Algoritmi greedy
parte II

Progettazione di Algoritmi a.a. 2023-24
Matricole congrue a 1
Docente: Annalisa De Bonis

Algoritmo di Dijkstra: analisi tempo di esecuzione

```
Dijkstra's Algorithm (G, ℓ, s)
Let S be the set of explored nodes
    For each u ∈ S, we store a distance d(u)
Initially S = {s} and d(s) = 0
While S ≠ V
    Select a node v ∉ S with at least one edge from S for which
       d'(v) = min_{e=(u,v):u∈S} d(u) + ℓ_e is as small as possible
    Add v to S and define d(v) = d'(v)
EndWhile
```

- While iterato n volte
- Se non usiamo nessuna struttura dati per trovare in modo efficiente il minimo d'(v) il calcolo del minimo richiede di scandire tutti gli archi che congiungono un vertice in S con un vertice non in S → O(m) ad ogni iterazione del while → O(nm) in totale.

Algoritmo di Dijkstra: come renderlo più efficiente?

```
Dijkstra's Algorithm (G, ℓ , s)
Let S be the set of explored nodes
    For each u ∈ S, we store a distance d(u)
Initially S = {s} and d(s) = 0
While S ≠ V
    Select a node v ∉ S with at least one edge from S for which
        d'(v) = min_{e=(u,v):u∈S} d(u) + ℓ_e is as small as possible
    Add v to S and define d(v) = d'(v)
EndWhile
```

Miglioramenti:

1. per ogni $x \notin S$, teniamo traccia dell'ultimo valore computato d'(x) (se non è mai stato calcolato poniamo d'(x)= ∞) e
2. ad ogni iterazione del while: per ogni $z \notin S$, ricomputiamo il valore d'(z) solo se è stato appena aggiunto ad S un nodo u per cui esiste l'arco (u,z)
   - per calcolare il nuovo valore di d'(z) basta calcolare min{d'(z), d(u)+ℓ_e }, dove e=(u,z) è l'arco che congiunge z al nodo u appena introdotto in S. Possiamo farlo perchè teniamo traccia dell'ultimo valore precedentemente computato d'(z).
3. Se per ogni $x \notin S$, memorizziamo (d'(x),x) in una coda a priorità, d'(v)=$\min_{x \notin S}$\{d'(x)\} può essere ottenuto invocando la funzione Min o direttamente ExtractMin che va anche a rimuovere l'entrata (d'(v),v). L'aggiornamento di d'(z) al punto 1 può essere fatto con ChangeKey.

PROGETTAZIONE DI ALGORITMI A.A. 2023-24
A. De Bonis

Algoritmo di Dijkstra con coda a priorità: analisi del tempo di esecuzione

```
Dijkstra's Algorithm (G,ℓ,s)
Let S be the set of explored nodes
For each u not in S, we store a distance d'(u)
Let Q be a priority queue of pairs (d'(u),u) s.t. u is not in S
For each u ∈ S, we store a distance d(u)
Insert(Q,(0,s))
For each u≠s insert (Q, ∞,u)  in Q EndFor
While  S≠ V
 (d(v),v)<-- ExtractMin(Q)
  Add v to S
  For each edge e=(v,z)
   If  z not in S && d(v)+ ℓe< d'(z)
     ChangeKey(Q,z, d(v)+ ℓe)

EndWhile
```

In una singola iterazione del while, il for è iterato un numero di volte pari al numero di archi uscenti da u.
Se consideriamo tutte le iterazioni del while, il for viene iterato in totale O(m) volte

Algoritmo di Dijkstra con coda a priorità: analisi del tempo di esecuzione

- Se usiamo una min priority queue che per ogni vertice v **non** in S contiene la coppia (d'[u],v) allora con un'operazione di ExtractMin possiamo ottenere il vertice v con il valore d'[v] più piccolo possibile

- Tempo inizializzazione: tempo per effettuare gli n inserimenti in Q

- Tempo While: O(n) se escludiamo il tempo per fare le n ExtractMin e il tempo del for. Il for esegue in totale O(m) iterazioni (si veda slide precedente). In al piu` m di queste viene eseguita una changeKey mentre ciascuna delle le altre ha tempo costante.

  Se la coda è implementata mediante una lista o con un array non ordinato:
  Inizializzazione: O(n)
  While: $O(n^2)$ per le n ExtractMin; O(m) per le O(m) ChangeKey
  Tempo algoritmo: $O(n^2)$

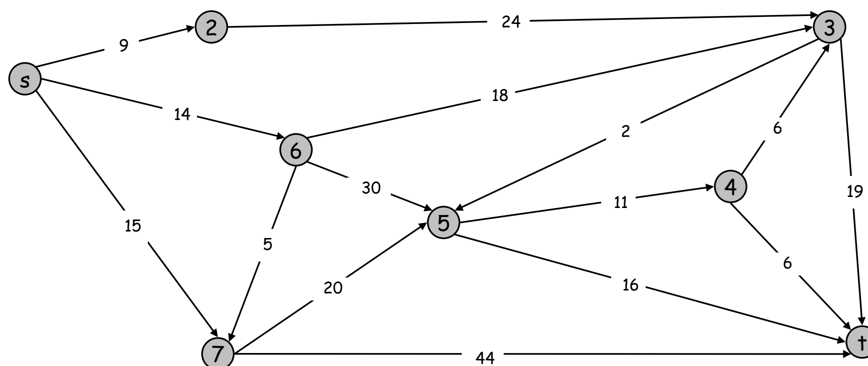
  Se la coda è implementata mediante un heap binario:
  Inizializzazione: O(n) con costruzione bottom up oppure O(n log n) con n inserimenti
  While: O(n log n) per le n ExtractMin; O(m log n) per le m ChangeKey
  Tempo algoritmo: O(n log n+ m log n)

