizialmente})$$

FOREACH $x \in \text{Adj}(v)$ DO

$$\left. \begin{array}{l} \text{IF } w(v, x) + C[x, j-1] < C[v, j] \\ C[v, j] = w(v, x) + C[x, j-1] \\ S[v, j] = x \end{array} \right\} \textcircled{2}$$

• SOLUZIONE OTTIMA (v, j)

IF $j \neq 0$ AND $v \neq t$

IF $S[v, j] = NIL$

SOLUZIONE OTTIMA ($v, j-2$)

ELSE

SOLUZIONE OTTIMA ($S[v, j], j-1$)

PRINT $S[v, j]$

L'algoritmo di ricostruzione della soluzione ottima ha costo $O(|V|)$ (lunghezza del percorso) mentre l'algoritmo di costo ottimo così strutturato ha costo $O(|V|^3)$ in tempo e $O(|V|^2)$ in spazio.

Si può notare che il ciclo FOR con i due FOREACH interni rappresenta ciò che farebbe una visita, quindi il costo è equivalente a $\Theta(|V| \cdot |E|)$.

OTTIMIZZAZIONE SPAZIALE

E' possibile ottimizzare in termini di spazio l'algoritmo (passando a $\Theta(|V|)$) osservando che ogni cella dipende solo dalla colonna precedente a quella corrente, quindi salvando una matrice $|V| \times 2$ e sovrascrivendo ogni volta la colonna precedente, avremo tutte le informazioni.

Si possono inoltre salvare le scelte in un array (che funziona un po' come l'array dei predecessori nell'algoritmo di BFS).

Quest'algoritmo è detto di Bellman-Ford.

PROGRAMMAZIONE GREEDY

L'approccio greedy permette spesso di ottimizzare in termini di spazio gli algoritmi proposti dalla programmazione dinamica.

Per problemi di ottimizzazione, gli algoritmi greedy si riducono ad una sottoclasse dei problemi di programmazione dinamica, quindi richiedono una decomposizione con la verifica delle 3 proprietà + una quarta fondamentale.

Vediamo l'esempio dell'algoritmo di Floyd-Warshall

$$C(v, j) = \begin{cases} 0 & \text{se } j=0 \wedge v=t \\ \infty & j=0 \wedge v \neq t \\ \min \left\{ C(v, j-1), \min_{x \in \text{Adj}_j(v)} \left\{ w(v, x) + C(x, j-1) \right\} \right\} & \text{altri casi} \end{cases}$$

Il modo in cui viene fatta la scelta e' provando le alternative e verificando quali sono le conseguenze di tale scelta.

L'approccio greedy confronta le scelte in base alle sole proprietà locali (ad esempio il peso dei singoli archi).

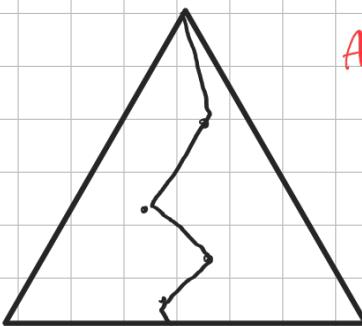
Questo e' possibile solo in alcuni casi (si veda ad esempio come questo caso non possa essere risolto con un algoritmo greedy)

la scelta greedy di una soluzione permette di eliminare le matrici dei costi e delle soluzioni sostanzialmente.

Inoltre, l'ottimalita' non sara' piu' data dal numero di sottoproblemi, bensì dalla politica di scelta locale, il che permette di semplificare alcune scelte di decomposizione.

Quindi non calcoleremo tutti i sottoproblemi, bensì solo una sequenza che ci portera' a una soluzione.

L'albero rappresenta l'approccio dinamico, il percorso quello greedy



ALBERO DELLE
SCELTE

OSS

In problemi di approssimazione o Euristicci, dove non e' richiesta l'ottimalita' e l'esattezza, gli algoritmi greedy sono largamente usati

SCHEDULAZIONE DI INTERVALLI (NON PESATI)

Le ipotesi sono quelle già viste, in questo caso però agli intervalli non c'è associata alcuna funzione di costo.

La massimizzazione della soluzione è definita sul numero di task.

La definizione di ommissibilità di una soluzione è la stessa, ovvero:

Sia $T = \{a_{i_1}, \dots, a_{i_K}\}$ insieme delle task dove a_i inizia al tempo s_i e termina al tempo f_i , $S \subseteq T$ è soluzione ommissibile se

$$\textcircled{1} \quad a_{i_j} \in T \quad \forall i \leq j \leq K$$

$$\textcircled{2} \quad a_{i_j} \cap a_{i_{j+1}} = \emptyset \quad \forall i \leq j < K$$

$$\textcircled{3} \quad f[a_{i_j}] < f[a_{i_{j+1}}] \quad \forall i \leq j < K$$

Assumendo $\textcircled{2} + \textcircled{3}$ possiamo dimostrare che $a_{i_j} \cap a_{i_2} = \emptyset \quad \forall i_2 > j$

DIMOSTRAZIONE

Essendo $\textcircled{2}$ e $\textcircled{3}$ le nostre ipotesi, sappiamo che

$$f[a_{i_j}] < s[a_{i_{j+1}}] < f[a_{i_{j+1}}]$$

$\underbrace{\hspace{10em}}$ sovrapposizione
 $\underbrace{\hspace{10em}}$ durata di una task

Ai conseguenza considerando le task $a_{i_{j+1}}$ e $a_{i_{j+2}}$ vera

$$a_{i_{j+1}} \cap a_{i_{j+2}} = \emptyset \Leftrightarrow f[a_{i_{j+1}}] \subset S[a_{i_{j+2}}]$$

Per transitività $f[a_{i_j}] \subset S[a_{i_{j+2}}] \Leftrightarrow a_{i_j} \cap a_{i_{j+2}} = \emptyset$

Avendo dimostrato un particolare passo induttivo ($j+2$) ed essendo la base inductive la nostra ipotesi, potremmo effettuare la dimostrazione formale per induzione su j .

ANALISI DEL PROBLEMA

La decomposizione già vista sarebbe del tutto valida
(immaginando tutti i pesi = 1).

Per un approccio greedy possiamo trovare una soluzione più semplice per ridurre la matrice delle soluzioni.

Il nostro problema $P(j)$ ha come soluzione la sequenza $a_{j+1} \dots a_n$ (in pratica il parametro i dice che la soluzione del problema dev'essere compatibile con a_j) quindi il problema iniziale è $P(0)$ con a_0 task fittizio

DECOMPOSIZIONE RICORSIVA

La decomposizione proposta è $S(0) = a_{i_1} a_{i_2} \dots a_{i_K} = a_{i_1} \cdot S(i_1)$ con:

$$S(j) = a_{i_1} \dots a_{i_K}$$

$$\textcircled{1} \quad a_{i_K} \in \{a_{j+1}, \dots, a_m\}$$

$$\textcircled{2} \quad a_{i_j} \cap a_{i_{j+1}} = \emptyset \quad \forall i \leq j < k$$

$$\textcircled{3} \quad f[a_{i_j}] < f[a_{i_{j+1}}] \quad \forall i \leq j < k$$

$$\textcircled{4} \quad a_j \cap a_{i_1} = \emptyset$$

Dobbiamo dimostrare i 3 teoremi ma a Bemby pesava il
culo, dovrebbero essere quasi identiche a quelle già viste.

FUNZIONE DI COSTO

$$C(j) = \begin{cases} 0 & j = m \\ \min_{\substack{j+1 \leq k \leq m \\ a_k \cap a_j = \emptyset}} \{f + C(k)\} & \text{costo di } a_k \end{cases} \quad (\text{non ci sono task compatibili})$$

Con la programmazione dinamica è facile calcolare la funzione
usando un array (un parmetro) e concludendo in tempo $\Theta(m^2)$

(questa decomposizione non è ottimale per la programmazione dinamica)

Quello che fa questa funzione è simulare una scelta e vedere che problema induce (calcolo del max).

Noi vogliamo evitare ciò' avendo una funzione ricorsiva.

APPROCCIO GREEDY

Sia $P(j)$ il problema che ha come prima scelta della soluzione $a_i, \dots, a_{j+1}, \dots, a_n$ (per ammissibilità).

Noi vogliamo sapere, confrontando solamente le task tra loro, quel è la migliore e spostarci sul suo sottoproblema (evitando così quel calcolo del max).

Cio' è possibile solo decidendo **A PRIORI** una **POLITICA DI SCELTA**, ovvero un criterio di scelta da un insieme.

Essendo la scelta (conoscendo un criterio valido) a tempo costante, il problema sarebbe risolto in tempo $\Theta(n)$.

Non è facile, e non è detto che sia possibile, individuare una politica valida.

Una politica in questo caso potrebbe essere scegliere l'attività che termina per prima (tra quelle compatibili).

Questa politica verifica il **TEOREMA DELLA SCELTA GREEDY**

TEOREMA DELLA SCELTA GREEDY

Dato un problema $P(j)$, la scelta greedy è la prima scelta di una soluzione ottima di $P(j)$.

Quindi partendo da $P(0)$ la soluzione ottima è facilmente ottenuta.

Dimostriamo la validità della nostra strategia greedy.

DI MOSTRAZIONE

Sia $S = a_{i_1}, a_{i_2}, \dots, a_{i_K}$ ottima per $P(j)$ tale che:

$$\textcircled{1} \quad a_{i_K} \in \{a_{j+1}, \dots, a_n\}$$

$$\textcircled{2} \quad a_{i_j} \cap a_{i_{j+1}} = \emptyset \quad \forall i \leq j < K$$

$$\textcircled{3} \quad a_j \cap a_{i_1} = \emptyset$$

$$\textcircled{4} \quad f[a_{i_j}] < f[a_{i_{j+1}}] \quad \forall i \leq j < K$$

Supponiamo che a_r sia la scelta greedy di $P(J)$ ($j \leq r \leq n$)

Per come è formulata verifica sicuramente ① e ③.

Essendo la scelta la task che termina per prima varrà:

$$f[a_r] \leq f[a_e] \underbrace{\forall j < l \leq n \wedge a_e \cap a_j = \emptyset}_{\text{task compatibili rimanenti}}$$

Essendo a_r compatibile con a_j (vincolo ③), per quanto appena visto sappiamo che $f[a_r] \leq f[a_{r+1}]$.

Se $S' = a_1 a_{r+1} \dots a_n$ è ommissibile sarà anche ottima perché ha la stessa lunghezza di S soluzione ottima e quindi avremo dimostrato il teorema per la nostra politica.

Dimostriamo i 6 vincoli di ommissibilità per S' :

① Banale perché $j < r \leq n$

③ È verificato per come è definita la politica

④ È verificato per come è definita la politica

② Dobbiamo dimostrare che $a_r \cap a_i$, in quanto l'ordine già dimostra il vincolo (dimostrato prima)

E' verificato perché $f[a_1] \leq f[a_i] \leq f[a_{i_2}]$ quindi per transitività
 $f[a_1] \leq f[a_{i_2}]$

DSS

Il modo in cui si dimostra la validità del teorema per la politica scelta e' sempre lo stesso:

Partendo da una soluzione ottima per un problema, prendo la scelta greedy per quel problema e sostituendo la prima scelta con la greedy. Se questa nuova scelta e' ottimale il teorema e' verificato

ALGORITMO GREEDY

SOL-GREEDY (τ)

J=0

FOR K=J+1 TO N DO

IF $f[a_j] < S[a_k]$

J=K

$S = S \cup \{a_k\}$

tiene conto dell'ultima scelta e quindi induce il sottoproblema

RETURNS

Il costo dell'algoritmo, supponendo la sequenza ordinata per tempi di terminazione, e' $\Theta(n)$

SCHEDULAZIONE DELLE TASK

Se $T = \{1, 2, \dots, n\}$ l'insieme delle attività con durata $d_i > 0$.

Vogliamo schedulare le task in modo tale da ottimizzare il tempo di completamento medio.

Una schedulazione può essere vista come una funzione

$$Sch : T \rightarrow \underline{\text{Tempo}}$$

Una soluzione ammessa è una sequenza $(i_1, t_1) \dots (i_n, t_n)$ tale che

- $i_j \in T \quad \forall i \leq j \leq n$

- $i_k \neq i_j \quad i_k \neq i_j \quad \forall i \leq k, j \leq n$

- $\underbrace{t_j + d_{i_j}}_{\text{istante di terminazione}} \leq t_{j+1} \quad \forall i \leq j \leq n$

condizione di compatibilità

istante di terminazione

Dove i_1, \dots, i_n è una permutazione di T .

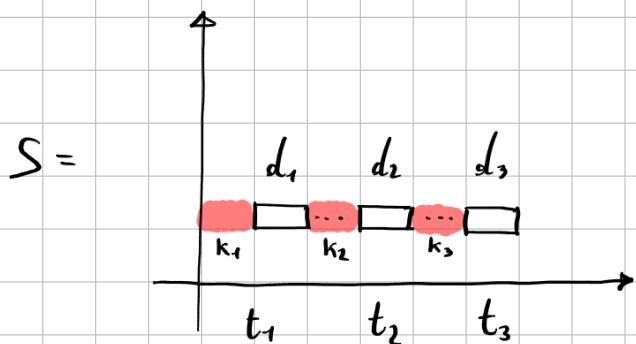
COSTO DI UNA SOLUZIONE

Il tempo medio di completamento è definito come

$$\text{Tempo Medio} = \frac{1}{n} \sum_{j=1}^n TC(j) \quad \text{dove } TC(j) = \underbrace{t_j + d_{i_j}}$$

ANALISI DEL PROBLEMA

Lo spazio di ricerca delle soluzioni è infinito perché esistono infinite funzioni di schedulazione, il che potrebbe essere un problema. Possiamo per fortuna ridurre la ricerca.



Una soluzione si farà non sarà MAI ottimale perché possiede periodi di inattività, quindi possiamo sempre cercare una soluzione con meno idle time.

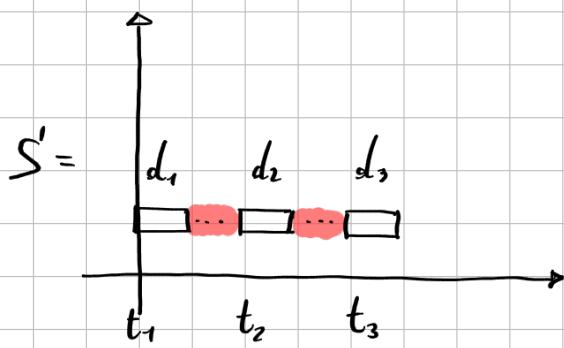
DIMOSTRAZIONE

Premettiamo in esame l'esempio precedente.

