Algoritmi greedy
II parte

Progettazione di Algoritmi a.a. 2015-16

Matricole congrue a 1

Docente: Annalisa De Bonis

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.

-

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 vendono rese note all'inizio). Non molto realistico!
- Assumiamo che inizialmente la cache sia piena, cioè contenga k elementi

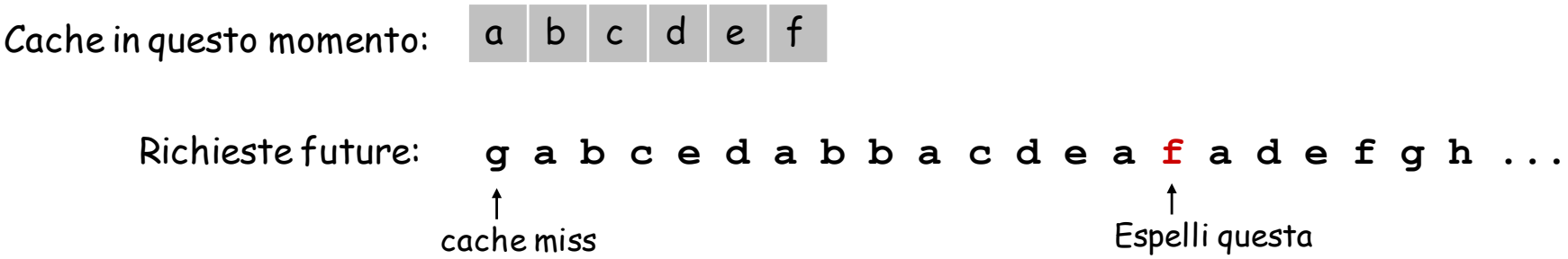
Def. Un eviction schedule è uno scheduling degli elementi da espellere, cioè una sequenza che indica quale elemento espellere quando c'è bisogno di far posto ad un elemento richiesto che non è in cache.

Obiettivo. Un eviction schedule che minimizzi il numero di cache miss.

.

caching offline ottimale: Farthest-In-Future

Farthest-in-future. Espelli dalla cache un elemento che non è richiesto se non dopo molto tempo



Teorema. [Bellady, 1960s] Farthest-in-future è uno schedule ottimo.

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

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

Algoritmo di Belady basato sulla strategia Farthest in Future (FF)

```
Assume the requests d1, d2, ... , dn are arranged in ascending order
of arrival time
For each element d, let L[d] list of j s.t.dj=d
Let Q a priority queue
for j = 1 to n {
  if(list L[dj] is empty and j is in the cache)
      insert (j, dj) to priority queue Q
  append j to list L[dj]
}
for j = 1 to n {
   if (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]
       p ← first element of L[dj]
       insert (p,dj) in Q
  }
```

O(n+k log k)
k = dimensione cache

O(n log k)
k = dimensione cache

Tempo O(n log k) se

- ad ogni elemento è associato un flag che è true se e solo l'elemento è in cache
- usiamo un heap come coda a priorità
- Consideriamo costante il tempo per espellere e inserire ciascun elemento in cache

Eviction Schedule ridotto

Def. Un eviction schedule ridotto è uno scheduling che 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.

Farthest-in-future è un 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	b	c
a	a	b	c

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

Farthest-In-Future: ottimalità

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

- Ogni schedule può essere trasformato in uno schedule ridotto senza aumentare il numero di inserimenti in cache
- Ogni schedule ridotto può essere trasformato nello schedule FF senza aumentare il numero di cache miss
- Possiamo quindi trasformare uno scheduling ridotto ottimo nello scheduling FF senza aumentare il numero di cache miss. Ciò implica che FF va incontro allo stesso numero di cache miss dell'algoritmo ottimo ed è quindi anch'esso ottimo

- NB: uno schedule ottimo ridotto è di fatto uno schedule ottimo tra tutti i possibili schedule in quanto, dato un algoritmo ottimo, possiamo trasformarlo in uno schedule ridotto senza aumentare il numero di inserimenti in cache.

Eviction Schedule Ridotto

Affermazione. Un qualsiasi eviction schedule S può essere trasformato in un eviction schedule ridotto S' senza aumentare il numero di elementi inseriti 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. e inserisce d solo 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, dove j è un intero maggiore o uguale di 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.

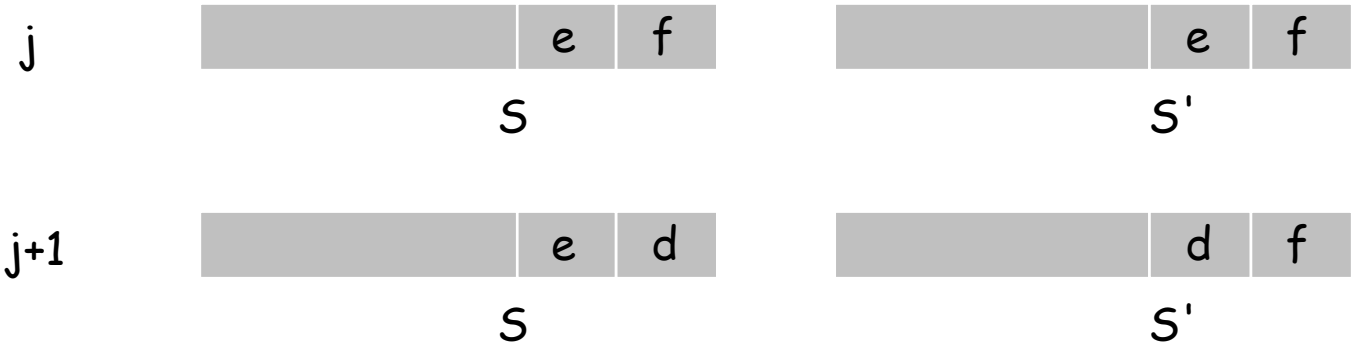
- Consideriamo la (j+1)-esima richiesta e sia $d = d_{j+1}$ l'elemento richiesto,
- Siccome S e S_{FF} hanno fatto le stesse scelte fino alla richiesta j-esima, quando arriva la richiesta (j+1)-esima il contenuto della cache per i due scheduling è lo stesso.
- Caso 1: d è gia nella cache. In questo caso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

Continua nella prossima slide

Farthest-In-Future: ottimalità

- Nei primi due casi basta porre S'=S visto che S ed S_{FF} hanno lo stesso comportamento anche per la (j+1)-esima richiesta.

- 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 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 determinino un numero di cache miss di S' maggiore di quello di S.

Continua nella prossima slide

Farthest-In-Future: ottimalità

Dopo la (j+1)-esima richiesta noi facciamo fare ad S' le stesse scelte di S fino a che ad un certo tempo j' accade per la prima volta uno dei casi illustrati nelle slide successive per cui non è possibile che S ed S' facciano la stessa scelta.

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'.

Indichiamo con g l'elemento richiesto al tempo j'.
I casi che avrebbero permesso ad S' di fare la stessa scelta di S sono:
g ≠ e, f ; g è presente nella cache di S: in questo caso g è presente anche nella cache di S' non fa niente come S.
g ≠ e, f ; g non è presente nella cache di S ; 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.

Continua nella prossima slide

Farthest-In-Future: ottimalità

Dobbiamo considerare quindi i 3 restanti casi:

Caso 3. 1. $g \neq e, f$; g non è né nella cache di S né in quella di S'; S espelle e.
Caso 3.2. $g = f$; S espelle e
Casi 3.3. $g = f$; S espelle $e' \neq e$.

E il caso $g = e$? Questo caso non è possibile perché S_{FF} ha espulso e al posto di f al tempo j+1 per cui la richiesta di $g = e$ deve essere preceduta da una richiesta di f . Se così fosse allora tra il tempo j+2 e il tempo j' si sarebbe verificata già una circostanza in cui S' non avrebbe potuto fare la stessa scelta di S (caso 3.2 o 3.3). Ciò contraddice il fatto che j' sia il primo tempo dopo j+1 in cui questo accade.

Continua nella prossima slide

Farthest-In-Future: ottimalità

Caso 3.1: $g \neq e, f$; g non è né nella cache di S né in quella di S' ; S espelle e.
In questo caso 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'.
Caso 3.2: $g = f$; 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.
Caso 3.3: $g = f$; S espelle $e' \neq e$. In questo caso e' è presente anche nella cache di S' . Facciamo in modo che S' espella e' al tempo j'. Da questo momento in poi la cache di S e quella di S' hanno lo stesso contenuto. In numero di cache miss dei due scheduling è lo stesso. Il teorema non è ancora dimostrato per questo caso in quanto S' non è ridotto. Dall'affermazione dimostrata precedentemente, possiamo rendere S' ridotto senza aumentare il numero di inserimenti. In questo scheduling ridotto il numero di cache miss è uguale al numero di cache miss di S.

Farthest-In-Future: ottimalità

- Teorema. Farthest-in-future produce un eviction schedule S_{FF} ottimo.
- Dim.
- Consideriamo un eviction schedule ridotto ottimo 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 va incontro allo stesso numero di cache miss di S*.
- Applicando il teorema precedente 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 va incontro allo stesso numero di cache miss di S_1 e quindi di S*.
- Continuiamo in questo modo applicando induttivamente il teorema precedente per j=1,2,…,m fino a che non arriviamo ad uno schedule S_m che effettua esattamente le stesse scelte di S_{FF} (S_m= S_{FF}) e va incontro allo stesso numero di cache miss di S^*. Si ha quindi che S_{FF} e S* vanno incontro allo stesso numero di cache miss e di conseguenza S* è ottimo

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'algortimo 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 e` indicata la distanza tra ciascuna coppia di intersezioni adiacenti
- Come si puo` individuare il percorso piu` breve da Napoli a Milano?

Cammini minimi

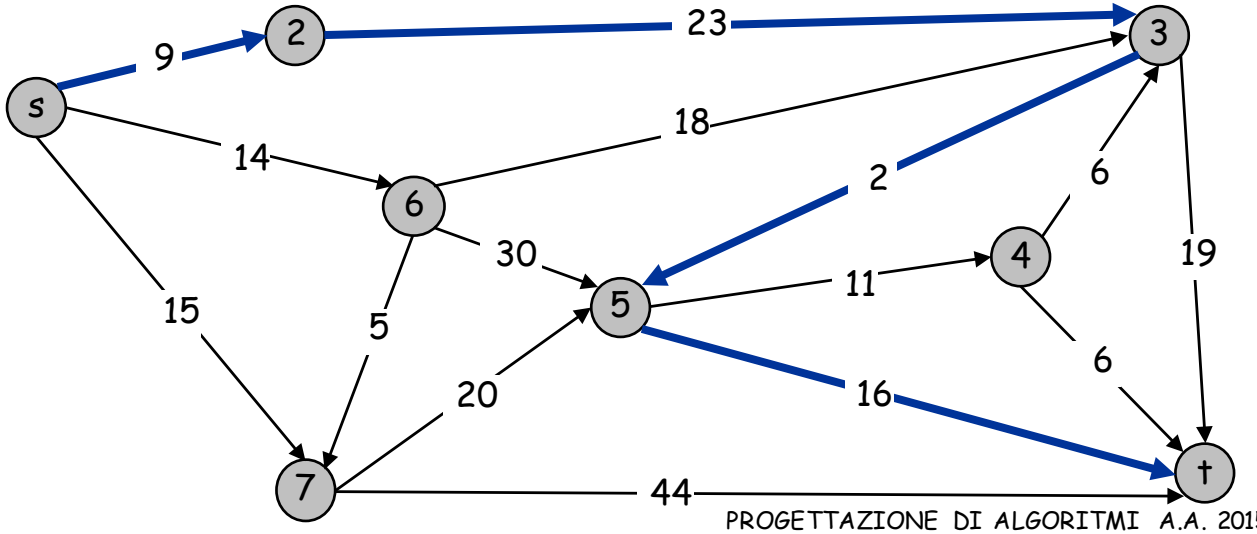
- Esempi di applicazioni dei cammini minimi in una rete
- Trovare il cammino di **tempo minimo** in una rete
- Se i pesi esprimono l'inaffidabilità delle connessioni in una rete, trovare il collegamento che è **più sicuro**

Il problema dei cammini minimi

- Input:
  - Grafo direzionato G = (V, E).
  - Per ogni arco e, ℓ_e = lunghezza del tratto rappresentato da 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 = 48.

