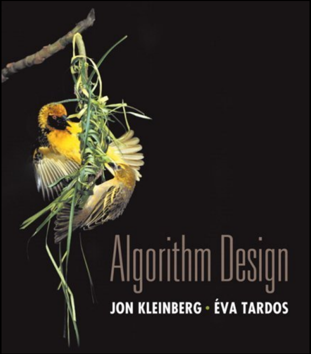


Chapter 4

Greedy Algorithms

Greedy Algorithms

Problemi di ottimizzazione. Vogliamo trovare non una semplice soluzione ma la *migliore* soluzione.

Un algoritmo greedy viene utilizzato per risolvere problemi di ottimizzazione (in alcuni casi ci riesce, in altri no).

Un algoritmo greedy lavora in passi:

- Ad ogni passo fa la scelta più promettente subito (secondo un criterio locale)
  - senza pensare alle conseguenze future
  - senza mai più riconsiderare questa scelta
- La speranza è che la scelta di un ottimo locale ad ogni passo, conduce ad un ottimo globale.

Greedy Algorithms

Problemi che vedremo:

- Interval Scheduling
- Interval Partitioning
- Scheduling to Minimize Lateness
- Optimal Caching
- Shortest Paths in a Graph
- Minimum Spanning Tree
- Huffman Codes and Data Compression

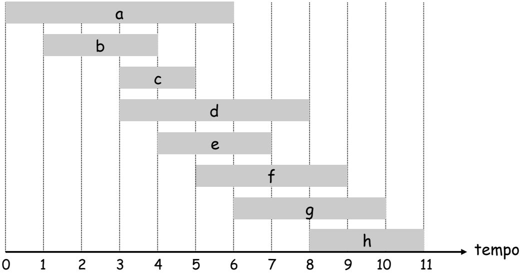
Esercizi:

- Coin Changing
- Selecting Breakpoints
- Fractional Knapsack problem
- Connecting wires
- Collecting coins

4.1 Interval Scheduling

Schedulazione intervalli.

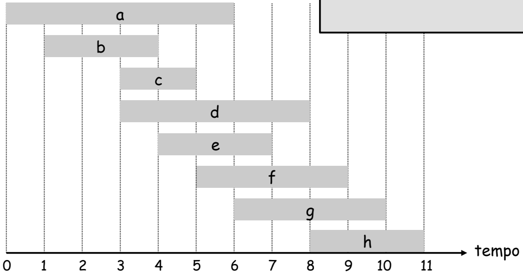
- Job j inizia a s_j e finisce a f_j.
- Due job sono compatibili se non hanno intersezione vuota.
- Obiettivo: trovare sottoinsieme massimale di job mutuamente compatibili.



Schedulazione intervalli.

- Job j inizia a s_j e finisce a f_j.
- Due job sono compatibili se non hanno intersezione vuota.
- Obiettivo: trovare sottoinsieme massimale di job mutuamente compatibili.

Qual'è la soluzione ottimale?

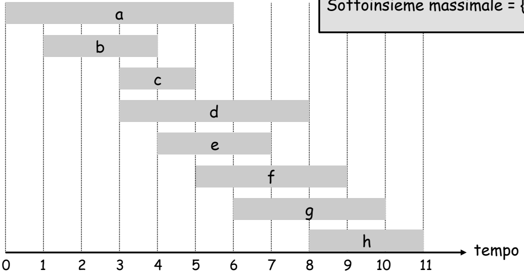


Schedulazione intervalli.

- Job j inizia a s_j e finisce a f_j.
- Due job sono compatibili se non hanno intersezione vuota.
- Obiettivo: trovare sottoinsieme massimale di job mutuamente compatibili.

Qual'è la soluzione ottimale?

Sottoinsieme massimale = {b,e,h}



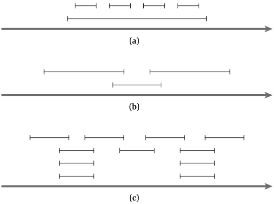
Schedulazione intervalli: Algoritmi Greedy

Scelta Greedy. Considera job in *qualche* ordine. Prendere job se è compatibile con quelli già presi.

- [tempo inizio minimo] Considera job in ordine crescente del tempo di inizio s_j.
- [tempo fine minimo] Considera job in ordine crescente del tempo di fine f_j.
- [intervallo più corto] Considera job in ordine crescente del tempo della lunghezza dell'intervallo $f_j - s_j$.
- [minori conflitti] Per ogni job j, contare il numero c_j di job che lo intersecano. Considera job in ordine crescente di conflitti c_j.

Schedulazione intervalli: Algoritmi Greedy

Scelta Greedy. Considera job in *qualche* ordine. Prendere job se è compatibile con quelli già presi.



tempo inizio minimo

intervallo più corto

minori conflitti

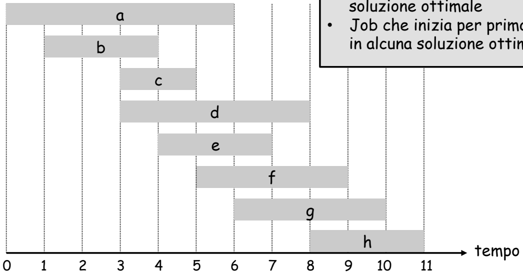
non portano all'ottimo!

Figure 4.1 Some instances of the Interval Scheduling Problem on which natural greedy algorithms fail to find the optimal solution. In (a), it does not work to select the interval that starts earliest; in (b), it does not work to select the shortest interval; and in (c), it does not work to select the interval with the fewest conflicts.

Schedulazione intervalli.

- Job j inizia a s_j e finisce a f_j.
- Due job sono compatibili se non hanno intersezione vuota.
- Obiettivo: trovare sottoinsieme massimale di job mutuamente compatibili.

Osservare:
- Job più corto c non è in alcuna soluzione ottimale
- Job che inizia per primo a non è in alcuna soluzione ottimale



Schedulazione intervalli: Algoritmo Greedy

Algoritmo Greedy. Considera job in ordine crescente del tempo di fine f_j. Prendere job se è compatibile con quelli già presi.

Initially let R be the set of all requests, and let A be empty
While R is not yet empty
  Choose a request $i \in R$ that has the smallest finishing time
  Add request i to A
  Delete all requests from R that are not compatible with request i
EndWhile
Return the set A as the set of accepted requests

Schedulazione intervalli: Algoritmo Greedy

Initially let R be the set of all requests, and let A be empty
While R is not yet empty
  Choose a request $i \in R$ that has the smallest finishing time
  Add request i to A
  Delete all requests from R that are not compatible with request i
EndWhile
Return the set A as the set of accepted requests

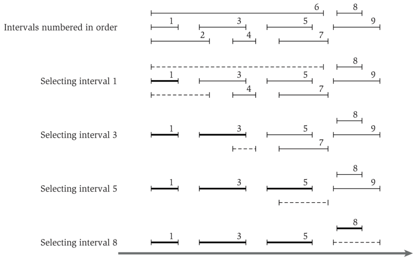


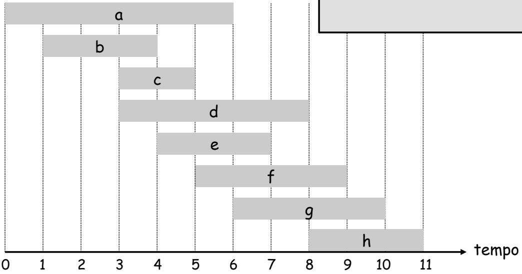
Figure 4.2 Sample run of the Interval Scheduling Algorithm. At each step the selected intervals are darker lines, and the intervals deleted at the corresponding step are indicated with dashed lines.

Schedulazione intervalli

Schedulazione intervalli.

- Job j inizia a s_j e finisce a f_j.
- Due job sono compatibili se non hanno intersezione vuota.
- Obiettivo: trovare sottoinsieme massimale di job mutuamente compatibili.

Qual'è la soluzione greedy?



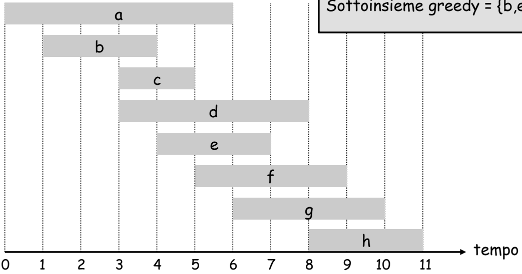
Schedulazione intervalli

Schedulazione intervalli.

- Job j inizia a s_j e finisce a f_j.
- Due job sono compatibili se non hanno intersezione vuota.
- Obiettivo: trovare sottoinsieme massimale di job mutuamente compatibili.

Qual'è la soluzione greedy?

Sottoinsieme greedy = {b,e,h}



Algoritmo Greedy. Considera job in ordine crescente del tempo di fine f_j. Prendere job se è compatibile con quelli già presi.

```
Ordina job per tempo di fine f₁ ≤ f₂ ≤ ... ≤ fₙ.

  insieme job scelti
A ← {1}
for j = 2 to n {
   if (job j è compatibile con A)
      A ← A ∪ {j}
}
return A
```

Schedulazione intervalli: Implementazione Algoritmo Greedy

Implementazione Algoritmo Greedy. O(n log n).

- Denota j* l'ultimo job aggiunto ad A.
- Job j è compatibile con A se $s_j \geq f_{j^*}$.

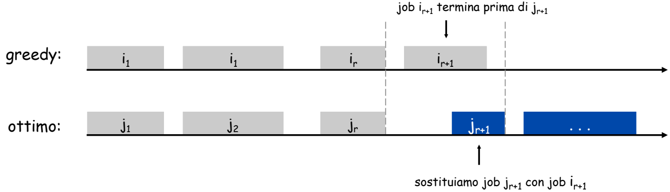
```
Ordina job per tempo di fine f_1 ≤ f_2 ≤ ... ≤ f_n.

A ← {1}
j* = 1
for j = 2 to n {
   if s_j ≥ f_j*
      {A ← A ∪ {j}
       j* = j
      }
}
return A
```

Schedulazione intervalli: Analisi

Teorema. L'algoritmo greedy è ottimale.

Prova. (per assurdo)

- Assumiamo che la scelta greedy non sia ottimale.
- Siano $i_1, i_2, \ldots i_k$ job scelti in modo greedy.
- Siano $j_1, j_2, \ldots j_m$ job in una soluzione ottimale con
  $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ dove r è il più grande valore possibile.

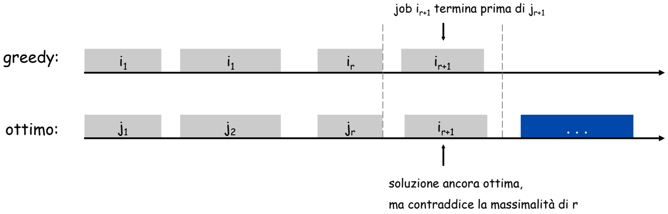


Schedulazione intervalli: Analisi

Teorema. L'algoritmo greedy è ottimale.

Prova. (per assurdo)

- Assumiamo che la scelta greedy non sia ottimale.
- Siano $i_1, i_2, \ldots i_k$ job scelti in modo greedy.
- Siano $j_1, j_2, \ldots j_m$ job in una soluzione ottimale con $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ dove r è il più grande valore possibile.

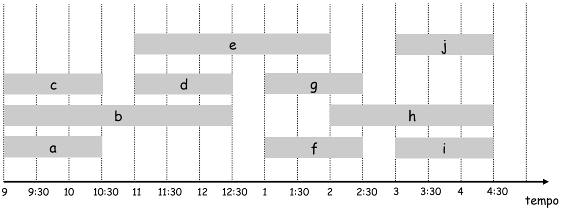


4.1 Interval Partitioning

Interval partitioning.

- Lezione j inizia ad s_j e termina a f_j.
- Obiettivo: trovare il minimo numero di aule per schedulare tutte le lezioni in modo che non ci siano due lezioni contemporaneamente nella stessa aula.

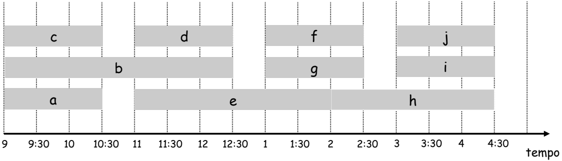
Esempio: Schedulazione con 4 aule per 10 lezioni.



Interval Partitioning

Interval partitioning.
- Lezione j inizia ad s_j e termina a f_j.
- Obiettivo: trovare il minimo numero di aule per schedulare tutte le lezioni in modo che non ci siano due lezioni contemporaneamente nella stessa aula.

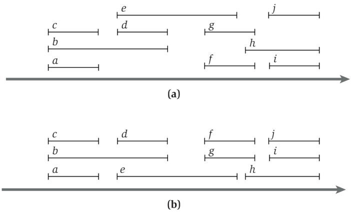
Esempio: Schedulazione con 3 aule per le stesse 10 lezioni.



Interval Partitioning

Interval partitioning.

- Lezione j inizia ad s_j e termina a f_j.
- Obiettivo: trovare il minimo numero di aule per schedulare tutte le lezioni in modo che non ci siano due lezioni contemporaneamente nella stessa aula.



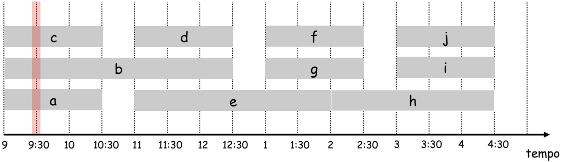
Si può far meglio?
Si possono usare 2 aule?

Figure 4.4 (a) An instance of the Interval Partitioning Problem with ten intervals (*a* through *j*). (b) A solution in which all intervals are scheduled using three resources: each row represents a set of intervals that can all be scheduled on a single resource.

Interval Partitioning: Limiti inferiori per soluzione ottima

Definizione. La profondità di un insieme di intervalli è il massimo numero di intervalli che contiene un dato tempo.

Esempio: Profondità insieme intervalli = 3
a, b, c contengono 9:30



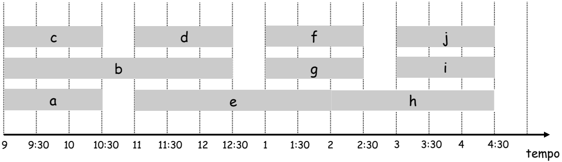
Interval Partitioning: Limiti inferiori per soluzione ottima

Definizione. La profondità di un insieme di intervalli è il massimo numero di intervalli che contiene un dato tempo.

Osservazione. Numero di aule che necessitano $\geq$ profondità.

Esempio: Aule nella schedulazione = 3 $\Longrightarrow$ schedulazione ottimale

Domanda. Esiste sempre una schedulazione uguale alla profondità degli intervalli?



Interval Partitioning: Algoritmo Greedy

Algoritmo Greedy. Consideriamo lezioni in ordine crescente di tempo di inizio: assegniamo lezioni ad aule compatibili.

```
Ordina intervalli per tempo di inizio s1 ≤ s2 ≤ ... ≤ sn.
d ← 0      numbero aule allocate

for j = 1 to n {
   if (lezione j è compatibile con qualche aula k)
      schedula lezione j nell'aula k
   else
      alloca una nuova aula d + 1
      schedula lezione j nell'aula d + 1
      d ← d + 1
}
```

Interval Partitioning: Algoritmo Greedy

Algoritmo Greedy. Consideriamo lezioni in ordine crescente di tempo di inizio: assegniamo lezioni ad aule compatibili.

```
Ordina intervalli per tempo di inizio s1 ≤ s2 ≤ ... ≤ sn.
d ← 0

for j = 1 to n {
   if (lezione j è compatibile con qualche aula k)
      schedula lezione j nell'aula k
   else
      alloca una nuova aula d + 1
      schedula lezione j nell'aula d + 1
      d ← d + 1
}
```

Implementazione. O(n log n).

- Per ogni aula k, manteniamo il tempo di fine dell'ultimo job aggiunto. *Coda a priorità* per i tempi di fine per ogni aula.

Interval Partitioning: Analisi Algoritmo Greedy

Osservazione. L'algoritmo greedy non schedula mai due lezioni incompatibili nella stessa aula.