Sia per assurdo ottimo. Il costo di S sarà:

$$(S) = \frac{1}{3} \sum_{j=1}^3 t_j + d_j$$

Costruiamo S' traslando tutte le task di un' unità di tempo all' indietro (riduciamo di t_1 l' idle time)



$$S = (1, t_1) (2, t_2) (3, t_3)$$

$$S' = (1, 0) (2, d_1 + k_2) (3, d_1 + k_2 + d_2 + k_3)$$

Si puo' notare che $d_1 + k_2 = t_2 - t_1$ e $d_1 + k_2 + d_2 + k_3 = t_3 - t_1$
 Calcoliamo il costo di S' sapendo che

$$TC(1) = (t_1 - t_1) + (d_1) = (t_1 + d_1) - t_1$$

$$TC(2) = (t_2 - t_1) + (d_2) = (t_2 + d_2) - t_1$$

$$TC(3) = (t_3 - t_1) + (d_3) = (t_3 + d_3) - t_1$$

$$C(S') = \frac{1}{3} \sum_{j=1}^3 TC(j) = \frac{1}{3} \sum_{j=1}^3 ((t_j + d_j) - t_1) = \frac{1}{3} \left(\sum_{j=1}^3 (t_j + d_j) - \sum_{j=1}^3 t_1 \right) = C(S) - t_1$$

Quindi abbiamo ottenuto un costo minore riducendo l'idle time

OSS

Da questa dimostrazione deduciamo che le task devono essere consecutive e quindi possiamo rappresentare una soluzione come una permutazione di Task

SOLUZIONE AMMISSIBILE

S è soluzione ammissibile per $P(T) \Leftrightarrow S$ è permutazione di T .

$$S = i_1, i_2, \dots, i_m$$

Il cui costo sarà

$$C(S) = \frac{1}{m} \sum_{j=1}^m T(C_j) = \frac{1}{m} \sum_{j=1}^m \sum_{k=1}^{j-1} d_k$$

DECOMPOSIZIONE DEL PROBLEMA

Per il problema $P(T)$ potremmo cercare una soluzione come

$$S(T) = i_1 \circ S(T \setminus \{i_1\})$$

Questa soluzione sarebbe assolutamente scorretta per un approccio dinamico che studia tutti i possibili sottoproblemi (che nel caso di un insieme come T sono $2^{|T|}$).

Tuttavia l'approccio greedy conosce a priori la soluzione ottima per il problema quindi non complica l'esecuzione perché prenderemo comunque $|T|$ scelte

la dimostrazione dei 3 teoremi è banale, vediamo velocemente

DIMOSTRAZIONE ①

Ricordiamo che $S = i_1 \dots i_m$ è soluzione ammissibile se è permutazione di T .

Se $S = i_1 \cdot S(T \setminus \{i_1\})$ ovviamente $S(T \setminus \{i_1\})$ sarà permutazione di $T \setminus \{i_1\}$.

DIMOSTRAZIONE ②

Se $S(T \setminus \{i_1\})$ è permutazione di $T \setminus \{i_1\}$, concatenando i_1 avremo sicuramente una permutazione di T

DIMOSTRAZIONE ③

$S(T)$ ottima per $P(T) \Rightarrow S(T \setminus \{i_1\})$ ottima per $T \setminus \{i_1\}$

$$C(S(T)) = \frac{1}{m} \sum_{j=1}^m \sum_{k=1}^J d_k$$

P. a. sia $S(T)$ ottima ma $S(T \setminus \{i_1\})$ no.

Dobbiamo definire la funzione di costo ricorsivamente:

$$C(S(T \setminus \{i_1\})) = \frac{1}{m-1} \sum_{j=2}^m \sum_{k=1}^j d_k$$

Estraiamo il primo termine con $j=1$ in $C(S(T))$

$$C(S(T)) = \frac{1}{m} \left(\sum_{k=1}^1 d_k + \sum_{j=2}^m \sum_{k=1}^j d_k \right)$$

Estraiamo il primo termine con $k=1$ in $C(S(T))$

$$C(S(T)) = \frac{1}{m} \left[d_1 + \sum_{j=2}^m \left(d_1 + \sum_{k=2}^j d_k \right) \right] = \frac{1}{m} \left[m(d_1) + \sum_{j=2}^m \sum_{k=2}^j d_k \right]$$

Il secondo termine (moltiplicato per $\frac{1}{m}$) è molto simile a $C(S(T \setminus \{i_1\}))$. Per verificare l'ugualianza possiamo dire che

$$C(S(T)) = d_1 + \frac{m-1}{m} C(S(T \setminus \{i_1\}))$$

Possiamo completare ora la dimostrazione:

$$\exists S'(T \setminus \{i_1\}) \mid C(S'(T \setminus \{i_1\})) < C(S(T \setminus \{i_1\}))$$

Per il termine ② possiamo definire:

$$C(S'(T \setminus \{i_1\})) = d_1 + \frac{m-1}{m} C(S(T \setminus \{i_1\}))$$

Ma per la definizione ricorsiva di $C(S(T))$ e sapendo che

$$\underline{C(S'(T \setminus \{i_1\}))} < \underline{C(S(T \setminus \{i_1\}))} \quad \text{allora} \quad C(S'(T \setminus \{i_1\})) > C(S(T \setminus \{i_1\}))$$

ASSURDO!

POLITICA GREEDY

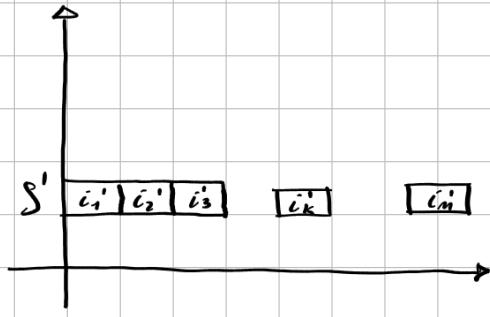
la politica che potremmo scegliere e quella di scegliere il task di durata minima dell'insieme delle task.

In pratica la soluzione e' un'ordinamento sulle task quindi risolvibile in tempo $\Theta(n \lg n)$

Dimostriamo la validita' della politica

DI MOSTRAZIONE

Preso $S(T) = i_1 \dots i_m$ ottima dimostriamo che $\exists S'(T) = i'_1 i'_2 \dots i'_m$ e' ottima con i'_j preso con la nostra politica.



Se $i_1 = i_1'$ abbiamo finito, supponiamo quindi $i_1 \neq i_1'$

i_1 sarà presente in un'altra posizione, immaginiamo la k-esima

Scambiamo i_1' e i_k' (i_1) ottenendo praticamente $S(T)$.

Sappiamo che $d_{i_1'} \leq d_{i_1}$ (per la politica).

Il tempo di terminazione di i_k' in S' e i_1' in S è lo stesso (perché la somma dei tempi di esecuzione è la stessa).

Quindi se c'è presente qualche differenza c'è prima delle task scappate, non dopo (che saranno assolutamente identiche)

Ma denando i_k' più di i_1' , possiamo dire che

$$\forall 1 \leq j, e < k \quad TC(i_j') \geq TC(i_e') \quad \forall i_j \in S \wedge i_e \in S'$$

Ma quindi la sommatoria totale di S sarà maggiore di S'

Essendo S ottima, S' è ottima a sua volta.

PROBLEMA DELLO ZAINO FRAZIONARIO

Sia $O = \{i_1, \dots, i_n\}$ insieme degli oggetti con costo $c(o_i)$ e peso $p(o_i)$.

In fine, sia W la capacità dello zaino.

In questo caso possiamo prendere solo una parte di oggetto.

Una soluzione ammessa sarà $S(O, W) = (i_1, q_1), \dots, (i_k, q_k)$

con q_j quantità e

- $i_j \in O \quad \forall i \leq j \leq k$
- $i_j \neq i_e \quad \forall j \neq e$
- $0 \leq q_j \leq P(i_j) \quad \forall i \leq j \leq k$
- $\sum_{j=1}^k q_j \leq W$

COSTO DI UNA SOLUZIONE

Seppur il concetto di costo è lo stesso, dobbiamo ridefinire il costo

$$C(S(O, W)) = \sum_{j=1}^k \frac{c(i_j)}{P(i_j)} \cdot q_j$$

Ricordiamo che una soluzione ommissibile per $P(O, W)$ e':

$S(O, W) = (i_1, q_1) \dots (i_k, q_k)$ tale che

$$\textcircled{1} \quad \{O_{i_1}, \dots, O_{i_k}\} \subseteq O$$

$$\textcircled{2} \quad q_{j_n} > 0 \quad \forall j \in \{1, \dots, k\}$$

$$\textcircled{3} \quad \sum_{j=1}^k q_j \leq W$$

OSS

Nonostante sia stato ampliato lo spazio delle soluzioni amm.

rispetto allo zaino 0-1, questo rilassamento ci aiuta per l'
appuccio greedy.

Questo e' comune perch' la complessita' del problema dipende
dai vicini posti piuttosto che dal numero di soluzioni disponibili.

DECOMPOSIZIONE RICORSIVA

Possiamo decomporre una soluzione nel seguente modo:

$$S(O, W) = (i_1, q_1) \cdot S(O \setminus \{i_1\}, W \setminus q_1)$$

Dimostrazione la validita'

DIMOSTRAZIONE ①

Sia $S(O, W) = (i_1, q_1) \cdot (i_2, q_2) \dots (i_k, q_k)$ amm. per $P(O, W)$

Dimostriamo che $S(O \setminus \{i_1\}, W \setminus q_1)$ e' ammissibile:

$$\textcircled{1} \quad \left\{ O_{i_2}, \dots, O_{i_k} \right\} \subseteq O \setminus \{O_{i_1}\}$$

\textcircled{2} ovvio

$$\textcircled{3} \quad \sum_{j=1}^k q_j \leq W \iff q_1 + \sum_{j=2}^k q_j \leq W \iff \sum_{j=2}^k q_j \leq W - q_1$$

DIMOSTRAZIONE ②

Sia $S(O \setminus \{i_1\}, W \setminus q_1) = (i_2, q_2) \dots (i_n, q_n)$ ammissibile:

\textcircled{1} $O_{i_1} \in O$ quindi e' verificato

\textcircled{2} ovvio

\textcircled{3} Stessa dimostrazione di prima al contrario

DIMOSTRAZIONE ③

Solita dimostrazione per assurdo.

POLITICA GREEDY

la politica che utilizziamo sceglie come primo oggetto quello che maximizza il costo per unità di peso:

$$\max \left(\frac{V_j}{P_j} \right) \quad V_j \in \{1, \dots, n\}$$

Ne prendiamo la massima quantità, quindi la prima scelta sarà:

$$(i_1, q_1) \mid q_1 = \min(W, P_{i_1})$$

l'algoritmo prenderà una permutazione di O , calcolerà per ogni oggetto il costo unitario e ordineremo in ordine crescente

Quindi se posso prendere una quantità W dell'oggetto col max valore attuale, il problema termina, altrimenti se prendo il max P_j e passo alla prossima scelta $W - P_j$.

Il costo sarà $\Theta(n \lg n)$ (ordinamento) + $\Theta(n)$ (scelte)

OSS

Il costo per lo zaino 0-1 era $\Theta(nW)$, quindi risulta più efficiente se $\lg n > W$.

Il fatto di avere una funzione di tempo che dipende da un

solo parametro c' è un grande valore aggiunto, perché da una soluzione uniforme per ogni istanza indipendentemente dal valore di w .

Dimostriamo la validità della politica

DIMOSTRAZIONE

la soluzione al problema $P(0, w)$ è:

$$(i_1, q_1) \mid \bullet \frac{v_{i_1}}{p_{i_1}} \geq \frac{v_j}{p_j} \quad \forall j (1, \dots, n) \quad (\text{massimo costo per peso})$$

$$\bullet q_i = \min \{w, p_i\}$$

L'approccio dimostrativo è sempre lo stesso: supponiamo una soluzione ottima che ha prima scelta non quella greedy, scambiando appunto queste due otteriamo una soluzione non peggiore.

Sia $S(0, w) = (i'_1, q'_1), \dots, (i'_n, q'_n)$ ottima.

Si noti che $\sum_{j=1}^n q'_j \leq w$, non possono quindi sostituire banalmente la scelta greedy con la prima (potremmo eccedere di peso)

Questo vale solo se $q_i < q'_i$ (q_i è della la scelta greedy)

Quindi cambiamo l'oggetto ma non la quantità:

$$S(0, w) = (i_1, q'_1), \dots, (i_k, q'_k)$$

Questa conserva sicuramente i vincoli.

Per quanto riguarda il costo delle scelte, sappiamo che differiscono solo sulla prima scelta.

Per la nostra politica sappiamo che

$$\frac{v_{i_1}}{p_{i_1}} \geq \frac{v_j}{p_j} \quad \forall j \in \{1, \dots, n\}$$

Quindi sapendo che nel nostro caso la quantità è la stessa, (i_1, q'_1) avrà per forza costo maggiore e quindi

Questa però non è la scelta greedy perché $q'_1 < q_1 \leq w$.

Per transitività $q'_1 < w$ quindi nello zaino c'è sicuramente spazio.

Ha potuto scegliere mai le quantità degli oggetti (perché lo zaino è frazionario) potremo aumentare il costo dello zaino fino a pareggiare w .

Finché non colmiamo il gap tra q_1 e q'_1 (e w quindi) aumentiamo q'_1 aggiungendo q'_2 , se $q'_1 + q'_2 < q_1$ continuo.

NETWORK FLOW

I problemi di network flow sono problemi di reti (di qualsiasi tipo) basate su rappresentazioni a grafo.

Ad esempio una rete stradale puo' essere vista come un grafo dove gli archi sono le strade e i nodi sono gli snodi.

L'idea di **FLUSSO** dipende dal singolo problema: in una rete stradale potrebbe essere il flusso di traffico, per calcolatori potrebbe essere il numero di pacchetti inviati.

Generalizzando, il problema e' rappresentabile da un grafo pesato orientato dove il peso di ogni arco e' strettamente positivo ed e' detto **CAPACITA'**.

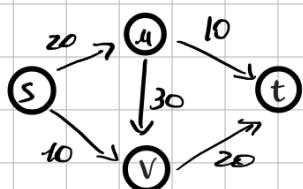
Assumeremo di avere 2 nodi source e target tali che $\forall u \in V \ (u, s) \in E \wedge (t, u) \in E$.

Inoltre assumeremo che ogni nodo ha almeno un arco entrante e uno uscente (eccezione fatta per source e target)

FUNZIONE DI FLUSSO

Il flusso è rappresentato da una funzione che associa ad ogni arco quanto flusso ha passato per quell'arco (a differenza della capacità che associa ad un arco il massimo flusso che può passare).

E.s.



• $f((s,u)) = 20 \quad f((u,v)) = 20 \quad f((v,t)) = 20$

TUTTO il flusso che parte da s deve arrivare a t (in questo caso 30).

Bondamente per un nodo dove "entrare" e "uscire" lo stesso flusso.

Quindi la LEGGE DI CONSERVAZIONE DEL FLUSSO sarà:

$$\sum_{(u,v) \in E} f(u,v) = \sum_{(v,z) \in E} f((v,z)) \quad \forall v \in V \setminus \{s,t\}$$

Quindi il flusso non può nascere dai nodi interni, bensì solo dalla sorgente. Analogamente vale per il target.

Possiamo dire che $0 \leq f(l(u,v)) \leq C_{(u,v)}$ $\forall (u,v) \in E$

Vediamo un altro flusso sulla stessa rete

• $f'(s,u) = 20$ $f'(u,v) = 10$ $f'(v,t) = 20$ $f'(u,t) = 10$ $f'(s,t) = 10$

L'obiettivo è **MASSIMIZZARE** il flusso che passa in una rete.