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

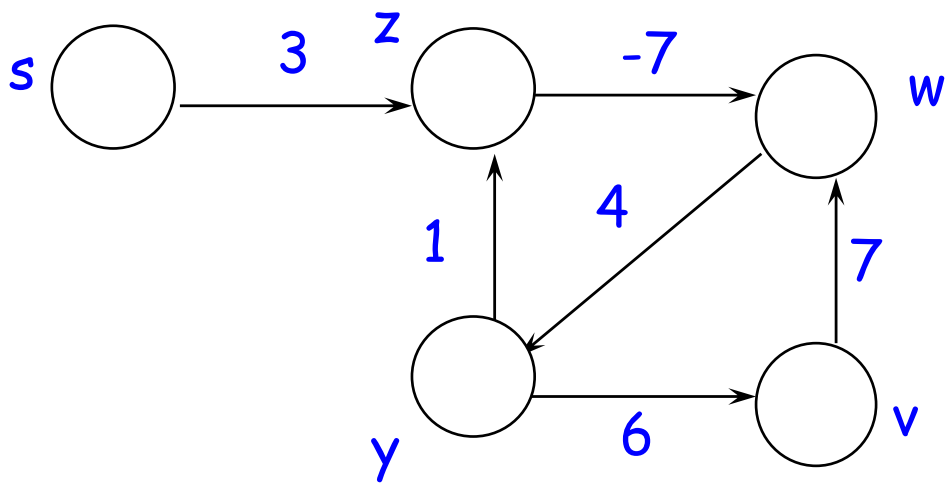
Cammini minimi

- 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



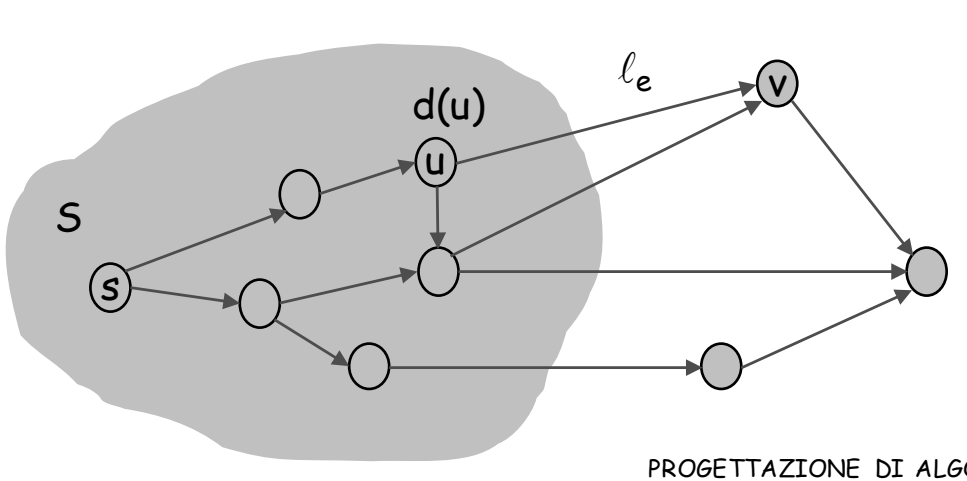
Il ciclo <z,w,y,z> ha peso -2

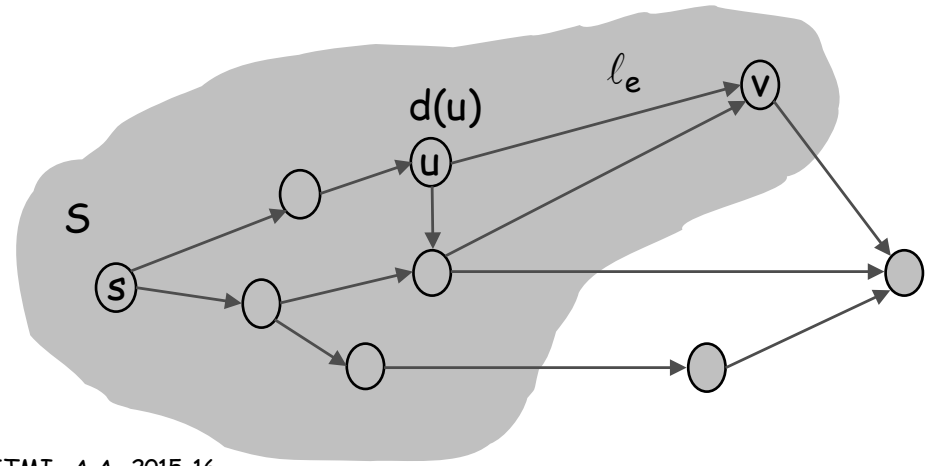
- Attraversando il ciclo <z,w,y,z> un numero arbitrario di volte possiamo trovare percorsi da s a v di peso arbitariamente piccolo

Algoritmo di Dijkstra

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).





Algoritmo di Dijkstra

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: 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.

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 $u \in S$ 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)

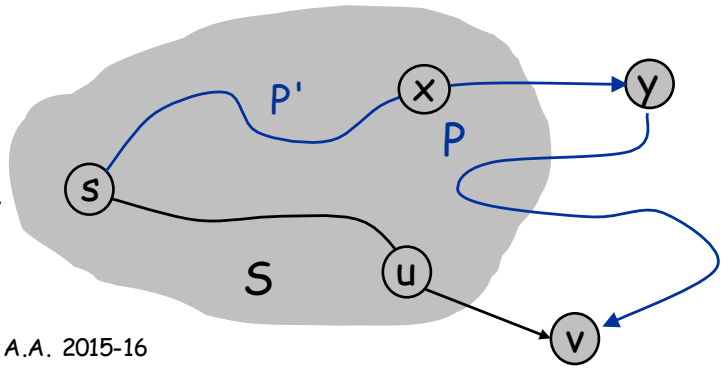
Base: |S| = 1. In questo caso S={s} e d(s)=0 per cui la tesi vale banalmente.

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 gia` lunghezza maggiore di d'[v] altrimenti l'algoritmo avrebbe scelto y.



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(u),u)<-- ExtractMin(Q)
  Add u to S
  For each edge e=(u,v)
   If  v not in S && d(u)+ ℓe< d'(v)
     ChangeKey(Q,v, d(u)+ ℓ(e))

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

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

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

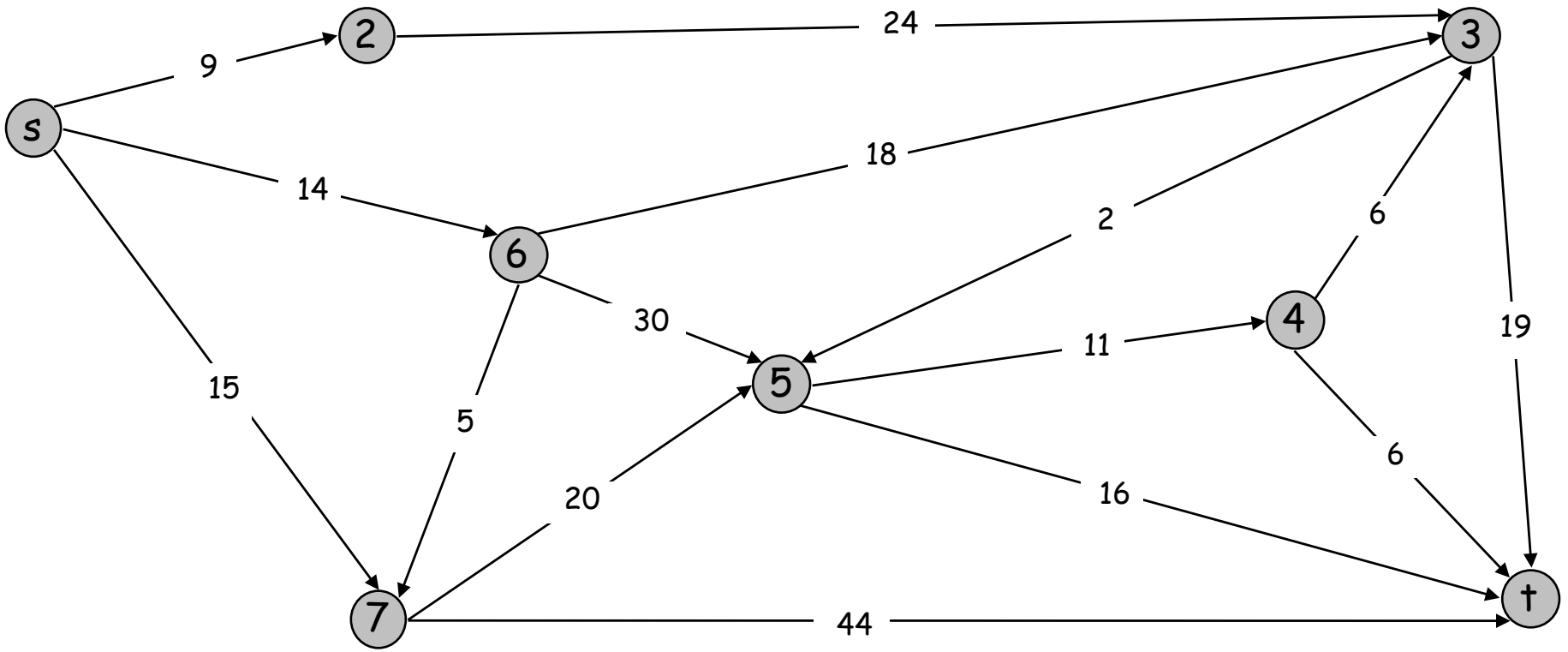
- Se usiamo una min priority queue che per ogni vertice v **non** in S contiene la coppia (y,d'[u]) allora con un'operazione di ExtractMin possiamo ottenere il vertice v con il valore d'[v] più piccolo possibile
- Tempo inizializzazione $O(n)$ più tempo per effettuare gli n inserimenti in Q
- Tempo While: $O(n)$ 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:
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: $O(n \log n + m \log n)$