25

25

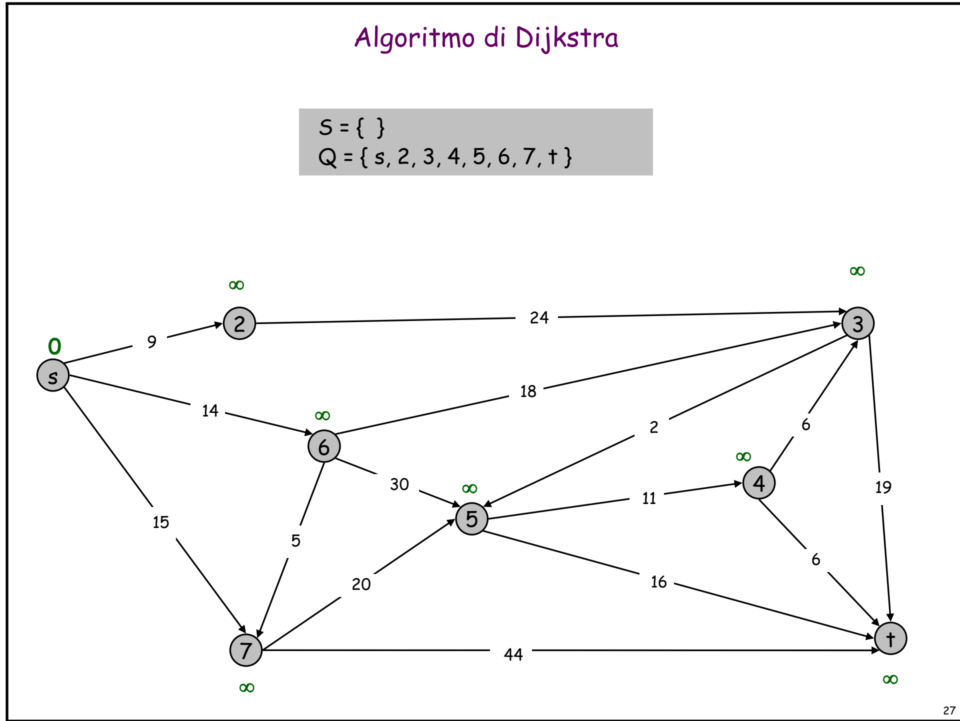
Algoritmo di Dijkstra

Trova i percorsi più corti

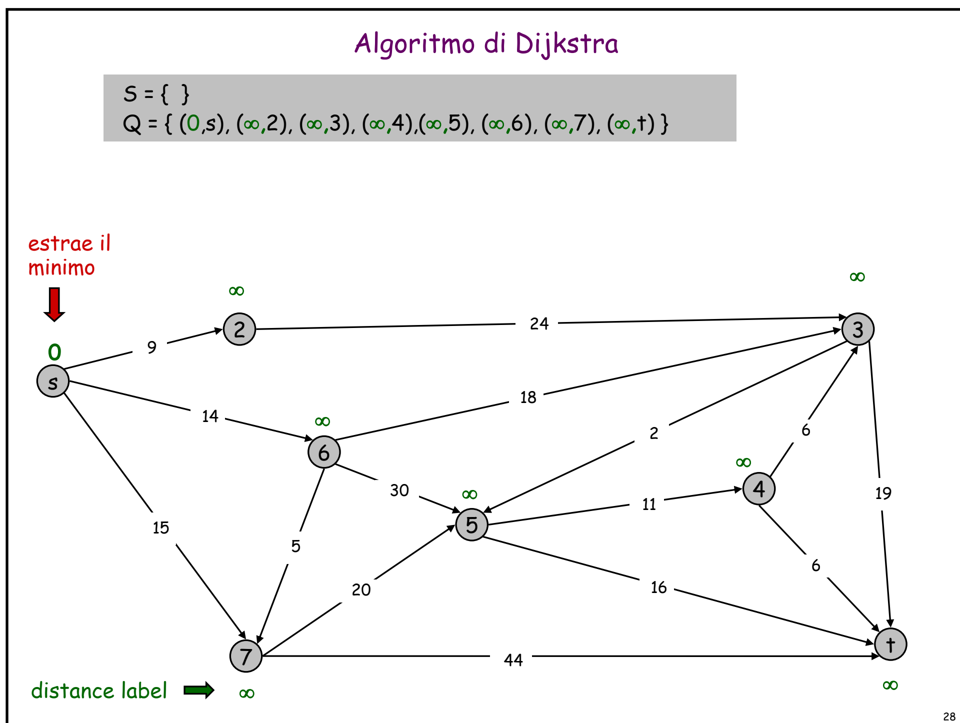


26

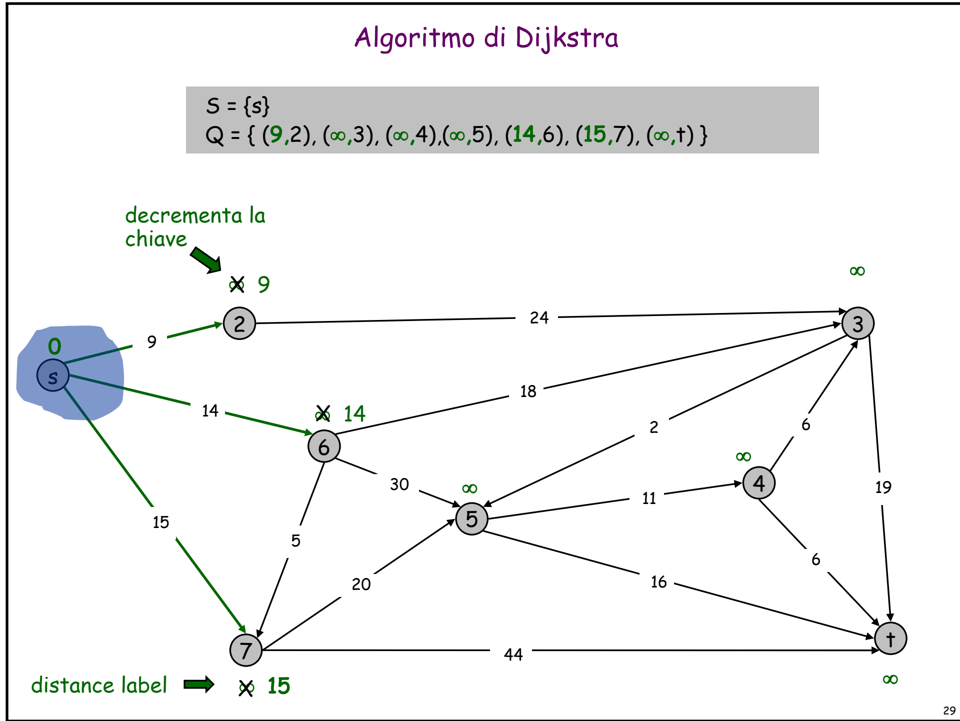
26



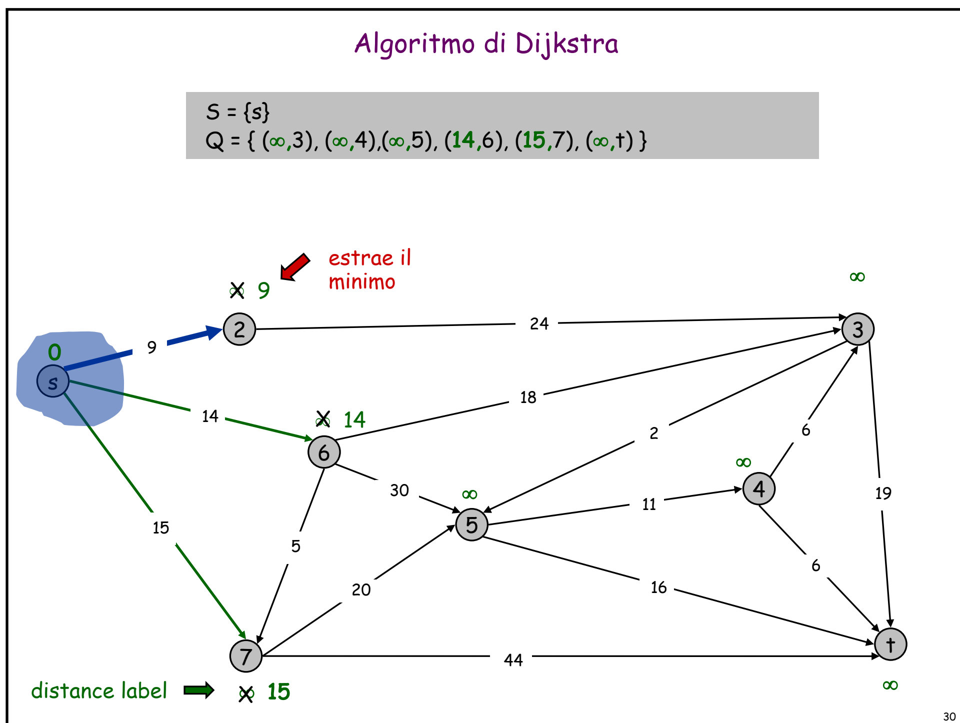
27



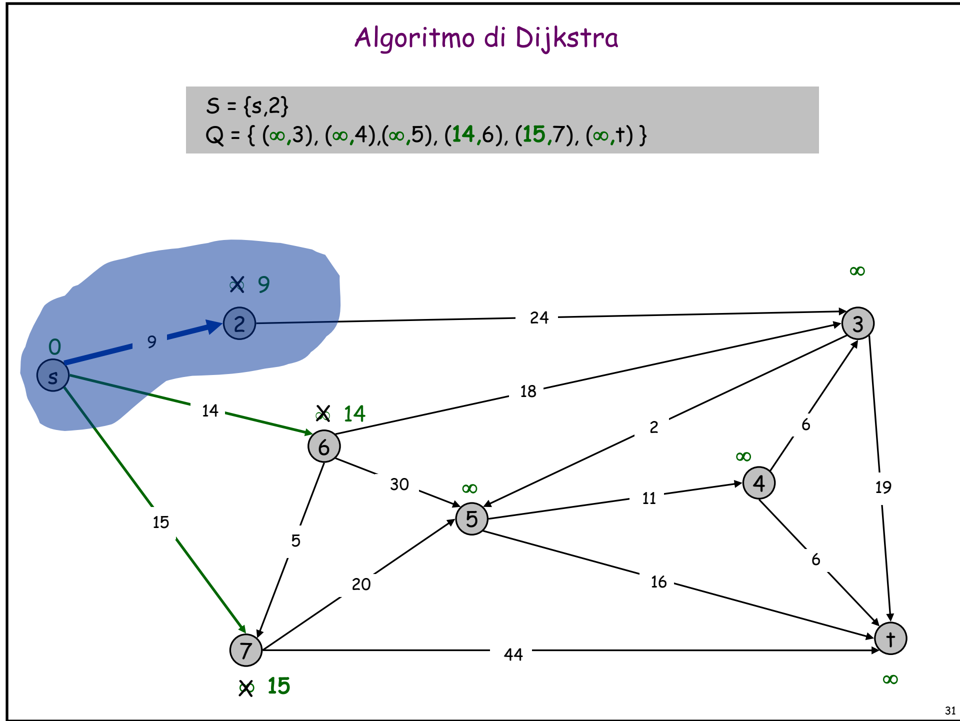
28



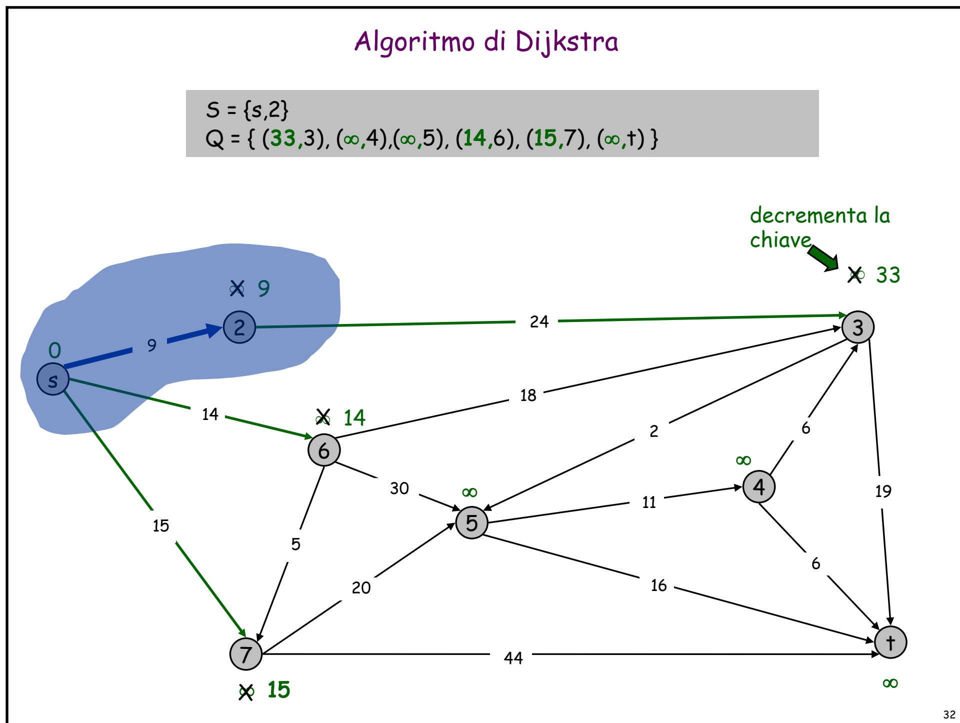
29



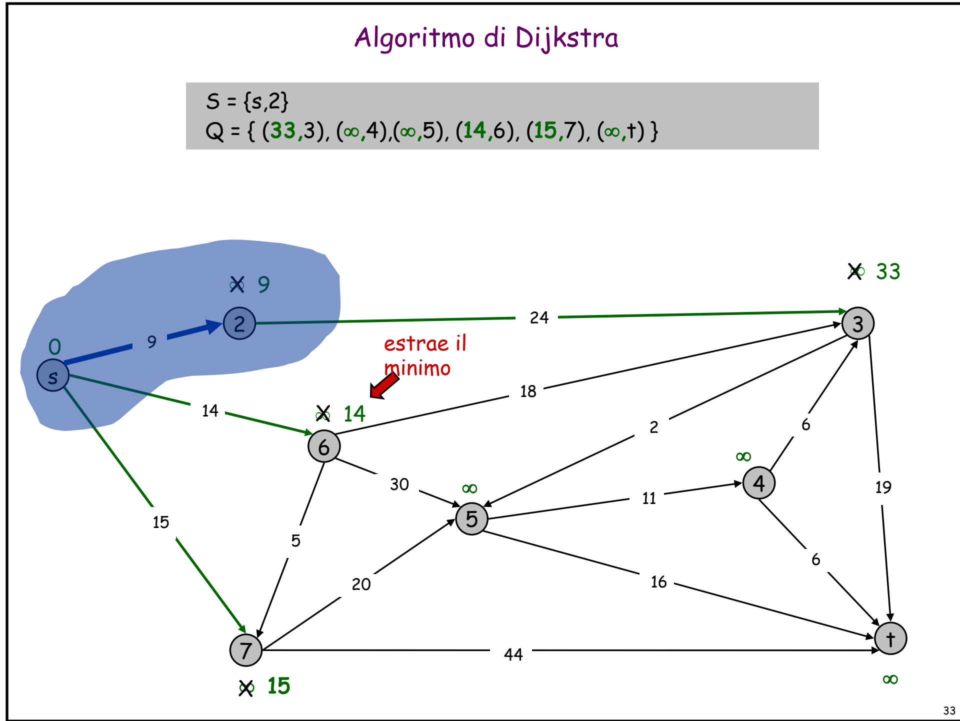
30



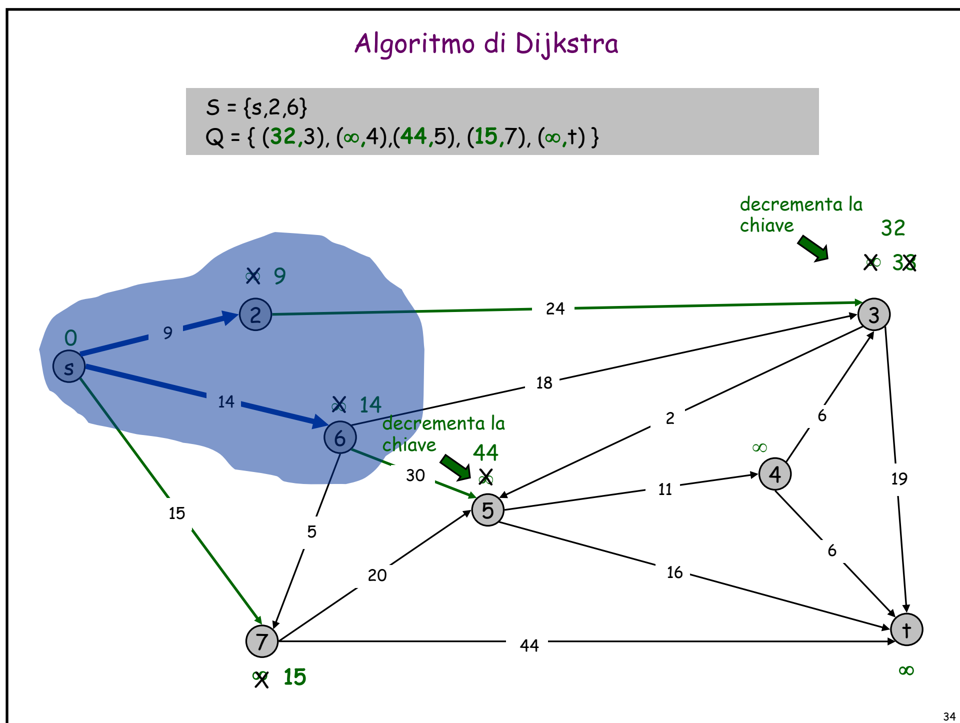
31



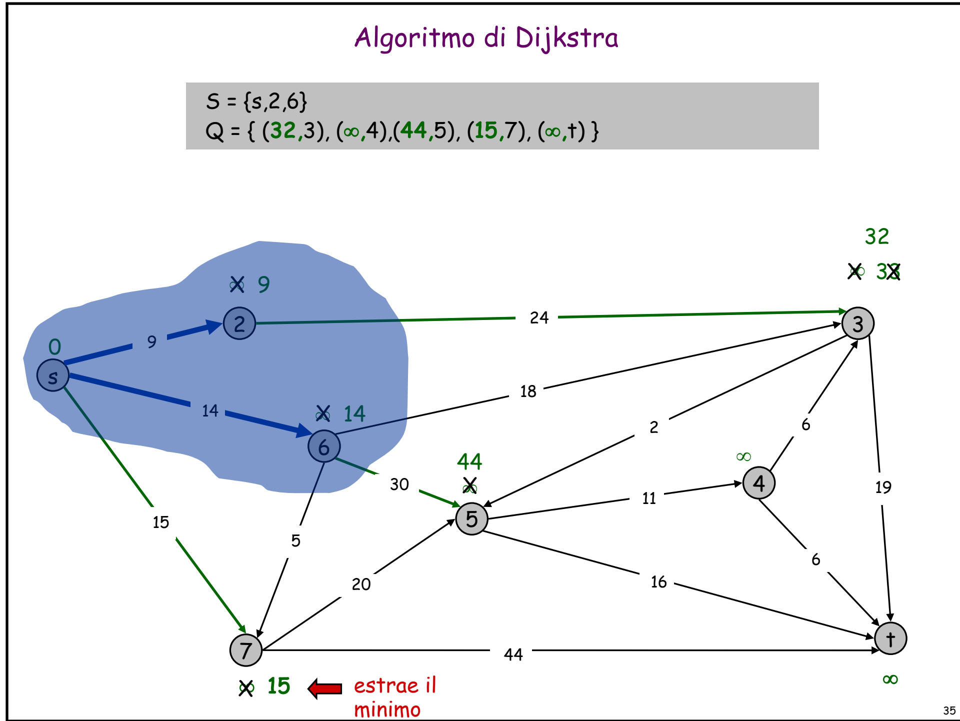
32



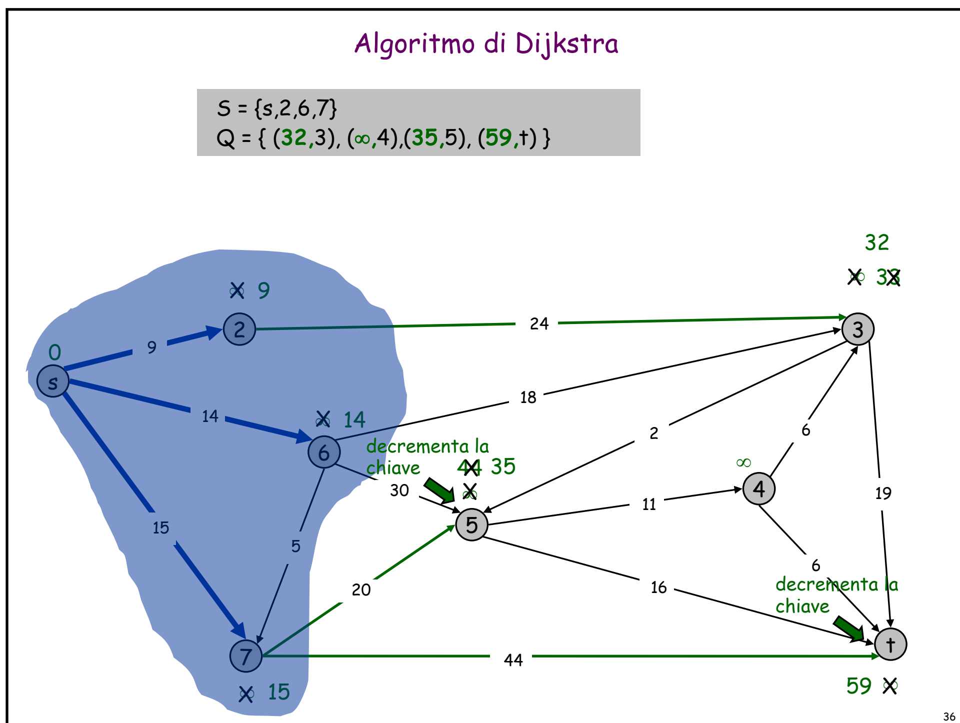
33



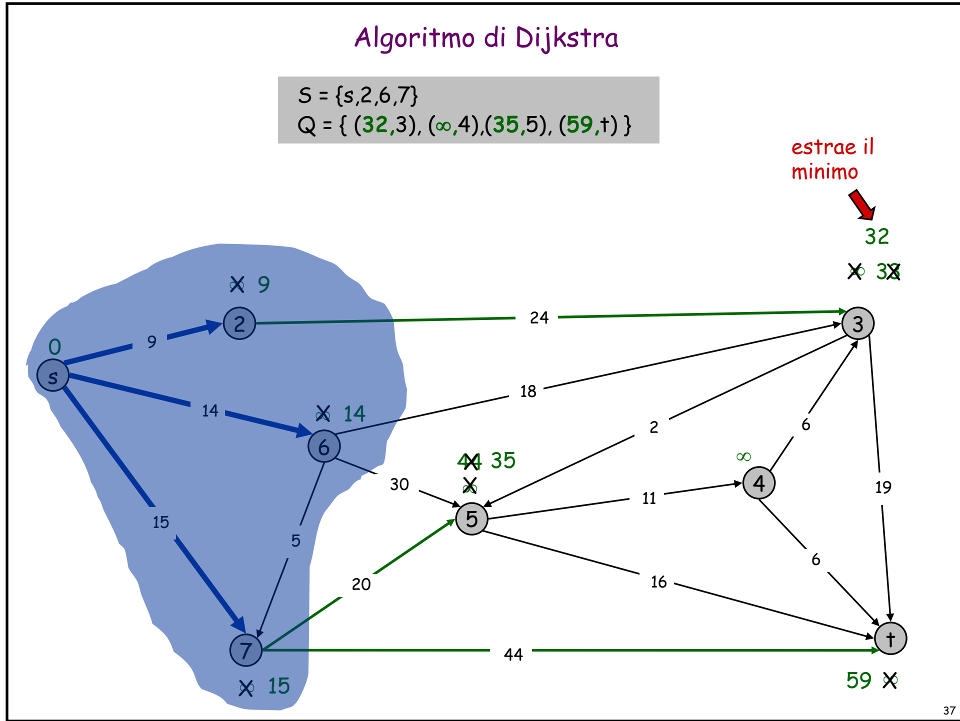
34



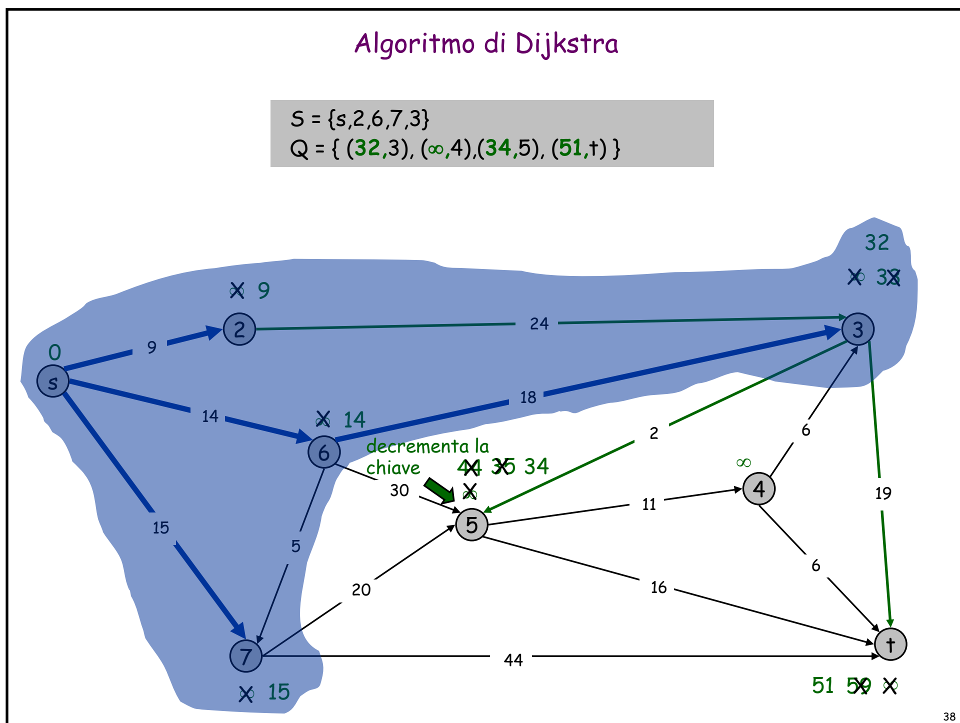
35



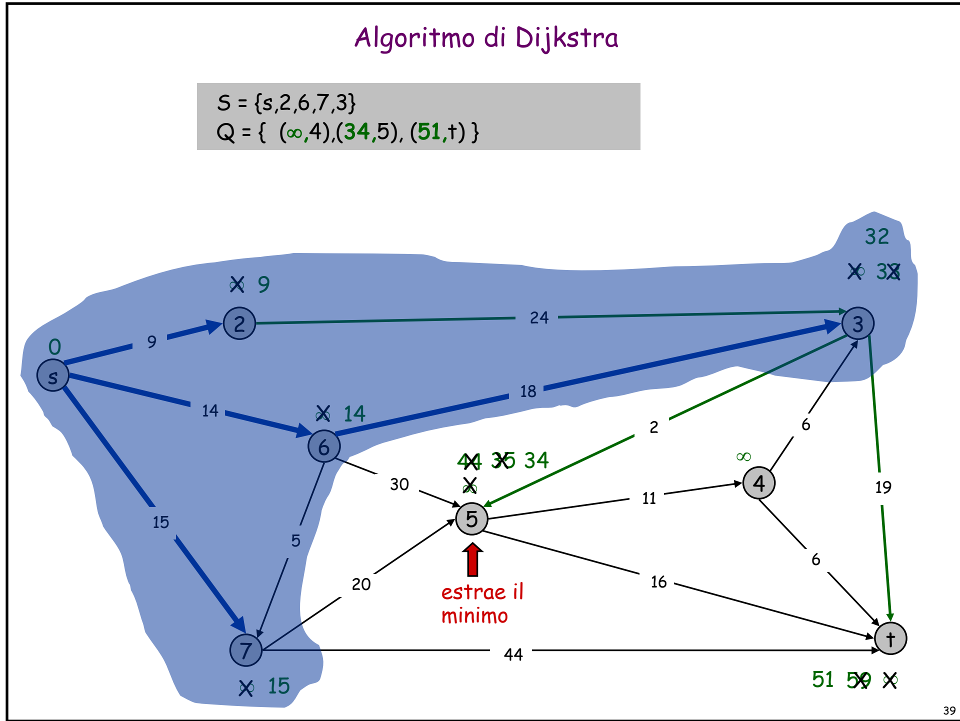
36



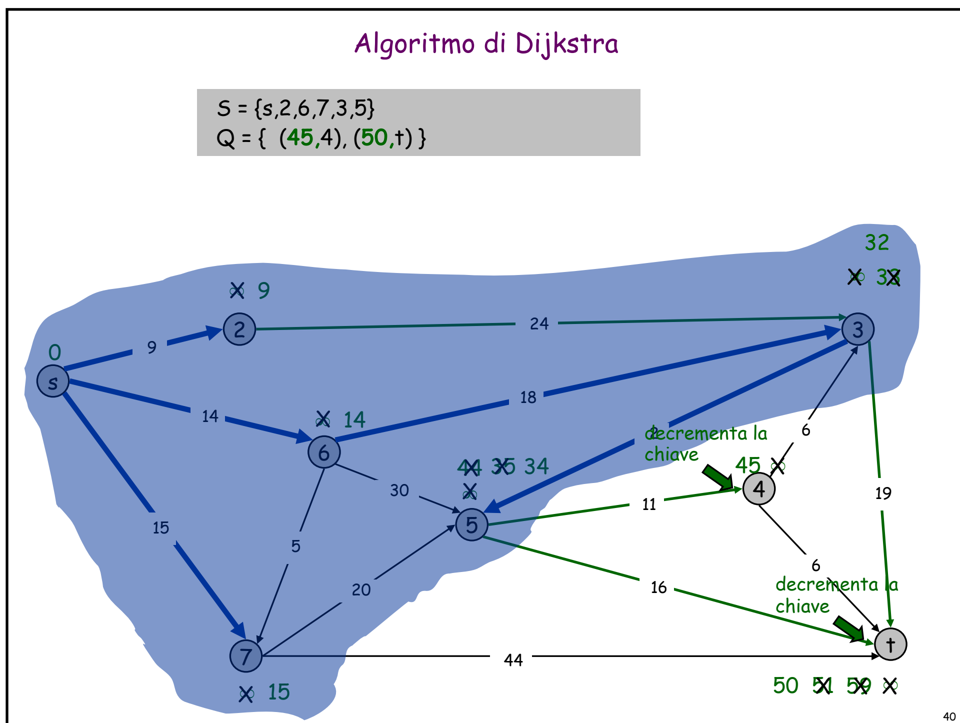
37



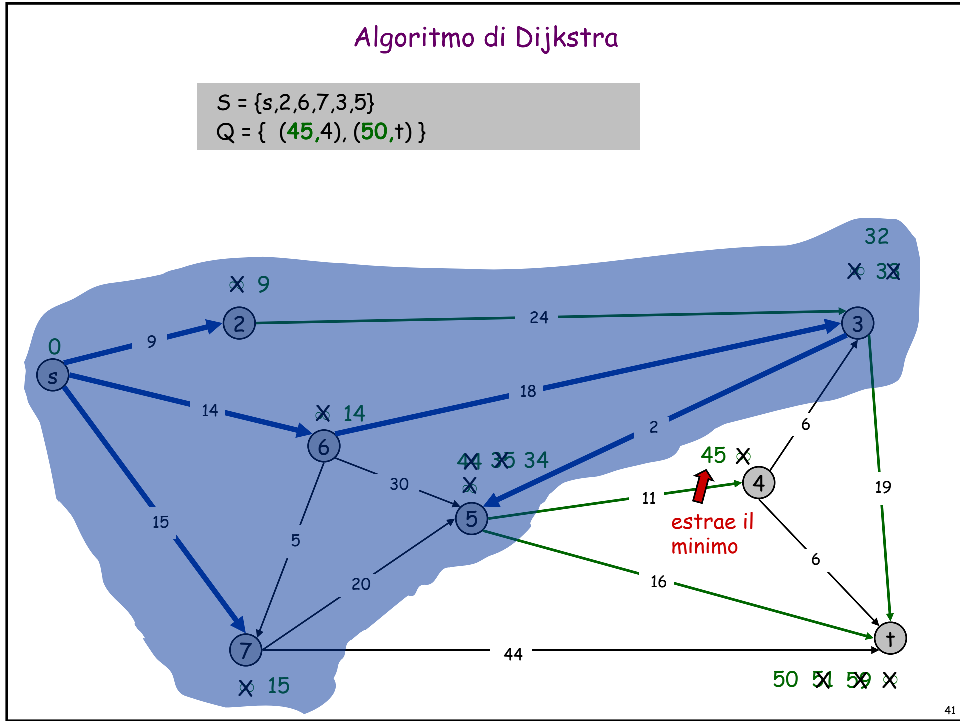
38



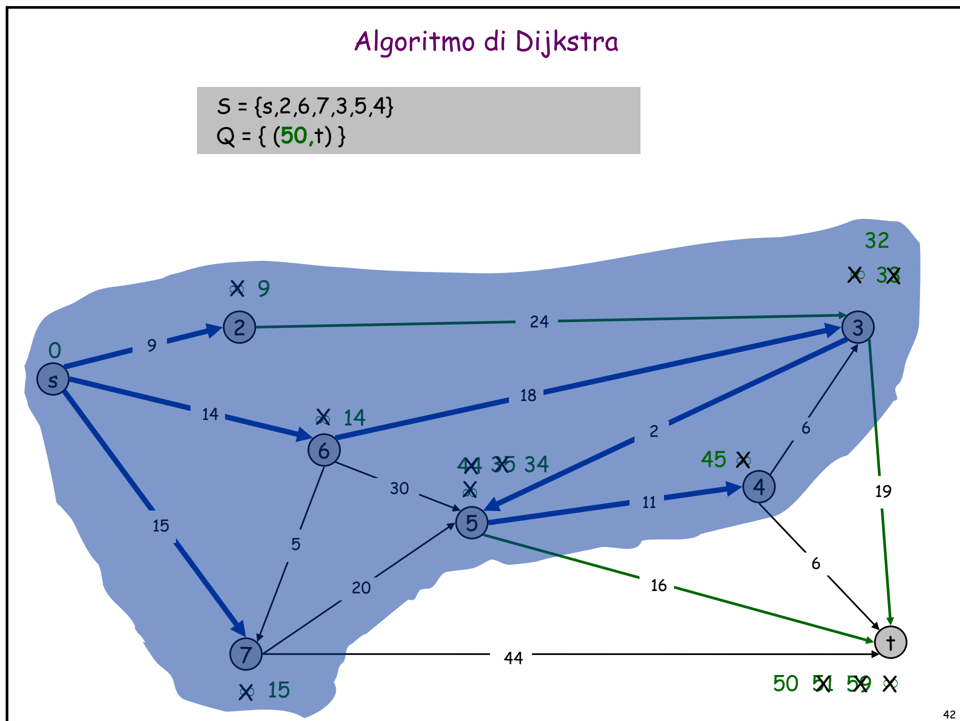
39



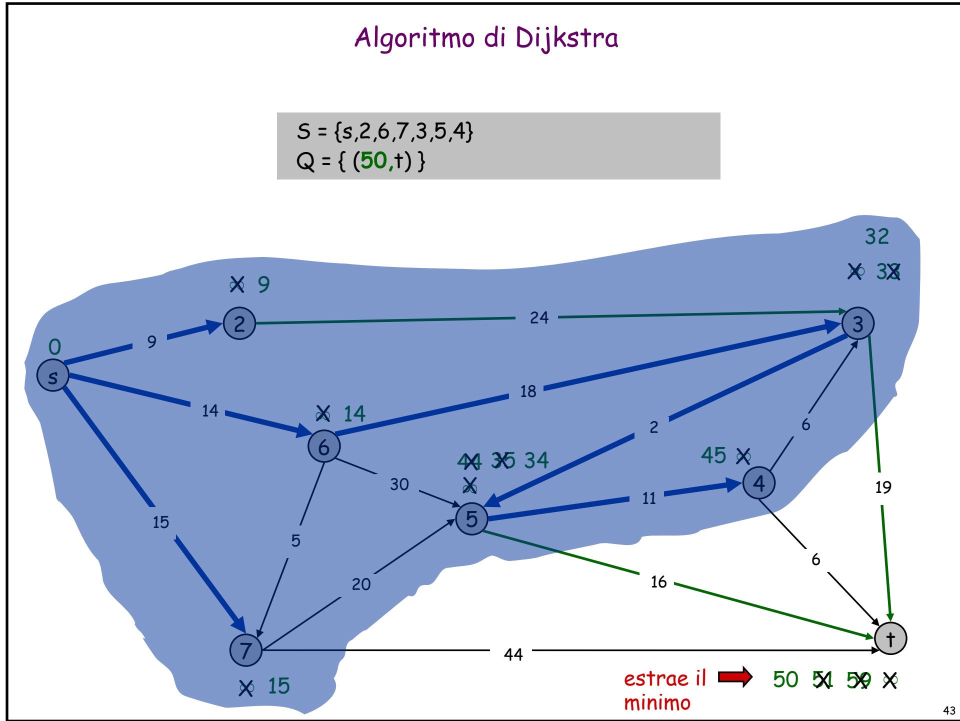
40