Il valore del flusso, per la conservazione, è necessariamente uguale al flusso generato da s (tutti gli archi uscenti).

(analogamente vale per il flusso assorbito da t per la conservazione)

f' risulta migliore di f perché genera 30 anziché 20

Ricordiamo alcune proprietà dei flussi:

Dato un arco $e \in E$, un flusso è una funzione $E \rightarrow \mathbb{R}$ tale che:

① $0 \leq f(e) \leq c_e$ dove c_e è la CAPACITÀ dell'arco

② $f_{(v)}^{\text{out}} = \sum_{(v,u) \in E} f_{(v,u)}$ $f_{(v)}^{\text{in}} = \sum_{(u,v) \in E} f_{(u,v)}$

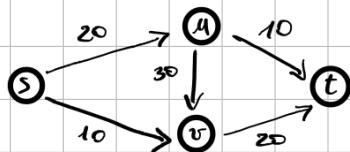
③ $f_{(v)}^{\text{out}} = f_{(v)}^{\text{in}} \quad \forall v \in V \setminus \{s, t\}$ LEGGE DI CONSERVAZIONE DEL FLUSSO

④ $f_{(s)}^{\text{out}} = 0 \wedge f_{(t)}^{\text{in}} = 0$

Il valore del flusso, che non vogliamo ottimizzare, è

$$\text{Vol}(f) = f_{(s)}^{\text{out}}$$

ANALISI DEL PROBLEMA



Consideriamo solo percorsi semplici da s a t .

L'idea è quella di migliorare progressivamente la funzione di flusso fino ad arrivare al massimo.

Potremo partire ad esempio da $f_0(e) = 0 \quad \forall e \in E$.

Prendiamo il percorso $\Pi = s \rightarrow u \rightarrow t$.

In questo percorso possiamo già migliorare così (rispettando tutti i vincoli):

$$f_1((s, u)) = 20$$

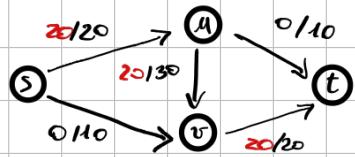
$$f_1((u, v)) = 20$$

$$f_1((v, t)) = 20$$

$$f_1((s, v)) = f_1((s, t)) = 0$$

$$f_1((u, t)) = f_1((u, v)) = 0$$

$$\left. \begin{array}{l} f_1((s, u)) = 20 \\ f_1((u, v)) = 20 \\ f_1((v, t)) = 20 \\ f_1((s, v)) = f_1((s, t)) = 0 \\ f_1((u, t)) = f_1((u, v)) = 0 \end{array} \right\} \text{Vol}(f_1) = 20$$



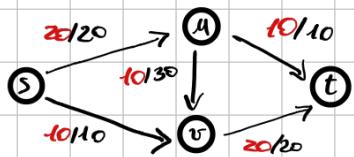
Questo flusso è valido ma non ottimale. Vediamo dove poter aumentare.

Gli archi $f_1((s, u))$ e $f_1((s, v))$ sono saturi.

Cambiando il percorso (ad esempio $t' = svt$) non risolviamo nulla.

Questa tecnica risulta quindi fallimentare.

L'idea è di distribuire il flusso di v sui due archi nel seguente modo:



Adesso $\text{Vol}(f_1) = 30$.

Dobbiamo capire come formularre questo metodo risolutivo
capendo quali archi possono aumentare il loro flusso e
quali possono ridurlo.

RETE RESIDUA

Dato un grafo G e un flusso f , $G^R = \langle V, E^R \rangle$ è detta **RETE RESIDUA** dove E^R è formato da:

● ARCHI IN AVANTI

Sono archi che hanno capacità residua maggiore di 0 in G .

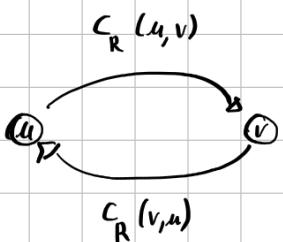
La **CAPACITÀ RESIDUA** di un arco è data da $c_R(u,v) = c(u,v) - f(u,v)$

Hanno come peso $c_R(u,v)$ stesso.

● ARCHI ALL'INDIETRO

Sono archi diretti in maniera opposta agli archi di

G con flusso positivo (insieme il flusso che "può essere trattenere")



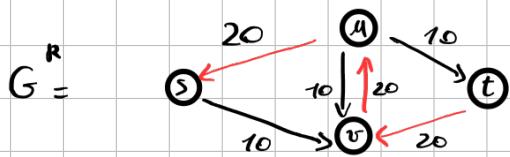
Se $f(u,v) > 0$ avremo l'arco all'indietro $v \rightarrow u$ con $c_R(v,u) = f(u,v)$.

Se $f(u,v) < c(u,v)$, avremo un arco in avanti $u \rightarrow v$ con $c_R(u,v) = c(u,v) - f(u,v)$.

$$\text{In definitiva } E^R = \{(u,v) \mid c_R(u,v) > 0\}$$

OTTIMIZZAZIONE DEL FLUSSO

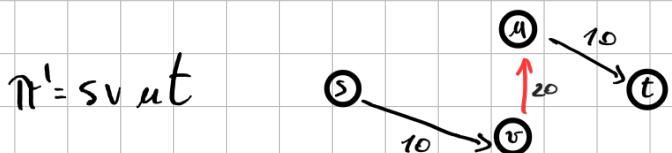
Costruiamo la rete residua di G cd flusso f_r .



Gli archi rossi sono quelli indietro, gli altri sono in avanti.

La dimensione di G^R è lineare su quella di G perché ha gli stessi nodi e $|E| \leq |E^R| \leq 2|E|$ (se ogni arco ne genera uno all'indietro)

Per seguire lo stesso ragionamento di prima, seguiamo il percorso



Far scorrere un flusso in un arco all'indietro corrisponde a trattenere il flusso.

Il massimo flusso della rete residua da s a t è 10.

Diciamo CAPACITA' MINIMA $c_{\min} = \min \left\{ c_R(e) \mid e \in \Pi \right\} = 10$

Questo ci dice che possiamo "ricavare" 10 di flow in qualche modo.

FLOW AUGMENTATION

Il processo di **AUMENTO DEL FLUSSO** consiste dati G ed ρ :

Costuiamo G^R e consideriamo il percorso da s a t in G^R

Dopo aver calcolato $c_{min}(\pi)$, la funzione di flusso sarà:

$$f_\pi(u,v) = \begin{cases} f(u,v) & (u,v) \in \pi \\ f(u,v) + c_{min}(\pi) & (u,v) \in \pi \text{ e' in avanti} \\ f(u,v) - c_{min}(\pi) & (v,u) \in \pi \text{ e' all'indietro} \end{cases}$$

Aumentiamo ogni arco in avanti del minimo aumento possibile.

Viceversa con gli archi all'indietro.

Quindi nel nostro caso specifico modificheremo $s \rightarrow v$, $u \rightarrow v$,

$$f_\pi(s,u) = 20$$

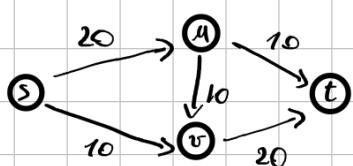
$$f_\pi(s,v) = 0 + 10 = 10$$

$$f_\pi(u,v) = 20 - 10 = 10$$

$$f_\pi(v,t) = 20$$

$$f_\pi(u,t) = 0 + 10 = 10$$

Il flusso massimo risultante e':



ALGORITMO DI FORD-FULKERSON

MAX_FLOW(G, s, t)

FOR EACH $e \in E$ DO

$$f(e) = 0$$

$G^R = \text{GRAFO_RESIDUO}(G, f)$

$\pi = \text{SEARCH_PATH}(G^R, s, t)$

DFS o BFS

WHILE $\pi \neq \emptyset$ DO

$$c_{\min} = \min \{c_R(e) \mid e \in \pi\}$$

$f = \text{AUGMENT}(f, c_{\min}, \pi)$

calcola gli incrementi e decrementi

$G^R = \text{UPDATE}(G^R, f, \pi)$

$\pi = \text{SEARCH_PATH}(G^R, s, t)$

verifica che ci sia del flow residuo

RETURN f

CORRETEZZA E COSTO DELL'ALGORITMO

Sono naturali 2 domande riguardo l'algoritmo

① L'algoritmo termina?

② Se sì, quanto costa?

③ Il flusso restituito è ottimo?

Prima di trovare queste risposte, verifichiamo che $\text{AUGMENT}(f, \pi, c_{\min})$

restituiscce un flusso (quindi l'equazione scritta rispetti i vincoli come invarianti del ciclo) e che sia effettivamente migliorato.

DIMOSTRAZIONE Ⓛ

Dimostriamo che $f_{\pi} = \text{AUGMENT}(f, \pi, c_{\min})$ rispetta i vincoli di ommissibilità che ricordiamo essere:

$$1) 0 \leq f_{\pi}(e) \leq c(e) \quad \forall e \in E$$

$$2) f_{\pi}^{out}(v) = f_{\pi}^{in}(v) \quad \forall v \in V \setminus \{s, t\}$$

$$3) f_{\pi}^{out}(t) = 0 \wedge f_{\pi}^{in}(s) = 0$$

$$f_{\pi}(u, v) = \begin{cases} f(u, v) & (u, v) \in \pi \\ f(u, v) + c_{\min}(\pi) & (u, v) \in \pi \text{ e' in avanti} \\ f(u, v) - c_{\min}(\pi) & (v, u) \in \pi \text{ e' all'indietro} \end{cases}$$

1) Per ipotesi f è un flusso quindi dobbiamo controllare in avanti e indietro di π . Sia quindi $(u, v) \in \pi$ e $(u, v) \mid (v, u) \in \pi$

$$\bullet f_{\pi}(u, v) = f(u, v) + c_{\min} \quad (\text{e' un arco in avanti})$$

Sappiamo che $c_{\min} = \min \left\{ c_e(e) \mid e \in \pi \right\} \leq c_R(u, v)$ (per definizione)

$$\underline{c_R(u, v) = c(u, v) - f(u, v)}$$

Sotto queste 3 ipotesi sappiamo che

$$f_{\pi}^{in}(u,v) \leq f(u,v) + c_R(u,v) \iff f_{\pi}^{in}(u,v) \leq \cancel{f(u,v)} + c(u,v) - \cancel{f(u,v)}$$

E' anche > 0 perché $c_{min} > 0$ e $f(u,v) > 0$.

Dimostrato per gli archi in avanti.

• $f_{\pi}^{in}(u,v) = f(u,v) - c_{min}$ (e' un arco all'indietro)

Se $f(u,v) \leq c(e) \forall e \in E$ e $c_{min} > 0$ allora $f_{\pi}^{in}(u,v) \leq c(e) \forall e \in E$ (banale)

$$c_R(u,v) = f(u,v) \text{ per ipotesi.}$$

$f(u,v) \geq c_{min}$ per definizione. Ma allora $f(u,v) - c_{min} \geq 0$

3) $f_{\pi}^{in}(s) = \sum_{(u,s) \in E} f_{\pi}^{in}(s,u)$

essendo π semplice, non contiene ripetizioni gli s.

Ma $(u,s) \in E \Rightarrow$ quindi $f_{\pi}^{in}(s) = 0$.

Analogamente vale per $f_{\pi}^{out}(t)$.

2) Prendiamo $v \in V \setminus \{s,t\}$.

Se v è π i suoi archi restano immutati quindi la proprietà è verificata per ipotesi.

Supponiamo $v \in \pi$.

$$\pi = s \dots v \dots t \in G_R$$

Per come è fatto π potremmo cambiare solo due archi che coinvolgono v ,
(altrimenti avremmo ripetizioni) uno in ingresso e uno in uscita, chiamiamoli α e β .

Abbiamo 4 possibilità:

- ① Sono entrambi in avanti
- ② Sono entrambi a ritraro
- ③ α in avanti, β a ritraro
- ④ β in avanti, α a ritraro

① Ipotizziamo di avere $\alpha \xrightarrow{v} \beta$

$$f_\pi((u, v)) = f((u, v)) + c_{\min}$$

$$f_\pi((v, z)) = f((v, z)) + c_{\min}$$

Essendo questi gli unici archi modificati, abbiamo

$$f_\pi^{out}(v) = f^{out}(v) + c_{\min} \quad \wedge \quad f_\pi^{in}(v) = f^{in}(v) + c_{\min}$$

Essendo incrementati nella stessa quantità (c_{\min}), varrà la proprietà 2

Analogamente vale col caso ②

$$③ f_{\overline{u}}((u,v)) = f_u((u,v)) + c_{HIN}$$

$$f_z((z,v)) = f_z((z,v)) - c_{HIN}$$

Viene modificato (z,v) e non (v,z) perché l'arco è a ritroso in G^R , quindi in G modificherebbe appunto (z,v) .

Sia (z,v) che (u,v) sono entranti in v , quindi varrà ancora

$$f_{\overline{u}}^{out}(v) = f_z^{out}(v).$$

Per $f_{\overline{u}}^{in}(v)$ non avremo modifiche perché sommando $f_{\overline{u}}((u,v))$ e $f_z((z,v))$ si annulleranno le modifiche ($+c_{HIN} - c_{HIN} = 0$) quindi avremo $f_{\overline{u}}^{in}(v) = f_z^{in}(v)$.

DIMOSTRAZIONE 0.5

Dimostriamo che, sia $f_{\pi} = \text{AUGMENT}(f, \pi, c_{\min})$, $\text{val}(f_{\pi}) > \text{val}(f)$ ricordando che π è un percorso da s a t in G_f .

Ricordiamo che $\text{val}(f_{\pi}) = \sum_{e=(s,u) \in E} f_{\pi}(e)$ e $c_{\min} = \{c_f(e) \mid e \in \pi\} > 0$.

Essendo π un percorso semplice da s a t , avrà la forma

$\pi = (s, u) \dots (x, t)$ per qualche $u, x \in V$.

(s, u) sarà in avanti o a ritroso ma non può essere a ritroso perché dovrebbe esistere $u \rightarrow s \in G$ con capacità > 0 (impossibile per ipotesi).

Ma se è in avanti, dopo l'aumento $f_{\pi}((s, u)) = f((s, u)) + c_{\min}$ con $c_{\min} > 0$.

Tutti gli altri archi uscenti da s sono invariati (non sono presenti ripetizioni di s in π), quindi:

$$f_{\pi}^{\text{out}}(s) = f(s) + c_{\min} \quad (\text{per via di } f_{\pi}((s, u))) \quad \text{e di conseguenza} \quad \text{val}(f_{\pi}) > \text{val}(f)$$

OSS

Questa dimostrazione ci aiuta a dimostrare la terminazione dell'algoritmo ma non basta da sola.

Infatti noi sappiamo che $\text{val}(f) \leq C$, ma $\text{val}(f)$ potrebbe tendere a C senza mai raggiungerlo (infiniti flussi = infiniti valori reali che sappiamo essere densi e quindi puo' convergere)

(cio' non vole cam gli interi.

