Algoritmi greedy
III parte

Progettazione di Algoritmi a.a. 2020-21
Matricole congrue a 1
Docente: Annalisa De Bonis

42

Problema del caching offline ottimale

- Caching. Una cache è un tipo di memoria a cui si può accedere molto velocemente. Una cache permette accessi più veloci rispetto alla memoria principale ma ha dimensioni molto più piccole.

- Possiamo pensare ad una cache come ad un posto in cui possiamo tenere a portata di mano le cose che ci servono ma che è di dimensione limitata per cui dobbiamo riflettere bene su cosa mettervi e su cosa togliere per evitare che ci serva qualcosa che non abbiamo a portata di mano.

  - Cache hit: elemento già presente nella cache quando richiesto.
  - Cache miss: elemento non presente nella cache quando richiesto: occorre portare l'elemento richiesto nella cache e se la cache è piena occorre espellere dalla cache alcuni elementi per fare posto a quelli richiesti.

43

Problema del caching offline ottimale

Caching. Formalizziamo il problema come segue:

- Memoria centrale contenente un insieme U di n elementi
- Cache con capacità di memorizzare k elementi.
- Sequenza di m richieste di elementi $d_1, d_2, \ldots, d_m$ fornita in input in modo offline (tutte le richieste vengono rese note all'inizio). Non molto realistico!
- Assumiamo che inizialmente la cache sia piena, cioè contenga k elementi

Def. Un **eviction schedule ridotto** è uno scheduling degli elementi da espellere, cioè una sequenza che indica quale elemento espellere **quando c'è bisogno di far posto ad un elemento richiesto** che non è in cache.

Un eviction schedule non ridotto è uno scheduling che può decidere di inserire in cache un elemento che non è stato richiesto

Problema del caching offline ottimale

Un eviction schedule ridotto inserisce in cache un elemento solo nel momento in cui è richiesto e se non è presente già in cache al momento della richiesta.

Osservazione. In un eviction schedule ridotto il numero di inserimenti in cache è uguale al numero di cache miss.

Obiettivo. Un eviction schedule ridotto che minimizzi il numero di inserimenti (o equivalentemente di cache miss).

.

Eviction Schedule ridotto

a	a	b	c
a	a	x	c
c	a	d	c
d	a	d	b
a	a	c	b
b	a	x	b
c	a	c	b
a	a	c	b
a	a	c	b

Uno schedule non ridotto

a	a	b	c
a	a	b	c
c	a	b	c
d	a	d	c
a	a	d	c
b	a	d	b
c	a	c	b
a	a	c	b
a	a	c	b

Uno schedule ridotto

46

46

caching offline ottimale: Farthest-In-Future

Farthest-in-future. Quando viene richiesto un elemento che non è presente in cache, espelli dalla cache l'elemento che sarà richiesto più in là nel tempo o che non sarà più richiesto.

Cache in questo momento: 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 ↑ Espelli questa ↑

Teorema. [Belady, 1960s] Farthest-in-future è uno schedule (ridotto) ottimo.

Dim. La tesi del teorema è intuitiva ma la dimostrazione è sottile.

47

Problema del caching offline ottimale

Esempio.

Cache di dimensione k = 2,
Inizialmente la cache contiene ab,
Le richieste sono a, b, c, b, c, a, a, b.
Usiamo farthest-in-future:
Quando arriva la prima richiesta di c viene espulso a perchè a verrà richiesto più in là nel tempo rispetto a b.
Quando arriva la seconda richiesta di a viene espulso c perchè c non viene più richiesto

Scheduling ottimo: 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

48

```
Input: requests d1,d2,…,dm arranged in ascending order of arrival time
For each element d, let L[d] the list of positions j s.t.dj=d;
initially L[d]=∅
Let Q be a priority queue //entries associated with elements in cache
for j = 1 to m {                                                    ┐
  if(list L[dj] is empty and dj is in the cache)                    │ O(m+k log k )
      insert (j,dj) in Q     //j is the key                         │ k = dimensione
  append j to list L[dj]                                            │ cache
}                                                                   ┘
for j = 1 to m {                                                    ┐
   if (dj is in the cache){                                         │
        remove first element from L[dj]                             │
        if(L[dj] is empty)                                          │
          replace key of dj with m+1 in Q                           │ O(m log k )
        else                                                        │ k = dimensione
          {p ← first element of L[dj]                               │ cache
           replace key of dj with p in Q }                          │
      }                                                             │
   else{       //dj needs to be brought into the cache             │
       (h,dh) ← ExtractMax(Q)                                       │
        evict dh from the cache and bring dj to the cache           │
        remove first element from L[dj]                             │
        if(L[dj] is NOT empty) {                                    │
            p ← first element of L[dj]                              │
            insert (p,dj ) in Q  }                                  │
        else insert (m+1,dj) in Q                                   ┘
  }
```

49

Strutture dati dell'algoritmo di Belady

- Per ogni elemento d nella sequenza delle richieste. la lista L[d] contiene le posizioni in cui d appare nella sequenza.
  - Nel primo for viene scandita la sequenza delle richieste e per ciascun elemento d della sequenza vengono inserite in L[d] tutte le posizioni in cui d appare nella sequenza
    - Ad esempio se la sequenza delle richieste è a, b, c, b, c, a, a, b allora L[b] = < 2 4 8>, 2 è in testa e 8 in coda alla lista.
  - Nella j-esima iterazione del secondo for viene eliminato l'intero in testa a $L[d_j]$ in modo che in testa alla lista venga a trovarsi l'intero che indica la prossima posizione della sequenza in cui verra` incontrato nuovamente d_j.

50

50

Strutture dati dell'algoritmo di Belady

- Per ogni elemento d in cache, la coda a priorita` Q contiene un'entrata (k,d), dove la chiave k è un intero che indica il punto della sequenza in cui verra` richiesto nuovamente l'elemento d. Se k=m+1 allora vuol dire che d non verra` piu` richiesto.
  - Nel primo for vengono inserite le entrate per gli elementi gia` presenti in cache con chiave uguale alla prima posizione in cui questi appaiono nella sequenza delle richieste.
    - Ad esempio se la sequenza delle richieste è a, b, c, b, c, a, a, b e inizialmente la cache contiene gli elementi **a** e **b** allora inizialmente Q contiene le entrate (a,1) (b,2)
  - Nella j-esima iterazione del secondo for, l'if-else gestisce i due seguenti casi.
    - d_j è presente in cache. In questo viene rimosso l'intero in testa a $L[d_j]$ e viene aggiornata la chiave di d_j con l'intero che si trova ora in testa a $L[d_j]$, sempre che $L[d_j]$ non sia vuota . Se $L[d_j]$ è vuota allora la chiave di d_j viene sostituita con m+1.
    - d_j non è presente in cache. In questo caso viene estratta da Q l'entrata (h, d_h) con chiave massima e h viene espulso dalla cache. Viene poi rimosso l'intero che si trova in testa a $L[d_j]$. Se dopo questa rimozione $L[d_j]$ non è vuota allora viene inserita in Q l'entrata (p, d_j), dove p indica l'intero che ora si trova in testa a $L[d_j]$. Se $L[d_j]$ è vuota allora in Q viene inserita l'entrata $(m+1, d_j)$

51

51

Analisi dell'algoritmo di Belady

L'algoritmo nella slide precedente richiende tempo O(m log k) se

- Ad ogni elemento è associato un flag che è true se e solo l'elemento è in cache
- Usiamo un heap binario come coda a priorità
  - assumiamo che l'heap supporti l'operazione changeKey che consente di modificare la chiave di un'entrata arbitraria dell'heap e l'operazione di remove che consente di cancellare un'entrata arbitaria. Queste operazioni possono essere implementata in modo da richiedere tempo O(log k).
- Consideriamo costante il tempo per espellere e inserire ciascun elemento in cache

52

52

Farthest-In-Future: ottimalità

La dimostrazione dell'ottimalità si basa sui seguenti fatti che andremo a dimostrare

1. Ogni schedule può essere trasformato in uno schedule ridotto senza aumentare il numero di inserimenti
2. Ogni schedule ridotto può essere trasformato nello schedule FF senza aumentare il numero di cache miss

- Per la 1 possiamo trasformare uno schedule ottimo S in uno schedule ridotto S' senza aumentare il numero di inserimenti
- Per la 2 possiamo trasformare S' nello schedule FF senza aumentare il numero di cache miss (= numero inserimenti)

→ FF va incontro allo stesso numero di inserimenti dell'algoritmo ottimo ed è quindi anch'esso ottimo

53

53

Farthest-In-Future: ottimalità

1. Un qualsiasi eviction schedule S può essere trasformato in un eviction schedule ridotto S' senza aumentare il numero di inserimenti nella cache.

Dim.

- Se ad un certo tempo t, S porta un certo elemento d in cache e d è stato richiesto al tempo t allora S' fa la stessa cosa.
- Se ad un certo tenpo t, S porta un certo elemento d in cache senza che d sia stata richiesto, S' fa finta di fare lo stesso ma di fatto non inserisce niente in cache ed eventualmente inserisce d successivamente quando d è richiesto.
- Il numero totale di inserimenti effettuati da S' è lo stesso di S se tutte le volte che S inserisce un elemento d non richiesto accade che d venga richiesto in seguito. Se invece qualcuno degli elementi inseriti da S non è richiesto nè in quel momento né successivamente allora S' effettua un numero minore di inserimenti.

Farthest-In-Future: ottimalità

Teorema. Sia S uno **scheduling ridotto** che fa le stesse scelte dello scheduling S_{FF} di farthest-in-future per i primi j elementi, per un certo $j \geq 0$. E' possibile costruire uno scheduling ridotto S' che fa le stesse scelte di S_{FF} per i primi j+1 elementi e determina un numero di cache miss non maggiore di quello determinato da S.

Dim.

Produciamo S' nel seguente modo.

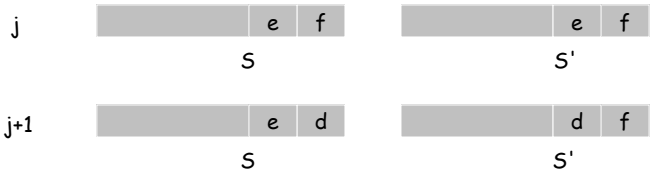
- Consideriamo la (j+1)-esima richiesta e sia $d = d_{j+1}$ l'elemento richiesto,
- Siccome S e S_{FF} hanno fatto le stesse scelte fino alla richiesta j-esima, quando arriva la richiesta (j+1)-esima il contenuto della cache per i due scheduling è lo stesso.
  - Caso 1: d è già nella cache. In questo caso sia S_{FF} che S non fanno niente perché entrambi sono ridotti.
  - Caso 2: d non è nella cache ed S espelle lo stesso elemento espulso da S_{FF}
- In questi due casi basta porre S'=S visto che S ed S_{FF} hanno lo stesso comportamento anche per la (j+1)-esima richiesta.