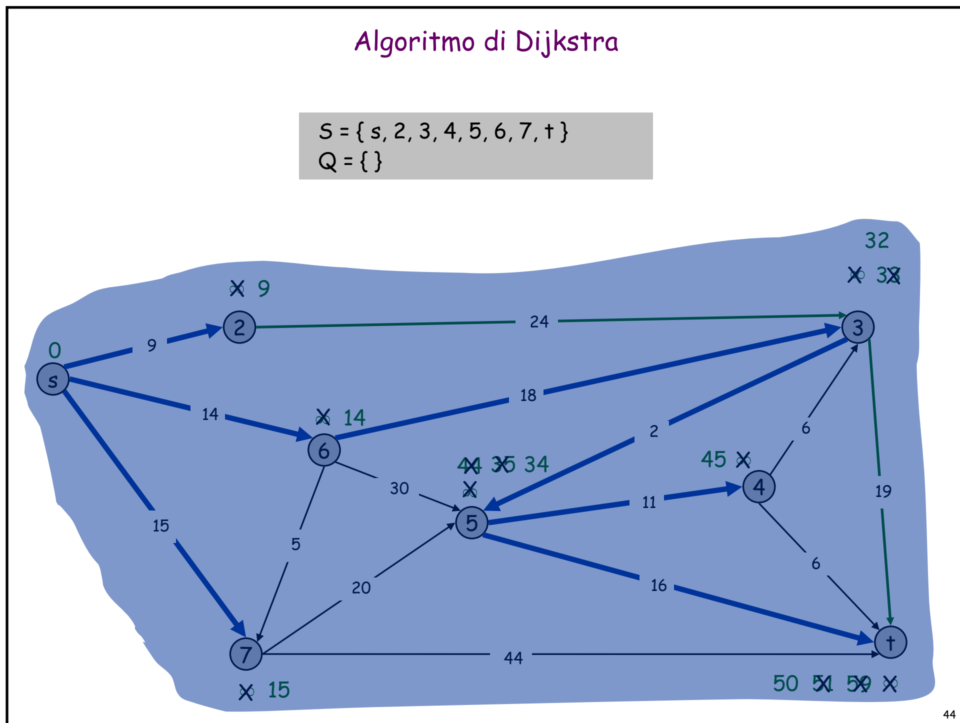
41



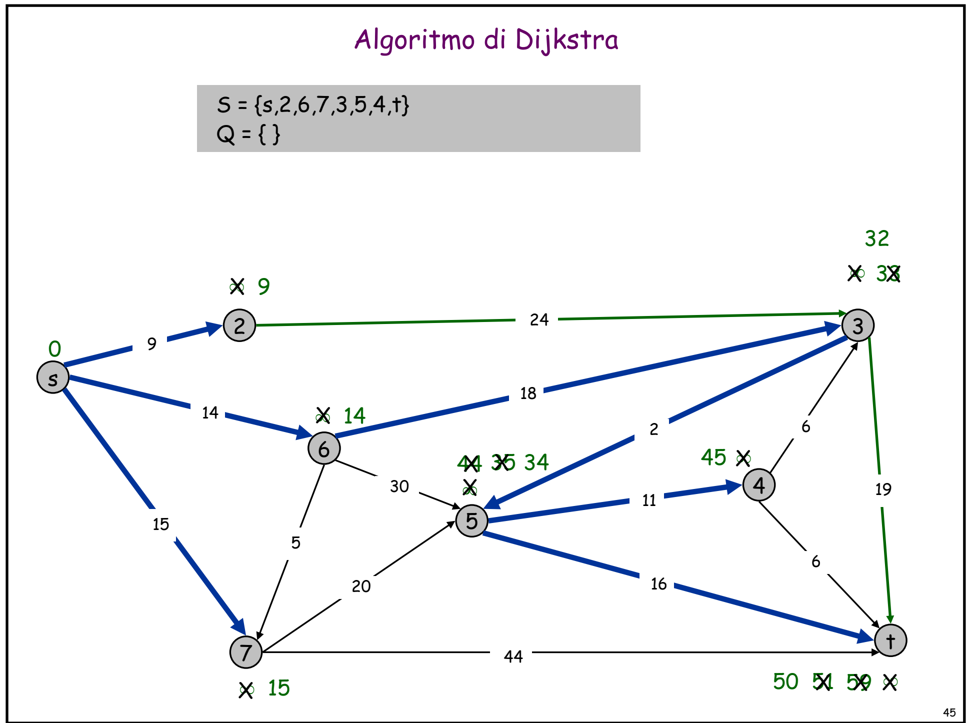
42



43



44



45

Scelta greedy

- Esempio di problema per cui questo approccio non porta alla soluzione ottima
- Problema dello zaino
- Input
  - n oggetti ed uno zaino
  - L'oggetto i pesa $w_i > 0$ chili e ha valore $v_i > 0$.
  - Lo zaino può trasportare fino a W chili.
- Obiettivo: riempire lo zaino in modo da massimizzare il valore totale degli oggetti inseriti tenendo conto che lo zaino trasporta al più W chili
- Esempio: { 3, 4 } ha valore 40.

W = 11

Oggetto	Valore	Peso
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Proviamo diverse strategie greedy

1. seleziona ad ogni passo l'oggetto con valore v_i più grande in modo che il peso totale dei pesi selezionati non superi w → {5,2,1} valore=35
