Algoritmi e Strutture Dati

Algoritmi greedy

Alberto Montresor

Università di Trento

2019/01/13

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

Sommario

Introduzione

Problemi di ottimizzazione

- Gli algoritmi per problemi di ottimizzazione eseguono una sequenza di decisioni

Programmazione dinamica

- In maniera bottom-up, valuta tutte le decisioni possibili
- Evitando però di ripetere sotto-problemi (decisioni) già percorse

Algoritmi greedy (ingordi, golosi)

- Seleziona una sola delle possibile decisioni...
- ... quella che sembra ottima (ovvero, è localmente ottima)
- E' però necessario dimostare che si ottiene un ottimo globale

Quando applicare la tecnica greedy?

Quando è possibile dimostrare che esiste una scelta ingorda

"*Fra le molte scelte possibili, ne può essere facilmente individuata una che porta sicuramente alla soluzione ottima.*"

Quando il problema ha sottostruttura ottima

"*Fatta tale scelta, resta un sottoproblema con la stessa struttura del problema principale.*"

Note

- Non tutti i problemi hanno una scelta ingorda
- In alcuni casi, soluzioni non ottime possono essere comunque interessanti

Insieme indipendente di intervalli

Input

Sia $S = \{1, 2, \ldots, n\}$ un insieme di intervalli della retta reale. Ogni intervallo $[a_i, b_i[$, con $i \in S$, è chiuso a sinistra e aperto a destra.

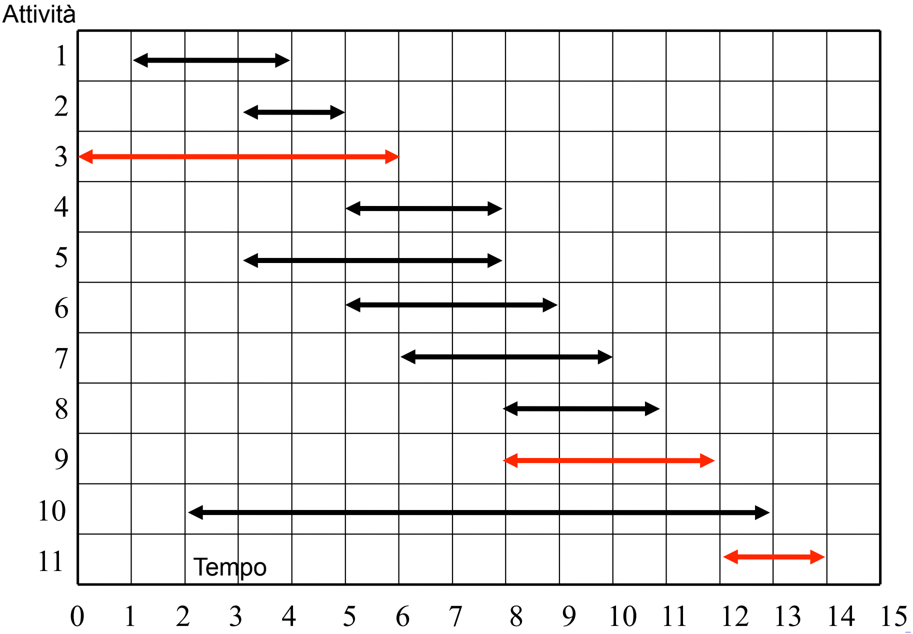
- a_i: tempo di inizio
- b_i: tempo di fine

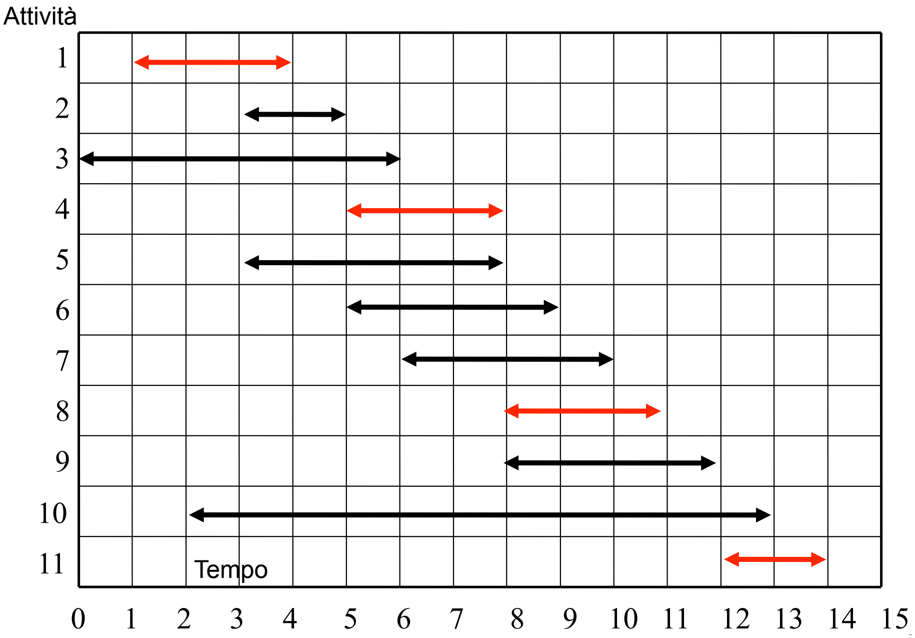
Definizione del problema

Trovare un insieme indipendente massimale è un sottoinsieme di massima cardinalità formato da intervalli tutti disgiunti tra loro.

i	a_i	b_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14

Confronta con "Insieme indipendente di intervalli pesati"


Attività
1
2
3
4
5
6
7
8
9
10
11
Tempo
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


Attività
1
2
3
4
5
6
7
8
9
10
11
Tempo
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Come affrontare il problema

Iniziamo con programmazione dinamica

- Individuiamo una sottostruttura ottima
- Scriviamo una definizione ricorsiva per la dimensione della soluzione ottima
- Scriviamo una versione iterativa bottom-up dell'algoritmo

Passiamo poi alla tecnica greedy

- Cerchiamo una possibile scelta ingorda
- Dimostriamo che la scelta ingorda porta alla soluzione ottima
- Scriviamo un algoritmo ricorsivo o iterativo che effettua sempre la scelta ingorda

Sottostruttura ottima

- Si assuma che gli intervalli siano ordinate per tempo di fine:

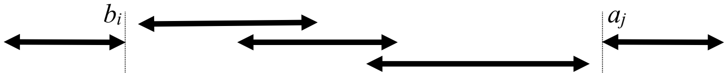
$$b_1 \leq b_2 \leq \ldots \leq b_n$$

- Definiamo il sottoproblema $S[i, j]$ come l'insieme di intervalli che iniziano dopo la fine di i e finiscono prima dell'inizio di j:

$$S[i, j] = \{k | b_i \leq a_k < b_k \leq a_j\}$$

- Aggiungiamo due intervalli fittizi:
    - Intervallo 0: $b_0 = -\infty$
    - Intervallo $n + 1$: $a_{n+1} = +\infty$

- Il problema iniziale corrisponde al problema $S[0, n + 1]$



Sottostruttura ottima

Teorema

Supponiamo che $A[i,j]$ sia una soluzione ottimale di $S[i,j]$ e sia k un intervallo che appartiene a $A[i,j]$; allora

- Il problema $S[i,j]$ viene suddiviso in due sottoproblemi
  - $S[i,k]$: gli intervalli di $S[i,j]$ che finiscono prima di k
  - $S[k,j]$: gli intervalli di $S[i,j]$ che iniziano dopo di k
- $A[i,j]$ contiene le soluzioni ottimali di $S[i,k]$ e $S[k,j]$
  - $A[i,j] \cap S[i,k]$ è la soluzione ottimale di $S[i,k]$
  - $A[i,j] \cap S[k,j]$ è la soluzione ottimale di $S[k,j]$

Dimostrazione

Utilizzando il metodo cut-and-paste

Definizione ricorsiva del costo della soluzione

Definizione ricorsiva della soluzione

$$A[i,j] = A[i,k] \cup \{k\} \cup A[k,j]$$

Definizione ricorsiva del suo costo

- Come determinare k? Analizzando tutte le possibilità
- Sia $DP[i,j]$ la dimensione del più grande sottoinsieme $A[i,j] \subseteq S[i,j]$ di intervalli indipendenti

$$DP[i,j] = \begin{cases} 0 & S[i,j] = \emptyset \\ \max_{k \in S[i,j]} \{DP[i,k] + DP[k,j] + 1\} & \text{altrimenti} \end{cases}$$

Verso una soluzione ingorda

Programmazione dinamica

- La definizione precedente ci permette di scrivere un algoritmo basato su programmazione dinamica o su memoization
- Complessità $O(n^3)$: bisogna risolvere tutti i problemi con $i < j$, con costo $O(n)$ per sottoproblema nel caso peggiore

Possiamo fare di meglio?