Continua nella prossima slide

55

Farthest-In-Future: ottimalità

- Caso 3: d non è nella cache e S_{FF} espelle e mentre S espelle f ≠ e.
  - Costruiamo S' a partire da S modificando la (j+1)-esima scelta in modo che S' espella e invece di f



  - ora S' ha lo stesso comportamento di S_{FF} per le prime j+1 richieste. Occorre dimostrare che S' riesce ad effettuare successivamente delle scelte che non determinano un numero di cache miss maggiore di quello di S.

Continua nella prossima slide

56

56

Farthest-In-Future: ottimalità

- Dopo la (j+1)-esima richiesta facciamo fare ad S' le stesse scelte di S fino a che, ad un certo tempo j', accade per la prima volta che non è possibile che S ed S' facciano la stessa scelta.
- A questo punto S' deve fare necessariamente una scelta diversa da quella di S. Facciamo però in modo che la scelta di S' renda il contenuto della cache di S' identico a quello della cache di S.
- Da questo punto in poi il comportamento di S' sarà identico a quello di S per cui andrà incontro allo stesso numero di cache miss.

Continua nella prossima slide

57

Farthest-In-Future: ottimalità

Notiamo che siccome i due scheduling fino al tempo j' si sono comportati in modo diverso un'**unica** volta, il contenuto della cache nei due scheduling differisce in un singolo elemento che è uguale ad e in S ed è uguale a f in S'.

S | | e | S' | | f |

Indichiamo con g l'elemento richiesto al tempo j'.

I casi che al tempo j avrebbero permesso ad S' di fare la stessa scelta di S sono:

- g ≠ e, g ≠ f, g è presente nella cache di S: in questo caso g è presente anche nella cache di S' ed S' non fa niente come S.
- g ≠ e, g ≠ f, g non è presente nella cache di S ed S espelle un elemento diverso da e: in questo caso g non è neanche nella cache di S' ed S' può espellere lo stesso elemento espulso da S.

Nella prossima slide vediamo i casi in cui S' non può fare la stessa scelta di S.

Farthest-In-Future: ottimalità

Caso 3.1: g ≠ e, g ≠ f, g non è nella cache di S ed S espelle e. In questo caso g non è neanche nella cache di S'. Facciamo in modo che S' espella f. In questo modo dopo il tempo j' il contenuto della cache di S è uguale a quello della cache di S'. Il numero di cache miss di S è lo stesso di S'.

S | | g | S' | | g |

Caso 3.2: g = f ed S espelle e. In questo caso S' non fa niente e da quel momento in poi le cache di S è uguale a quello di S'. Il numero di cache miss di S' è minore di quello di S.

S | | f | S' | | f |

Caso 3.3: g = f ed S espelle e' ≠ e. In questo caso e' è presente anche nella cache di S' . Facciamo in modo che, al tempo j', S' espella e' ed inserisca e. Da questo momento la cache di S e quella di S' hanno lo stesso contenuto e il numero di cache miss in cui incorreranno i due scheduling sarà lo stesso. Il teorema non è ancora dimostrato per questo caso in quanto S' non è ridotto. Abbiamo però dimostrato che possiamo rendere S' ridotto senza aumentare il numero di inserimenti. Lo scheduling ridotto farà le stesse scelte di S_{FF} per i primi j+1 elementi in quanto sara identico ad S' fino al tempo j'-1.

S | | f | | e | S' | | e | | f |

Farthest in Future: ottimalità

Resterebbe il caso g=e.

- Notiamo che al tempo j' non può accadere che g=e. Vediamo perché.
  - Al tempo j+1 S_{FF} ha espulso e al posto di f per cui, dopo il tempo j+1, e viene richiesto più tardi di f o non viene richiesto affatto.
  - → Se dopo il tempo j+1 vi è una richiesta di e allora questa richiesta deve essere preceduta da una richiesta di f.

  - Come abbiamo visto nella slide precedente (casi 3.2 e 3.3) la richiesta di f in un tempo successivo al tempo j+1 porterebbe S' a fare una scelta diversa da S ma ciò non è possibile perché stiamo assumendo che j' è il primo momento (successivo al tempo j+1) in cui accade che S' non può fare la stessa scelta di S.

60

60

Farthest-In-Future: ottimalità

2. Ogni schedule ridotto può essere trasformato nello schedule FF senza aumentare il numero di cache miss

Dim.

- Consideriamo un eviction schedule ridotto S.
- Applicando il teorema precedente con j=0, si ha che possiamo trasformare S in uno schedule ridotto S_1 che per la prima richiesta si comporta come S_{FF} e determina un numero di cache miss non maggiore del numero di cache miss di S.
- Applicando il teorema con j=1, si ha che possiamo trasformare S_1 in uno schedule ridotto S_2 che per la prime due richieste si comporta come S_{FF} e determina un numero di cache miss non maggiore del numero di cache miss di S_1 e quindi di S.
- Continuando in questo modo, applicando cioè il teorema precedente per j=0,1,…,m-1, arriviamo ad uno schedule S_m che effettua esattamente le stesse scelte di S_{FF} (S_m= S_{FF}) e determina un numero di cache miss non maggiore del numero di cache miss di S.

61

Farthest-In-Future: ottimalità

Teorema. Farthest-in-future produce un eviction schedule S_{FF} ottimo.
Dim.

- Sia S^* uno schedule ridotto ottimo. Per il punto 2 (slide precedente) si ha che S^* può essere trasformato nello schedule S_{FF} senza aumentare il numero di cache miss. Di conseguenza S_{FF} determina lo stesso numero di cache miss di S^* ed è quindi uno schedule ridotto ottimo.

- Osserviamo che S_{FF} è ottimo non solo se restringiamo la nostra attenzione agli schedule ridotti ma è ottimo se consideriamo tutti i tipi di schedule (ridotti e non ridotti) perchè per il punto 1 possiamo trasformare uno schedule ottimo in uno schedule ridotto che effettua lo stesso numero di inserimenti.
  - NB: in questo caso parliamo di numero di inserimenti

Il problema del caching nella realtà

- Il problema del caching è tra i problemi più importanti in informatica.
- Nella realtà le richeste non sono note in anticipo come nel modello offline.
- E' più realistico quindi considerare il modello online in cui le richieste arrivano man mano che si procede con l'esecuzione dell'algoritmo.
- L'algoritmo che si comporta meglio per il modello online è l'algoritmo basato sul principio *Least-Recently-Used* o su sue varianti.
- *Least-Recently-Used (*LRU). Espelli la pagina che è stata richiesta meno recentemente
  - Non è altro che il principio Farthest in Future con la direzione del tempo invertita: più lontano nel passato invece che nel futuro
  - E' efficace perché in genere un programma continua ad accedere alle cose a cui ha appena fatto accesso (locality of reference). E' facile trovare controesempi a questo ma si tratta di casi rari.

Cammini minimi

- Si vuole andare da Napoli a Milano in auto percorrendo il minor numero di chilometri
- Si dispone di una mappa stradale su cui sono evidenziate le intersezioni tra le strade ed è indicata la distanza tra ciascuna coppia di intersezioni adiacenti
- Come si può individuare il percorso più breve da Napoli a Milano?

64

Cammini minimi

- Esempi di applicazioni dei cammini minimi in una rete
- Trovare il cammino di **tempo minimo** in una rete
- Se i pesi esprimono l'inaffidabilità delle connessioni in una rete, trovare il collegamento che è **più sicuro**

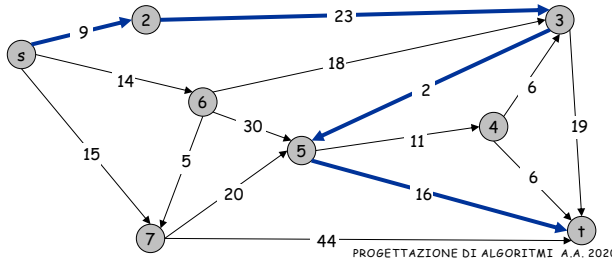
65

Il problema dei cammini minimi

- Input:
  - Grafo direzionato G = (V, E).
  - Per ogni arco e, valore ℓ_e (costo, peso o lunghezza dell'arco e)
  - s = sorgente
- Def. Per ogni percorso direzionato P, $\ell(P)$ = somma delle lunghezze degli archi in P.

Il problema dei cammini minimi: trova i percorsi direzionati più corti da s verso tutti gli altri nodi.

NB: Se il grafo non è direzionato possiamo sostituire ogni arco (u,v) con i due archi direzionati (u,v) e (v,u)



Costo del percorso da s a t
s-2-3-5-t = 9 + 23 + 2 + 16 = 50

Varianti del problema dei cammini minimi

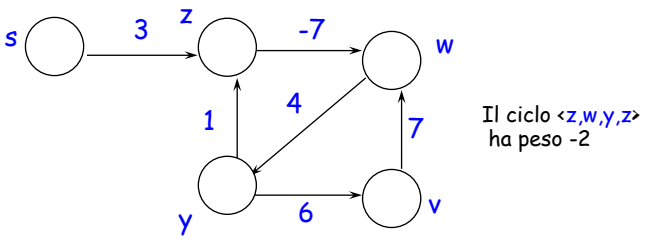
- Single Source Shortest Paths: determinare il cammino minimo da un dato vertice sorgente s ad ogni altro vertice
- Single Destination Shortest Paths: determinare i cammini minimi ad un dato vertice destinazione t da tutti gli altri vertici
  - Si riduce a Single Source Shortest Path invertendo le direzioni degli archi
- Single-Pair Shortest Path: per una data coppia di vertici u e v determinare un cammino minimo da un dato vertice u a v
  - i migliori algoritmi noti per questo problema hanno lo stesso tempo di esecuzione asintotico dei migliori algoritmi per Single Source Shortest Path.
- All Pairs Shortest Paths: per ogni coppia di vertici u e v, determinare un cammino minimo da u a v

Cammini minimi

- Soluzione inefficiente:
  - si considerano tutti i percorsi possibili e se ne calcola la lunghezza
  - l'algoritmo non termina in presenza di cicli
- Si noti che l'algoritmo di visita BFS è un algoritmo per Single Source Shortest Paths nel caso in cui tutti gli archi hanno lo stesso peso

Cicli negativi

- Se esiste un ciclo negativo lungo un percorso da s a v, allora non è possibile definire il cammino minimo da s a v



Il ciclo <z,w,y,z> ha peso -2

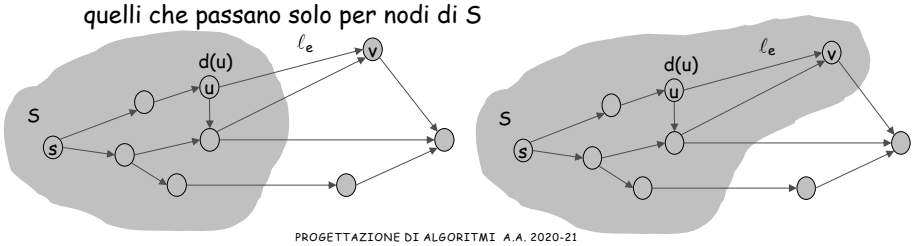
- Attraversando il ciclo <z,w,y,z> un numero arbitrario di volte possiamo trovare percorsi da s a v di peso arbitariamente piccolo

Algoritmo di Dijkstra

in G solo archi con lunghezze maggiori o uguali di zero

Algoritmo di Dijkstra (1959).