Teorema. L'algoritmo greedy è ottimale.

Prova.

- Sia d = numero aule allocate dall'algoritmo greedy.
- Aula d è allocata perchè si deve schedulare un job, diciamo j, che è incompatibile con le altre d-1 aule.
- Dato l'ordinamento rispetto al tempo di inizio, tutte le incompatibiità sono dovute a lezioni che iniziano prima di s_j.
- Quindi, ci sono d lezioni con intersezione al tempo $s_j + \varepsilon$.
- Profondità almeno d.
- Tutte le schedulazioni necessitano di un numero di aule $\geq$ d. ▪

4.2 Scheduling to Minimize Lateness

Schedulazione per minimizzare ritardo

Problema minimizzazione ritardo.

- Una singola risorsa processa un job alla volta.
- Job j richiede t_j unità di tempo e deve terminare al tempo d_j.
- Se j inizia al tempo s_j, finisce al tempo $f_j = s_j + t_j$.
- Ritardo: $\ell_j = \max \{ 0, f_j - d_j \}$.
- Obiettivo: schedulare tutti job per minimizzare massimo ritardo $L = \max \ell_j$.

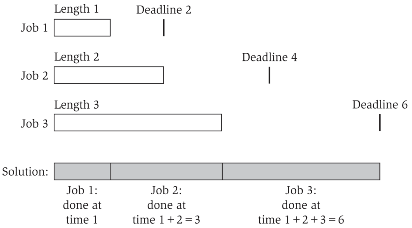


Figure 4.5 A sample instance of scheduling to minimize lateness.

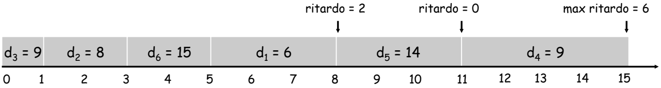
Schedulazione per minimizzare ritardo

Problema minimizzazione ritardo.

- Una singola risorsa processa un job alla volta.
- Job j richiede t_j unità di tempo ed deve terminare al tempo d_j.
- Se j inizia al tempo s_j, finisce al tempo $f_j = s_j + t_j$.
- Ritardo: $\ell_j = \max\{0, f_j - d_j\}$.
- Obiettivo: schedulare tutti job per minimizzare massimo ritardo $L = \max \ell_j$.

Esempio:

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Schedulazione per minimizzare ritardo: Algoritmi greedy

Metodo Greedy. Considera job in qualche ordine.

- [Minimo tempo di processamento] Considera job in ordine crescente di tempo di processamento t_j.

- [Minima deadline] Considera job in ordine crescente di deadline d_j.

- [Slack minimo] Considera job in ordine crescente di tempo di slack $d_j - t_j$.

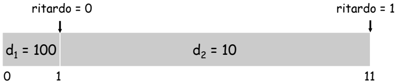
Minimizzare Ritardo: Greedy Algorithms

Metodo Greedy. Considera job in qualche ordine.

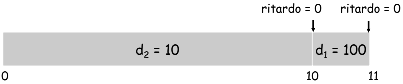
- [Minimo tempo di processamento] Considera job in ordine crescente di tempo di processamento t_j.

	1	2
t_j	1	10
d_j	100	10

controesempio



scelta greedy



ottimo

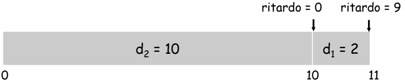
Minimizzare Ritardo: Greedy Algorithms

Metodo Greedy. Considera job in qualche ordine.

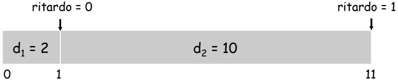
- [Slack minimo] Considera job in ordine crescente di tempo di slack $d_j - t_j$.

	1	2
t_j	1	10
d_j	2	10

controesempio



scelta greedy

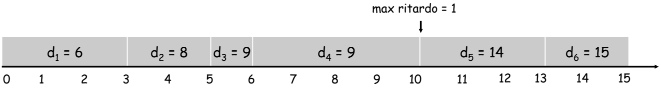


ottimo

Minimizzare Ritardo: Greedy Algorithm

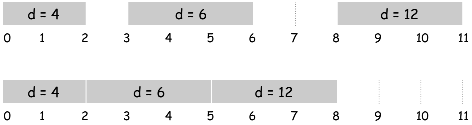
Greedy algorithm. Minima deadline.

```
Ordina n job per deadline d1 ≤ d2 ≤ … ≤ dn

t ← 0
for j = 1 to n
   Assegna job j all'intervallo [t, t + tj]
   sj ← t, fj ← t + tj
   t ← t + tj
output intervalli [sj, fj]
```



Minimizzare Ritardo: Nessun tempo inattivo

Osservazione. Vi è una schedulazione ottimale senza tempi inattivi.



Osservazione. La schedulazione greedy non ha tempi inattivi.

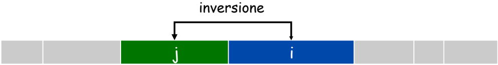
Minimizzare Ritardo: Ottimalità Greedy Algorithm

Exchange argument.

- Consideriamo una schedulazione ottimale
- Modifichiamola gradualmente, preservando l'ottimalità ad ogni passo
- L'ultima schedulazione modificata è quella greedy

Minimizzare Ritardo: Inversions

Definizione. Una inversione nella schedulazione S è una coppia di job i e j tale che: i < j ma j è schedulato prima di i.

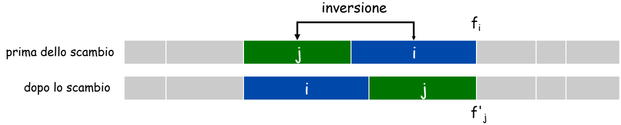


Osservazione. La schedulazione greedy non ha inversioni.

Osservazione. Se una schedulazione (senza tempi inattivi) ha una inversione, allora ne ha una con job invertiti schedulati consecutivamente.

Minimizzare Ritardo: Inversioni

Definizione. Una inversione nella schedulazione S è una coppia di job i e j tale che: i < j ma j è schedulato prima di i.



Claim. Scambiando due job adiacenti ed invertiti, riduce il numero di inversioni di 1 e non incrementa il ritardo massimo.

Prova. Sia ℓ il ritardo prima dello scambio, e sia ℓ' dopo lo scambio.

- $\ell'_k = \ell_k$ per tutti $k \neq i, j$
- $\ell'_i \leq \ell_i$
- Se job j è in ritardo:

ricorda: $\ell_j = \max\{0, f_j - d_j\}$

$$\begin{aligned}
\ell'_j &= f'_j - d_j && \text{(definizione)} \\
&= f_i - d_j && (j \text{ finisce al tempo } f_i) \\
&\leq f_i - d_i && (i < j) \\
&\leq \ell_i && \text{(definizione)}
\end{aligned}$$

Minimizzare Ritardo: Analisi Algoritmo Greedy

Teorema. La schedulazione greedy S è ottimale.

Prova. Definiamo S* come una schedulazione ottimale con il minimo numero di inversioni.

- Possiamo assumere che S* non ha tempi inattivi.
- Se S* non ha inversioni, allora S = S*.
- Se S* ha almeno una inversione, sia i-j una inversione adiacente.
  - scambiando i e j non incrementiamo il ritardo massimo e diminuiamo il numero di inversioni.
  - questo contraddice la definizione di S* ▪

Metodi per mostrare correttezza algoritmi greedy

L'algoritmo greedy è sempre in vantaggio. Mostrare che dopo ogni passo dell'algoritmo greedy, la sua soluzione è almeno tanto buona come quella di ogni altro algoritmo.

Argomento di scambio (Exchange argument). Trasformare gradualmente una soluzione ottima a quella dell'algoritmo greedy lasciando il suo valore ottimo.

Strutturale. Trovare un semplice limite "strutturale" che ogni soluzione deve rispettare. Mostrare poi che l'algoritmo greedy raggiunge sempre quel limite.

4.3 Optimal Caching

Caching offline ottimale

Caching.

- Cache con capacità di memorizzare k elementi.
- Sequenza di m richieste di elementi $d_1, d_2, \ldots, d_m$.
- *Cache hit:* elemento già nella cache quando richiesto.
- *Cache miss:* elemento non nella cache quando richiesto: è necessario portarlo nella cache e rimuovere un altro elemento (se la cache è piena).

Obiettivo. Schedulazione rimozioni che minimizzano il numero di cache miss.

Esempio: k = 2, cache all'inizio = ab,
richieste: a, b, c, b, c, a, a, b.

Scedulazione ottimale rimozione: 2 cache miss

richieste	cache	
a	a	b
b	a	b
c	c	b
b	c	b
c	c	b
a	a	b
a	a	b
b	a	b

a	a	b
b	a	b
c	a	c
b	a	b
c	c	b
a	a	b
a	a	b
b	a	b

Caching offline ottimale: Farthest-In-Future

Farthest-in-future. Rimuovere elemento nella cache che sarà richiesto il più tardi possibile nel futuro.

occupazione cache: a b c d e f

richieste future: g a b c e d a b b a c d e a f a d e f g h ...

cache miss

rimuovere elemento f dalla cache

When d_i needs to be brought into the cache,
  evict the item that is needed the farthest into the future

Teorema. Farthest-In-Future è una schedulazione ottimale delle rimozioni.

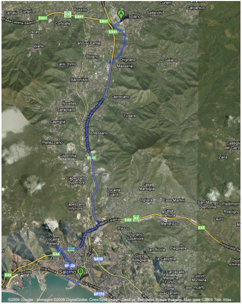
Prova. L'algoritmo ed il teorema sono intuitivi. Ma la prova non è immediata, richiede attenzione e comporta una distinzione di casi. La trovate nel libro.

[L. Belady. A study of replacement algorithms for virtual storage computers. IBM *Systems Journal 5* (1966), 78-101.]

4.4 Shortest Paths in a Graph

Cammino minimo dal campus Fisciano
alla stazione ferroviaria di Salerno

Google maps



Indicazioni stradali per Piazza Vittorio Veneto
14,5 km – circa 22 min

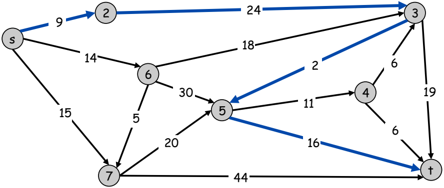
Problema cammino minimo

Dati input:

- Grafo diretto $G = (V, E)$.
- Sorgente s, destinazione t.
- ℓ_e = lunghezza arco e.

Problema del cammino minimo: trovare cammino minimo diretto da s a t.

costo cammino = somma costi archi del cammino



Costo cammino s-2-3-5-t
= 9 + 24 + 2 + 16
= 51.

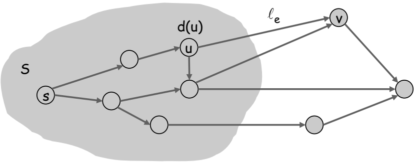
Algoritmo di Dijkstra.

- Mantenere l'insieme dei nodi esplorati S per i quali abbiamo determinato le distanze dei cammini minimi d(u) da s ad u.
- Inizializzare S = { s }, d(s) = 0.
- Ripetutamente scegliere un nodo non esplorato v che minimizza

$$\pi(v) = \min_{e = (u,v) \,:\, u \in S} d(u) + \ell_e ,$$

aggiungere v a S, e porre d(v) = π(v).

Cammino minimo a qualche u nella parte esplorata, seguito da un singolo arco (u, v)



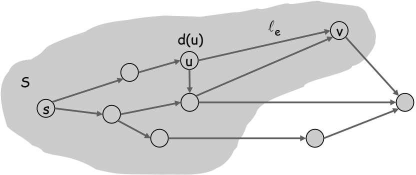
Algoritmo di Dijkstra.

- Mantenere l'insieme dei nodi esplorati S per i quali abbiamo determinato le distanze dei cammini minimi d(u) da s ad u.
- Inizializzare S = { s }, d(s) = 0.
- Ripetutamente scegliere un nodo non esplorato v che minimizza

$$\pi(v) = \min_{e = (u,v)\,:\, u \in S} d(u) + \ell_e ,$$

aggiungere v a S, e porre d(v) = π(v).

Cammino minimo a qualche u nella parte esplorata, seguito da un singolo arco (u, v)



Algoritmo di Dijkstra

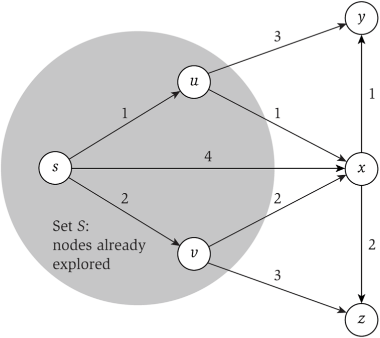


Figure 4.7 A snapshot of the execution of Dijkstra's Algorithm. The next node that will be added to the set S is x, due to the path through u.

Algoritmo di Dijkstra: Prova di correttezza

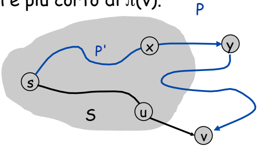
Invariante. Per ogni nodo $u \in S$, $d(u)$ è la lunghezza del cammino minimo s-u.

Prova. (per induzione su $|S|$)

Caso base: $|S| = 1$ è banale.

Ipotesi induttiva: Assumiamo vera per $|S| = k \geq 1$.

- Sia v il prossimo nodo aggiunto ad S, e sia u-v l'arco scelto.
- Il cammino minimo s-u più (u, v) è un cammino minimo s-v di lunghezza $\pi(v)$.
- Sia P un cammino minimo s-v. Vediamo che non è più corto di $\pi(v)$.
- Sia x-y il primo arco in P che lascia S, e sia P' il cammino da s a x (parte di P).
- P è già troppo lungo appena lascia S.



$$\ell(P) \geq \ell(P') + \ell(x,y) \geq d(x) + \ell(x,y) \geq \pi(y) \geq \pi(v)$$

- $\ell(P) \geq \ell(P') + \ell(x,y)$: pesi non negativi
- $\ell(P') + \ell(x,y) \geq d(x) + \ell(x,y)$: ipotesi induttiva $\ell(P') \geq d(x)$
- $d(x) + \ell(x,y) \geq \pi(y)$: definizione di $\pi(y)$
- $\pi(y) \geq \pi(v)$: Dijkstra sceglie v invece di y

Algoritmo di Dijkstra: Implementazione

Per ogni nodo non ancora esplorato, calcola $\pi(v) = \min_{e=(u,v)\,:\,u \in S} d(u) + \ell_e$.

- Prossimo nodo da esplorare = nodo con minimo $\pi(v)$.
- Quando esploriamo v, per ogni arco incidente e = (v, w), aggiorna

$$\pi(w) = \min \{\ \pi(w),\ \pi(v) + \ell_e\ \}.$$

Dijkstra's Algorithm (G, ℓ)

Let S be the set of explored nodes

    For each $u \in S$, we store a distance $d(u)$

Initially $S = \{s\}$ and $d(s) = 0$

While $S \neq V$

    Select a node $v \notin S$ with at least one edge from S for which

        $d'(v) = \min_{e=(u,v):u \in S} d(u) + \ell_e$ is as small as possible

    Add v to S and define $d(v) = d'(v)$

EndWhile

Algoritmo di Dijkstra: Implementazione

Per ogni nodo non ancora esplorato, calcola $\pi(v) = \min_{e=(u,v) : u \in S} d(u) + \ell_e$.

- Prossimo nodo da esplorare = nodo con minimo $\pi(v)$.
- Quando esploriamo v, per ogni arco incidente e = (v, w), aggiorna

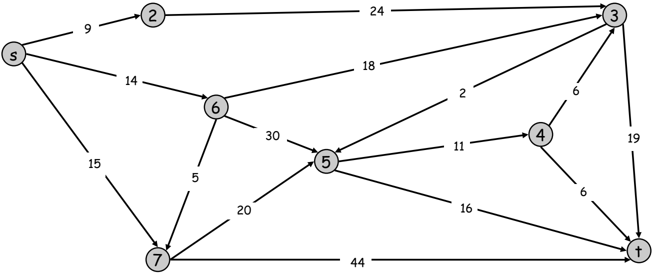
$$\pi(w) = \min \{ \pi(w), \pi(v) + \ell_e \}.$$

Implementazione efficiente. Mantenere una coda a priorità dei nodi non esplorati, sul campo $\pi(v)$.