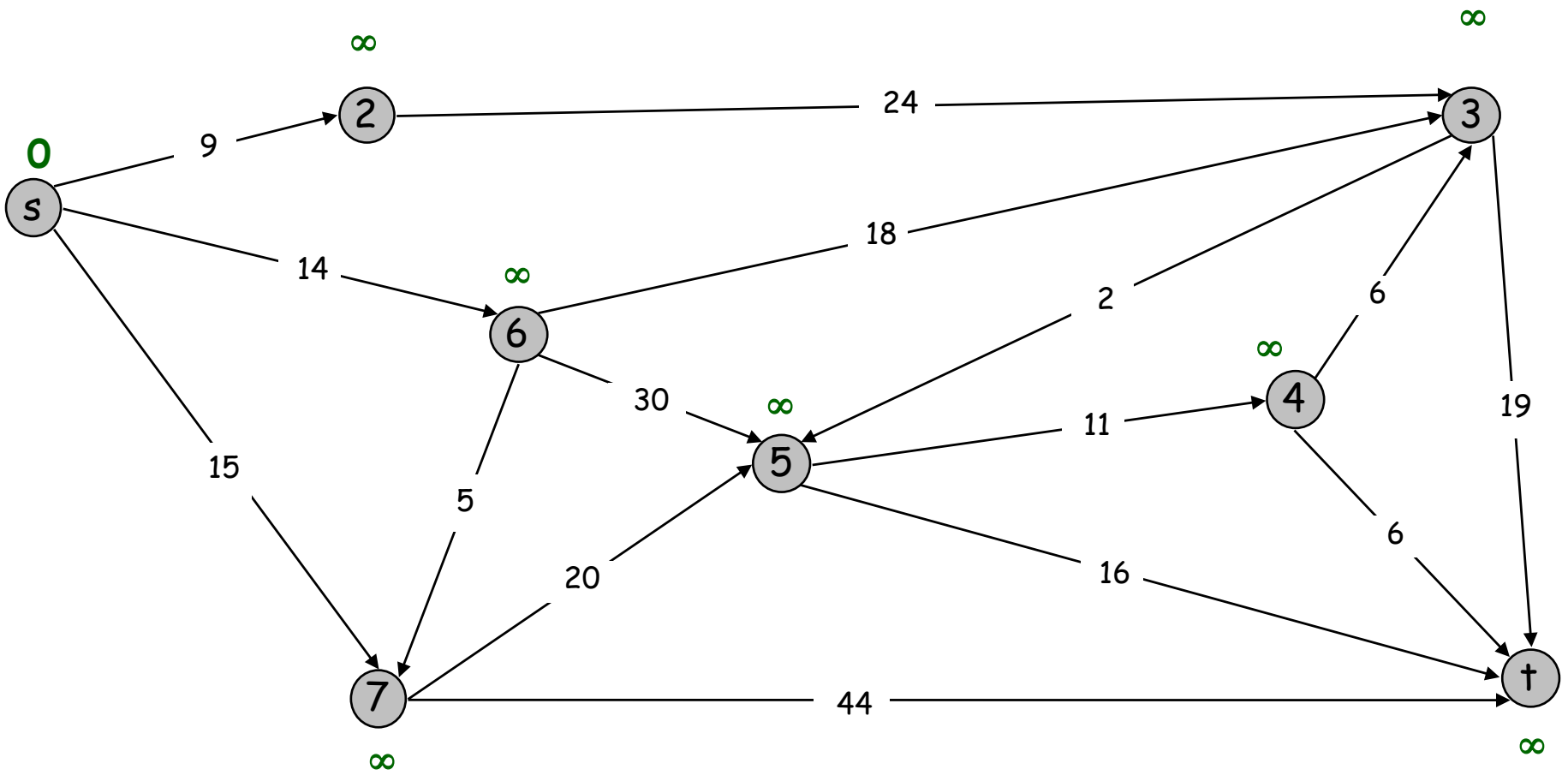
Algoritmo di Dijkstra

Trova i percorsi più corti



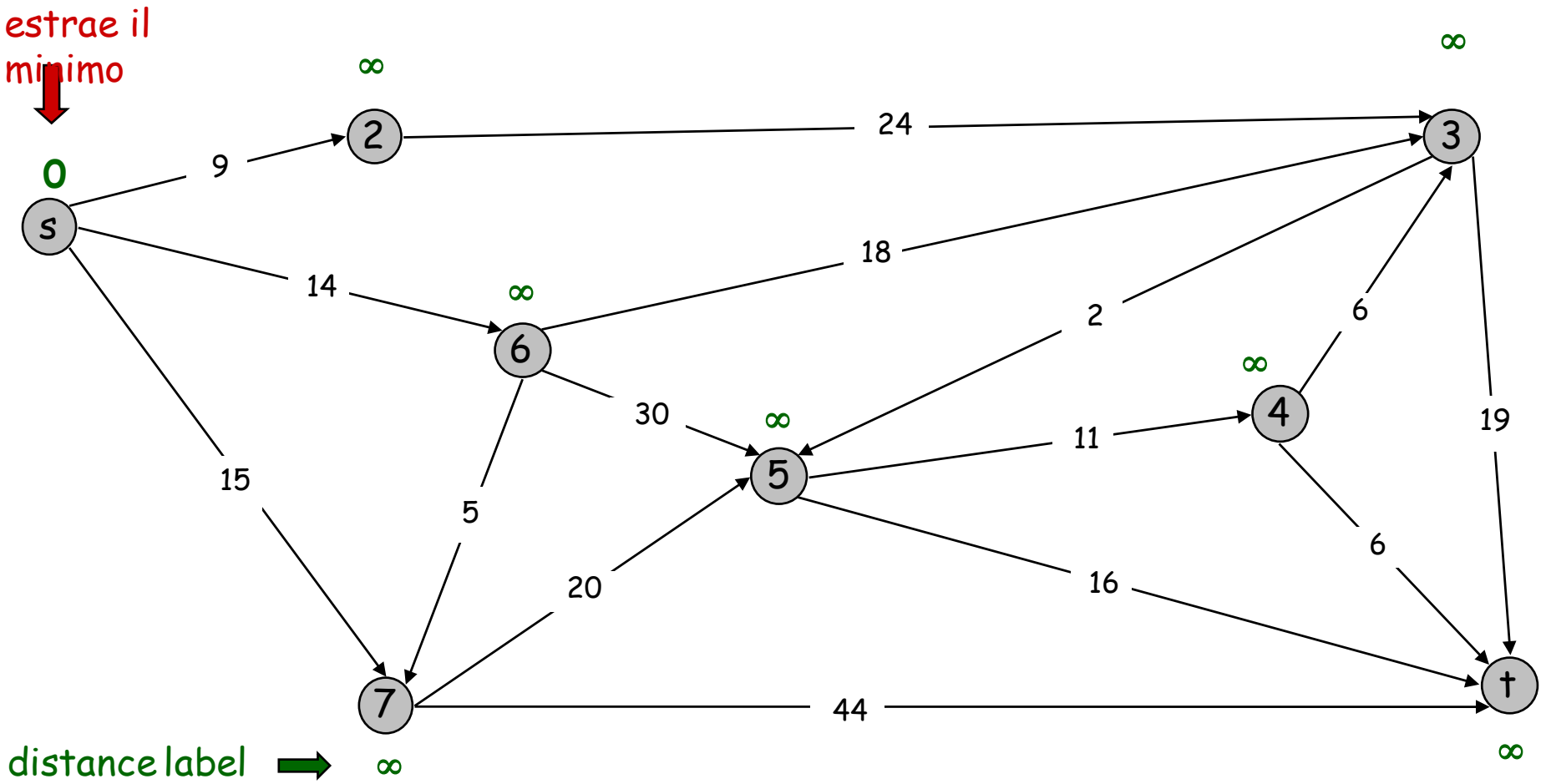
Algoritmo di Dijkstra

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



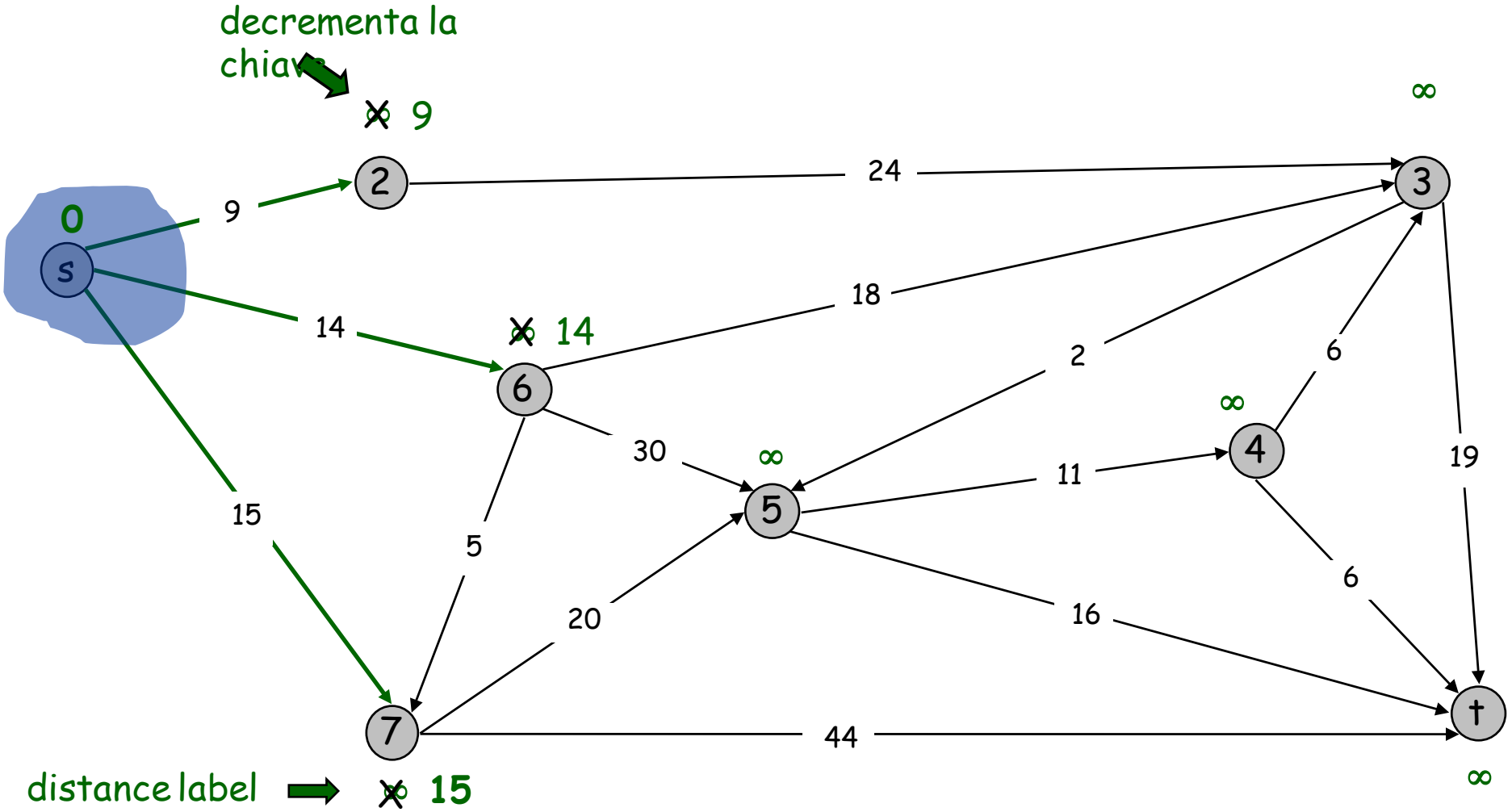
Algoritmo di Dijkstra

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



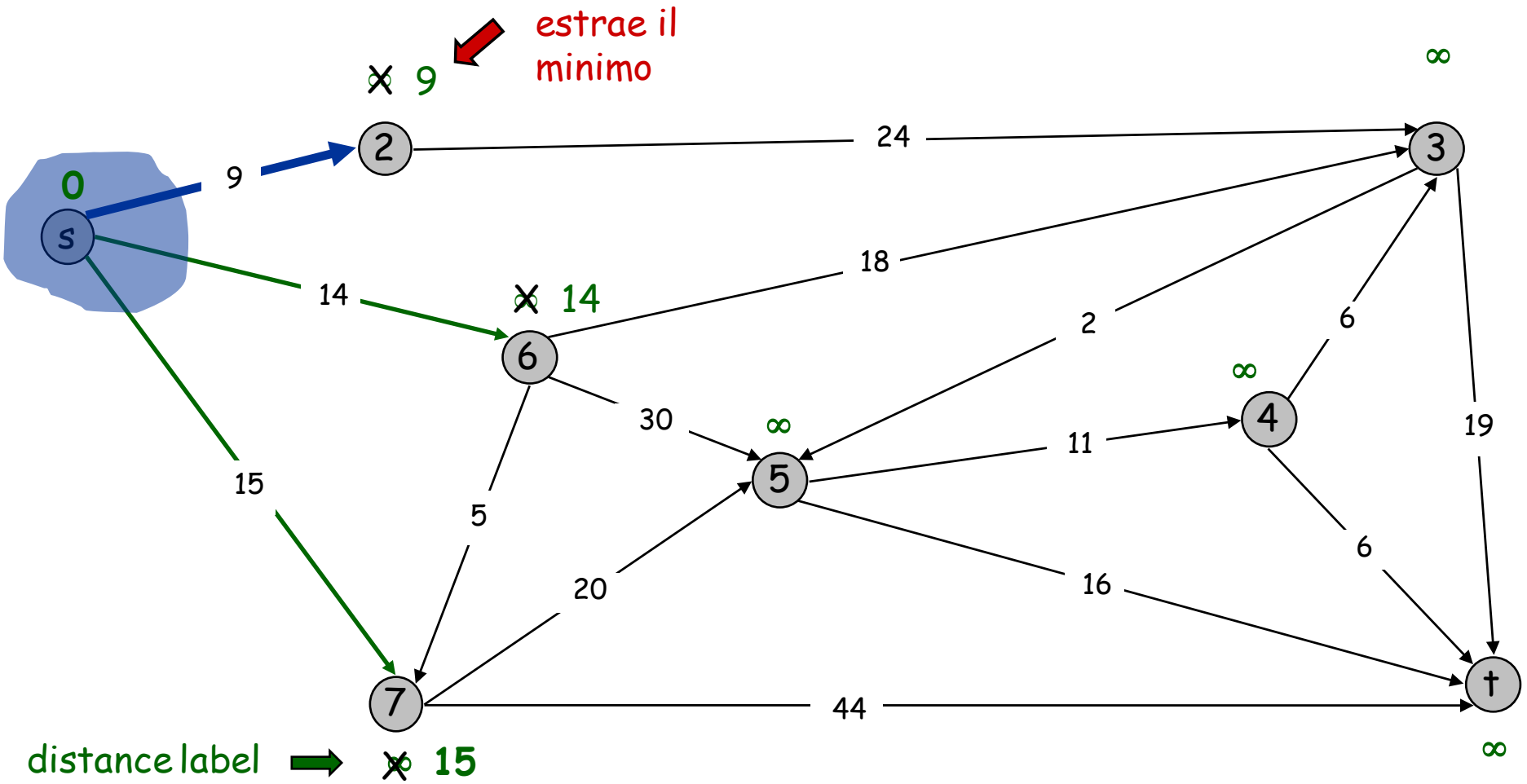