- Ad ogni passo mantiene l'insieme S dei nodi esplorati, cioé di quei nodi u per cui è già stata calcolata la distanza minima d(u) da s.
- Inizializzazione S = { s }, d(s) = 0.
- Ad ogni passo, sceglie tra i nodi non ancora in S ma adiacenti a qualche nodo di S, quello che può essere raggiunto nel modo più economico possibile (scelta greedy)
- In altra parole sceglie v che minimizza $d'(v) = \min_{e=(u,v):\, u \in S} d(u) + \ell_e$, aggiunge v a S e pone d(v) = d'(v).
- d'(v) rappresenta la lunghezza del percorso più corto da s a v tra quelli che passano solo per nodi di S



70

Algoritmo di Dijkstra

in G solo archi con lunghezze maggiori o uguali di zero

```
Dijkstra's Algorithm (G, ℓ)
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
```

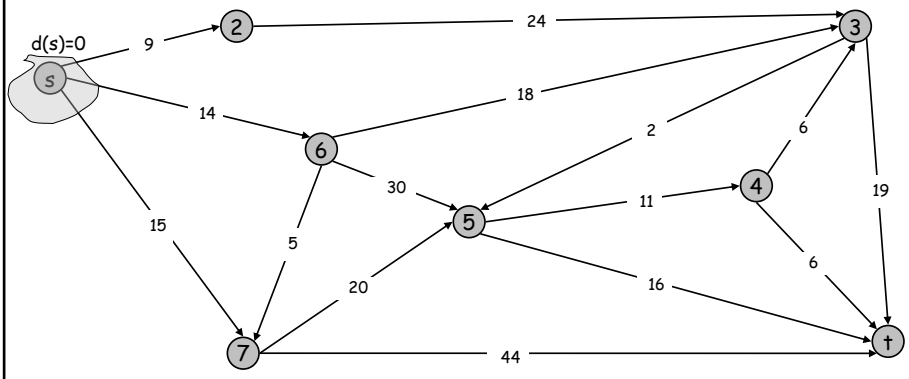
Questa versione dell'algoritmo di Dijkstra assume che tutti i nodi siano raggiungibili da s.

Esercizio: Ad ogni passo l'algoritmo di Dijkstra aggiunge un certo nodo v ad S. Dimostrare per induzione che il percorso più corto da s a v computato dall'algoritmo passa solo attraverso nodi già inseriti in S.

71

Algoritmo di Dijkstra

S={s} computo d'(2)=9 , d'(6)=14, d'(7)=15 → scelgo 2 e pongo d(2)=9, shortest path = s-2

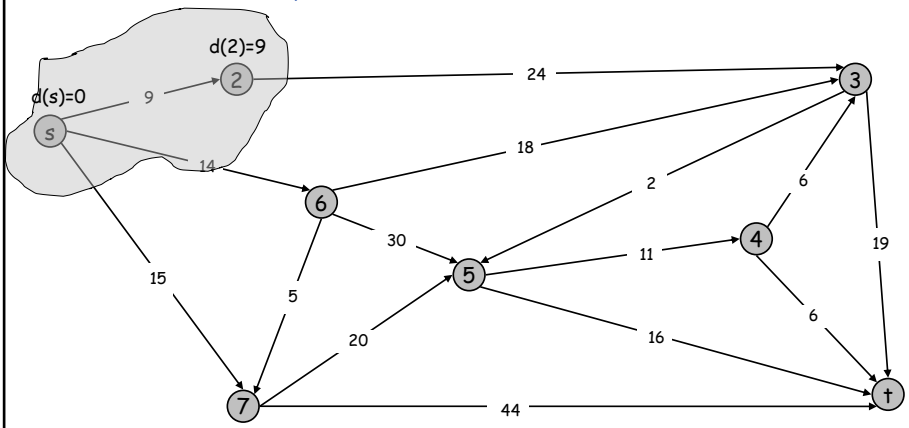


72

72

Algoritmo di Dijkstra

S={s,2} computo d'(6)=14, d'(7)=15, d'(3)=9+24=33 → scelgo 6 e pongo d(6)=14 , shortest path = s-6

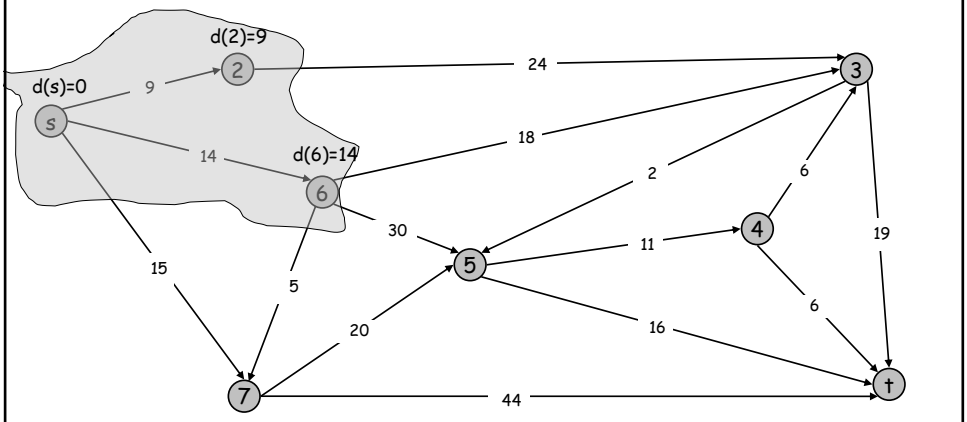


73

73

Algoritmo di Dijkstra

S={s,2,6} computo d'(7)=min{15,14+5}=15, d'(5)=14+30=44, d'(3)=min{9+24, 14+18} =32 → scelgo 7 e pongo d(7)=15, shortest path = s-7

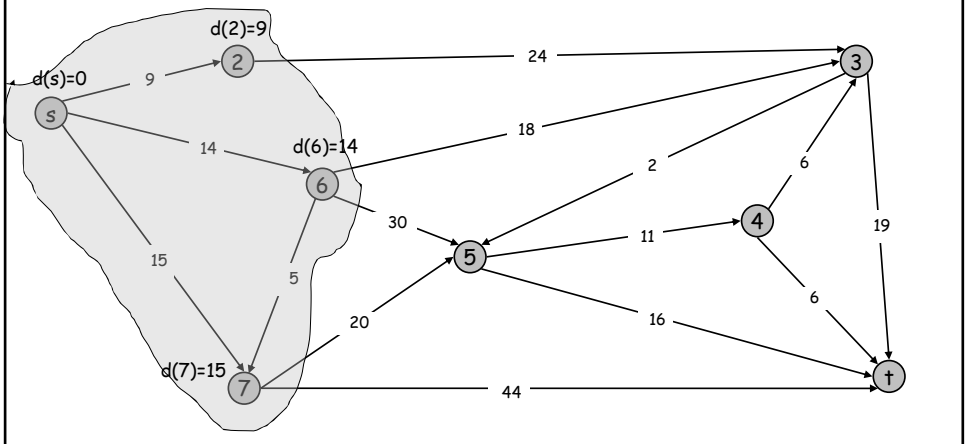


74

74

Algoritmo di Dijkstra

S={s,2,6,7} computo d'(5)=min{14+30,15+20}=35, d'(3)=min{9+24, 14+18} =32, d'(t)=15+44=59 → scelgo 3 e pongo d(3)=32 , shortest path = s-6-3

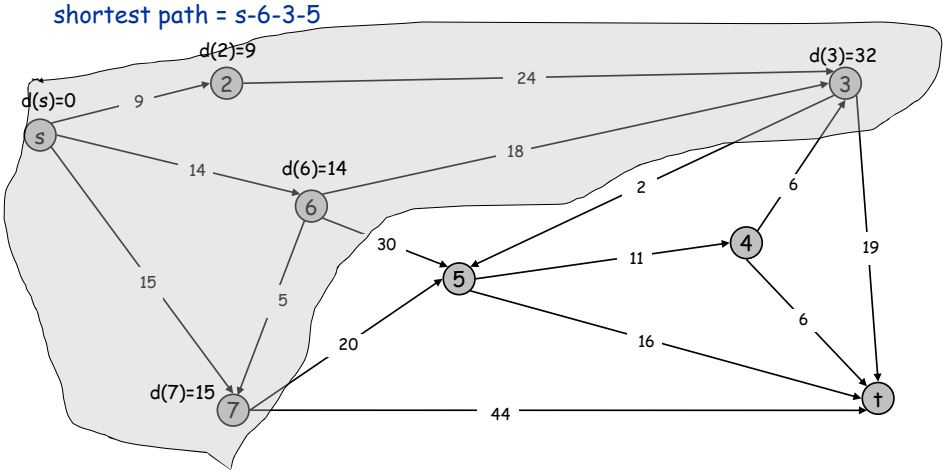


75

75

Algoritmo di Dijkstra

S={s,2,6,7,3} computo d'(5)=min{14+30,15+20,32+2}=34,
d'(t)=min{15+44, 32+19}=51 → scelgo 5 e pongo d(5)=34 ,
shortest path = s-6-3-5

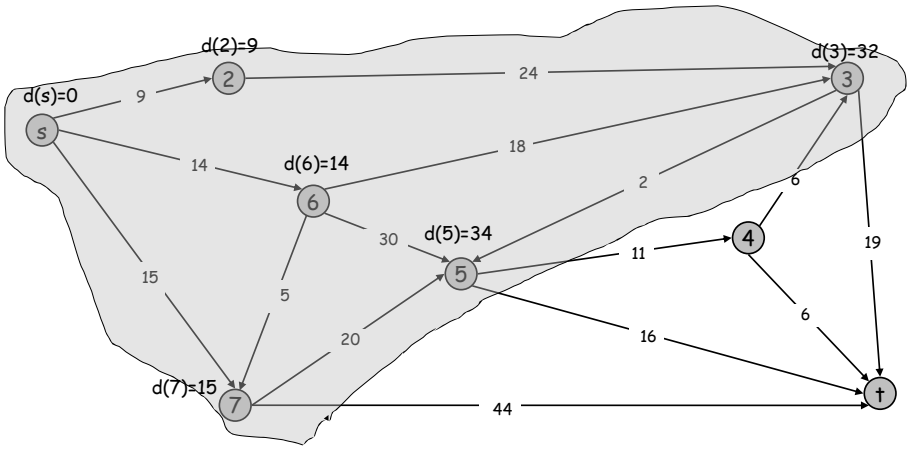


76

76

Algoritmo di Dijkstra

S={s,2,6,7,3,5} computo d'(4)=34+11=45, d'(t)=min{15+44, 32+19, 34+16}=50
→ scelgo 4 e pongo d(4)=45 , shortest path = s-6-3-5-4