Se riuscissimo a verificare che $\text{val}(f)$ sia un intero, tanto al
fatto che $\text{val}(f) \leq C$, allora saremmo sicuri che l'algoritmo termina
in quanto non possiamo generare una sequenza infinita di interi
crescente non divergente.

Per fare cio', ci serve dimostrare un'invariante del nostro
algoritmo.

INVARIANTE DELL'ALGORITMO DI FORD-FULKERSON

Nell'algoritmo di FORD-FULKERSON vengono generati solo flussi a
valore intero.

$\forall e \in E \quad f(e) \in \mathbb{N} \quad \text{se } c(a) \in \mathbb{N} \quad \forall a \in V$

DI MOSTRAZIONE

L'algoritmo ha un ciclo while, dimostriamo per induzione
sul numero di iterazioni.

● CASO BASE

$$f_0(e) = 0 \quad e \in E \quad \checkmark$$

● PASSO INDUTTIVO

Per ipotesi induuttiva $f(e)$ è a valori interi, dimostriamo che

f_{π} è a valori interi

$\forall e \in E_R \quad c(e) \in \mathbb{N}$ (perché $c(x) \in \mathbb{N} \quad x \in E$ e i flussi sono interi per ipotesi induuttiva)

Quindi $\forall \pi \quad c_{\min} = \min \left\{ c(e) \mid e \in \pi \right\}$ quindi $c_{\min} \in \mathbb{N}$.

Gli archi di G_R possono essere in avanti o a ritroso, ovvero:

$$\begin{aligned} f_{\pi}(e) &= f(e) + c_{\min} \\ f_{\pi}(e) &= f(e) - c_{\min} \end{aligned} \Rightarrow \text{sono entrambe intere}$$

DIMOSTRAZIONE ①

Dimostriamo che l'algoritmo termini sotto le ipotesi che

$$1) \quad \text{val}(f_{\pi}(e)) > \text{val}(f(e))$$

$$2) \quad \text{val}(f(e)) \leq \sum_{(s,u) \in E} c(s,u) = C \quad \forall f$$

$$3) \quad f(e) \in \mathbb{N} \quad \forall e \in E$$

P.a. l'algoritmo non termina.

Significa che genera una sequenza infinita crescente di flussi interi tutti minori di C .

Ma questo è un assurdo perché dopo n possi l'algoritmo raggiungeremo inevitabilmente il valore C , se continuasse violerebbe l'ipotesi 2.

NUMERO DI ITERAZIONI MASSIME

Sappiamo che $f_0 = 0 \quad \forall e \in E \Rightarrow \text{val}(f_0) = 0$,

dopo n possi avere f_n con $\text{val}(f_n) \leq C$ incrementando ogni volta il flusso di **ALMENO** 1.

Quindi avremo al massimo C iterazioni.

COSTO DELL' ALGORITMO DI FORD-FULKERSON

la creazione del grafo residuo costa lineare su $|G|$ (tra $|G|$ e $2|G|$)

la ricerca del percorso costa lineare su $|G|$.

l'aumento di f costa tanto quanti sono gli archi di Π , quindi $O|V|$.

Quindi ogni iterazione costa $\Theta(|G|)$, di conseguenza anche

l'algoritmo sarà lineare su $|G|$ (in particolare $C \cdot |G|$)

OSS

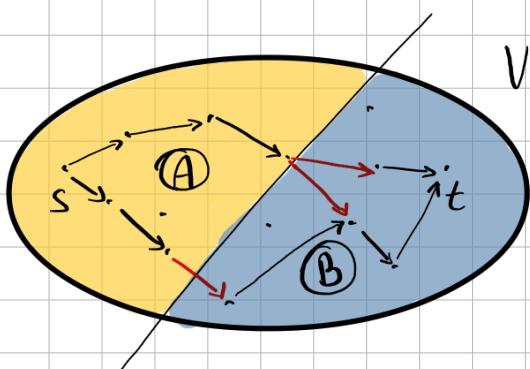
- Alcune varianti dell'algoritmo riducono il fattore C a $\log n$ rendendolo di fatto polinomiale e non **PSEUDOPOLINOMIALE** (dipende da entità differenti) oppure si può usare una variante dello zaino 0-1.
- Per dimostrare l'ottimalità dell'algoritmo ci aiuterà il suo problema duale

PROBLEMA DEL TAGLIO OTTIMO DEL FLUSSO

Sia G una rete di flusso.

Definiamo il **TAGLIO** di G come (A, B) tale che:

- $A \cup B = V$
- $A \cap B = \emptyset$
- $s \in A$
- $t \in B$



Definiamo **CAPACITÀ** di un taglio (A, B) come $\sum_{\substack{(u,v) \in E \\ u \in A \\ v \in B}} c(u,v)$

PROPRIETA'

Sia f un flusso di G e (A, B) un suo taglio.

$$\text{vol}(f) = f^{\text{out}}(A) - f^{\text{in}}(A)$$

dove $f^{\text{out}}(A)$ è il flusso che da A va verso B e viceversa $f^{\text{in}}(A)$.

$$f^{\text{out}}(A) = \sum_{\substack{(u,v) \in E \\ u \in A \\ v \in B}} f(c_{u,v})$$

$$f^{\text{in}}(A) = \sum_{\substack{(u,v) \in E \\ v \in A \\ u \in B}} f(c_{u,v})$$

DI MOSTRAZIONE

Per definizione di flusso $\text{vol}(f) = f^{\text{out}}(s)$ e $f^{\text{in}}(s) = 0$.

Quindi $\text{vol}(f) = f^{\text{out}}(s) - f^{\text{in}}(s)$

Per il principio di conservazione del flusso

$$\forall v \in V \setminus \{s\} \quad f^{\text{out}}(v) = f^{\text{in}}(v) \rightarrow f^{\text{out}}(v) - f^{\text{in}}(v) = 0 \Rightarrow \sum_{v \in A \setminus \{s\}} (f^{\text{out}}(v) - f^{\text{in}}(v)) = 0$$

Se all'insieme $A \setminus \{s\}$ (dove è definita la sommatoria) aggiungiamo s

avremo formalmente

$$\sum_{v \in A} (f^{\text{out}}(v) - f^{\text{in}}(v)) = \text{vol}(f).$$

$$f^{\text{out}}(v) = \sum_{(v,u) \in E} f(c_{v,u}) \quad \text{e} \quad f^{\text{in}}(v) = \sum_{(u,v) \in E} f(c_{u,v}) \quad \text{per definizione.}$$

Preni due vertici $u, v \in A$ l'arco (v, u) contribuisce sia a $f^{\text{in}}(u)$ che a $f^{\text{out}}(v)$.

Quindi nel calcolo di $\text{val}(f)$, al variare di v prima lo sommeremo (come $f_{(v)}^{\text{out}}$) e poi lo sottraiamo (come $f_{(v)}^{\text{in}}$) eliminandosi.

Quindi li possiamo escludere tutti nel calcolo di $\text{val}(f)$.

Gli unici che contano saranno quelli che attraversano il taglio.

Possiamo ridefinire $\text{val}(f)$ come

$$\text{val}(f) = \sum_{\substack{x \in A \\ y \in B}} f((x,y)) - \sum_{\substack{x \in A \\ y \in B}} f((y,x)) = f^{\text{out}}(A) - f^{\text{in}}(A).$$

OSS

- Tutto deriva dalla legge di conservazione che non permette ai nodi interni di assorbire flusso.
- Il taglio puo' essere applicato anche al nostro problema originale vedendo come partizioni $\{s\}$ e $V \setminus \{s\}$ e quindi

$$\text{val}(f) = f^{\text{out}}(s) - f^{\text{in}}(s) = f^{\text{out}}(s) \quad (\text{definizione originale}).$$

Se consideriamo invece $B = \{t\}$ e $A = V \setminus \{t\}$ allora

$$\text{val}(f) = f^{\text{out}}(A) - f^{\text{in}}(A) \quad \text{ma } f^{\text{in}}(A) = 0 \text{ perche' } t \text{ non ha archi uscenti quindi}$$

$$\text{val}(f) = f^{\text{out}}(A) = \sum_{(v,t) \in E} f((v,t))$$

TEOREMA

Sia f un flusso e (A, B) taglio di G

$$\text{val}(f) \leq C(A, B)$$

DI MOSTRAZIONE

Ogni flusso è limitato superiormente dalla capacità dell'arco che attraversa ($f(e) \leq c(e)$)

Ricordiamo che $C(A, B) = \sum_{\substack{(u, v) \in E \\ u \in A \\ v \in B}} c(u, v)$ e $\text{val}(f) = f^{out}_{(A)} - f^{in}_{(A)}$

Essendo $f^{in}_{(A)} > 0$, $\text{val}(f) \leq f^{out}_{(A)} = \sum_{\substack{(u, v) \in E \\ u \in A \\ v \in B}} f(u, v) \leq \sum_{\substack{(u, v) \in E \\ u \in A \\ v \in B}} c(u, v) = C(A, B)$

OSS

Se f è il flusso **MAXIMO** di G e (A, B) è il taglio di capacità **MINIMA**, allora ancora $\text{val}(f) \leq C(A, B)$

TEOREMA

Sia f un flusso e G una rete.

$$\exists (A^*, B^*) \text{ taglio di } G \mid \text{val}(f) = C(A^*, B^*)$$

Se questo si verifica, significa che tutti gli archi da A^* a B^* saturano le loro capacità, quindi nessuno di questi archi potrà aumentare il suo flusso.

Questo significa che il flusso individuato è proprio il flusso **MASSIMO** (altrimenti ce ne sarebbe un altro che però violerebbe l'ultimo teorema dimostrato)

OSS

Per dimostrare la correttezza dell'algoritmo, dobbiamo dimostrare che "se in G_k non esiste un percorso da s a t (condizione di terminazione dell'algoritmo) il flusso è massimo".

Sotto l'ipotesi che l'algoritmo è terminato, dimostriamo quest'ultimo teorema che ci darà anche la correttezza dell'algoritmo

DIMOSTRAZIONE ③

Per ipotesi in G_k non esiste percorso da s a t .

$A^* = \{v \in V \mid v \in \text{Reach}(G_R, s)\}$ (tutti i nodi raggiungibili da s).

$t \in B^*$ complemento di A^*

Vogliamo dimostrare che in G $\forall x, k \in A^* \wedge \forall y, z \in B^*$

$$f(x, y) = c(x, y) \wedge f(z, k) = 0$$

Se questo fosse vero, siccome $\text{val}(f) = f^{out}_{(A)} - f^{in}_{(A)}$, $f^{in}_{(A)} = 0$ e

$f^{out}_{(A)} = \sum_{\substack{(u, v) \in E \\ u \in A \\ v \in B}} c(u, v) = C(A, B)$ avremmo dimostrato che $\text{val}(f) = C(A^*, B^*)$

• Supponiamo p.a. che $f(x, y) < c(x, y)$

Esiste quindi in G_R un arco in avanti tale che $c_R(x, y) = c(x, y) - f(x, y)$

ASSURDO! perché $y \in B^*$ ma essendo $x \in A^*$, anche y è raggiungibile da s per transitività, quindi dovrebbe appartenere ad A^* .

Non potrebbe essere $f(x, y) > c(x, y)$, allora $f(x, y) = c(x, y)$

• Supponiamo p.a. che $f(z, k) > 0$

Per un ragionamento analogo, in G_R esiste l'arco a ritroso

$k \rightarrow z$ con $c(k, z) = f(z, k) > 0$.

Nuovamente k è raggiungibile da s tramite x ma non appartiene a A^*

OSS

Ne deriva che il problema del taglio minimo del flusso è
il duale del problema del flusso massimo e quindi si puo'
risolvere con FORD-FULKERSON effettuando una visita da s
su G_{f^*} (alla fine dell' algoritmo) dove i vertici neri
appartengono ad A^* e i bianchi a B^* (asintoticamente
non complica il problema)

MATCHING BIPARTITO

Il problema del matching bipartito ha in input un grafo $G = (V, E)$ (non necessariamente orientato) tale che (ϵ bipartito.)

$$\exists X, Y \mid X \cup Y = V \wedge X \cap Y = \emptyset \wedge \forall (x, y) \in E \quad x \in X \wedge y \in Y$$

BIPARTIZIONE

PROPRIETA' DEGLI ARCHI

Date le partizioni, e quindi $G = (X \cup Y, E)$

Un matching bipartito M è sottinsieme di archi tale che:

$$\forall v \in X, u \in Y \quad \exists \text{ al piu' un } e \in M \mid e = (v, u)$$

Nessun nodo è presente in più di un arco del matching

OSS

Un matching bipartito rappresenta generalmente un problema di distribuzione di risorse e task.

Una soluzione ammmissibile sarà un sottinsieme di E

che può essere anche visto come una funzione iniettiva parziale

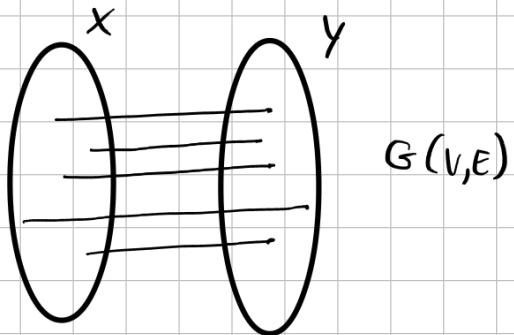
$$f: X \rightarrow Y$$

Vogliamo trovare il matching di dimensione maggiore.

ANALISI DEL PROBLEMA

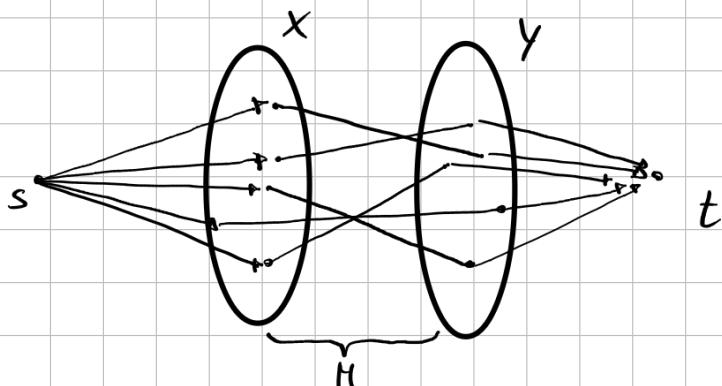
Essendo la soluzione un sottinsieme di un insieme finito, ammette sicuramente soluzioni.

Possiamo ridurre questo problema ad uno di ricerca di flusso massimo creando un opportuna rete.



Costruiamo il grafo **ORIENTATO** $G' = (V', E', c)$ tale che

- $V' = V \cup \{s, t\}$
- $E' = \{(x, y) \in E \mid x \in X, y \in Y\} \cup \{(s, x) \mid x \in X\} \cup \{(y, t) \mid y \in Y\}$
- $c: V' \times E' \rightarrow \mathbb{N}, c(e) = 1$



CORRELAZIONE TRA FLUSSO E MATCHING

La nostra soluzione sarà $M = \{(x,y) \in E \mid f(x,y) \neq 0\}$.