2. seleziona ad ogni passo l'oggetto con peso w_i più piccolo in modo che il peso totale dei pesi selezionati non superi w → {1,2,3} valore=25
3. seleziona ad ogni passo l'oggetto con il rapporto v_i/w_i più grande in modo che il peso totale dei pesi selezionati non superi w → { 5,2,1 }, valore = 35

Progettazione di Algoritmi A.A. 2023-24
A. De Bonis

46

46

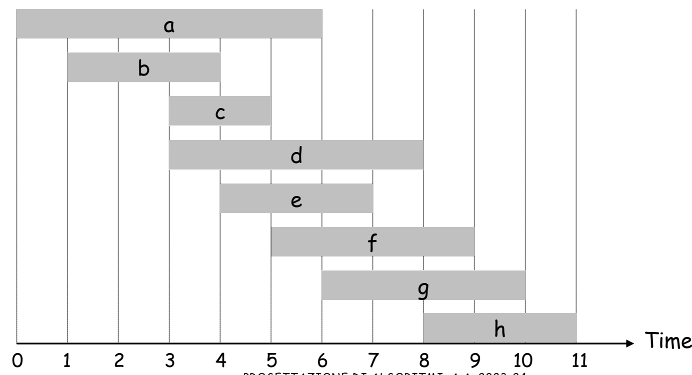
Interval Scheduling

- Supponiamo di avere un'unica risorsa, quale potrebbe essere, ad esempio, un supercomputer, una sala di lettura, uno strumento di laboratorio.
- Molte persone potrebbero avere bisogno di usare la risorsa in determinati periodi di tempo e per questo sottomettono la propria richiesta specificando di avere bisogno della risorsa a partire da un certo tempo s fino ad un tempo t.
- La risorsa puo` essere usata da un un'unica persona alla volta.