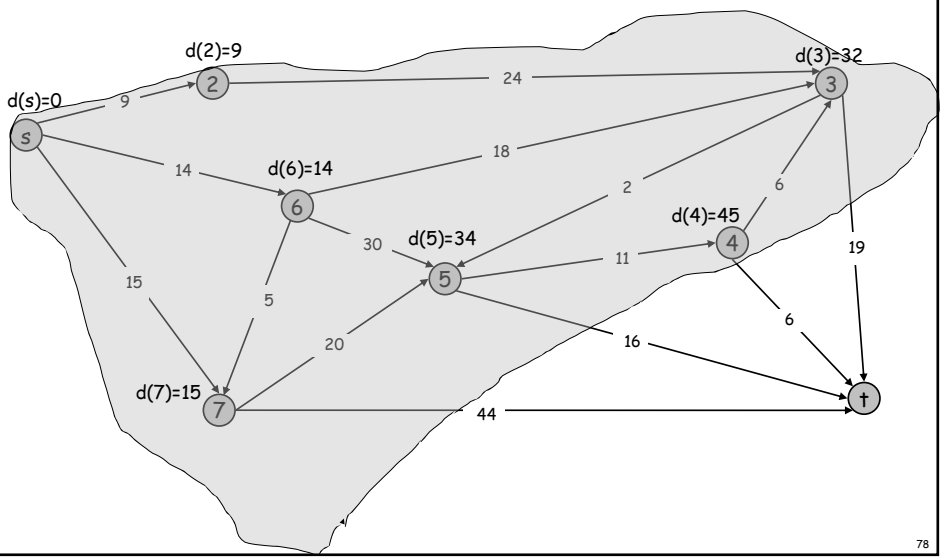


77

77

Algoritmo di Dijkstra

S={s,2,6,7,3,5,4} computo d'(t)=min{15+44, 32+19, 34+16,45+6}=50
→ scelgo t e pongo d(t)=50 , shortest path = s-6-3-5-t

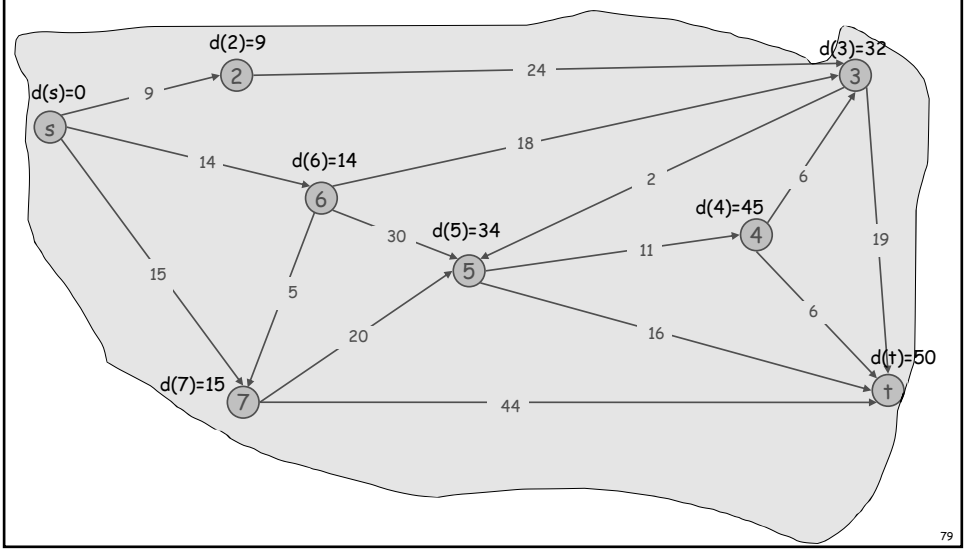


78

78

Algoritmo di Dijkstra

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



79

79

Algortimo di Dijksta: Correttezza

Teorema. Sia G un grafo in cui per ogni arco e è definita una lunghezza $\ell_e \geq 0$ (è fondamentale che ℓ_e **non sia negativa**). Per ogni nodo u ∈ S il valore d(u) calcolato dall'algoritmo di Dijkstra è la lunghezza del percorso più corto da s a u.

Dim. (per induzione sulla cardinalità |S| di S)

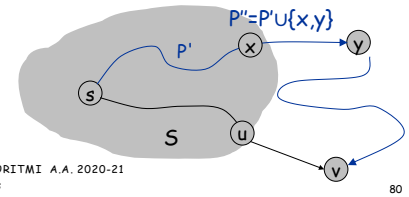
Base: |S| = 1. In questo caso S={s} e d(s)=0 per cui la tesi vale banalmente.

Ipotesi induttiva: Assumiamo vera la tesi per |S| = k ≥ 1.

- Sia v il prossimo nodo inserito in S dall'algoritmo e sia (u,v) l'arco attraverso il quale è stato raggiunto v, cioè quello per cui si ottiene

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

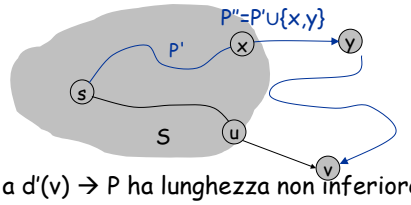
- Consideriamo il percorso di lunghezza d'(v), cioé quello formato dal percorso più corto da s ad u più l'arco (u,v)
- Consideriamo un **qualsiasi** altro percorso P da s a v. Dimostriamo che P non è più corto di d'(v)
- Sia (x,y) il primo arco di P che esce da S.
- Sia P' il sottocammino di P fino a x.
- P' più l'arco (x,y) ha già lunghezza non più piccola di d'(v) altrimenti l'algoritmo non avrebbe scelto v.



80

Algortimo di Dijksta: Correttezza

- Dimostriamo che il percorso P'' formato da P' più l'arco (x,y) ha già lunghezza non più piccola di d'(v).
- $\ell(P') + \ell((x,y)) \geq d(x) + \ell((x,y))$ siccome d(x) è per ipotesi induttiva uguale alla lunghezza del percorso più corto da s a x
- $d(x) + \ell((x,y)) \geq d'(y)$ siccome $d'(y) = \min_{(u,y):\, u \in S} d(u) + \ell((u,y))$
- $d'(y) \geq d'(v)$ altrimenti alla k-esima iterazione l'algoritmo non avrebbe inserito v in S



- P'' ha lunghezza non inferiore a d'(v) → P ha lunghezza non inferiore a d'(v)
  - l'implicazione è vera perchè P'' è il sottocammino di P cha va da s a y e il sottocammino di P che va da y a v non può avere costo negativo.

81

Algoritmo di Dijkstra: analisi tempo di esecuzione

```
Dijkstra's Algorithm (G, ℓ)
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.

82

Algoritmo di Dijkstra: come renderlo più efficiente?

```
Dijkstra's Algorithm (G, ℓ)
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 ∉ 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 ∉ 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∉ 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.