- Abbiamo visto una soluzione $O(n \log n)$ nel caso di intervalli pesati
- E' possibile utilizzare quella soluzione con pesi pari a 1
- Questa soluzione è peggiore, ma...
- Siamo sicuri che sia necessario analizzare tutti i possibili valori k?

Scelta ingorda (Greedy Choice)

Teorema

Sia $S[i,j]$ un sottoproblema non vuoto, e m l'intervallo di $S[i,j]$ con il minor tempo di fine, allora:

1. il sottoproblema $S[i,m]$ è vuoto
2. m è compreso in qualche soluzione ottima di $S[i,j]$

Dimostrazione 1

Sappiamo che:	$a_m < b_m$	(Definizione di intervallo)
Sappiamo che:	$\forall k \in S[i,j] : b_m \leq b_k$	(m ha minor tempo di fine)
Ne consegue che:	$\forall k \in S[i,j] : a_m < b_k$	(Transitività)

Se nessun intervallo in $S[i,j]$ termina prima di a_m, allora $S[i,m] = \emptyset$

Scelta ingorda (Greedy Choice)

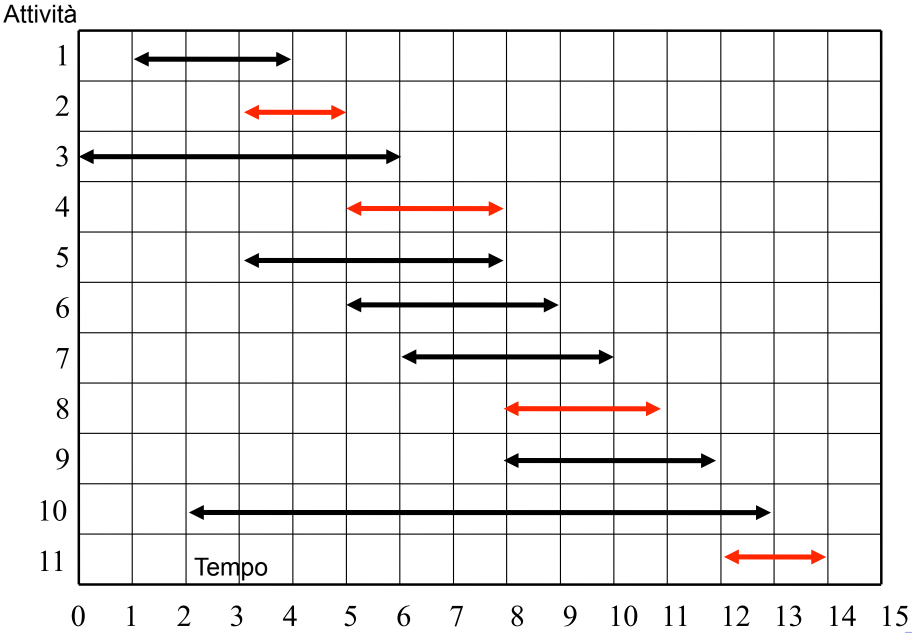
Teorema

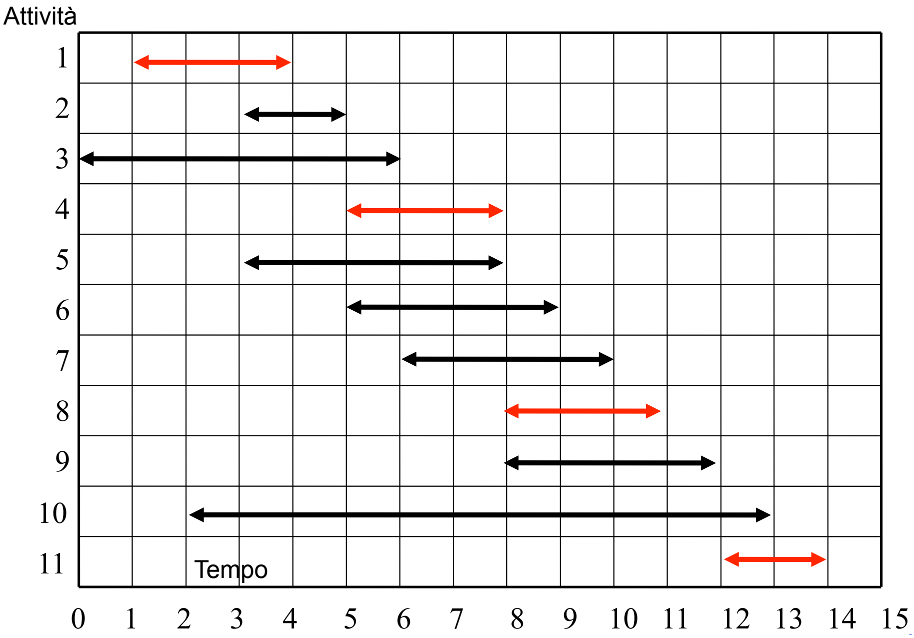
Sia $S[i, j]$ un sottoproblema non vuoto, e m l'intervallo di $S[i, j]$ con il minor tempo di fine, allora:

1. il sottoproblema $S[i, m]$ è vuoto
2. m è compreso in qualche soluzione ottima di $S[i, j]$

Dimostrazione 2

- Sia $A'[i, j]$ una soluzione ottima di $S[i, j]$
- Sia $m' \in A'[i, j]$ l'intervallo con minor tempo di fine in $A'[i, j]$
- Sia $A[i, j] = A'[i, j] - \{m'\} \cup \{m\}$ una nuova soluzione ottenuta togliendo m' e aggiungendo m ad $A'[i, j]$
- $A'[i, j]$ è una soluzione ottima che contiene m, in quanto ha la stessa dimensione di $A'[i, j]$ e tutti gli intervalli sono indipendenti.


Attività
1
2
3
4
5
6
7
8
9
10
11
Tempo
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


Attività
1
2
3
4
5
6
7
8
9
10
11
Tempo
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Conseguenze

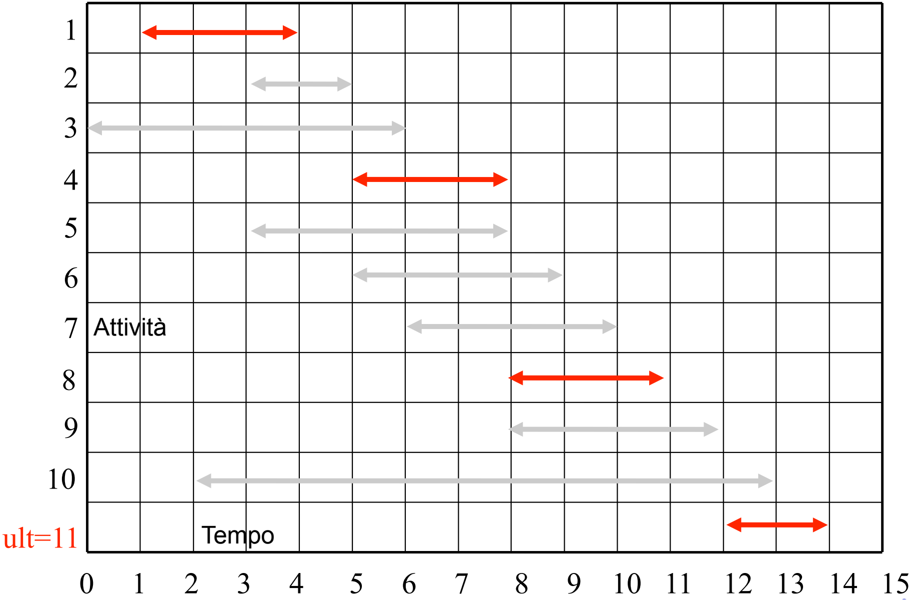
- Non è più necessario analizzare tutti i possibili valori di k
  - Faccio una scelta "ingorda", ma sicura: seleziono l'attività m con il minor tempo di fine
- Non è più necessario analizzare due sottoproblemi:
  - Elimino tutte le attività che non sono compatibili con la scelta ingorda
  - Mi resta solo un sottoproblema da risolvere: $S[m, j]$

Algoritmo

SET independentSet(**int**[ ] a, **int**[ ] b)

```
{ ordina a e b in modo che b[1] ≤ b[2] ≤ ··· ≤ b[n] }
SET S = Set()
S.insert(1)
int last = 1                          % Ultimo intervallo inserito
for i = 2 to n do
    if a[i] ≥ b[last] then            % Controllo indipendenza
        S.insert(i)
        last = i
return S
```

Complessità: $O(n \log n)$ se input non è ordinato
$O(n)$ se l'input è già ordinato.