Operazione PQ	Dijkstra	Array	Binary heap
Insert	n	n	log n
ExtractMin	n	n	log n
ChangeKey	m	1	log n
IsEmpty	n	1	1
Total		n^2	m log n

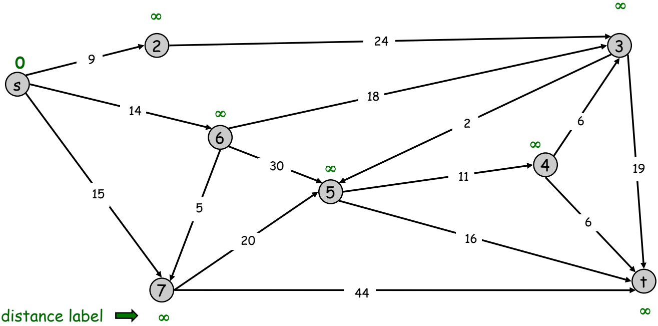
Algoritmo di Dijkstra

Trovare cammino minimo da s a t.



Algoritmo di Dijkstra

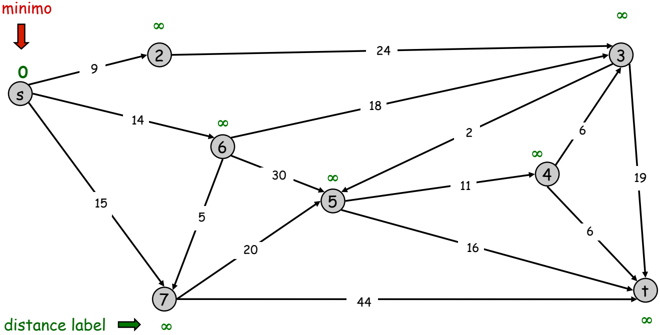
S = { }
PQ = { s, 2, 3, 4, 5, 6, 7, t }



distance label

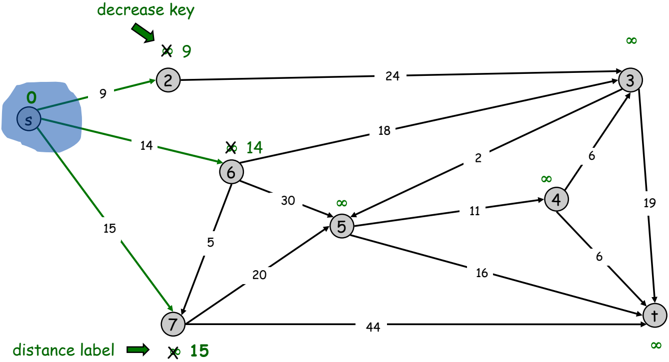
Algoritmo di Dijkstra

S = { }
PQ = { s, 2, 3, 4, 5, 6, 7, t }



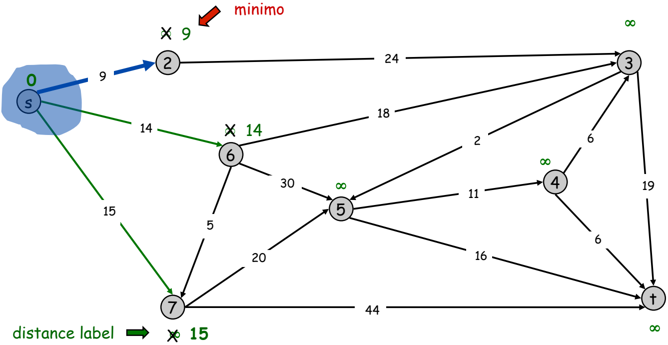
Algoritmo di Dijkstra

S = { s }
PQ = { 2, 3, 4, 5, 6, 7, t }



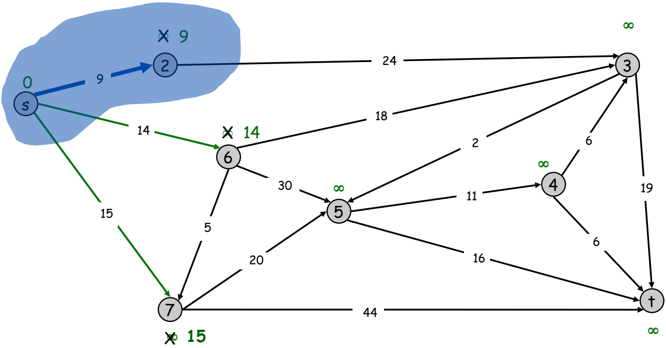
Algoritmo di Dijkstra

S = { s }
PQ = { 2, 3, 4, 5, 6, 7, t }



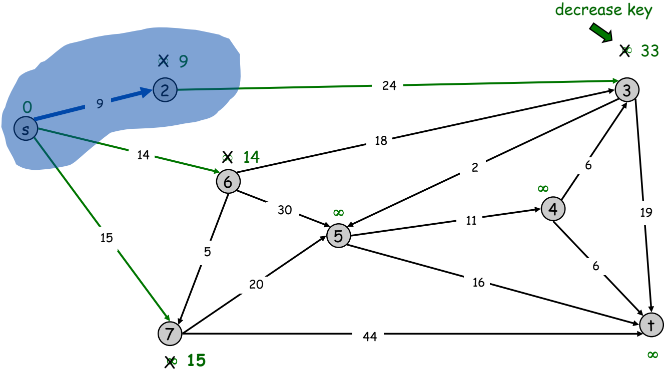
Algoritmo di Dijkstra

S = { s, 2 }
PQ = { 3, 4, 5, 6, 7, t }



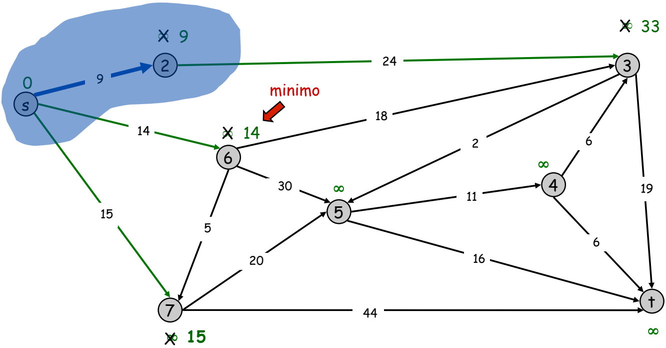
Algoritmo di Dijkstra

S = { s, 2 }
PQ = { 3, 4, 5, 6, 7, t }



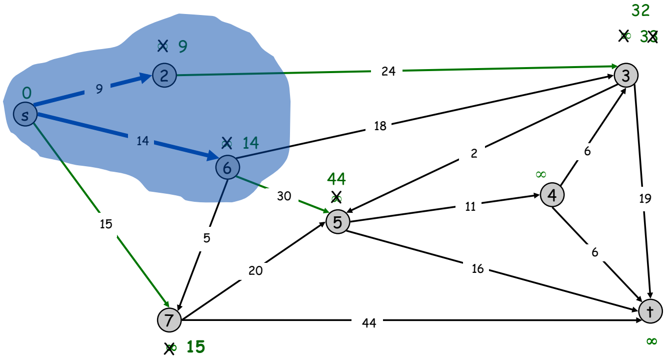
Algoritmo di Dijkstra

S = { s, 2 }
PQ = { 3, 4, 5, 6, 7, t }



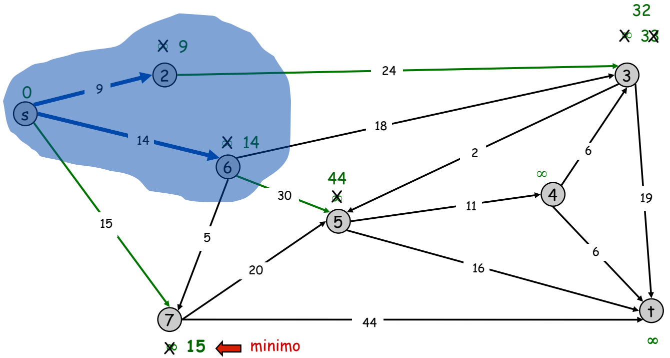
Algoritmo di Dijkstra

S = { s, 2, 6 }
PQ = { 3, 4, 5, 7, t }



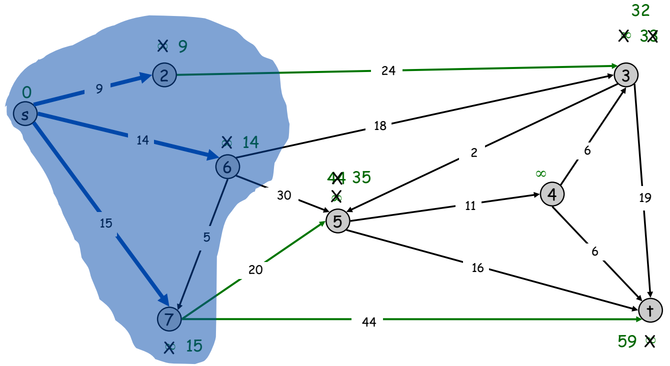
Dijkstra's Shortest Path Algorithm

S = { s, 2, 6 }
PQ = { 3, 4, 5, 7, t }



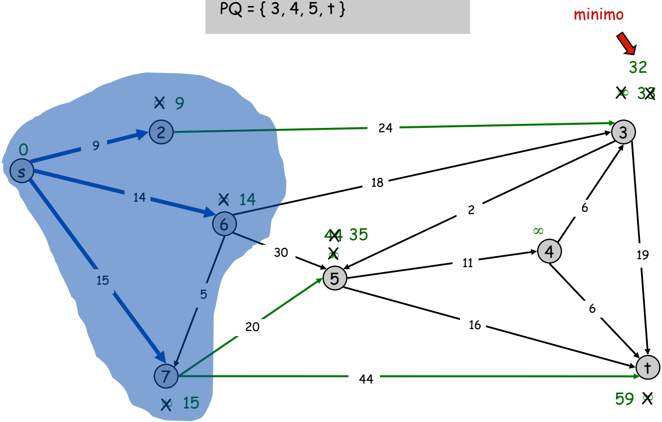
Dijkstra's Shortest Path Algorithm

S = { s, 2, 6, 7 }
PQ = { 3, 4, 5, t }



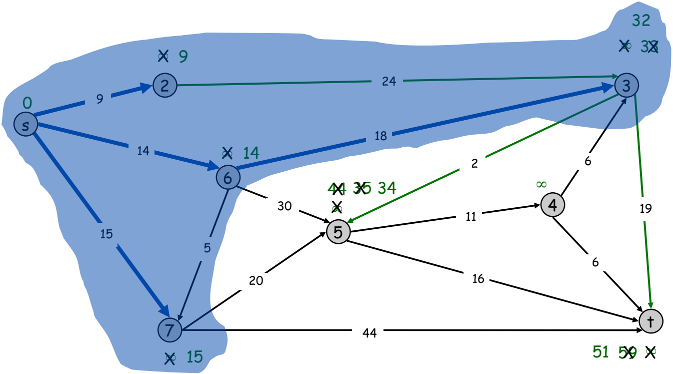
Dijkstra's Shortest Path Algorithm

S = { s, 2, 6, 7 }
PQ = { 3, 4, 5, t }



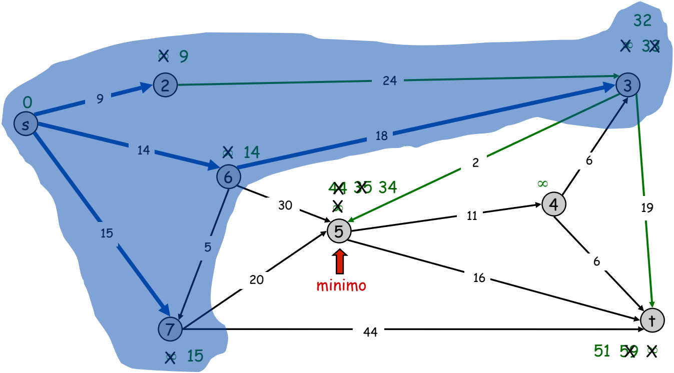
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 6, 7 }
PQ = { 4, 5, t }



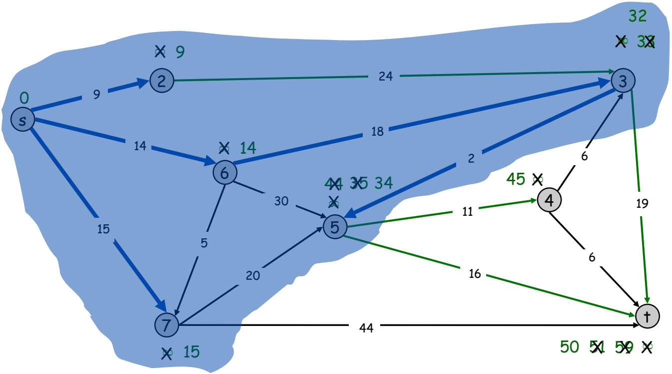
Algoritmo di Dijkstra

S = { s, 2, 3, 6, 7 }
PQ = { 4, 5, t }



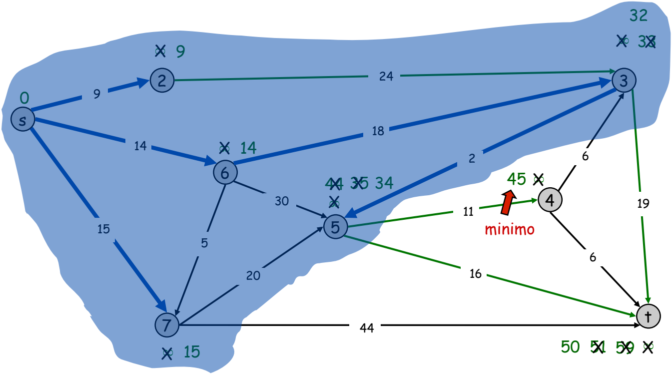
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 5, 6, 7 }
PQ = { 4, t }



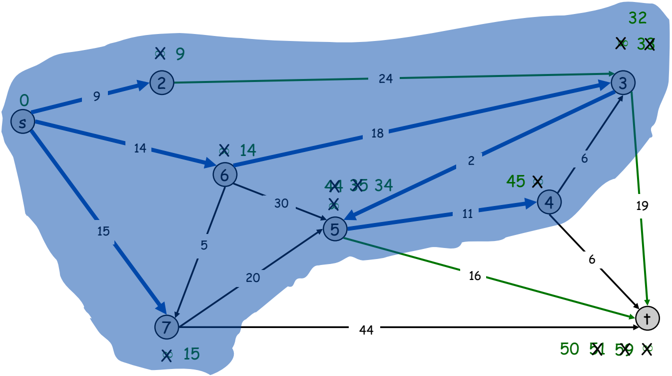
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 5, 6, 7 }
PQ = { 4, t }



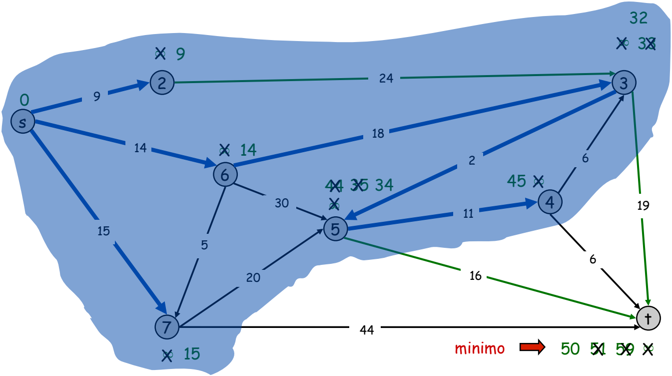
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7 }
PQ = { t }