83

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

```
Dijkstra's Algorithm (G,s,ℓ)
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 m volte

- Tempo inizializzazione: tempo per effettuare gli n inserimenti in Q
- Tempo While: $O(n)$ più il tempo per fare le n ExtractMin e le al più m changeKey

84

84

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: $O(n)$ più il tempo per fare le n ExtractMin e le m changeKey

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 m ChangeKey
Tempo algoritmo: $O(n^2)$

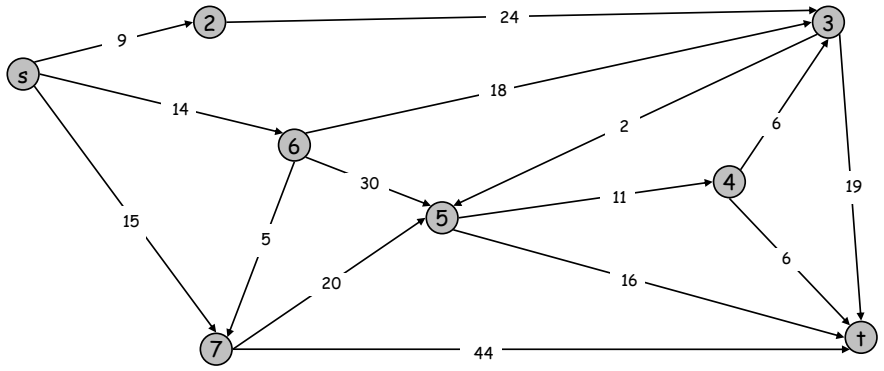
Se la coda è implementata mediante un heap binario:
Inizializzazione: $O(n)$ con costruzione bottom up oppure $O(n \log n)$ con n inserimenti
While: $O(n \log n)$ per le n ExtractMin; $O(m \log n)$ per le m ChangeKey
Tempo algoritmo: $O(n \log n + m \log n)$

85

85

Algoritmo di Dijkstra

Trova i percorsi più corti

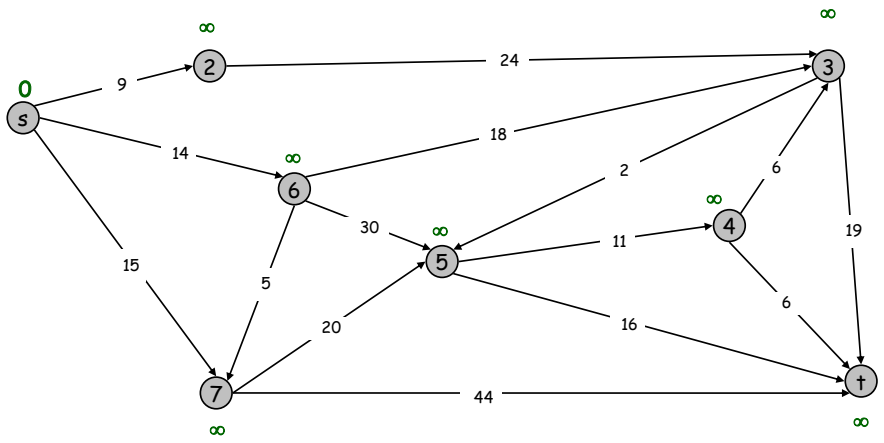


86

86

Algoritmo di Dijkstra

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

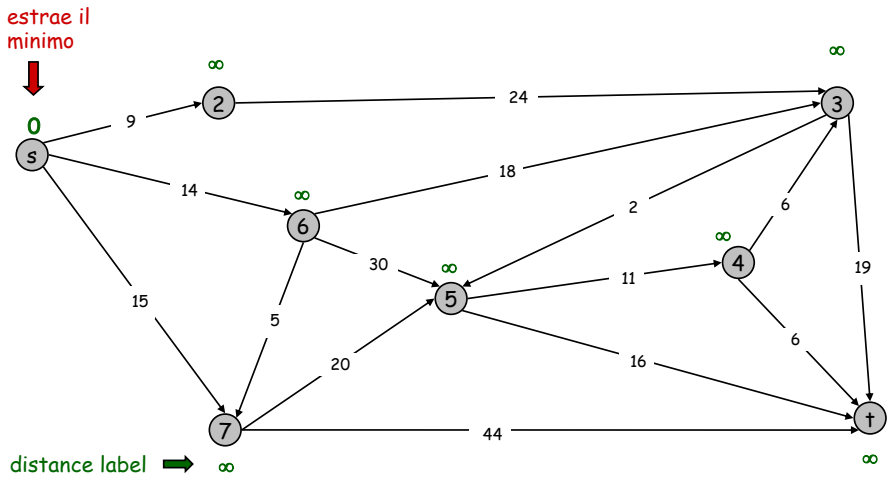


87

87

Algoritmo di Dijkstra

S = { }