Algoritmo di Dijkstra

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



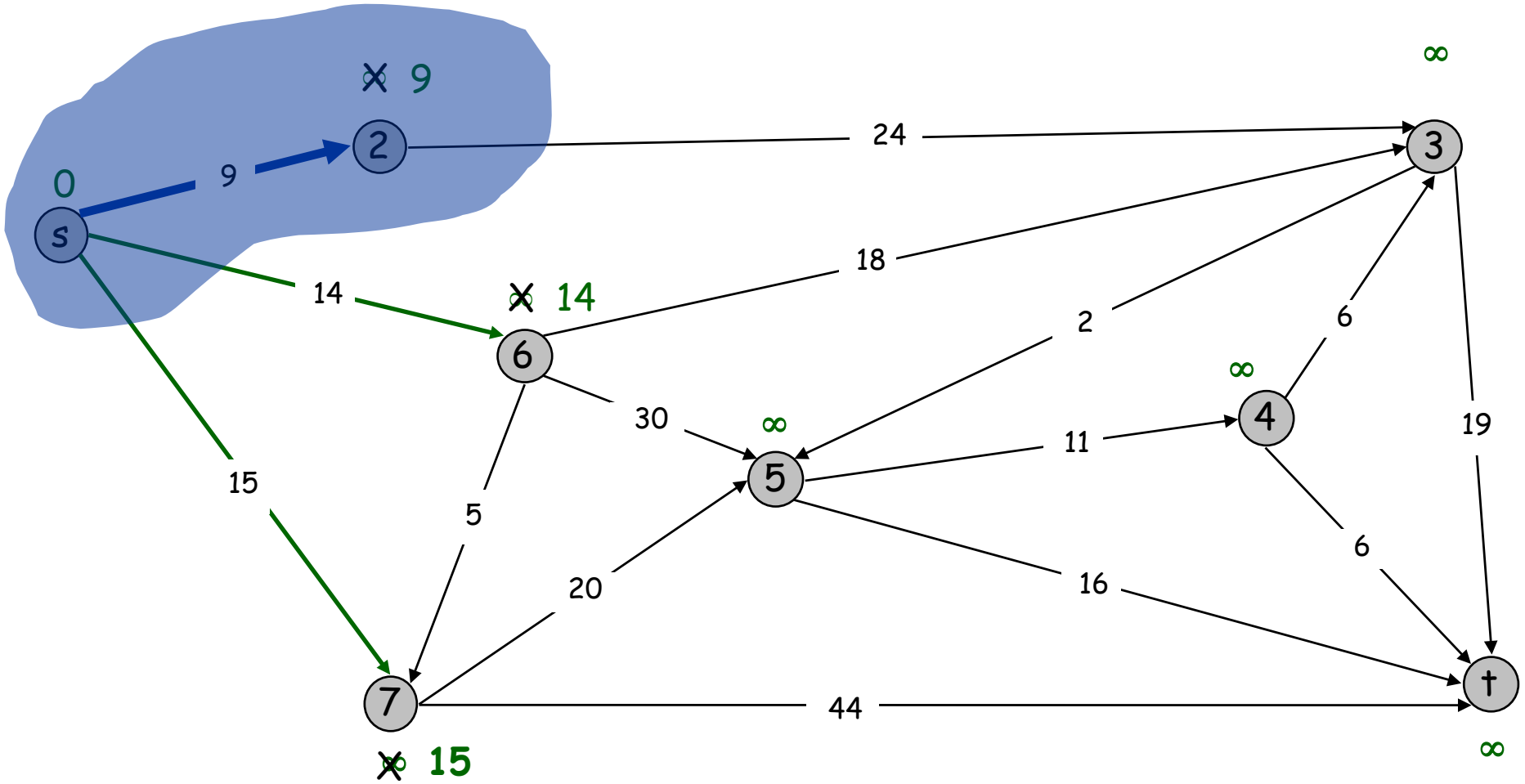
Algoritmo di Dijkstra

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



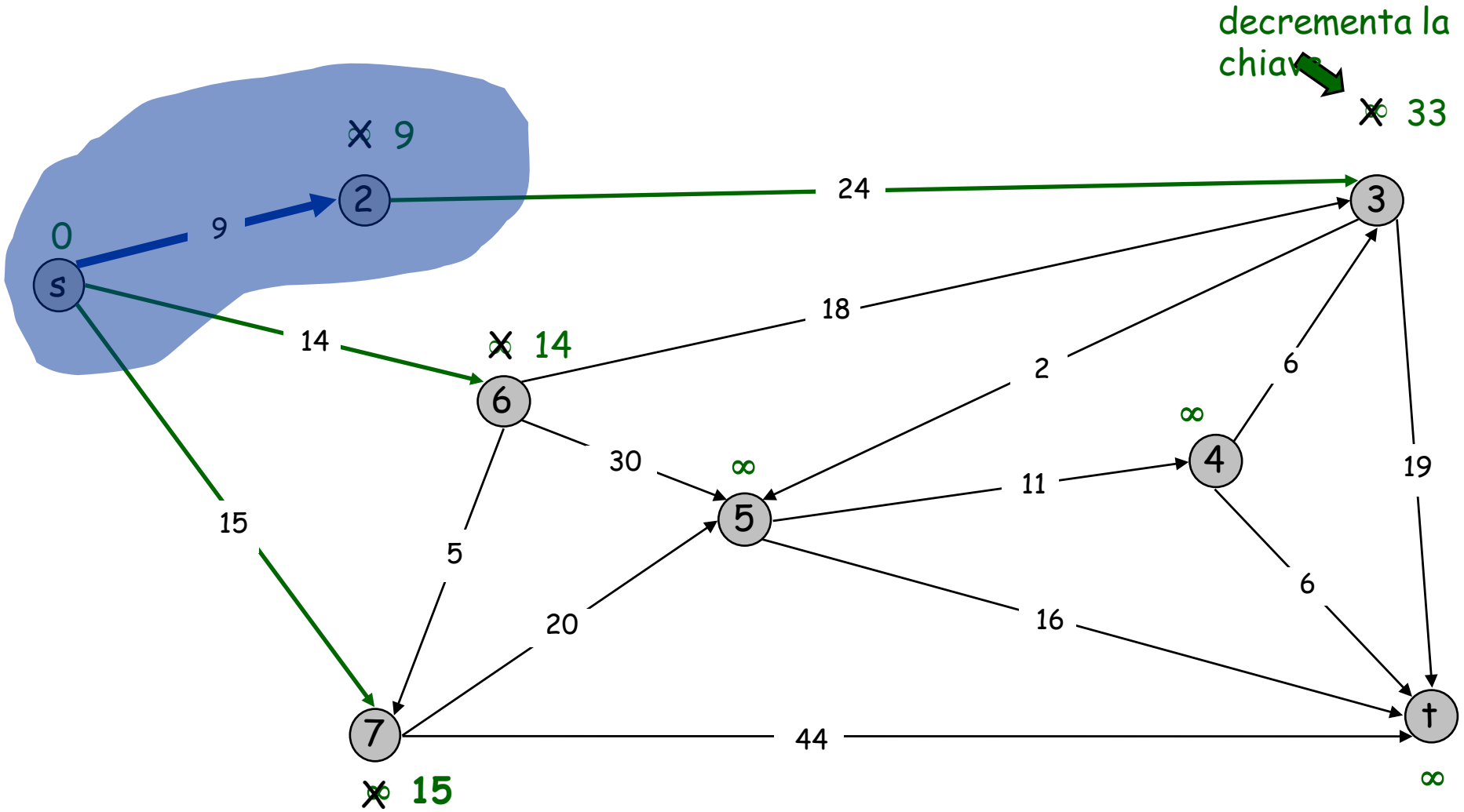
Algoritmo di Dijkstra

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



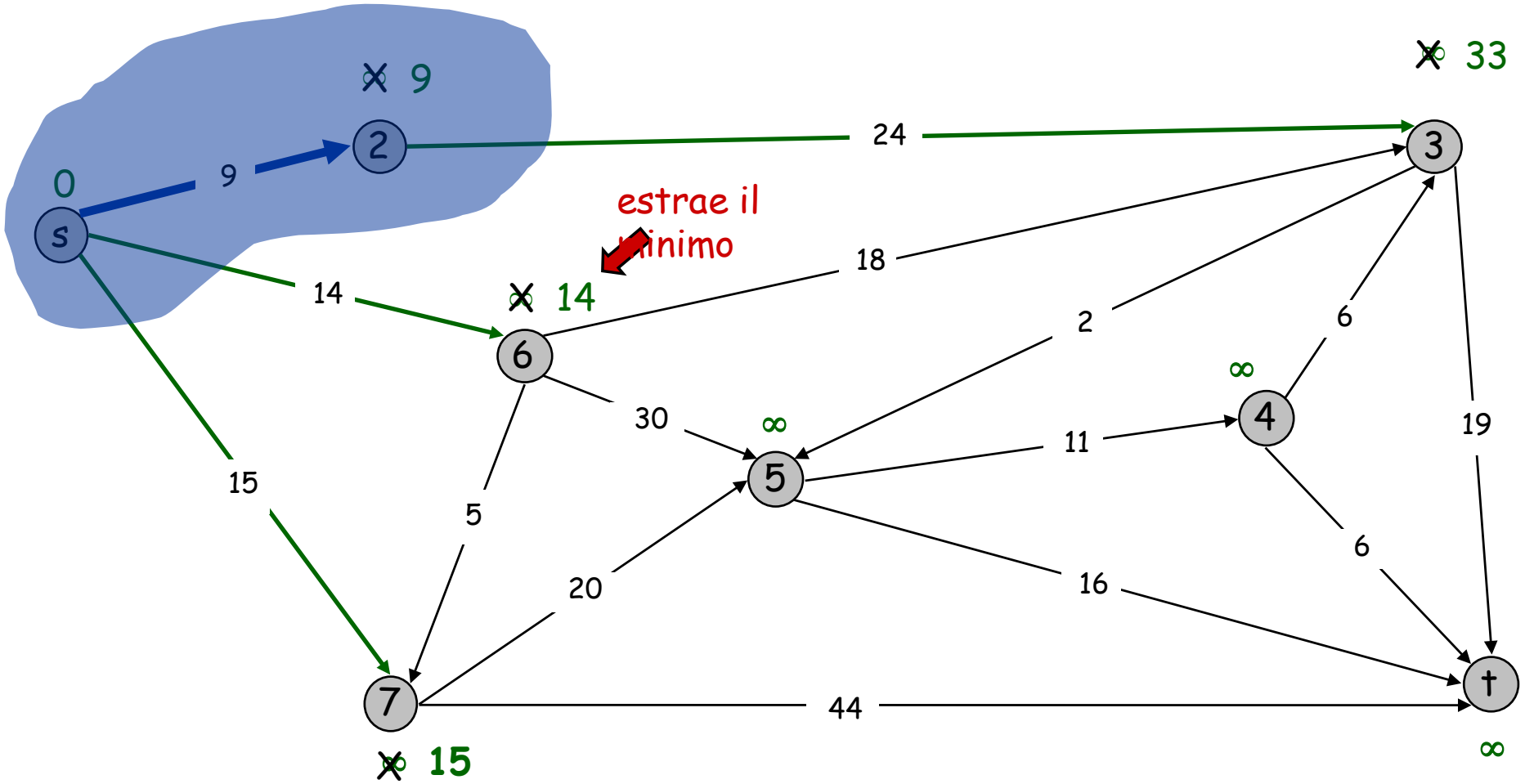
Algoritmo di Dijkstra