- Lo scheduler una volta ricevute tutte le richieste, seleziona un sottoinsieme di richieste che non si sovrappongono nel tempo e rifiuta le restanti.
- Lo scopo è di massimizzare il numero di richieste selezionate dallo scheduler (vogliamo uno scheduling di dimensione massima)

Interval Scheduling

Interval scheduling.

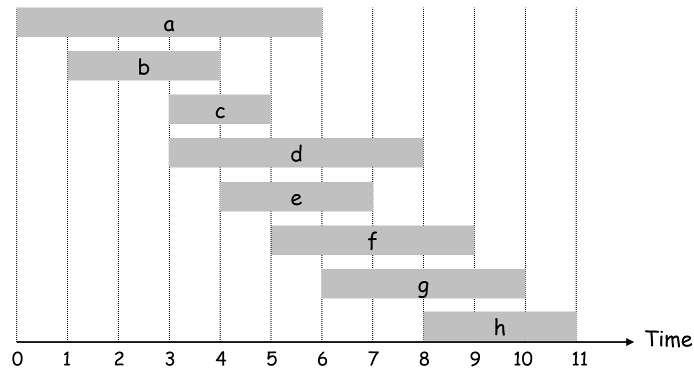
- Input: un insieme di attività ognuna delle quali inizia ad un certo instante s_j e finisce ad un certo istante f_j. Può essere eseguita al più un'attivita alla volta.
- Obiettivo: fare in modo che vengano svolte quante più attività è possibile.



Interval Scheduling

Due job i e j si dicono compatibili se $f_i \leq s_j$ oppure $f_j \leq s_i$.

- Possiamo riformulare l'obiettivo del problema nel modo seguente.
- Obiettivo: trovare un sottoinsieme di cardinalità massima di job a due a due compatibili.



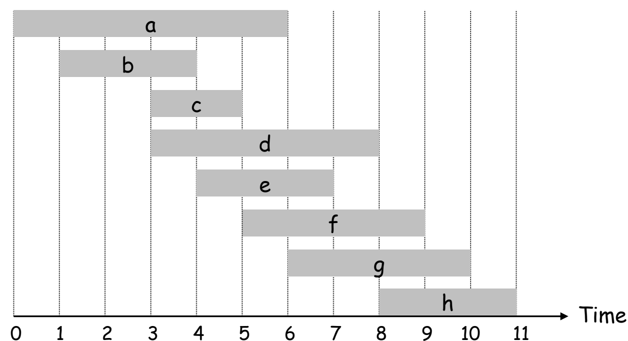
