Algoritmi greedy
II parte

Progettazione di Algoritmi 2017-18

Matricole congrue a 1

Docente: Annalisa De Bonis

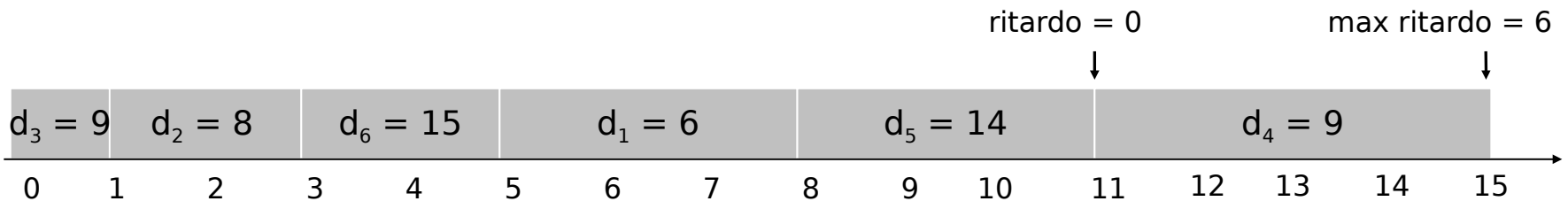
Scheduling per Minimizzare i Ritardi

Problema della minimizzazione dei ritardi.

- Una singola risorsa in grado di elaborare un unico job.
- Il job j richiede t_j unità di tempo e deve essere terminato entro il tempo d_j (scadenza).
- Se j comincia al tempo s_j allora finisce al tempo $f_j = s_j + t_j$.
- Def. Ritardo è definito come $\ell_j = \max \{ 0, f_j - d_j \}$.
- **Input**: n job che richiedono tempi di esecuzione $t_1,\ldots,t_n$ e devono essere terminato entro i tempi $d_1,\ldots,d_n$
- **Obiettivo**: trovare uno scheduling di tutti i job che minimizzi il ritardo massimo $L = \max \ell_j$.

Esempio:

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



Minimizzare il ritardo: Algoritmo Greedy

Schema greedy. Considera i job in un certo ordine.

- [Shortest processing time first] Considera i job in ordine non decrescente dei tempi di elaborazione t_j.

- [Earliest deadline first] Considera i job in ordine non decrescente dei tempi entro i quali devono essere ultimati d_j.

- [Smallest slack] Considera i job in ordine non decrescente degli scarti $d_j - t_j$.

Minimizzare il ritardo: Algoritmo Greedy

- [Shortest processing time first] Considera i job in ordine non decrescente dei tempi di elaborazione t_j.

	1	2
t_j	1	10
d_j	100	10

controesempio

Viene eseguito prima il job 1. Ritardo massimo è 11-10=1. Se avessimo eseguito prima il job 2 avremmo avuto $l_1 = \max\{0, 10-10\}=0$ e $l_2 = \max\{0,11-100\}=0$ per cui il ritardo massimo sarebbe stato 0.

- [Smallest slack] Considera i job in ordine non decrescente degli scarti $d_j - t_j$.

	1	2
t_j	1	10
d_j	2	10

controesempio

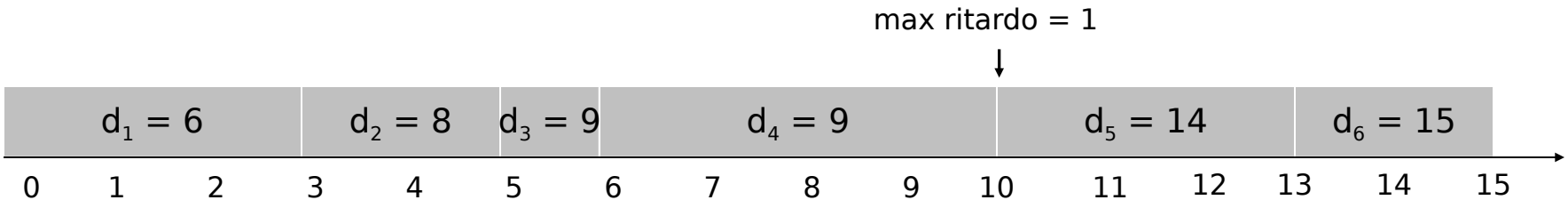
Viene eseguito prima il job 2. Ritardo massimo è 11-2=9. Se avessimo eseguito prima il job 1 il ritardo massimo sarebbe stato 11-10=1

Minimizzare il ritardo: Algoritmo Greedy

Algoritmo greedy. Earliest deadline first: Considera i job in ordine non decrescente dei tempi d_j entro i quali devono essere ultimati.

```
Sort n jobs by deadline so that d1 ≤ d2 ≤ … ≤ dn

t ← 0
for j = 1 to n
   Assign job j to interval [t, t + tj]
   sj ← t, fj ← t + tj
   t ← t + tj
output intervals [sj, fj]
```



Minimizzare il ritardo: Inversioni

Def. Un' inversione in uno scheduling S è una coppia di job i e j tali che:
$d_i < d_j$ ma j viene eseguito prima di i.

Ottimalità soluzione greedy

La dimostrazione dell'ottimalità si basa sulle seguenti osservazioni che andremo poi a dimostrare

1. La soluzione greedy ha le seguenti due proprietà:

a. Nessun idle time. Non ci sono momenti in cui la risorsa non è utilizzata
b. Nessuna inversione. Se un job j ha scadenza maggiore di quella di un job i allora viene eseguito dopo i

2. Tutte le soluzioni che hanno in comune con la soluzione greedy le caratteristiche a e b, hanno lo stesso ritardo massimo della soluzione greedy.

3. Ogni soluzione ottima può essere trasformata in un'altra soluzione ottima per cui valgono la a e la b

Dimostrazioni delle osservazioni 1. 2. e 3.

1. La soluzione greedy ha le seguenti due proprietà:

a. Nessun idle time. Non ci sono momenti in cui la risorsa non è utilizzata
b. Nessuna inversione. Se un job j ha scadenza maggiore di quella di un job i allora viene eseguito dopo i

Dim.

Il punto a discende dal fatto che ciascun job comincia nello stesso istante in cui finisce quello precedente.

Il punto b discende dal fatto che i job sono esaminati in base all'ordine non decrescente delle scadenze.

Dimostrazioni delle osservazioni 1. 2. e 3.

Prima di dimostrare il punto 2 consideriamo i seguenti fatti

Fatto I. In uno scheduling con le caratteristiche a e b i job con una stessa scadenza d sono disposti uno di seguito all'altro.

Dim.

Siano i e j due job con $d_i=d_j=d$ e assumiamo senza perdere di generalità (da ora in poi s.p.d.g.) che i venga eseguito prima di j . Supponiamo **per assurdo** che tra i e j venga eseguito il job q con $d_i \neq d_q$.

Se $d < d_q$ allora la coppia j, q è un'inversione. Se $d > d_q$ allora la coppia i, q è un'inversione. Ciò contraddice la proprietà b.

Ne consegue che tra due job con una stessa scadenza d non vengono eseguiti job con scadenza diversa da d e poiché lo scheduling non ha idle time, i job con una stessa scadenza vengono eseguiti uno di seguito all'altro.

Dimostrazioni delle osservazioni 1. 2. e 3.

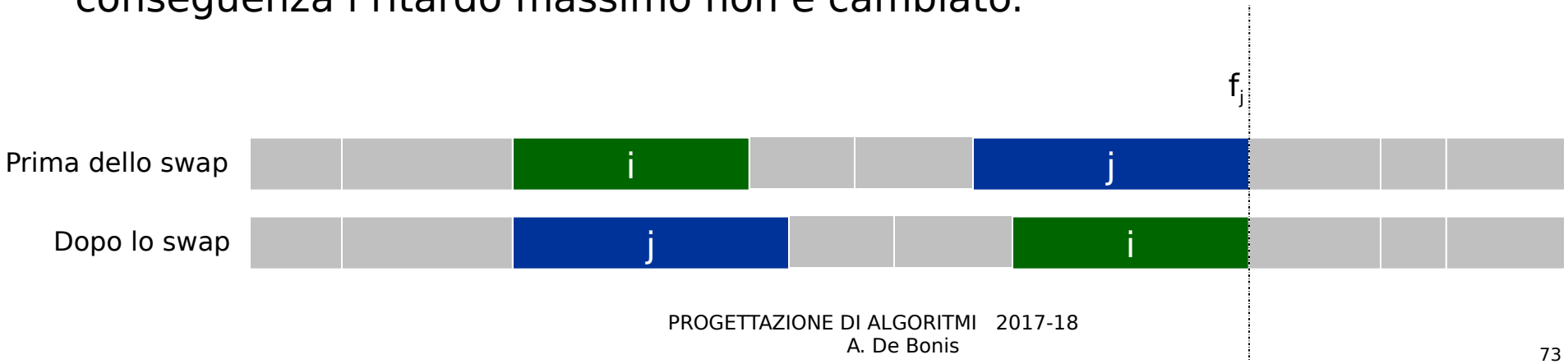
Fatto II. Se in uno scheduling con le caratteristiche a e b scambiamo due job con la stessa scadenza, il ritardo massimo non cambia.

Dim.

Consideriamo due job i e j con $d_i=d_j$ e supponiamo s.p.d.g. che i preceda j in S.

Per il fatto I, tra i e j vengono eseguiti solo job con la stessa scadenza di i e j. Ovviamente il ritardo di j è maggiore o uguale del ritardo di i e dei ritardi di tutti i job eseguiti tra i e j perché j finisce dopo tutti questi job e ha la loro stessa scadenza.

Se scambiamo i con j in S otteniamo che il ritardo di j non può essere aumentato mentre quello di i è diventato uguale a quello che aveva prima j in quanto i finisce nello stesso istante in cui finiva prima j e la scadenza di i è la stessa di j. Il ritardo dei job compresi tra i e j potrebbe essere aumentato ma non può superare il ritardo che aveva prima j. Di conseguenza i ritardo massimo non è cambiato.



Dimostrazioni delle osservazioni 1. 2. e 3.

2. Tutte le soluzioni che hanno in comune con la soluzione greedy le caratteristiche a e b, hanno lo stesso ritardo massimo della soluzione greedy

Dim.

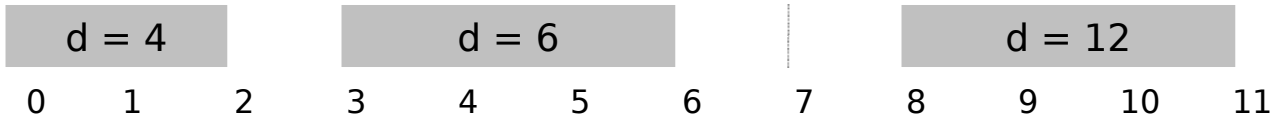
- Dimostriamo che dati due scheduling S ed S’ di n job entrambi aventi le caratteristiche a e b, S può essere trasformato in S’ senza che il suo ritardo massimo risulti modificato.
- Osserviamo che S ed S’ possono differire solo per il modo in cui sono disposti tra di loro job con la stessa scadenza. Infatti se un job in S dovesse essere scambiato con un job con scadenza diversa dalla sua, si avrebbe un’inversione.
- Di conseguenza S può essere trasformato in S’ scambiando tra di loro di posto coppie di job con la stessa scadenza.
- Per il fatto II, scambiando coppie di job con la stessa scadenza il ritardo max non cambia. Di conseguenza possiamo trasformare S in S’ senza che cambi il ritardo max. In altre parole S ed S’ hanno lo stesso ritardo max. Prendendo S uguale ad un qualsiasi scheduling con le caratteristiche a e b ed S’ uguale allo scheduling greedy si ottiene la tesi.

Dimostrazioni delle osservazioni 1. 2. e 3.

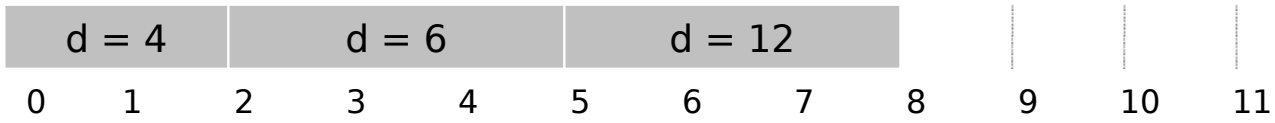
Prima di dimostrare il punto 3 consideriamo i seguenti fatti.

Fatto III. Una soluzione ottima può essere trasformata in una soluzione con nessun tempo di inattività (idle time).
Dim. Se tra il momento in cui finisce l'elaborazione di un job e quello in cui inizia il successivo vi è un gap, basta shiftare all'indietro l'inizio del job successivo in modo che cominci non appena finisce il precedente.
Ovviamenti i ritardi dei job non aumentano dopo ogni shift

Esempio: Soluzione ottima con idle time



Esempio: Soluzione ottima con nessun idle time



.

Dimostrazioni delle osservazioni 1. 2. e 3.

Fatto IV. Se uno scheduling privo di idle time ha un'inversione allora esso ha una coppia di job invertiti che cominciano uno dopo l'altro.

Dim.

- Consideriamo tutte le coppie di job i e j tali che $d_i < d_j$ e j viene eseguito prima di i nello scheduling. Supponiamo per assurdo tutte le coppie di questo tipo siano separate da un job. Tra tutte le coppie siffatte prendiamo quella più vicina nello scheduling.
- Deve esistere un job $k \neq i$ eseguito subito dopo j che non forma un'inversione né con i né con j altrimenti i e j non formerebbero l'inversione più vicina. Deve quindi essere $d_j \leq d_k$ e $d_k \leq d_i$. Le due diseguaglianze implicano $d_j \leq d_i$ il che contraddice il fatto che la coppia i, j e un'inversione.