1
2
3
4
5
6
7
Attività
8
9
10
ult=11
Tempo
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Approccio a partire da Programmazione Dinamica

- Abbiamo cercato di risolvere il problema della selezione delle attività tramite programmazione dinamica:
  - Abbiamo individuato una sottostruttura ottima
  - Abbiamo scritto una definizione ricorsiva per la dimensione della soluzione ottima
- Abbiamo dimostrato la proprietà della scelta greedy:
  - Per ogni sottoproblema, esiste almeno una soluzione ottima che contiene la scelta greedy
  - Abbiamo scritto un algoritmo iterativo che effettua sempre la scelta ingorda

Problema del resto

Input

- Un insieme di "tagli" di monete, memorizzati in un vettore di interi positivi $t[1 \ldots n]$.
- Un intero R rappresentante il resto che dobbiamo restituire.

Definizione del problema

Trovare il più piccolo numero intero di pezzi necessari per dare un resto di R centesimi utilizzando i tagli di cui sopra, assumendo di avere un numero illimitato di monete per ogni taglio.

Matematicamente: trovare un vettore x di interi non negativi tale che:

$R = \sum_{i=1}^{n} x[i] \cdot t[i]$ e $m = \sum_{i=1}^{n} x[i]$ ha valore minimo

Soluzione basata su programmazione dinamica

Sottostruttura ottima

- Sia $S(i)$ il problema di dare un resto pari ad i
- Sia $A(i)$ una soluzione ottima del problema $S(i)$, rappresentata da un multi-insieme; sia $j \in A(i)$
- Allora, $S(i - t[j])$ è un sottoproblema di $S(i)$, la cui soluzione ottima è data da $A(i) - \{j\}$.

Definizione ricorsiva

- Sia $DP[0 \ldots R]$ la tabella utilizzata per memorizzare le soluzioni
- $DP[i]$: minimo numero di monete per risolvere il problema $S[i]$

$$DP[i] = \begin{cases} 0 & i = 0 \\ \min_{1 \le j \le n} \{ DP[i - t[j]] \mid t[j] \le i \} + 1 & i > 0 \end{cases}$$

Algoritmo

```
resto(int[] t, int n, int R)
DP = new int[0 ... R]
S = new int[0 ... R]
DP[0] = 0
for i = 1 to R do
    DP[i] = +∞
    for j = 1 to n do
        if i > t[j] and DP[i − t[j]] + 1 < DP[i]
         then
            DP[i] = DP[i − t[j]] + 1
            S[i] = j
while R > 0 do
    print t[S[R]]
    R = R − t[S[R]]
```

Complessità

$O(nR)$

Scelta greedy

Domanda

E' possibile pensare ad una soluzione greedy?

Risposta

Selezionare la moneta j più grande tale per cui $t[j] \leq R$, e poi risolvere il problema $S(R - t[j])$.

Esempi

- Tagli: $200, 100, 50, 20, 10, 5, 2, 1$
- Tagli: $50, 10, 5, 1$
- Tagli: $10, 8, 1$
- Tagli: $c^k, c^{k-1}, \ldots, c, 1 \quad (c \in \mathbb{Z}^+)$

Algoritmo

```
resto(int[] t, int n, int R, int[] x)
{ Ordina le monete in modo decrescente }
for i = 1 to n do
    x[i] = ⌊R/t[i]⌋
    R = R − x[i] · t[i]
```

Complessità: $O(n \log n)$ se input non è ordinato
$O(n)$ se l'input è già ordinato.

Dimostrazione scelta greedy $50, 10, 5, 1$

- Sia x una qualunque soluzione ottima; quindi
$$\sum_{i=1}^{4} x[i] \cdot t[i] = R \qquad m = \sum_{i=1}^{4} x[i] \quad \text{è minimo}$$
- Sia m_k la somma delle monete di taglio inferiore a $t[k]$:
$$m_k = \sum_{i=k+1}^{4} x[i] \cdot t[i]$$
- Se dimostriamo che $\forall k : m_k < t[k]$, allora la soluzione (ottima) è proprio quella calcolata dall'algoritmo

$$\begin{aligned}
m_4 &= 0 && < 1 = t[4] \\
m_3 &= 1 \cdot x[4] && < 5 = t[3] \\
m_2 &= 5 \cdot x[3] + m_3 && \leq 5 + m_3 < 5 + 5 = 10 = t[2] \\
m_1 &= 10 \cdot x[2] + m_2 && \leq 40 + m_2 < 40 + 10 = 50 = t[1]
\end{aligned}$$

Approccio greedy, senza programmazione dinamica

- Evidenziare i "passi di decisione"
  - Trasformare il problema di ottimizzazione in un problema di "scelte" successive
- Evidenziare una possibile scelta ingorda
  - Dimostrare che tale scelta rispetto il "principio della scelta ingorda"
- Evidenziare la sottostruttura ottima
  - Dimostrare che la soluzione ottima del problema "residuo" dopo la scelta ingorda può essere unito a tale scelta
- Scrittura codice: top-down, anche in maniera iterativa
  - Nota: può essere necessario pre-processare l'input

Scheduling

Input

Supponiamo di avere un processore e n job da eseguire su di esso, ognuno caratterizzato da un tempo di esecuzione $t[i]$ noto a priori.

Problema

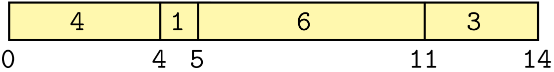
Trovare una sequenza di esecuzione (permutazione) che minimizzi il tempo di completamento medio.

Dato un vettore $A[1 \ldots n]$ contenente una permutazione di $\{1, \ldots n\}$, il tempo di completamento dell'h-esimo job nella permutazione è:

$$T_A(h) = \sum_{i=1}^{h} t[A[i]]$$

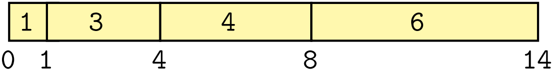
Esempio

Esempio



Tempo di completamento medio: $(4 + 5 + 11 + 14)/4 = 34/4 = 8.5$

Shortest job first



Tempo di completamento medio: $(1 + 4 + 8 + 14)/4 = 27/4 = 6.75$

Dimostrazione di correttezza

Teorema - Scelta greedy

Esiste una soluzione ottima A in cui il job con minor tempo di fine m si trova in prima posizione ($A[1] = m$).

Teorema – Sottostruttura ottima

Sia A una soluzione ottima di un problema con n job, in cui il job con minor tempo di fine m si trova in prima posizione. La permutazione dei seguenti $n - 1$ job in A è una soluzione ottima al sottoproblema in cui il job m non viene considerato.

Dimostrazione di correttezza

Dimostrazione - Scelta greedy

- Si consideri una permutazione ottima $B = [\ B[1], B[2], \ldots, B[n]\ ]$
- Sia k la posizione in cui si trova in B il job con minor tempo di fine
- Si consideri una permutazione in cui i job in posizione 1, k vengono scambiati: $A = [\ B[k], B[2], \ldots, B[k-1], B[1], B[k+1], \ldots, B[n]\ ]$
- Il tempo di completamento medio di A è minore o uguale al tempo di completamento medio di B
  - Job in posizione $1, \ldots, k-1$ in A hanno tempo di completamento $\leq$ dei job in posizione $1, \ldots, k-1$ in B
  - Job in posizione $k, \ldots, n$ in A hanno tempo di completamento $\leq$ dei job in posizione $k, \ldots, n$ in B
- Poichè B è ottima, A non può avere tempo di completamento medio minore e quindi anche A è ottima.

Problema dello zaino

Input

- Un intero positivo C - la capacità dello zaino
- n oggetti, tali che l'oggetto i-esimo è caratterizzato da
  - un profitto $p_i \in \mathbb{Z}^+$
  - un peso $w_i \in \mathbb{Z}^+$

Zaino 0/1

Trovare un sottoinsieme S di $\{1, \ldots, n\}$ di oggetti tale che il loro peso totale non superi la capacità massima e il loro profitto totale sia massimo.

Zaino reale (o Zaino frazionario)

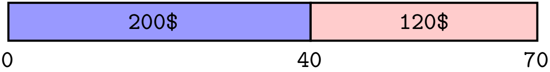
E' possibile prendere frazioni di oggetti.

