Algoritmi greedy
III parte

Progettazione di Algoritmi a.a. 2016-17

Matricole congrue a 1

Docente: Annalisa De Bonis

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

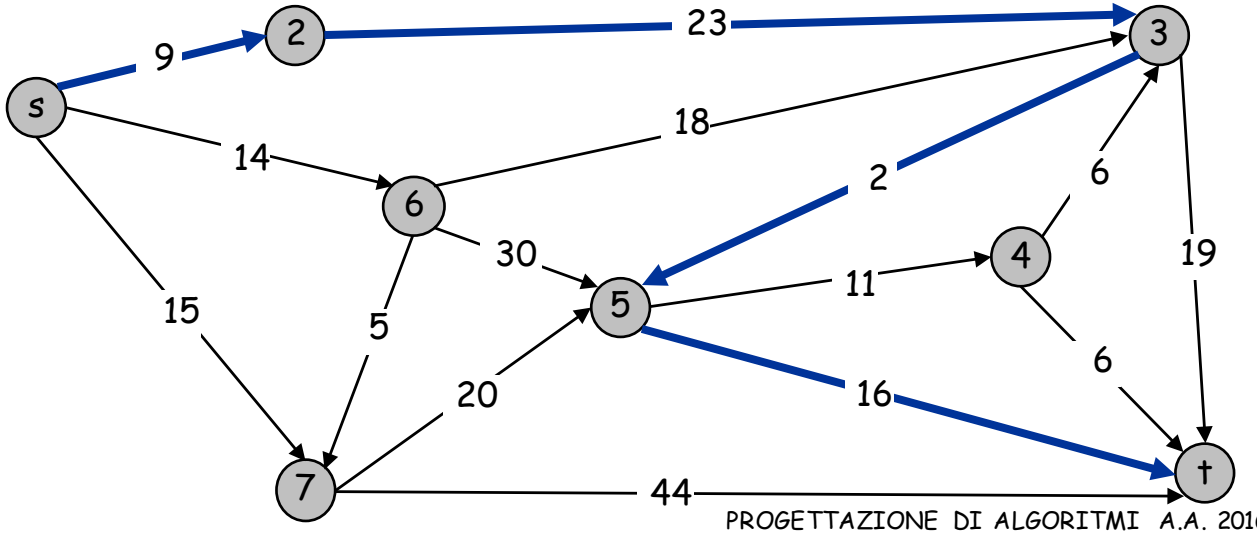
- Esempi di applicazioni dei cammini minimi in una rete
- Trovare il cammino di **tempo minimo** in una rete
- Se i pesi esprimono l'inaffidabilità delle connessioni in una rete, trovare il collegamento che è **più sicuro**

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

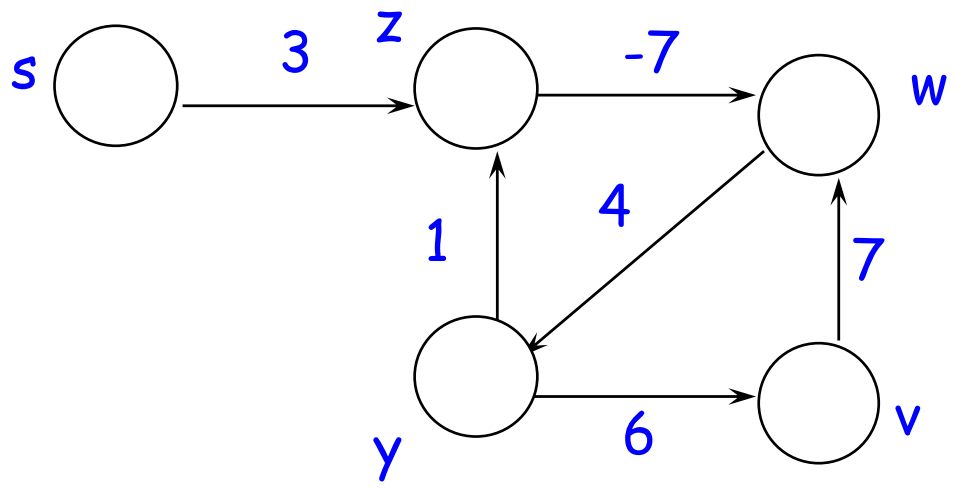
Cammini minimi

- 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



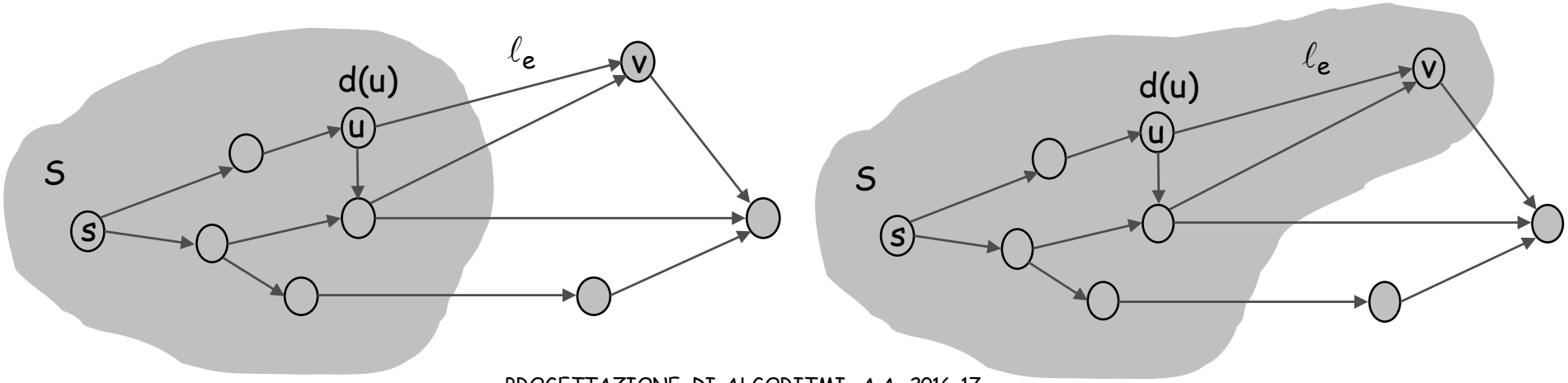
Il ciclo <z,w,y,z> ha peso -2

- 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)

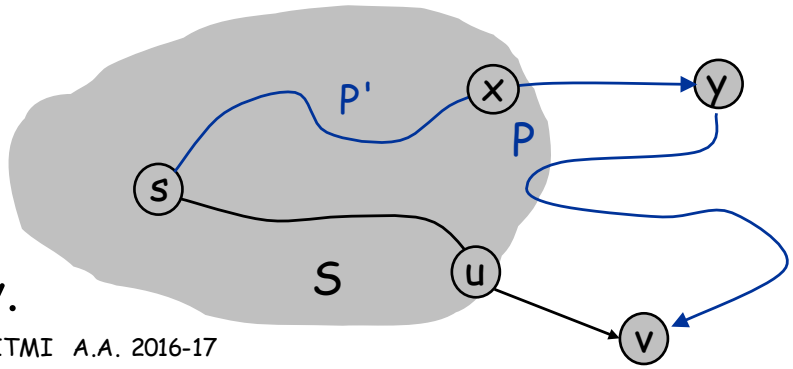
Base: |S| = 1. In questo caso S={s} e d(s)=0 per cui la tesi vale banalmente.

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 (d'[u],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 (d'[u],v) 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

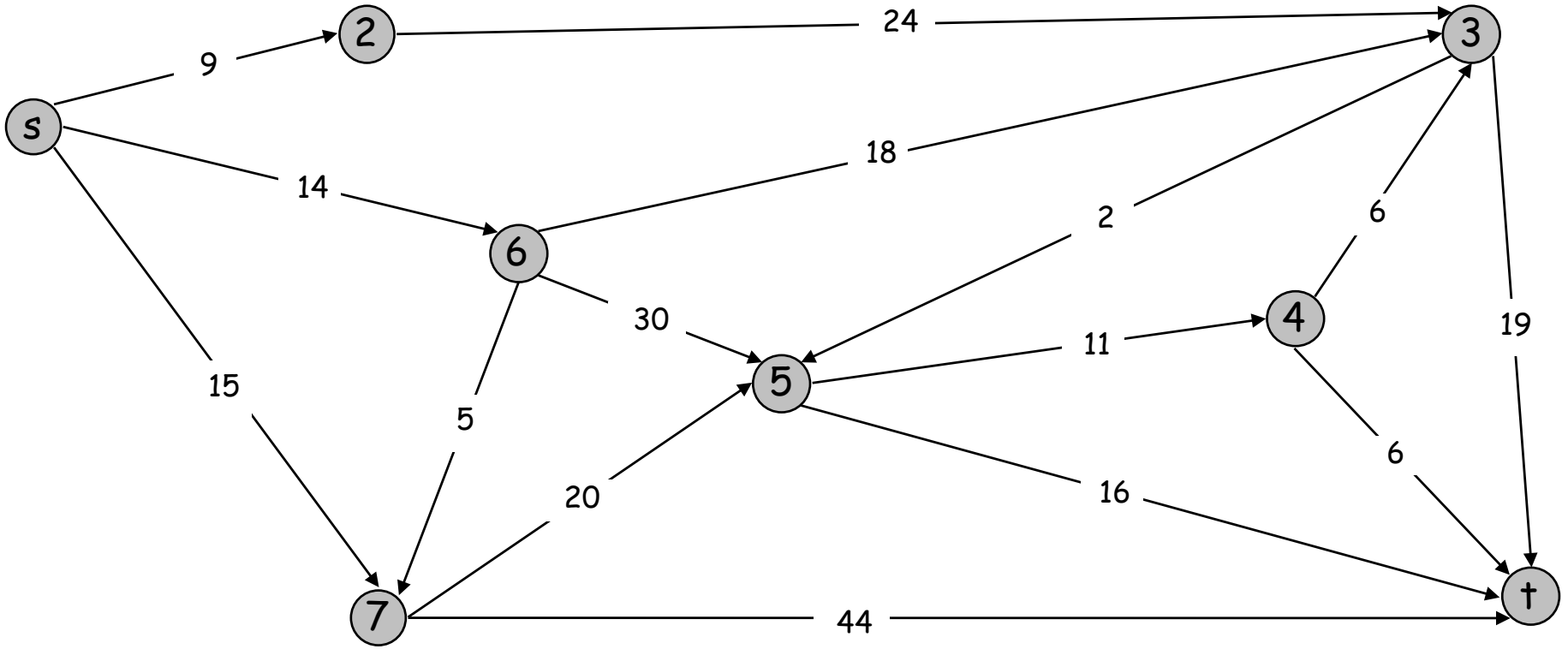
- Tempo While: $O(n)$ più il tempo per fare le n ExtractMin e le m changeKey