Dimostrazioni delle osservazioni 1. 2. e 3.

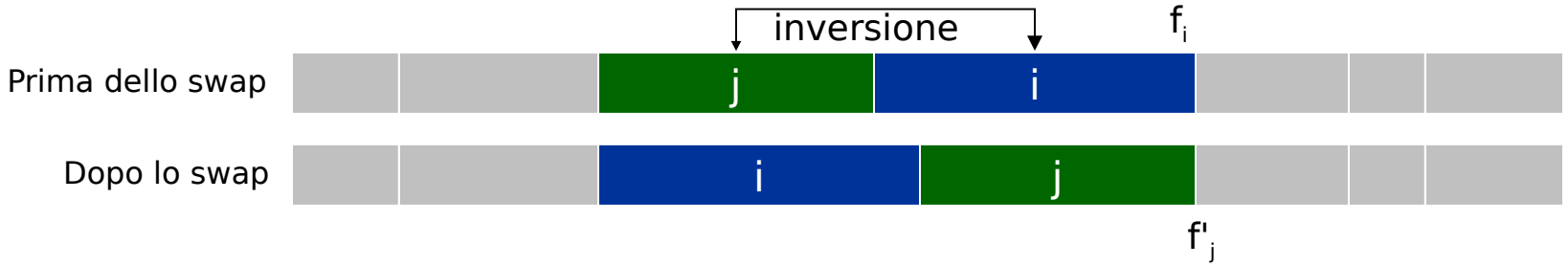
Fatto V. Scambiare due job **adiacenti** i**nvertiti** i e j riduce il numero totale di inversioni di uno e non fa aumentare il ritardo massimo.

Dim. Supponiamo $d_i < d_j$ e che j precede i nello scheduling.
Siano $\ell_1,\ldots,\ell_n$ i ritardi degli n job e siano $\ell'_1,\ldots,\ell'_n$ i ritardi degli n job dopo aver scambiato i e j di posto. Si ha che

- $\ell'_k = \ell_k$ per tutti i $k \neq i, j$
- $\ell'_i \leq \ell_i$ perchè viene anticipata la sua esecuzione.
- Vediamo se ritardo di j è aumentato al punto da far aumentare il ritardo max. Ci basta considerare il caso in cui $\ell'_j > 0$ altrimenti vuol dire che il ritardo di j non è aumentato. Si ha quindi

$$\begin{aligned} \ell'_j &= f'_j - d_j && \text{(per la definizione di ritardo)} \\ &= f_i - d_j && \text{(dopo lo swap, j finisce al tempo } f_i\text{)} \\ &< f_i - d_i && \text{(in quanto } d_i < d_j\text{)} \\ &\leq \ell_i && \text{(per la definizione di ritardo)} \end{aligned}$$

e quindi il ritardo max non è cambiato.



Dimostrazioni delle osservazioni 1. 2. e 3.

- 3. Ogni soluzione ottima D può essere trasformata in un'altra soluzione ottima per cui valgono la a e la b

  Dim.
- Il fatto III implica che la soluzione ottima S può essere trasformata in una soluzione ottima per cui non ci sono idle time.

- Il fatto IV implica che se la soluzione ottima S contiene inversioni allora S contiene una coppia di job **adiacenti** invertiti. Il fatto V implica che se scambiamo le posizioni di questi due job otteniamo ancora una soluzione ottima con un numero inferiore di inversioni. Quindi possiamo scambiare di posto coppie di job adiacenti invertiti fino a che non ci sono più inversioni nella soluzione ottima.

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, …, d_n 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 ad un certo passo i può decidere di inserire in cache un elemento che non è stato richiesto

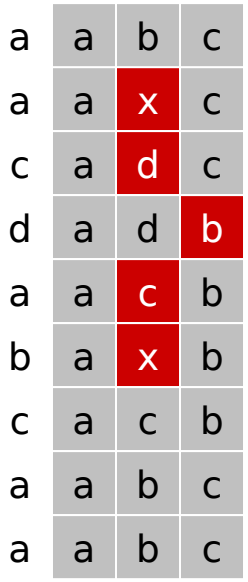
Problema del caching offline ottimale

- 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.

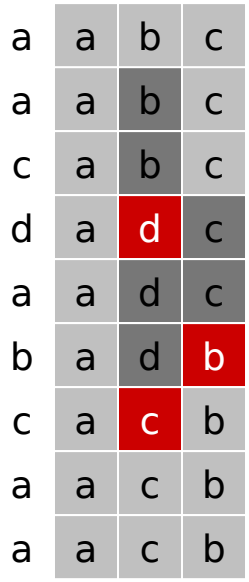
- Osservazione. In un eviction schedule ridotto il numero di inserimenti in cache è uguale al numero di cache miss.

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

Eviction Schedule ridotto



Uno schedule non ridotto



Uno schedule ridotto

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:

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. [Bellady, 1960s] Farthest-in-future è uno schedule (ridotto) 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: implementazione

- Per ogni elemento d manteniamo una lista L[d] contenente i tempi di arrivo delle richieste dell'elemento d
- Usiamo una coda a priorità che per ogni elemento **in cache** mantiene il tempo in cui verrà richiesto la prossima volta.
- All'inizio l'algoritmo scandisce tutte le richieste. Sia d_j la richiesta che arriva al tempo j.
  - Se d_j non è stato incontrato prima ($L[d_j]$ vuota), l'algoritmo inserisce d_j nella coda a priorità con chiave j.
  - L'algoritmo inserisce j alla fine della lista $L[d_j]$.
- L'algoritmo scandisce tutte le richieste
  - Per ogni j rimuove la testa della lista $L[d_j]$
  - Se d_j non è in cache allora l'algoritmo estrae dalla coda l'elemento d_h con chiave minima (max tempo di arrivo) e inserisce d_j al suo posto
  - Se $L[d_j]$ è vuota la chiave di d_j diventa `-∞;` se invece $L[d_j]$ non è vuota la chiave di d_j diventa -p, dove p è l'intero in testa a $L[d_j]$ `;`

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] the list of j s.t.dj=d
Let Q a priority queue
for j = 1 to n {
  if(list L[dj] is empty and dj is in the cache)
      Insert(Q,dj,-j)      //first time dj is requested
  append j to list L[dj]
}
for j = 1 to n {
   remove first element from L[dj]
   if (dj is NOT in the cache){
      //dj needs to be brought into the cache
      dh ← ExtractMin(Q)
      evict dh from the cache and bring dj to the cache
      }
    else remove(Q,dj) //it is removed to be inserted
                      //with a new key
    if(L[dj] is empty)   // no further request of dj
       Insert(Q,dj ,-∞)
    else
       {p ← first element of L[dj]
        Insert (Q,dj ,-p)   }
  }
```

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

O(n log k)

Analisi dell’algoritmi di Belady

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

Farthest-In-Future: ottimalità

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

- 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

-

Farthest-In-Future: ottimalità

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 tempo 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 $j \geqq 0$. 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.

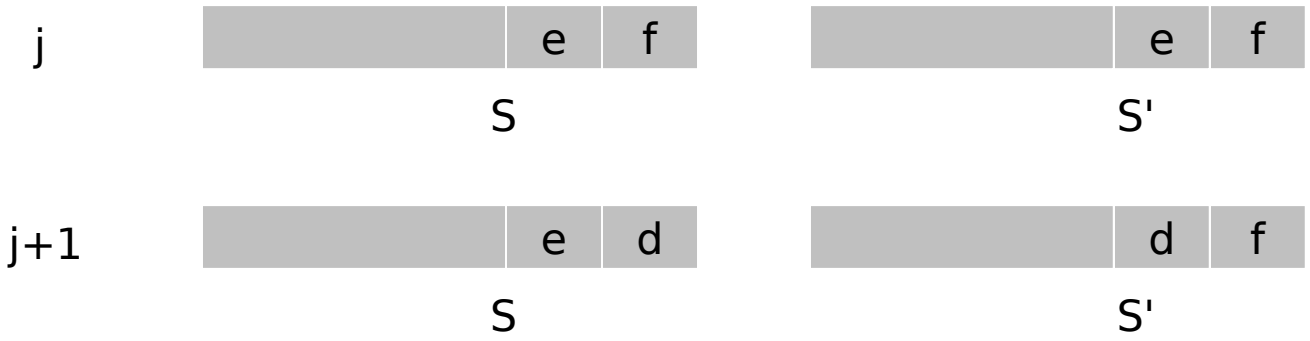
Produciamo S' nel seguente modo.

- 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.
  - 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

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 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

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.

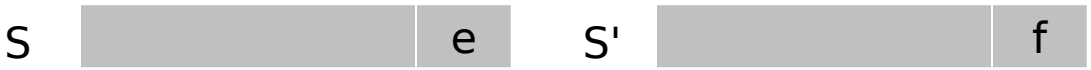
- A questo punto S’ deve fare necessariamente una scelta diversa da quella di S. Facciamo però in modo che la scelta di S’ renda il contenuto della cache di S’ identico a quello della cache di S.

- 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

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'.



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** e **g** è presente nella cache di S: in questo caso **g** è presente anche nella cache di S' che non fa niente come S.
- **g ≠ e**, **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**, **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’** ≠ **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. In questo scheduling ridotto il numero di cache miss è uguale al numero di cache miss di S.

Farthest-In-Future: ottimalità

- NB: nel caso 3.3 il numero di inserimenti di S e quello di S’ fino al passo j’ (incluso) sono uguali. Il numero di inserimenti di S e quello di S’ (prima della trasformazione in uno scheduling ridotto) continuano ad essere uguali fino alla fine perche’ dal passo j’ in poi S ed S’ hanno in cache gli stessi elementi e possono fare le stesse scelte. Indichiamo con m il numero totale di inserimenti. S è ridotto per cui si ha che il numero di cache miss di S è uguale ad m. Nel trasformare S’ in uno scheduling ridotto il numero di inserimenti totale

  1. non aumenta per cui risulta minore o uguale di m.
  2. diventa uguale al numero totale di cache miss.

Ne consegue che il numero totale di cache miss di S’ è minore o uguale di m ed è quindi minore o guale del numero di cache miss di S.

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 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.

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,…,n fino a che non arriviamo ad uno schedule S_n che effettua esattamente le stesse scelte di S_{FF} (S_n= 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 è indicata la distanza tra ciascuna coppia di intersezioni adiacenti
- Come si può individuare il percorso più breve da Napoli a Milano?

Cammini minimi

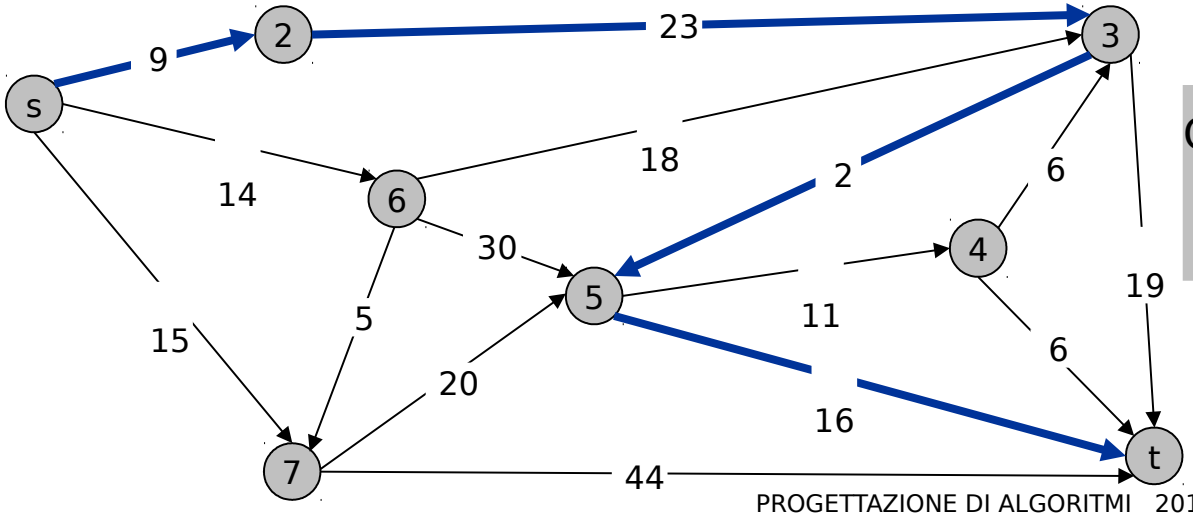
Esempi di applicazioni dei cammini minimi in una rete

- 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 blu da s a t
s-2-3-5-t = 9 + 23 + 2 + 16
= 48.

Varianti del problema dei cammini minimi

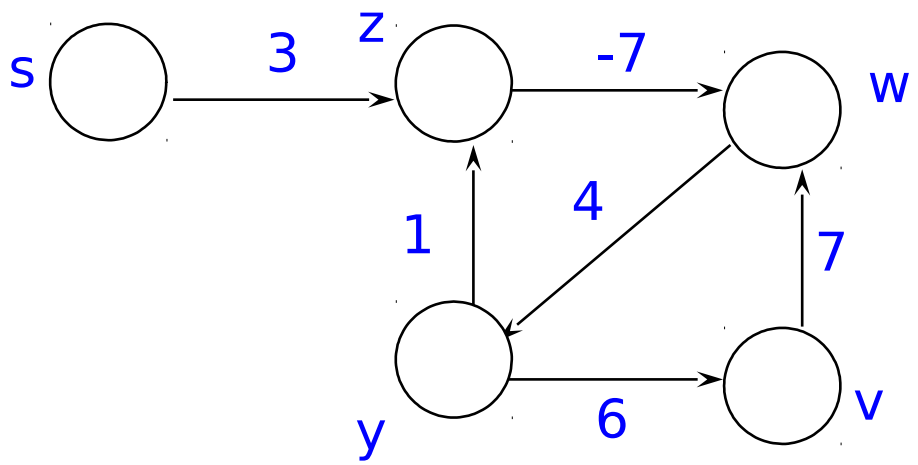
- Single Source Shortest Paths: determinare il cammino minimo da un dato vertice sorgente s ad ogni altro vertice
- Single Destination Shortest Paths: determinare i cammini minimi ad un dato vertice destinazione t da tutti gli altri vertici
  - Si riduce a Single Source Shortest Path invertendo le direzioni 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



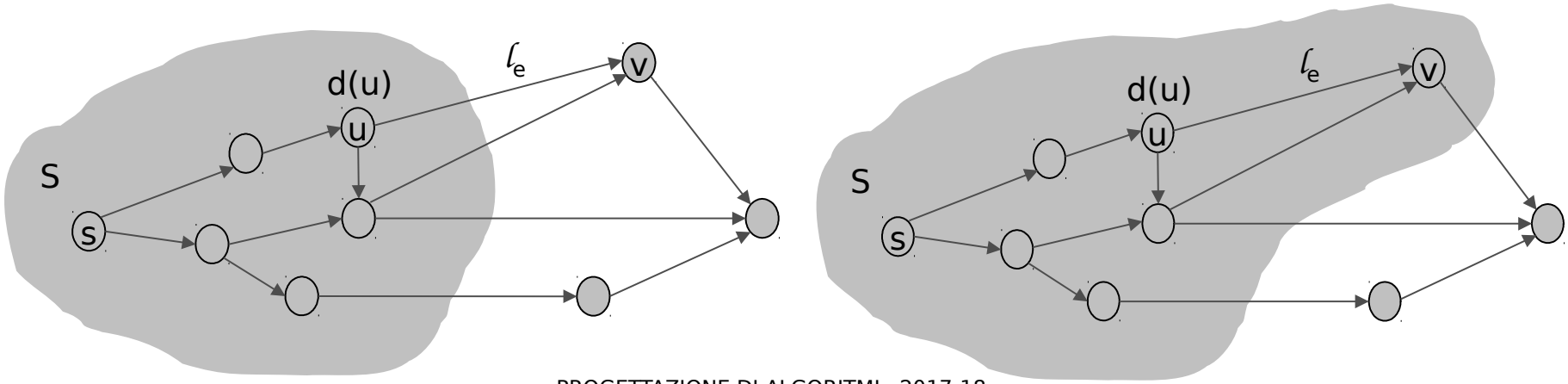
Il ciclo <z,w,y,z> ha peso -2

- 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 con il valore $d'(v) = \min_{e=(u,v):\, u \in S} d(u) + \ell_e$, più piccolo e aggiunge v a S ponendo d(v) = d'(v).



Algoritmo di Dijkstra