Interval Scheduling

Se all'inizio scegliamo a poi possiamo scegliere o g o h.

- Dopo aver scelto a e g oppure a e h, non possiamo scegliere nessun altro job. Totale = 2

Se all'inizio scegliamo b poi possiamo scegliere uno tra e, f, g e h.

- Se dopo b scegliamo e poi possiamo scegliere anche h. Totale = 3
- Se dopo b scegliamo f poi non possiamo scegliere nessun altro job. Totale = 2



Interval Scheduling: Algoritmi Greedy

Schema greedy. Considera i job in un certo ordine. Ad ogni passo viene esaminato il prossimo job nell'ordinamento e se il job è compatibile con quelli scelti nei passi precedenti allora il job viene inserito nella soluzione.

L'ordinamento dipende dal criterio che si intende ottimizzare localmente.

Diverse strategie basate su diversi tipi di ordinamento

- [Earliest start time] Considera i job in ordine crescente rispetto ai tempi di inizio s_j. Scelta greedy consiste nel provare a prendere ad ogni passo il job che inizia prima tra quelli non ancora esaminati.
- [Earliest finish time] Considera i job in ordine crescente rispetto ai tempi di fine f_j. Scelta greedy consiste nel provare a prendere ad ogni passo il job che finisce prima tra quelli non ancora esaminati.
- [Shortest interval] Considera i job in ordine crescente rispetto alle lloro durate $f_j - s_j$. Scelta greedy consiste nel provare a prendere ad ogni passo il job che dura meno tra quelli non ancora esaminati.
- [Fewest conflicts] Per ogni job, conta il numero c_j di job che sono in conflitto con lui e ordina in modo crescente rispetto al numero di conflitti. Scelta greedy consiste nel provare a prendere ad ogni passo il job che ha meno conflitti tra quelli non ancora esaminati.