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: $O(n \log n + m \log n)$

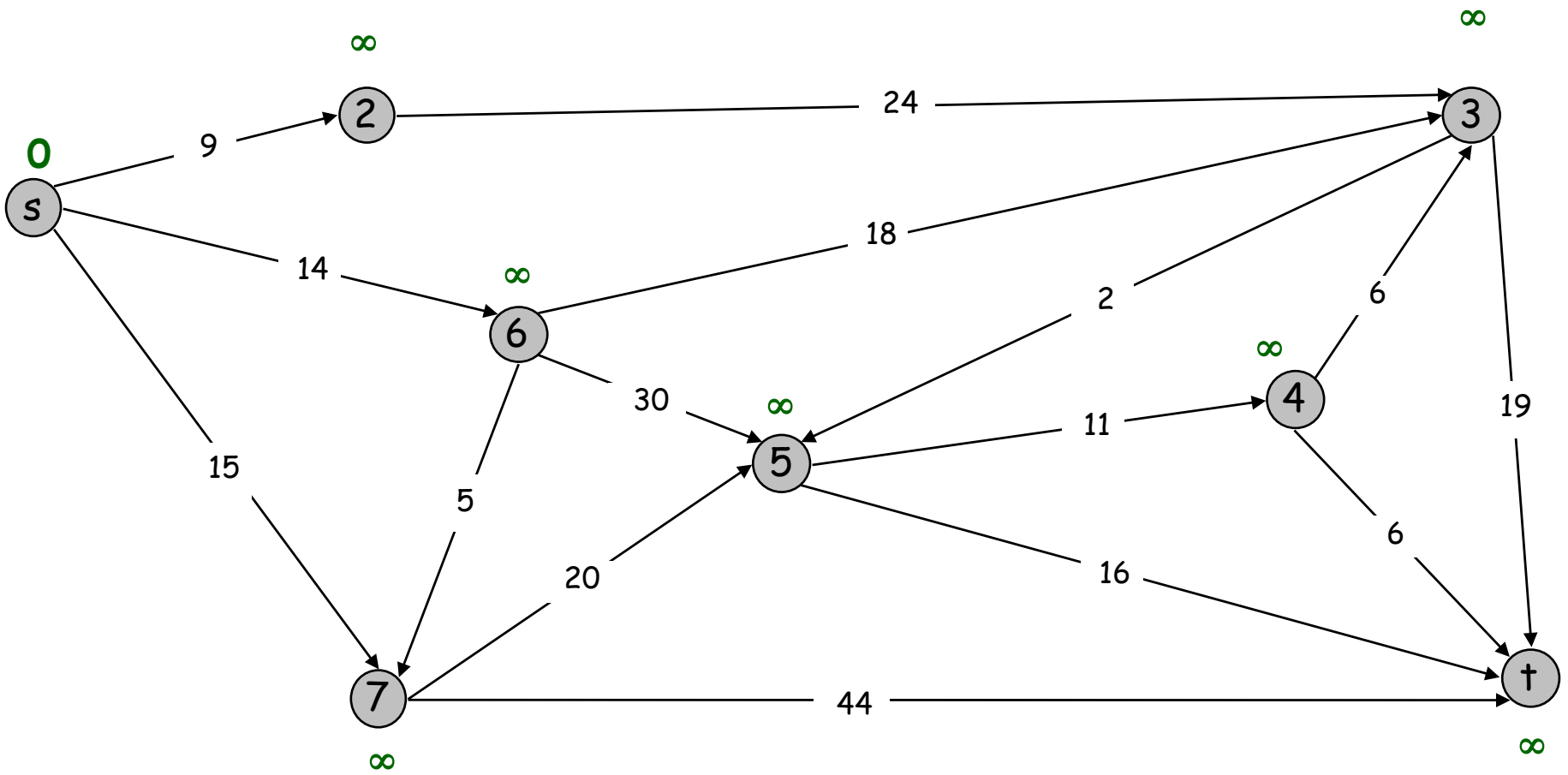
Algoritmo di Dijkstra

Trova i percorsi più corti



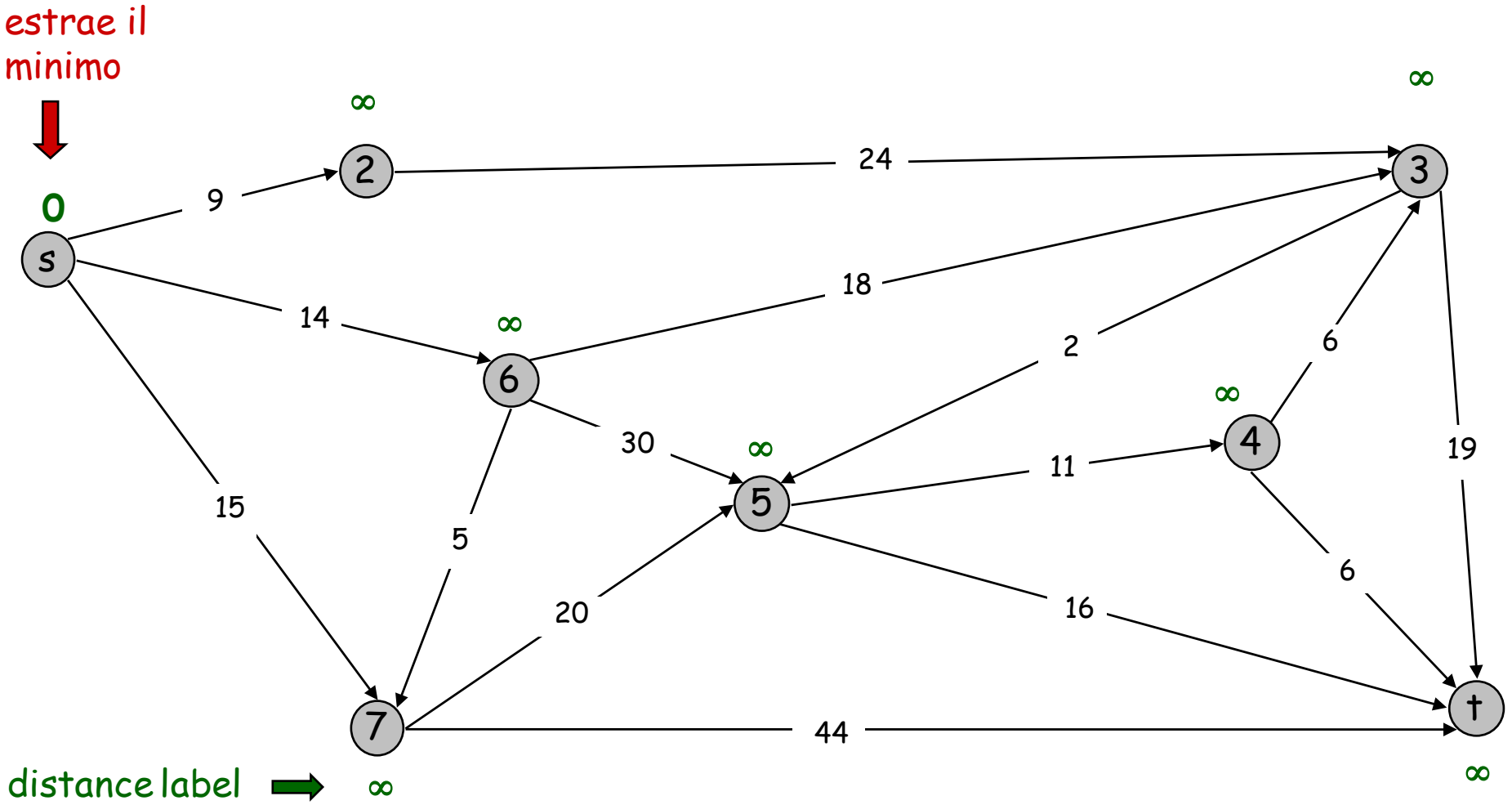
Algoritmo di Dijkstra

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



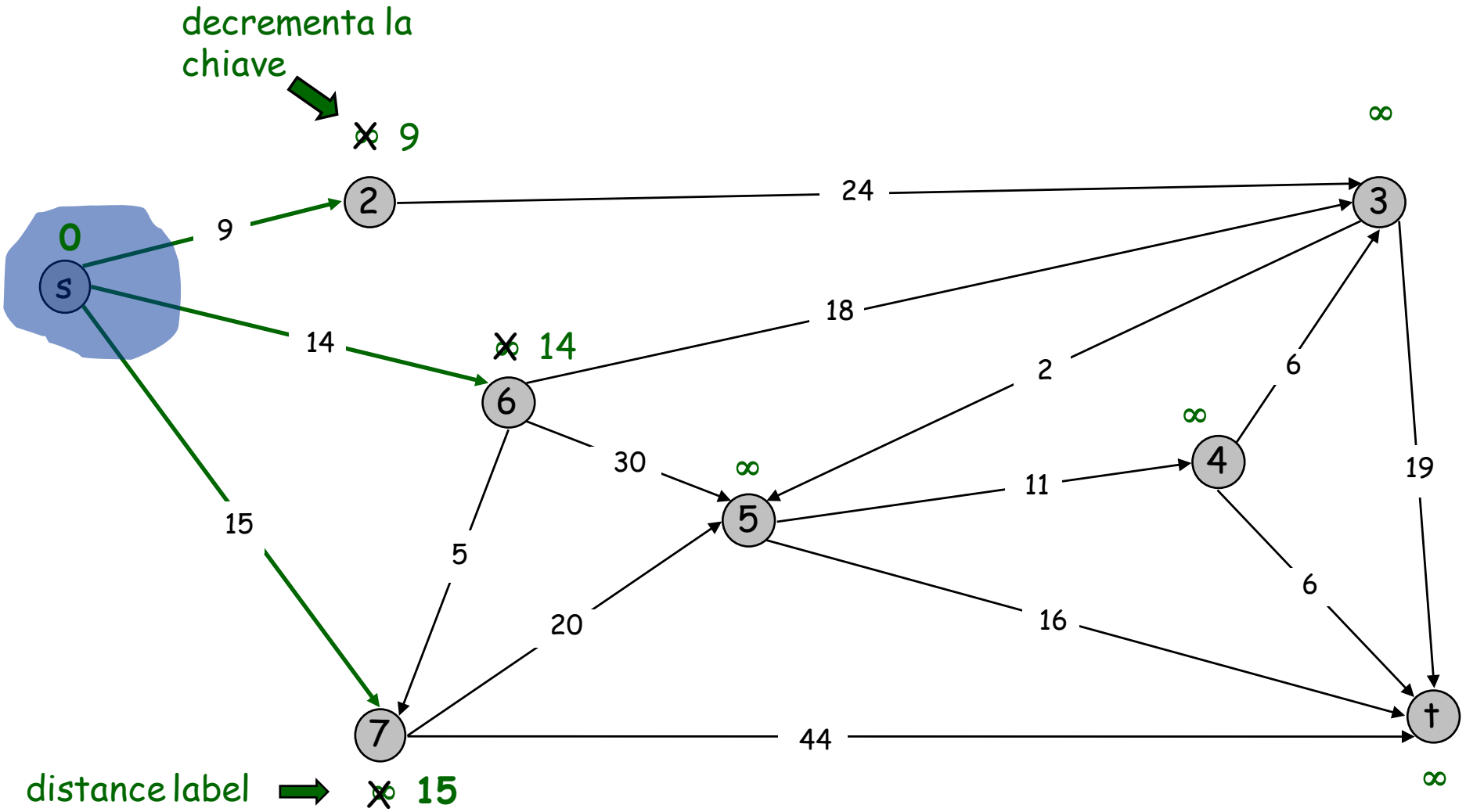
Algoritmo di Dijkstra

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



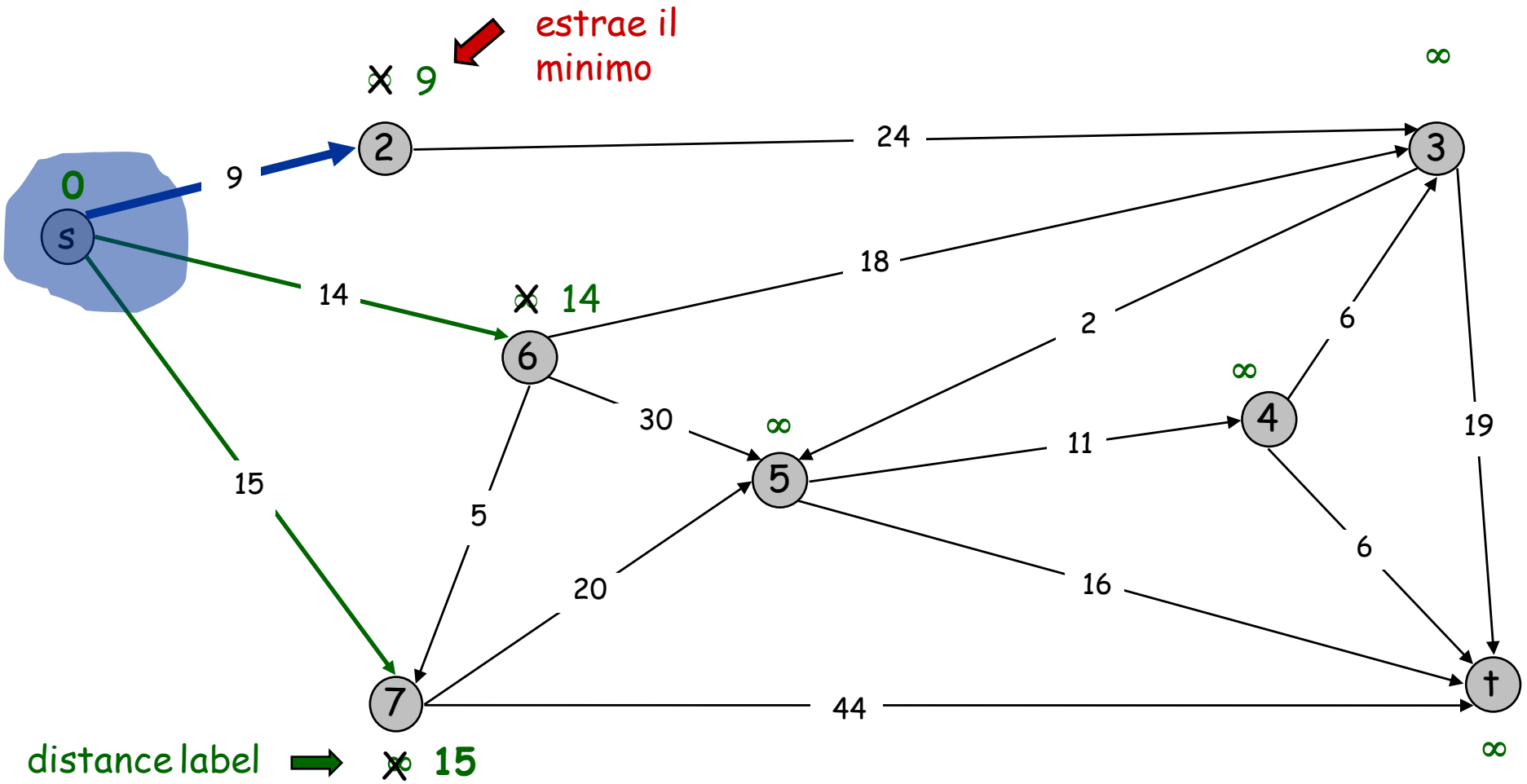
Algoritmo di Dijkstra

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



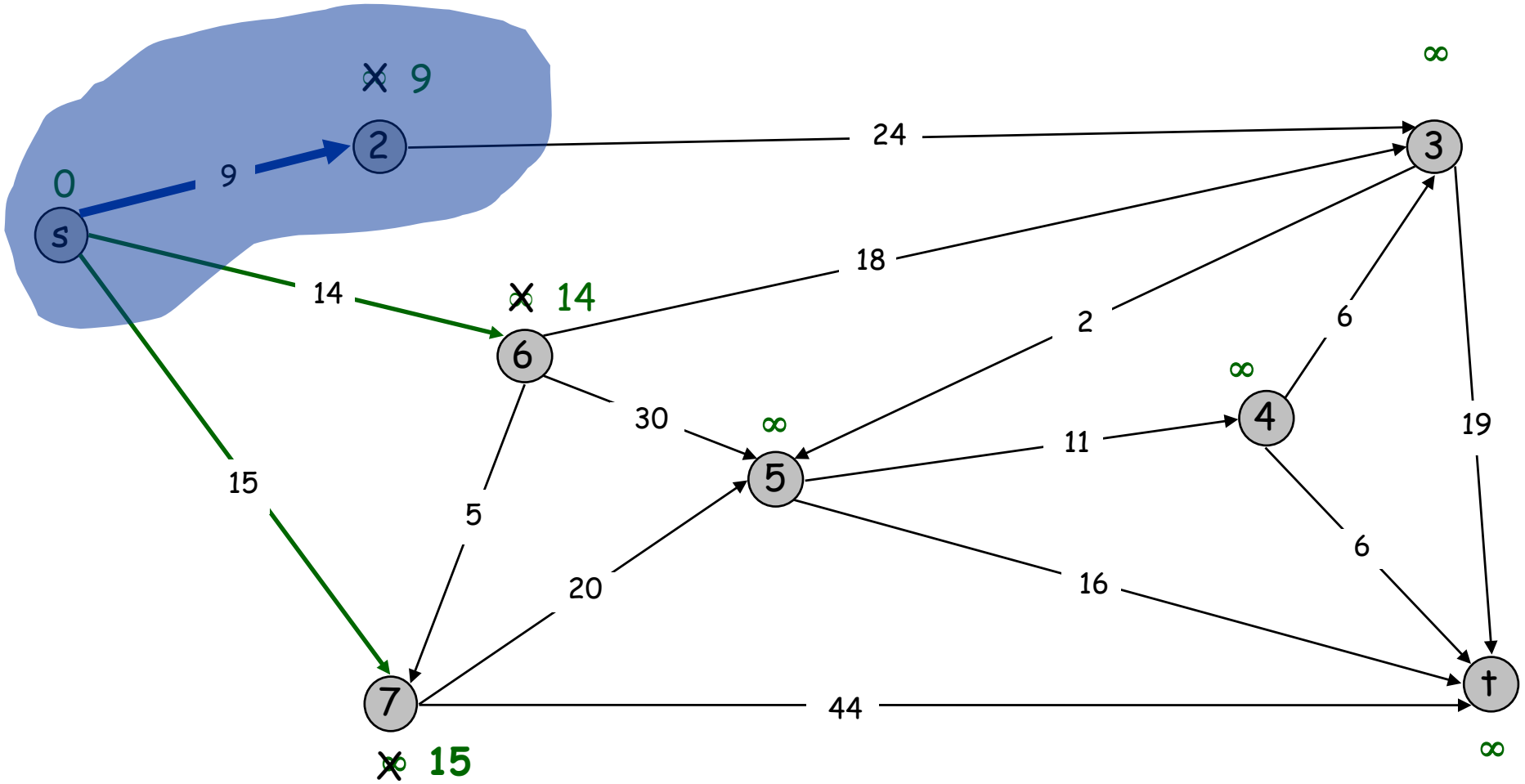
Algoritmo di Dijkstra

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



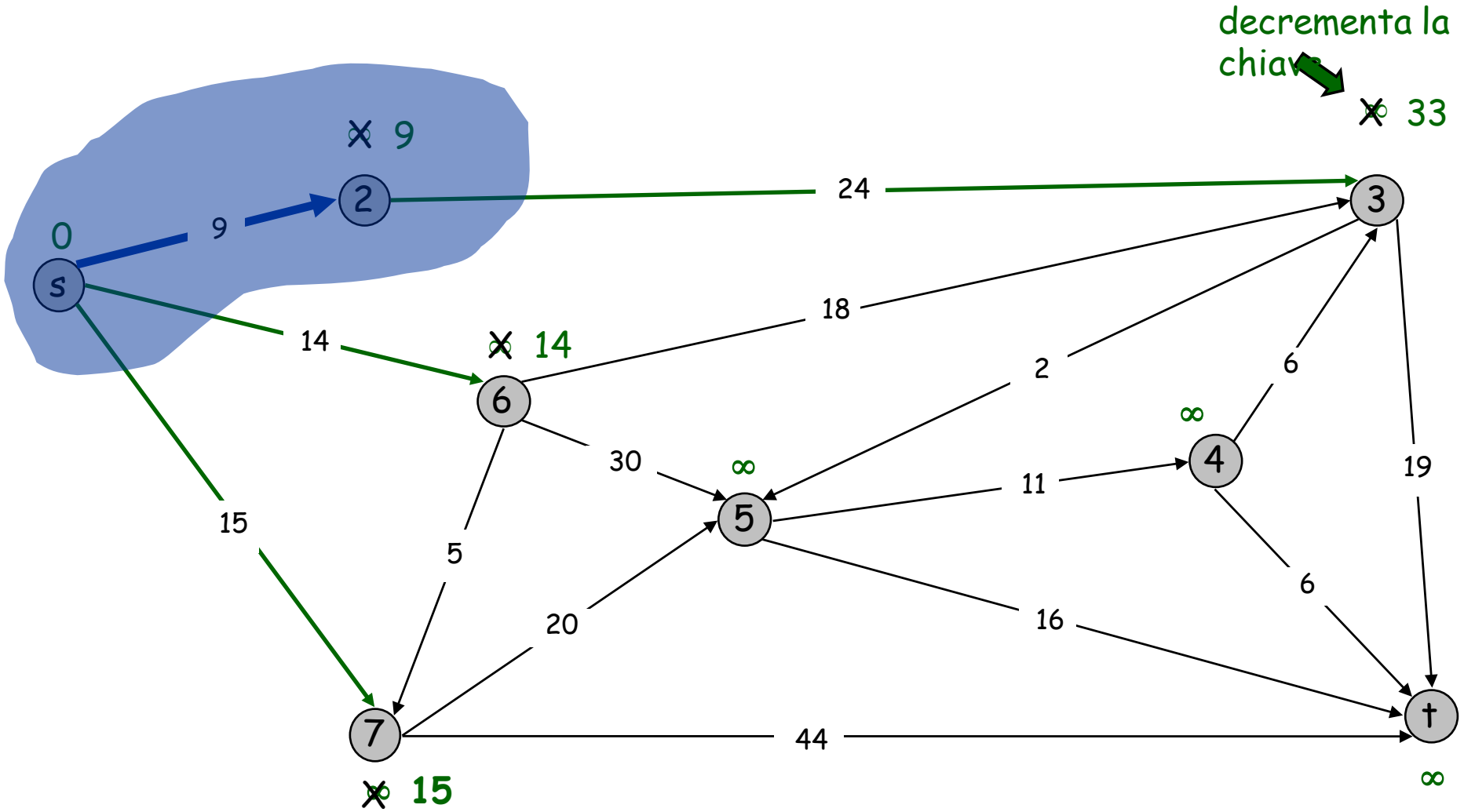
Algoritmo di Dijkstra

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



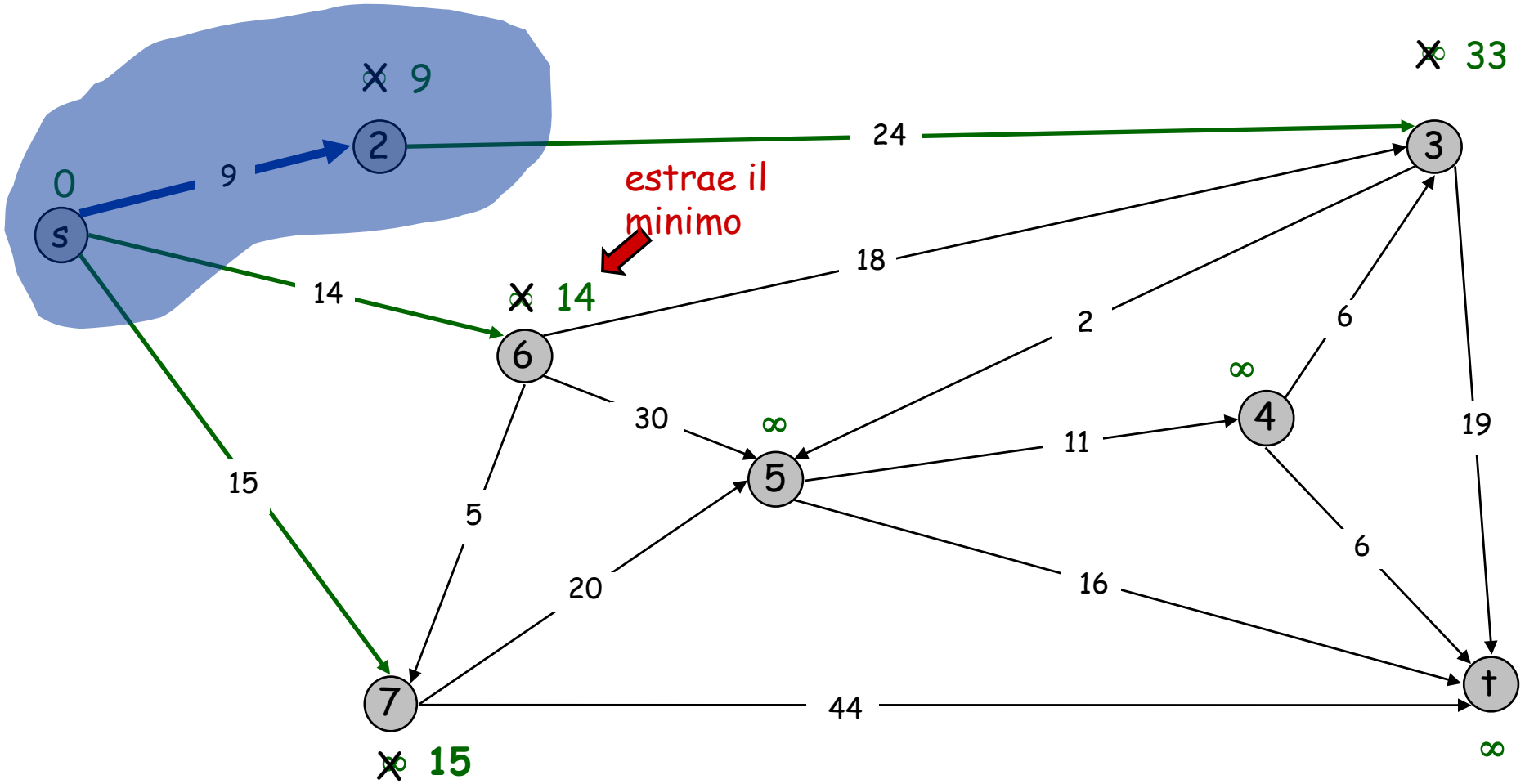
Algoritmo di Dijkstra

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



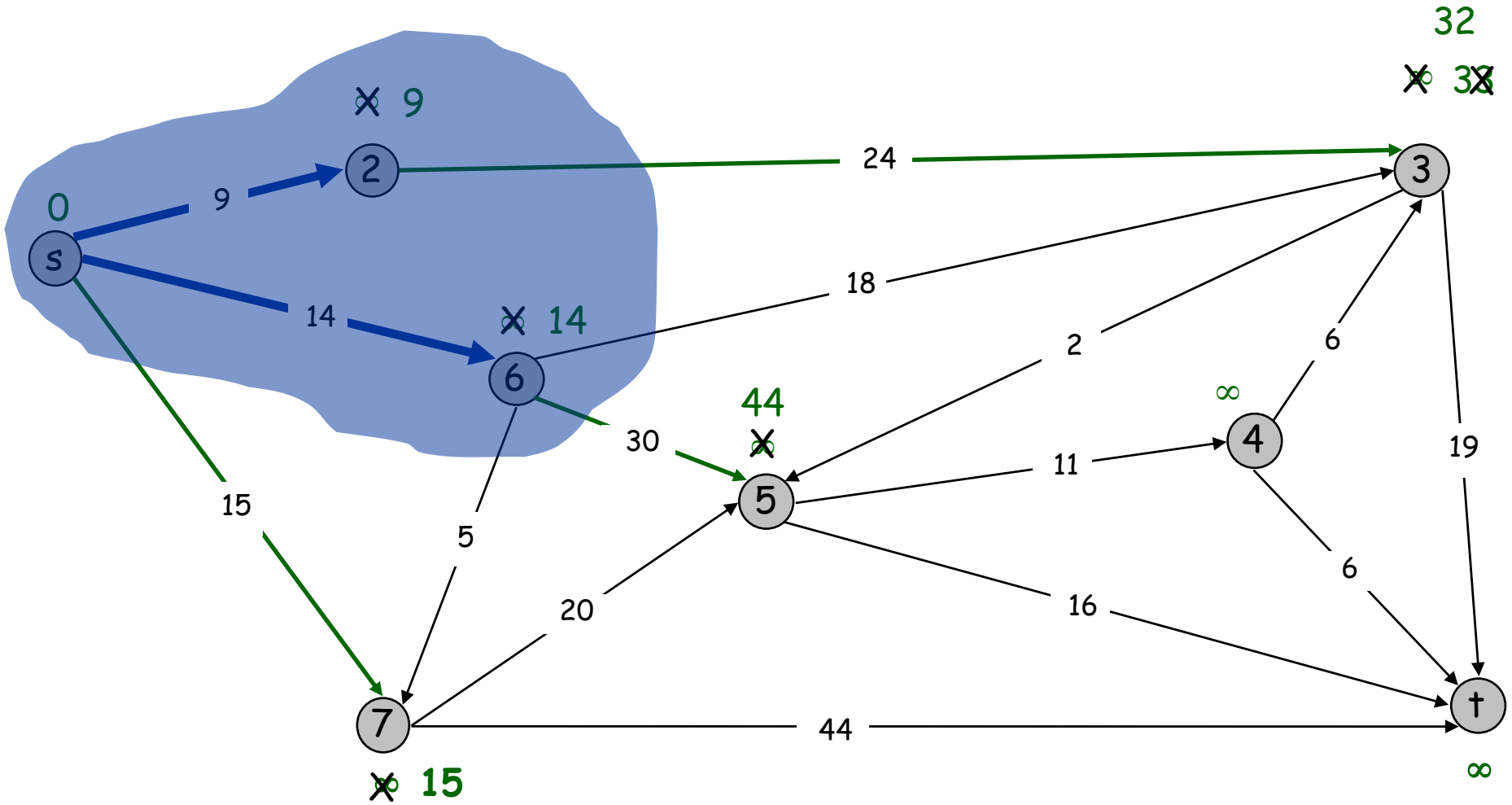
Algoritmo di Dijkstra

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



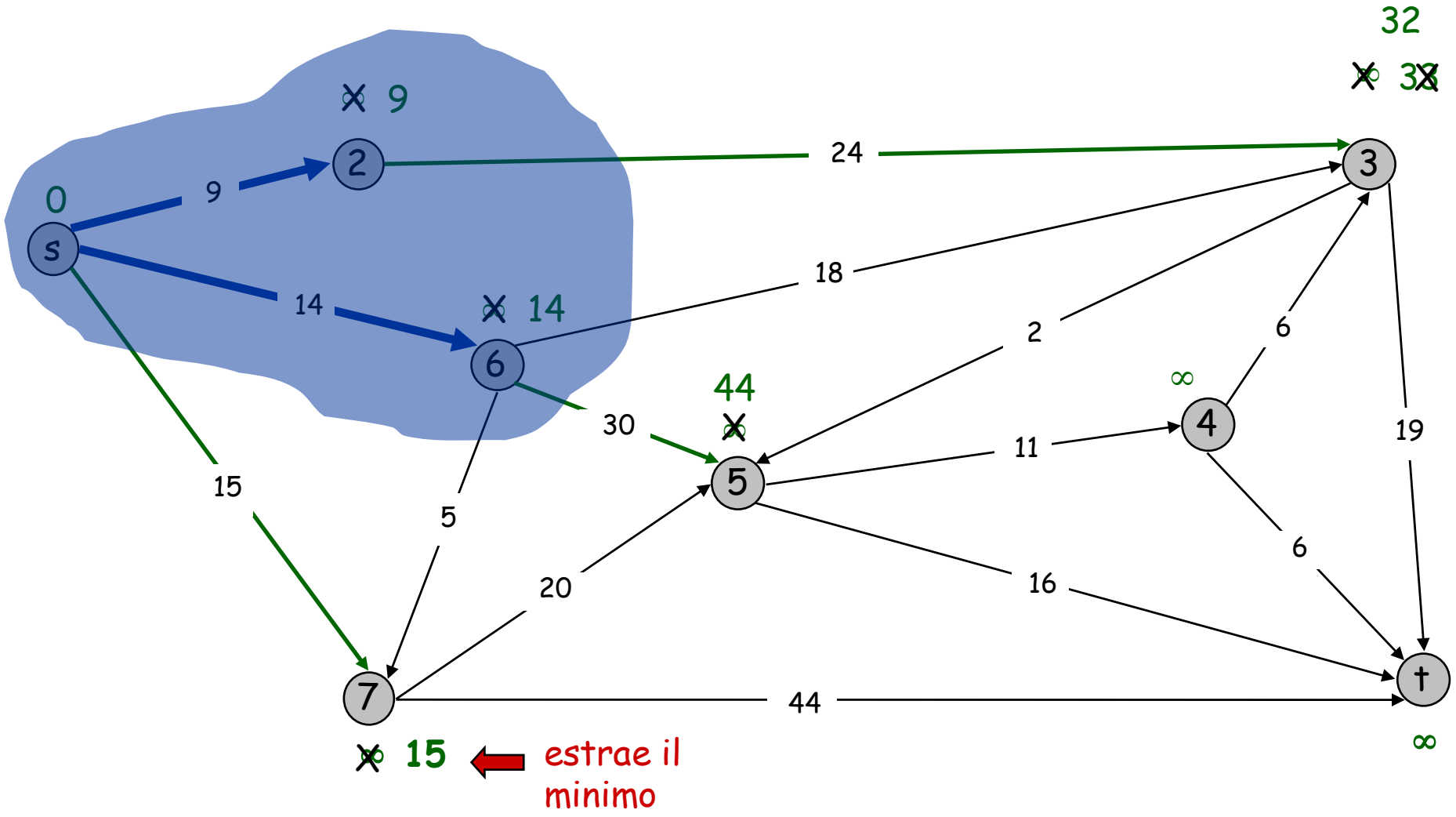
Algoritmo di Dijkstra

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



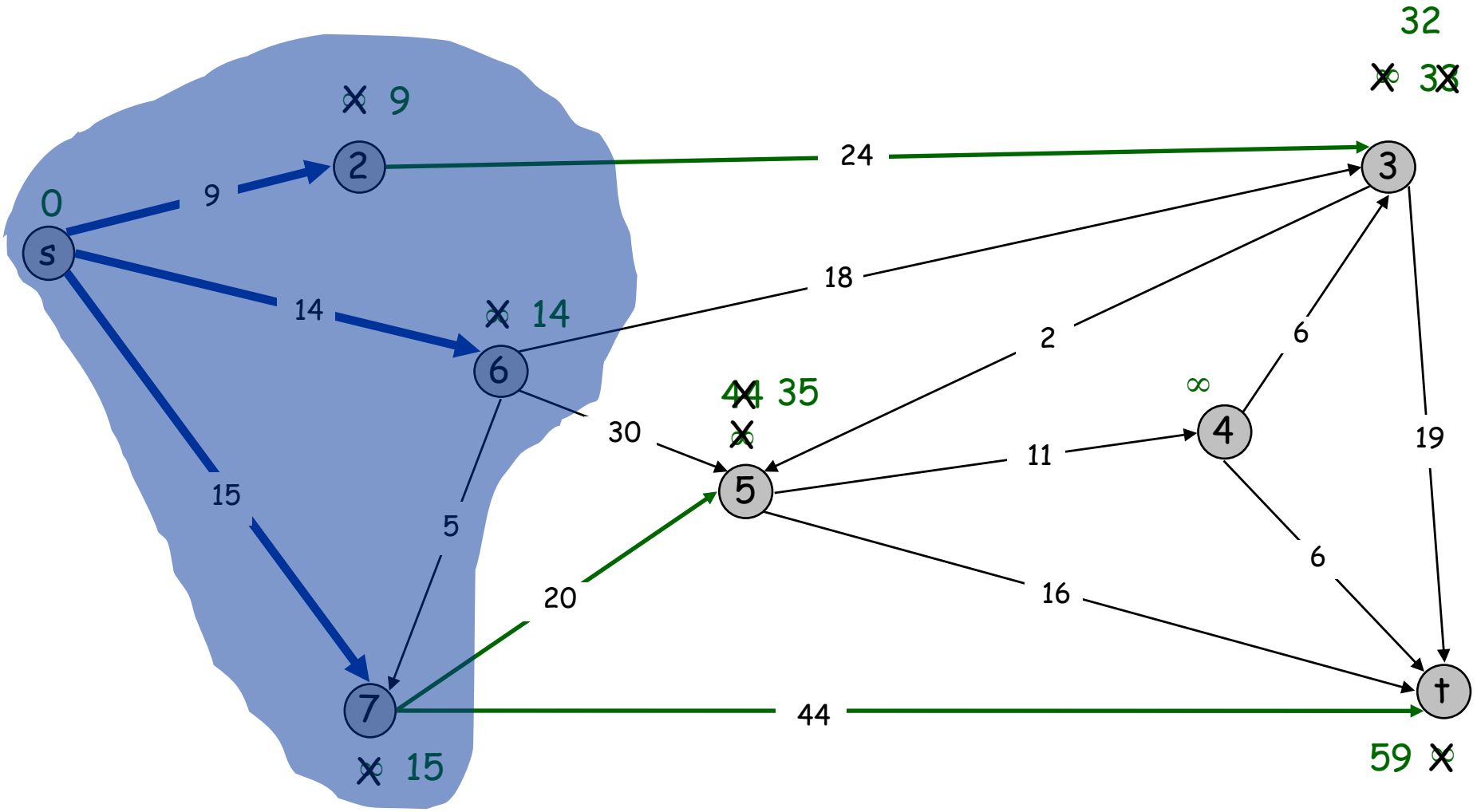
Algoritmo di Dijkstra

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



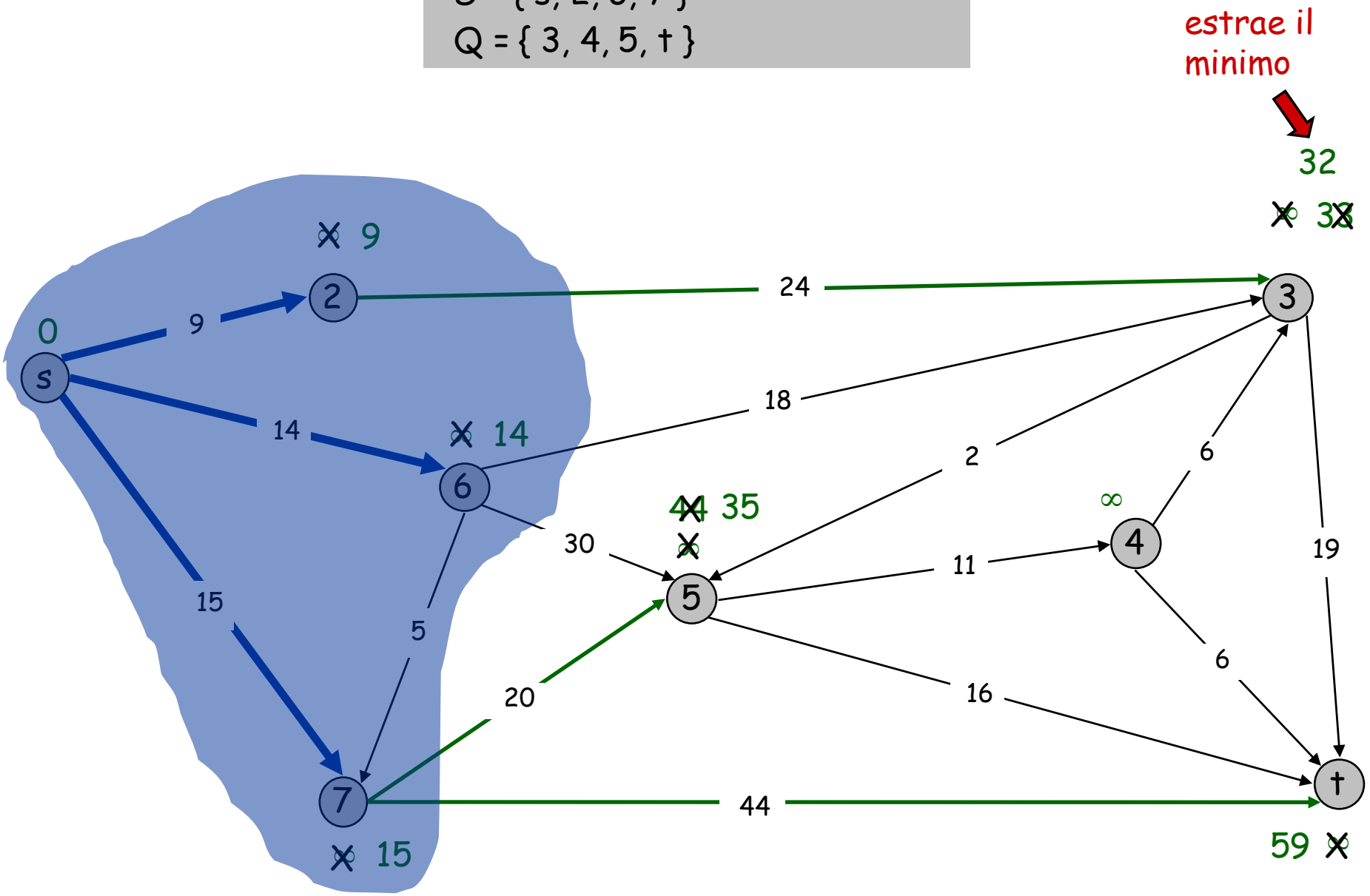
Algoritmo di Dijkstra

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



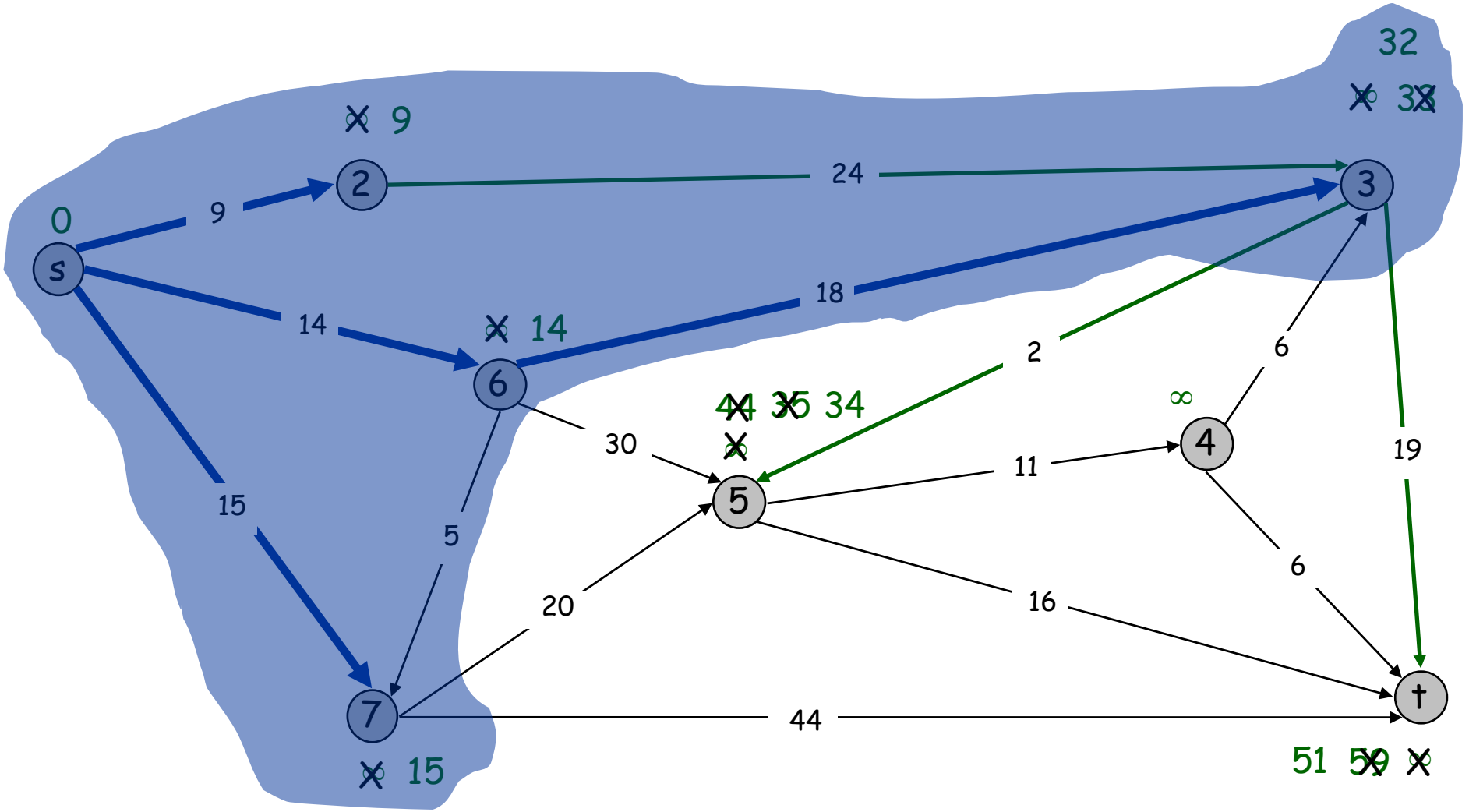
Algoritmo di Dijkstra

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



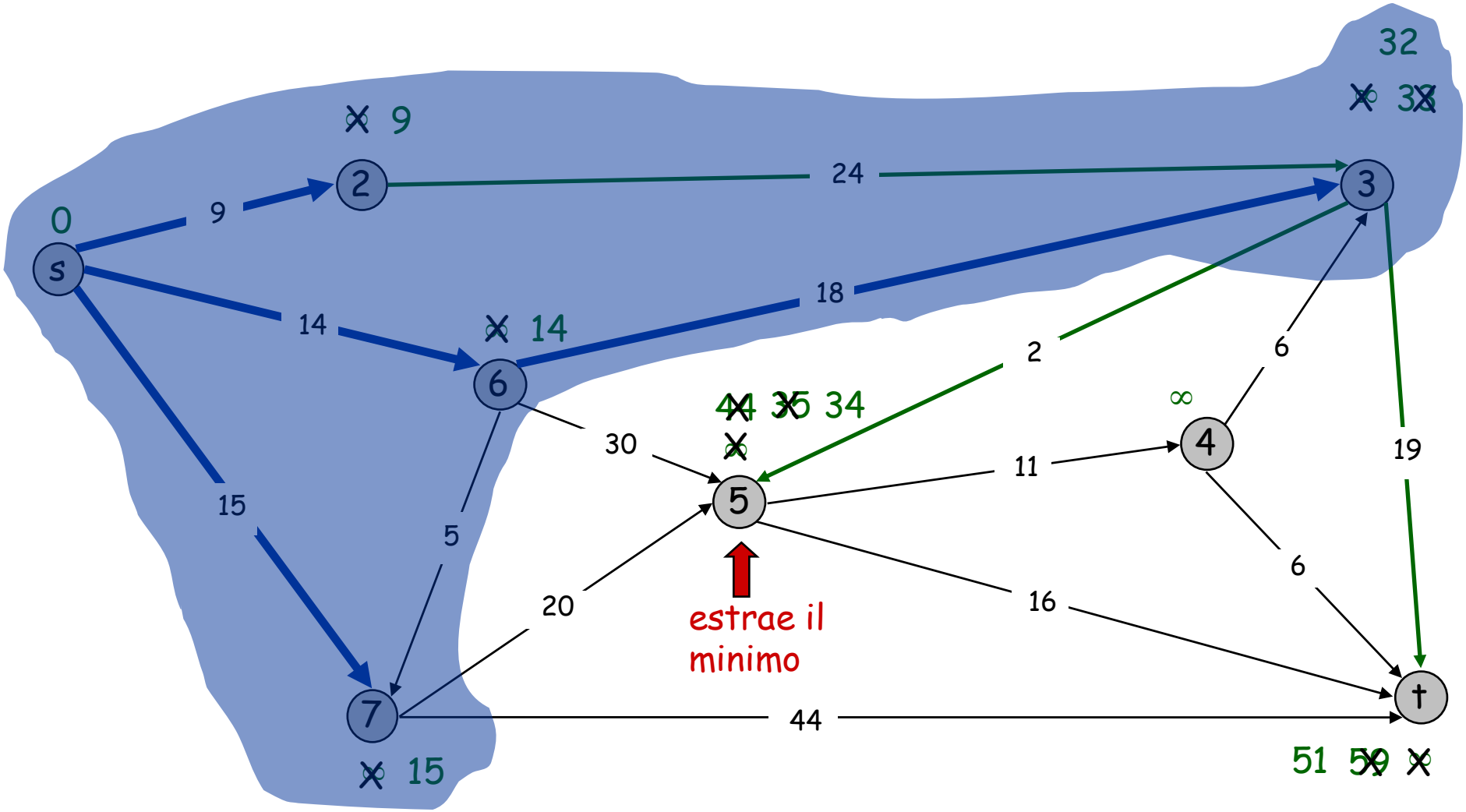
Algoritmo di Dijkstra

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