```
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
```

In questa versione dell'algoritmo di Dijkstra occorre assumere che ogni vertice sia raggiungibile da s

In alternativa, si potrebbe modificare il codice in questo modo:

- Per ogni nodo v diverso da s, d'(v) e d(v) vengono inizialmente posti uguali a ∞
- Nel ciclo di while,
  - d'(v) viene computato solo per i nodi che non sono in S e che sono adiacenti a qualche nodo di S, così come accade nell'algoritmo in figura,
  - al contrario di quanto accade nell'algoritmo in figura, il nodo v che minimizza d'(v) viene selezionato tra tutti i nodi che non sono in S e non solo tra quelli adiacenti a qualche nodo di S. In questo modo, nelle ultime iterazioni vengono inseriti in S gli eventuali nodi non raggiungibili a partire da s.

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 $\ell_e ? 0$ (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)

Dimostriamo **anche** che per ogni nodo u $\in$ S, il percorso di lunghezza d[u] è formato solo da nodi che sono già in S.

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

Algortimo di Dijksta: Correttezza

Passo induttivo: 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 da s ad u di lunghezza d(u) più l'arco **(u,v)**

1. Per ipotesi induttiva d(u) è la lunghezza di un percorso minimo da S ad u e questo percorso passa solo attraverso nodi di S. Quindi il percorso di lunghezza d(u)+ℓ(**(u,v)**) è il percorso più corto da s a v che termina con l'arco **(u,v)** e questo percorso passa solo attraverso nodi di S.
2. **(u,v)** è l'arco in corrispondenza del quale si ottiene

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

- 1. e 2. → d'(v) è la lunghezza del percorso da s a v più corto tra quelli che passano solo per nodi di s.

Continua nella prossima slide

Algortimo di Dijksta: Correttezza

....passo induttivo:

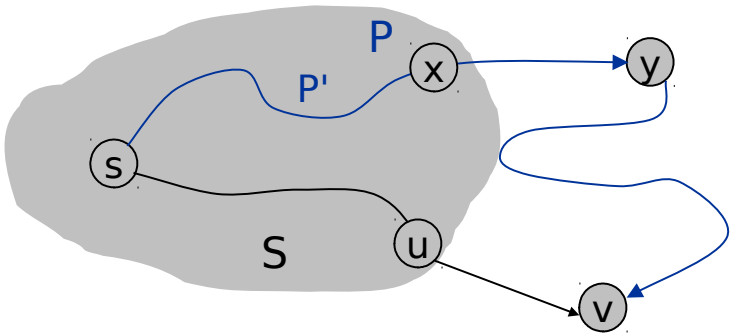
- I punti 1 e 2 valgono per un qualsiasi nodo v non in S che ha almeno un arco entrante proveniente da un nodo di S.
- Tra tutti i nodi di questo tipo, noi stiamo considerando il nodo v inserito in S al (k+1)-esimo passo, cioè quello con il valore d'(v) più piccolo.
- Se fossimo certi che il percorso minimo da s a questo nodo v si trova tra quelli che passano solo attraverso nodi di S, il passo induttivo sarebbe già dimostrato in quanto potremmo dire che il percorso minimo da s a v è quello di lunghezza d'(v). Abbiamo infatti già osservato che d'(v) è proprio la lunghezza del percorso più corto da s a v passante solo per nodi di S.
- Per essere certi che non vi sia un percorso da s a v più corto di d'(v), occorre dimostrare che qualsiasi percorso da s a v che attraversa uno o più nodi che non sono in S ha lunghezza maggiore o uguale a d'(v).

Continua nella prossima slide

Algortimo di Dijksta: Correttezza

...passo induttivo

- Consideriamo un **qualsiasi** percorso P da s a v che passa per uno o più nodi non contenuti in S. 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 maggiore o uguale di d'[v] altrimenti l'algoritmo avrebbe scelto y al posto di v. Infatti se per assurdo la lunghezza ℓ(P')+ℓ(x,y) fosse minore di d'(v) allora si avrebbe d'(y)≤ d(v) in quanto d'(y)≤ d(x)+ℓ(x,y)≤ℓ(P')+ℓ(x,y).
- Siccome gli archi hanno lunghezze ≥ 0
  → lunghezza di P ≥
  lunghezza di P' più l'arco (x,y) ≥ d'(v).



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
Let Q be a priority queue of the nodes u s.t. u is not in S
For each u not in S, the key key(u), associated with u in Q, stores
the distance d'(u)
For each u in S, we store a distance d(u)
Set key(s)=0 and for each u diverso da s set key(u)=infinito
For each u in V
  InsertQ,u)
While  S diverso da V
  u <-- ExtractMin(Q)
  d(u)=key(u)
  Add u to S
  For each edge e=(u,v)
   If  v not in S && d(u)+ ℓe< key(v)
     ChangeKey(Q,v, d(u)+ ℓ(e))
```

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

- Usiamo una priority queue Q che contiene ogni vertice u **non** in S e la chiave key(u) di u in Q è d'(u). 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 al più m ChangeKey

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

Se la coda è implementata mediante una lista non ordinata 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: $O(n^2)$

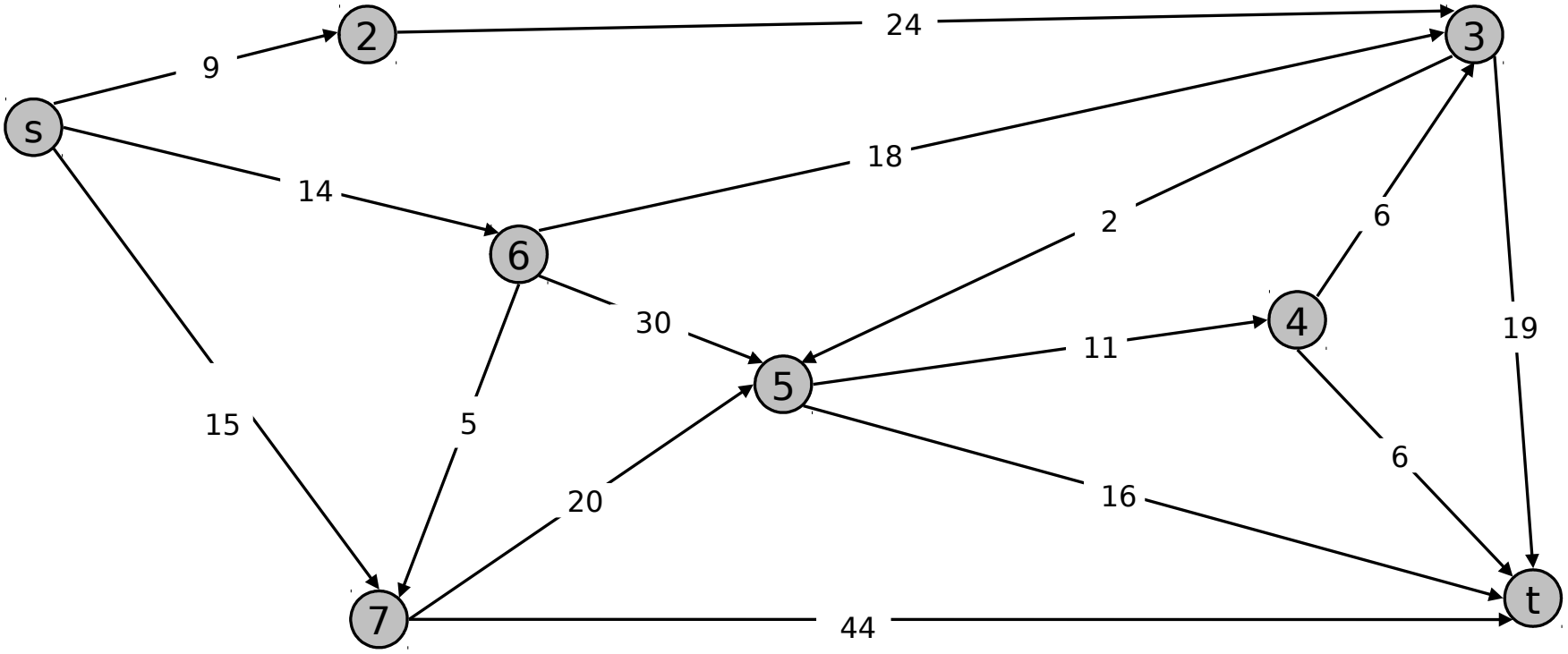
Se la coda è implementata mediante un heap:

- Inizializzazione: O(n) con costruzione bottom up oppure O(n logn) con n inserimenti
- While: O(nlog n) per le n ExtractMin; O(m log n) per le m ChangeKey

-->Tempo algoritmo: O(n log n+ m log n)