In pratica vogliamo dimostrare che

$$\textcircled{1} \quad \forall M \mid |M|=k \Rightarrow \exists f \text{ flusso} \mid \text{val}(f)=k$$

$$\textcircled{2} \quad \forall f \text{ flusso} \mid \text{val}(f)=k \Rightarrow \exists M \mid |M|=k$$

Così avremo una relazione diretta tra matching e flusso

DIMOSTRAZIONE $\textcircled{1}$

Sia M un matching per $G \mid |M|=k$

Per definizione di matching

$$\forall (x,y), (a,b) \in M \mid x, a \in X \text{ e } y, b \in Y \quad x \neq a \wedge y \neq b \quad (\text{niente duplicati})$$

Sia f_M la funzione di flusso in $G' \mid \text{val}(f_M)=k$

Definiamo f_M come:

$$\bullet \quad \forall (x,y) \in M \quad f_M((x,y)) = 1 \quad (\text{gli archi del matching hanno flusso 1})$$

$$\bullet \quad \forall (s,x) \in E' \quad x \text{ e' coperto da } M \Rightarrow f_M((s,x)) = 1$$

$$\bullet \quad \forall (y,t) \in E' \quad y \text{ e' coperto da } M \Rightarrow f_M((y,t)) = 1$$

Tutti gli altri hanno valore 0.

Dobbiamo dimostrare che $f: E \rightarrow \{0,1\}$ è un flusso dimostrando la validità dei vincoli di definizione:

- Rispetta fondamentalmente i vincoli di capacità

- Dimostriamo il principio di conservazione

$$f_H^{\text{out}}(x) = f_H^{\text{in}}(x) \quad \forall x \text{ coperto da } H$$

Il flusso in ingresso in x proviene solo da s per costruzione.

Il flusso vale 1 solo se x è coperto, altrimenti vale 0.

- $f_H^{\text{in}}(x)=0$ (x non coperto da H)

Dobbiamo garantire che $f_H^{\text{out}}(x)=0$.

x non essendo coperto, per la prima delle 3 ipotesi, sarà per forza 0 (altrimenti sarebbe coperto).

- $f_H^{\text{in}}(x)=1$ (x coperto da H)

P. a. $f_H^{\text{out}}(x)=0$

Per l'unicità di x in H (ha solo un arco entrante e uno uscente), l'unico arco uscente di x ha valore 0.

ASSURDO! Perché $f_H^{\text{in}}(x)=1$ per ipotesi, quindi non rispetterebbe la legge di conservazione del flusso.

Verificato che f_H è effettivamente un flusso, dimostriamo adesso che $\text{vol}(f_H) = K$.

Sappiamo che $|H| = K \Rightarrow$ in H sono coperti K vertici di X , ognuno di questi flussi vale 1 e quindi per definizione $\text{vol}(f_H) = K$

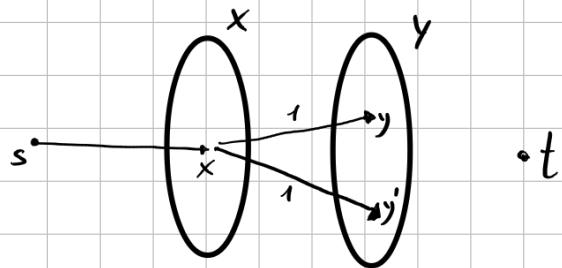
DIMOSTRAZIONE ②

Sia f flusso a valori interi in $G' \mid \text{vol}(f) = K \in \mathbb{N}$

Dimostriamo che $M_f = \{(x,y) \mid x \in X, y \in Y \wedge f((x,y)) \neq 0\}$ è un matching.

Per essere un matching ogni vertice deve comparire una sola volta.

P. a. $\exists (x,y), (x,y') \in M_f$.



Essendo entrambi gli archi nel matching, hanno entrambi valore 1.

Quindi $f(x) = 2$. **ASSURDO!** perché $f(x) = 1$ quindi non varrebbe la conservazione del flusso.

Analogamente vale se esistessero $(x,y), (x',y) \in M_f$.

Dimostriamo ora che $|H_f| = K$.

Ogni vertice di X ha un solo arco uscente verso un nodo di Y che puo' valere 1 o 0.

Per la legge di conservazione, per ogni nodo coperto, ci sono uno e uno solo arco che parte da s tale che

$$f((s, x)) = 1 \quad (\text{e quindi } f^{in}(x) = 1 = f^{out}(x)).$$

Poiche' ogni nodo con arco uscente a flusso non nullo appartiene al matching, e ogni arco ha valore 1, avremo tanto archi quanto $\text{val}(f) = K$.

APPLICAZIONE DI FORD-FULKERSON

Trovato f_{\max} con l'algoritmo di FORD-FULKERSON, per quanto appena dimostrato, possiamo ricavare un matching $H_{f_{\max}} \mid |H_{f_{\max}}| = \text{val}(f_{\max})$ dove X e Y sono le partizioni.

Siamo sicuri che $H_{f_{\max}}$ e' massimo perch'e la cardinalita' e' pari a $\text{val}(f_{\max})$, quindi se esistesse un matching maggiore avremmo un flusso maggiore, impossibile perch'e f_{\max} e' massimo.

COSTO DELL' ALGORITMO

I passaggi sono

① Costruzione di G' ($\Theta(|G|)$)

Alla dimensione di G aggiungiamo due vertici (s, t) e

nella costruzione di E' tutti gli archi da s a X e da Y a t

$$|E'| = |E| + |V|$$

② Applicazione di Ford-Fulkerson ($\Theta(|G| \cdot |V|)$)

Ricordando che C è la somma delle capacità uscenti da s ,

$$C \text{ sarà } |X| \leq |V|.$$

Quindi il costo sarebbe $\Theta(|G| \cdot C) \rightarrow \Theta(|G| \cdot |V|)$

③ Costruzione del matching $\Theta(|E|)$

Controlliamo ogni arco da X a Y se ha valore 0 o 1

Quindi in definitiva il costo è $\Theta(|G| \cdot |V|) \leq \Theta(|V|^3)$

SEGMENTAZIONE OTTIMA

Il problema prende in input un'immagine e un classificatore.

Un classificatore è un dispositivo che:

A pixel $p \in I$ restituisce una stima della probabilità che il pixel appartenga al background (b_p) o al foreground (f_p).

Vogliamo due partizioni: quello dei pixel in background e quello dei pixel in foreground.

Questa partizione non può essere presa bontemente verificando $b_p > f_p$ perché, per pixel adiacenti, se vengono inseriti in partizioni diverse viene "pagata una penalità" (nonostante questo a volte sia inevitabile, come sui "bordi" degli oggetti in foreground).

Quindi una segmentazione di I è $(B, F) \mid B \cup F = I \wedge B \cap F = \emptyset$.

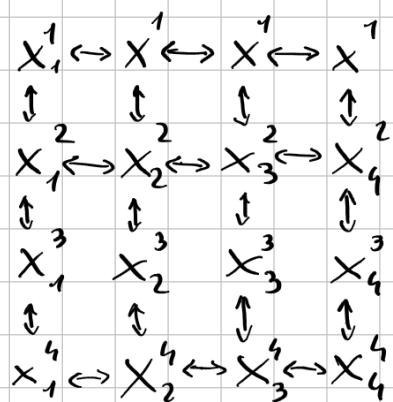
$$Q(u) = \underbrace{\sum_{x \in B} b_x + \sum_{y \in F} f_y}_{\text{STIMA CORRETTA}} - \underbrace{\sum_{x \in B} p_{xy}}_{\text{PENALITÀ}}$$

ANALISI DEL PROBLEMA

Una soluzione è una bipartizione.

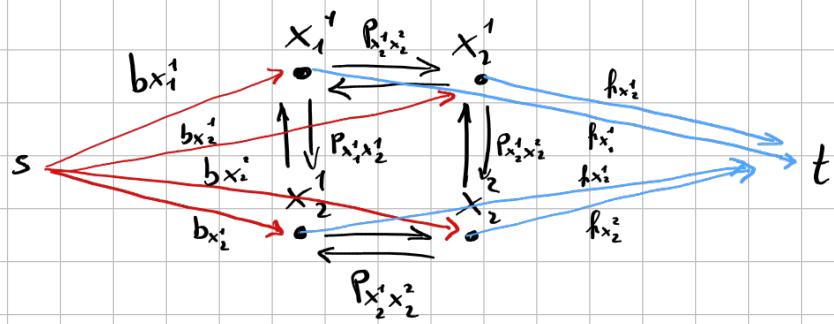
Vogliendo associare questo problema ad un problema di taglio minimo, ogni nodo è un pixel.

Possiamo vedere un'immagine come un grafo quadrato si fatto



CONVERSIONE A FLUSSO DI RETE

Prendiamo la sorgente s che raggiunge tutti i nodi e il modo t raggiunto da tutti i nodi



Dove la capacità degli archi è data da:

- b_x : gli archi entranti al flusso
- f_y : gli archi uscenti dal flusso
- P_{xy} : gli archi interni al flusso

Siccome noi cerchiamo la bipartizione tale che

$$q(B,F) = \sum_{x \in B} b_x + \sum_{y \in F} f_y - \sum_{x \in B} P_{xy} \text{ sia massimo.}$$

Tuttavia per adattarlo al problema del taglio dobbiamo cercare una quantità da minimizzare per ottenere lo stesso risultato.

Sia $Q = \sum_{x \in I} b_x + \sum_{y \in I} f_y$, presa una partizione (B,F) , definiamo

$$q(B,F) = Q - \underbrace{\sum_{x \in F} b_x - \sum_{y \in B} f_y - \sum_{x \in B} P_{xy}}$$

(rappresenta una stima dell' **ERRORE** totale su (B,F))

(fa non confondere con la stima di correttezza)

DIMOSTRAZIONE

Dimostriamo che $Q - \sum_{x \in F} b_x - \sum_{y \in B} f_y - \sum_{x \in B} P_{xy} = \sum_{x \in B} b_x + \sum_{y \in F} f_y - \sum_{x \in B} P_{xy}$

Sostituendo q e concentriamoci sul primo membro

$$\sum_{x \in I} b_x + \sum_{y \in I} f_y - \sum_{x \in F} b_x - \sum_{y \in B} f_y - \sum_{x \in B} p_{xy}$$

$$\sum_{x \in I} b_x = \sum_{x \in B} b_x + \sum_{x \in F} b_x$$

$$\sum_{y \in I} f_y = \sum_{y \in B} f_y + \sum_{y \in F} f_y$$

Sostituiamo nell' espressione

$$\begin{aligned} & \cancel{\sum_{x \in B} b_x} + \cancel{\sum_{x \in F} b_x} + \cancel{\sum_{y \in B} f_y} + \sum_{y \in F} f_y - \cancel{\sum_{x \in F} b_x} - \cancel{\sum_{y \in B} f_y} - \sum_{x \in B} p_{xy} = \\ & = \sum_{x \in B} b_x + \sum_{y \in F} f_y - \sum_{x \in B} p_{xy} \end{aligned}$$

Che e' proprio la prima definizione di $q(B, F)$

OSS

Per cui abbiamo ridefinito $q(B, F)$, per minimizzarlo
dobbiamo minimizzare la stima dell' errore

$$q^*(B, F) = \sum_{x \in F} b_x + \sum_{y \in B} f_y + \sum_{x \in B} p_{xy}$$

Abbiamo messo in evidenza il segno, ma vale banalmente

$$q(B, F) = Q - q^*(B, F)$$

PROPRIETA'

\forall taglio (A, D) di G ,

$$C(A, D) = q^*(A \setminus \{s\}, D \setminus \{t\})$$

Nel nostro problema $A \setminus \{s\} = B$ e $D \setminus \{t\} = F$

Dimostrazione

Prendiamo un taglio $(A, D) \mid s \in A \wedge t \in D$.

$$C(A, D) = \sum_{\substack{(x, y) \in E \\ x \in A \\ y \in D}} c(x, y) \quad \text{con}$$

$$\textcircled{1} \quad (x, y) = (s, y) \Rightarrow c(x, y) = b_y$$

$$\textcircled{2} \quad (x, y) = (x, t) \Rightarrow c(x, y) = f_x$$

$$\textcircled{3} \quad (x, y) \mid x \neq s \wedge y \neq t \Rightarrow c(x, y) = b_{x, y}$$

Vediamo cosa succede nel calcolo di $C(A, D)$ per i 3 tipi di archi

$\textcircled{1}$ y appartiene a D ($x = s \in A$) che è uguale a $F \cup \{t\}$.

Ma $y \neq t$ quindi $y \in F$.

$\textcircled{2}$ analogamente $x \in B$

③ Essendo i due modi diversi da set, sono evidentemente nesi di I

La sommatoria sarà quindi

$$\sum_{x \in F} b_x + \sum_{y \in B} f_y + \sum_{x \in B} p_{xy} = q^*(B, F)$$

OSS

① Con questa dimostrazione possiamo dire che il minimo taglio corrisponde al minimo $q^*(B, F)$ e, per quanto visto prima, il minimo $q(B, F)$

② Costruendo (B, F) come dimostrato, siamo sicuri che $q^*(B, F)$ sarà minima perché essendoci una corrispondenza biunivoca tra tagli e bipartizioni se ci fosse una $q^*(B', F')$ minore indurrebbe (per quanto appena dimostrato) a un taglio di capacità minore di quello minimo. ASSURDO!

③ Applicando FORD-FULKERSON risolveremo in tempo $|G|V|$ ma

$$|V| = m+2$$

$$|E| \leq m + m + \underbrace{m}_{\substack{\text{ogni da} \\ \text{set}}} = \Theta(m) \rightarrow \text{tempo} = \Theta(m^2)$$

o n° di adiacenti
pixel (caso peggiore)

COMPLESSITÀ COMPUTAZIONALE

P_i è un PROBLEMA se $P_i = \{x \mid x \text{ è una soluzione del problema}\}$

La teoria della COMPLESSITÀ COMPUTAZIONALE punta a calcolare

" $\forall x \in P_i$ quanto COSTA calcolare una soluzione a x "

dove il costo può essere definito in più modi (tempo, spazio...).

PROBLEMI

Possiamo suddividere i problemi in "facili" (TRATTABILI) o

"difficili" (INTRATTABILE) per l'insieme dei DECIDIBILI.

Esistono poi anche i problemi INDECIDIBILI (calcolo del predicato HALT) che però la teoria della complessità computazionale non tratta.

PROBLEMI TRATTABILI

Per convenzione si considera TRATTABILE qualsiasi problema

$P_i \mid \exists$ un algoritmo che calcola ogni $x \in P_i$ in tempo $O(n^k)$
dove $n = |x|$ e k è una costante ≥ 0 .

Quindi sono tutti i problemi risolvibili in tempo polinomiale.