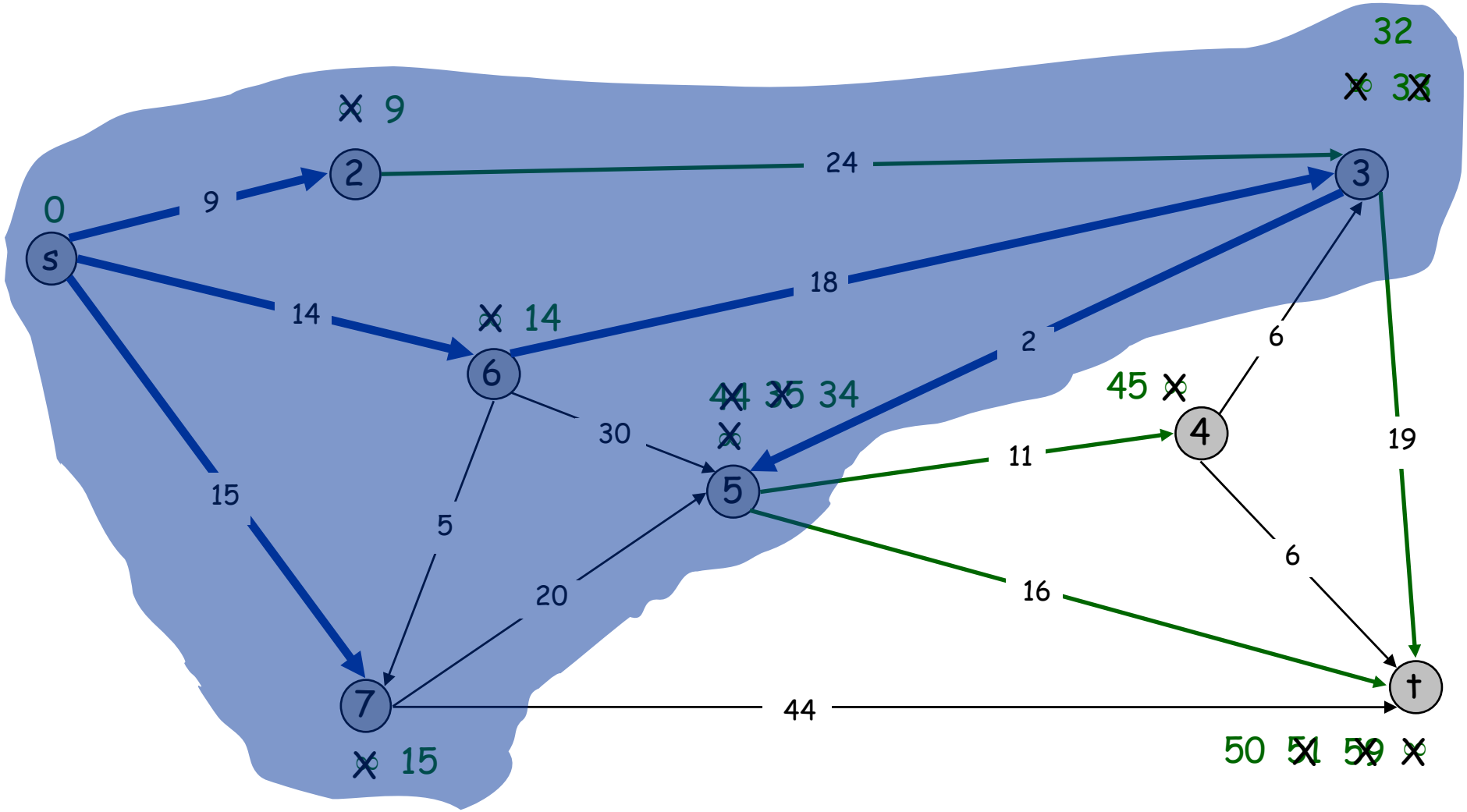
Algoritmo di Dijkstra

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



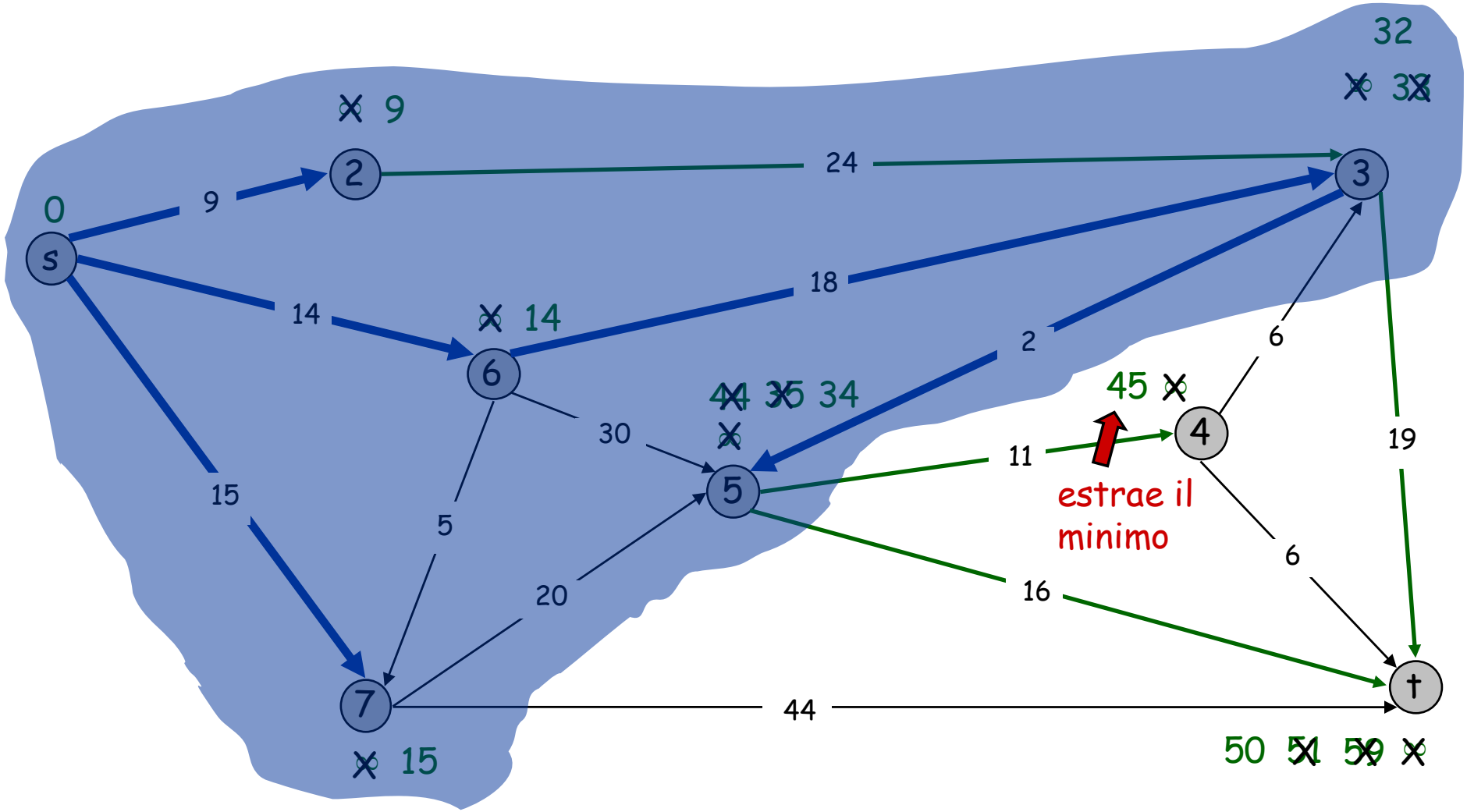
Algoritmo di Dijkstra

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



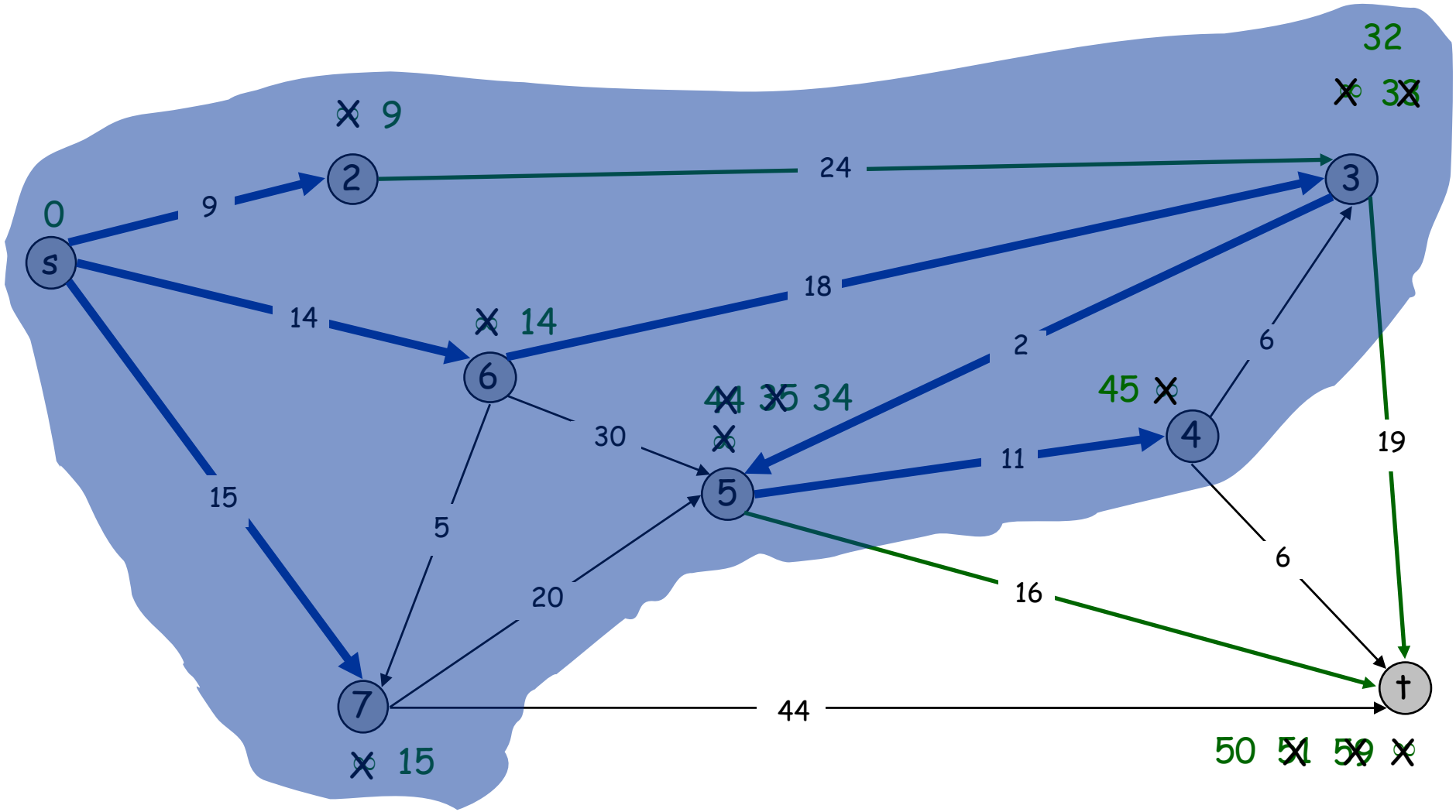
Algoritmo di Dijkstra

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



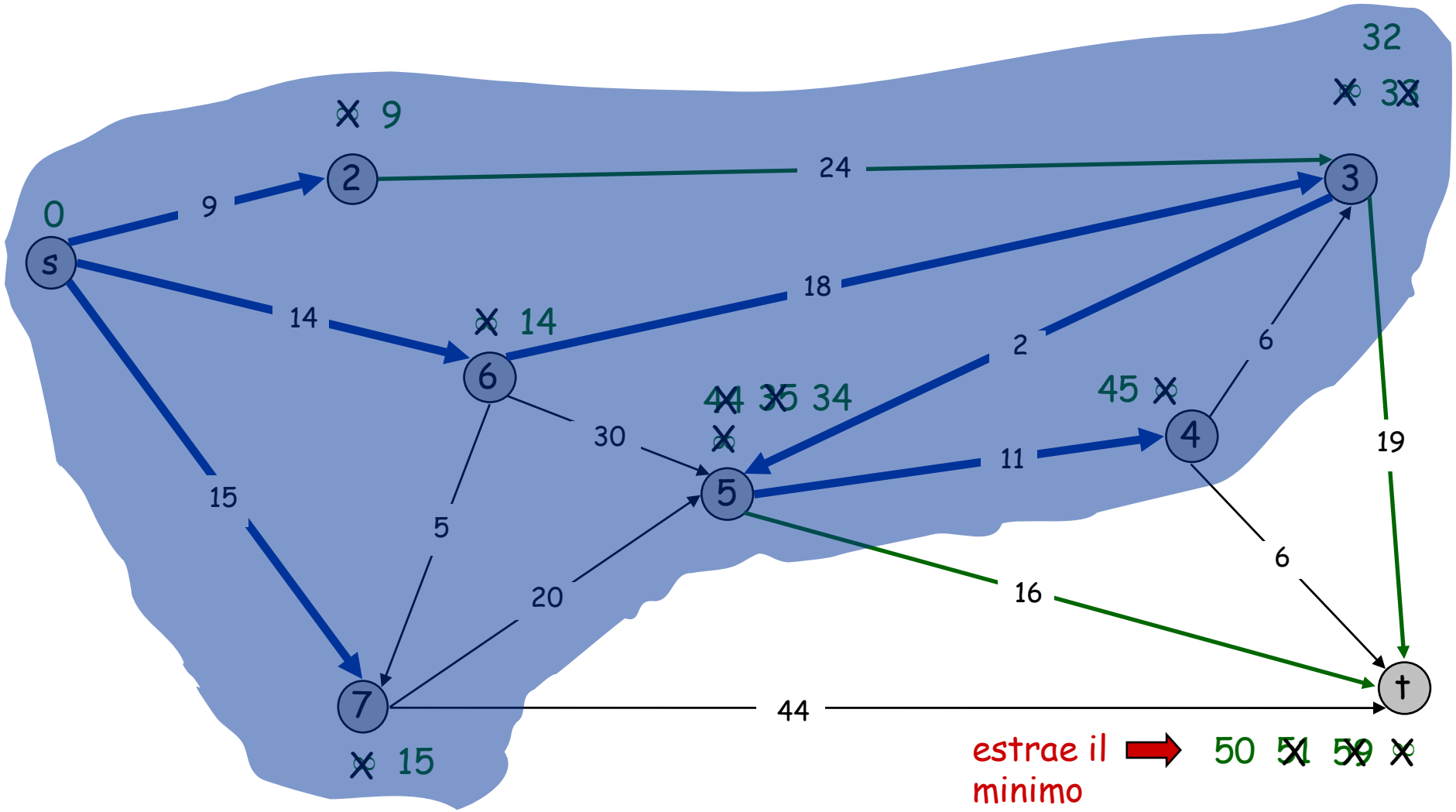
Algoritmo di Dijkstra

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



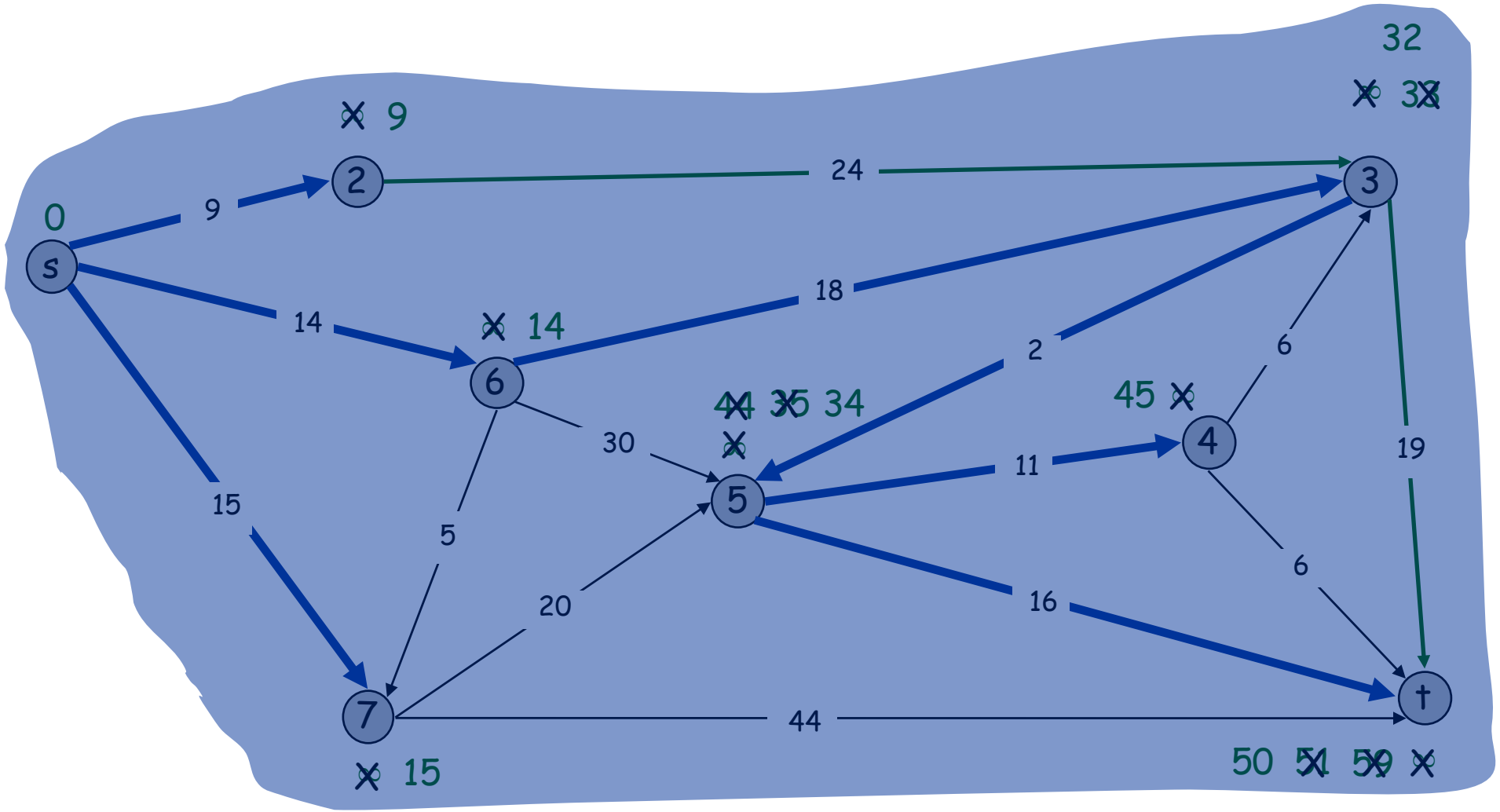
Algoritmo di Dijkstra

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



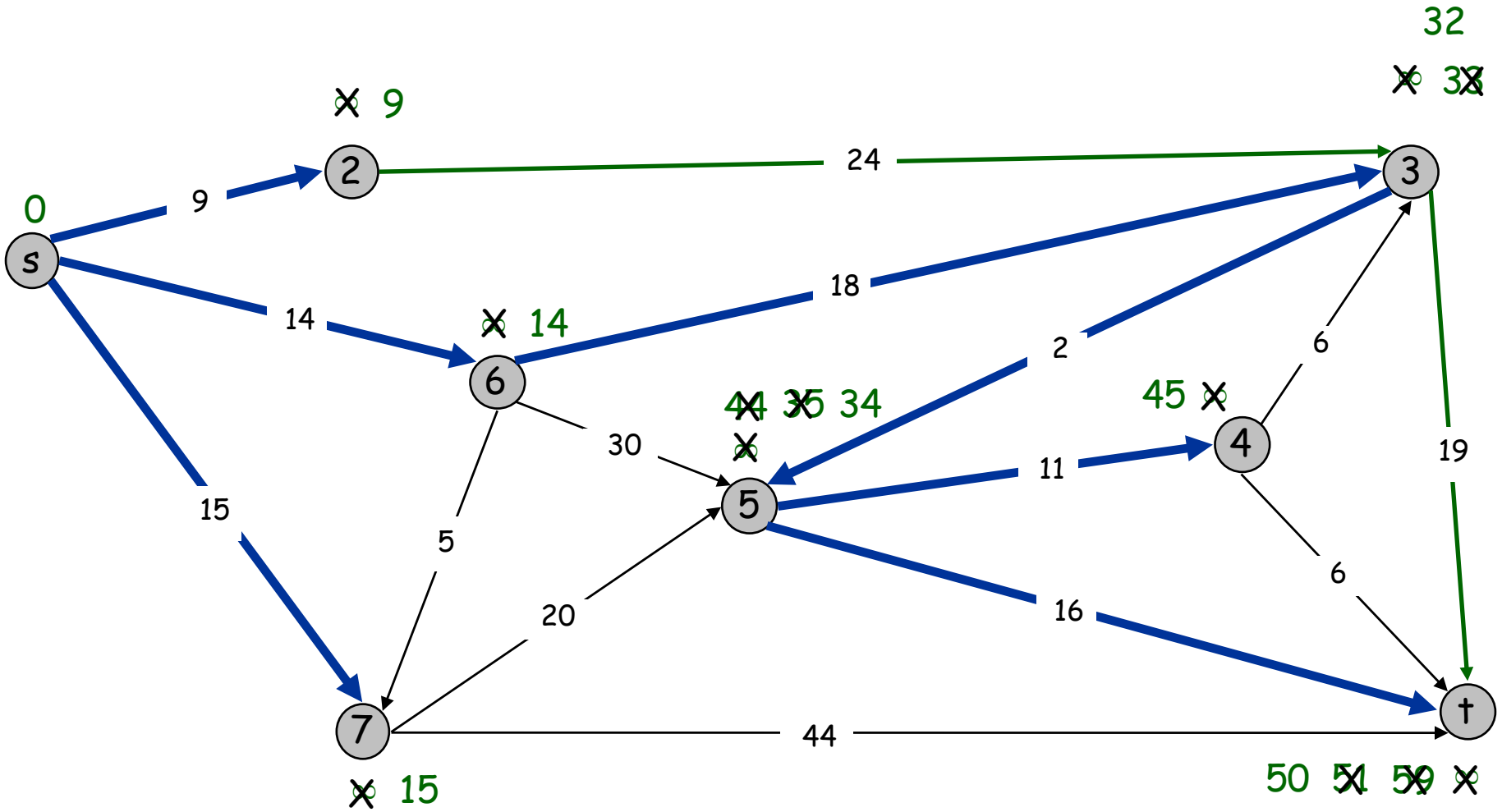
Algoritmo di Dijkstra

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



Algoritmo di Dijkstra

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



Minimo albero ricoprente (Minimum spanning tree)

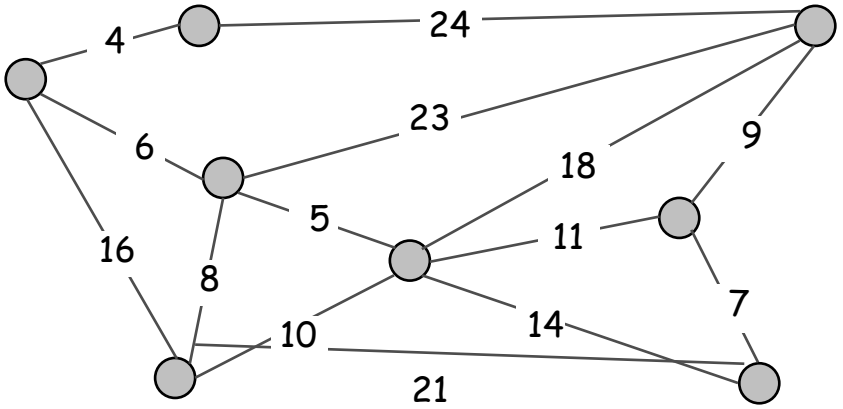
- Supponiamo di avere un insieme di pin $V = \{v_1, v_2, \ldots, v_n\}$ che decono essere interconnessi in modo che per ogni coppia di pin esista un percorso che li collega.
- Alcune coppie di pin possono essere collegate direttamente.
- Stabilire un collegamento diretto tra una coppia di pin ha un costo che dipende dalla posizione dei due pin collegati
- L'obiettivo è di utilizzare esattamente n-1 di questi collegamenti diretti tra coppie di pin in modo da connettere l'intera rete e da minimizzare la somma dei costi degli n-1 collegamenti stabiliti
- Altri esempi di applicazione alla progettazione di una rete sono.
  - Reti telefoniche, idriche, televisive, stradali, di computer