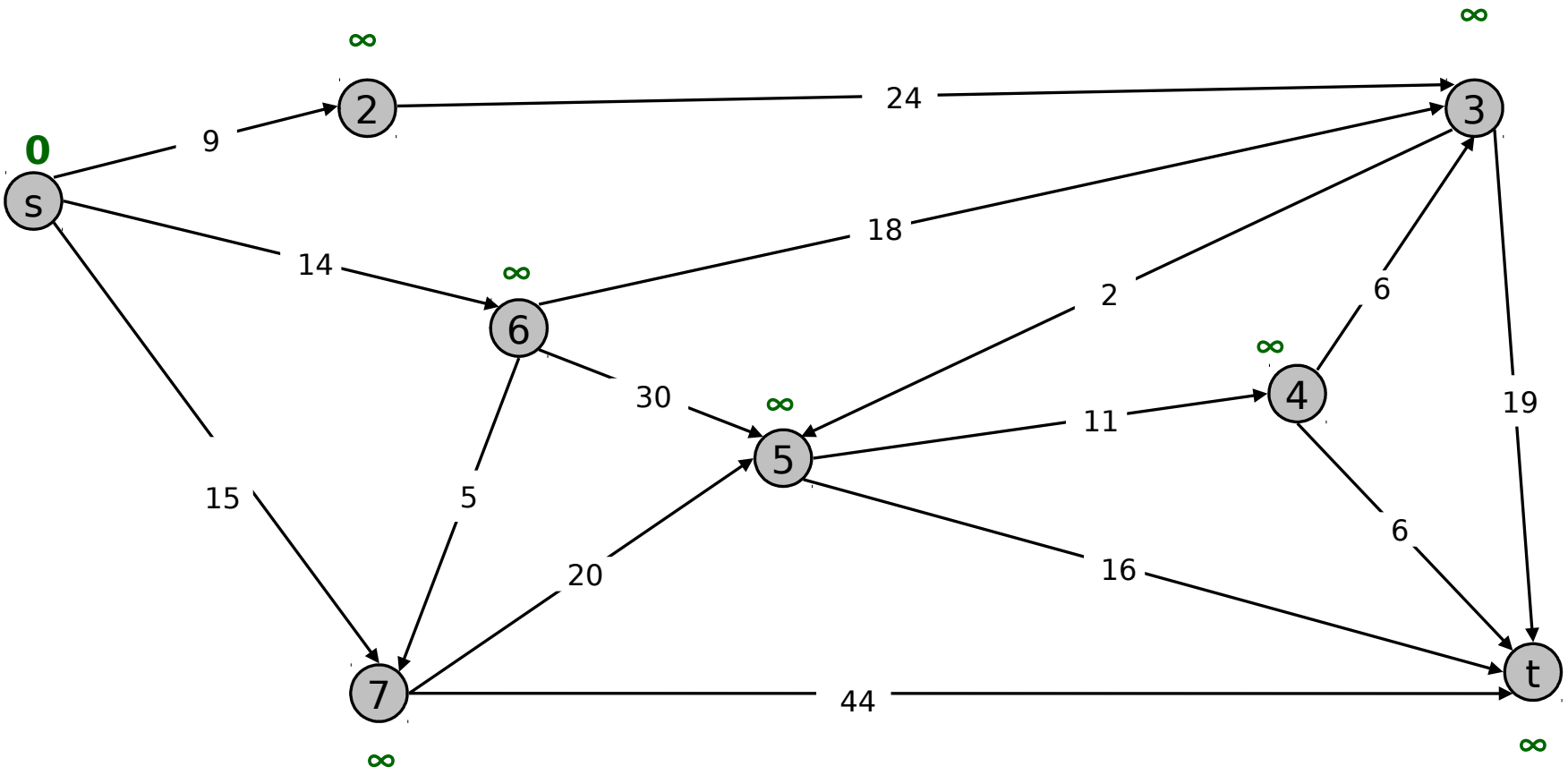
Algoritmo di Dijkstra

Trova i percorsi più corti



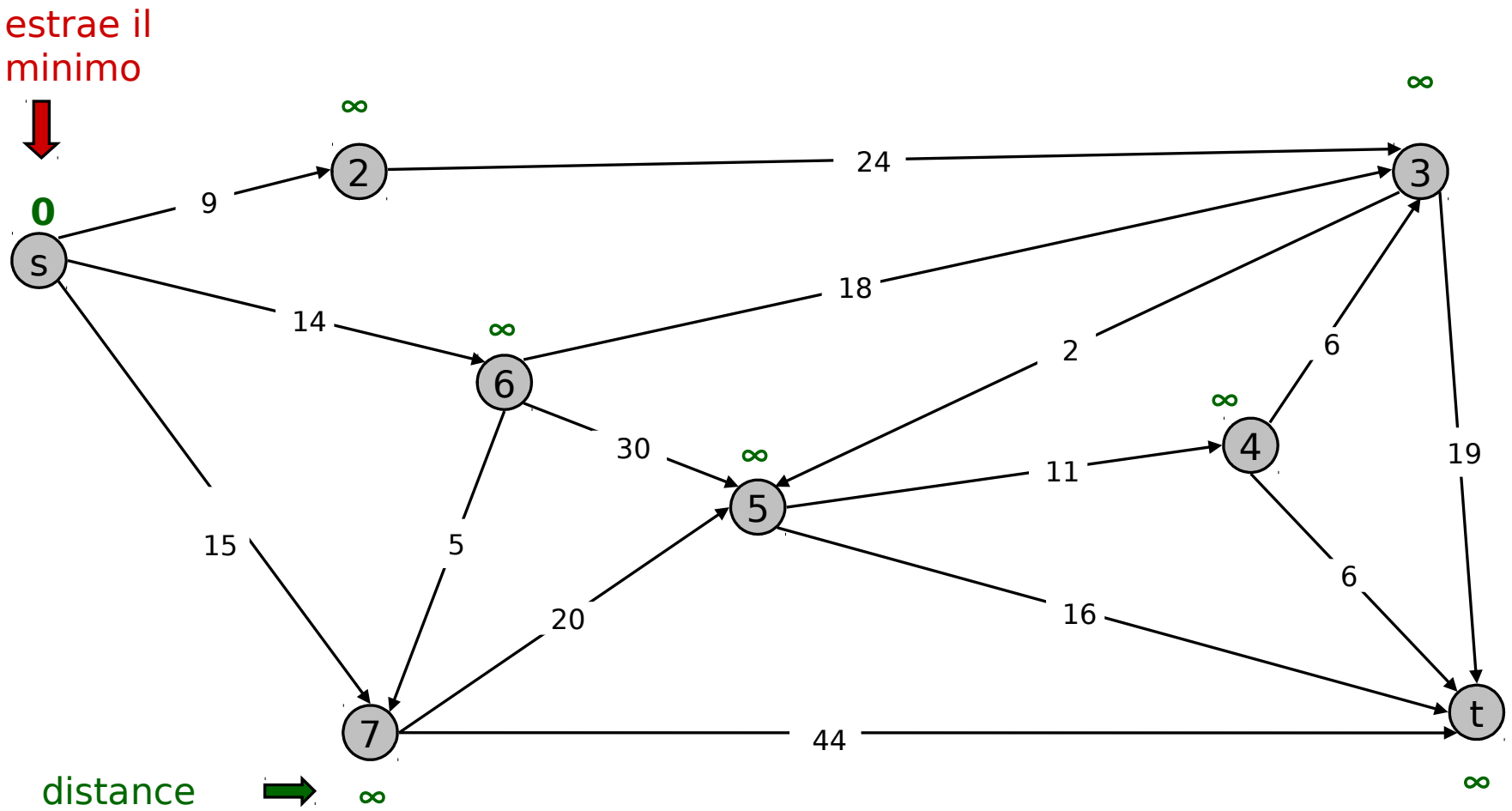
Algoritmo di Dijkstra

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



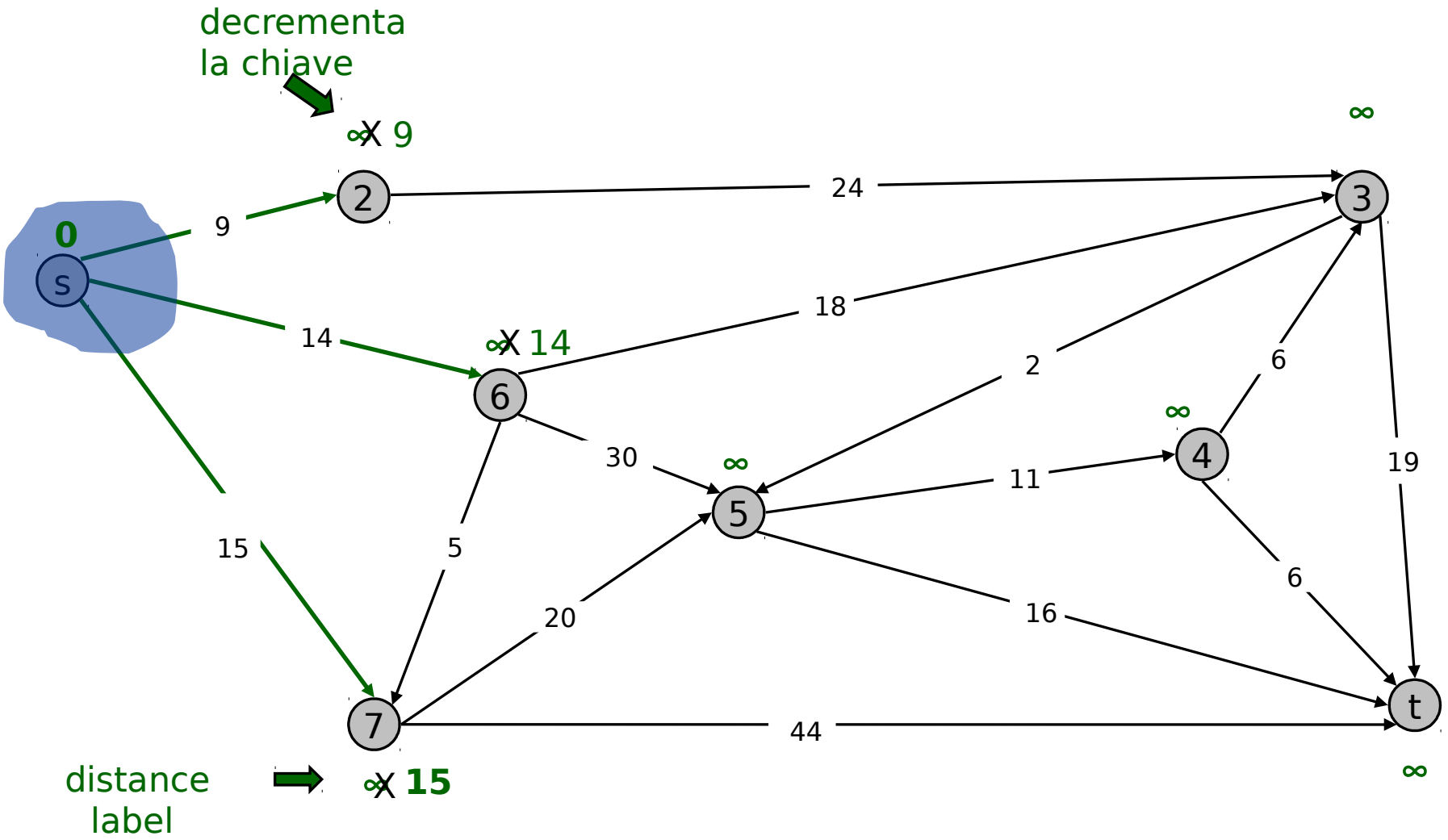
Algoritmo di Dijkstra

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



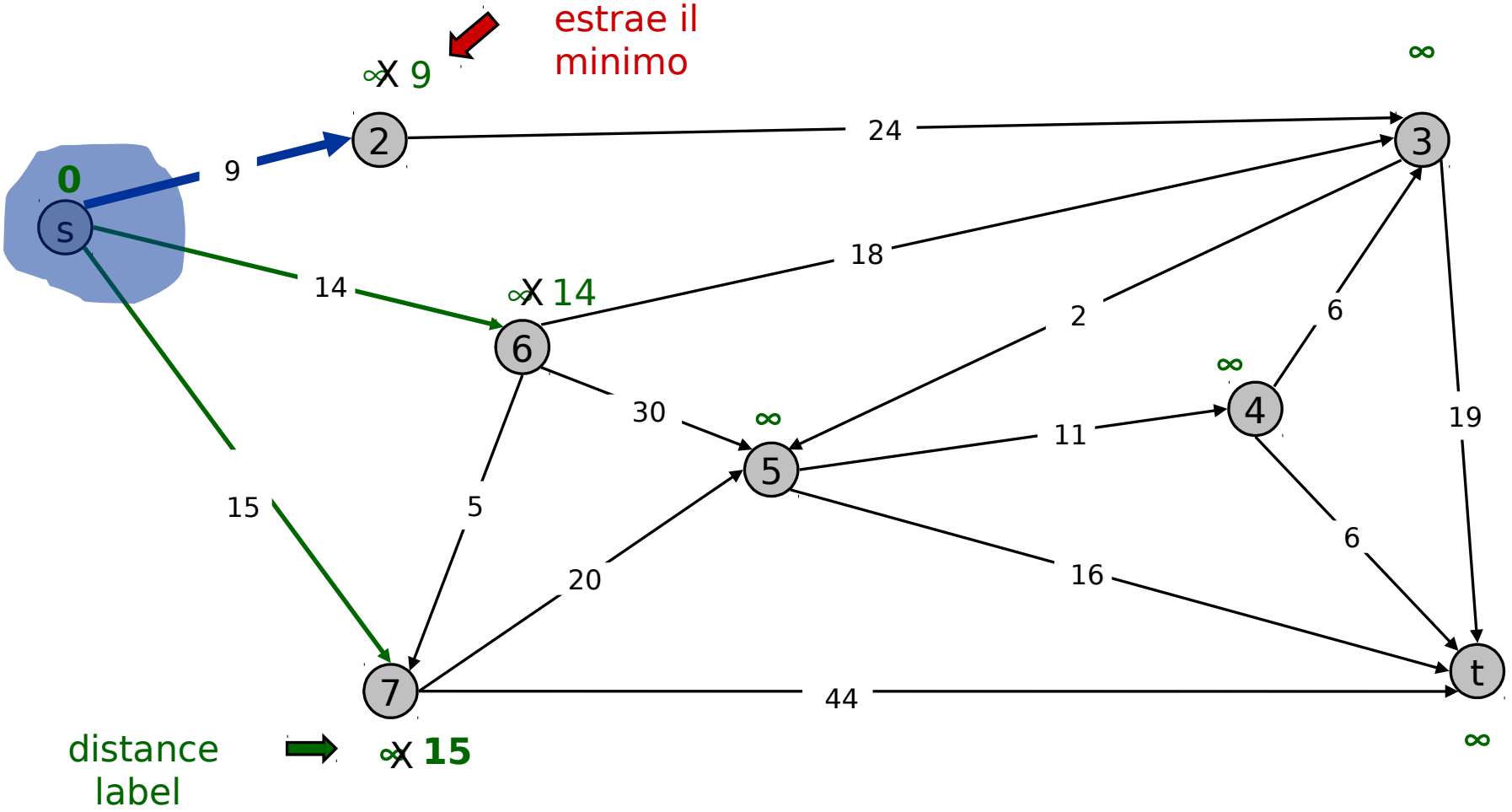
Algoritmo di Dijkstra

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