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



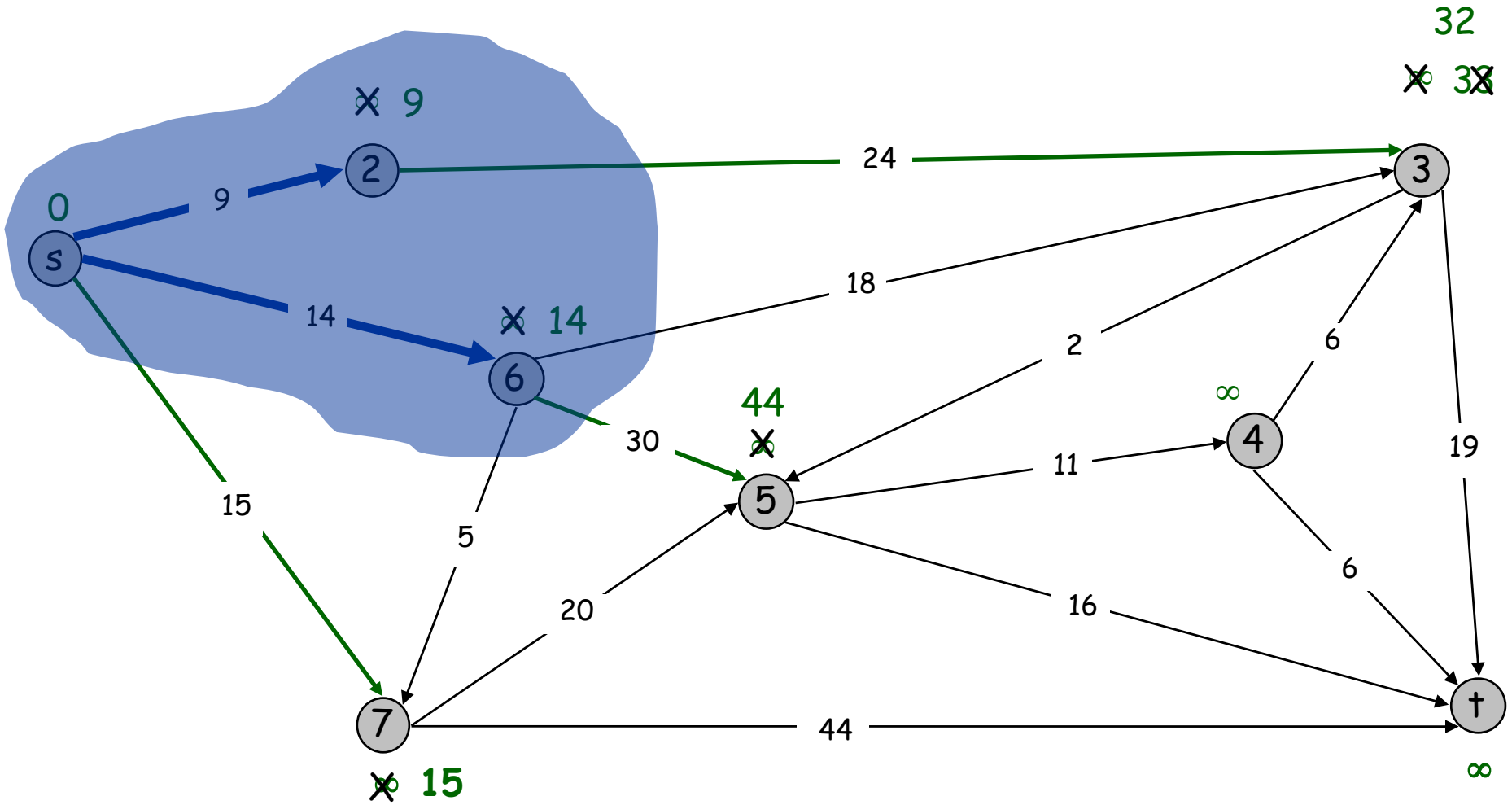
Algoritmo di Dijkstra

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



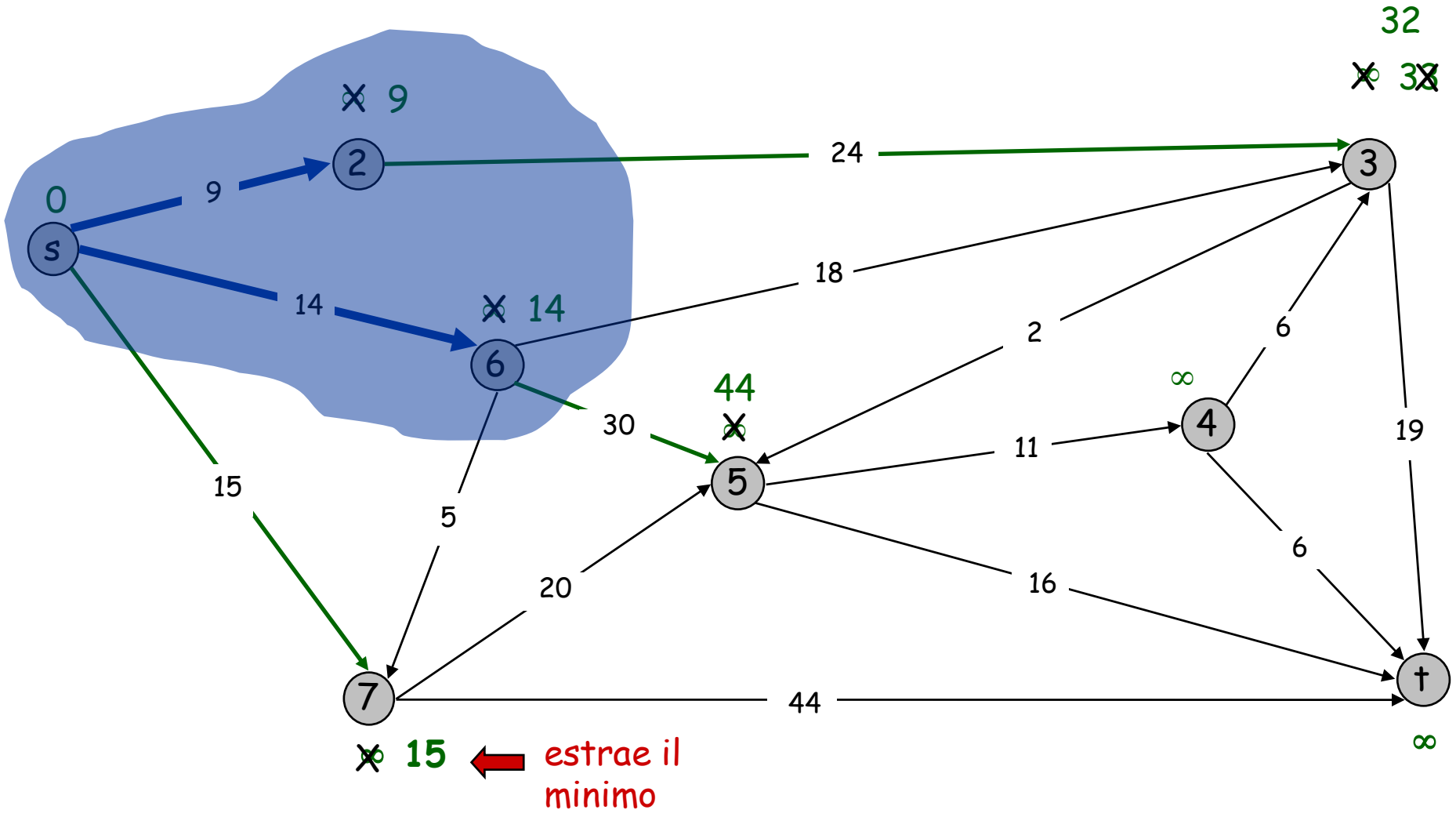
Algoritmo di Dijkstra

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



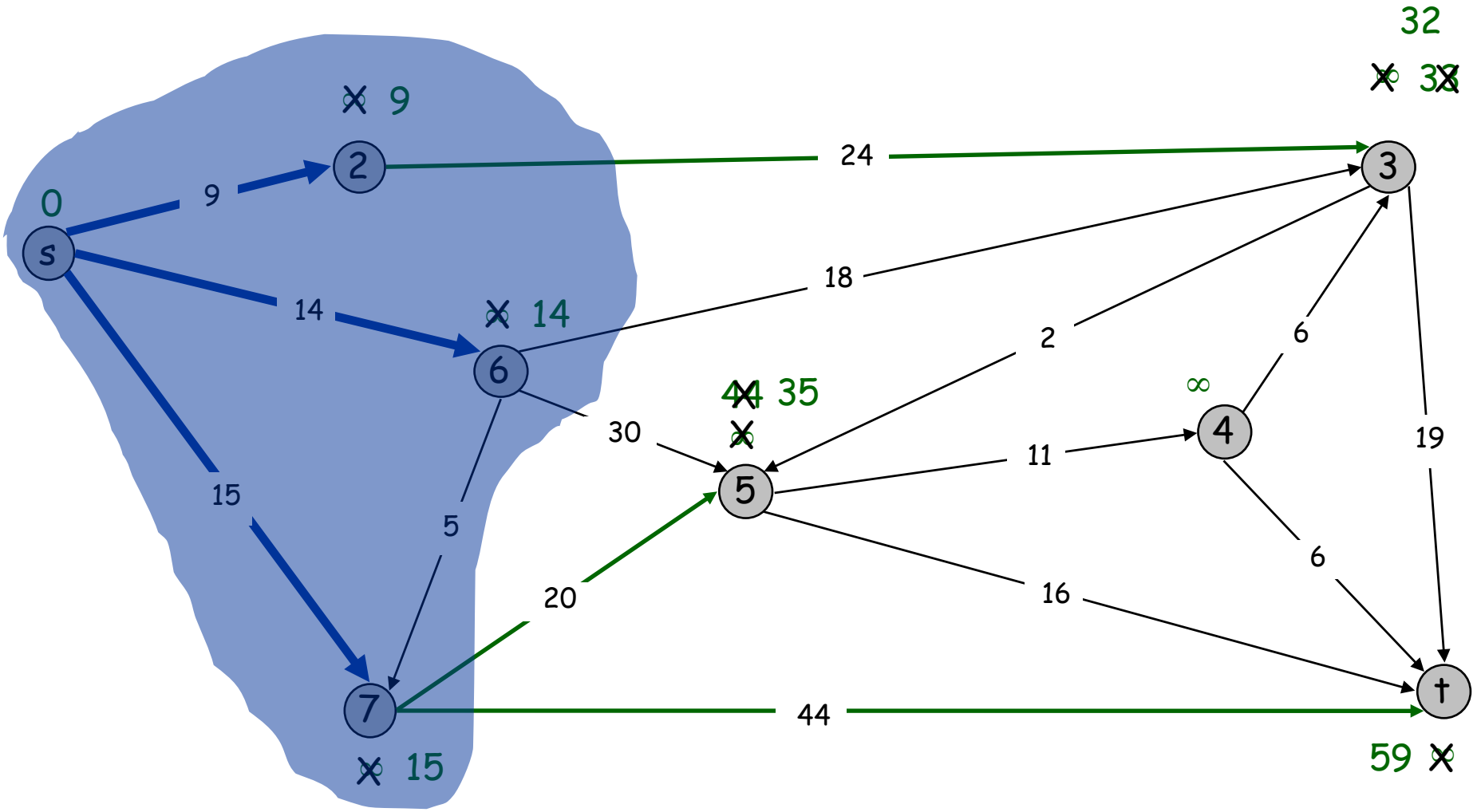
Algoritmo di Dijkstra