Minimo albero ricoprente (Minimum spanning tree o in breve MST)

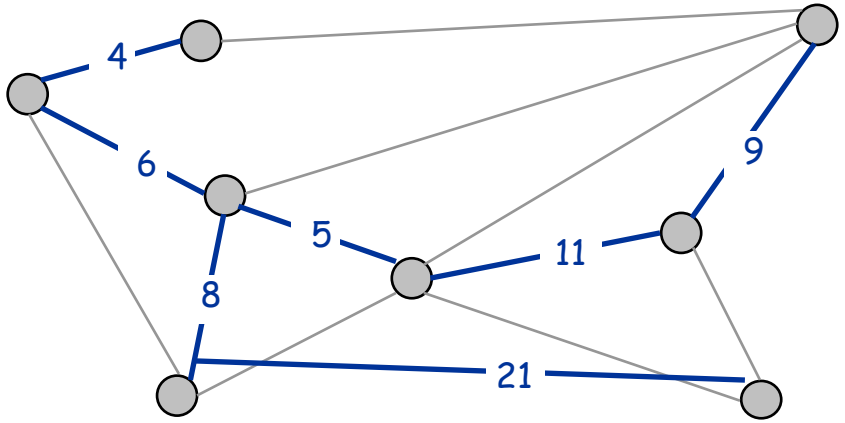
- Grafo non direzionato connesso G = (V, E).
- Per ogni arco e, c_e = costo dell'arco e (c_e numero reale).

- Def. Albero ricoprente (spanning tree). Sia dato un grafo non direzionato connesso G = (V, E). Uno spanning tree di G è un sottoinsieme di archi $T \subseteq E$ tale che |T|=n-1 e gli archi in T non formano cicli (in altre parole T forma un albero che ha come nodi tutti i nodi di G).
- Def. Sia dato un grafo non direzionato connesso G = (V, E) tale che ad ogni arco e di G è associato un costo c_e .Per ogni albero ricoprente T di G , definiamo il costo di T come $\sum_{e \in T} c_e$

.



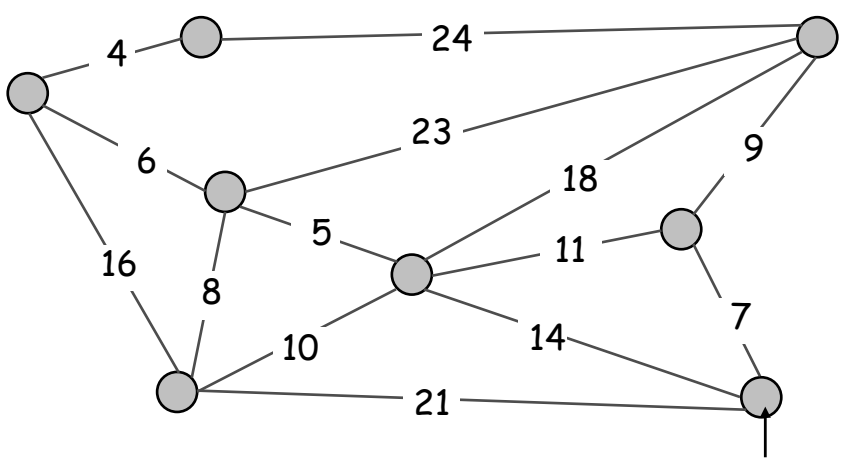
G = (V, E)



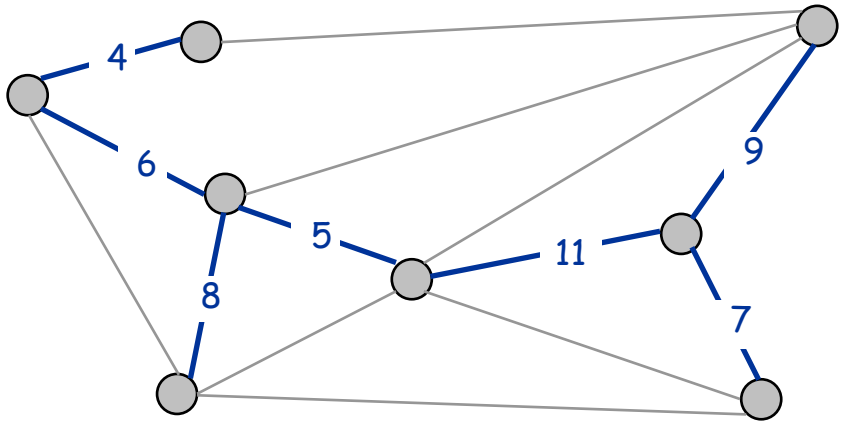
T, $\Sigma_{e \in T} c_e = 64$

Minimo albero ricoprente (Minimum spanning tree o in breve MST)

- Input:
  - Grafo non direzionato connesso $G = (V, E)$.
  - Per ogni arco e, c_e = costo dell'arco e.
- Minimo albero ricoprente. Sia dato un grafo non direzionato connesso $G = (V, E)$ con costi c_e degli archi a valori reali. Un minimo albero ricoprente è un sottoinsieme di archi $T \subseteq E$ tale che T è un albero ricoprente di costo minimo.



$G = (V, E)$



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

Minimo albero ricoprente (Minimum spanning tree o in breve MST)

- Il problema di trovare un minimo albero ricoprente non può essere risolto con un algoritmo di forza bruta
- Teorema di Cayley. Ci sono n^{n-2} alberi ricoprenti del grafo completo K_n.

Algoritmi greedy per MST