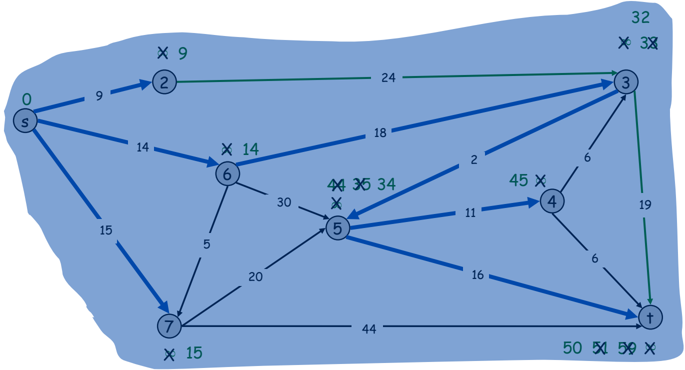
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7 }
PQ = { t }



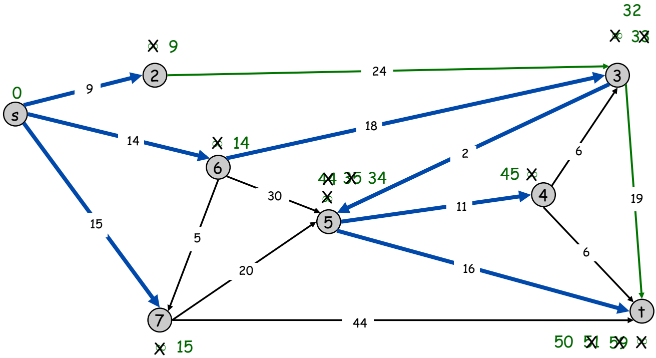
Algoritmo di Dijkstra

S = { s, 2, 3, 4, 5, 6, 7, t }
PQ = { }



Algoritmo di Dijkstra

S = { s, 2, 3, 4, 5, 6, 7, t }
PQ = { }



Edsger Wybe Dijkstra

Rotterdam, 11 maggio 1930 – Nuenen, 6 agosto 2002

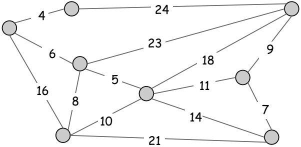
A note on two problems in connection with graphs
Numerische Matematik, vol. 1, 1959, pp. 269-271

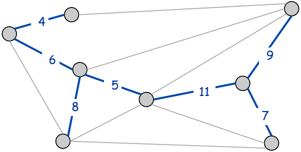
4.5 Minimum Spanning Tree

Minimum Spanning Tree

Abbiamo un insieme di locazioni e vogliamo costruire una rete di comunicazione tra loro:

- rete connessa: ci deve essere un cammino tra ogni coppia
- economicità: costo totale più basso possibile

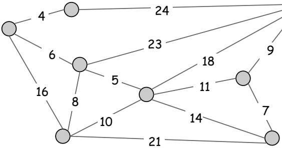




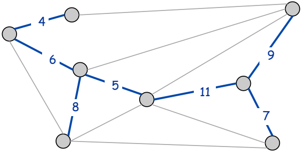
Costo totale 50

Minimum Spanning Tree

Dato un grafo connesso $G = (V, E)$ con pesi a valori reali $c_e > 0$ sugli archi, trovare un insieme di archi $T \subseteq E$ tale che il grafo (V,T) è connesso e la somma dei pesi degli archi in T è minimizzata.

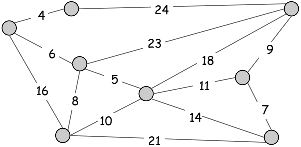


$G = (V, E)$

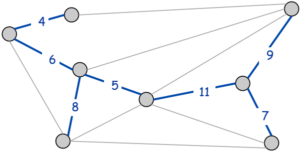


$T, \ \Sigma_{e \in T} c_e = 50$

Dato un grafo connesso $G = (V, E)$ con pesi a valori reali $c_e > 0$ sugli archi, trovare un insieme di archi $T \subseteq E$ tale che il grafo (V,T) è connesso e la somma dei pesi degli archi in T è minimizzata.



$G = (V, E)$



$T, \ \Sigma_{e \in T} c_e = 50$

Lemma. Sia T la soluzione. Allora (V,T) è un albero.

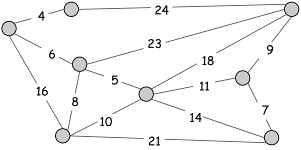
Prova. (V,T) è connesso. Facciamo vedere che non ha cicli.

Per assurdo: se ci fosse un ciclo, prendiamo un arco in tale ciclo.

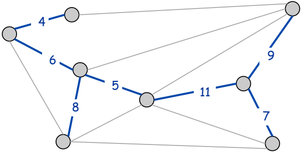
Il grafo $(V,T-\{e\})$ è ancora connesso ed ha costo minore. Contraddizione.

Minimum Spanning Tree (MST)

Minimum spanning tree. Dato un grafo connesso $G = (V, E)$ con pesi a valori reali c_e sugli archi, un MST è un sottoinsieme degli archi $T \subseteq E$ tale che T è uno spanning tree la cui somma dei pesi sugli archi è minimizzata.



$G = (V, E)$



$T, \; \Sigma_{e \in T} c_e = 50$

Teorema di Cayley. Ci sono n^{n-2} spanning tree di K_n.

non si può risolvere con *brute force*

Applicazioni

MST è un problema fondamentale con svariate applicazioni.

- Network design.
  - telephone, electrical, hydraulic, TV cable, computer, road
- Approximation algorithms for NP-hard problems.
  - traveling salesperson problem, Steiner tree
- Indirect applications.
  - max bottleneck paths
  - LDPC codes for error correction
  - image registration with Renyi entropy
  - learning salient features for real-time face verification
  - reducing data storage in sequencing amino acids in a protein
  - model locality of particle interactions in turbulent fluid flows
  - autoconfig protocol for Ethernet bridging to avoid cycles in a network
- Cluster analysis.

Algoritmi Greedy

Algoritmo di Kruskal.
- Inizia con T = ϕ.
- Considera archi in ordine crescente di costo.
- Inserisci l'arco in T se non crea un ciclo.

Algoritmo di Reverse-Delete.
- Inizia con T = E.
- Considera archi in ordine decrescente di costo.
- Cancella l'arco da T se non disconnette T.

Algoritmo di Prim.
- Inizia con T = ϕ e con un nodo radice s.
- Aggiungi arco a T che è incidente solo su un nodo in T e con costo minimo.

Nota bene. Tutti e tre gli algoritmi producono un MST.

Algoritmi di Prim e di Kruskal

Algoritmo di Prim.

- Inizia con T = ϕ e con un nodo radice s.
- Aggiungi arco a T che è incidente solo su un nodo in T e con il costo minimo.

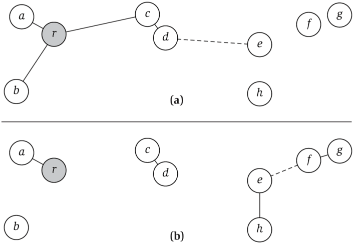


Figure 4.9 Sample run of the Minimum Spanning Tree Algorithms of (a) Prim and (b) Kruskal, on the same input. The first 4 edges added to the spanning tree are indicated by solid lines; the next edge to be added is a dashed line.

Algoritmo di Kruskal.

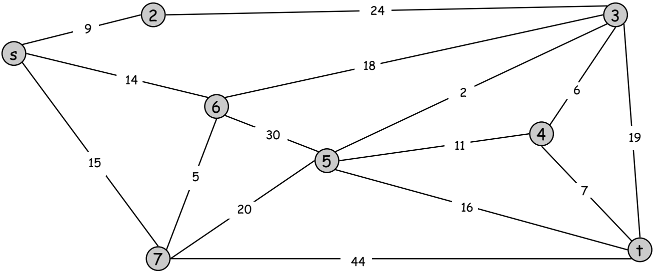
- Inizia con T = ϕ.
- Considera archi in ordine crescente di costo.
- Inserisci l'arco in T se non crea un ciclo.

Algoritmo di Prim

Trovare Minimo Spanning Tree.

- Inizia con T = ϕ e con un nodo radice s.
- Aggiungi arco a T che è incidente solo su un nodo in T e con costo minimo.

T = ϕ

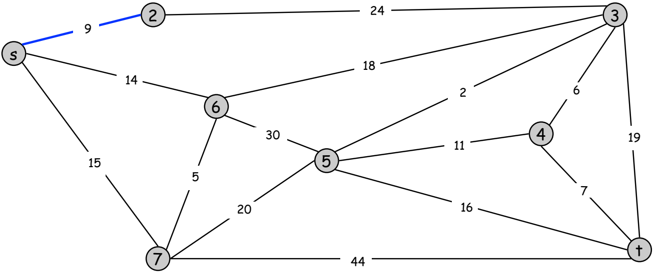


Algoritmo di Prim

Trovare Minimo Spanning Tree.

- Inizia con $T = \phi$ e con un nodo radice s.
- Aggiungi arco a T che è incidente solo su un nodo in T e con costo minimo.

T = { {s,2} }

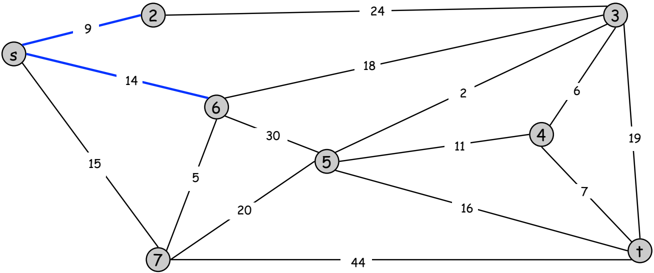


Algoritmo di Prim

Trovare Minimo Spanning Tree.

- Inizia con T = ϕ e con un nodo radice s.
- Aggiungi arco a T che è incidente solo su un nodo in T e con costo minimo.

T = { {s,2}, {s,6} }

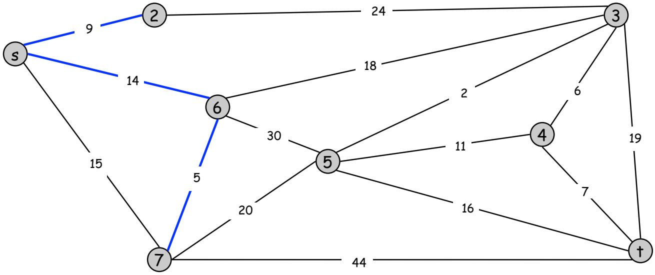


Algoritmo di Prim

Trovare Minimo Spanning Tree.

- Inizia con $T = \phi$ e con un nodo radice s.
- Aggiungi arco a T che è incidente solo su un nodo in T e con costo minimo.

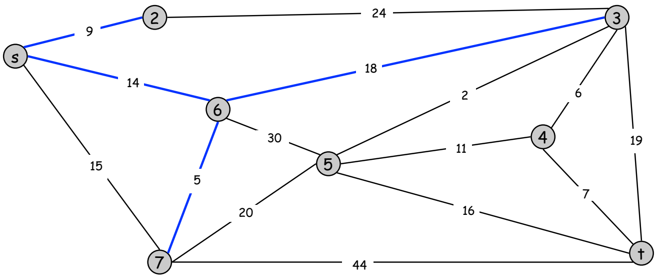
T = { {s,2}, {s,6}, {6,7} }



Trovare Minimo Spanning Tree.

- Inizia con T = ϕ e con un nodo radice s.
- Aggiungi arco a T che è incidente solo su un nodo in T e con costo minimo.

T = { {s,2}, {s,6}, {6,7}, {6,3} }

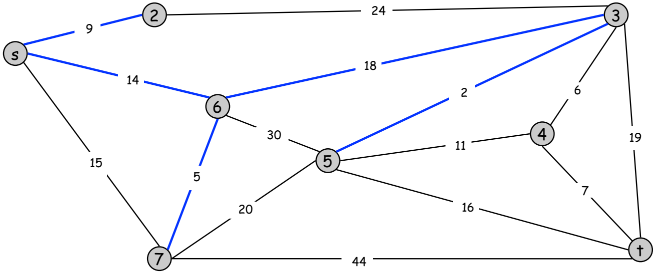


Algoritmo di Prim

Trovare Minimo Spanning Tree.

- Inizia con T = ϕ e con un nodo radice s.
- Aggiungi arco a T che è incidente solo su un nodo in T e con costo minimo.

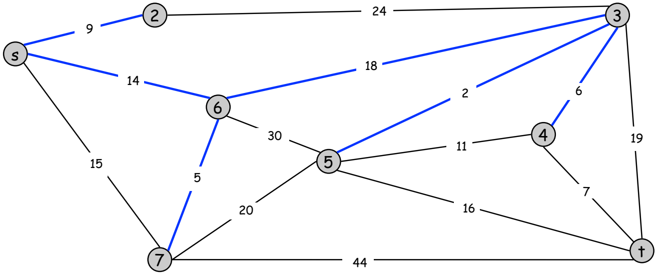
T = { {s,2}, {s,6}, {6,7}, {6,3}, {3,5} }



Trovare Minimo Spanning Tree.

- Inizia con $T = \phi$ e con un nodo radice s.
- Aggiungi arco a T che è incidente solo su un nodo in T e con costo minimo.

T = { {s,2}, {s,6}, {6,7}, {6,3}, {3,5}, {3,4} }

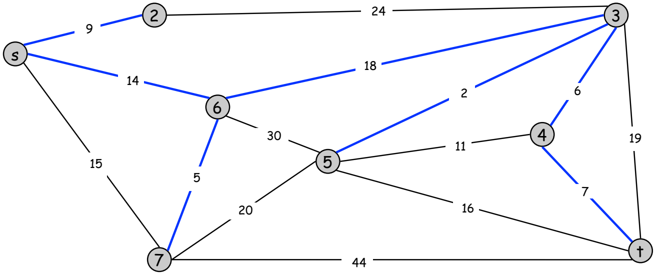


Algoritmo di Prim

Trovare Minimo Spanning Tree.

- Inizia con T = ϕ e con un nodo radice s.
- Aggiungi arco a T che è incidente solo su un nodo in T e con costo minimo.

T = { {s,2}, {s,6}, {6,7}, {6,3}, {3,5}, {3,4}, {4,t} }

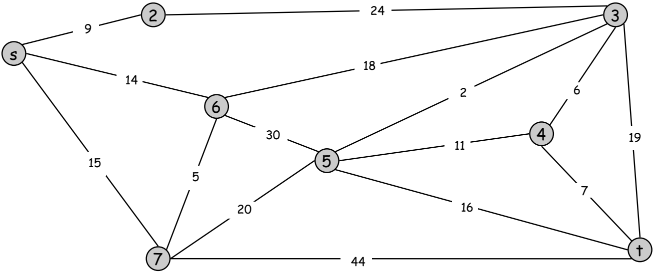


Algoritmo di Kruskal

Trovare Minimo Spanning Tree.

- Inizia con $T = \phi$.
- Considera archi in ordine crescente di costo.
- Inserisci l'arco in T se non crea un ciclo.

$T = \phi$

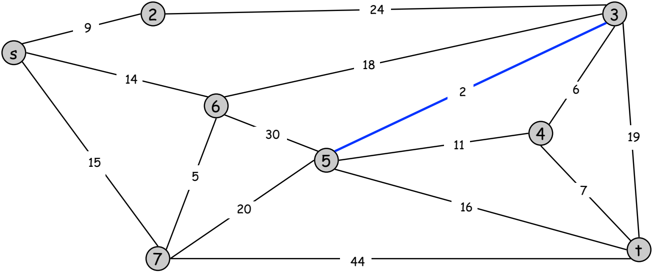


Algoritmo di Kruskal

Trovare Minimo Spanning Tree.

- Inizia con T = ϕ.
- Considera archi in ordine crescente di costo.
- Inserisci l'arco in T se non crea un ciclo.

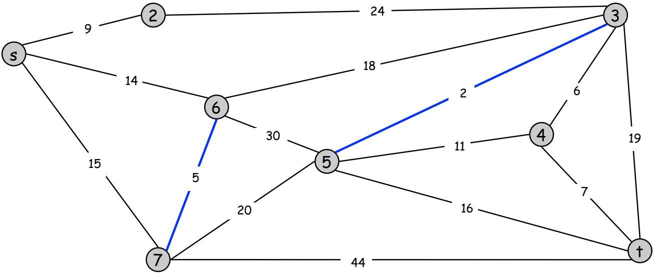
T = { {3,5} }



Trovare Minimo Spanning Tree.