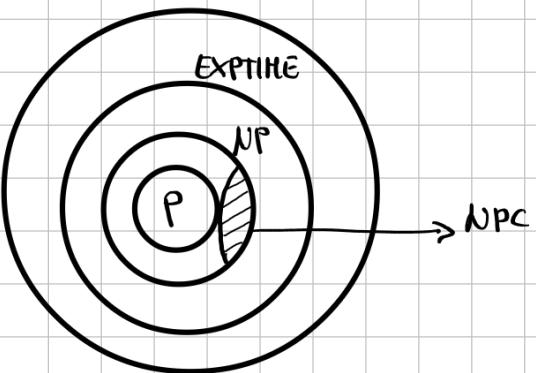
E' ovvio che esistono **GRADI** di trattabilita' ($O(n) < O(n^x)$)
ma per le proprietà necessarie non è rilevante.

Infatti tratteremo queste complessità come polinomi con le loro
proprietà di chiusura ammesse.

I problemi decidibili non risolvibili a tempo polinomiale sono
INTRATTABILI

SCHERZO GENERALE

I problemi **EXPTIME** sono risolti al massimo in tempo
esponenziale sulla dimensione ($O(k^n)$)



I problemi **NP** (Non-deterministic Polynomial) sono problemi
che sarebbero trattabili se avessimo macchine non deterministiche

Si suppone, ma non è stato dimostrato, che $P \subseteq NP$.

I problemi NP **COMPLETI** sono i problemi più difficili di NP .

Questo significa che se avessimo un algoritmo che risolve un problema in NP_C , potremmo risolvere **TUTTI** i problemi di NP .

Di conseguenza, se potessimo risolvere un problema in NP_C a tempo polinomiale, avremmo dimostrato che $P=NP$.

Queste ultime 2 soluzioni ancora non sono state trovate.

PSPACE

A differenza delle altre classi, la classe PSPACE è definita in termini di spazio: comprende tutti i problemi che richiedono SPAZIO polinomiale sull'input (o istanza)

OSS

Noi siamo certi solo che $P \subseteq \text{EXP-TIME}$, ma non abbiamo certezze sui rapporti di inclusione di NP e PSPACE

PROBLEMI DI DECISIONE

Sono problemi con risultato booleano. È possibile dire che

Un problema di ottimizzazione è un problema di decisione associato

Ese.

- Lunghezza del percorso minimo tra due vertici di un grafo.

Il problema di decisione associato sarà quello che ci dice, presi in input il grafo, due vertici e un intero K , se esiste un percorso tra i due vertici di lunghezza $\leq K$.

Per calcolare il risultato, calcoliamo la distanza tra

i due vertici e verificare che sia $\leq k$.

OSS

Si puo' quindi notare che il problema di decisione associato non e' mai piu' semplice del problema di ottimizzazione.

Questa traduzione pero' ha un costo, dobbiamo assicurci che non sia eccessiva perché sara' questa la tecnica che useremo per confrontare problemi tra loro.

- Lunghezza del percorso massimo tra due vertici di un grafo.

Ricordiammo che nel caso di grafi ciclici questo problema potrebbe non avere soluzione. Supponendo quindi un grafo aciclico, il problema di decisione associato sara' quello che ci dice presi in input il grafo, due vertici e un intero k , se esiste un percorso tra i due vertici di lunghezza $\geq k$.

OSS

Dimostriamo che un problema di decisione non e' trattabile, non sara' trattabile il problema associato.

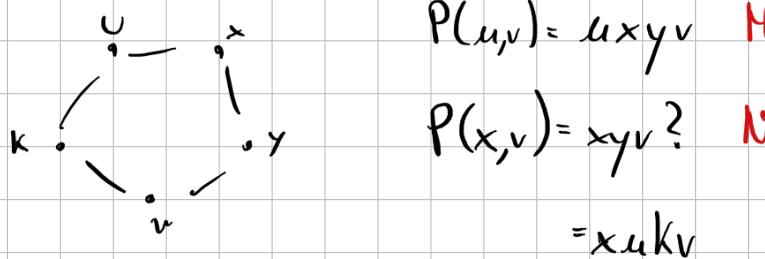
Ad esempio, quest'ultimo esempio è complesso, così come l'algoritmo di individuazione del percorso massimo.

MINPATH e MAXPATH

L'algoritmo di ricerca del percorso minimo è più semplice rispetto al secondo perché gode della proprietà di sottostruzione ottima.

Il percorso massimo no.

Esempio:



Dovremo passare al sotto problema l'insieme dei vertici non più utilizzabili, ma questo induce un numero di sottoproblemi esponenziale.

DIMENSIONE DELL' INPUT

Sia x l'input al problema, questo può assumere molte forme. Vediamo ad esempio un notizie e un grafo.

Dobbiamo considerare le possibili rappresentazioni:

Per i grafici abbiamo matrici e liste d'adiacenza.

Per un naturale potremo avere più rappresentazioni:

$$c^2(s) = 100 \text{ oppure } c^1(s) = 111.$$

Quindi $|c^2(s)| > |c^1(s)|$.

L'algoritmo di Fibonacci ha complessità $T_{\text{FIB}}(x) = \Theta(x)$, noi supponiamo che $|x| = |cx|$ (dimensione = codifica) ma la codifica talvolta può essere minore, ad esempio $c^2(x) = \log_2(x)$ quindi l'algoritmo di Fibonacci costa $T_{\text{FIB}} = \Theta(2^{|x|})$

Il problema dello zaino ha $T_z = \Theta(n \cdot w)$ che abbiamo definito polinomiale, ha complessità $T_z = \Theta(n \cdot 2^{lw})$.

ALGORITMI PSEUDO-POLINOMIALI

Sono algoritmi che hanno complessità polinomiale con una determinata codifica, altri punti possono arrivare ad essere esponenziale.

COMPLESSITÀ ALGORITMI SUI GRAPPI

Gli algoritmi sui grafpi visti non soffrono del problema della rappresentazione:

$$G = (V, E) \Rightarrow |G| = |V| + |E| \quad \text{sic } |V| = n$$

Ogni insieme V puo' essere rappresentato da una sequenza di bit in cui ogni elemento e' rappresentato con $\lg_2 x$ (con $x = \text{vertice}$ e una rappresentazione binaria), quindi avremo $|V| \leq n \lg_2 m$ bit.

Ogni arco e' rappresentato da due vertici e abbiamo al piu' $|V|^2$ archi quindi $|E| \leq n^2 (2 \lg_m)$

Calcolando $|G| = |V| + |E|$ abbiamo $|G| \leq n \lg_2 m + n^2 (2 \lg_m) \leq n^3$

Quindi le nostre approssimazioni sono sempre state corrette.

Questo perche' la rappresentazione del grafo e' polinomiale sul numero di nodi (a differenza dei numeri naturali).

OSS

Assumeremo ogni input per problema come una stringa di bit

ALGORITMO DI SOLUZIONE

E' un algoritmo che dato un problema e un'istanza restituisce la **SOLUZIONE** per quell'istanza ($A_{P_1}(x)$).

ALGORITMO DI VERIFICA (per P_1)

E' un algoritmo $B_{P_1}(x, c)$ con x =istanza e c =**CERTIFICATO** verifica c sia una soluzione valida per l'istanza x del problema P_1 .

Il certificato ci permette di dire se x e' un'istanza sì o un'istanza no e dipende dal problema e dall'istanza.

E.s.

Stiamo supponendo **SOLO** algoritmi di decisione.

PATH ($\underbrace{G, u, v, k}_x$) restituisce si se $\exists \pi(u, v) \mid |\pi| \leq k$

Un certificato puo' essere un percorso da u a v ovvero una sequenza di vertici.

Se il certificato ha lunghezza $\leq k$ dimostra che x e' un'istanza sì.

L'algoritmo di verifica $B_{\text{PATH}}(\underbrace{(G, u, v, k)}_{\pi}, \pi)$ deve verificare che:

- $\pi[0] = u$
- $|\pi| \leq k$
- $\pi[i+1] = v$
- $\pi \in G$ (verifica che ogni coppia di π è di nodi di V e sta in E)

OSS

l'algoritmo di soluzione è molto più potente dell'algoritmo di verifica perché restituisce un'istanza sì o no, mentre l'altro verifica solo se c'è un'istanza sì.

E' anche però più complesso perché lavora sullo spazio di delle soluzioni, l'algoritmo di verifica no.

CLASSI P E NP

- la classe P è formata da problemi per i quali esiste un algoritmo di **SOLUZIONE** $A_p(x)$ che impiega tempo polinomiale.
- la classe NP è formata da problemi per i quali esiste un algoritmo di **VERIFICA** $B_{p_i}(x, c)$ tale che:

① $B_{P_2}(x, c)$ impiega tempo polinomiale su $|x|$ con $x \in P_2$

② x è istanza SI $\Leftrightarrow \exists c$ certificato $|c|=O(|x|^k) \wedge B_{P_2}(x, c)=SI$

OSS

● Un algoritmo è NP se può essere verificato in tempo polinomiale

● L'algoritmo di verifica NON è obbligato ad usare il certificato in questa definizione per verificare l'istanza.

In questo senso l'algoritmo di verifica potrebbe coincidere con quello di soluzione se può essere risolto in tempo polinomiale.

Questo non accade però per nessun problema NP ad oggi.

● Di conseguenza vale sicuramente $P \subseteq NP$

DIMOSTRAZIONE

Supponiamo $x \in P$.

Per ipotesi $\exists A_P(x)$ di soluzione in tempo polinomiale.

Verifichiamo i punti ① e ② per dimostrare $x \in NP$

① Verificato per ipotesi

② Usiamo $A_P(x)$ come algoritmo di verifica.

CERTIFICATI

$\forall x \in P_n$ Esistono certificati perché la lunghezza dell'input è finita e per ipotesi $|c| \leq |x|^k$.

$A_x(x)$ (algoritmo di soluzione) potrebbe essere quindi bivalente
FOR EACH certificato c

IF $B_x(x, c) = \text{SI}$

RETURN SI

RETURN NO

Quindi è sempre possibile ottenere un algoritmo di SOLUZIONE
conoscendo l'algoritmo di VERIFICA

RIDUZIONE DI UN PROBLEMA

Dati 2 problemi X e Y, diremo

$$Y \leq_p X$$

Se e solo se

- X NON è più facile di Y oppure
- Y NON è più difficile di X

La definizione formale di \leq_p e':

$y \leq_p X$ se \exists funzione $R: Y \rightarrow X \mid \forall y \in Y$

• $R(y)$ e' un' istanza SI di $X \Leftrightarrow y$ e' un' istanza SI di Y .

• R puo' essere calcolata in tempo polinomiale su $|Y|$

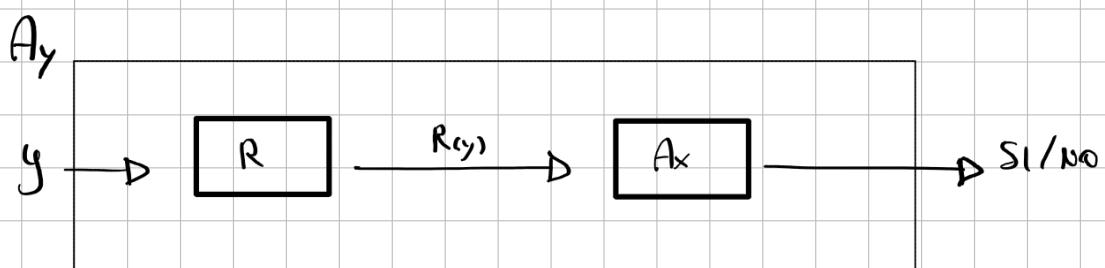
OSS

R deve richiedere tempo polinomiale perche' altrimenti la complessita' degli $y \in Y$ sera' "assorbita" dall'algoritmo di risoluzione.

TEOREMA

Sia $x \in P$, $y \leq_p X \Rightarrow y \in P$

Per dimostrarlo dobbiamo trovare un algoritmo di soluzione che impiega tempo polinomiale saperendo che $\exists A_x(x)$ polinomiale e $\exists R: Y \rightarrow X \mid |R(y)| \leq |Y|^k$



(riduzione di un problema)

A_y (algoritmo di soluzione per y) richiede tempo polinomiale

DIMOSTRAZIONE

Sappiamo che $|R(y)| \leq |y|^k \wedge A_x$ impiega $\underbrace{O(|R(y)|^{k'})}$ con $k, k' > 0$
 A_x polinomiale

Il tempo di risoluzione di A_x sull'istanza $R(y)$ è $\leq (|y|^k)^{k'}$

A_y impiegherà $|y|^k + |y|^{k''} \cdot |y|^{k''} = |y|^{k''}$ $k'' > 0$
↑ {
 Trasformazione Risoluzione A_x

CONFRONTARE DUE PROBLEMI

Abbiamo dimostrato che

$$(x \in P) \wedge (y \leq_P x) \Rightarrow y \in P$$

Applicando De Morgan possiamo dire che

$$\neg(y \in P) \Rightarrow \neg(x \in P) \vee \neg(y \leq_P x)$$

E quindi vale sicuramente

$$y \in P \wedge y \leq_P x \Rightarrow x \in P$$

Questa implicazione ci permette di confrontare la

complessità di due problemi in quanto ci dice che:

X è almeno difficile quanto Y se esiste una riduzione polinomiale da Y a X .

DI MOSTRAZIONI DI RIDUCIBILITÀ

Vogliamo dimostrare che Y e X sono egualmente difficili riducendoli reciprocamente l'uno nell'altro.

Scelgiamo due problemi su graphi:

① VERTEX COVER MINIMO

Dato $G = (V, E)$

Un vertex cover è un insieme $V' \subseteq V$ | $\forall (x, y) \in E \quad x \in V' \vee y \in V'$

Il problema di decisione associato sarà:

$VC(G, k) = S \Leftrightarrow \exists$ un vertex cover di taglia $\leq k$

② MAXIMAL INDEPENDENT SET

Dato $G = (V, E)$

Un indipendent set è un insieme $V' \subseteq V$ | $\forall (x, y) \in E \quad x \in V' \vee y \in V'$

Il problema di decisione associato sarà:

$IS(G, k) = S \Leftrightarrow \exists$ un indipendent set $\geq k$

Vogliamo dimostrare $VC \leq_p IS$ e $IS \leq_p VC$

Questo si può fare dimostrando seguente proprietà:

$$VC(G, k) \text{ è SI} \Leftrightarrow IS(G, |V|-k) \text{ è SI}$$

DIMOSTRAZIONE

\Rightarrow

$$(G, k) \text{ è istanza SI di } VC \Rightarrow \exists V' \subseteq V \mid |V'| \leq k \wedge \forall (x, y) \in E \quad x \in V' \vee y \in V'$$

$\forall (x, y) \in E \quad x \in (V \setminus V') \vee y \in (V \setminus V')$ per definizione di VC

Questo implica che $(V \setminus V')$ sia proprio un indipendent set.

$$\text{Se } |V|=n \text{ e } |V'| \leq k \Rightarrow |V \setminus V'| \geq |V|-k$$

Che significa proprio che $IS(G, |V|-k)$ è SI.

\Leftarrow

Simmetricamente

$$IS(G, k) \text{ è SI} \Rightarrow \exists V' \subseteq V \mid |V'| \geq k \wedge \forall (x, y) \in E \quad x \in V' \vee y \in V'$$

Questo implica che $\forall (x, y) \in E \quad x \in (V \setminus V') \vee y \in (V \setminus V')$.