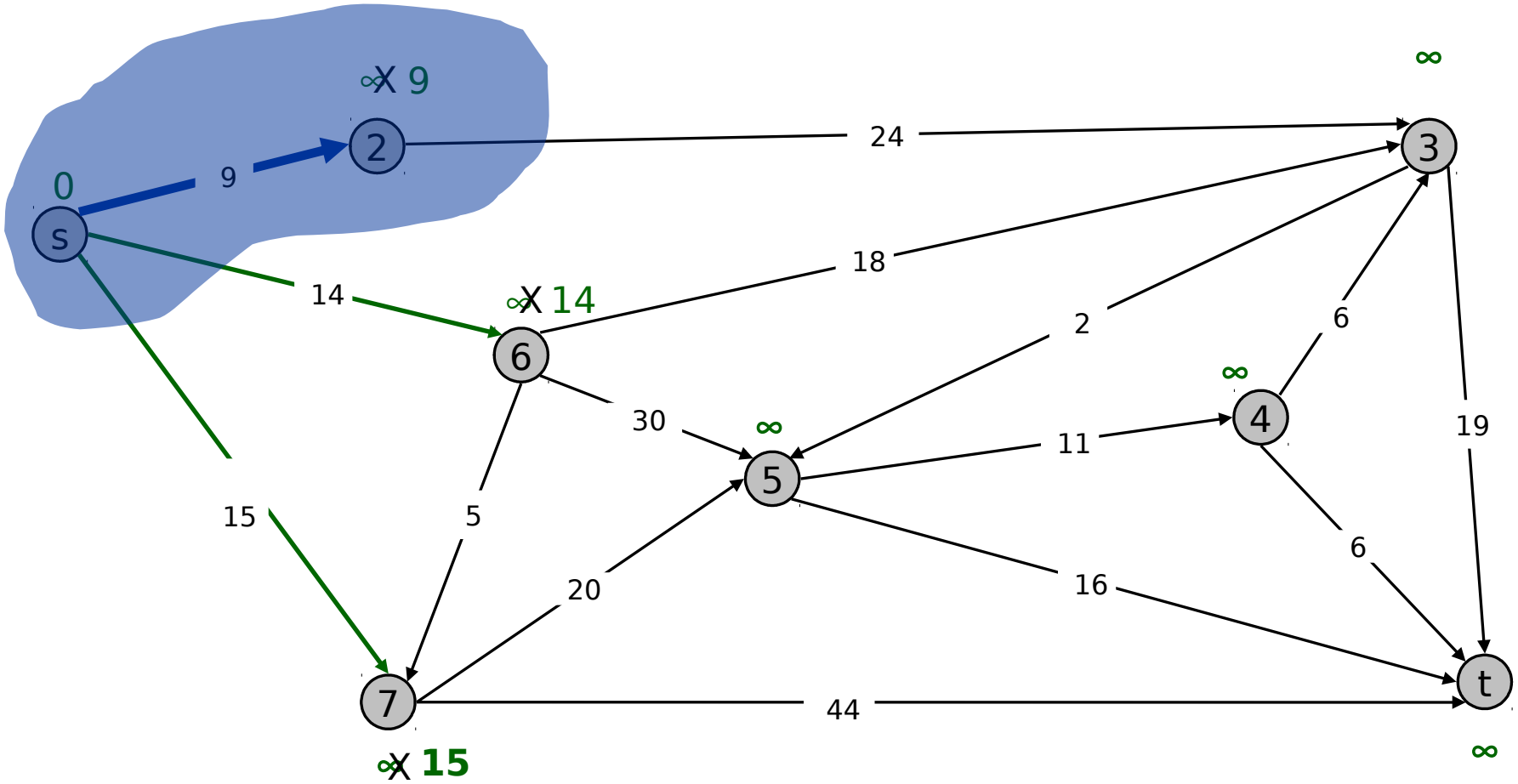
Algoritmo di Dijkstra

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



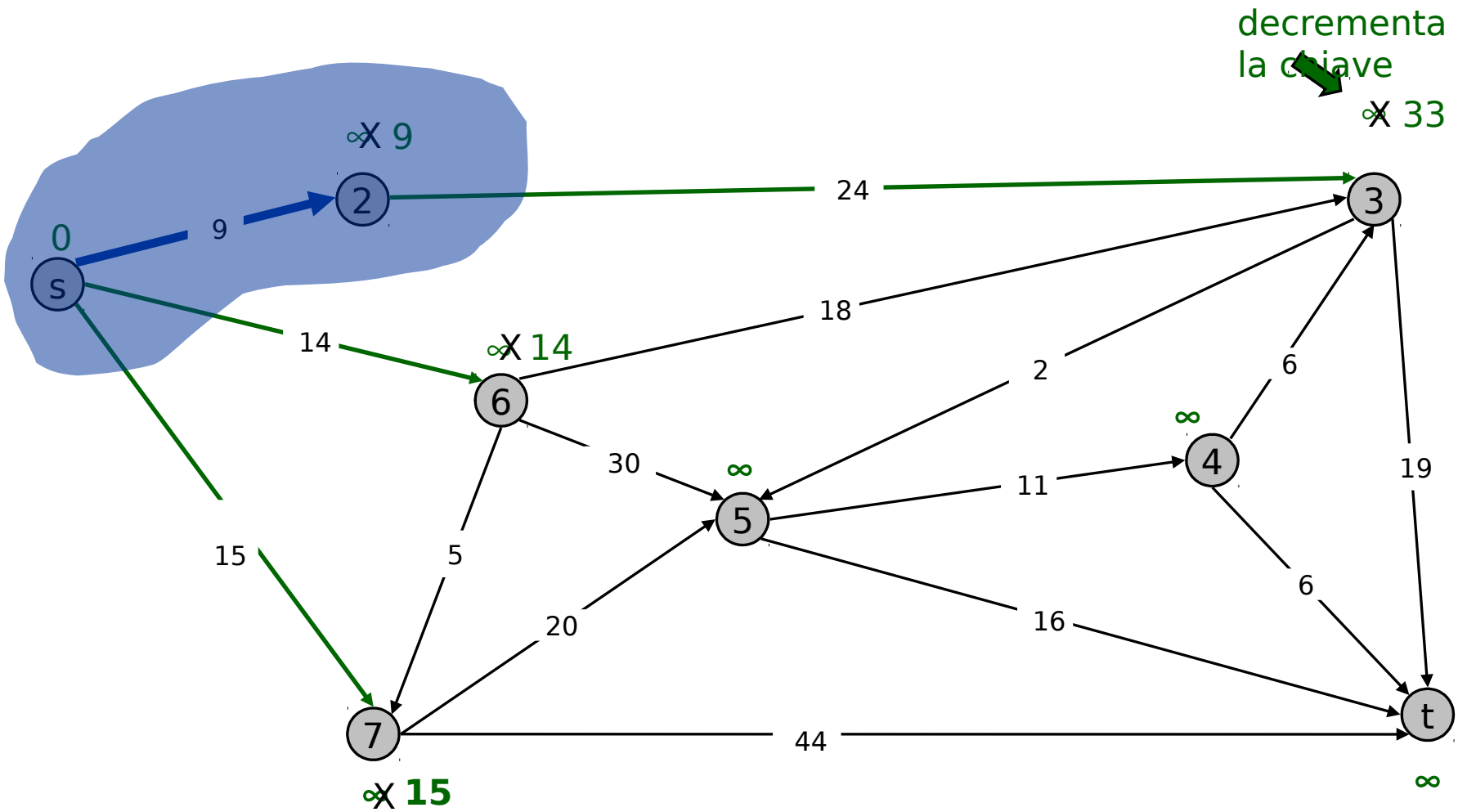
Algoritmo di Dijkstra

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



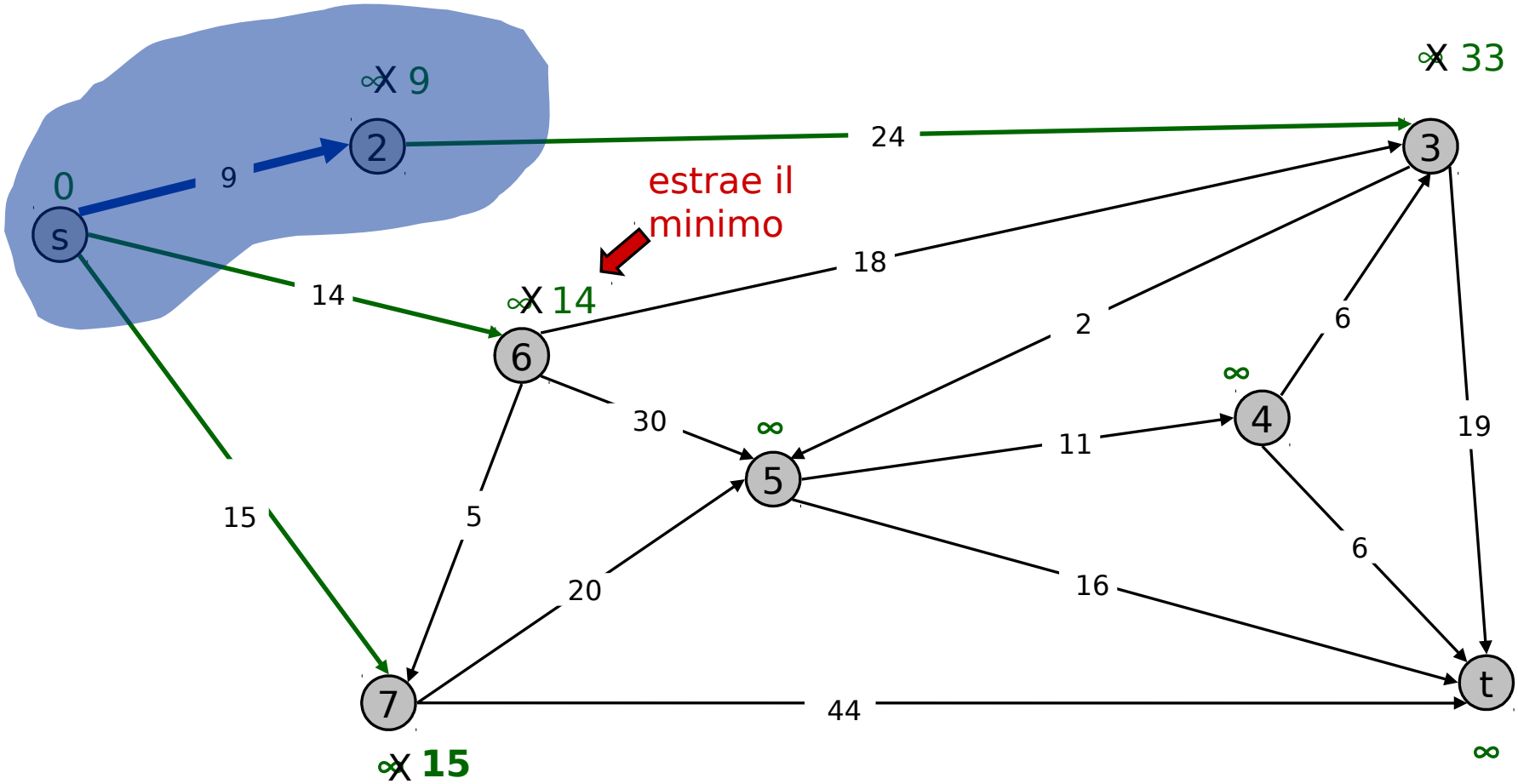
Algoritmo di Dijkstra

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



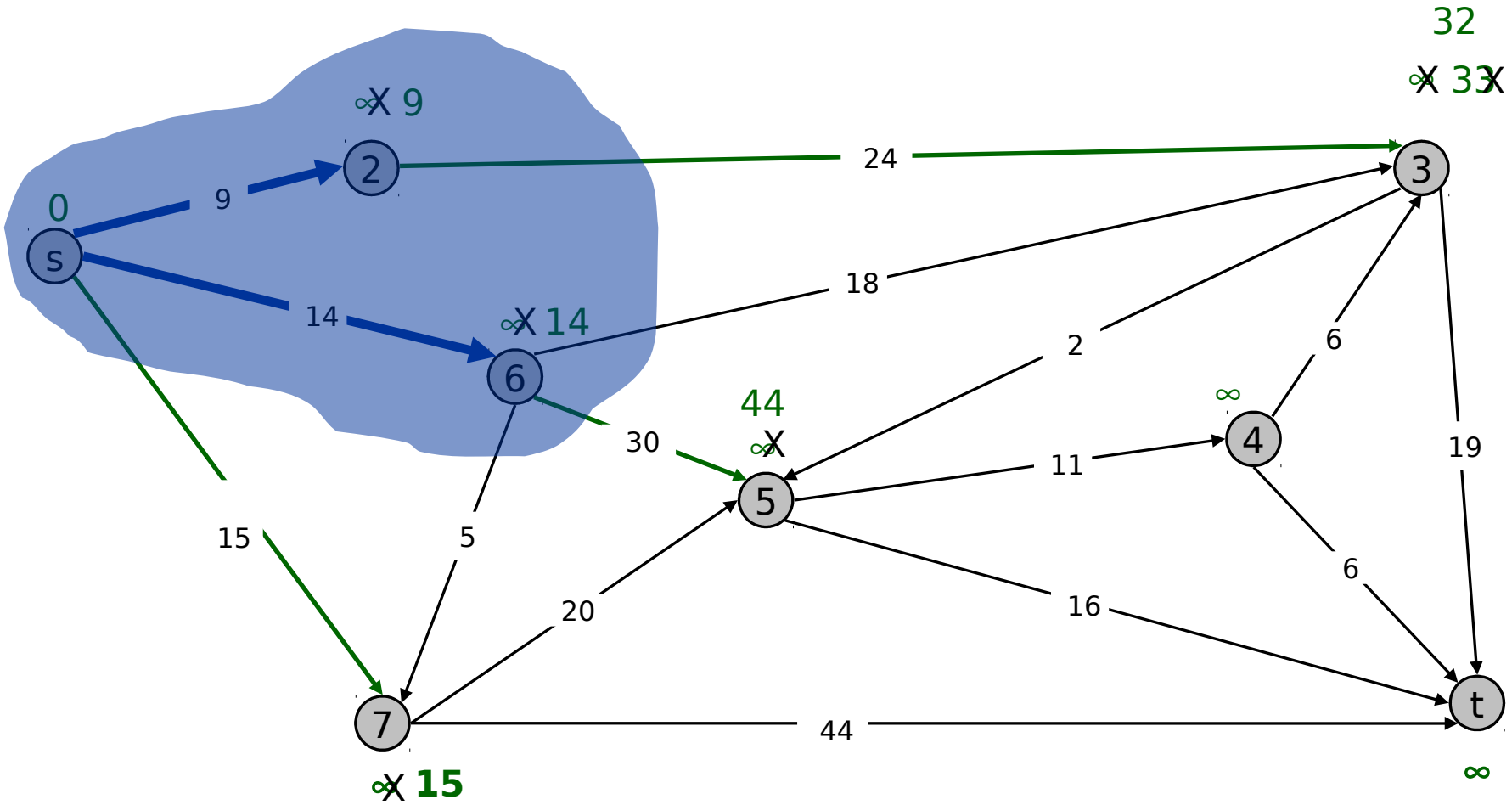