Q = { (0,s), (∞,2), (∞,3), (∞,4),(∞,5), (∞,6), (∞,7), (∞,t) }

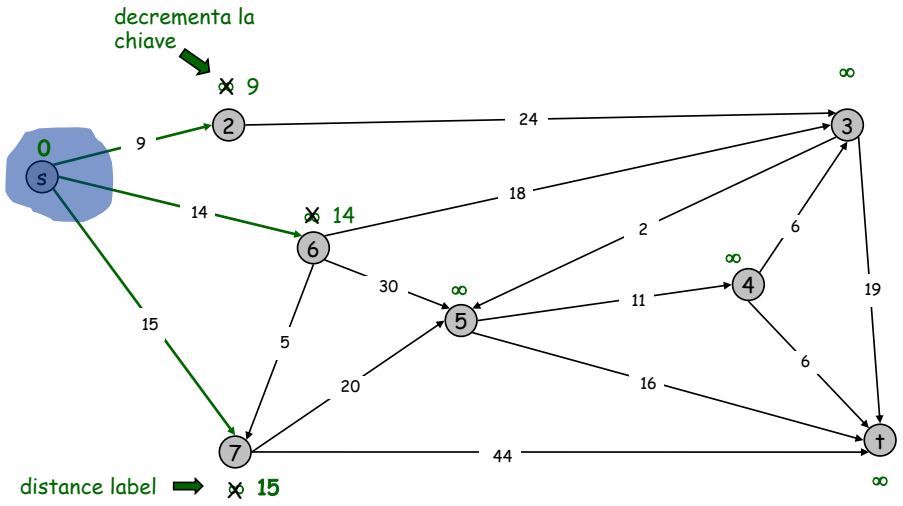


88

88

Algoritmo di Dijkstra

S = {s}
Q = { (**9**,2), (∞,3), (∞,4),(∞,5), (**14**,6), (**15**,7), (∞,t) }

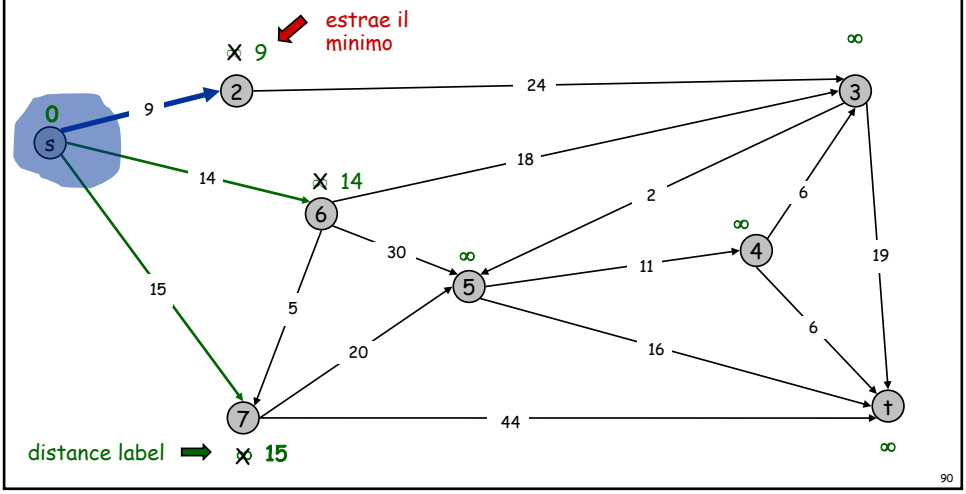


89

89

Algoritmo di Dijkstra

S = {*s*}
Q = { (∞,3), (∞,4),(∞,5), (14,6), (15,7), (∞,t) }

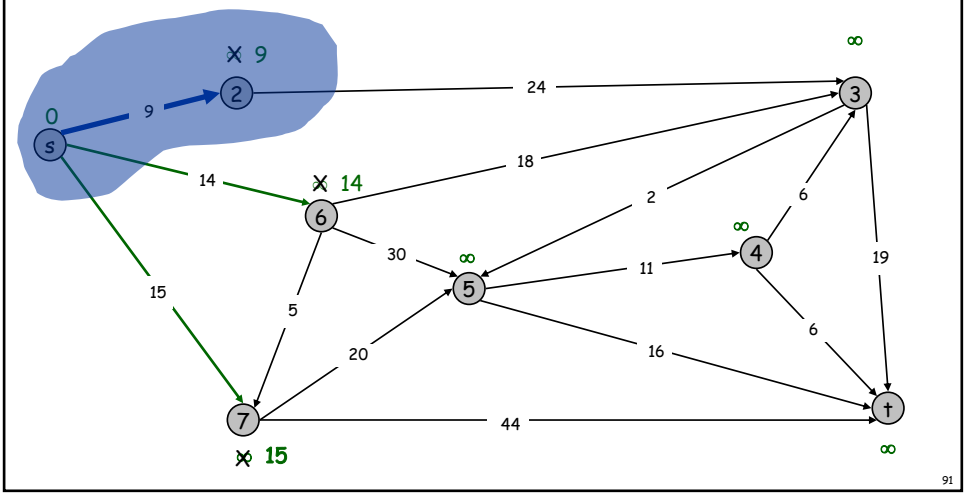


90

90

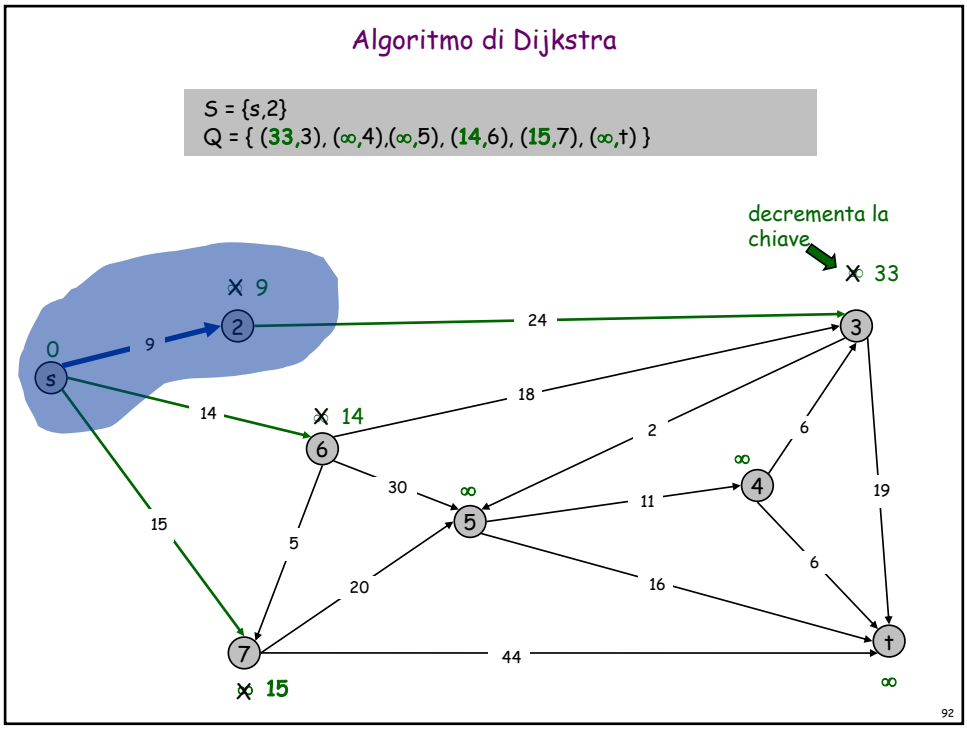
Algoritmo di Dijkstra

S = {*s*,2}
Q = { (∞,3), (∞,4),(∞,5), (14,6), (15,7), (∞,t) }

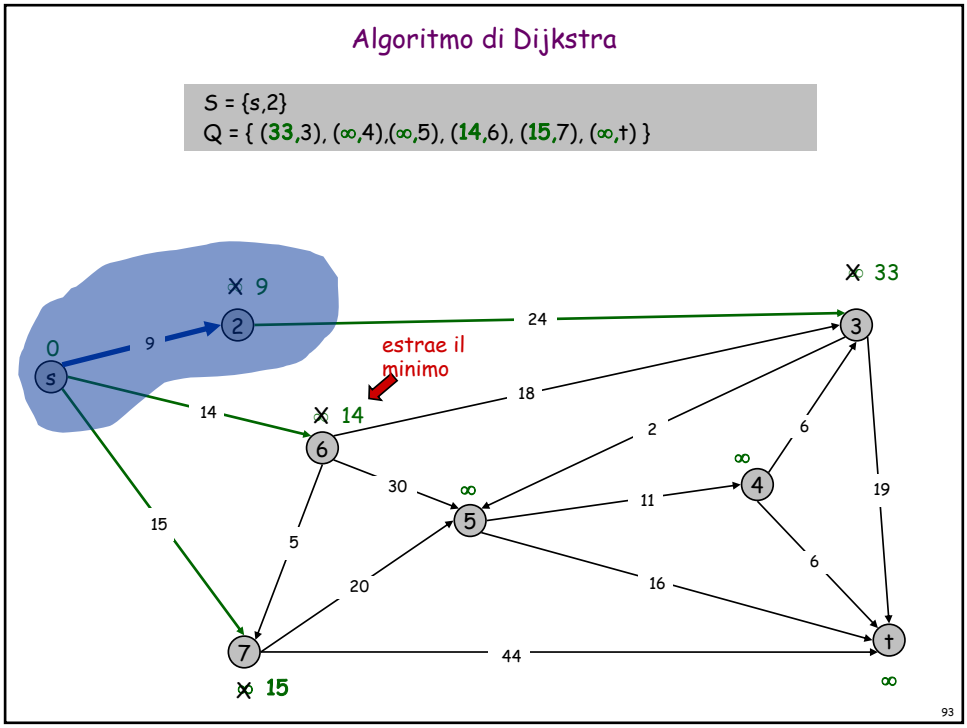


91

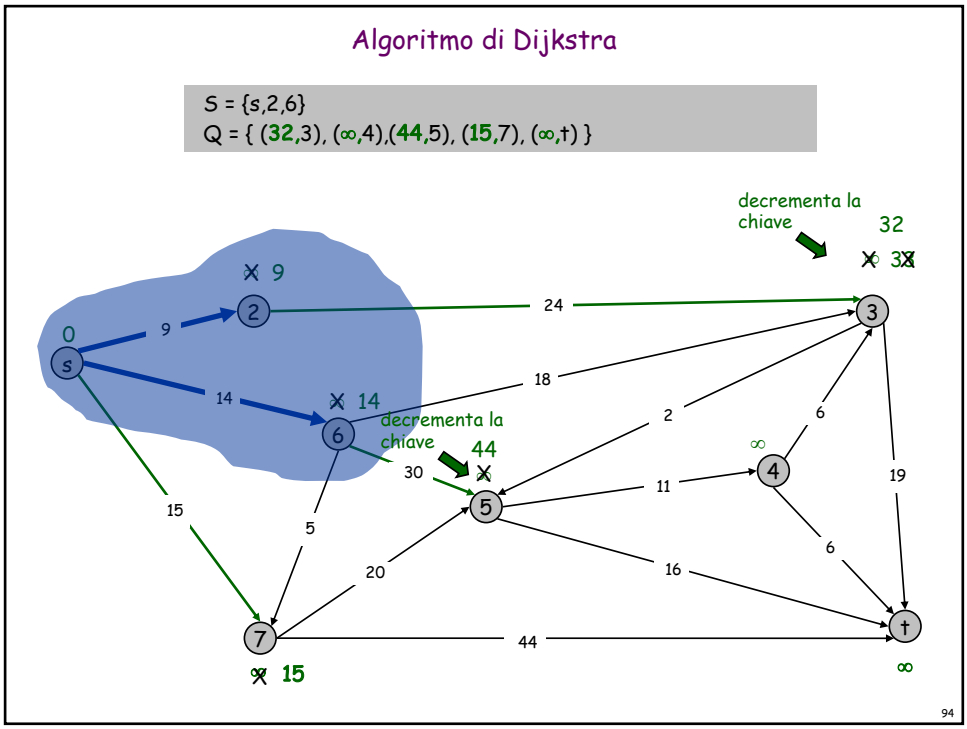
91


Algoritmo di Dijkstra
S = {s,2}
Q = { (33,3), (∞,4),(∞,5), (14,6), (15,7), (∞,t) }
decrementa la chiave
9
33
14
15
0
s
2
3
4
5
6
7
t
24
18
14
2
6
30
∞
11
19
15
5
20
16
6
44
∞
92

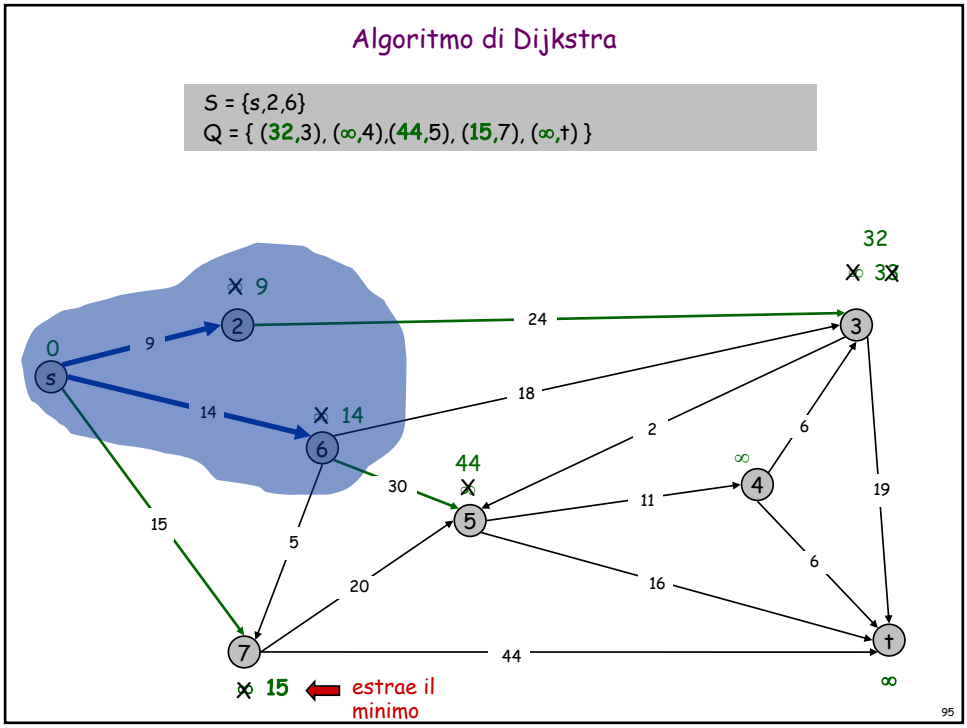
92


Algoritmo di Dijkstra
S = {s,2}
Q = { (33,3), (∞,4),(∞,5), (14,6), (15,7), (∞,t) }
estrae il minimo
9
33
14
15
0
s
2
3
4
5
6
7
t
24
18
14
2
6
30
∞
11
19
15
5
20
16
6
44
∞
93

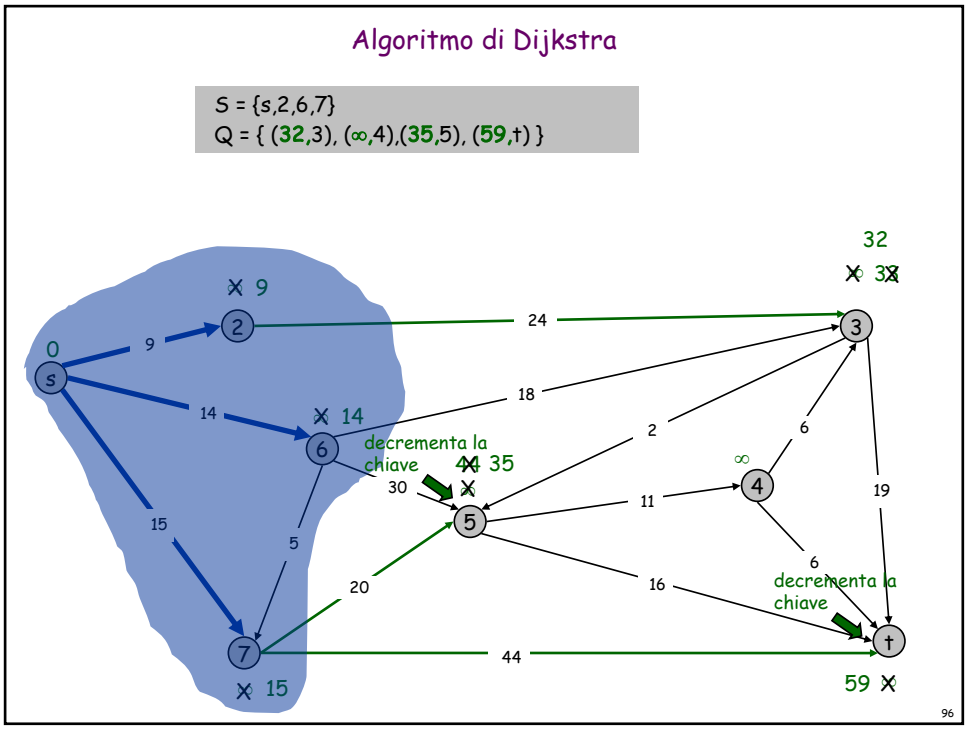
93


Algoritmo di Dijkstra
S = {s,2,6}