$(V \setminus V')$ è proprio un VC di cardinalità $|V \setminus V'| \leq |V|-k$

Quindi in definitiva trasmuto il VCM avremo anche il HIS.

Sappiamo ora che i due problemi hanno la stessa complessità

TEOREMA

Siamo $X, Y \in \mathcal{Z}$ 3 problemi

$$X \leq_p Y \wedge Y \leq_p Z \Rightarrow X \leq_p Z$$

Dimostrazione

$$X \leq_p Y \Rightarrow \exists R_y : X \rightarrow Y$$

$$Y \leq_p Z \Rightarrow \exists R_z : Y \rightarrow Z$$

Per composizione possiamo ottenere $R_{xz} : X \rightarrow Z$ in particolare

$$R_{xz} = R_z(R_y(x))$$

$$\underbrace{\quad}_{\in Z} \quad \underbrace{\quad}_{\in Y}$$

Quindi la funzione è ben posta, dimostriamo che è polinomiale

Sia $|X| = n$, in tempo al più n^k ($k > 0$) ottieniamo $R_y(x)$.

Sappiamo che $|R_y(x)| \leq n^k$ e che $R_z(y)$ impiega tempo $|y|^{k'}$ $k' > 0$

Di conseguenza $R_z(R_y(x))$ si ottiene in tempo $|R_y(x)|^{k'}$ e quindi

$$\text{al più } (n^k)^{k'} = n^{k''}$$

OSS

Possiamo usare quindi delle riduzioni intermedie per confrontare problemi non facilmente riducibili.

PROBLEMI NP-COMPLETI

Un problema è appartenente ad NPC se:

$$\textcircled{1} \quad x \in \text{NP}$$

$$\textcircled{2} \quad y \leq_p x \quad \forall y \in \text{NP} \quad \text{NP-HARDNESS}$$

la seconda proprietà ci dice che sono i più complessi perché ogni problema di NP può essere ridotto a loro e quindi non possono essere più difficili.

OSS

Se ottenessi un algoritmo polinomiale per risolvere un qualsiasi problema NPC, avremmo dimostrato che $P=NP$ perché potremmo risolvere polinomialmente ogni problema di NP riducendo in tempo polinomiale.

SODDISFACIBILITÀ DEI CIRCUITI

E' una variante del problema esposto nel **TEOREMA DI COOK**

Abbiamo un circuito combinatorio formato da porte AND, OR e NOT che prendono in input valori booleani.

Ogni porta logica calcola una funzione booleana.

$$\text{AND: } f(x,y) = x \wedge y$$

$$\text{OR: } f(x,y) = x \vee y$$

$$\text{NOT: } f(x) = \neg y$$

Un circuito C si dice **SODDISFACIBILE** se esistono valori booleani per i fili di input delle porte tali che il filo di output produce valore 1.

Il problema che verifica se un circuito è soddisfacibile è detto **CIRCUIT-SAT** (istruzione del nostro problema)

Esiste una dimostrazione di NP completezza per la soddisfabilità dei circuiti che mai vediamo in maniera parziale perché dimostrare bene la proprietà ② richiede strumenti che mai non abbiamo

DIMOSTRAZIONE

Il circuito C ha una dimensione data dalla somma del numero di fili + il numero di porte.

Per dimostrare la NP COMPLETEZZA dobbiamo verificare le due proprietà della definizione

① CIRCUIT-SAT E NP

Possiamo prendere come certificato un assegnamento di valori booleani per i fili di input delle porte:

$$\sigma : \{x_1, \dots, x_n\} \rightarrow \{0, 1\}$$

L'algoritmo di verifica sarà semplicemente la sostituzione dei valori del certificato ai fili e la valutazione delle porte (e quindi dell'output del circuito).

In tempo $O(|C|)$ possiamo sapere se CIRCUIT-SAT di x è un'istanza SI ma $|C| \leq n^k$ (dimensione di C) quindi l'algoritmo di verifica impiega tempo polinomiale.

② NP HARDNESS di CIRCUIT-SAT

Preso $x \in \text{NP}$ dobbiamo dimostrare che esiste R_x tale che

- $R_x: x \rightarrow \text{CIRCUIT-SAT}$

- R_x impiega tempo polinomiale

- x è istanza sì di $X \Leftrightarrow R_x(x)$ è un'istanza sì di CIRCUIT-SAT

Per l'appartenenza di x a NP $\exists A_x(x, c)$ tale che

- 1) A_x impiega tempo polinomiale

- 2) x è istanza sì $\Leftrightarrow \exists c \mid |c| \leq |x|^k \wedge A_x(x, c) = \text{sì}$

Dobbiamo dimostrare che fornendo solamente x a A_x possiamo ottenere

$R_x(x)$ tale che x è istanza sì $\Leftrightarrow R_x(x)$ è istanza sì di CIRCUIT-SAT.

Il certificato dev'essere per forza polinomiale su x perché deve valere

$$T_{A_x}(x) \leq |x|^k \quad \text{per } k > 0$$

Sapendo già per ipotesi che esiste un algoritmo di verifica che preso un certificato ci dice se x è istanza sì, $R_x(x)$ deve costruire un circuito che simula questo stesso algoritmo di verifica con certificato variabile e x fisso.

la funzione $A_x(x, c)$ (con x fissato) ha come dominio l'insieme dei certificati $\{0,1\}^m$ (con $m \leq |x|^k$ e $K > 0$) e codominio $\{0,1\}^l$.

Abbiamo quindi una funzione booleana che prende in input un certificato, non dobbiamo costruire un circuito che faccia esattamente ciò.

Ma se i circuiti implementano funzioni booleane, quindi sicuramente anche quello dell'algoritmo di verifica.

Nel pratico $R_x(x)$ simula tutti le configurazioni del circuito

OSS

Per la definizione di \leq_p , se riusciamo a dimostrare che $\text{CIRCUIT-SAT} \leq_p y \wedge y \in \text{NP} \Rightarrow y \in \text{NPC}$.

L'idea generale per dimostrare che un problema appartiene a NPC sarà proprio sfruttare questa implicazione (non necessariamente grazie a CIRCUIT-SAT)

CLAUSOLA

In algebra booleana una CLAUSOLA è una disgiunzione di termini su $X(x, \bar{x})$:

$$t_1 \vee t_2 \vee \dots \vee t_m \quad \text{con } t_i \in \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$$

FORMA CNF

La forma CNF (conjunctive normal form) di un' espressione booleana è una congiunzione di clausole:

$$C_1 \wedge C_2 \wedge \dots \wedge C_k$$

Un'espressione booleana \exists un' espressione in CNF equivalente

La forma 3-CNF è una forma CNF dove ogni clausola contiene esattamente 3 termini

$$C_1 = t_1 \vee t_2 \vee t_3$$

3-SAT

Il problema del 3-SAT si occupa di verifica che un circuito in forma 3-CNF è soddisfacibile.

Dimostriamo la NP-COMPLETITÀ di 3-SAT

DI MOSTRAZIONE

① 3-SAT ∈ NP

DATA una formula 3-CNF, un certificato è un'assegnamento $\sigma: x \rightarrow \{0,1\}$ che associa un valore booleano ad ogni variabile dell'espressione. Un algoritmo di verifica prende la formula (φ) e un particolare assegnamento (σ) e valuta l'espressione sostituita.

Questo viene fatto in tempo lineare sull'espressione che è polinomiale.

② NP-HARDNESS

Per dimostrare la seconda proprietà dimostreremo che

$$\text{CIRCUIT-SAT} \leq_p 3\text{-SAT}.$$

Saranno una procedura che dato un circuito lo trasforma (in tempo polinomiale) in una formula equisoddisfacibile.

Aggiungiamo tante variabili booleane quante sono le porte logiche, gli input del circuito e una per l'output del circuito.

Le variabili associate alle porte assumono il valore dell'espressione volutata dalla porta.

E' sempre possibile trasformare un'espressione booleana in forma CNF.

Se ϕ_i e' la formula CNF di ogni espressione associata alle variabili, la formula CNF del circuito sara'

$$\Phi_c = \phi_1 \wedge \phi_2 \wedge \dots \wedge \phi_m \quad \text{con } m = O(|C|)$$

Φ_c **NON** e' in forma 3-CNF. Dobbiamo trasformarla modificando le ϕ_i .

Per farlo introduciamo due variabili booleane $p \neq q$.

Vediamo le seguenti implicazioni per i 2 tipi di Φ_i che possiamo generare (2-CNF e 1-CNF):

$$(x_i \vee x_j) \Leftrightarrow (x_i \vee x_j \vee p) \wedge (x_i \vee x_j \vee \bar{p})$$

$$(x_i) \Leftrightarrow (x_i \vee q \vee p) \wedge (x_i \vee q \vee \bar{p}) \wedge (x_i \vee \bar{q} \vee p) \wedge (x_i \vee \bar{q} \vee \bar{p})$$

Le variabili introdotte non provocano alcun effetto.

Φ_c e' polinomiale su $|C|$ quindi la tesi e' verificata.

INDEPENDENT SET

Independent Set è un problema NP completo.

Per farlo sfrutteremo 3-SAT.

DIMOSTRAZIONE

Come sempre dimostriamo le 2 proprietà.

① INDEPENDENT-SET ∈ NP

Un certificato può essere un insieme di vertici.

L'algoritmo di verifica deve, in tempo polinomiale, controllare che $|C| \geq K$ e che C sia indipendente.

Il primo controllo è lineare su $|C|$, il secondo può essere verificato in tempo $\Theta(|E| \cdot |C|)$ (per ogni arco $|C|$).

$$\Theta(|E| \cdot |C|) = \Theta(|E| \cdot |V|) \leq |G|^2 \text{ quindi il tempo è polinomiale su } |E|$$

② 3-SAT \leq_p INDEPENDENT-SET (NP HARDNESS)

Data la formula 3-SAT della forma

$$\Phi = C_1 \wedge C_2 \wedge \dots \wedge C_k \quad \text{con } C_i = (l_1 \vee l_2 \vee l_3) \quad l_i = \{x, \bar{x}\}$$

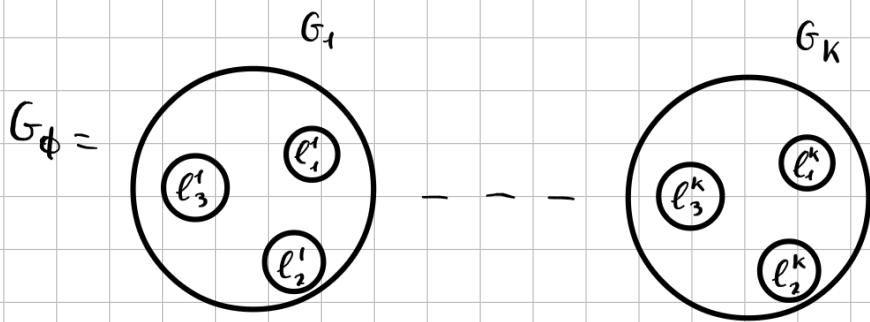
Vogliamo trovare un modo per costruire un grafo G_{Φ} che ammetta un I.S. di taglia K .

Così facendo, avremo ridotto 3-SAT a INDEPENDENT-SET e quindi $3\text{-SAT} \leq_p \text{I.S.}$

Costruiamo K sottografi, uno per ogni clausola.

Ogni grafo G_i contiene 3 vertici (uno per ogni litterale della clausola)

Si ricordi che più clausole possono condividere litterali
o tenere diversi dello stesso litterale.



Il nostro fine è costruire un Grafo G_ϕ che ammetta un independent set con capacità maggiore di K pertenente da una sostituzione a K clausole che soddisfi ϕ .

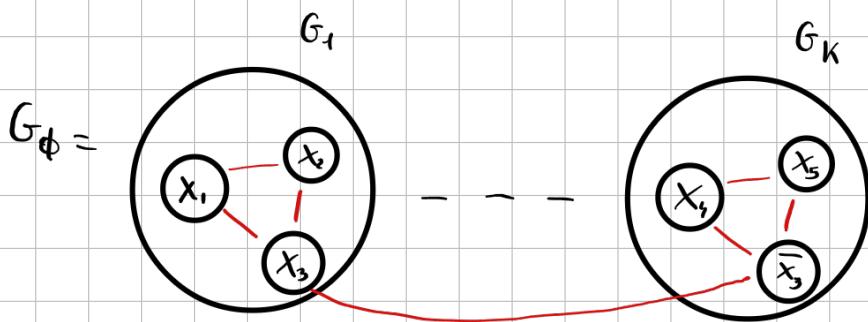
Per essere vero ϕ sappiamo dover essere vere tutte le clausole.

Per essere vera una clausola dev'essere vero almeno un

letterale.

Per garantirlo dobbiamo costruire un I.S. che prenda un vertice da ogni sottografo e sia vero il letterale associato a quel vertice (e avremo proprio K vertici).

Per garantire questa proprietà dei nodi dell'I.S. mettiamo i vertici dello stesso sottografo (così VG_i potranno avere un solo nodo) e i vertici che rappresentano letterali opposti (perché uno dei due sarebbe per forza falso).



Ora passiamo a dimostrare che

$$\phi \text{ e' soddisfacibile} \iff \exists \text{ I.S. I di } G \mid |I|=k$$

\Rightarrow

Per ipotesi $\exists \sigma$ assegnamento $| \forall 1 \leq i \leq k \sigma \text{ soddisfa } c_i \Rightarrow \sigma \text{ soddisfa almeno un letterale di } c_i$

$\forall 1 \leq i \leq k, 1 \leq j \leq 3$ sia l_j^i il letterale soddisfatto.

Costruiamo $I = \{l_{j_i}^i \mid 1 \leq i \leq k\}$ ($|I| = k$).

Ogni $l_{j_i}^i$ è associato a un vertice di G_ϕ .

Per come abbiamo costruito G_ϕ , I è proprio un I.S. di taglio K .

\Leftarrow

Dobbiamo definire $S(x)$ (funzione baverna).

$$S(x) = \begin{cases} 1 & \text{se } \bar{x} \in I \\ 0 & \text{altrimenti} \end{cases}$$

Questa sostituzione garantisce che i letterali associati ad ogni vertice di I sia vero. Quindi ogni clausola avrà almeno un letterale soddisfatto e di conseguenza ϕ sarà soddisfatto.

OSS

Avevolo dimostrato che $\text{INDEPENDENT-SETS}_p \leq_p \text{VERTEX COVER}$ e $\text{VERTEX COVER} \in \text{NP}$, allora è NPC.

SUBSET-SUM

Dato un insieme di interi V e un intero W , esiste un sottinsieme di V la cui somma è W ?

SUBSET-SUM è NP COMPLETO

DI MOSTRAZIONE

① SUBSET-SUM ∈ NP

Un certificato può essere un insieme di interi il cui calcolo della somma è lineare.