Esempio

Consideriamo i tre oggetti a lato ed una capacità di 70

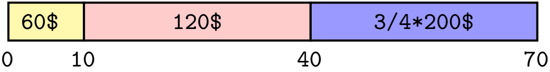
i	p_i	w_i
1	60$	10
2	200$	40
3	120$	30

Approccio 1: Ordinati per profitto decrescente



200$ +120$ = 320$

Approccio 2: Ordinati per peso crescente



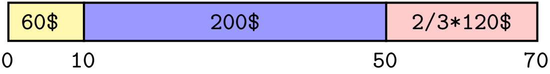
60$ + 120$ + 150$ = 330$

Esempio

Consideriamo i tre oggetti a lato ed una capacità di 70

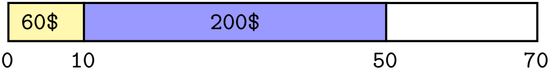
i	p_i	w_i	p_i/w_i
1	60$	10	6$
2	200$	40	5$
3	120$	30	4$

Approccio 3: Ordinati per profitto specifico p_i/w_i decrescente



60$ +200$ + 80$ = 340$

Approccio 3 non funziona per Zaino 0/1



60$ +200$ = 260$

Algoritmo

$\text{zaino}(\textbf{float}[\,]\ p,\ \textbf{float}[\,]\ v, \textbf{float}\ C,\ \textbf{int}\ n,\ \textbf{float}[\,]\ x)$

{ ordina p e v in modo che $p[1]/w[1] \geq p[2]/w[2] \geq \cdots \geq p[n]/w[n]$}
for $i = 1$ **to** n **do**
  $x[i] = \mathsf{min}(C/w[i], 1)$
  $C = C - x[i] \cdot w[i]$

Complessità: $O(n \log n)$ se input non è ordinato
$O(n)$ se l'input è già ordinato.

$x[i] \in [0, 1]$ rappresenta la proporzione dell'oggetto i-esimo che deve essere prelevata.

Correttezza

Informalmente

- Assumiamo che gli oggetti siano ordinati per profitto specifico decrescente
- Sia x una soluzione ottima
- Supponiamo che $x[1] < \mathsf{min}(C/w[i], 1) < 1$
- Allora possiamo costruire una nuova soluzione in cui $x'[1] = \mathsf{min}(C/w[i], 1)$ e la proporzione di uno o più oggetti è ridotta di conseguenza
- Otteniamo così una soluzione x' di profitto uguale o superiore, visto che il profitto specifico dell'oggetto 1 è massimo

Problema della compressione

Rappresentare i dati in modo efficiente

- Impiegare il numero minore di bit per la rappresentazione
- Obiettivo: risparmio spazio su disco e tempo di trasferimento

Una possibile tecnica di compressione: codifica di caratteri

- Tramite funzione di codifica $f : f(c) = x$
  - c è un possibile carattere preso da un alfabeto Σ
  - x è una rappresentazione binaria
  - "c è rappresentato da x"

Possibili codifiche

Esempio

- Supponiamo di avere un file di n caratteri
- Composto da caratteri nell'alfabeto `abcdef`
- Di cui conosciamo la frequenza relativa

Caratteri	`a`	`b`	`c`	`d`	`e`	`f`	**Dim.**
Frequenza	45%	13%	12%	16%	9%	5%	
ASCII	`01100001`	`01100010`	`01100011`	`01100100`	`01100101`	`01100110`	$8n$
Codifica 1	`000`	`001`	`010`	`011`	`100`	`101`	$3n$

Possiamo fare di meglio?

Possibili codifiche

Esempio

- Supponiamo di avere un file di n caratteri
- Composto da caratteri nell'alfabeto `abcdef`
- Di cui conosciamo la frequenza relativa

Caratteri	`a`	`b`	`c`	`d`	`e`	`f`	**Dim.**
Frequenza	45%	13%	12%	16%	9%	5%	
ASCII	`01100001`	`01100010`	`01100011`	`01100100`	`01100101`	`01100110`	$8n$
Codifica 1	`000`	`001`	`010`	`011`	`100`	`101`	$3n$
Codifica 2	`0`	`100`	`101`	`111`	`1100`	`1101`	$2.24n$

Costo totale: $(0.45{\cdot}1+0.13{\cdot}3+0.12{\cdot}3+0.16{\cdot}3+0.09{\cdot}4+0.05{\cdot}4){\cdot}n = 2.24n$

Codifica a prefissi

Codice a prefisso

In un codice a prefisso (meglio sarebbe "senza prefissi"), nessun codice è prefisso di un altro codice (condizione necessaria per la decodifica).

Esempio 1

- `"babaca"`: $\texttt{100} \cdot \texttt{0} \cdot \texttt{100} \cdot \texttt{0} \cdot \texttt{101} \cdot \texttt{0}$

Esempio 2

- Codice: `"a"` $\rightarrow$ `0`, `"b"` $\rightarrow$ `1`, `"c"` $\rightarrow$ `11`
- `111111`?

Rappresentazione ad albero per la codifica

Alcune domande

- E' possibile che il testo codificato sia più lungo della rappresentazione con 3 bit?
- Esistono testi "difficili" per una rappresentazione di questo tipo?
- Come organizzare un algoritmo per la decodifica?

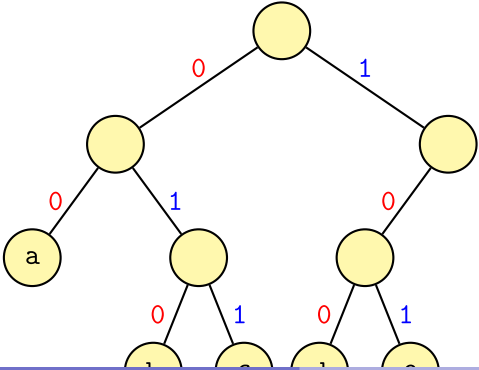
Cenni storici

- David Huffman, 1952
- Algoritmo ottimo per costruire codici prefissi
- Oggi utilizzato come complemento di altri metodi di compressione (Ad esempio in PKZIP, ZIP, WINRAR)

Rappresentazione ad albero per la decodifica

Alberi binari di decodifica

- Figlio sinistro/destro: 0 / 1
- Caratteri dell'alfabeto sulle foglie



a	b	c	d	e
00	010	011	100	101

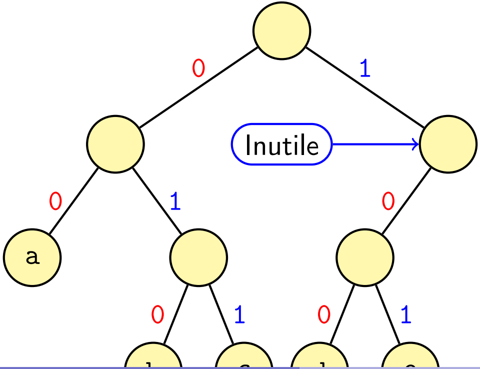
Algoritmo di decodifica

parti dalla radice
while file non è finito **do**
    leggi un bit
    if bit è zero **then**
        vai a sinistra
    else
        vai a destra
    if nodo foglia **then**
        stampa il carattere
        torna alla radice

Rappresentazione ad albero per la decodifica

Alberi binari di decodifica

- Figlio sinistro/destro: 0 / 1
- Caratteri dell'alfabeto sulle foglie



a	b	c	d	e
00	010	011	100	101

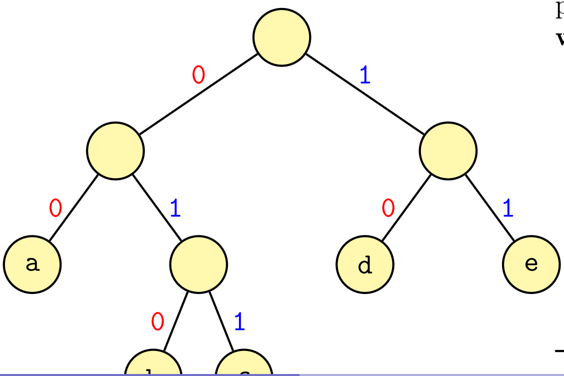
Algoritmo di decodifica

parti dalla radice
while file non è finito **do**
  leggi un bit
  if bit è zero **then**
    vai a sinistra
  else
    vai a destra
  if nodo foglia **then**
    stampa il carattere
    torna alla radice

Rappresentazione ad albero per la decodifica

Alberi binari di decodifica

- Figlio sinistro/destro: 0 / 1
- Caratteri dell'alfabeto sulle foglie



a	b	c	d	e
00	010	011	10	11

Algoritmo di decodifica

parti dalla radice
while file non è finito **do**
  leggi un bit
  if bit è zero **then**
    vai a sinistra
  else
    vai a destra
  if nodo foglia **then**
    stampa il carattere
    torna alla radice

Definizione formale del problema

Input

- un file F composto da caratteri nell'alfabeto Σ

Quanti bit sono richiesti per codificare il file?