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



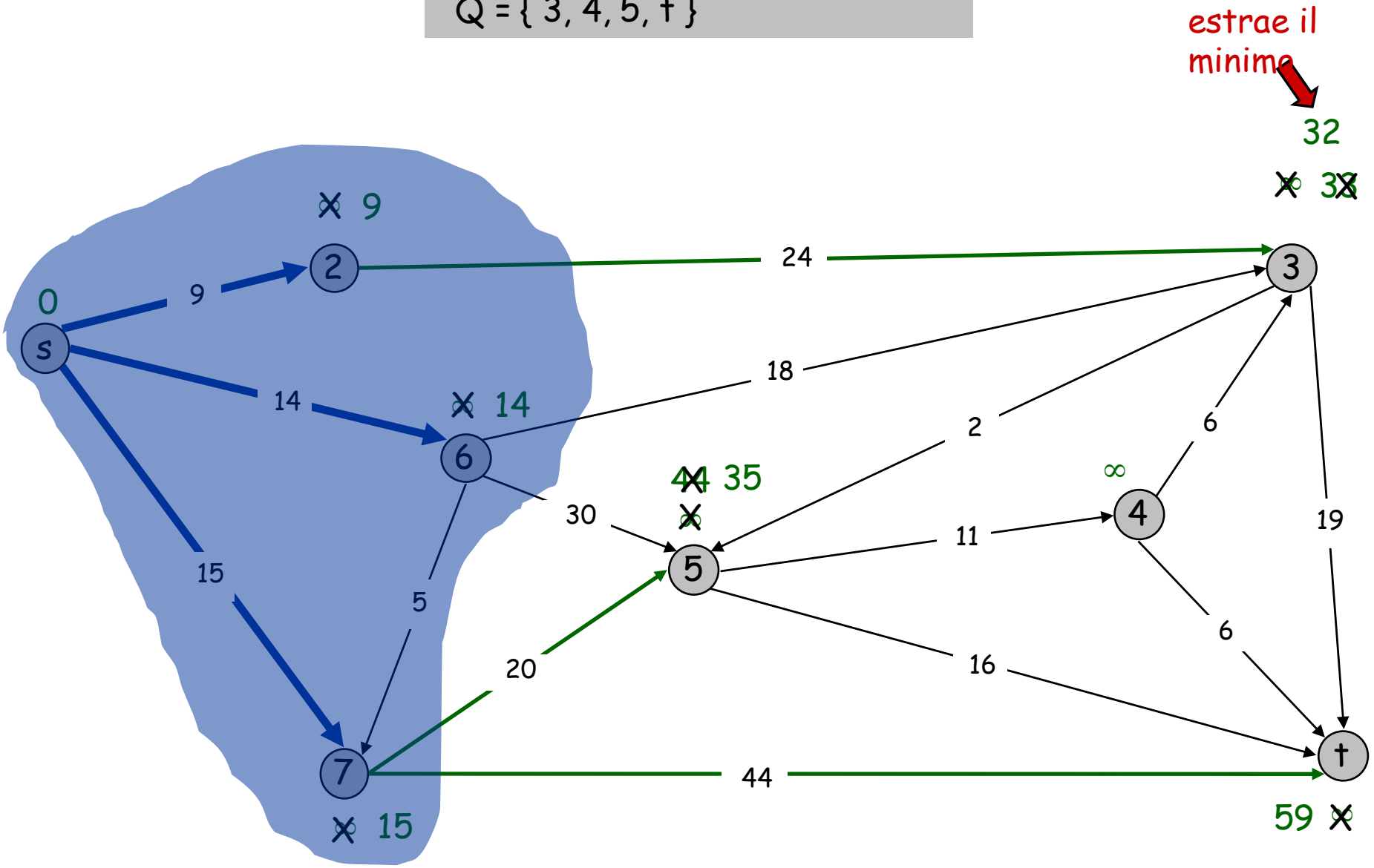
Algoritmo di Dijkstra

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



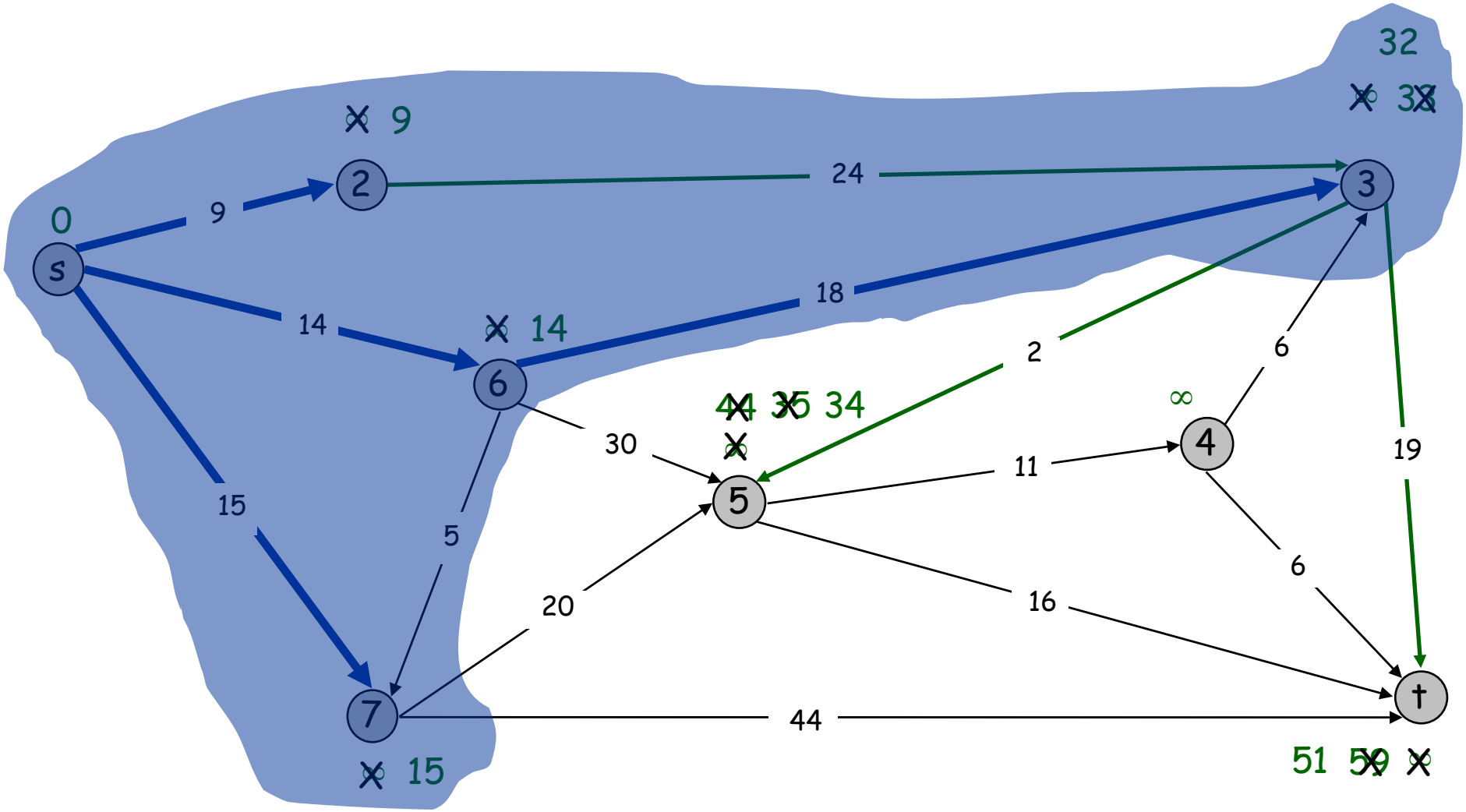
Algoritmo di Dijkstra

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



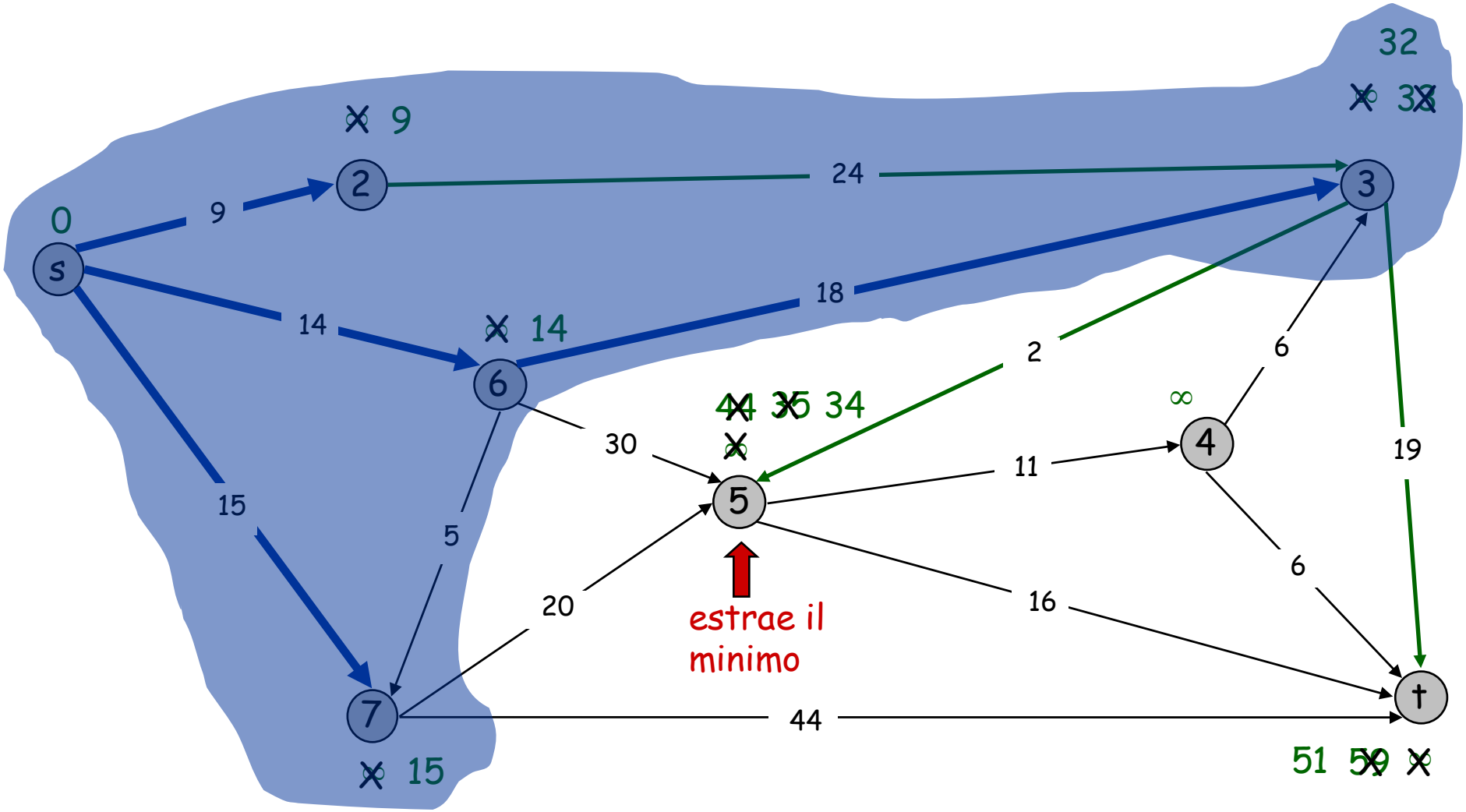
Algoritmo di Dijkstra

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