Algoritmo di Dijkstra

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



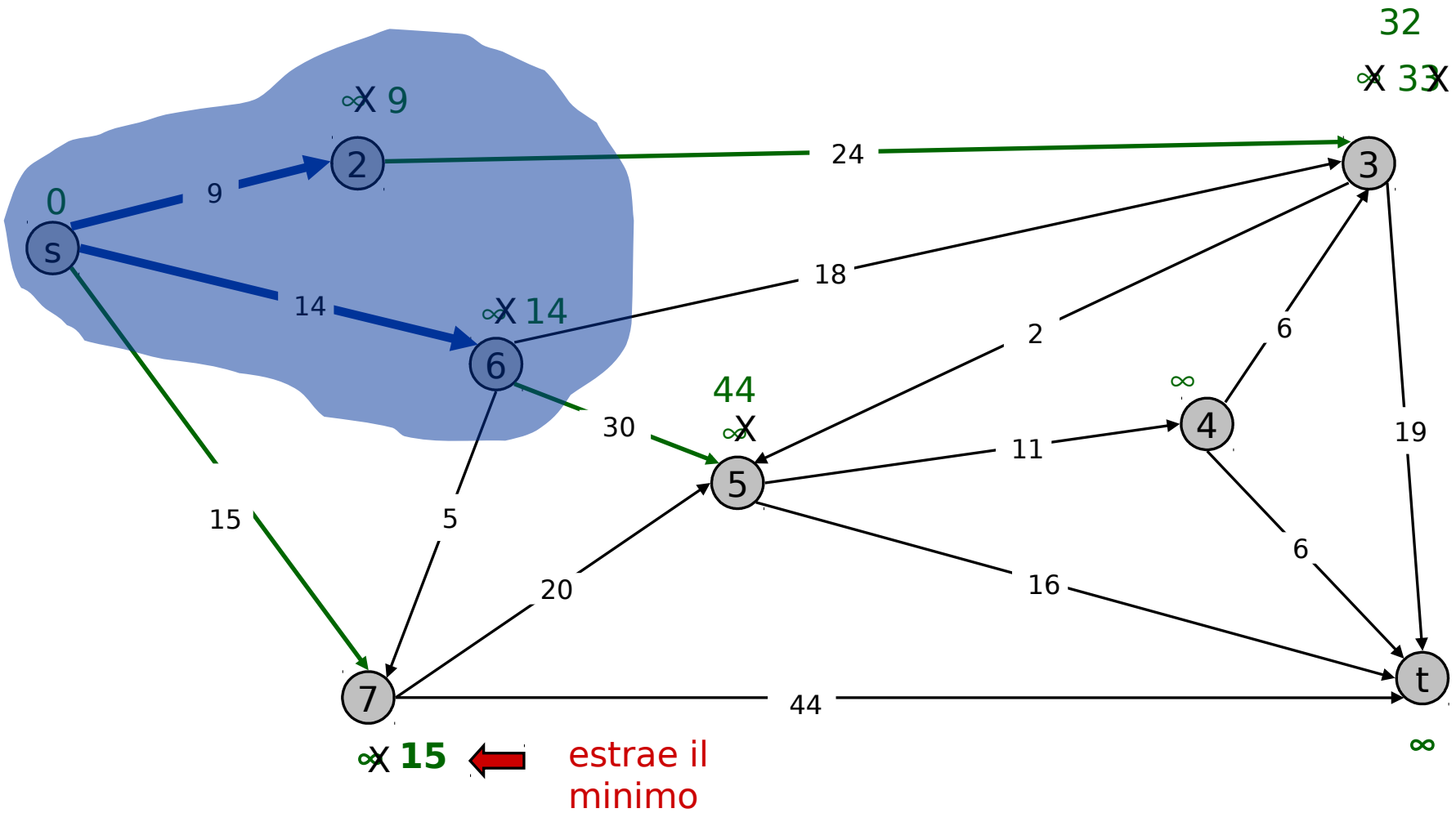
Algoritmo di Dijkstra

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



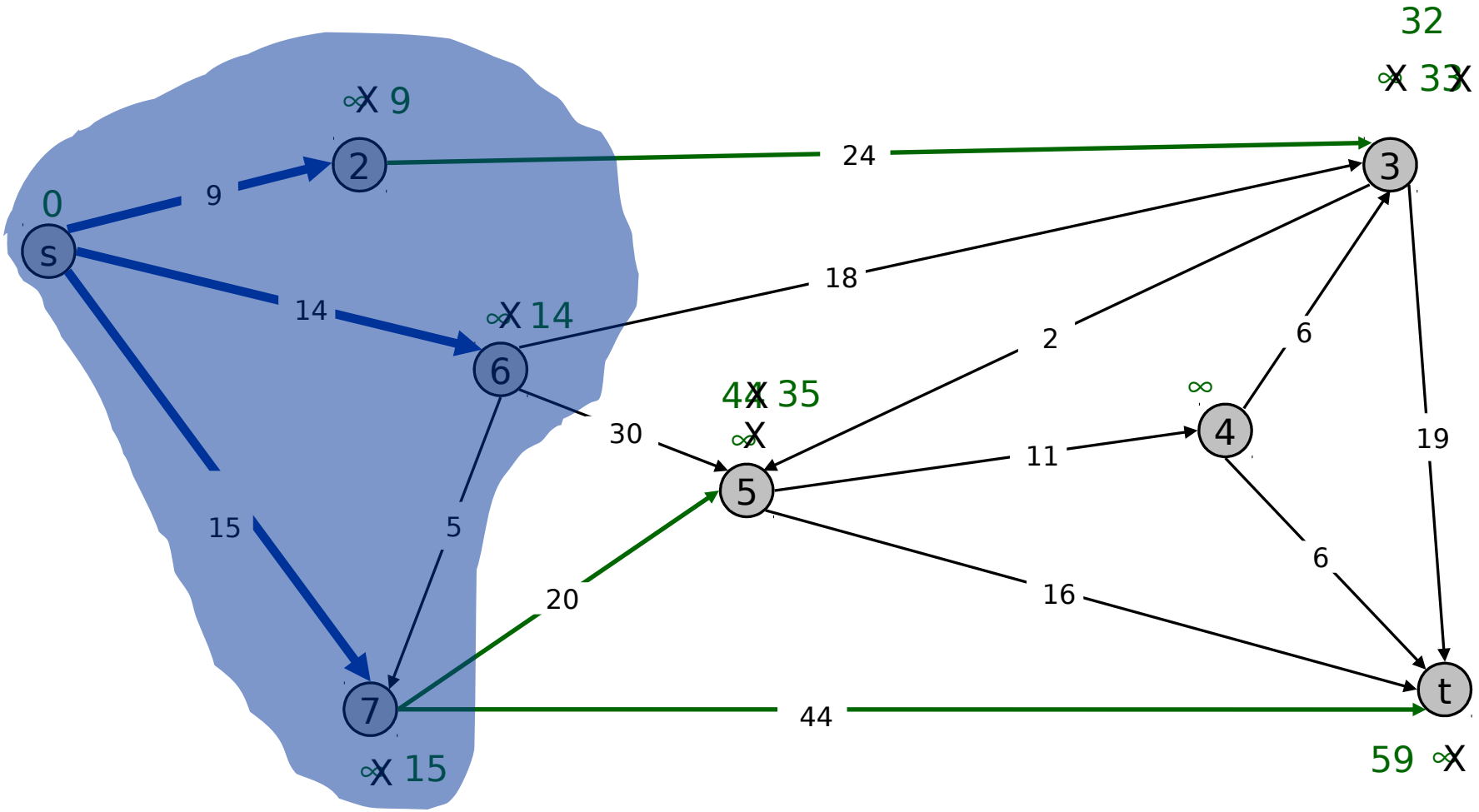
Algoritmo di Dijkstra

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



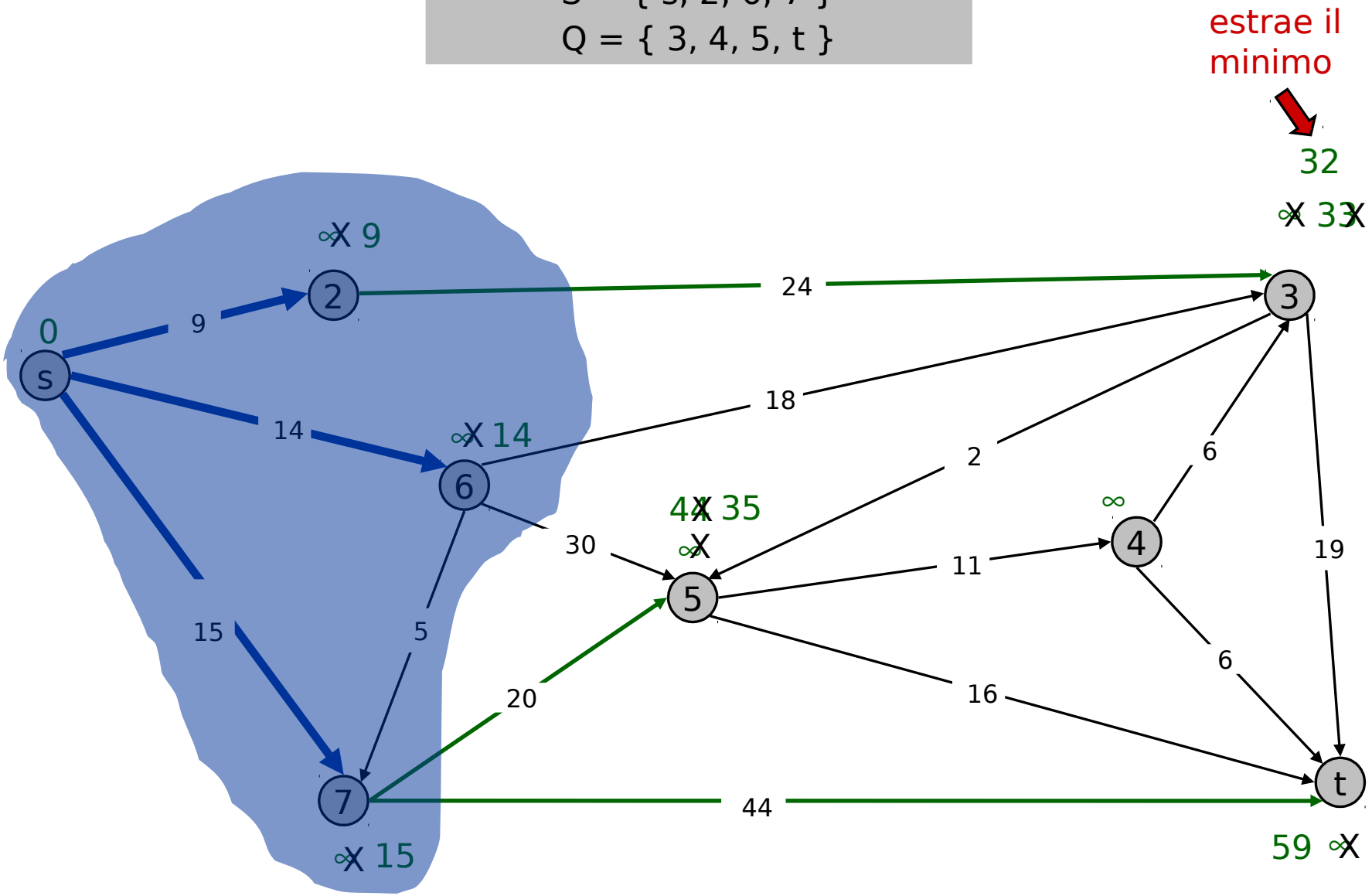
Algoritmo di Dijkstra

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