Interval Scheduling: Algoritmi Greedy

La strategia "Earliest Start Time" sembra la scelta più ovvia ma…

Problemi con la strategia "Earliest Start Time". Può accadere che il job che comincia per primo finisca dopo tutti gli altri o dopo molti altri.

Interval Scheduling: Algoritmi Greedy

Ma se la lunghezza dei job selezionati incide sul numero di job che possono essere selezionati successivamente perché non provare con la strategia "Shortest Interval"?

Questa strategia va bene per l'input della slide precedente ma…

Problemi con la strategia "Shortest Interval". Può accadere che un job che dura meno di altri si sovrapponga a due job che non si sovrappongono tra di loro. Se questo accade invece di selezionare due job ne selezioniamo uno solo.

Interval Scheduling: Algoritmi Greedy

Visto che il problema sono i conflitti, perché non scegliamo i job che confliggono con il minor numero di job?

Questa strategia va bene per l'input nella slide precedente ma…

Problemi con la strategia "Fewest Conflicts". Può accadere che un job che genera meno conflitti di altri si sovrapponga a due job che non si sovrappongono tra di loro. Se applichiamo questa strategia all'esempio in questa slide, invece di selezionare 4 job ne selezioniamo solo 3.

Interval Scheduling: Algoritmo Greedy Ottimo

L'algoritmo greedy che ottiene la soluzione ottima è quello che usa la stategia "Earliest Finish Time".

```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
f ← 0
A ← ϕ
for j = 1 to n {
   if (sj ≥ f)
      A ← A ∪ {j}
      f ← fj
}
return A
```

Analisi tempo di esecuzione. $O(n \log n)$.

- Costo ordinamento $O(n \log n)$
- Costo for $O(n)$: mantenendo traccia del tempo di fine f dell'ultimo job selezionato, possiamo capire se il job j è compatibile con A verificando che $s_j \geq f$

55

55

Interval Scheduling: Ottimalità soluzione greedy

Teorema. L'algoritmo greedy basato sulla strategia "Earliest Finish Time" è ottimo.

Dim.

- Denotiamo con $i_1, i_2, \dots i_k$ l'insieme di job selezionati dall'algoritmo greedy nell'ordine in cui sono selezionati, cioè in ordine non decrescente rispetto ai tempi di fine.
- Denotiamo con $j_1, j_2, \dots j_m$ l'insieme di job nella soluzione ottima, disposti in ordine non decrescente rispetto ai tempi di fine.

1. Dimostriamo prima che l'esecuzione dei job $i_1, i_2, \dots i_k$ termina non più tardi di quella dei job $j_1, j_2, \dots j_k$
2. Usiamo il punto 1 per dimostrare che non è possibile che k sia minore di m e che quindi la soluzione greedy è ottima.

56

56

Interval Scheduling: Ottimalità soluzione greedy

Dimostriamo il punto 1.:

- Dimostriamo per induzione che per ogni indice r, $1 \le r \le k$, si ha che il tempo di fine di i_r è **non** più grande di quello di j_r .
- **Base.** r=1 : Banalmente vero perchè la prima scelta greedy seleziona la richiesta con il minimo tempo di fine.
- **Passo induttivo. Ipotesi induttiva:** per $r-1 \ge 1$, il tempo di fine di i_{r-1} è **non** più grande di quello di j_{r-1} .
- Il job j_r è compatibile con j_{r-1} e quindi $f(j_{r-1}) \le s(j_r)$ e siccome per ipotesi induttiva $f(i_{r-1}) \le f(j_{r-1})$ → $f(i_{r-1}) \le s(j_r)$ → j_r è compatibile con i_{r-1} .
- I tempi di fine di $i_1, i_2, \ldots i_{r-2}$ sono non più grandi di quello di i_{r-1} → $f(i_p) \le s(j_r)$ per ogni $1 \le p \le r-1$ → j_r è compatibile con $i_1, i_2, \ldots i_{r-2}\ i_{r-1}$ (Usiamo gia\` questa osservazione nell'algoritmo nella slide 54. In che punto?).
- All'r-esimo passo l'algoritmo greedy sceglie i_r → i_r finisce non più tardi di tutti i job compatibili con $i_1, i_2, \ldots i_{r-1}$ e quindi anche non più tardi di j_r.

57

57

Interval Scheduling: Ottimalità soluzione greedy

- Abbiamo dimostrato che, per ogni indice $r \le k$, il tempo di fine di i_r è **non** più grande di quello di j_r .

  → il tempo di fine di i_k è non più grande di quello di j_k .
- Poiche' i job $i_1, i_2, \ldots i_k$ sono ordinati in base ai tempi di fine

  → i_k è il job che finisce piu\` tardi tra i job $i_1, i_2, \ldots i_k$
- → e → insieme implicano che tutti i job $i_1, i_2, \ldots i_k$ terminano non piu\` tardi di $j_k$ e di conseguenza non piu\` tardi della sequenza di job $j_1, j_2, \ldots j_k$

Continua nella prossima slide

58

58

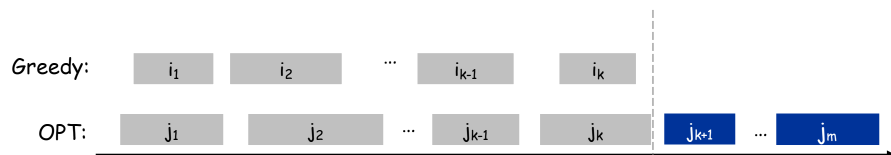
Interval Scheduling: Ottimalità soluzione greedy

Dimostriamo il punto 2.

• Supponiamo per assurdo che la soluzione greedy non sia ottima e quindi che $k < m$. Quindi la sequenza $j_1, j_2, \dots j_m$ conterrà il job j_{k+1}

• Per il punto 1, l'esecuzione di $i_1, i_2, \dots i_k$ termina **non** più tardi dell'esecuzione di $j_1, j_2, \dots j_k$. e si ha quindi che $i_1, i_2, \dots i_k$ sono compatibili con j_{k+1}

• L'algoritmo greedy, dopo aver inserito $i_1, i_2, \dots i_k$ nella soluzione, passa ad esaminare i job con tempo di fine maggiore o uguale a quelli di $i_1, i_2, \dots i_k$. Tra questi job vi è j_{k+1} che è compatibile con $i_1, i_2, \dots i_k$ per cui è impossibile che l'algoritmo greedy inserisca nella soluzione solo k job. Siamo giunti ad una contraddizione e quindi è impossibile che $k < m$.



59

59

Provare l'ottimalità della soluzione greedy

Come abbiamo provato l'ottimalità della soluzione greedy?

- Abbiamo prima di tutto dimostrato che la soluzione greedy "sta sempre avanti" a quella ottima.
  - Cosa vuol dire "sta sempre avanti"?
  - L'idea alla base della strategia greedy Earliest Finish Time è la seguente: quando si usa la risorsa è bene liberarla il prima possibile perchè ciò massimizza il tempo a disposizione per eseguire le restanti richieste
  - In questa ottica, una soluzione per il problema dell'interval scheduling "sta sempre avanti" ad un'altra se ad ogni passo seleziona una richiesta che termina non più tardi della corrispondente richiesta della soluzione ottima.
- Abbiamo usato il fatto che la soluzione greedy "sta sempre avanti" a quella ottima per provare che la soluzione greedy non può contenere un numero di richieste inferiore a quello della soluzione ottima

60

60