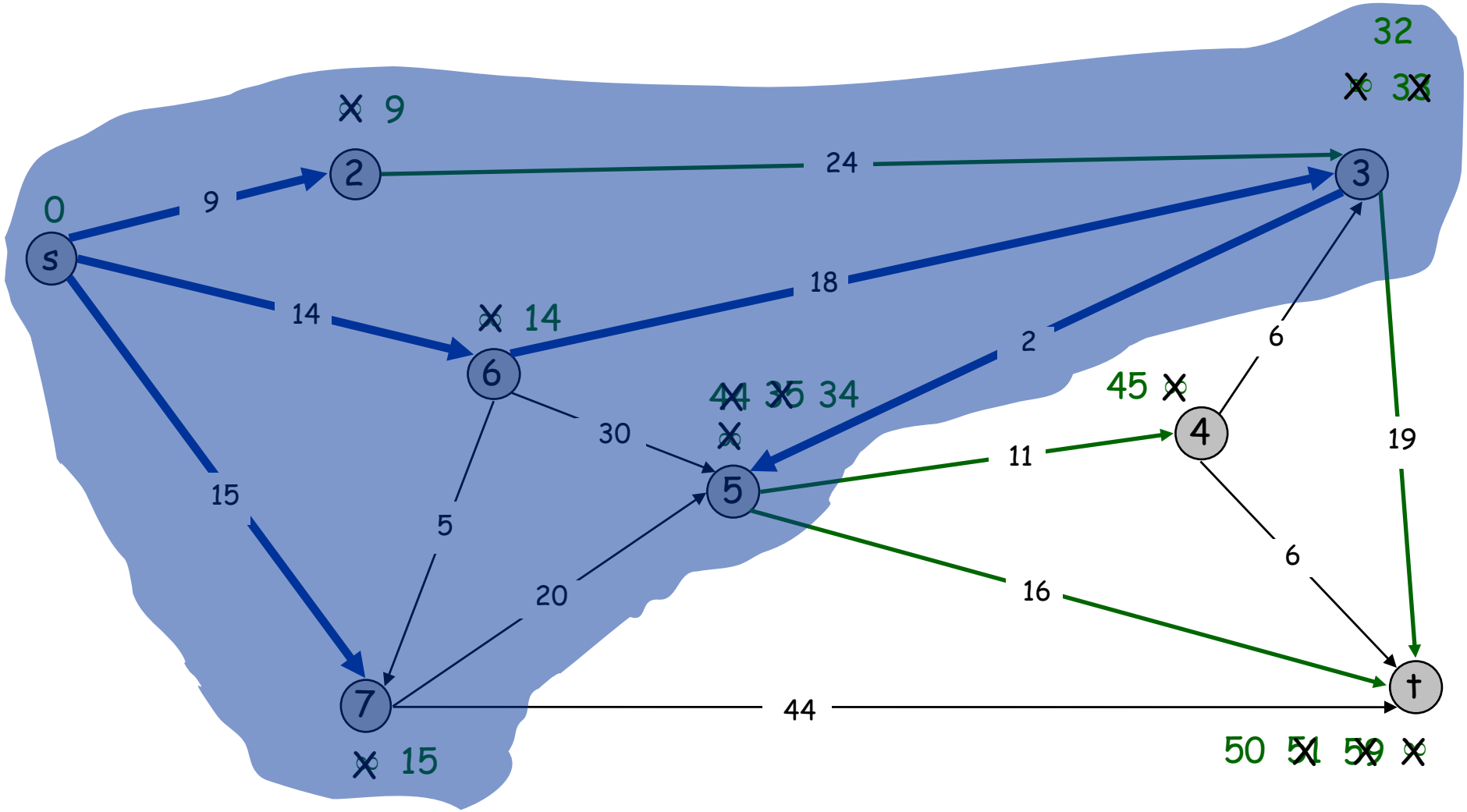
Algoritmo di Dijkstra

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



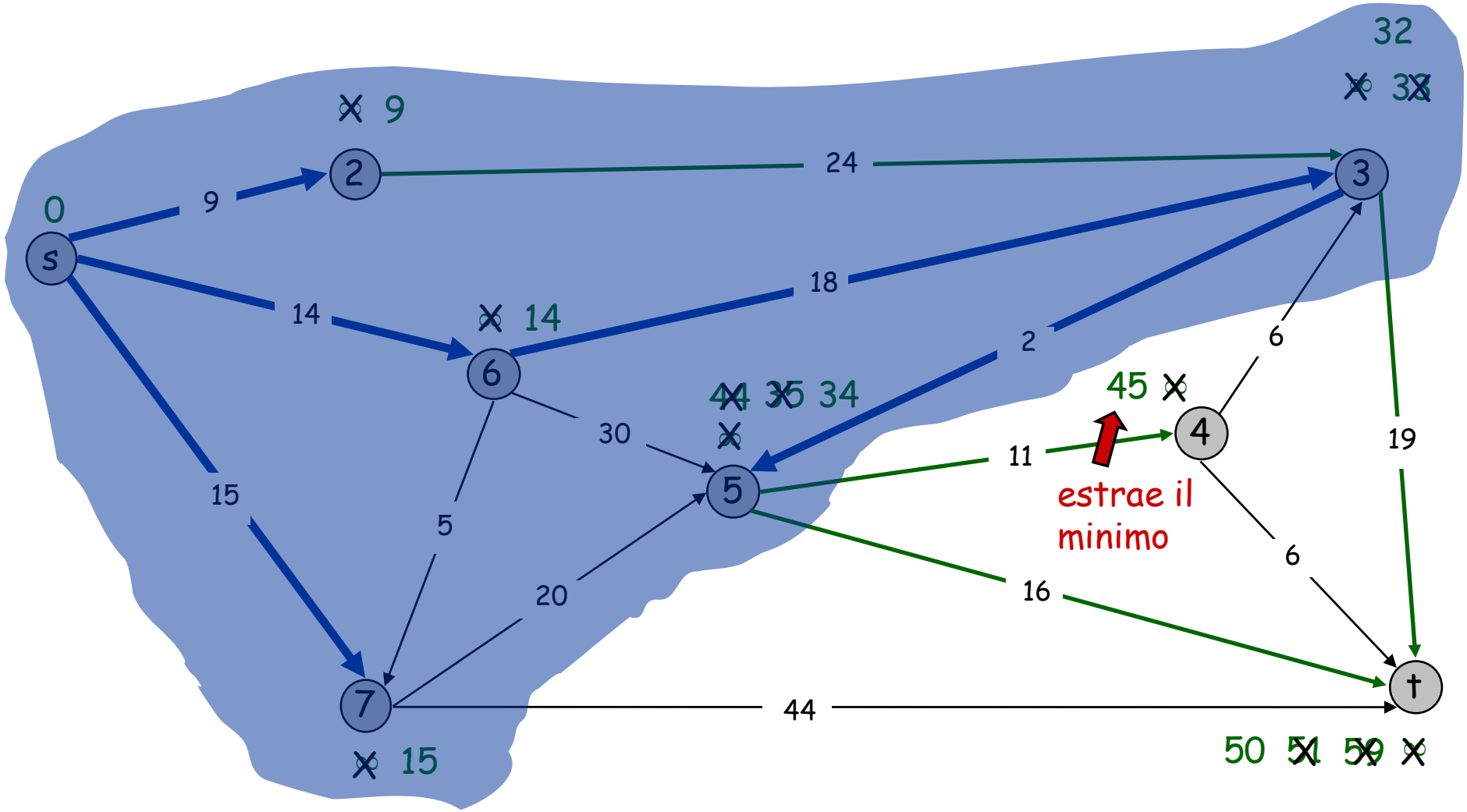
Algoritmo di Dijkstra

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



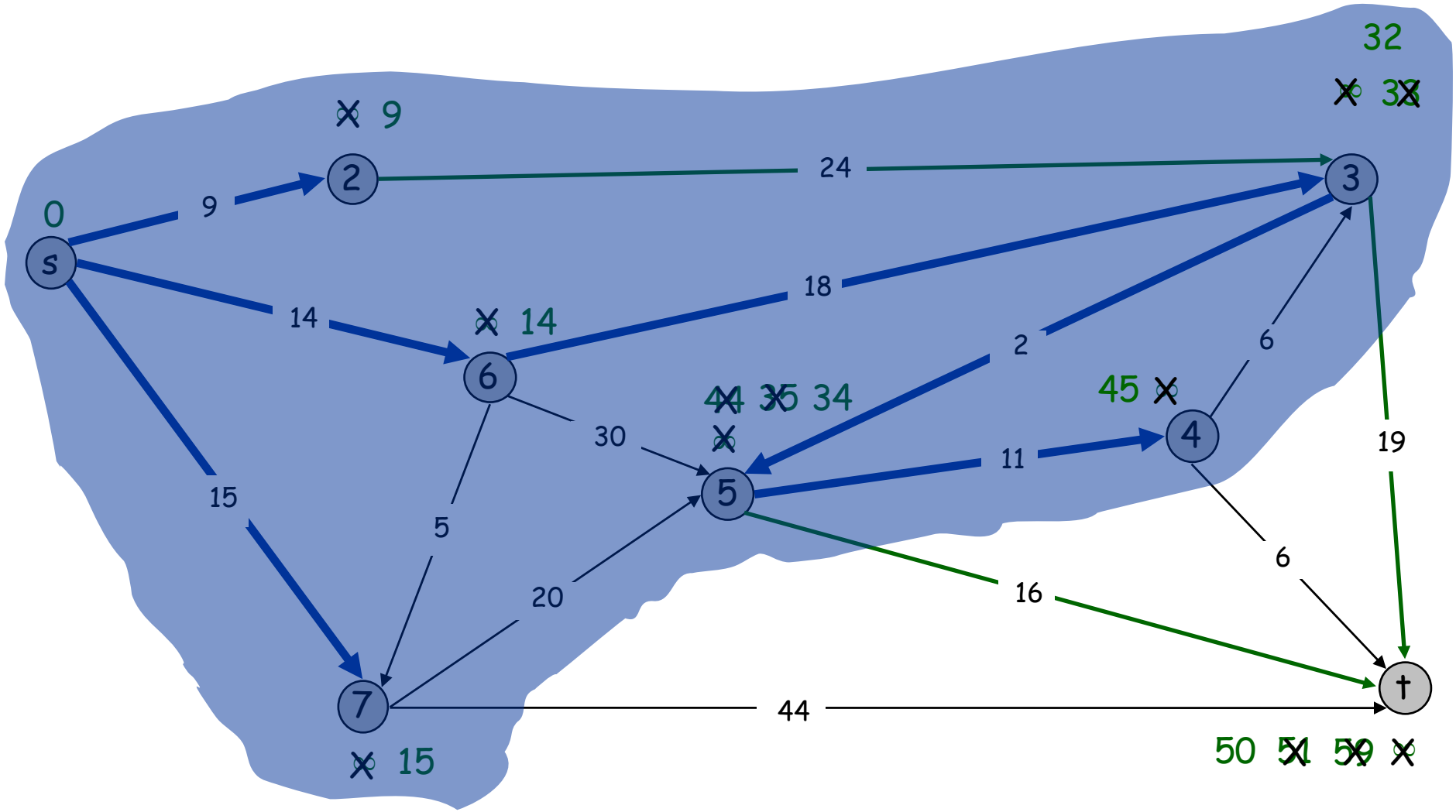
Algoritmo di Dijkstra

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