- Sia T un albero che rappresenta la codifica
- Per ogni $c \in \Sigma$, sia $d_T(c)$ la profondità della foglia che rappresenta c
- Il codice per c richiederà allora $d_T(c)$ bit
- Se $f[c]$ è il numero di occorrenze di c in F, allora la dimensione della codifica è

$$C[F, T] = \sum_{c \in \Sigma} f[c] \cdot d_T(c)$$

Algoritmo di Huffman

Principio del codice di Huffman

- Minimizzare la lunghezza dei caratteri che compaiono più frequentemente
- Assegnare ai caratteri con la frequenza minore i codici corrispondenti ai percorsi più lunghi all'interno dell'albero

Un codice è progettato per un file specifico

- Si ottiene la frequenza di tutti i caratteri
- Si costruisce il codice
- Si rappresenta il file tramite il codice
- Si aggiunge al file una rappresentazione del codice, per la decodifica

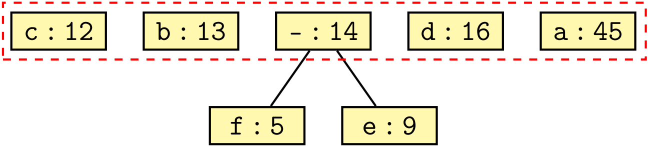
Funzionamento algoritmo

- Costruire una lista ordinata di nodi foglia per ogni carattere, etichettato con la propria frequenza

f : 5	e : 9	c : 12	b : 13	d : 16	a : 45

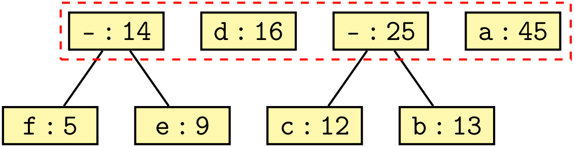
Funzionamento algoritmo

- Rimuovere i due nodi con frequenze minori f_x, f_y
- Creare un nodo padre con etichetta "-" e frequenza $f_x + f_y$
- Collegare i due nodi rimossi con il nuovo nodo
- Aggiungere il nodo così creato alla lista, mantenendo l'ordine



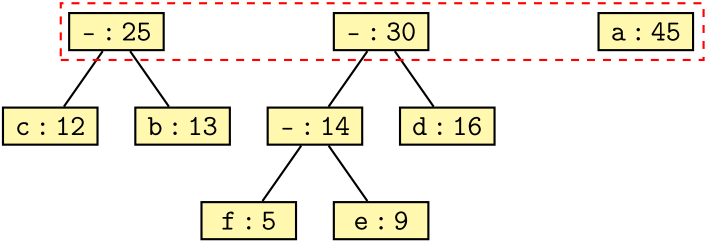
Funzionamento algoritmo

- Rimuovere i due nodi con frequenze minori f_x, f_y
- Creare un nodo padre con etichetta "-" e frequenza $f_x + f_y$
- Collegare i due nodi rimossi con il nuovo nodo
- Aggiungere il nodo così creato alla lista, mantenendo l'ordine



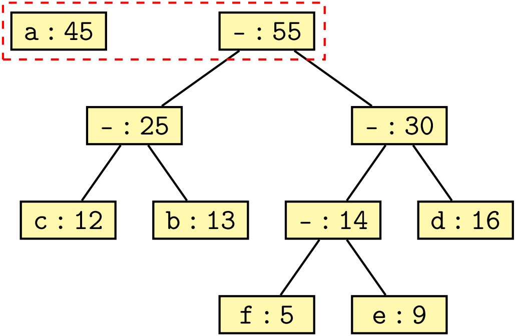
Funzionamento algoritmo

- Rimuovere i due nodi con frequenze minori f_x, f_y
- Creare un nodo padre con etichetta "-" e frequenza $f_x + f_y$
- Collegare i due nodi rimossi con il nuovo nodo
- Aggiungere il nodo così creato alla lista, mantenendo l'ordine



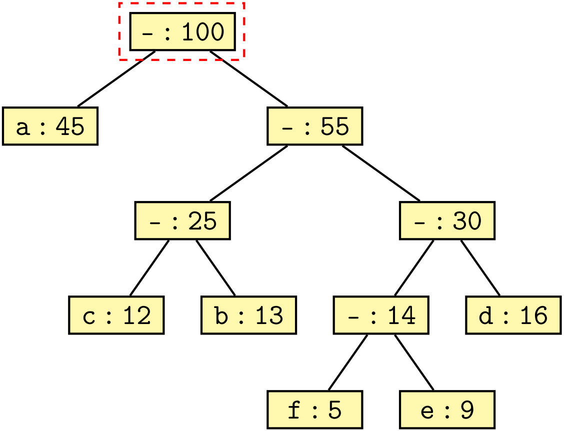
Funzionamento algoritmo

- Rimuovere i due nodi con frequenze minori f_x, f_y
- Creare un nodo padre con etichetta "-" e frequenza $f_x + f_y$
- Collegare i due nodi rimossi con il nuovo nodo
- Aggiungere il nodo così creato alla lista, mantenendo l'ordine



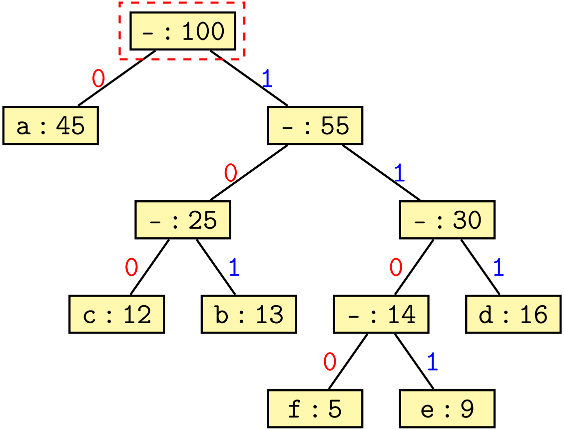
Funzionamento algoritmo

- Si termina quando resta un solo nodo nella lista



Funzionamento algoritmo

- Al termine, si etichettano gli archi dell'albero con bit `0,1`



a	0
b	100
c	101
d	111
e	1100
f	1101

Algoritmo

```
TREE huffman(int[] c, int[] f, int n)
PRIORITYQUEUE Q = MinPriorityQueue()
for i = 1 to n do
    Q.insert(f[i], Tree(f[i], c[i]))
for i = 1 to n − 1 do
    z1 = Q.deleteMin()
    z2 = Q.deleteMin()
    z = Tree(z1.f + z2.f, nil)
    z.left = z1
    z.right = z2
    Q.insert(z.f, z)
return Q.deleteMin()
```

Input

- n: dimensione dell'alfabeto
- $c[1 \ldots n]$: caratteri alfabeto
- $f[1 \ldots n]$: frequenze

```
TREE
c        % Carattere
f        % Frequenza
left     % Figlio sinistro
right    % Figlio destro
```

Complessità: $O(n \log n)$

Correttezza

Teorema

L'output dell'algoritmo Huffman per un dato file è un codice a prefisso ottimo

Schema della dimostrazione

- Proprietà della scelta greedy
  - Scegliere i due elementi con la frequenza più bassa conduce sempre ad una soluzione ottimale
- Sottostruttura ottima
  - Dato un problema sull'alfabeto Σ, è possibile costruire un sottoproblema con un alfabeto più piccolo

Scelta greedy

Ipotesi

- Σ un alfabeto
- f un vettore di frequenze
- x, y i due caratteri che hanno frequenza più bassa

Tesi

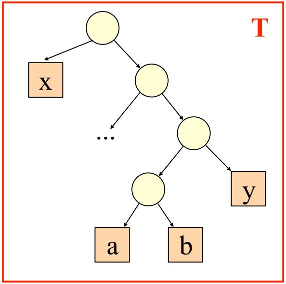
- Esiste un codice prefisso ottimo per Σ in cui x, y hanno la stessa profondità massima e i loro codici differiscono solo per l'ultimo bit (sono foglie sorelle)

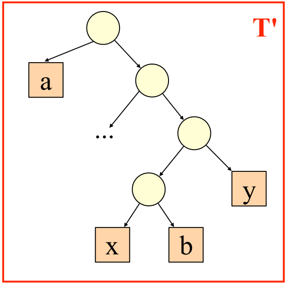
Dimostrazione

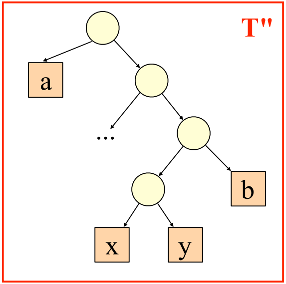
- Al solito, basata sulla trasformazione di una soluzione ottima
- Supponiamo che esista un codice ottimo T in cui due caratteri a, b con profondità massima e questi siano diversi da x, y