- Kruskal. Comincia con T = ϕ. Considera gli archi in ordine non decrescente di costo. Inserisce un arco e in T se e solo il suo inserimento non determina la creazione di un ciclo in T

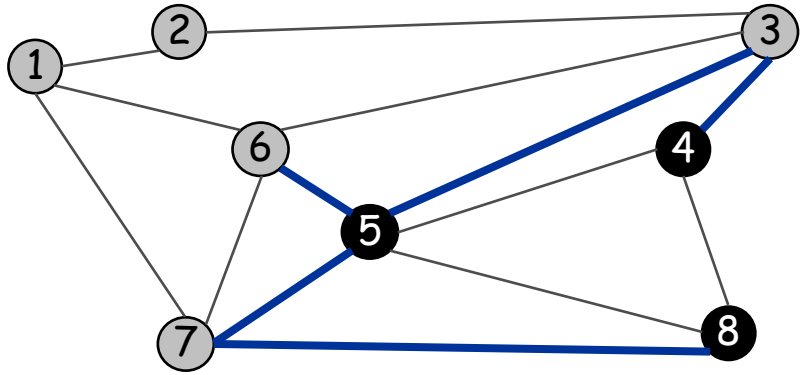
- Inverti-Cancella. Comincia con T = E. Considera gli archi in ordine non crescente dei costi. Cancella e da T se e solo se la sua cancellazione non rende T disconnesso.

- Prim. Comincia con un certo nodo s e costruisce un albero T avente s come radice. Ad ogni passo aggiunge a T l'arco di peso più basso tra quelli che hanno esattamente una delle due estremità in T (se un arco avesse entrambe le estremità in T, la sua introduzione in T creerebbe un ciclo)

Taglio

- Taglio. Un taglio è una partizione [S,V-S] dell'insieme dei vertici del grafo.

- Insieme di archi che attraversano il taglio [S,V-S].
  Sottoinsieme D di archi che hanno un'estremità in S e una in V-S.



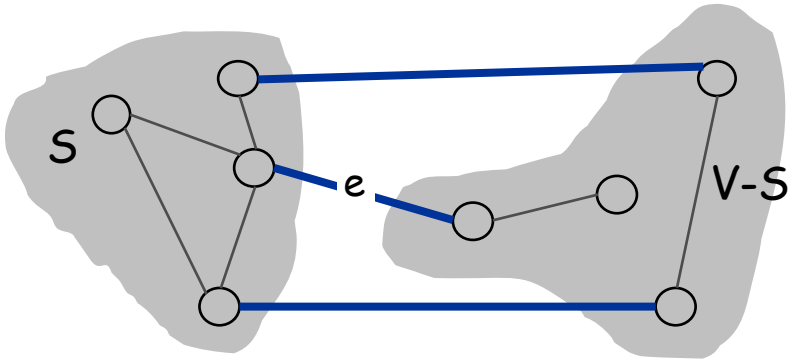
Taglio [S,V-S] = ({ 4, 5, 8 }, { 1,2,3,6,7 })
Archi che attraversano [S,V-S] D = 5-6, 5-7, 3-4, 3-5, 7-8

Algoritmi Greedy

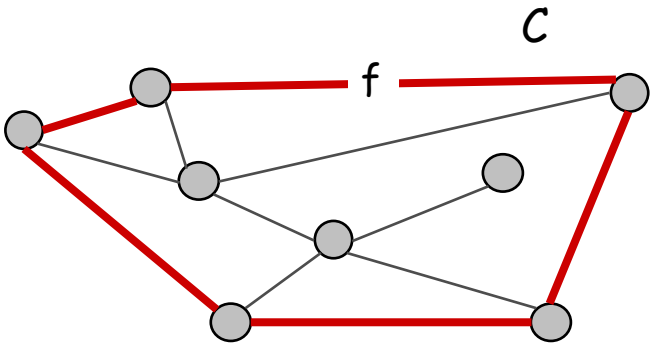
- **Per il momento** assumiamo per semplicità che i pesi degli archi siano a due a due distinti
  - Vedremo che questa assunzione implica che il minimo albero ricoprente è unico.

- Proprietà del taglio. Sia S un qualsiasi sottoinsieme di nodi e sia e l'arco di costo minimo che attraversa il taglio [S,V-S]. Ogni minimo albero ricoprente contiene e.

- Proprietà del ciclo. Sia C un qualsiasi ciclo e sia f l'arco di costo massimo in C. Nessun minimo albero ricoprente contiene f.