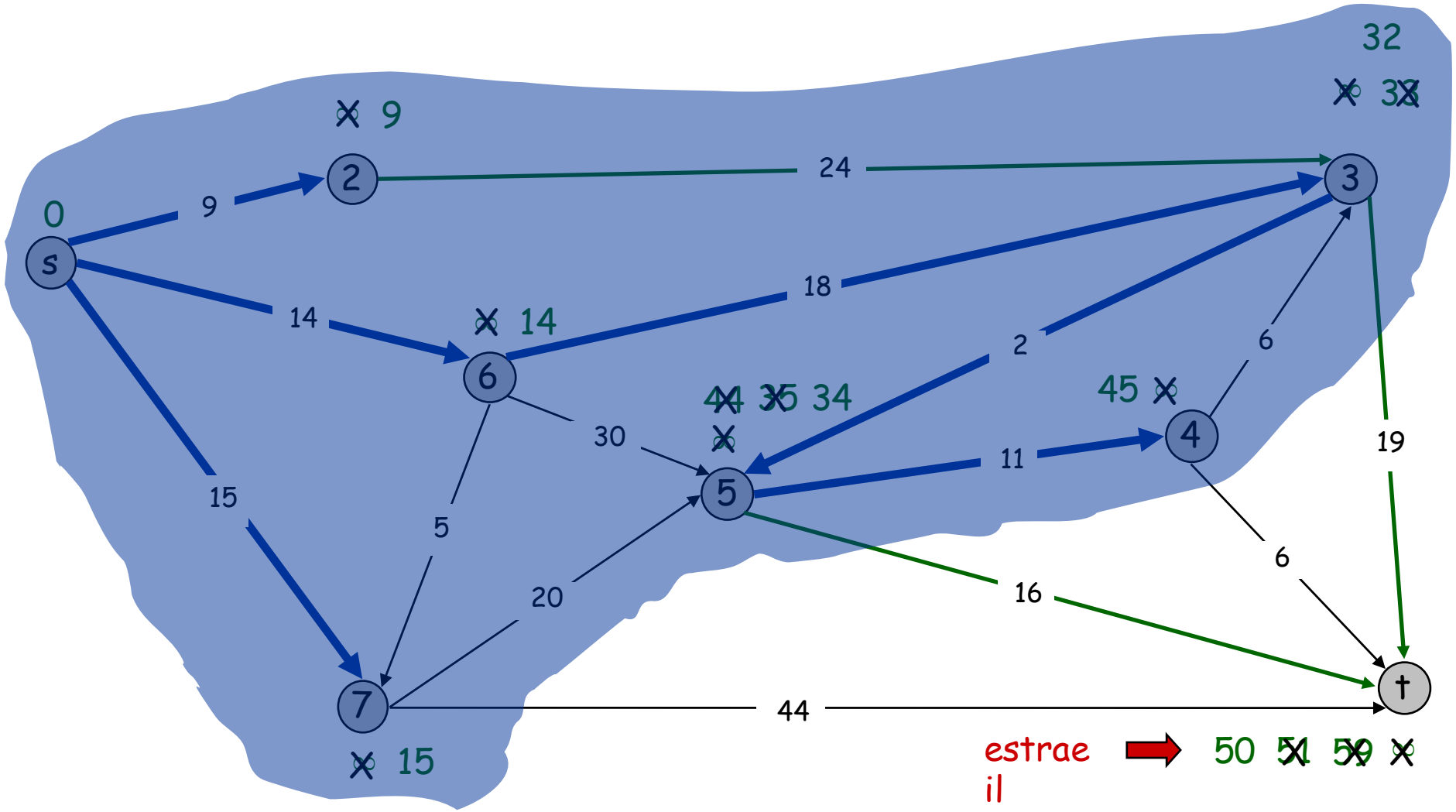
Algoritmo di Dijkstra

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



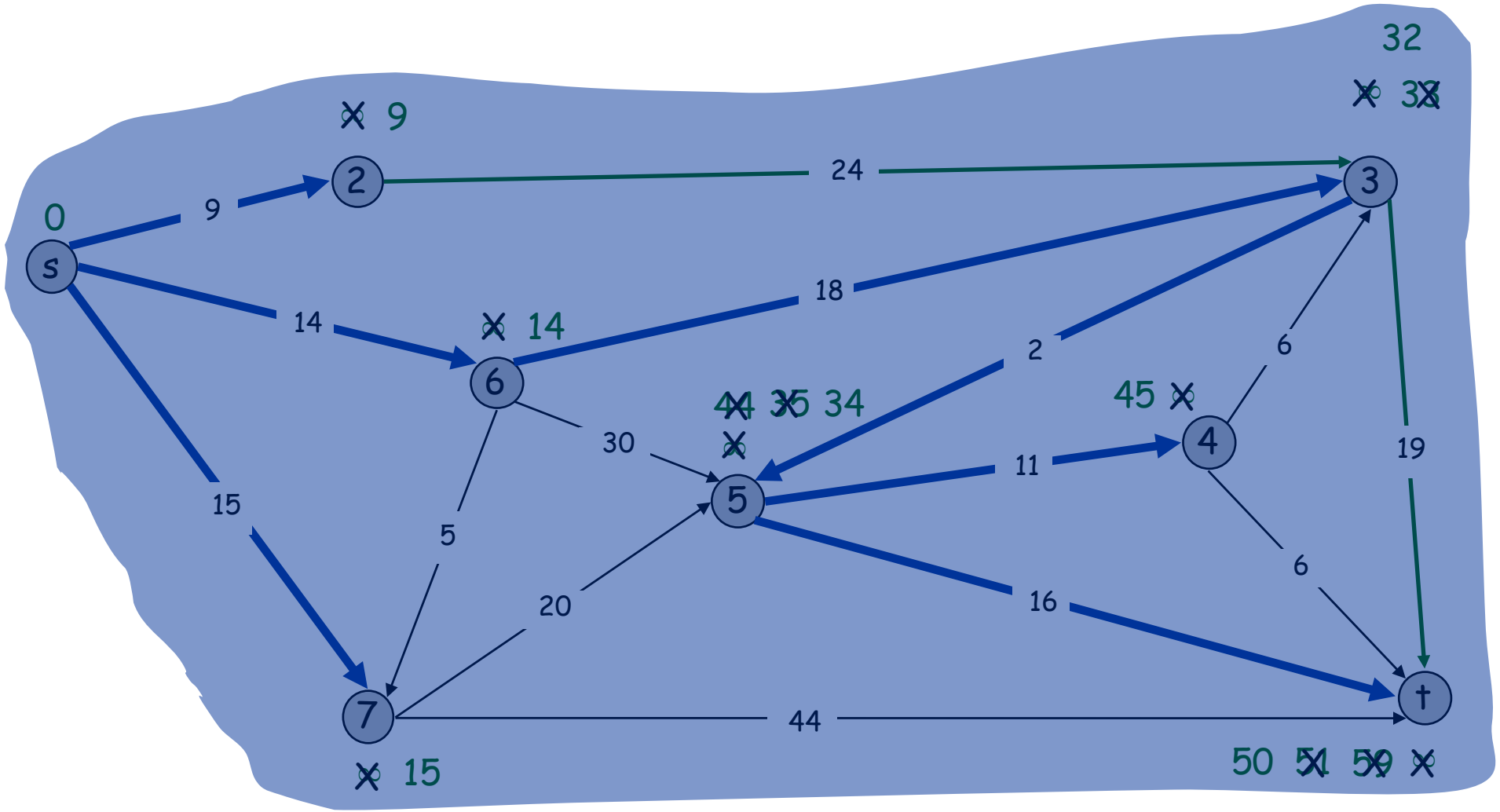
Algoritmo di Dijkstra

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



Algoritmo di Dijkstra

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



Algoritmo di Dijkstra

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