Q = { (32,3), (∞,4),(44,5), (15,7), (∞,t) }
decrementa la chiave
32
decrementa la chiave
44
9
14
15
∞
94

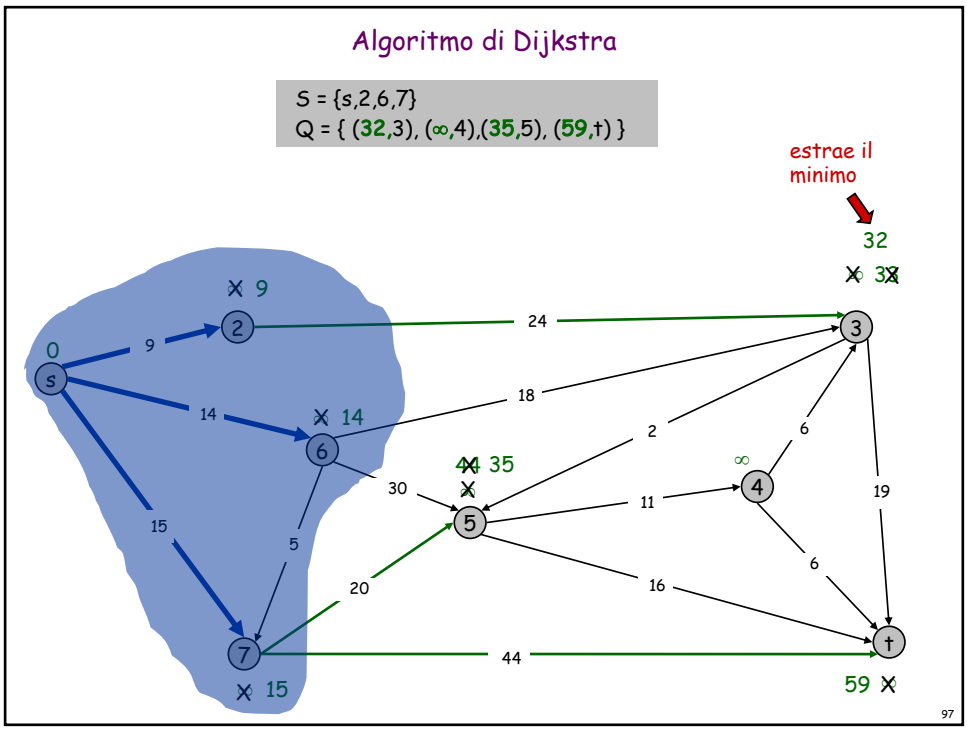
94


Algoritmo di Dijkstra
S = {s,2,6}
Q = { (32,3), (∞,4),(44,5), (15,7), (∞,t) }
32
9
14
15
∞
estrae il minimo
95

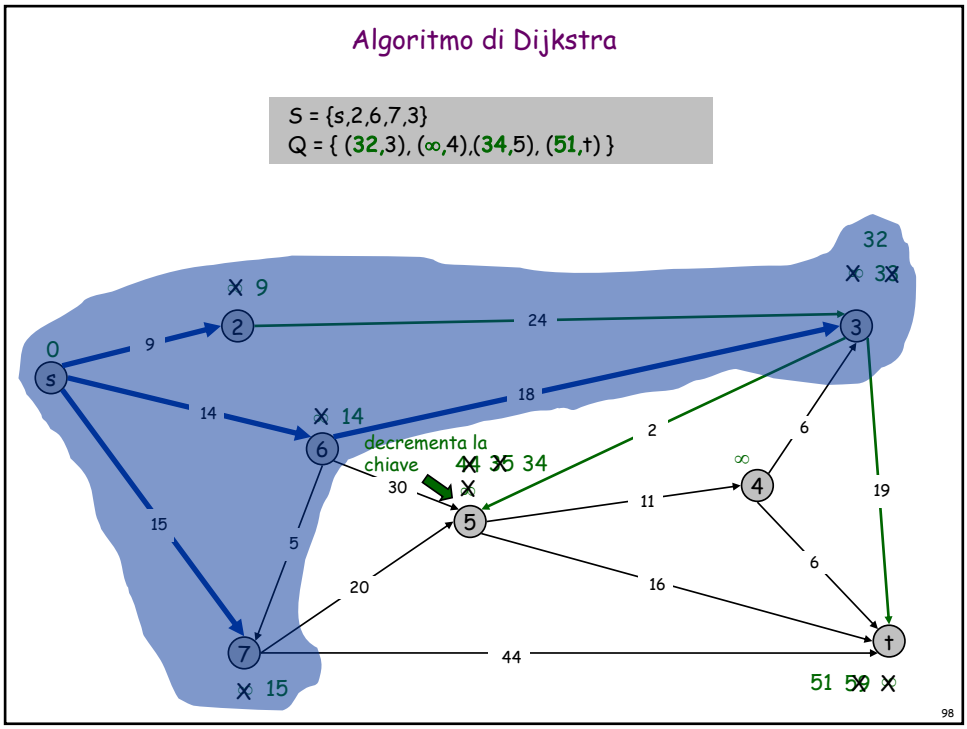
95


Algoritmo di Dijkstra
S = {s,2,6,7}
Q = { (32,3), (∞,4),(35,5), (59,t) }
0
s
9
2
24
3
32
14
6
18
2
6
∞
4
19
decrementa la chiave
35
30
5
11
15
5
6
20
16
decrementa la chiave
7
44
t
15
59
96

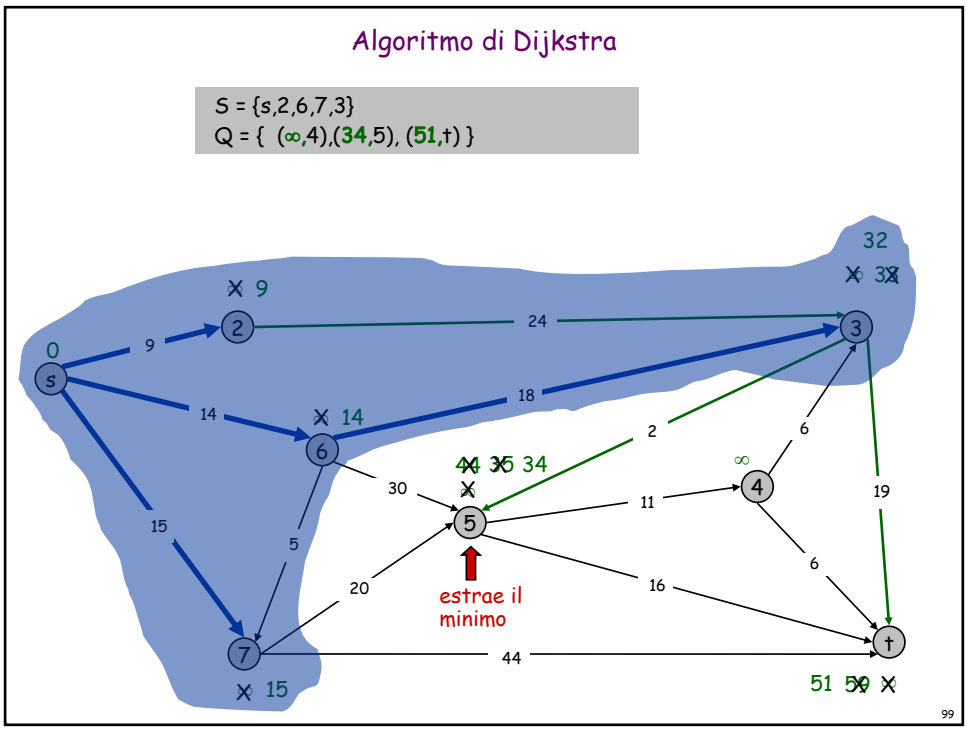
96


Algoritmo di Dijkstra
S = {s,2,6,7}
Q = { (32,3), (∞,4),(35,5), (59,t) }
estrae il minimo
0
s
9
2
24
3
32
14
6
18
2
6
∞
4
19
35
30
5
11
15
5
6
20
16
7
44
t
15
59
97

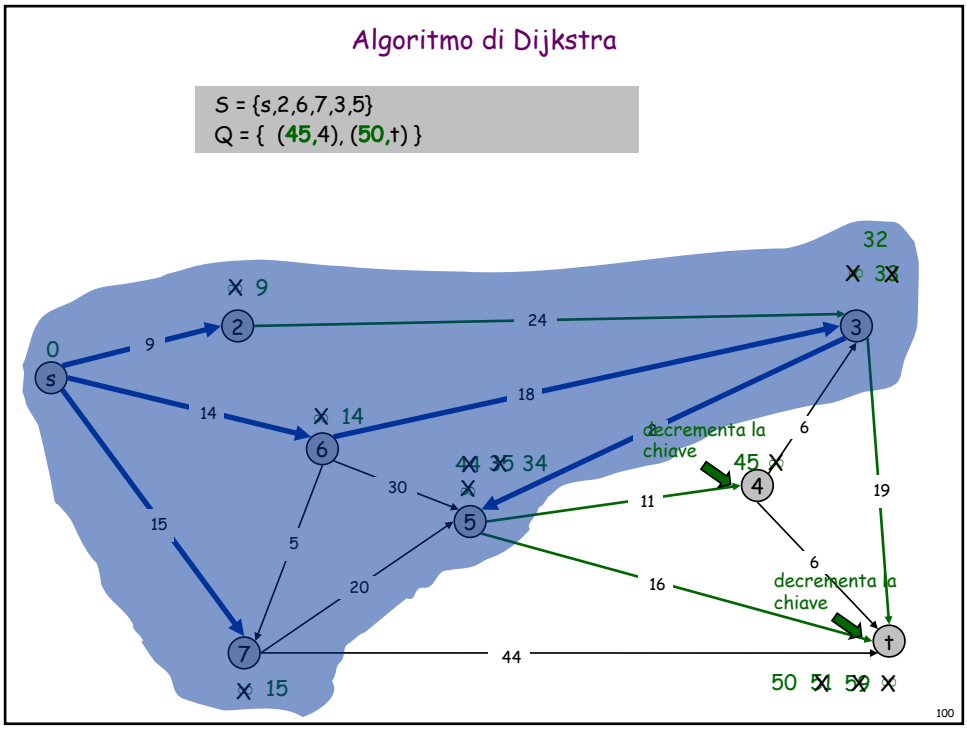
97


Algoritmo di Dijkstra
S = {s,2,6,7,3}
Q = { (32,3), (∞,4),(34,5), (51,t) }
0
s
9
2
24
3
14
6
18
decrementa la chiave
30
5
2
6
∞
4
11
19
15
5
20
16
6
7
44
t
15
51
98

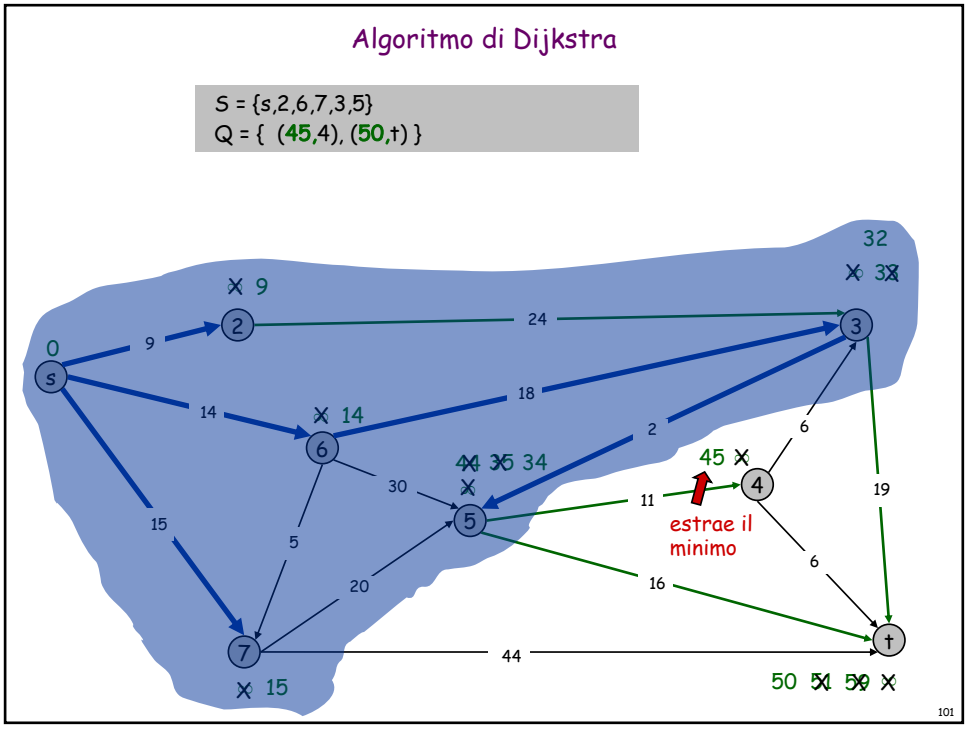
98


Algoritmo di Dijkstra
S = {s,2,6,7,3}
Q = { (∞,4),(34,5), (51,t) }
0
s
9
2
24
3
14
6
18
30
5
2
6
∞
4
11
19
15
5
20
estrae il minimo