- Inizia con T = ϕ.
- Considera archi in ordine crescente di costo.
- Inserisci l'arco in T se non crea un ciclo.

T = { {3,5}, {6,7} }

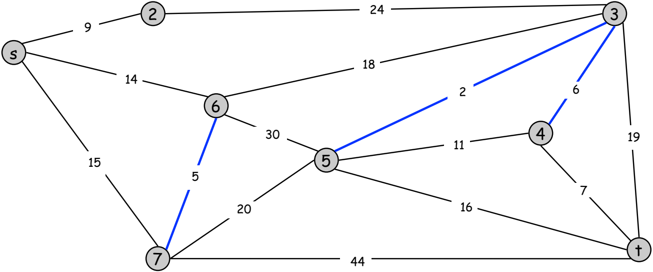


Algoritmo di Kruskal

Trovare Minimo Spanning Tree.

- Inizia con T = ϕ.
- Considera archi in ordine crescente di costo.
- Inserisci l'arco in T se non crea un ciclo.

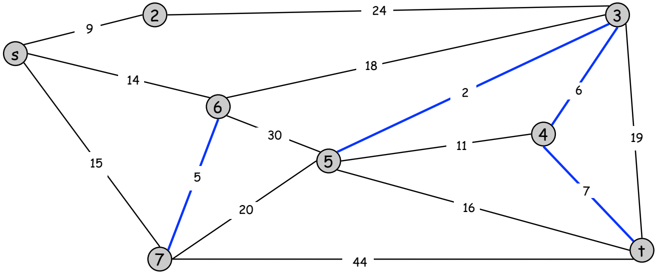
T = { {3,5}, {6,7}, {3,4} }



Trovare Minimo Spanning Tree.

- Inizia con T = ϕ.
- Considera archi in ordine crescente di costo.
- Inserisci l'arco in T se non crea un ciclo.

T = { {3,5}, {6,7}, {3,4}, {4,t} }

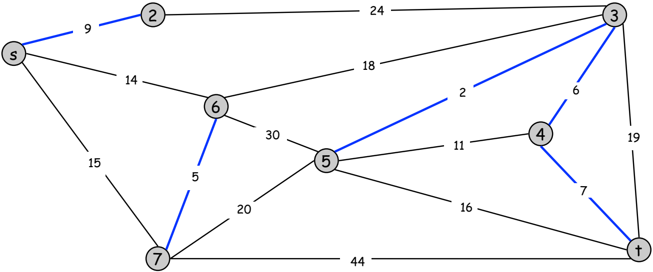


Algoritmo di Kruskal

Trovare Minimo Spanning Tree.

- Inizia con $T = \phi$.
- Considera archi in ordine crescente di costo.
- Inserisci l'arco in T se non crea un ciclo.

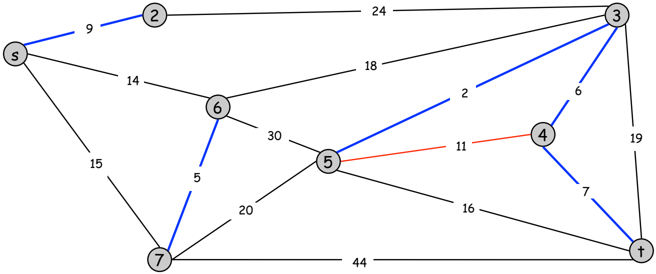
T = { {3,5}, {6,7}, {3,4}, {4,t}, {s,2} }



Trovare Minimo Spanning Tree.

- Inizia con T = ϕ.
- Considera archi in ordine crescente di costo.
- Inserisci l'arco in T se non crea un ciclo.

T = { {3,5}, {6,7}, {3,4}, {4,t}, {s,2} }

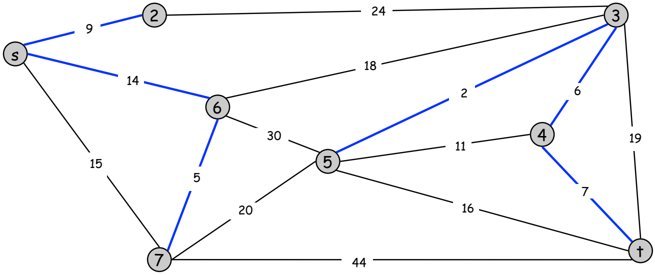


Algoritmo di Kruskal

Trovare Minimo Spanning Tree.

- Inizia con T = ϕ.
- Considera archi in ordine crescente di costo.
- Inserisci l'arco in T se non crea un ciclo.

T = { {3,5}, {6,7}, {3,4}, {4,t}, {s,2}, {s,6} }

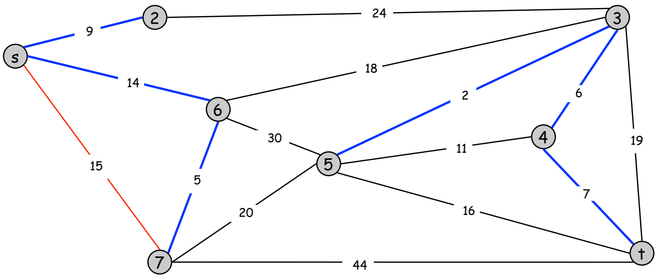


Algoritmo di Kruskal

Trovare Minimo Spanning Tree.

- Inizia con T = ϕ.
- Considera archi in ordine crescente di costo.
- Inserisci l'arco in T se non crea un ciclo.

T = { {3,5}, {6,7}, {3,4}, {4,t}, {s,2}, {s,6} }

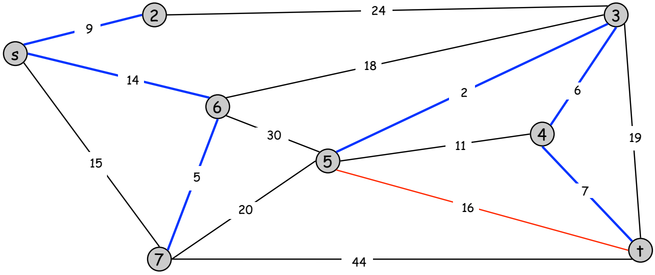


Algoritmo di Kruskal

Trovare Minimo Spanning Tree.

- Inizia con $T = \phi$.
- Considera archi in ordine crescente di costo.
- Inserisci l'arco in T se non crea un ciclo.

T = { {3,5}, {6,7}, {3,4}, {4,t}, {s,2}, {s,6} }

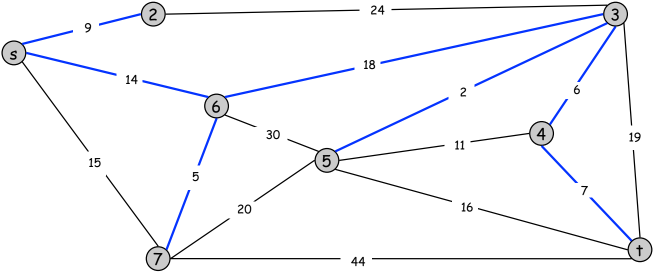


Algoritmo di Kruskal

Trovare Minimo Spanning Tree.

- Inizia con $T = \phi$.
- Considera archi in ordine crescente di costo.
- Inserisci l'arco in T se non crea un ciclo.

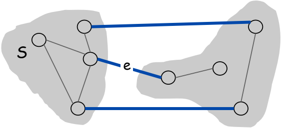
T = { {3,5}, {6,7}, {3,4}, {4,t}, {s,2}, {s,6}, {3,6} }



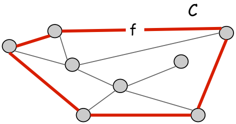
Assunzione per semplificare. Tutti i costi degli archi c_e sono distinti.

Proprietà di taglio. Sia S un sottoinsieme di nodi, e sia e l'arco con il minimo costo che è incidente su esattamente un nodo in S. Allora l'MST contiene e.

Proprietà del ciclo. Sia C un ciclo, e sia f l'arco con il massimo costo in C. Allora l'MST non contiene f.



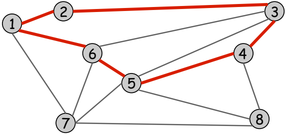
e è nell'MST



f non è nell'MST

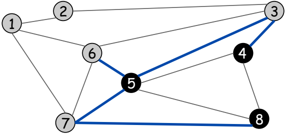
Cicli e tagli

Ciclo. Insieme di archi della forma a-b, b-c, c-d, …, y-z, z-a.



Ciclo C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1

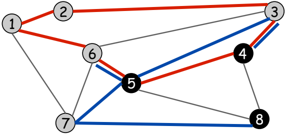
Cutset (Insieme di taglio). Un cut (taglio) è un insieme di nodi S. L'insieme di taglio corrispondente D è il sottoinsieme di archi che sono incidenti su esattamente un nodo in S.



Cut S = { 4, 5, 8 }
Cutset D = 5-6, 5-7, 3-4, 3-5, 7-8

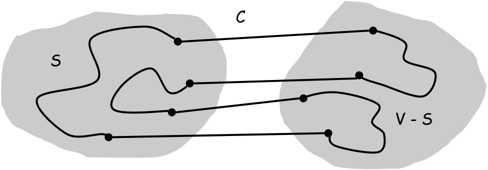
Intersezione Ciclo-Taglio

Claim. Un ciclo ed un cutset si intersecano in un numero pari di archi.



Ciclo C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1
Cutset D = 3-4, 3-5, 5-6, 5-7, 7-8
Intersezione = 3-4, 5-6

Prova.

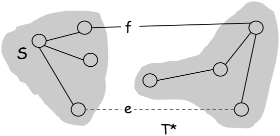


Assunzione per semplificare. Tutti i costi degli archi c_e sono distinti.

Proprietà del taglio. Sia S un sottoinsieme di nodi e sia e l'arco di costo minimo che incide su esattamente un nodo in S. Allora l'MST T* contiene e.

Prova. (per assurdo, usando una argomentazione di scambio)

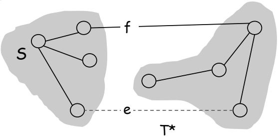
- Supponiamo che e non appartiene a T*.
- Aggiungere e a T* crea un ciclo C in T*.
- L'arco e è sia nel ciclo C e nel cutset D corrispondente ad S $\Rightarrow$ esiste un altro arco, diciamo f, che è sia in C che in D.



Assunzione per semplificare. Tutti i costi degli archi c_e sono distinti.

Proprietà del taglio. Sia S un sottoinsieme di nodi e sia e l'arco di costo minimo che incide su esattamente un nodo in S. Allora l'MST T* contiene e.

Prova. (per assurdo, usando una argomentazione di scambio)
- Supponiamo che e non appartiene a T*.
- Aggiungere e a T* crea un ciclo C in T*.
- L'arco e è sia nel ciclo C e nel cutset D corrispondente ad S $\Rightarrow$ esiste un altro arco, diciamo f, che è sia in C che in D.
- $T' = T^* \cup \{ e \} - \{ f \}$ è ancora uno spanning tree.
- Dato che $c_e < c_f$, cost(T') < cost(T*).
- Questa è una contraddizione. ▪



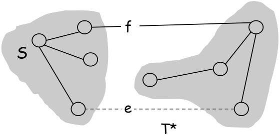
Algoritmi greedy

Assunzione per semplifare. Tutti i costi degli archi c_e sono distinti.

Proprietà del ciclo. Sia C un ciclo in G, e sia f l'arco di costo massimo in C. Allora l'MST T* non contiene f.

Prova. (per assurdo, usando una argomentazione di scambio)

- Supponiamo che f sia in T*.
- La cancellazione di f da T* crea un taglio S in T*.
- L'arco f è sia nel ciclo C sia nel cutset D corrispondente ad S $\Rightarrow$ esiste un altro arco, diciamo e, che è sia in C che in D.



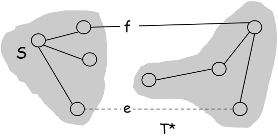
Algoritmi greedy

Assunzione per semplifare. Tutti i costi degli archi c_e sono distinti.

Proprietà del ciclo. Sia C un ciclo in G, e sia f l'arco di costo massimo in C. Allora l'MST T* non contiene f.

Prova. (per assurdo, usando una argomentazione di scambio)

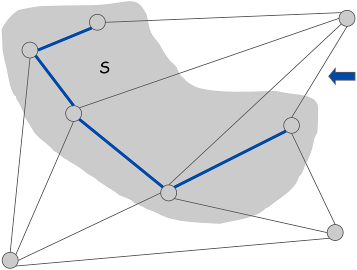
- Supponiamo che f sia in T*.
- La cancellazione di f da T* crea un taglio S in T*.
- L'arco f è sia nel ciclo C sia nel cutset D corrispondente ad S $\Rightarrow$ esiste un altro arco, diciamo e, che è sia in C che in D.
- $T' = T^* \cup \{ e \} - \{ f \}$ è ancora uno spanning tree.
- Dato che $c_e < c_f$, cost(T') < cost(T*).
- Questa è una contraddizione.



Algoritmo di Prim: Prova di correttezza

Algoritmo di Prim. [Jarník 1930, Dijkstra 1957, Prim 1959]

- Initializza S = un nodo qualsiasi.
- Applica la proprietà del taglio ad S.
- Aggiungi l'arco di costo minimo nel cutset corrispondente ad S a T, ed aggiungi un nuovo nodo esplorato u ad S.



Algoritmo di Prim: implementazione

Implementazione. Usare Coda a Priorità.

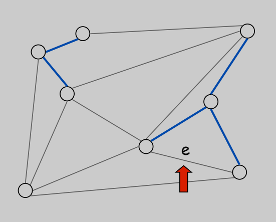
- Mantenere insieme di nodi esplorati S.
- Per ogni nodo non esplorato v, mantenere a[v] = costo minimo di un cammino per v con nodi in S.
- $O(n^2)$ con un array; $O(m \log n)$ con un binary heap.

```
Prim(G, c) {
   foreach (v ∈ V) a[v] ← ∞
   Initialize an empty priority queue Q
   foreach (v ∈ V) insert v onto Q
   Initialize set of explored nodes S ← ϕ

   while (Q is not empty) {
      u ← delete min element from Q
      S ← S ∪ { u }
      foreach (edge e = (u, v) incident to u)
         if ((v ∉ S) and (c_e < a[v]))
            decrease priority a[v] to c_e
}
```

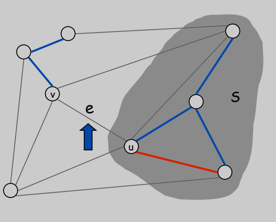
Algoritmo di Kruskal: Prova di correttezza

Algoritmo di Kruskal. [Kruskal, 1956]

- Considera archi in ordine crescente di peso.
- Caso 1: Se l'aggiunta di e a T crea un ciclo, ignora e.
- Caso 2: Altrimenti, inserisci e = (u, v) in T (proprietà del taglio dove S = insieme di nodi nella componente connessa di u).



Case 1



Case 2

Algoritmo di Kruskal: implementazione

Implementazione. Usa la struttura dati union-find.

- Costruisci insieme T di archi nel MST.
- Mantenere insieme per ogni componente connessa.
- $O(m \log n)$ per ordinamento e $O(\underbrace{m\,\alpha(m, n)}_{\text{essenzialmente una costante}})$ per union-find.

$m \le n^2 \Rightarrow \log m$ è $O(\log n)$

```
Kruskal(G, c) {
   Sort edges weights so that c1 ≤ c2 ≤ ... ≤ cm.
   T ← φ

   foreach (u ∈ V) costruisci insieme con solo u

   for i = 1 to m
      (u,v) = ei
      if (u e v sono in insiemi diversi) {
         T ← T ∪ {ei}
         merge insiemi contenenti u e v
      }
   return T
}
```

Sono u e v in componenti connesse diverse?

merge due componenti connesse

Lexicographic Tiebreaking

Per rimuovere l'assunzione che tutti gli archi hanno costi distinti: perturba tutti i costi di piccolissime quantità per evitare lo stesso costo.