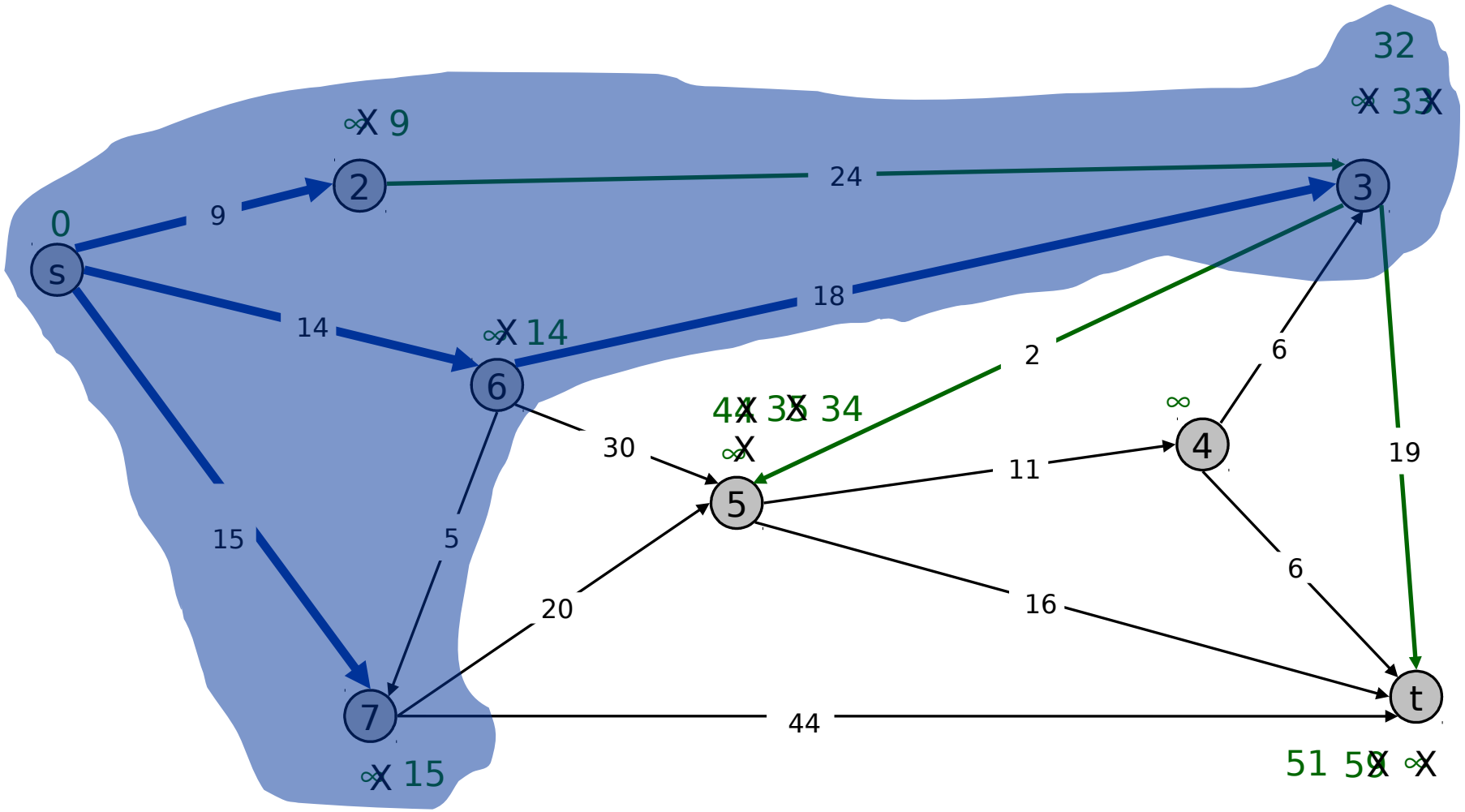
Algoritmo di Dijkstra

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



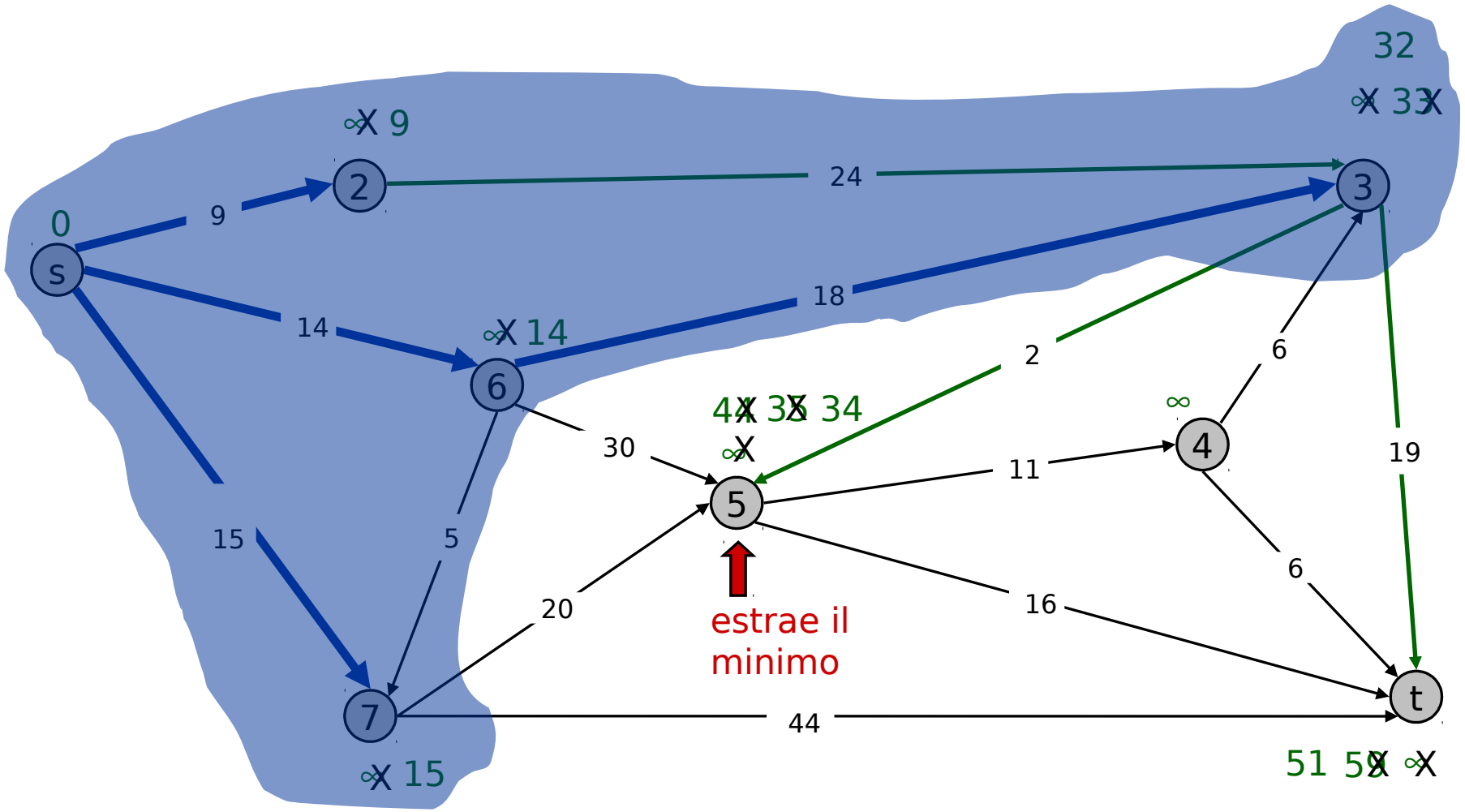
Algoritmo di Dijkstra

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



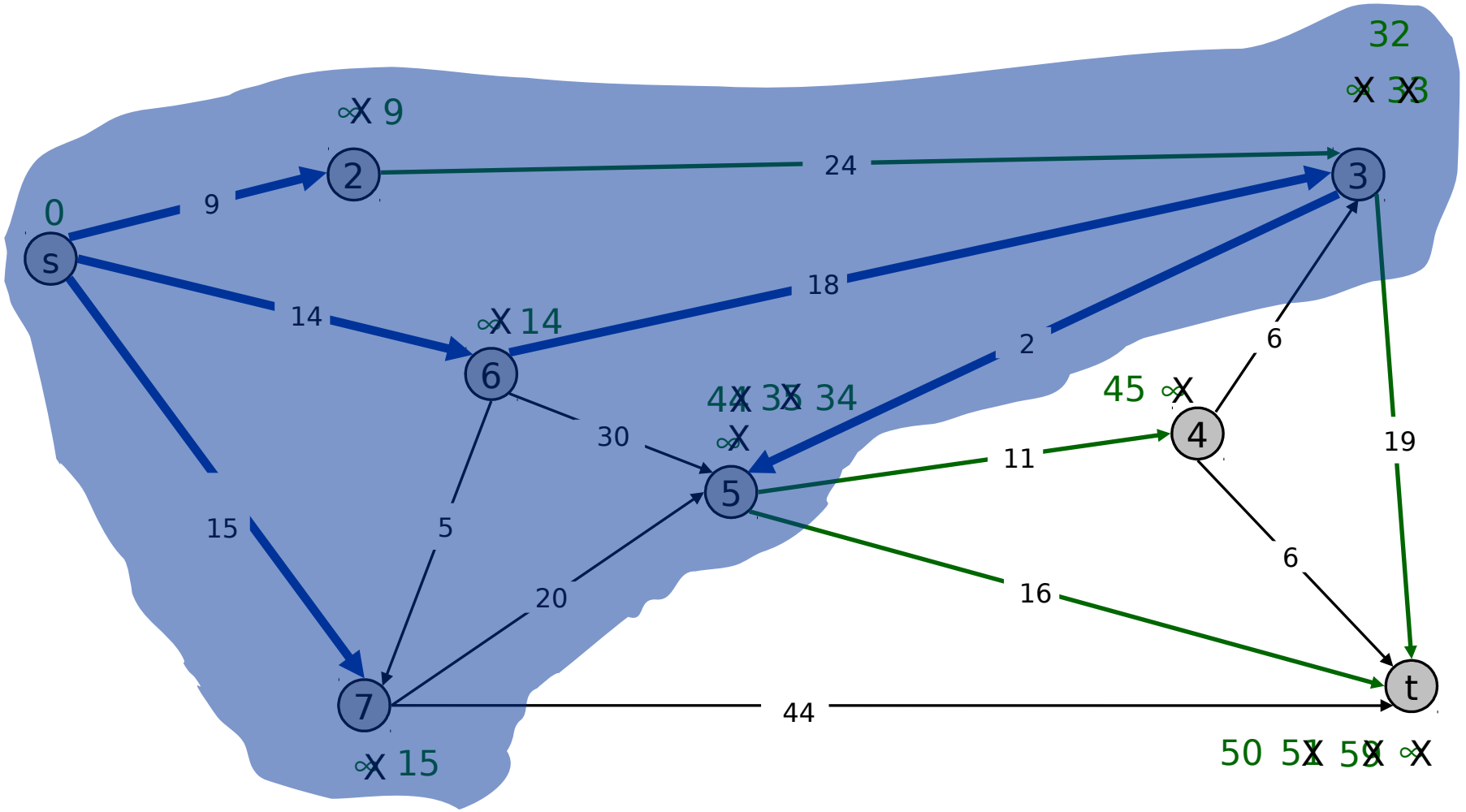
Algoritmo di Dijkstra

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



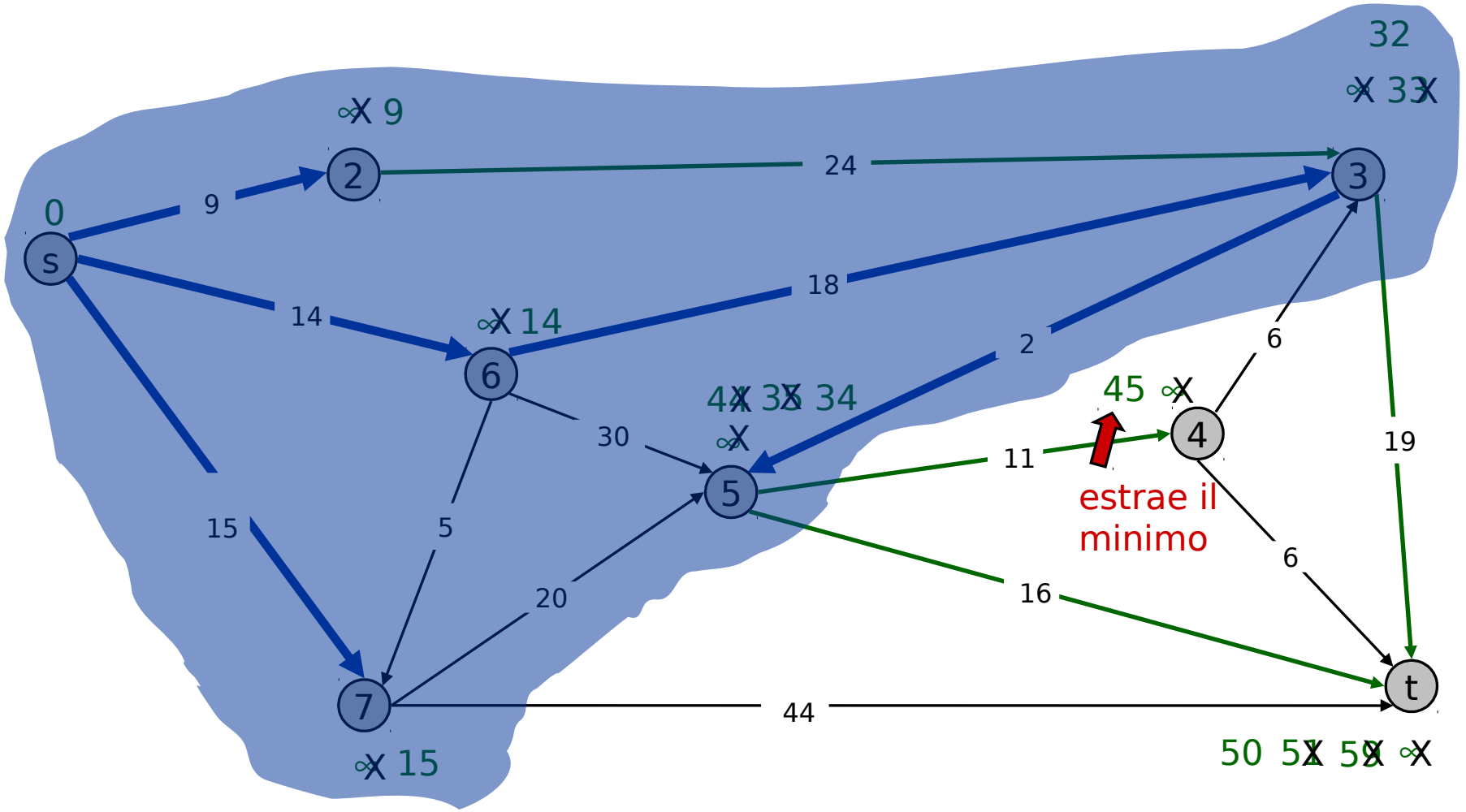
Algoritmo di Dijkstra