Scelta greedy

- Assumiamo senza perdere di generalità: $f[x] \leq f[y]$, $f[a] \leq f[b]$
- Poiché le frequenze di x e y sono minime: $f[x] \leq f[a]$, $f[y] \leq f[b]$
- Scambiamo x con a: otteniamo T'
- Scambiamo y con b: otteniamo T"







Scelta greedy

- Dimostriamo che: $C(f, T") \leq C(f, T') \leq C(f, T)$

$$\begin{array}{rcl} C(T) - C(T') & = & \sum_{c \in \Sigma} f[c] d_T(c) - \sum_{c \in \Sigma} f[c] d_{T'}(c) \\ & = & (f[x]d_T(x) + f[a]d_T(a)) - (f[x]d_{T'}(x) + f[a]d_{T'}(a)) \\ & = & (f[x]d_T(x) + f[a]d_T(a)) - (f[x]d_T(a) + f[a]d_T(x)) \\ & = & (f[a] - f[x])(d_T(a) - d_T(x)) \\ & \geq & 0 \\ C(T') - C(T'') & \geq & 0 \qquad \text{Come sopra} \end{array}$$

- Ma poiché T è ottimo, sappiamo anche che: $C(f, T) \leq C(f, T")$
- Quindi $T"$ è anch'esso ottimo

Albero di copertura di peso minimo

Problema

Dato un grafo pesato, determinare come interconnettere tutti i suoi nodi minimizzando il costo del peso associato ai suoi archi.

- Albero di copertura (di peso) minimo
- Albero di connessione (di peso) minimo
- Minimum spanning tree

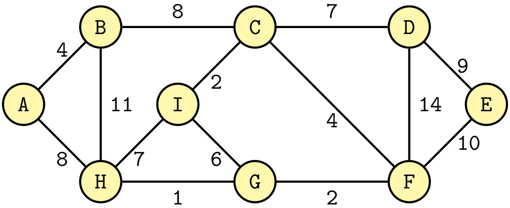
Esempio di applicazione

Una compagnia di telecomunicazioni deve stendere una nuova rete in un quartiere; deve seguire le connessioni esistenti (la rete stradale) e ogni arco ha un costo associato distinto (costi di scavo, etc.)

Definizione del problema

Input

- $G = (V, E)$: un grafo non orientato e connesso
- $w : V \times V \to \mathbb{R}$: una funzione di peso (costo di connessione)
  - se $[u, v] \in E$, allora $w(u, v)$ è il peso dell'arco $[u, v]$
  - se $[u, v] \notin E$, allora $w(u, v) = +\infty$
- Poiché G non è orientato, $w(u, v) = w(v, u)$

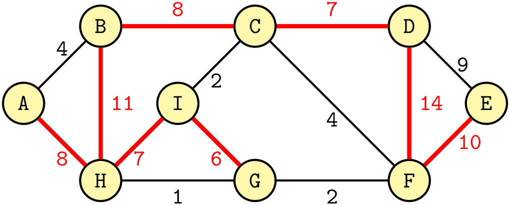


Definizione del problema

Albero di copertura (Spanning tree)

Dato un grafo $G = (V, E)$ non orientato e connesso, un albero di copertura di G è un sottografo $T = (V, E_T)$ tale che

- T è un albero
- $E_T \subseteq E$
- T contiene tutti i vertici di G



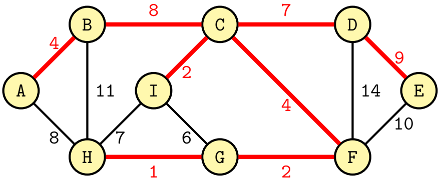
Definizione del problema

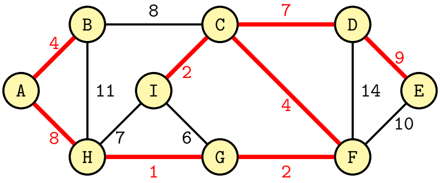
Output: albero di copertura di peso minimo

Trovare l'albero di copertura il cui peso totale sia minimo rispetto a ogni altro albero di copertura.

$$w(T) = \sum_{[u,v] \in E_T} w(u, v)$$

Non è detto che l'albero di copertura minimo sia univoco





Algoritmo generico

Schema della lezione

- Progettiamo un algoritmo di tipo "goloso" generico
- Mostriamo due "istanze" di questo algoritmo: Kruskal e Prim

Approccio

L'idea è di accrescere un sottoinsieme A di archi in modo tale che venga sempre rispettata la seguente invariante:

- A è un sottoinsieme di qualche albero di connessione minimo

Algoritmo generico

Arco sicuro

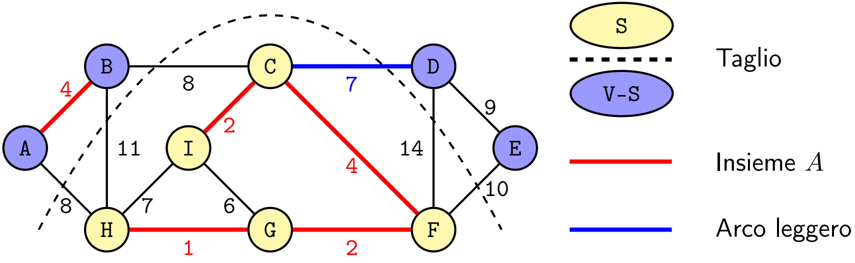
Un arco $[u, v]$ è detto sicuro per A se $A \cup \{[u, v]\}$ è ancora un sottoinsieme di qualche albero di connessione minimo.

SET mst-generico(GRAPH G, **int**[ ] w)

SET $A = \emptyset$
while A non forma un albero di copertura **do**
    trova un arco sicuro $[u, v]$
    $A = A \cup \{[u, v]\}$
return A

Definizioni

- Un taglio $(S, V - S)$ di un grafo non orientato $G = (V, E)$ è una partizione di V in due sottoinsiemi disgiunti
- Un arco $[u, v]$ attraversa il taglio se $u \in S$ e $v \in V - S$
- Un taglio rispetta un insieme di archi A se nessun arco di A attraversa il taglio
- Un arco che attraversa un taglio è leggero nel taglio se il suo peso è minimo fra i pesi degli archi che attraversano un taglio



Arco sicuro

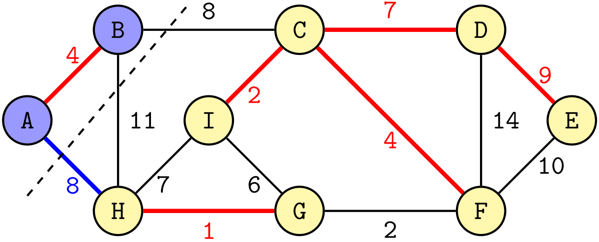
Teorema

- Sia $G = (V, E)$ un grafo non orientato e connesso
- Sia $w : V \times V \to \mathbb{R}$
- Sia $A \subseteq E$ un sottoinsieme contenuto in un qualche albero di copertura minimo per G
- Sia $(S, V - S)$ un qualunque taglio che rispetta A
- Sia $[u, v]$ un arco leggero che attraversa il taglio

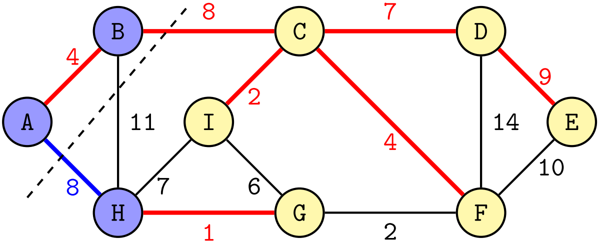
Allora l'arco $[u, v]$ è sicuro per A

Esempio: arco non sicuro perché il taglio non rispetta A

Arco blu sicuro

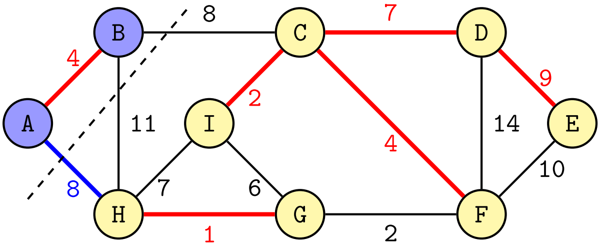


Arco blu non sicuro

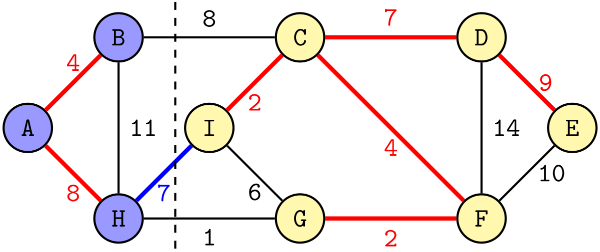


Esempio: arco non sicuro perché non leggero

Arco blu sicuro



Arco blu non sicuro

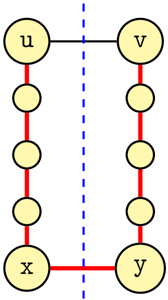


Dimostrazione

Sia T un albero di copertura minimo che contiene A. Due casi:

- $(u, v) \in T$: allora (u, v) è sicuro per A
- $(u, v) \notin T$: trasformiamo T in un albero T' contenente (u, v) e dimostriamo che T' è un albero di copertura minimo

- u, v sono connessi da un cammino $C \subseteq T$ (per definizione di albero)
- u, v stanno in lati opposti del taglio ((u, v) attraversa il taglio)
- $\exists (x, y) \in C$ che attraversa il taglio

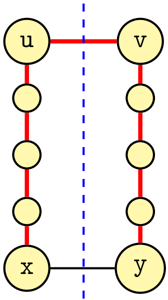


Dimostrazione

Sia T un albero di copertura minimo che contiene A. Due casi

- $(u, v) \in T$: allora (u, v) è sicuro per A
- $(u, v) \notin T$: trasformiamo T in un albero T' contenente (u, v) e dimostriamo che T' è un albero di copertura minimo

- $T' = T - \{(x, y)\} \cup \{(u, v)\}$
- T' è un albero di copertura
- $w(T') \leq w(T)$ (perchè $w(u, v) \leq w(x, y)$)
- $w(T) \leq w(T')$ (perchè T minimo)



Archi sicuri

Corollario

- Sia $G = (V, E)$ un grafo non orientato e connesso
- Sia $w : V \times V \to \mathbb{R}$
- Sia $A \subseteq E$ un sottoinsieme contenuto in un qualche albero di copertura minimo per G
- Sia C una componente connessa (un albero) nella foresta $G_A = (V, A)$
- Sia $[u, v]$ un arco leggero che connette C a qualche altra componente in G_A

Allora l'arco $[u, v]$ è sicuro per A

Algoritmo di Kruskal

Idea

- Ingrandire sottoinsiemi disgiunti di un albero di copertura minimo connettendoli fra di loro fino ad avere l'albero complessivo
- Si individua un arco sicuro scegliendo un arco $[u, v]$ di peso minimo tra tutti gli archi che connettono due distinti alberi (componenti connesse) della foresta
- L'algoritmo è greedy perché ad ogni passo si aggiunge alla foresta un arco con il peso minore

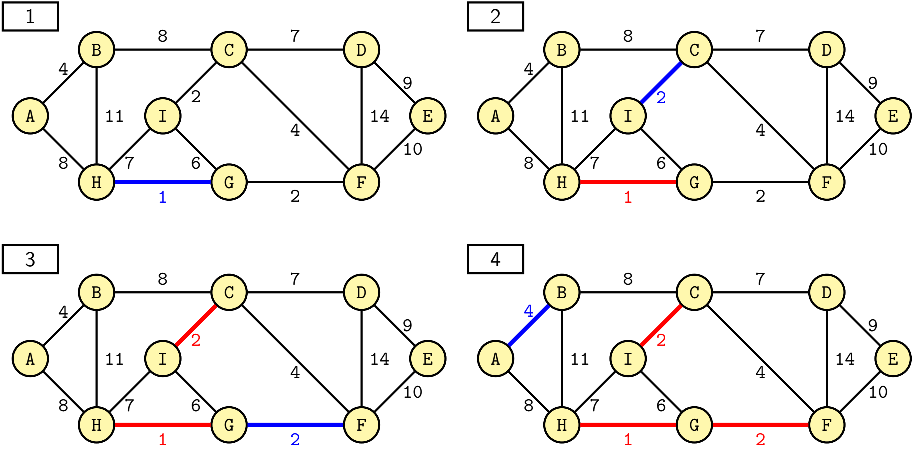
Implementazione

- Si utilizza una struttura dati Merge-Find Set

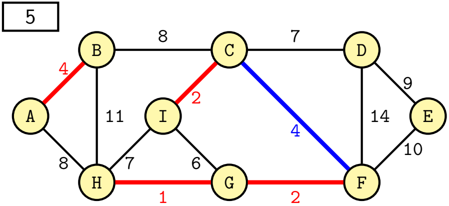
Algoritmo di Kruskal

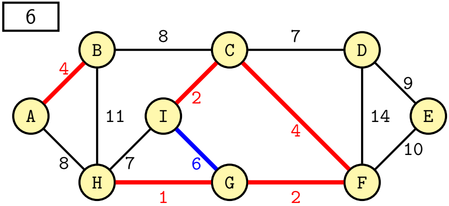
```
SET kruskal(EDGE[] A, int n, int m)
SET T = Set()
MFSET M = Mfset(n)
{ ordina A[1,...,m] in modo che A[1].peso ≤ ··· ≤ A[m].peso }
int c = 0
int i = 1
while c < n − 1 and i ≤ m do        % Termina quando l'albero è costruito
    if M.find(A[i].u) ≠ M.find(A[i].v) then
        M.merge(A[i].u, A[i].v)
        T.insert(A[i])
        c = c + 1
    i = i + 1
return T
```

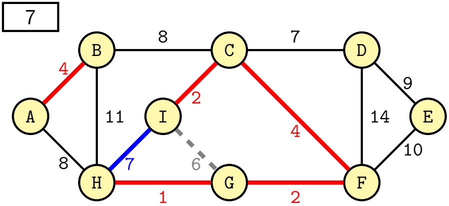
Esempio

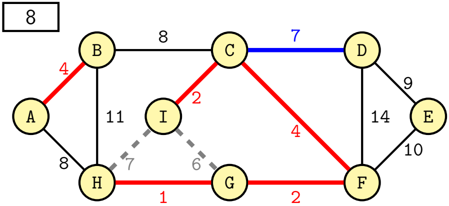

1
A B C D E F G H I
4 8 7 9 11 2 14 4 10 8 7 6 1 2
2
A B C D E F G H I
4 8 7 9 11 2 14 4 10 8 7 6 1 2
3
A B C D E F G H I
4 8 7 9 11 2 14 4 10 8 7 6 1 2
4
A B C D E F G H I
4 8 7 9 11 2 14 4 10 8 7 6 1 2

Esempio

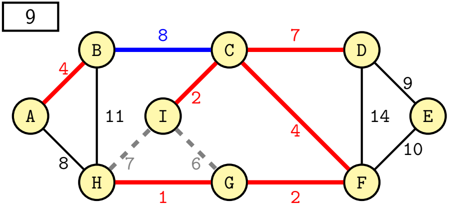

5
A B C D E F G H I
4 8 7 9 11 2 4 14 10 8 7 6 1 2

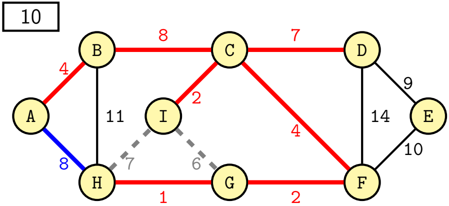

6
A B C D E F G H I
4 8 7 9 11 2 4 14 10 8 7 6 1 2

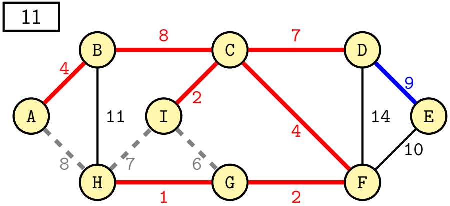

7
A B C D E F G H I
4 8 7 9 11 2 4 14 10 8 7 6 1 2

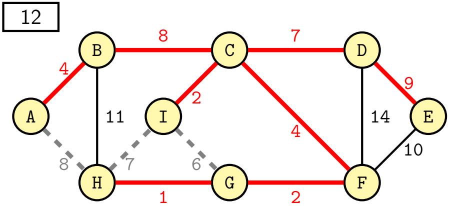

8
A B C D E F G H I
4 8 7 9 11 2 4 14 10 8 7 6 1 2

Esempio


9
A
B
C
D
E
F
G
H
I
4
8
7
11
2
14
9
4
10
8
7
6
1
2


10
A
B
C
D
E
F
G
H
I
4
8
7
11
2
14
9
4
10
8
7
6
1
2


11
A
B
C
D
E
F
G
H
I
4
8
7
11
2
14
9
4
10
8
7
6
1
2


12
A
B
C
D
E
F
G
H
I
4
8
7
11
2
14
9
4
10
8
7
6
1
2

Analisi della complessità algoritmo di Kruskal

Analisi della complessità

- Il tempo di esecuzione per l'algoritmo di Kruskal dipende dalla realizzazione della struttura dati per Merge-Find Set
- Utilizziamo la versione con euristica sul rango + compressione
  - L'inizializzazione richiede $O(n)$
  - L'ordinamento richiede $O(m \log m) = O(m \log n^2) = O(m \log n)$
  - Vengono eseguite $O(m)$ operazioni sulla foresta di insiemi disgiunti, con tempo ammortizzato $O(1)$
- Totale: $O(n + m \log n + m) = O(m \log n)$