Impatto. Kruskal e Prim effettuano solo confronti tra costi. Se le perturbazioni sono molto piccole, l'MST con i costi perturbati è l'MST con i costi originali.

e.g., se tutti i costi fossero interi, perturba il costo di e_i di i / n^2

Implementazione. Possiamo gestire perturbazioni arbitrariamente piccole in modo implicito: lessicograficamente, secondo gli indici.

```
boolean less(i, j) {
   if      (cost(e_i) < cost(e_j)) return true
   else if (cost(e_i) > cost(e_j)) return false
   else if (i < j)                  return true
   else                             return false
}
```

Esercizio 8 pag. 192

Sia G un grafo connesso in cui tutti gli archi hanno costi distinti.
Allora G ha un *unico* MST.

Prova. (per assurdo, usando una argomentazione di scambio)

- Supponiamo che G abbia due spanning tree minimi diversi: T1 e T2.
- Sia e l'arco di minimo costo che si trova in uno dei due spanning tree e non nell'altro. Supponiamo che l'arco e sia in T1 ma non in T2.
- Aggiungere l'arco e a T2 crea un ciclo C.
- Vi deve essere almeno un arco nel ciclo C che non è in T1. (Altrimenti T1 conterrebbe un ciclo). Sia f tale arco.
- $T2' = T2 \cup \{ e \} - \{ f \}$ è ancora uno spanning tree.
- Dato che $c_e < c_f$, $cost(T2') < cost(T2)$.
- Questa è una contraddizione della minimalità di T2.

4.8 Huffman Codes and Data Compression

Codifica simboli usando bit

Codifica di 32 caratteri: 26 lettere, spazio, ed i cinque simboli , . ?!'

5 bit per un totale di 2^5=32 simboli

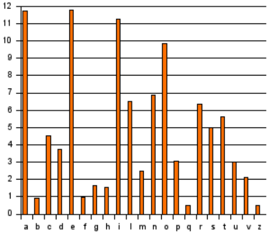
00000 ➔ a

00001 ➔ b

…

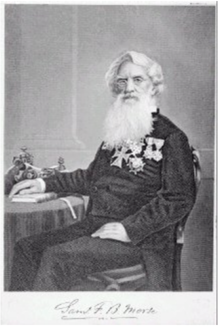
I simboli hanno diverse frequenze di occorrenza

"e" occorre più frequentemente di "q"

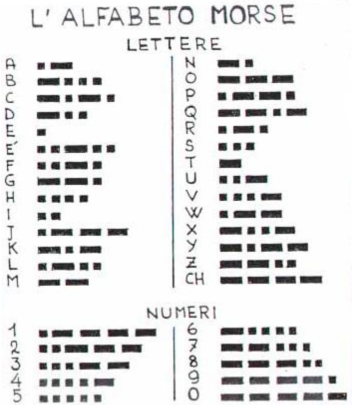


italiano

Codice Morse



Samuel Finley Breese Morse
(Apr. 27, 1791-Apr. 2. 1872)



Compressione Dati

Spesso è importante trovare un modo *efficiente* per rappresentare i dati, cercando di minimizzare il numero di bit usati

- si occupa meno spazio in memoria
- si risparmia sul tempo di trasferimento

Codici

carattere	a	b	c	d	e	f
Frequenza su 100	45	13	12	16	9	5
Lunghezza fissa	000	001	010	011	100	101
Lunghezza variabile	0	100	101	111	1101	1100

Codice a lunghezza fissa

(Tutti i caratteri sono codificati con lo stesso numero di bit)

per 100.000 caratteri servono 300.000 bit

Codice a lunghezza variabile

(I caratteri sono codificati con un numero variabile di bit)

per 100.000 caratteri servono solo

45.000x1 + 13.000x3 + 12.000x3 + 16.000x3 + 9.000x4 + 5.000x4 = 224.000 bit

Codifica e decodifica

carattere	a	b	c	d	e	f
Frequenza su 100	45	13	12	16	9	5
Lunghezza fissa	000	001	010	011	100	101
Lunghezza variabile	0	100	101	111	1101	1100

Codice a lunghezza fissa:

a a c …

000000010…

Codice a lunghezza variabile:

a a c …

00101…

Codifica e decodifica

carattere	a	b	c	d	e	f
Frequenza su 100	45	13	12	16	9	5
Lunghezza fissa	000	001	010	011	100	101
Lunghezza variabile	0	100	101	111	1101	1100

Codice a lunghezza fissa: a a c …

000000010…

Codice a lunghezza variabile: a a c …

00101…

Codice prefisso: nessuna parola codice è prefisso di altra parola codice

Codifica e decodifica

carattere	a	b	c	d	e	f
Frequenza su 100	45	13	12	16	9	5
Lunghezza fissa	000	001	010	011	100	101
Lunghezza variabile	0	100	101	111	1101	1100

Codice a lunghezza fissa:

a a c …

000000010…

Codice a lunghezza variabile:

a a c …

00101…

Codice prefisso: nessuna parola codice è prefisso di altra parola codice

Se avessimo codificato a → 1
Come si decodifica 1101 ?
ac e

Codifica e decodifica

carattere	a	b	c	d	e	f
Frequenza su 100	45	13	12	16	9	5
Lunghezza fissa	000	001	010	011	100	101
Lunghezza variabile	0	100	101	111	1101	1100

Codice a lunghezza variabile:

a a c …

00101…

Codice prefisso: nessuna parola codice è prefisso di altra parola codice

Decodifica codici prefisso: Scorrendo la stringa di bit,

- si leggono bit in sequenza fino a che non troviamo una parola codice,
- scriviamo il carattere corrispondente.

Esercizio decodifica

A	0
B	11
C	100
D	1010
E	1011

11010010010101011

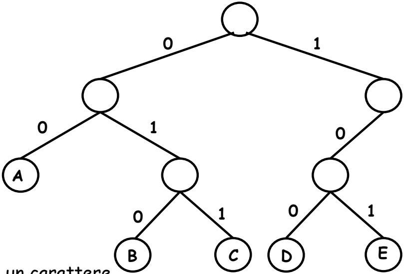
A	00
B	01
C	10
D	110
E	111

100100101010

Alberi binari e codici prefissi

I codici prefisso possono essere rappresentati come alberi binari.

A	00
B	010
C	011
D	100
E	101



- Ogni foglia rappresenta un carattere
- Ogni arco sinistro è etichettato con 0, ed ogni arco destro è etichettato con 1
- Parola codice di ogni foglia è la sequenza di etichette del percorso dalla radice

Alberi binari e codici prefissi

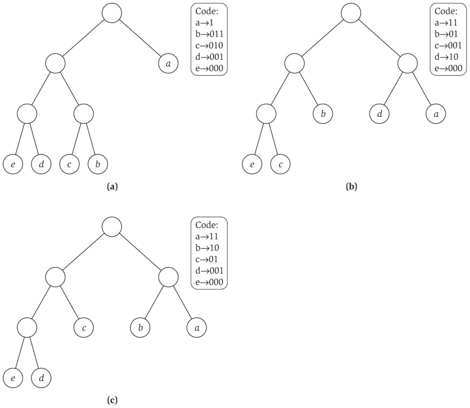


Figure 4.16 Parts (a), (b), and (c) of the figure depict three different prefix codes for the alphabet $S = \{a, b, c, d, e\}$.

Lunghezza codifica

Abbiamo:

- un file composto da caratteri nell'alfabeto S
- un albero T che rappresenta la codifica

Quanti bit occorrono per codificare il file con T ?

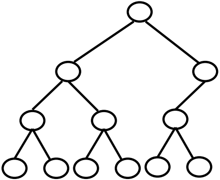
- Per ogni $x \in S$, sia $d_T(x)$ la profondità in T della foglia che rappresenta x
- La parola codice per x è lunga $d_T(x)$ bit
- $f(x)$ è il numero di volte che x occorre nel file.

La dimensione del file codificato è

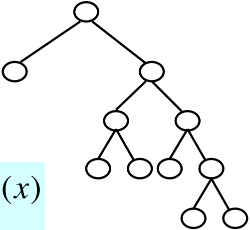
$$\sum_{x \in S} f(x) d_T(x)$$

Codici

carattere	a	b	c	d	e	f
Frequenza su 100	45	13	12	16	9	5
Lunghezza fissa	000	001	010	011	100	101
Lunghezza variabile	0	100	101	111	1101	1100



$$\sum_{x \in S} f(x) d_T(x)$$



45x3 + 13x3 + 12x3 + 16x3 + 9x3 + 5x3

= 300.000 bit

45x1 + 13x3 + 12x3 + 16x3 + 9x4 + 5x4

= 224.000 bit

Problema: trovare il codice prefisso ottimo

Problema del codice prefisso ottimo: Dato un alfabeto S e le frequenze dei caratteri f(x), per x∈S, trovare il codice prefisso T che minimizza

albero

$$\sum_{x \in S} f(x) d_T(x)$$

Problema: trovare il codice prefisso ottimo

Problema del codice prefisso ottimo: Dato un alfabeto S e le frequenze dei caratteri f(x), per x∈S, trovare il codice prefisso T che minimizza

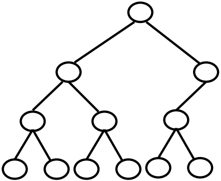
albero

$$\sum_{x \in S} f(x) d_T(x)$$

Vediamo qualche proprietà del codice/albero ottimale

Domanda

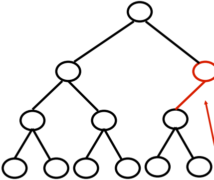
Potrebbe essere ottimo per un alfabeto S con 6 caratteri e frequenze $f(x)>0$, per $x \in S$?

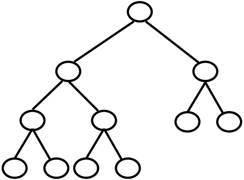


A	000
B	001
C	010
D	011
E	100
F	101

Domanda

Potrebbe essere ottimo per un alfabeto S con 6 caratteri e frequenze f(x)>0, per x∈S ? No!





A	000
B	001
C	010
D	011
E	100
F	101

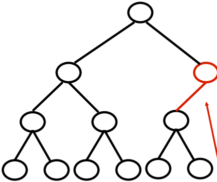
inutile

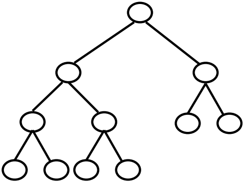
A	000
B	001
C	010
D	011
E	10
F	11

E' migliore!

Proprietà: gli alberi ottimi sono pieni

L'albero binario corrispondente ad un codice ottimo è pieno (full).
Ovvero, ogni nodo interno ha due figli.





A	000
B	001
C	010
D	011
E	100
F	101

inutile

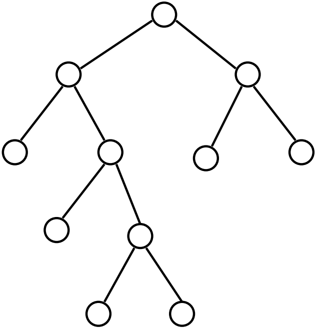
A	000
B	001
C	010
D	011
E	10
F	11

E' migliore!

Domanda

Potrebbe essere ottimo questo codice prefisso?

	frequenza	parola codice
A	25	00
B	25	010
C	6	0110
D	6	0111
E	13	10
F	25	11

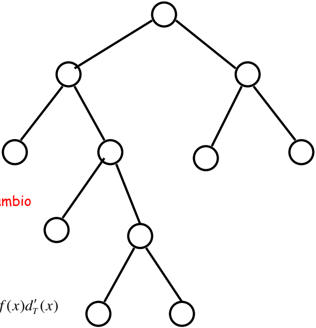


Domanda

Potrebbe essere ottimo questo codice prefisso? No!

	frequenza	parola codice
A	25	00
B	25	010
C	6	0110
D	6	0111
E	13	10
F	25	11

scambio



Guadagno in lunghezza $\sum_{x \in S} f(x) d_T(x) - \sum_{x \in S} f(x) d'_T(x)$

(25x3 + 13x2) – (25x2 + 13x3) = (75+26)-(50+39) = 101-89 = 12

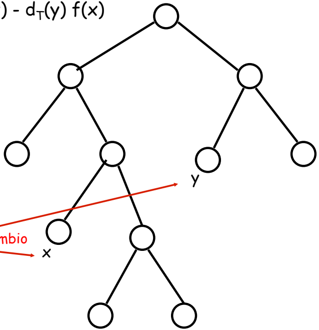
Proprietà: associazione caratteri con parole codice

Teorema. Sia T un albero prefisso ottimale per l'alfabeto S con frequenze $f(x)$, per $x \in S$. Se $d_T(x) > d_T(y)$ allora $f(x) \leq f(y)$.

Prova. (per assurdo) Supponiamo che $f(x) > f(y)$.
Scambiamo le parole codice. Il guadagno in lunghezza è:
$d_T(x)\, f(x) + d_T(y)\, f(y) - d_T(x)\, f(y) - d_T(y)\, f(x)$
$= (d_T(x) - d_T(y))\, (f(x) - f(y)) > 0$

	frequenza	parola codice
A	25	00
x	25	010
C	6	0110
D	6	0111
y	13	10
F	25	11

scambio

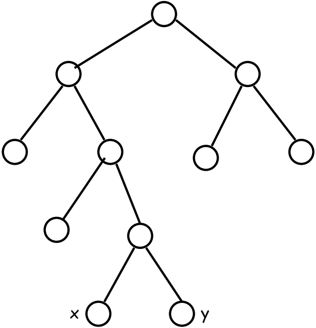


Proprietà: associazione caratteri con parole codice

Teorema. Sia S un alfabeto con frequenze f(x), per $x \in S$. Siano $x,y \in S$ caratteri con le frequenze più basse, $f(x) \leq f(y)$. Allora esiste un codice ottimale con albero T* in cui x,y corrispondono a foglie che sono fratelli.

Esempio

	frequenza	parola codice
A	25	00
B	25	010
x	6	0110
y	6	0111
E	13	10
F	25	11

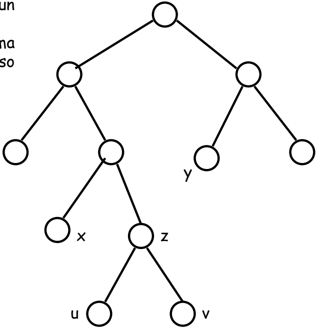


Proprietà: scelta greedy

Teorema. Sia S un alfabeto S con frequenze f(x), per x∈S. Siano x,y∈S caratteri con le frequenze più basse, f(x) ≤ f(y). Allora esiste un codice ottimale con albero T* in cui x,y corrispondono a foglie che sono fratelli.

Prova. Sia T un albero che rappresenta un codice ottimo.
Costruiamo un albero T* ancora ottimo ma con i caratteri x,y foglie figlie dello stesso padre z e a profondità massima in T*.

Assumiamo f(u) ≤ f(v).
Quindi f(x) ≤ f(u) e f(y) ≤ f(v).



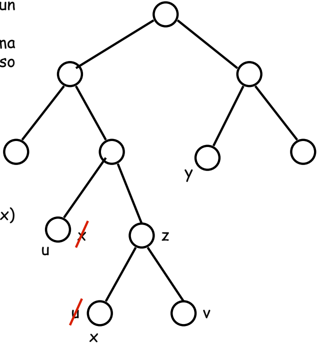
Proprietà: scelta greedy

Teorema. Sia S un alfabeto S con frequenze f(x), per x∈S. Siano x,y∈S caratteri con le frequenze più basse, f(x) ≤ f(y). Allora esiste un codice ottimale con albero T* in cui x,y corrispondono a foglie che sono fratelli.

Prova. Sia T un albero che rappresenta un codice ottimo.
Costruiamo un albero T* ancora ottimo ma con i caratteri x,y foglie figlie dello stesso padre z e a profondità massima in T*.

Assumiamo f(u) ≤ f(v).
Quindi f(x) ≤ f(u) e f(y) ≤ f(v).