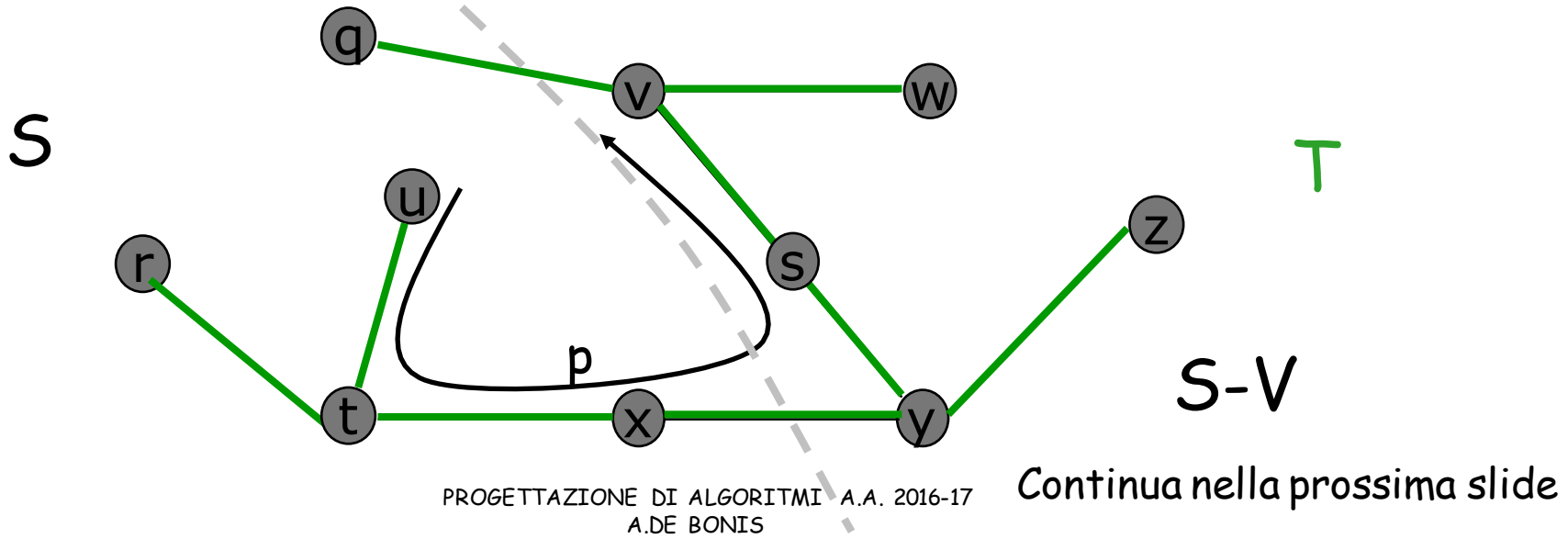
e è nello MST



f non è nello MST

Proprietà del taglio

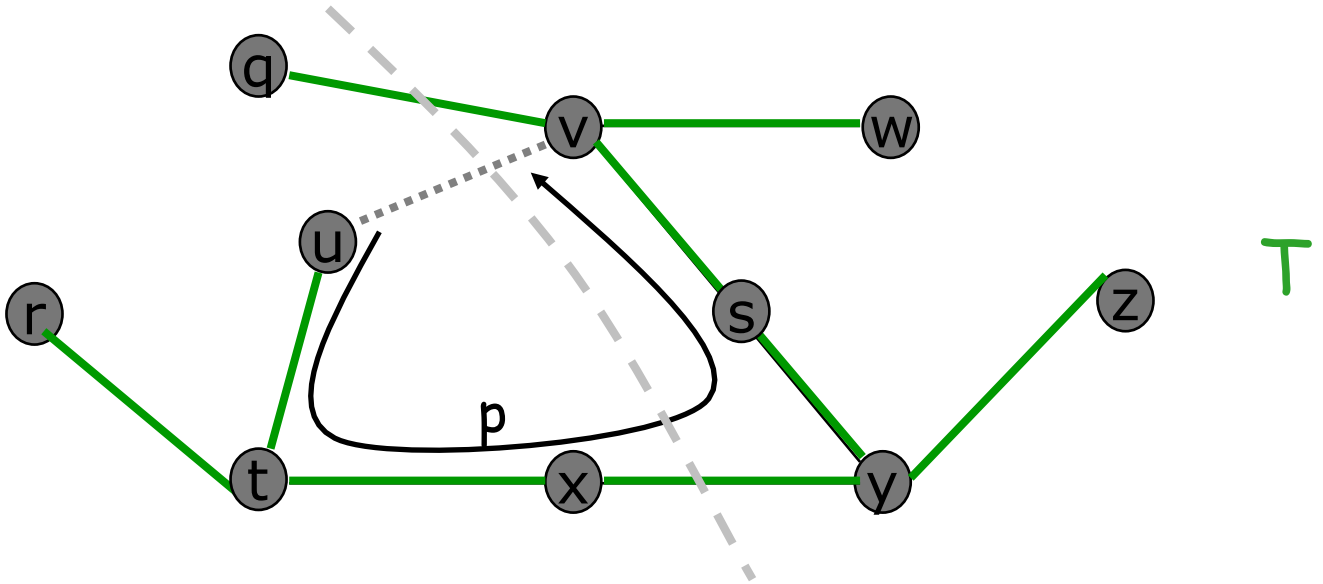
- Stiamo assumendo (per il momento) che i costi degli archi siano a due a due distinti
- Proprietà del taglio. Sia S un qualsiasi sottoinsieme di nodi e sia e=(u,v) l'arco di costo minimo che attraversa il taglio [S,V-S]. Ogni minimo albero ricoprente contiene e.
- Dim. (nel caso in cui i costi degli archi sono a due a due distinti)
- Sia T un albero ricoprente tale che e=(u,v) $\notin$T. Dimostriamo che T non può essere un minimo albero ricoprente.
- Sia p il percorso da u a v in T. In T non ci sono altri percorsi da u a v altrimenti ci sarebbe un ciclo



Continua nella prossima slide

Proprietà del taglio

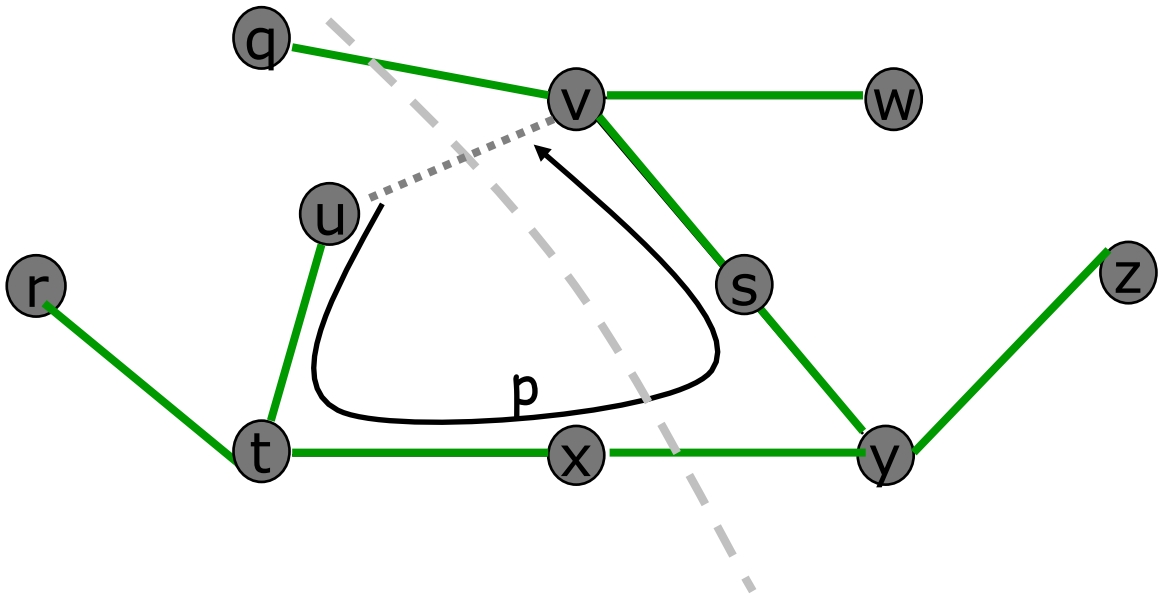
- Osserviamo che se aggiungiamo e=(u,v) a T generiamo un ciclo
- Siccome u e v sono ai lati opposti del taglio [S,V-S] allora il ciclo deve comprendere un arco f=(x,y) di T che attraversa il taglio [S,V-S]



Continua nella prossima slide

Proprietà del taglio

- Poichè (x,y) ≠ (u,v) e poichè entrambi attraversano il taglio e (u,v) è l'arco di peso minimo tra quelli che attraversano il taglio allora si ha $c_e < c_f$

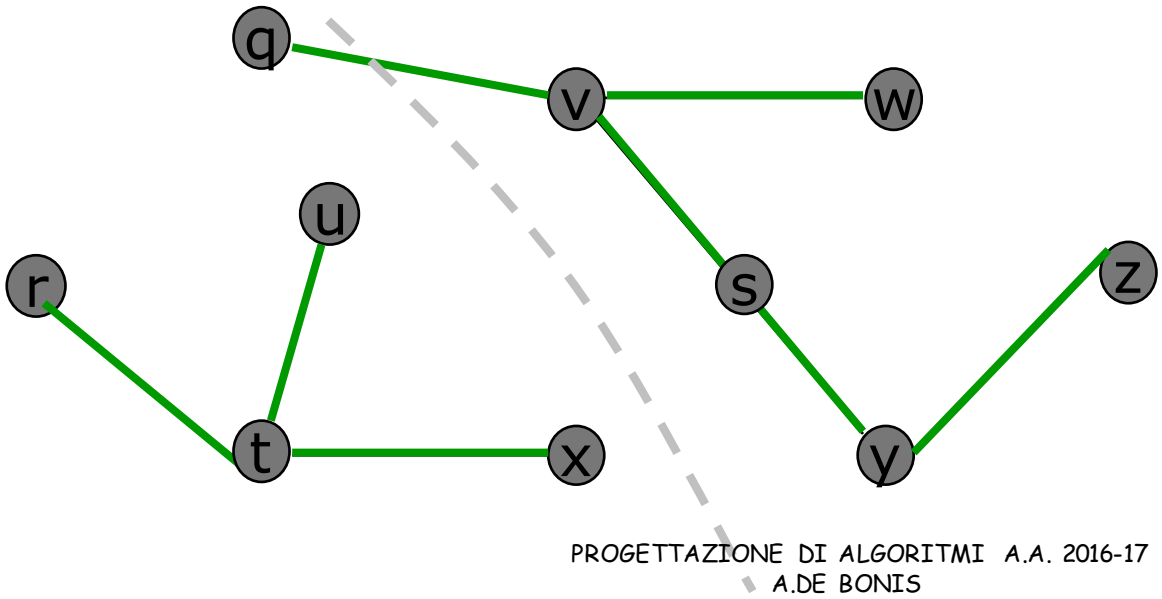


Proprietà del taglio

- Poichè (x,y) ≠ (u,v) e poichè entrambi attraversano il taglio e (u,v) è l'arco di peso minimo tra quelli che attraversano il taglio allora si ha $c_e < c_f$

- f=(x,y) si trova sull'unico percorso che connette u a v in T

- Se togliamo f=(x,y) da T dividiamo T in due alberi, uno contente u e l'altro contenente v



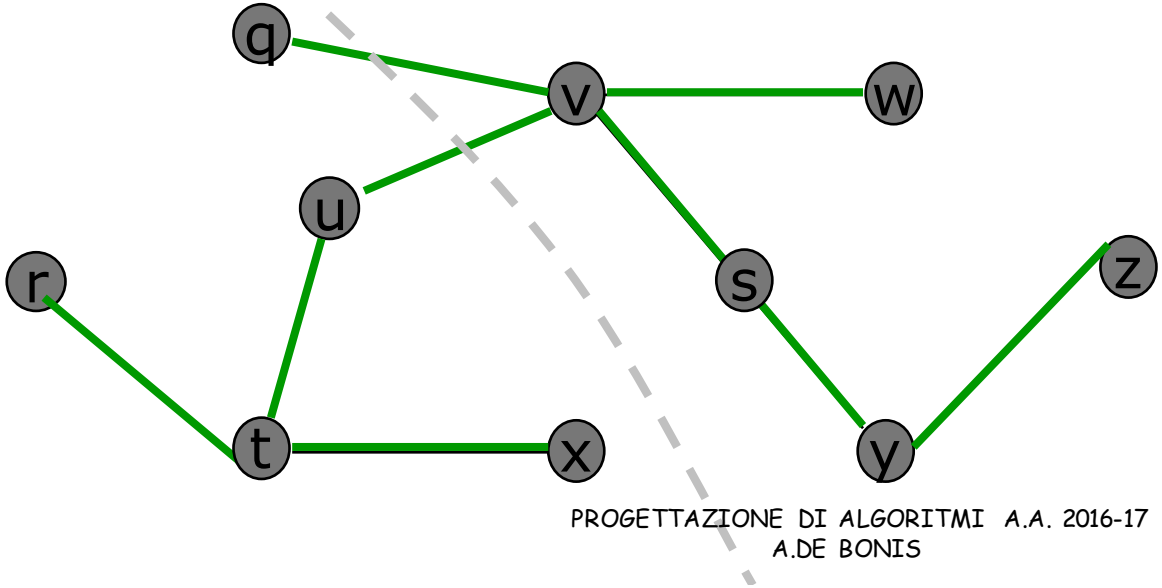
Proprietà del taglio

- Poichè (x,y) ≠ (u,v) e poichè entrambi attraversano il taglio e (u,v) è l'arco di peso minimo tra quelli che attraversano il taglio allora si ha $c_e < c_f$

- f=(x,y) si trova sull'unico percorso che connette u a v in T

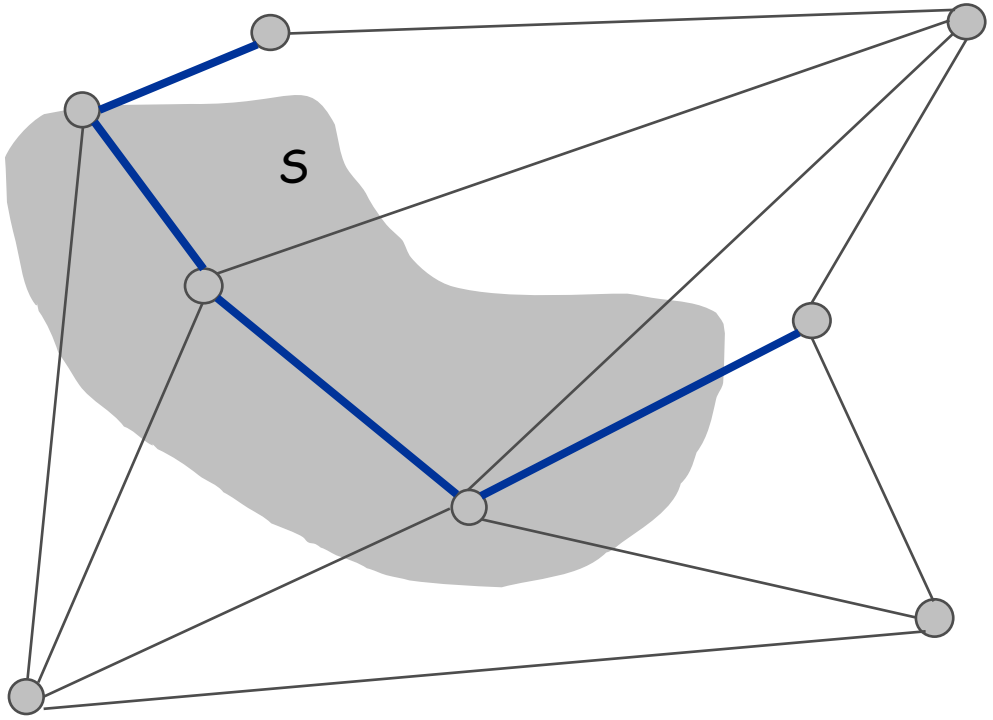
- Se togliamo f=(x,y) da T dividiamo T in due alberi, uno contente u e l'altro contenente v

- Se introduciamo e=(u,v) riconnettiamo i due alberi ottenendo un nuovo albero T' = T-{f}∪{e} di costo c(T')=c(T)-c_f+c_e<c(T). Ciò vuol dire che T non è un minimo albero ricoprente.



Algoritmo di Prim

- Algoritmo di Prim. [Jarník 1930, Prim 1957, Dijkstra 1959,]
- Ad ogni passo T è un sottoinsieme di archi dello MST
- S= insieme di nodi di T
  - Inizializzazione: Pone in S un qualsiasi nodo u. Il nodo u sarà la radice dello MST
  - Ad ogni passo aggiunge a T l'arco (x,y) di costo minimo tra tutti quelli che congiungono un nodo x in S ad un nodo y in V-S (scelta greedy)
  - Termina quando S=V

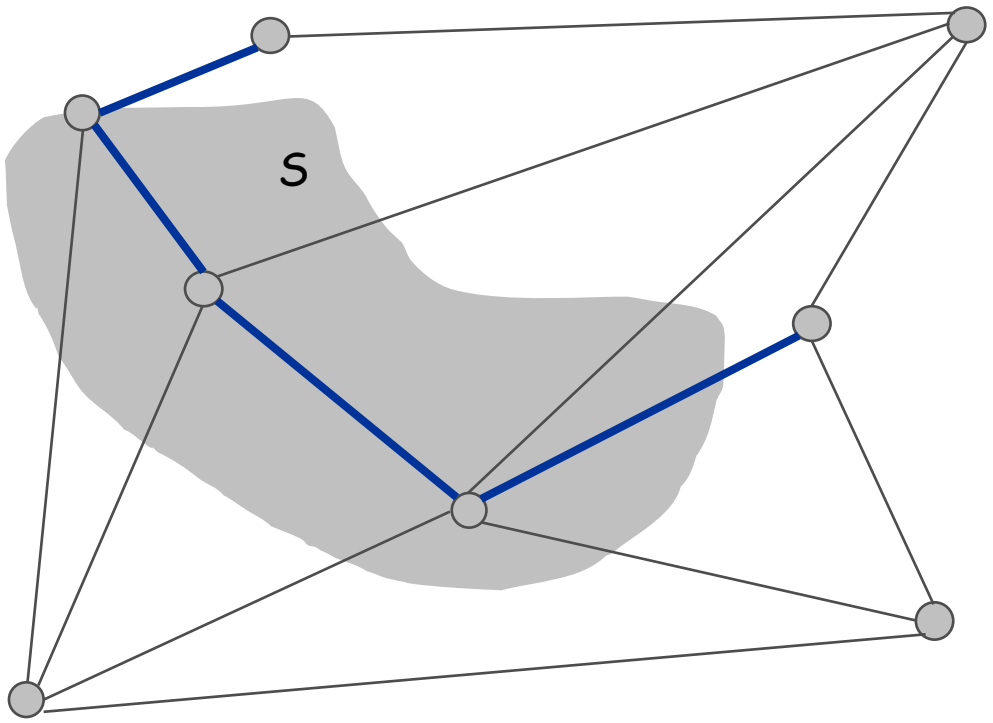


Correttezza dell'algoritmo di Prim

- L'algoritmo di Prim ottiene il minimo albero ricoprente in quanto ad ogni passo seleziona l'arco di costo minimo tra quelli che attraversano il taglio [S,V-S].
- La proprietà del taglio implica che ogni albero ricoprente deve contenere ciascuno degli archi selezionati dall'algoritmo → T sottoinsieme di MST
- Ci resta da dimostrare che T è un albero ricoprente. Ovviamante T non contiene cicli in quanto un arco viene inserito solo se una delle sue estremità non è ancora attaccata all'albero. Poiché inoltre l'algoritmo si ferma solo quando S=V cioè quando ha attaccato tutti i vertici all'albero, si ha che T è un albero ricoprente.

In conclusione T è un albero ricoprente che contiene esclusivamente archi che fanno parte dello MST e quindi T è lo MST.