Algoritmo di Prim

Idea

- L'algoritmo di Prim procede mantenendo in A un singolo albero
- L'albero parte da un vertice arbitrario r (la radice) e cresce fino a quando non ricopre tutti i vertici
- Ad ogni passo viene aggiunto un arco leggero che collega un vertice in V_A con un vertice in $V - V_A$, dove V_A è l'insieme di nodi raggiunti da archi in A

Correttezza

- $(V_A, V - V_A)$ è un taglio che rispetta A (per definizione)
- Per il corollario, gli archi leggeri che attraversano il taglio sono sicuri

Implementazione

Una struttura dati per i nodi non ancora nell'albero

- Durante l'esecuzione, i vertici non ancora nell'albero si trovano in una coda con min-priorità Q ordinata in base alla seguente definizione di priorità
- "La priorità del nodo v è il peso minimo di un arco che collega v ad un vertice nell'albero, o $+\infty$ se tale arco non esiste""

Albero registrato come vettore dei padri

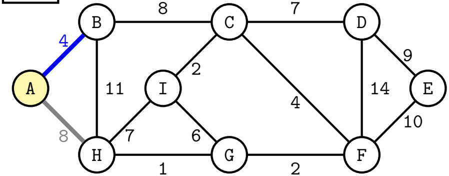
- Ogni nodo v mantiene un puntatore al padre $p[v]$
- A è mantenuto implicitamente: $A = \{[v, p[v]] \mid v \in V - Q - \{r\}\}$

Algoritmo di Prim

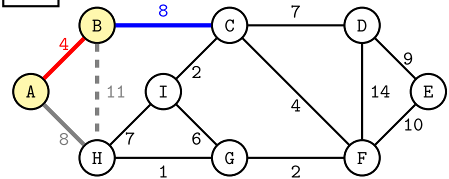
```
prim(GRAPH G, NODE r, int[] p)
PRIORITYQUEUE Q = MinPriorityQueue()
PRIORITYITEM[] pos = new PRIORITYITEM[1 ... G.n]
foreach u ∈ G.V() − {r} do
    pos[u] = Q.insert(u, +∞)
pos[r] = Q.insert(r, 0)
p[r] = 0
while not Q.isEmpty() do
    NODE u = Q.deleteMin()
    pos[u] = nil
    foreach v ∈ G.adj(u) do
        if pos[v] ≠ nil and w(u, v) < pos[v].priority then
            Q.decrease(pos[v], w(u, v))
            p[v] = u
```

Esempio

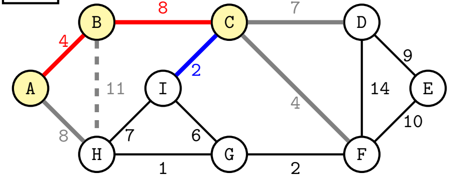
1



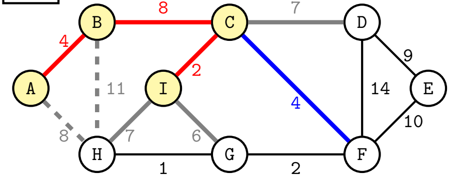
2



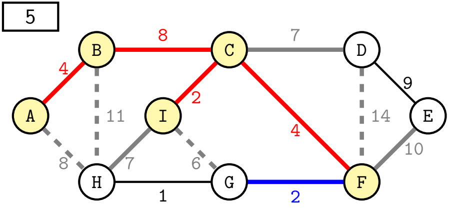
3

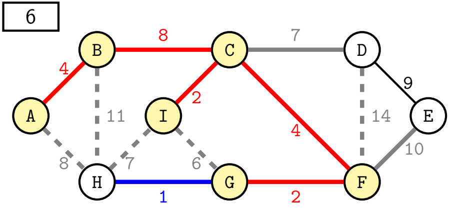


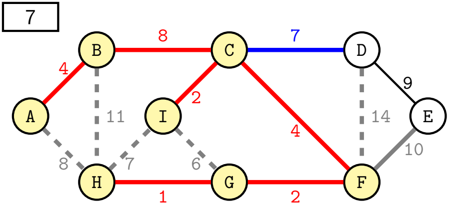
4

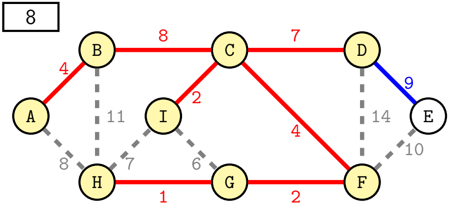


Esempio

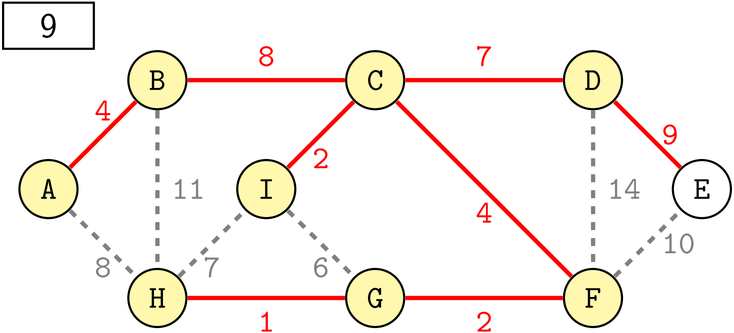

5
A
B
C
D
E
F
G
H
I
4
8
7
9
11
2
14
4
10
8
7
6
1
2


6
A
B
C
D
E
F
G
H
I
4
8
7
9
11
2
14
4
10
8
7
6
1
2


7
A
B
C
D
E
F
G
H
I
4
8
7
9
11
2
14
4
10
8
7
6
1
2


8
A
B
C
D
E
F
G
H
I
4
8
7
9
11
2
14
4
10
8
7
6
1
2

Esempio



Algoritmo di Prim: Analisi

L'efficienza dell'algoritmo di Prim dipende dalla coda con priorità

- Se si utilizza uno heap binario:
  - Inizializzazione: $O(n \log n)$
  - Il ciclo principale viene eseguito $O(n)$ volte
    - Ogni operazione extractMin() costa $O(\log n)$
  - Il ciclo interno viene eseguito $O(m)$ volte
    - Ogni operazione decreaseKey() costa $O(\log n)$
  - Tempo totale: $O(n + n \log n + m \log n) = O(m \log n)$
    - Asintoticamente uguale a quello di Kruskal
- Cosa succede se la coda con priorità è implementata tramite vettore non ordinato?

Discussione

Vero o falso

- L'arco con peso minimo è sicuro
- L'arco con il secondo peso minimo è sicuro
- L'arco con il terzo peso minimo è sicuro

Albero di copertura minima in un piano

- Input: n punti nel piano
- Il peso di una coppia di punti è dato dalla distanza euclidea fra di essi
- Trovare un insieme di connessioni di peso minimo
- Da non confendere con gli Steiner tree

Applicazioni

Applicazioni dirette nella progettazione di

- Reti di telecomunicazione
- Reti idriche
- Reti di trasporto
- Reti elettriche

Alcuni utilizzi particolari

- Segmentazione di immagini
- Riconoscimento scrittura manuale
- Disegno di circuiti elettronici
- Progettazione tassonomie

Prospettiva storica

- $O(m \log n)$:
  - Primo algoritmo: Boruvka (1926)
  - Kruskal (1956)
  - Prim (1957), ma anche Jarnik (1930)
- $O(m + n \log n)$:
  - Fredman-Tarjan (1987)
  - Modifica di Prim che utilizza gli heap di Fibonacci
- $O(m + n)$:
  - Algoritmo probabilistico di Karger, Klein, Tarjan (1995)
  - Vari algoritmi in tempo lineare per casi particolari
  - Questione aperta se si possa risolvere il problema in tempo lineare deterministico

Conclusioni

Vantaggi

- Semplici da programmare
- Molto efficienti
- Quando è possibile dimostrare la proprietà di scelta ingorda, danno la soluzione ottima
- La soluzione sub-ottima può essere accettabile

Svantaggi

- Non sempre applicabili se si vuole la soluzione ottima