Scambiamo x ed u.
Il guadagno in lunghezza è:
$d_T(x)\, f(x) + d_T(u)\, f(u) - d_T(x)\, f(u) - d_T(u)\, f(x)$
$= (d_T(x) - d_T(u))\, (f(x) - f(u)) > 0$



Proprietà: scelta greedy

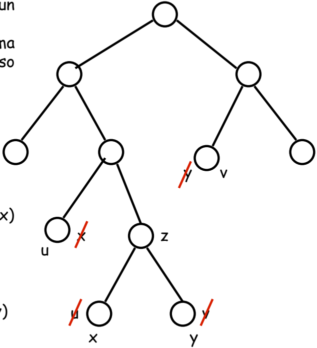
Teorema. Sia S un alfabeto S con frequenze f(x), per $x \in S$. Siano $x,y \in S$ caratteri con le frequenze più basse, $f(x) \leq f(y)$. Allora esiste un codice ottimale con albero T* in cui x,y corrispondono a foglie che sono fratelli.

Prova. Sia T un albero che rappresenta un codice ottimo.
Costruiamo un albero T* ancora ottimo ma con i caratteri x,y foglie figlie dello stesso padre z e a profondità massima in T*.

Assumiamo $f(u) \leq f(v)$.
Quindi $f(x) \leq f(u)$ e $f(y) \leq f(v)$.

Scambiamo x ed u.
Il guadagno in lunghezza è:
$d_T(x)\, f(x) + d_T(u)\, f(u) - d_T(x)\, f(u) - d_T(u)\, f(x)$
$= (d_T(x) - d_T(u))\, (f(x) - f(u)) > 0$

Scambiamo y e v.
Il guadagno in lunghezza è:
$d_T(y)\, f(y) + d_T(v)\, f(v) - d_T(y)\, f(v) - d_T(v)\, f(y)$
$= (d_T(y) - d_T(v))\, (f(y) - f(v)) > 0$



Sottoproblema

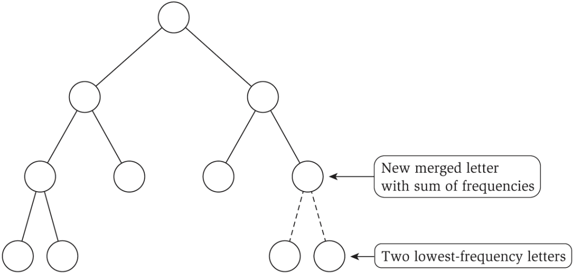


Figure 4.17 There is an optimal solution in which the two lowest-frequency letters label sibling leaves; deleting them and labeling their parent with a new letter having the combined frequency yields an instance with a smaller alphabet.

Esempio

F:5	E:9	C:12	B:13	D:16	A:45

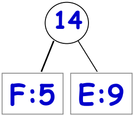
Esempio

C:12	B:13	D:16	A:45

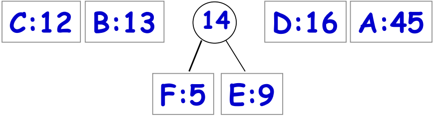
F:5	E:9

Esempio

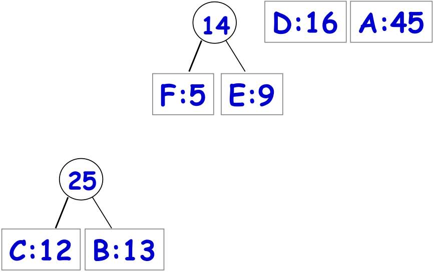
C:12 | B:13 | D:16 | A:45



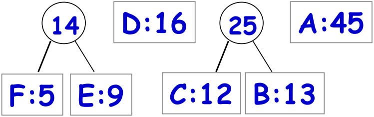
Esempio



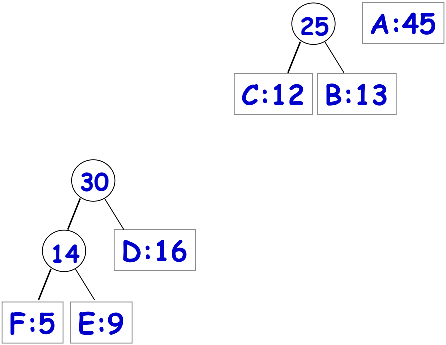
Esempio



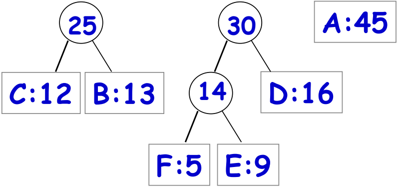
Esempio



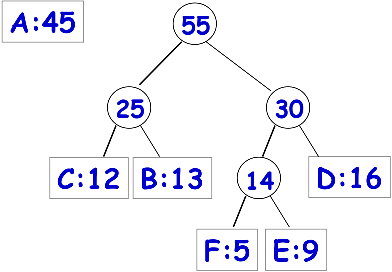
Esempio



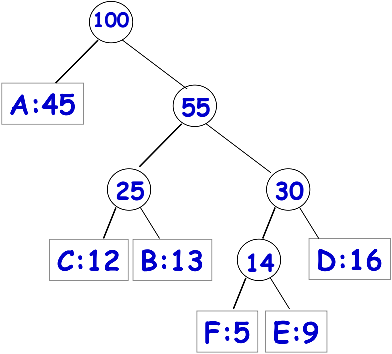
Esempio



Esempio

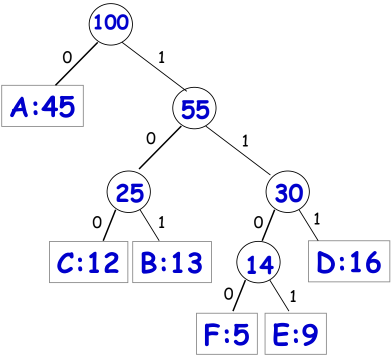


Esempio



Esempio

	frequenza	parola codice
A	45	0
B	13	101
C	12	100
D	16	111
E	9	1100
F	5	1101



Algoritmo di Huffman

```
To construct a prefix code for an alphabet S, with given frequencies:
  If S has two letters then
    Encode one letter using 0 and the other letter using 1
  Else
    Let y* and z* be the two lowest-frequency letters
    Form a new alphabet S' by deleting y* and z* and
      replacing them with a new letter ω of frequency f_{y*} + f_{z*}
    Recursively construct a prefix code γ' for S', with tree T'
    Define a prefix code for S as follows:
      Start with T'
      Take the leaf labeled ω and add two children below it
        labeled y* and z*
   Endif
```

Algoritmo di Huffman: implementazione

```
Huffman(A, n) {
   Inizializza coda a priorità vuota Q
   for i=1 to n {
        insert A[i] in Q
   }
   for i=1 to n-1 {
        x ← estrai elemento minimo da Q
        y ← estrai elemento minimo da Q
        "Crea nuovo albero con radice z e
         frequenza f(x)+f(y) e figli x e y"
        Insert z in Q
   }
}
```

Operazione PQ	Huffman	Array	Binary Heap
Insert	2n-1	n	log n
ExtractMin	2n-1	n	log n
ChangeKey	0	1	log n
IsEmpty	0	1	1
Total		$O(n^2)$	$O(n \log n)$

Codici di Huffman: correttezza

Teorema: L'algoritmo di Huffman produce un codice prefisso ottimo

Prova.

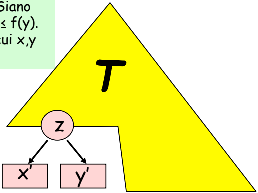
Dato che l'algoritmo è ricorsivo, la prova è per induzione sulla cardinalità dell'alfabeto.

L'asserto è vero per k=2, cioè per un alfabeto con due simboli.

Supponiamo che sia ottimale per tutti gli alfabeti con k-1 simboli.

Mostreremo che è ottimale per gli alfabeti con k simboli.

Ricordiamo: Teorema. Sia S un alfabeto con frequenze f(x). Siano $x,y \in S$ caratteri con le frequenze più basse, $f(x) \leq f(y)$. Allora esiste un codice ottimale con albero T in cui x,y corrispondono a foglie che sono fratelli.



Sia T ' =T U {z} - {x',y'}

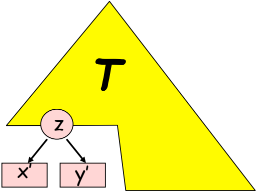
Mostreremo che:

$$\sum_{x \in S} f(x) d_{T'}(x) = \sum_{c \in S} f(x) d_T(x) - f(z)$$

Mostreremo che se T non fosse ottimale allora

non lo sarebbe neanche T '.

Codici di Huffman: sottostruttura ottima



Lemma: Sia T ' =T U {z} - {x',y'}.

$$\sum_{x\in S} f(x)d_{T'}(x) = \sum_{x\in S} f(x)d_T(x) - f(z)$$

Prova.

$$\sum_{x\in S} f(x)d_T(x) = f(x')d_T(x') + f(y')d_T(y') + \sum_{x\neq x',y'} f(x)d_T(x)$$

$$= (f(x') + f(y'))(1 + d_T(z)) + \sum_{x\neq x',y'} f(x)d_{T'}(x)$$

$$= f(z)(1 + d_{T'}(z)) + \sum_{x\neq x',y'} f(x)d_{T'}(x)$$

$$= f(z) + f(z)d_{T'}(z) + \sum_{x\neq x',y'} f(x)d_{T'}(x)$$

$$= f(z) + \sum_{x\in S} f(x)d_{T'}(x)$$

Codici di Huffman: correttezza

Teorema: L'algoritmo di Huffman produce un codice prefisso ottimo

Prova.

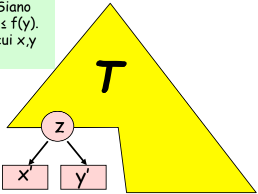
Dato che l'algoritmo è ricorsivo, la prova è per induzione sulla cardinalità dell'alfabeto.

L'asserto è vero per k=2, cioè per un alfabeto con due simboli.

Supponiamo che sia ottimale per tutti gli alfabeti con k-1 simboli.

Mostreremo che è ottimale per gli alfabeti con k simboli.

Ricordiamo: Teorema. Sia S un alfabeto con frequenze f(x). Siano x,y∈S caratteri con le frequenze più basse, f(x) ≤ f(y). Allora esiste un codice ottimale con albero T in cui x,y corrispondono a foglie che sono fratelli.



Sia T ' =T U {z} - {x',y'}

Abbiamo visto che:

$$\sum_{x \in S} f(x) d_{T'}(x) = \sum_{c \in S} f(x) d_T(x) - f(z)$$

Mostreremo che se T non fosse ottimale allora non lo sarebbe neanche T '.

Codici di Huffman: correttezza

Teorema: L'algoritmo di Huffman produce un codice prefisso ottimo

Prova.

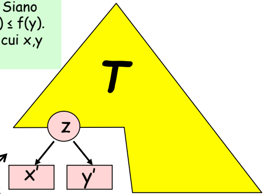
Dato che l'algoritmo è ricorsivo, la prova è per induzione sulla cardinalità dell'alfabeto.

L'asserto è vero per k=2, cioè per un alfabeto con due simboli.

Supponiamo che sia ottimale per tutti gli alfabeti con k-1 simboli.

Mostreremo che è ottimale per gli alfabeti con k simboli.

Ricordiamo: Teorema. Sia S un alfabeto con frequenze f(x). Siano x,y∈S caratteri con le frequenze più basse, f(x) ≤ f(y). Allora esiste un codice ottimale con albero T in cui x,y corrispondono a foglie che sono fratelli.



Sia T ' =T U {z} - {x',y'}

Abbiamo visto che:

$$\sum_{x \in S} f(x) d_{T'}(x) = \sum_{c \in S} f(x) d_T(x) - f(z)$$

Assumiamo T non ottimale. Sia T* ottimale della forma

Allora T*' =T* U {z} - {x',y'} sarebbe migliore di T '. Contraddizione.

Articolo di Huffman

David A. Huffman,

"A method for the construction of minimum-redundancy codes",

Proceedings of the I.R.E., settembre 1952, pagg. 1098-1102

Coin Changing (Cambio monete)

Cambio monete (Cambio monete)

Obiettivo. Dati i valori delle monete U.S.A. : 1, 5, 10, 25, 100, trovare un metodo per pagare un fissato ammontare ad un cliente usando il numero minore di monete.

Esempio: 34¢.

Algoritmo del cassiere. Ad ogni iterazione, aggiungere moneta con il maggior valore che non supera l'ammontare da pagare.

Esempio: $2.89

Cambio monete: Algoritmo greedy

Algoritmo del cassiere. Ad ogni iterazione, aggiungere moneta di maggior valore che non supera l'ammontare da pagare.

```
Sort coins denominations by value: c1 < c2 < … < cn.

  monete selezionate
S ← φ
while (x ≠ 0) {
   sia k più grande intero tale che ck ≤ x
   if (k = 0)
      return "soluzione non trovata"
   x ← x - ck
   S ← S ∪ {k}
}
return S
```

L'algoritmo del cassiere è ottimale?

Cambio monete: Analisi algoritmo greedy

Teorema. Greedy è ottimale per il conio U.S.A. : 1, 5, 10, 25, 100.
Prova. (induzione su x)

- Considera il modo ottimale per $c_k \leq x < c_{k+1}$: greedy sceglie moneta k.
- Ogni soluzione ottimale contiene la moneta k.
  - altrimenti, ci sarebbero monete di tipo $c_1, \ldots, c_{k-1}$ che sommano ad x
  - nessuna soluzione ottimale può farlo, come si vede dalla tabella

k	c_k	Tutte le soluzioni ottimali soddisfano	Massimo valore delle monete 1, 2, …, k-1 in una sol. ottimale
1	1	penny ≤ 4	-
2	5	nickel ≤ 1	4
3	10	nickel + dime ≤ 2	4 + 5 = 9
4	25	quarter ≤ 3	20 + 4 = 24
5	100	Senza limiti	75 + 24 = 99

$5 \leq x < 10$
Al massimo
- 4 penny

$5 \leq x < 10$
Se ci fossero 5 penny potrei migliorare con "1 nickel"

Cambio monete: Analisi algoritmo greedy

Teorema. Greedy è ottimale per il conio U.S.A. : 1, 5, 10, 25, 100.
Prova. (induzione su x)

- Considera il modo ottimale per $c_k \leq x < c_{k+1}$: greedy sceglie moneta k.
- Ogni soluzione ottimale contiene la moneta k.
  - altrimenti, ci sarebbero monete di tipo $c_1, \ldots, c_{k-1}$ che sommano ad x
  - nessuna soluzione ottimale può farlo, come si vede dalla tabella

k	c_k	Tutte le soluzioni ottimali soddisfano	Massimo valore delle monete 1, 2, …, k-1 in una sol. ottimale
1	1	penny ≤ 4	-
2	5	nickel ≤ 1	4
3	10	nickel + dime ≤ 2	4 + 5 = 9
4	25	quarter ≤ 3	20 + 4 = 24
5	100	Senza limiti	75 + 24 = 99

$10 \leq x < 25$
Al massimo
- 4 penny e
- 1 nickel

$10 \leq x < 25$
Se ci fossero 5 penny potrei migliorare con "1 nickel"
Se ci fossero 2 nickel potrei migliorare con "1 dime"

Cambio monete: Analisi algoritmo greedy

Teorema. Greedy è ottimale per il conio U.S.A. : 1, 5, 10, 25, 100.
Prova. (induzione su x)

- Considera il modo ottimale per $c_k \le x < c_{k+1}$: greedy sceglie moneta k.
- Ogni soluzione ottimale contiene la moneta k.
  - altrimenti, ci sarebbero monete di tipo $c_1, \ldots, c_{k-1}$ che sommano ad x
  - nessuna soluzione ottimale può farlo, come si vede dalla tabella

k	c_k	Tutte le soluzioni ottimali soddisfano	Massimo valore delle monete 1, 2, …, k-1 in una sol. ottimale
1	1	penny ≤ 4	-
2	5	nickel ≤ 1	4
3	10	nickel + dime ≤ 2	4 + 5 = 9
4	25	quarter ≤ 3	20 + 4 = 24
5	100	Senza limiti	75 + 24 = 99

$25 \le x < 100$
Al massimo
- 4 penny e
- 2 dime

$25 \le x < 100$
Se ci fossero 3 dime potrei migliorare con "1 quarter + 1 nickel"
Se ci fossero 2 dime e 1 nickel potrei migliorare con "1 quarter"
Se ci fossero 1 dime e 1 nickel avrebbe valore inferiore a "2 dime"

nickel + dime ≤ 2

Cambio monete: Analisi algoritmo greedy

Teorema. Greedy è ottimale per il conio U.S.A. : 1, 5, 10, 25, 100.