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



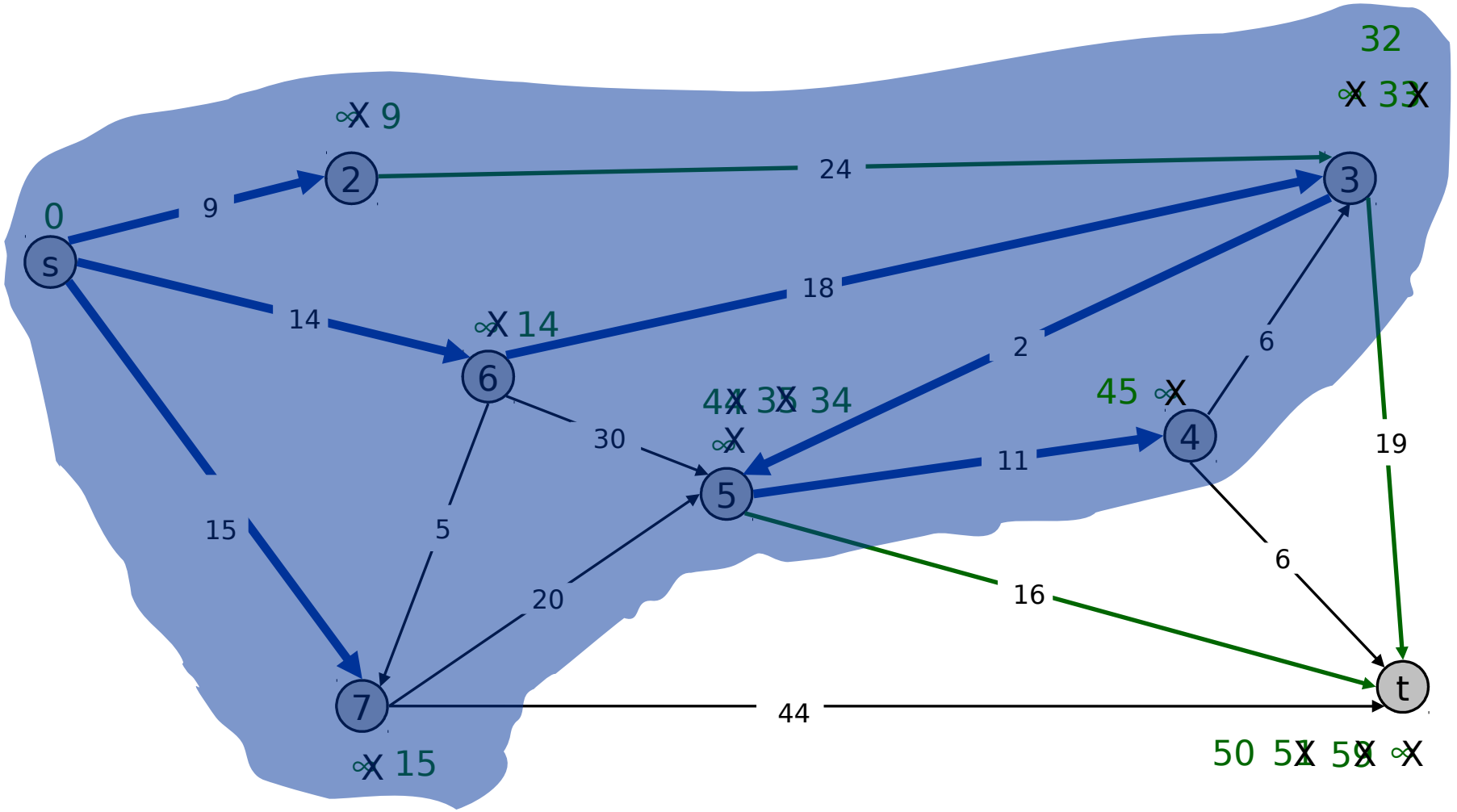
Algoritmo di Dijkstra

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



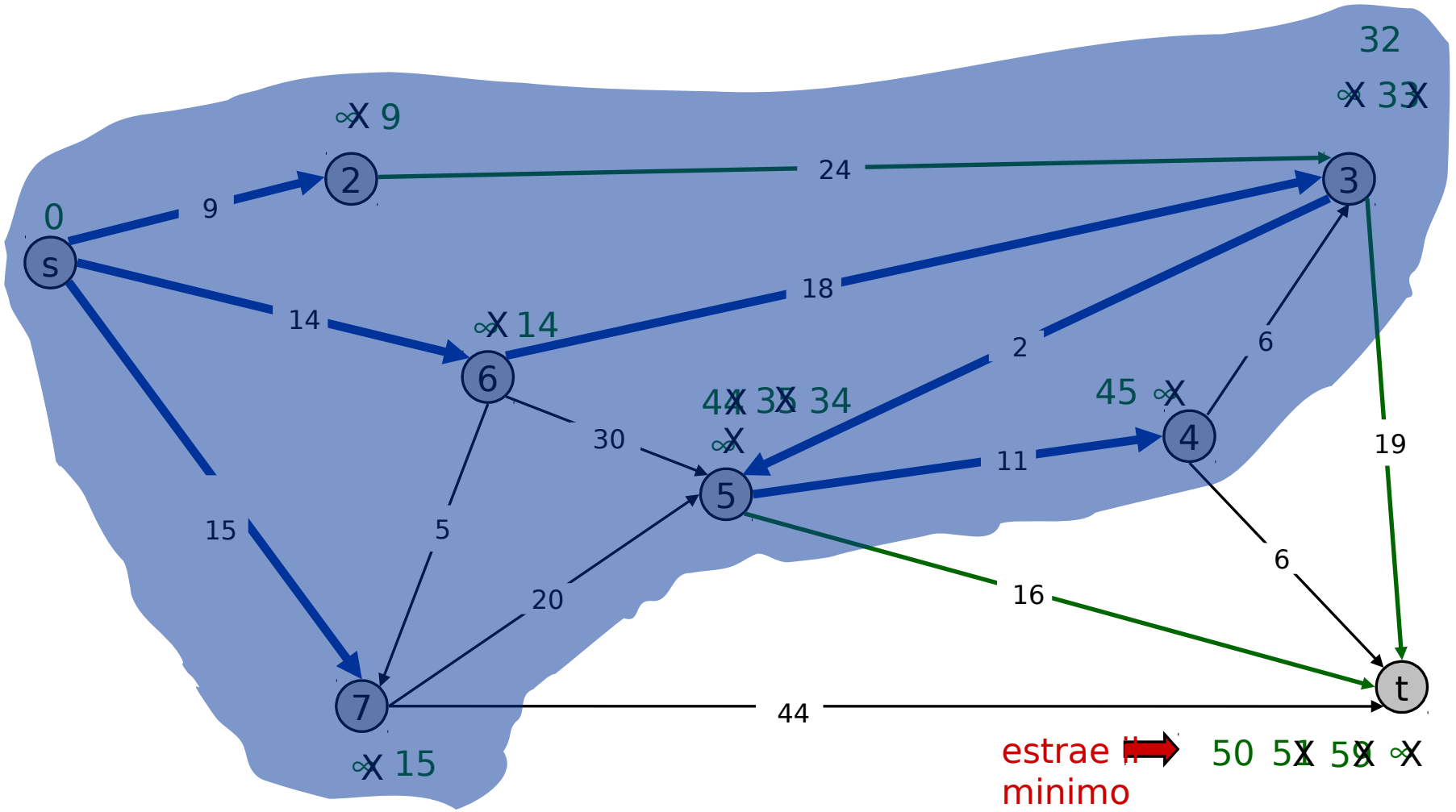
Algoritmo di Dijkstra

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



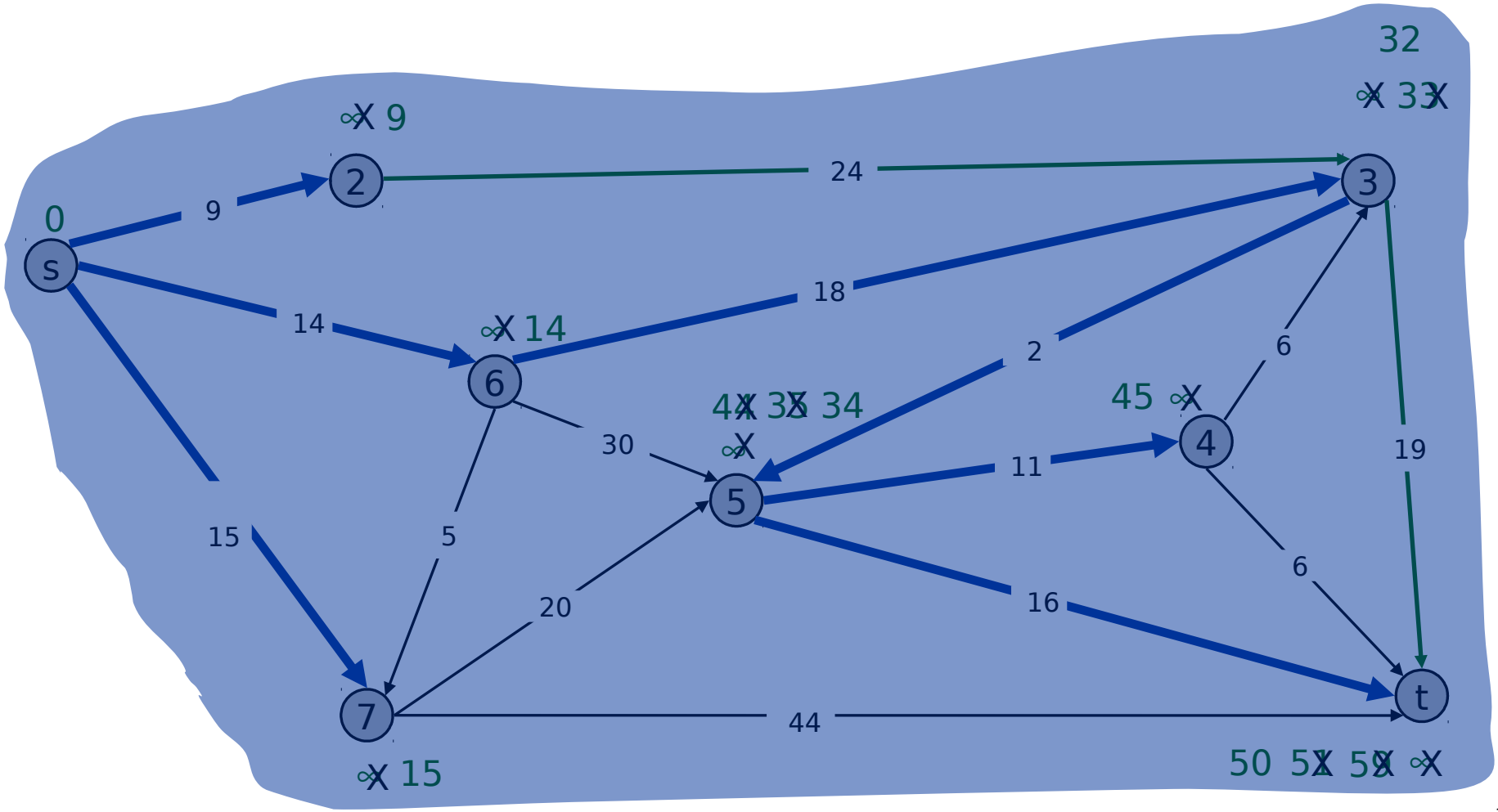
Algoritmo di Dijkstra

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 = { }