NB: quando i costi sono a due a due distinti c'è un unico MST in quanto per ogni taglio c'è un unico arco di costo minimo che lo attraversa



Implementazione dell'algoritmo di Prim con coda a priorità

- Mantiene un insieme di vertici esplorati S.
- Per ogni nodo non esplorato v, mantiene a[v] = costo dell'arco di costo più basso tra quelli che uniscono v ad un nodo in S
- Mantiene coda a priorità Q delle coppie (a[v],v)
- Stessa analisi dell'algoritmo di Prim con coda a priorità:
- $O(n^2)$ con array o lista non ordinati;
- $O(m \log n + n \log n)$ con heap. Siccome nel problema dello MST il grafo è connesso allora $m \geq n-1$ e $O(m \log n + n \log n) = O(m \log n)$

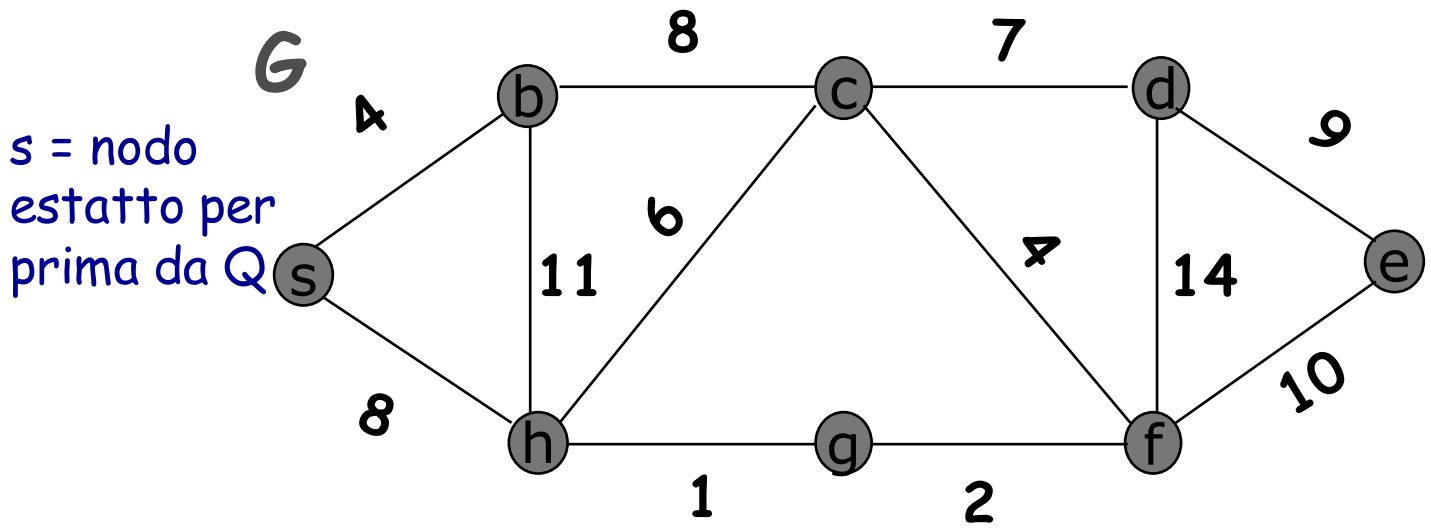
```
Prim's Algorithm (G,c)
Let S be the set of explored nodes
For each u not in S, we store the cost a[u]
Let Q be a priority queue of pairs (a[u],u) s.t. u is not in S

For each u in V insert (Q, ∞,u)  in Q EndFor
While (Q is not empty)
 (a[u],u)<-- ExtractMin(Q)
  Add u to S
  For each edge e=(u,v)
   If  ((v not in S) && (c_e< a[v]))
     ChangeKey(Q,v, c_e)

EndWhile
```

Un esempio

In questo esempio, per ciascun nodo u manteniamo anche un campo $\pi[u]$ che alla fine è uguale al padre di u nello MST

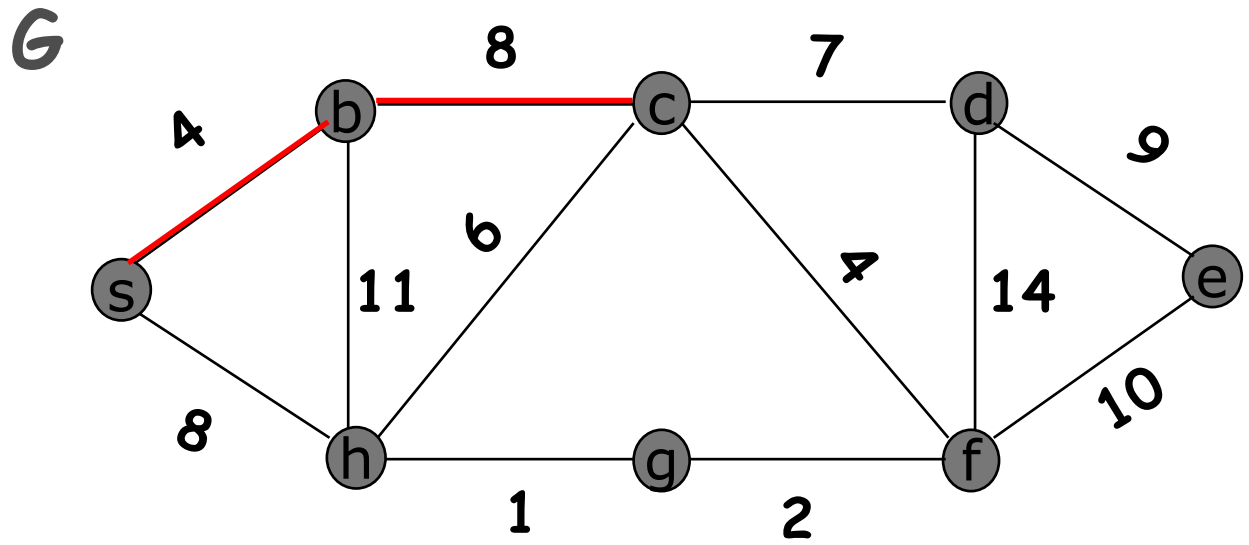


$Q = \{[s, \infty],[b, \infty],[c, \infty],[d, \infty],[e, \infty],[f, \infty],[g, \infty],[h, \infty]\}$

Si estrae **s** da Q e si aggiornano i campi a e π dei nodi adiacenti ad **s**

$Q = \{[b, 4],[c, \infty],[d, \infty],[e, \infty],[f, \infty],[g, \infty],[h, 8]\}$

Un esempio



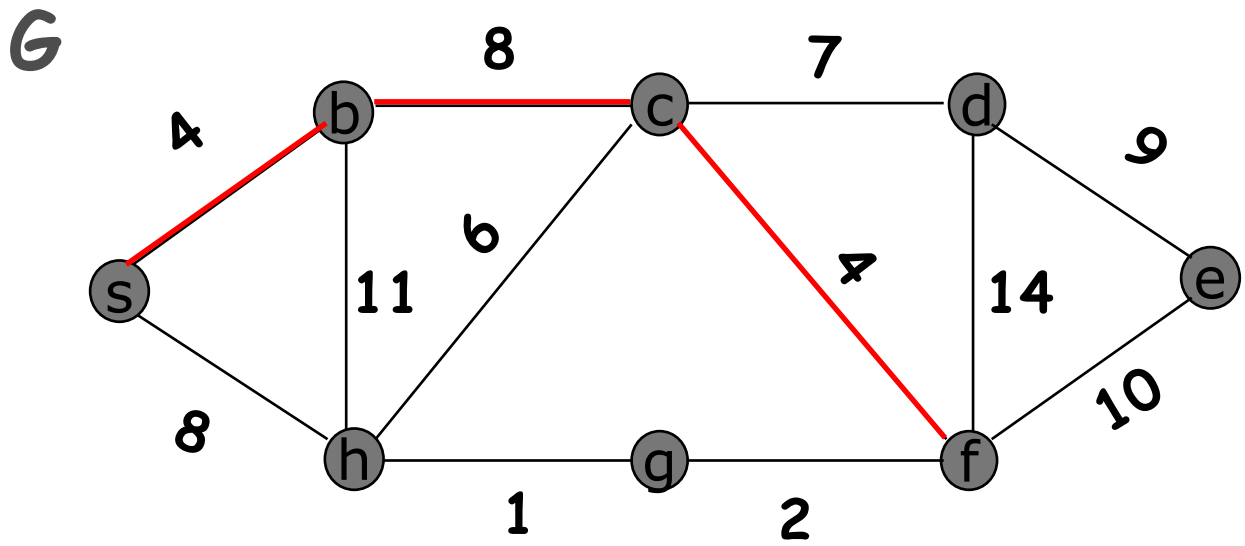
$Q = \{[c, 8],[d, \infty],[e, \infty],[f, \infty],[g, \infty],[h, 8]\}$

- A seconda dell'implementazione di Q si estrae c oppure h da Q.

- Assumiamo che venga estratto c: si aggiornano i campi a e π dei nodi adiacenti a c che si trovano in Q

$Q = \{[d, 7],[e, \infty],[f, 4],[g, \infty],[h, 6]\}$

Un esempio

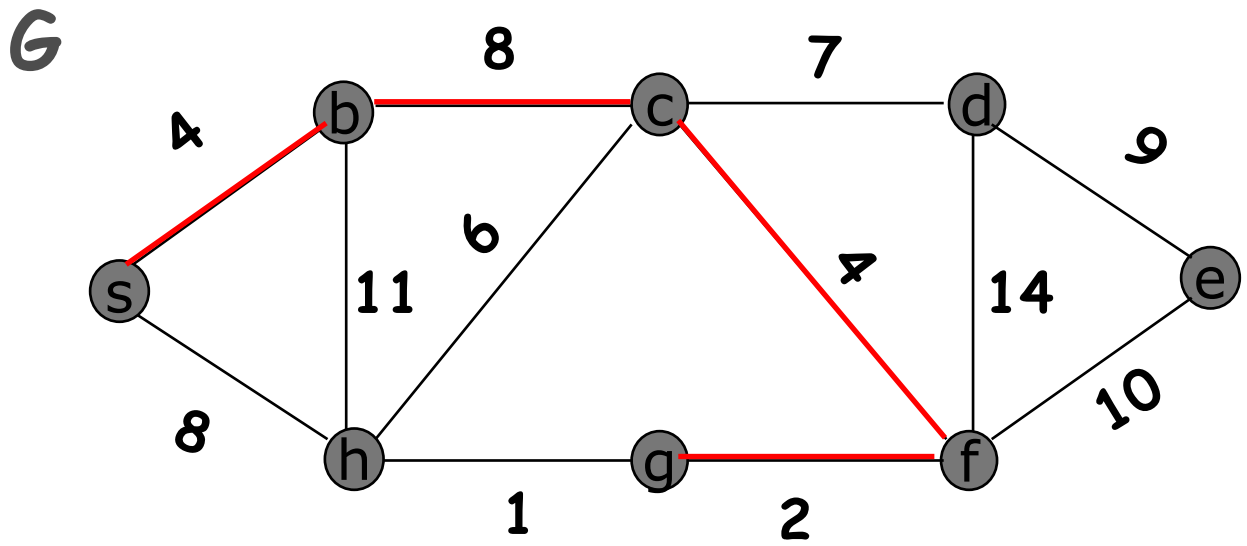


$Q = \{[d, 7],[e, \infty],[f, 4],[g, \infty],[h, 6]\}$

Si estrae f da Q e si aggiornano i campi a e π dei nodi adiacenti a f che si trovano in Q

$Q = \{[d, 7],[e, 10],[g, 2],[h, 6]\}$

Un esempio

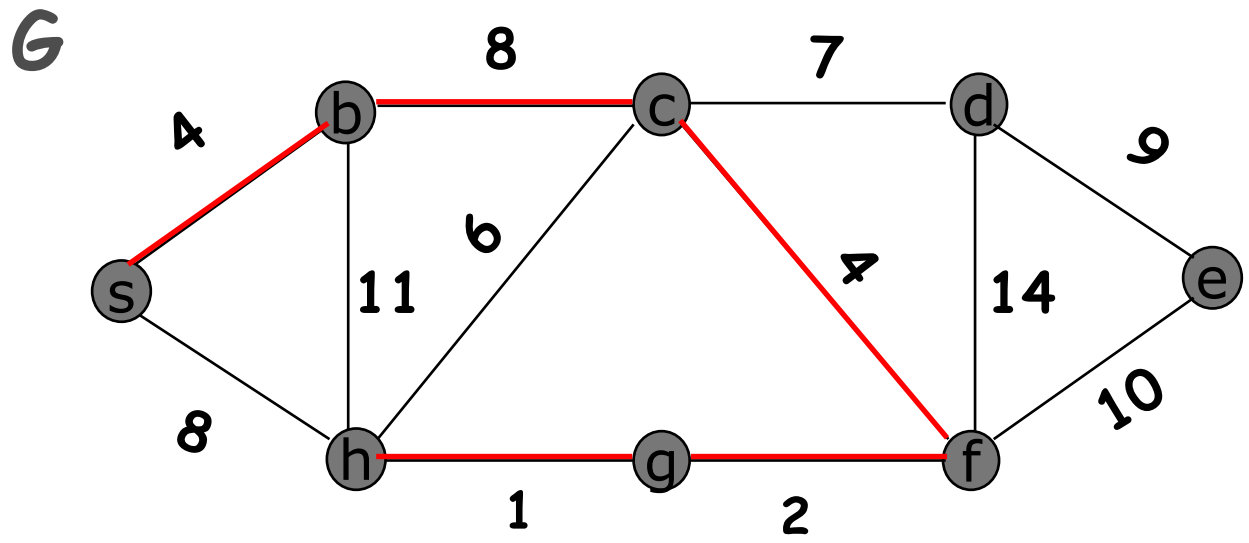


Q = {[d, 7],[e, 10],[g, 2],[h, 6]}

Si estrae g da Q e si aggiornano i campi a e π dei nodi adiacenti a g che si trovano in Q

Q = {[d, 7],[e,10],[h, 1]}

Un esempio

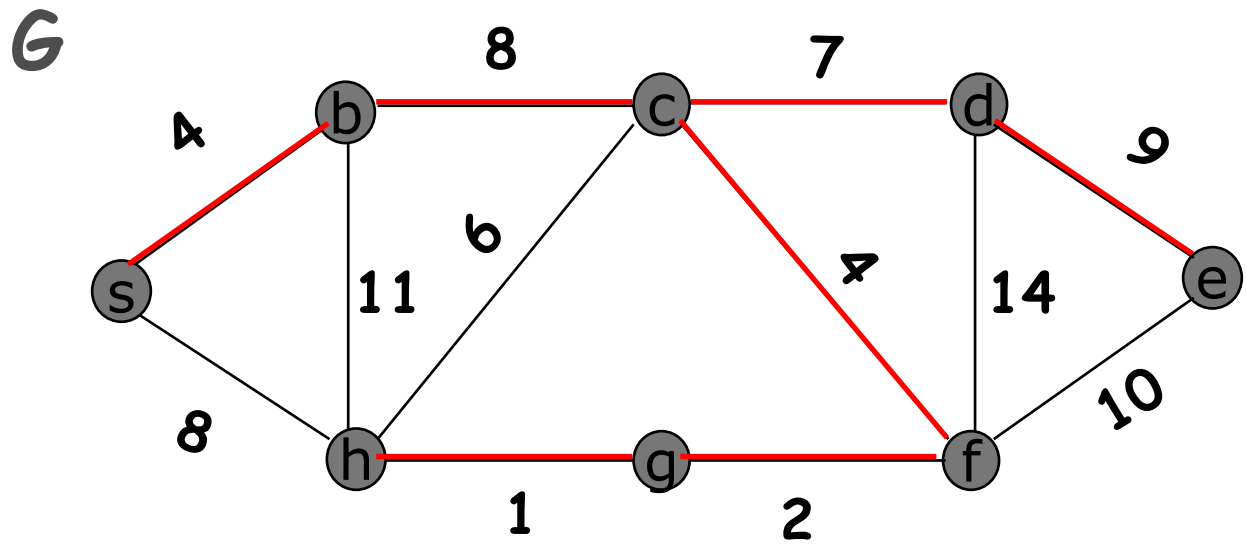


$Q = \{[d, 7],[e, 10],[h, 1]\}$

Si estrae h da Q e si aggiornano i campi a e π dei nodi adiacenti a h che si trovano in Q

$Q = \{[d, 7],[e, 10]\}$

Un esempio



Q = {[d, 7],[e, 10]}

Si estrae d da Q e si aggiorna il campo a e π di e

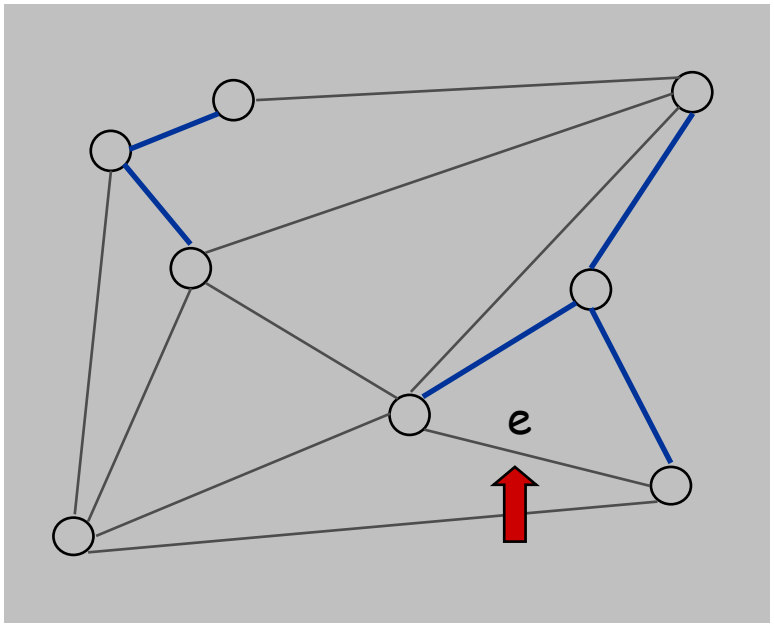
Q = {[e, 9]}

Si estrae e da Q

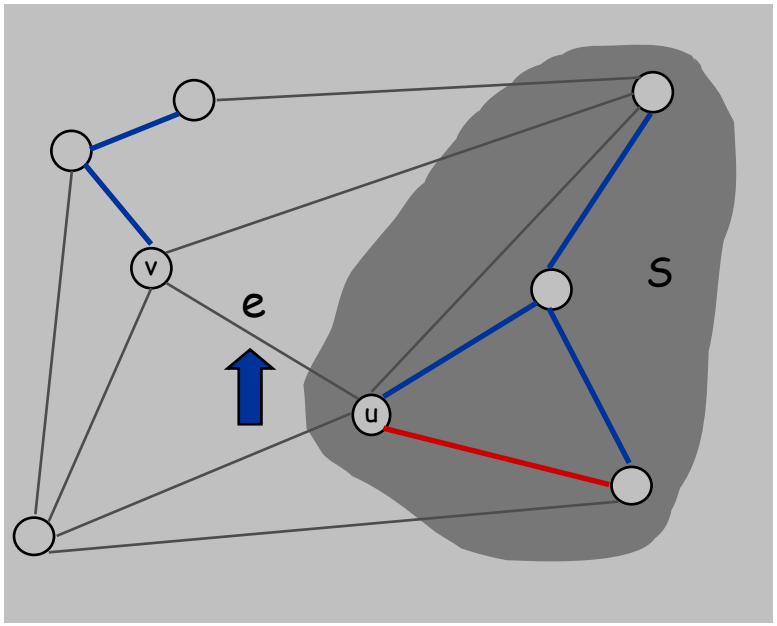
Algoritmo di Kruskal

- Algoritmo di Kruskal's . [Kruskal, 1956]
  - Considera ciascun arco in ordine non decrescente di peso
  - Caso 1: Se e crea un cliclo allora scarta e
  - Case 2: Altrimenti inserisce e in T

  - NB: durante l'esecuzione T è una foresta composta da uno o più alberi



Caso 1



Caso 2

Esempio

Archi in MST : rossi (già selezionati) e verdi (non ancora selezionati)