16
6
7
44
t
15
51
99

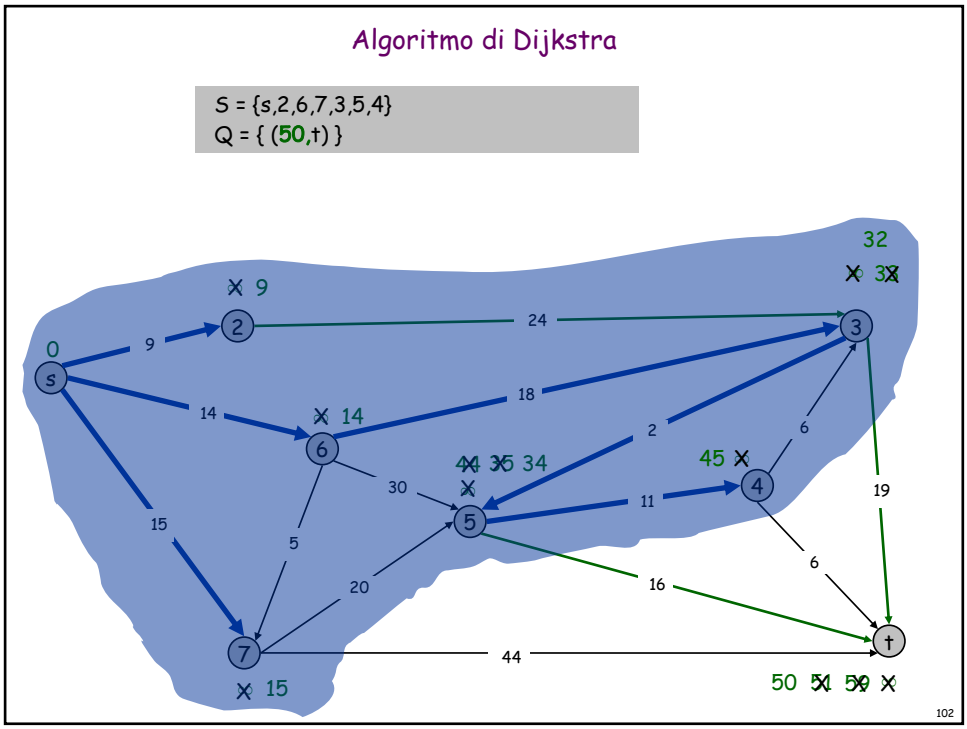
99


Algoritmo di Dijkstra
S = {s,2,6,7,3,5}
Q = { (45,4), (50,t) }
0
s
9
2
24
3
32
14
6
18
14
30
5
34
4
45
11
19
6
decrementa la chiave
15
5
20
16
decrementa la chiave
7
44
t
15
50
100

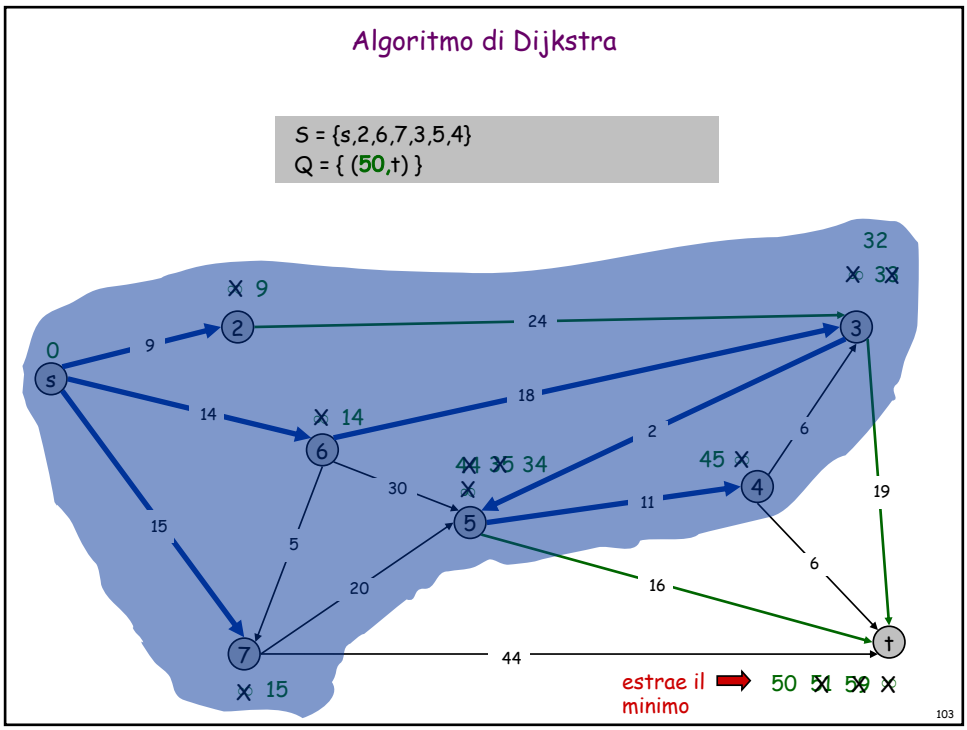
100


Algoritmo di Dijkstra
S = {s,2,6,7,3,5}
Q = { (45,4), (50,t) }
0
s
9
2
24
3
32
14
6
18
14
30
5
34
2
4
45
11
19
6
estrae il minimo
15
5
20
16
7
44
t
15
50
101

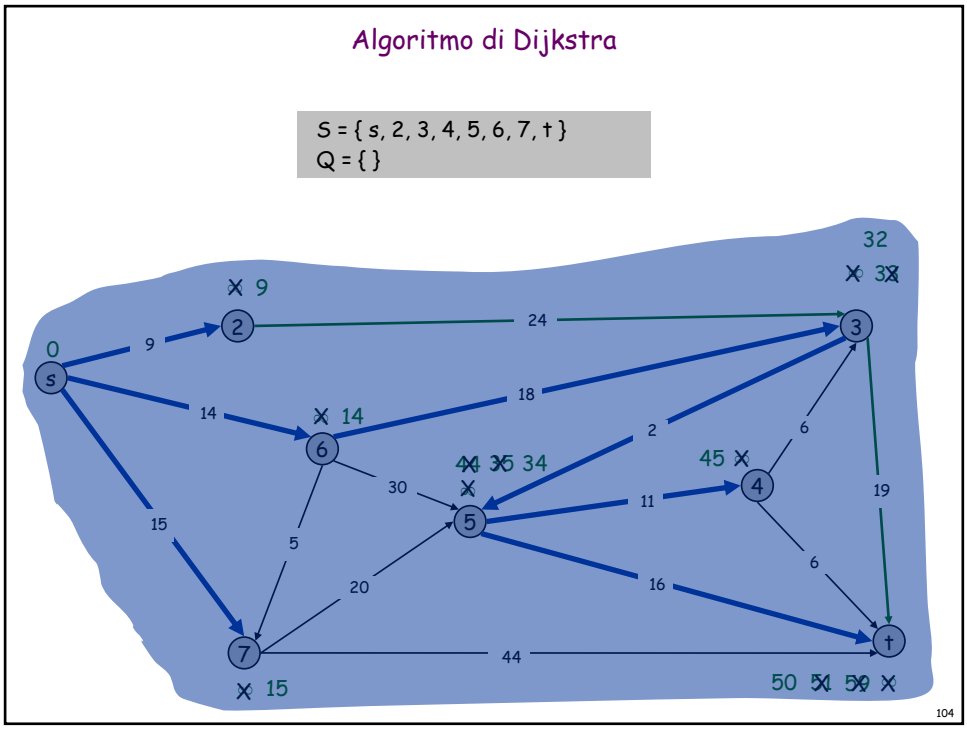
101


Algoritmo di Dijkstra
S = {s,2,6,7,3,5,4}
Q = { (50,t) }
0
s
9
2
24
3
14
6
18
2
6
45
4
19
30
5
11
15
5
20
16
6
7
44
t
15
50
102

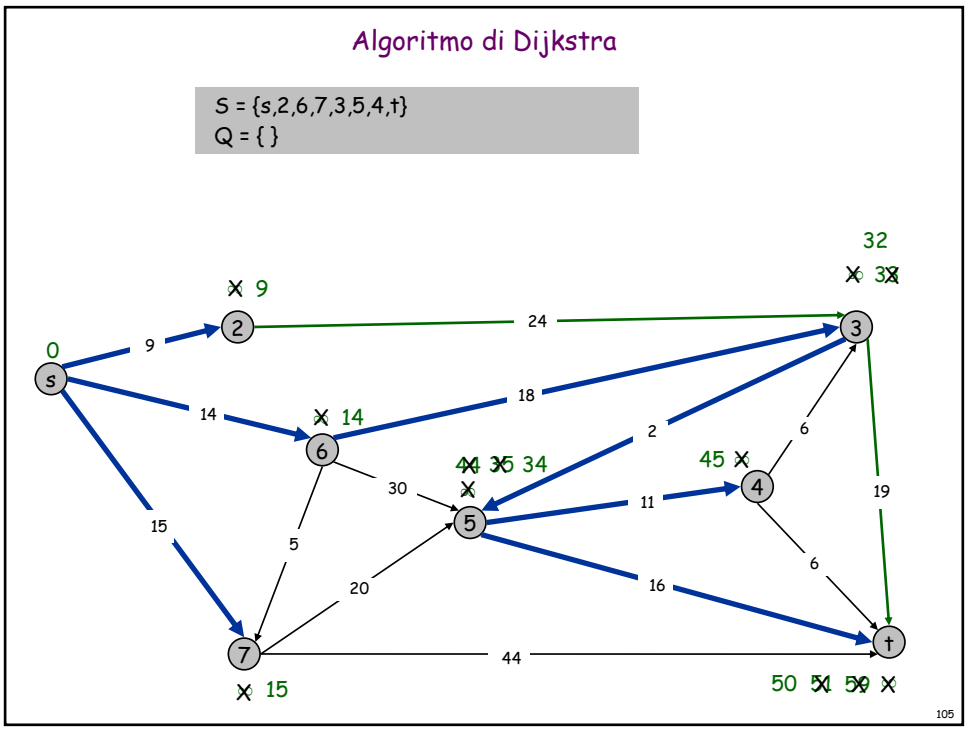
102


Algoritmo di Dijkstra
S = {s,2,6,7,3,5,4}
Q = { (50,t) }
0
s
9
2
24
3
14
6
18
2
6
45
4
19
30
5
11
15
5
20
16
6
7
44
t
15
estrae il
minimo
50
103

103


Algoritmo di Dijkstra
S = { s, 2, 3, 4, 5, 6, 7, t }
Q = { }
0
s
9
2
24
3
32
14
6
18
2
6
45
4
19
30
5
11
15
5
20
16
6
7
44
t
15
50
104

104


Algoritmo di Dijkstra
S = {s,2,6,7,3,5,4,t}
Q = { }
0
s
9
2
24
3
32
14
6
18
2
6
45
4
19
30
5
11
15
5
20
16
6
7
44
t
15
50
105

105