Prova. (induzione su x)

- Considera il modo ottimale per $c_k \le x < c_{k+1}$: greedy sceglie moneta k.
- Ogni soluzione ottimale contiene la moneta k.
  - altrimenti, ci sarebbero monete di tipo $c_1, \ldots, c_{k-1}$ che sommano ad x
  - nessuna soluzione ottimale può farlo, come si vede dalla tabella

k	c_k	Tutte le soluzioni ottimali soddisfano	Massimo valore delle monete 1, 2, …, k-1 in una sol. ottimale
1	1	penny ≤ 4	-
2	5	nickel ≤ 1	4
3	10	nickel + dime ≤ 2	4 + 5 = 9
4	25	quarter ≤ 3	20 + 4 = 24
5	100	Senza limiti	75 + 24 = 99

100 ≤ x
Al massimo
- 3 quarter

100 ≤ x
Se ci fossero 4 quarter potrei migliorare con "1 dollaro"

Cambio monete: Analisi algoritmo greedy

Teorema. Greedy è ottimale per il conio U.S.A. : 1, 5, 10, 25, 100.

Prova. (induzione su x)

- Considera il modo ottimale per $c_k \leq x < c_{k+1}$: greedy sceglie moneta k.
- Ogni soluzione ottimale contiene la moneta k.
  - altrimenti, ci sarebbero monete di tipo $c_1, \ldots, c_{k-1}$ che sommano ad x
  - nessuna soluzione ottimale può farlo, come si vede dalla tabella

k	c_k	Tutte le soluzioni ottimali soddisfano	Massimo valore delle monete 1, 2, …, k-1 in una sol. ottimale
1	1	penny ≤ 4	-
2	5	nickel ≤ 1	4
3	10	nickel + dime ≤ 2	4 + 5 = 9
4	25	quarter ≤ 3	20 + 4 = 24
5	100	Senza limiti	75 + 24 = 99

- Riduzione al cambio di $x - c_k$ centesimi, che per induzione, è risolto in modo ottimale dall'algoritmo greedy. ▪

Cambio monete: Analisi algoritmo greedy

Osservazione. L'algoritmo greedy non è ottimo per i valori dei francobolli americani: 1, 10, 21, 34, 70, 100, 350, 1225, 1500.

Controesempio. 140¢.
- Greedy: 100, 34, 1, 1, 1, 1, 1, 1.
- Ottimo: 70, 70.

Cambio monete: Esercizio

Conio euro: 1, 2, 5, 10, 20, 50, 100, 200.

Greedy è ottimale?

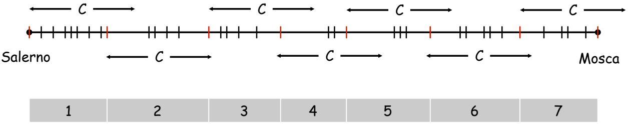
Selecting Breakpoints

Selecting Breakpoints

Selecting breakpoints (scelta fermate).

- Viaggio in auto da Salerno a Mosca per una percorso fissato.
- Distributori di benzina a punti fissi lungo il percorso.
- Capacità serbatoio = C.
- Obiettivo: minimizzare numero di fermate per rifornimento benzina.

Algoritmo Greedy. Viaggiare il più possibile prima del rifornimento.



Selecting Breakpoints: Algoritmo Greedy

Algoritmo greedy.

```
Sort breakpoints so that: 0 = b0 < b1 < b2 < ... < bn = L

S ← {0}  ←— breakpoints selected
x ← 0    ←— current location

while (x ≠ bn)
   let p be largest integer such that bp ≤ x + C
   if (bp = x)
      return "no solution"
   x ← bp
   S ← S ∪ {p}
return S
```

Implementazione. $O(n \log n)$

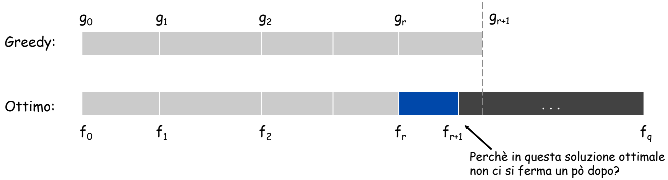
- Usare ricerca binaria per scegliere ogni breakpoint p.

Selecting Breakpoints: Correttezza

Teorema. L'algoritmo greedy è ottimale.

Prova. (per assurdo)

- Assumiamo che l'algoritmo greedy non sia ottimale.
- Sia $0 = g_0 < g_1 < \ldots < g_p = L$ insieme di breakpoints scelto da greedy.
- Sia $0 = f_0 < f_1 < \ldots < f_q = L$ insieme di breakpoints nella soluzione ottimale con $f_0 = g_0$, $f_1 = g_1, \ldots, f_r = g_r$ per il massimo valore possibile per r.
- Nota: $g_{r+1} > f_{r+1}$ per la scelta greedy dell'algoritmo.

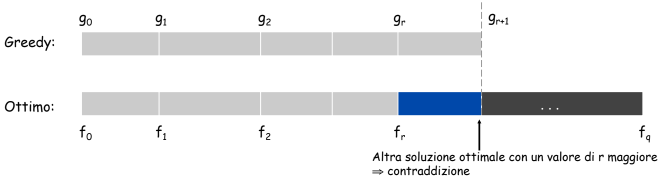


Selecting Breakpoints: Correttezza

Teorema. L'algoritmo greedy è ottimale.

Prova. (per assurdo)

- Assumiamo che l'algoritmo greedy non sia ottimale.
- Sia $0 = g_0 < g_1 < \ldots < g_p = L$ insieme di breakpoints scelto da greedy.
- Sia $0 = f_0 < f_1 < \ldots < f_q = L$ insieme di breakpoints nella soluzione ottimale con $f_0 = g_0$, $f_1 = g_1, \ldots, f_r = g_r$ per il massimo valore possibile per r.
- Nota: $g_{r+1} > f_{r+1}$ per la scelta greedy dell'algoritmo.



Fractional Knapsack problem

Fractional Knapsack problem

Fractional Knapsack problem (Problema dello zaino frazionato).

- n oggetti, ognuno con peso $w_i > 0$ e valore $p_i > 0$
- Zaino con capacità M
- Obiettivo: Riempire lo zaino con gli oggetti (presi anche in parte) massimizzando il valore totale

Massimizzare $\sum_{1 \le i \le n} p_i x_i$

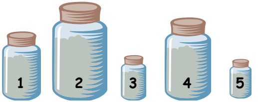
Vincoli $\sum_{1 \le i \le n} w_i x_i \le M$ con $0 \le x_i \le 1$, $1 \le i \le n$

parte frazionaria presa

Fractional Knapsack problem

Fractional Knapsack problem (Problema dello zaino frazionato).

- n oggetti, ognuno con peso $w_i > 0$ e valore $p_i > 0$
- Zaino con capacità M
- Obiettivo: Riempire lo zaino con gli oggetti (presi anche in parte) massimizzando il valore totale



oggetti	1	2	3	4	5
peso:	4 ml	8 ml	2 ml	6 ml	1 ml
valore:	€12	€32	€40	€30	€50

"knapsack"

10 ml

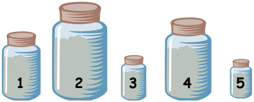
Fractional Knapsack problem

Fractional Knapsack problem (Problema dello zaino frazionato).

- n oggetti, ognuno con peso $w_i > 0$ e valore $p_i > 0$
- Zaino con capacità M
- Obiettivo: Riempire lo zaino con gli oggetti (presi anche in parte) massimizzando il valore totale

Scelta greedy: selezionare l'oggetto con massimo valore relativo e prenderne la maggior quantità possibile

oggetti



	1	2	3	4	5
peso:	4 ml	8 ml	2 ml	6 ml	1 ml
valore:	€12	€32	€40	€30	€50

"knapsack"

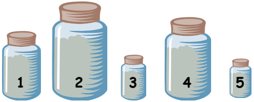
10 ml

Fractional Knapsack problem (Problema dello zaino frazionato).

- n oggetti, ognuno con peso $w_i > 0$ e valore $p_i > 0$
- Zaino con capacità M
- Obiettivo: Riempire lo zaino con gli oggetti (presi anche in parte) massimizzando il valore totale

Scelta greedy: selezionare l'oggetto con massimo valore relativo e prenderne la maggior quantità possibile

oggetti



peso:	4 ml	8 ml	2 ml	6 ml	1 ml
valore:	€12	€32	€40	€30	€50
valore: (€ per ml)	3	4	20	5	50

10 ml

"knapsack"

Soluzione

- 1 ml di 5 $x_5=1$
- 2 ml di 3 $x_3=1$
- 6 ml di 4 $x_4=1$
- 1 ml di 2 $x_2=1/8$
- 0 ml di 1 $x_1=0$

Valore totale = € 50+40+30+32/8
= € 124

Fractional Knapsack problem

Algoritmo del cassiere. Ad ogni iterazione, aggiungere la massima parte dell'oggetto di maggior valore relativo senza superare W.

```
Sort oggetti per valore relativo: p₁/w₁ < p₂/w₂ < … < pₙ/wₙ.

  Peso degli oggetti selezionati
w ← 0
x₁ ← 0; x₂← 0; …; xₙ← 0
i ← 1
while (w < W) and (i < n){
   if (wᵢ < W - w)
      then xᵢ ← 1
           w ← w + wᵢ
      else xᵢ ← (W – w) / wᵢ
           w ← W
   i ← i + 1
}
return (x₁, x₂ …, xₙ)
```

Running time O(n log n).

Provare che l'algoritmo è ottimale (quale tecnica usare?)

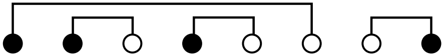
Connecting wires

Connecting wires (Fili di connessione)

- Ci sono n punti bianchi ed n punti neri su una linea, equispaziati (cioè la distanza tra 2 punti consecutivi è sempre la stessa)

- Vogliamo connettere ogni punto bianco con uno nero minimizzando la lunghezza totale di fili

- Esempio:

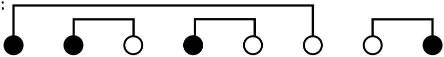


Lunghezza totale 1 + 1 + 1 + 5 = 8

- Trovare un algoritmo greedy per risolvere il problema, analizzando le varie scelte greedy.
- Per ogni scelta greedy individuata, verificare se
  - è corretta (prova)
  - non è corretta (mostrare un controesempio)

Connecting wires (Fili di connessione)

- Ci sono n punti bianchi ed n punti neri su una linea, equispaziati (cioè la distanza tra 2 punti consecutivi è sempre la stessa)
- Vogliamo connettere ogni punto bianco con uno nero minimizzando la lunghezza totale di fili
- Esempio:

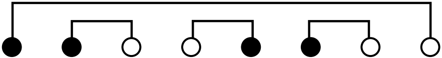


Lunghezza totale 1 + 1 + 1 + 5 = 8

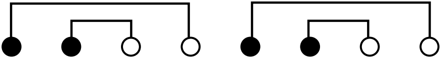
- Un esempio di scelta greedy:
  - Connettere tutte le coppie a distanza 1
  - Connettere tutte le coppie rimanenti a distanza 2
  - ...

Connecting wires (Fili di connessione)

Controesempio:



Lunghezza totale 1 + 1 + 1 + 7 = 10



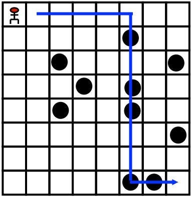
Lunghezza totale 1 + 3 + 1 + 3 = 8

- Un esempio di scelta greedy:
  - Connettere tutte le coppie a distanza 1
  - Connettere tutte le coppie rimanenti a distanza 2
  - ...

Collecting coins

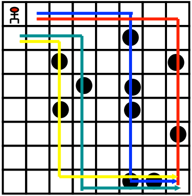
Collecting coins

- Su una scacchiera ci sono alcune monete.
- Un robot parte dall'angolo in alto a sinistra e cammina fino all'angolo in basso a destra
- Il robot può muovere solo in 2 direzioni: destra ed in giù
- Il robot colleziona monete al suo passaggio
- Problema: collezionare tutte le monete usando il numero minimo di robot.
- Esempio:



Collecting coins

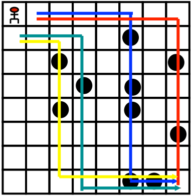
- Su una scacchiera ci sono alcune monete.
- Un robot parte dall'angolo in alto a sinistra e cammina fino all'angolo in basso a destra
- Il robot può muovere solo in 2 direzioni: destra ed in giù
- Il robot colleziona monete al suo passaggio
- Problema: collezionare tutte le monete usando il numero minimo di robot.
- Esempio:



4 robot sono sufficienti a collezionare tutte le 10 monete

Collecting coins

- Su una scacchiera ci sono alcune monete.
- Un robot parte dall'angolo in alto a sinistra e cammina fino all'angolo in basso a destra
- Il robot può muovere solo in 2 direzioni: destra ed in giù
- Il robot colleziona monete al suo passaggio
- Problema: collezionare tutte le monete usando il numero minimo di robot.
- Esempio:

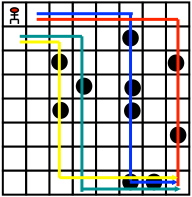


Trovare un algoritmo greedy per risolvere il problema, analizzando le varie scelte greedy.

Per ogni scelta greedy individuata, verificare se
- è corretta (prova)
- non è corretta (mostrare un controesempio)

Collecting coins

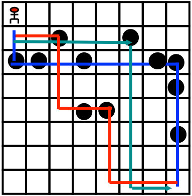
- Su una scacchiera ci sono alcune monete.
- Un robot parte dall'angolo in alto a sinistra e cammina fino all'angolo in basso a destra
- Il robot può muovere solo in 2 direzioni: destra ed in giù
- Il robot colleziona monete al suo passaggio
- Problema: collezionare tutte le monete usando il numero minimo di robot.
- Esempio:



Un esempio di scelta greedy:

- Il primo robot prende il maggior numero di monete
- Il secondo robot prende il maggior numero delle restanti monete
- …

- Controesempio:



Un esempio di scelta greedy:

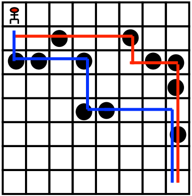
- Il primo robot prende il maggior numero di monete
- Il secondo robot prende il maggior numero delle restanti monete
- …

Scelta greedy: 3 robot

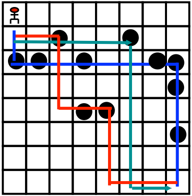
- Il primo prende 7 monete
- Il secondo prende 3 monete
- Il terzo prende l'ultima moneta

Collecting coins

- Controesempio:



Ottimo: 2 robot



Un esempio di scelta greedy:

- Il primo robot prende il maggior numero di monete
- Il secondo robot prende il maggior numero delle restanti monete
- …

Scelta greedy: 3 robot

- Il primo prende 7 monete
- Il secondo 3 monete
- Il terzo l'ultima moneta

Riepilogo Capitolo 4, Greedy Algorithms

4.1 Interval Scheduling

4.1 Interval Partitioning

4.2 Scheduling to Minimize Lateness

4.3 Optimal Caching (senza prova correttezza)

4.4 Shortest Paths in a Graph

4.5 Minimum Spanning Tree

4.6 The Union-Find Data Structure (no)

4.7 Clustering (no)

4.8 Huffman Codes and Data Compression

4.9 Minimum-Cost Arborescences (no)

Esercizi:

- Coin Changing
- Selecting Breakpoints
- Fractional Knapsack problem
- Connecting wires
- Collecting coins