L'algoritmo di verifica controlla che questa somma faccia proprio W , in tempo lineare sull'insieme dato

② 3-SAT ≤_p SUBSET-SUM

Vogliamo trovare un algoritmo che data una formula 3-CNF ϕ a k clauses verifica che

$$\phi \text{ è soddisfacibile} \iff \exists I = \{i_1, \dots, i_m\} \subseteq V \quad \left| \sum_{j=1}^{|I|} i_j = W \right.$$

Possiamo vedere ogni intero come una sequenza di cifre.

Costruiamo una struttura nel seguente modo:

- Inseriamo 2 numeri per ogni variabile (positiva e negativa) } righe
- Inseriamo 2 numeri per ogni clausola
- Ogni numero ha
 - 1 cifra decimale per ogni variabile } colonne
 - 1 cifra decimale per ogni clausola

Le cifre dei numeri sono stabiliti come segue:

- v_{x_i} e $v_{\bar{x}_i}$ hanno 1 nella cifra x_i e 0 nelle altre
- v_{c_j} e $v_{\bar{c}_j}$ hanno 1 nella cifra c_j e 0 nelle altre

Per chiarire costruiamo una struttura partendo da:

$$\phi = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \bar{x}_3)$$

informazioni sulle variabili

supporto per la somma di C_i in W

	x_1	x_2	x_3	C_1	C_2	C_3	C_4
v_{x_1}	1	0	0	1	1	0	1
$v_{\bar{x}_1}$	1	0	0	0	0	1	0
v_{x_2}	0	1	0	0	1	0	1
$v_{\bar{x}_2}$	0	1	0	1	0	1	0
v_{x_3}	0	0	1	1	1	1	0
$v_{\bar{x}_3}$	0	0	1	0	0	0	1
v_{C_1}	0	0	0	1	0	0	0
$v_{C_1'}$	0	0	0	1	0	0	0
v_{C_2}	0	0	0	0	1	0	0
$v_{C_2'}$	0	0	0	0	1	0	0
v_{C_3}	0	0	0	0	0	1	0
$v_{C_3'}$	0	0	0	0	0	1	0
v_{C_4}	0	0	0	0	0	0	1
$v_{C_4'}$	0	0	0	0	0	0	1
W	1	1	1	3	3	3	3

In questo modo ogni numero (riga) ci dà informazioni sulla polarità e sulla clausola di appartenenza.

W deve avere per ogni decimale la somma della colonna che ci aspettiamo (ovvero 1 sola variabile positiva o negativa per ogni littorello) e 3 littoroli veri al massimo, perché i littoroli si potrebbero mettere in più clausole. Per risolvere questo problema ci aiuteremo con v_{c_i} e $v_{c'_i}$ che infatti hanno 0 sulle variabili (non devono contribuire) e 1 in corrispondenza della clausola c_i associata. In questo modo possiamo garantire che indipendentemente da quanti littoroli sono 1 in una clausola, la somma potrà essere sempre 3.

Possiamo ora dimostrare che

ϕ è soddisfacibile $\Leftrightarrow P_\phi$ è istanza SI di SUBSET-SUM

\Leftarrow

Per ipotesi $\exists C \subseteq V$

Definiamo $\sigma(x)$ che soddisfi ϕ

$$\sigma(x_i) = \begin{cases} 1 & \text{se } v_{x_i} \in C \\ 0 & \text{altrimenti} \end{cases} \quad (1 \text{ in Tabella})$$

Questa sostituzione soddisfa ϕ perché per farlo σ deve soddisfare $c_j \forall i \leq j \leq k$, e questo è verificato perché i decimali di w associati alle clausole (gli ultimi 4) sono pari a 3 mentre le variabili di supporto sommano al massimo 2 quindi una verificata ci deve essere per forza.



Per ipotesi σ soddisfa almeno un letterale per clause.

Costruiamo un certificato c per P_ϕ come segue:

c contiene $v_{x_i} \neq \sigma(x_i) = 1$, $v_{\bar{x}_i}$ altrimenti. Ogni variabile viene quindi selezionata una sola volta.

Se ϕ non rende veri tutti e 3 i letterali, inseriremo in c anche le variabili di supporto che abbiamo introdotto.

L'insieme risultante è un certificato valido per C_J

KNAPSACK

Dato un insieme O di oggetti con peso w_i e valore v_i , dare numeri naturali $K \in W$, esiste un sottinsieme di O con peso complessivo $\leq W$ e valore $\geq K$?

KNAPSACK è la versione decisionale dello ZAINO 0-1 ed è NP COMPLETO.

DI MOSTRAZIONE

① KNAPSACK ∈ NP

Un certificato può essere banalmente un insieme e in tempo lineare se $|C| \leq |O|$ possiamo calcolare la somma dei pesi e dei valori per verificare che il certificato sia valido.

② SUBSET-SUM \leq_p KNAPSACK

Dato un'istanza (V, W) definiamo la seguente istanza di KNAPSACK come $(O, \{P_i\}, \{v_i\}, W, k)$ dove

- $O = \{1, 2, \dots, n\}$ è l'insieme di oggetti
- $P_i = v_i \quad \forall 1 \leq i \leq n$

• $v_i = x_i \quad \forall 1 \leq i \leq n$ Pesi e valori coincidono

• $K = W$

x_i sono gli oggetti di V .

Un certificato per KNAPSACK sarà $c_k \subseteq \{1, \dots, n\}$ $\left| \begin{array}{l} \sum_{i \in c_k} p_i = W \wedge \sum_{i \in c_k} v_i = W \end{array} \right.$

Trovato c_k avremo proprio l'insieme di pesi (o di valori) le cui somme è proprio W

Viceversa, possiamo usare lo stesso certificato perché la somma degli oggetti sarà W , ma W è sia peso che valore, quindi è verificata la bontà del certificato.

ALGORITMI DI APPROSSIMAZIONE

Per i problemi **NON TRATTABILI**, vista la loro complessità, è difficile definire algoritmi efficienti che ci diano una soluzione esatta.

In questo ci aiutano gli **ALGORITMI DI APPROSSIMAZIONE**, ovvero algoritmi che restituiscono soluzioni sub-ottime e di cui è nota a priori "la distanza" dalla soluzione ottima.

OSS

Non esistono sempre algoritmi di ottimizzazione per ogni problema.

Gli algoritmi di approssimazione sono una sottoclasse degli algoritmi **EURISTICI** in quanto per questi ultimi non c'è noto a priori il rapporto di approssimazione.

RAPPORTO DI OTTIMIZZAZIONE

Sia X un problema di ottimizzazione difficile.

Sia C^* il costo della soluzione ottima e C il costo della soluzione sub-ottima.

Supponendo X problema di minimizzazione v'ha: $C^* \leq C$.

la "distanza" tra le due soluzioni (**RAPPORTO DI OTTIMIZZAZIONE**)

c'

$$\frac{C}{C^*} \geq 1$$

Maggiore e' il valore, maggiore e' la "distanza".

Questo valore puo' cambiare da istanza a istanza.

Data un'istanza $x \in X$ e una funzione $p(n)$, un algoritmo e' di **$p(n)$ -APPROXIMAZIONE** se per ogni istanza

$$\frac{C}{C^*} \leq p(n) \quad n = |x|$$

Analogamente varia per i problemi di massimizzazione ricordando che il rapporto di approssimazione e' sempre ≥ 1 .

In generale, dato un problema di ottimizzazione X e una funzione $p(n)$, un algoritmo si dice **$p(n)$ -APPROXIMATO** se

$$\text{MAX} \left(\frac{C}{C^*}, \frac{C^*}{C} \right) \leq \underline{p(n)} \quad \forall x \in X, n = |x|$$

Se un algoritmo restituisce sempre soluzioni con costo al piu' del doppio dell' ottimo, sara' detto **2-APPROXIMATO**

VERTEX COVER MINIMO 2-APPROXIMATO

L'intuizione per l'algoritmo è, dato un grafo, prendere casualmente un arco ed inserire momentaneamente gli estremi nel Vertex Cover e li rimuoviamo dal grafo, in questo modo al termine avremo trovato proprio un VC.

● APPROX-VERTEX-COVER(G)

$$VC = \emptyset$$

soluzione

$$E' = E$$

insieme degli archi disponibili

$$A = \emptyset$$

ci serve solo per la dimostrazione, contiene gli archi selezionati

WHILE $E' \neq \emptyset$ DO

SCEGLI $(v, u) \in E'$

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

$VC = VC \cup \{v, u\}$

$E' = E' \setminus \{(x, y) \in E \mid x=v \vee y=u\}$

rimuoviamo tutti gli archi uscenti da v e u

RETURN VC

Quest'algoritmo ha costo polinomiale su $|G|$

DIMOSTRAZIONE

Dimostriamo anzitutto che la soluzione è un vertex cover ovvero.

$$\forall (x,y) \in E \quad x \in VC \quad \forall y \in VC.$$

Nell'algoritmo tutti gli archi di A sono coperti da VC (addirittura ci sono entrambi gli estremi).

In A non veniamo rinvianti tutti archi già coperti da VC e quindi rimossi da E' .

Siano VC_i e E'_i i due insiemini all' i -esima iterazione,

$$VC_0 = \emptyset, E'_0 = E, VC_k = VC, E'_k = \emptyset.$$

Per il criterio con cui rimoviamo i nodi da E' , in E'_i ci saranno sempre gli archi non coperti da VC_i (invarianza del ciclo), quindi si tiene ovvero sicuramente un Vertex Cover.

Dimostriamo ora che il costo è polinomiale su $|G|$.

Il ciclo while ha costo al più $|E|$ perché a ogni iterazione rimoviamo sempre un arco (v,u) .

I due inserimenti in A e S sono effettuate a tempo costante e la rimozione degli archi da E' costa al più $|E|$ quindi il costo è al più.

PROPRIETA' (2-APPROSSIMAZIONE)

Se VC^* è il minimo Vertex Cover su G e VC è il risultato dell'algoritmo approssimato allora

$$\frac{|VC|}{|VC^*|} \leq 2$$

DIMOSTRAZIONE

$\forall e_1, e_2 \in A \quad e_1 \cap e_2 = \emptyset$ (non hanno vertici in comune).

$|VC^*| \geq |A|$ poiché dovranno coprire tutto il grafo, dovrà avere almeno un vertice per arco.

$\forall (x, y) \in A$ (sono tutti disgiunti) $x, y \in VC$ quindi banalmente $|VC| = 2|A|$

$$|VC^*| \geq |A| = \frac{|VC|}{2} \Rightarrow \frac{|VC|}{|VC^*|} \leq 2$$

ZAINO 0-1 2-APPROSSIMATO

Sia $I^{\alpha_1}(0, w)$ il nostro problema e $I^f(0, w)$ la sua versione frazionaria.

Definiamo $I_{j_1}^{\alpha_1}(0, w)$ e $I_{j_1}^f(0, w)$ varianti del problema in cui sia garantita la presenza dell'oggetto j_1 (che potrebbe non avere soluzione se il peso di j_1 è $> w$) **PER INTERO**.

PROPRIETA'

la soluzione ottima di $I^{\alpha_1}(0, w)$ è una soluzione ottima di un $I_{j_1}^{\alpha_1}(0, w)$

DI MOSTRAZIONE

Sia $S^* = j_1, j_2 \dots j_k$ soluzione ottima di $I^{\alpha_1}(0, w)$, lo sarà anche per $I_{j_1}^{\alpha_1}(0, w)$.

P.a. S^* non è ottima per $I_{j_1}^{\alpha_1}(0, w)$, esiste allora

$$S = j_1, j_2' \dots j_r' \quad | \quad |S'| < |S|$$

S rispetta ovviamente tutti i vincoli di S^* (più il vincolo su j_1) quindi è ottima anche per $I^{\alpha_1}(0, w)$. **ASSURDO!**

PROPRIETA'

Per risolvere $I_J^k(0, w)$ va prima verificato che $p_J \leq w$.

A questo punto

$$S(I_J^k(0, w)) = \underbrace{\{(j, p_j)\} \cup S(I_{J \setminus \{j\}}^k(w - p_j))}_{\text{j per intero}} \underbrace{\}_{\text{zaino frazionario classico } \Theta(n)}$$

Il costo totale sarebbe $n \cdot \Theta(n) = \Theta(n^2)$.

l'algoritmo greedy per lo zaino frazionario inserisce V_j tutto l'oggetto (in ordine di costo), solo l'ultimo oggetto sarà effettivamente frazionario (per riempire lo zaino).

Quindi se noi escludiamo la scelta frazionaria della soluzione di I_J^k avremo sicuramente una soluzione approssimata per $I_J^{0/1}$.

PROPRIETA'

Sia Q_j ottimo per I_J^k e \hat{Q}_j appross di $I_J^{0/1}$ e p_j ottimo per $I_J^{0/1}$

$$\frac{C(p_j)}{C(\hat{Q}_j)} \geq 2$$

DIMOSTRAZIONE

Anzitutto $C(Q_j) \geq C(P_j)$ poiché ogni soluzione intera è soluzione per la versione frazionaria, quindi se p.a. $C(Q_j) < C(P_j)$ avremmo una soluzione con costo migliore dell'ottimo (**ASSURDO**).

$$Q_j = (j, P_{j,1}, \dots, j_{k-1}, P_{k-1})(j_k, q_k) \quad \text{con } q_k \leq P_{j_k} \quad (\text{soltanto frazionaria})$$

$$\hat{Q}_j = (j, P_{j,1}, \dots, j_{k-1}, P_{k-1})$$

Sappiamo che $\text{val}(j) \geq \text{val}(j_{-1})$ (li scegliamo per valore)

$$\text{Vale di conseguenza } \text{val}(j) \geq q_k \cdot \frac{\text{val}(j_k)}{P_{j_k}} \quad (q_k \leq P_{j_k})$$

$$\text{Banalmente v'abb' } \text{val}(j) + \text{val}(j_1) + \dots + \text{val}(j_{k-1}) \geq q_k \cdot \frac{\text{val}(j_k)}{P_{j_k}}$$

Sommando ambo i membri dell'equazione il primo membro avremo

$$2(\text{val}(j) + \text{val}(j_1) + \dots + \text{val}(j_{k-1})) \geq (\text{val}(j) + \text{val}(j_1) + \dots + \text{val}(j_{k-1})) + q_k \cdot \frac{\text{val}(j_k)}{P_{j_k}}$$

$$2C(\hat{Q}_j) \geq C(Q_j) \geq C(P_j) \Rightarrow \frac{C(P_j)}{C(\hat{Q}_j)} \leq 2$$

Quindi abbiamo un algoritmo 2 approssimato per $I_j^{0/1}$

Ma se non sappiamo che l'ottimo per $I_j^{0/1}$ è l'ottimo di un

$I_{\bar{j}}^{0/1}$ abbiamo un algoritmo 2 approssimato per $I_{\bar{j}}^{0/1}$

ALGORITMO APROSSIMATO

L'algoritmo approssimato calcola tutti i Q_i ($O(n^2)$), troviamo i corrispettivi \hat{Q}_i (trasformazione consta $O(n)$ per n quindi $O(n^2)$).

Calcoliamo i costi dei \hat{Q}_i ($O(n^2)$) e ne prendiamo il massimo

OSS

Gli algoritmi approssimati possono essere semplificazioni di algoritmi greedy oppure approcci random (e ne sono tantissimi altri).