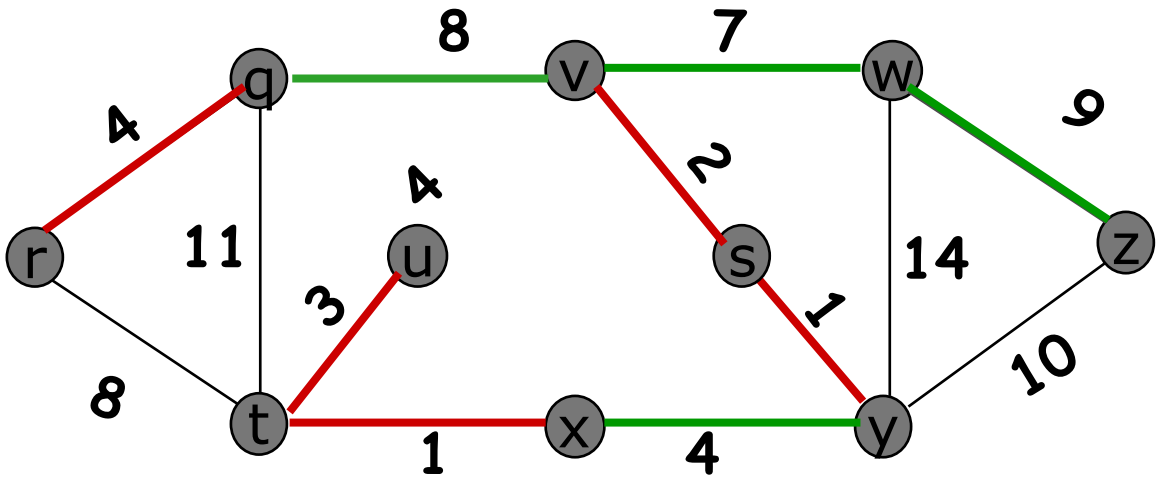
T={(r,q),(t,u),(t,x),(v,s),(s,y)} archi dell'MST già inseriti

Componenti connesse (alberi) in $G_T=(V,T)$:

$C_1=\{(r,q)\}$

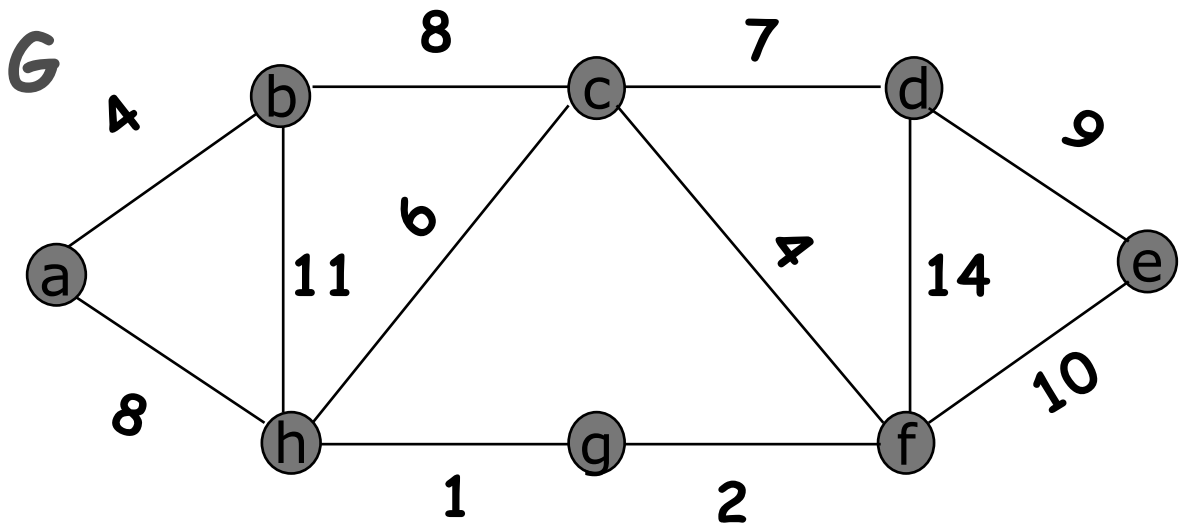
$C_2=\{(t,x),(t,u)\}$

$C_3=\{(s,y),(v,s)\}$



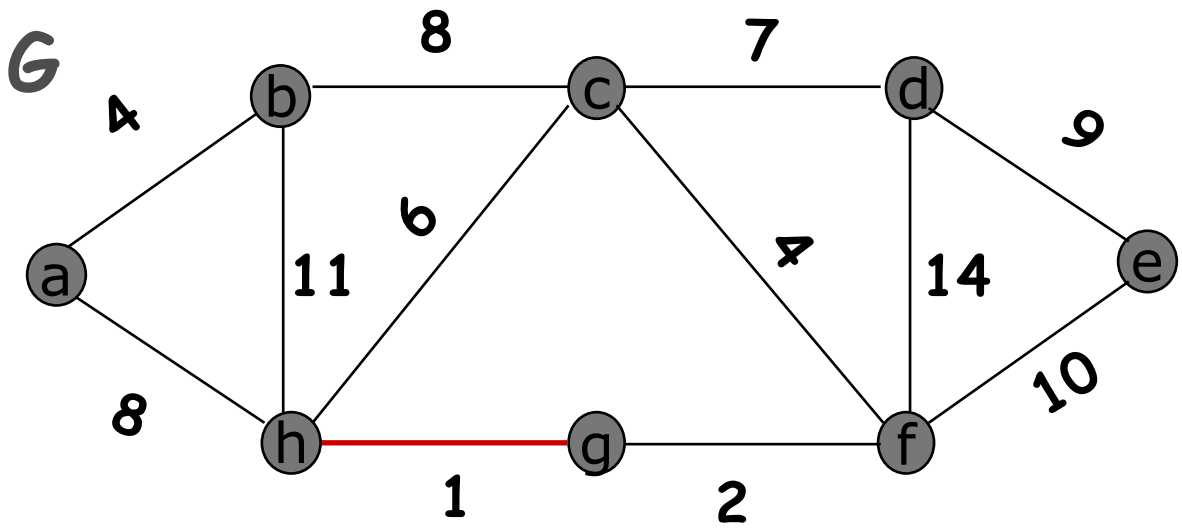
Il prossimo arco selezionato è (x,y)

Un esempio



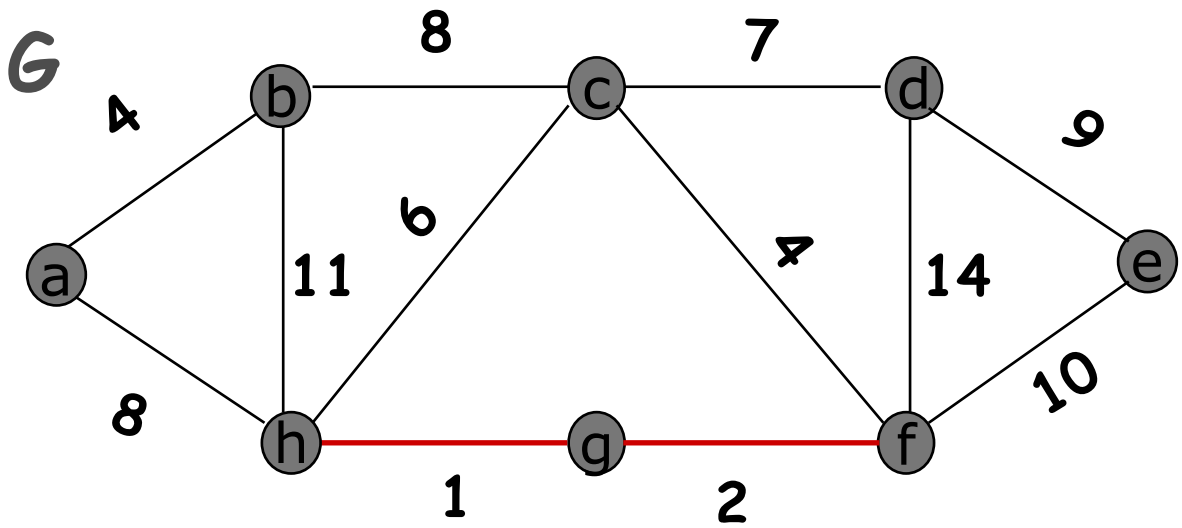
T = { }

Un esempio



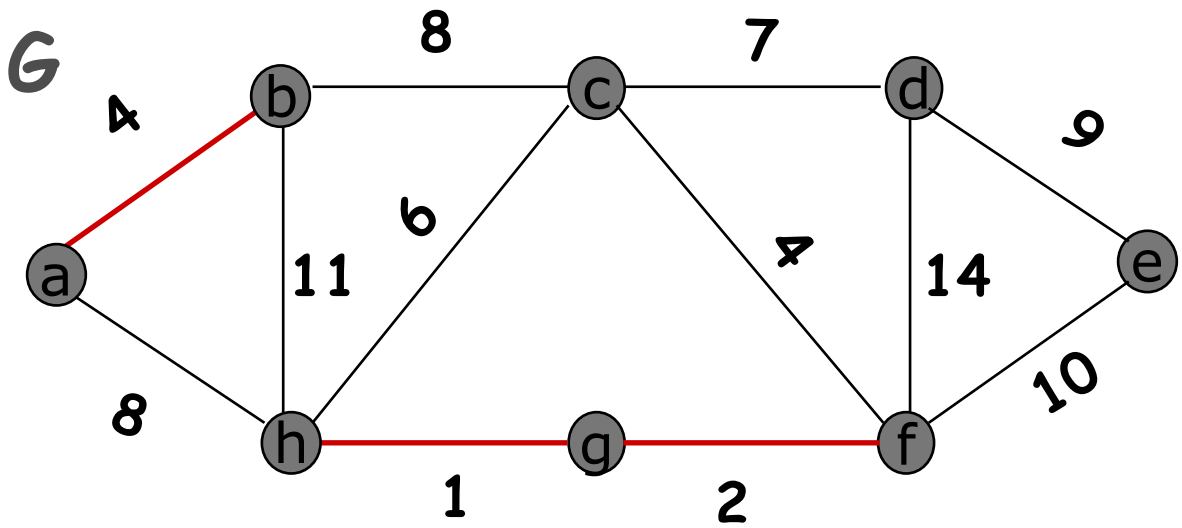
T = {(h,g) }

Un esempio



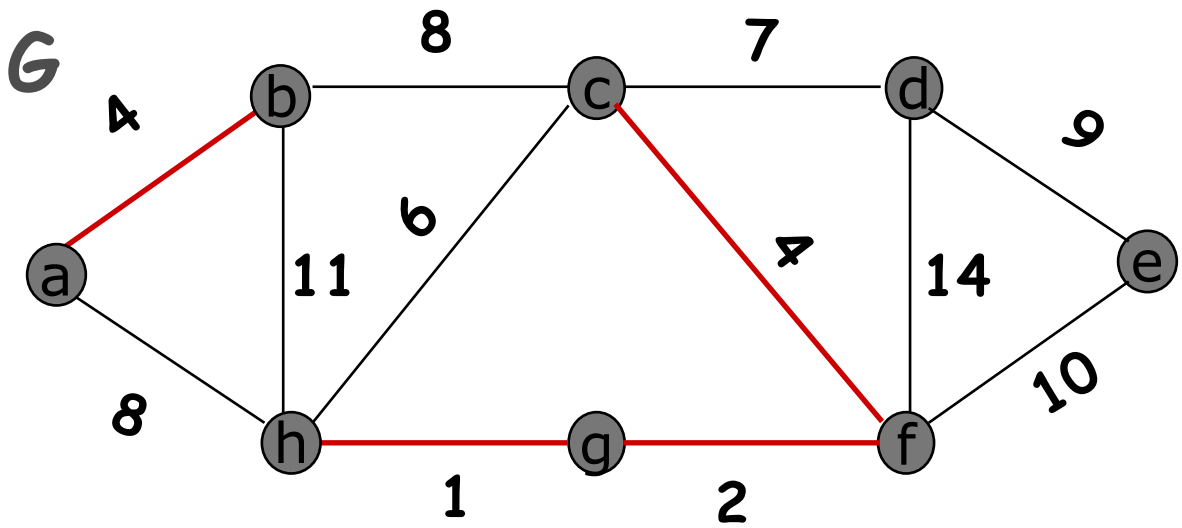
T = {(h,g), (g,f) }

Un esempio



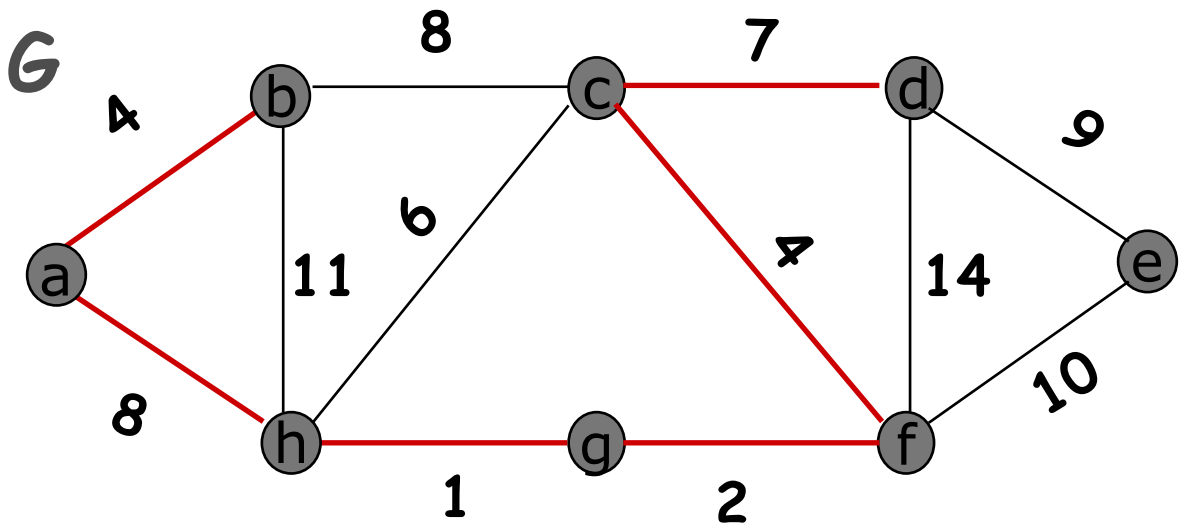
T = {(h,g), (g,f), (a,b) }

Un esempio



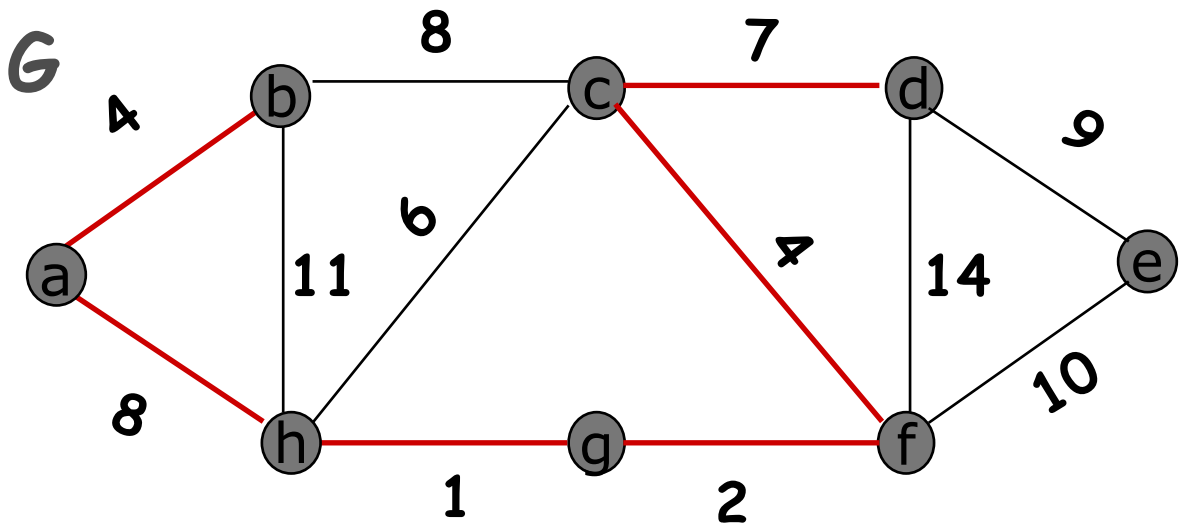
T = {(h,g), (g,f), (a,b),(c,f) }

Un esempio



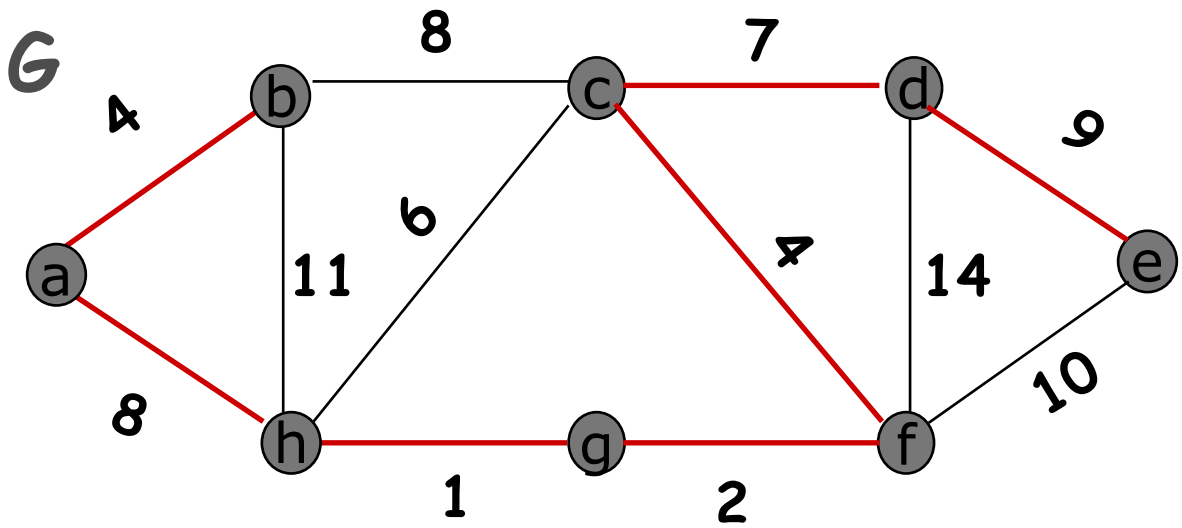
T = {(h,g), (g,f), (a,b),(c,f),(c,d), (a,h) }

Un esempio



T = {(h,q), (g,f), (a,b),(c,f),(c,d), (a,h) }

Un esempio



T = {(h,g), (g,f), (a,b),(c,f),(c,d), (a,h),(d,e) }

Correttezza dell'algoritmo di Kruskal

- L'insieme di archi T prodotto dall'algoritmo di Kruskal è un MST

- Dim. (per il caso in cui glia archi hanno costi a due a due distinti)

- Prima dimostriamo che ogni arco di T è anche un arco di MST
- Ad ogni passo l'algoritmo inserisce in T l'arco **e=(u,v)** di **peso minimo** tra quelli **non ancora esaminati** e **che non creano cicli in T**.
- Il fatto che l'arco **e** non crea cicli in T vuol dire che T fino a quel momento non contiene un percorso che collega **u** a **v**.
- Ricordiamo che gli archi di T formano uno o più alberi. Consideriamo l'albero contenente **u** e chiamiamo S l'insieme dei suoi vertici. Ovviamente **v** non è in S altrimenti esisterebbe un percorso da **u** a **v**.
  - NB: se non abbiamo ancora inserito in T archi incidenti su u, S contiene solo u.
- L'arco **e=(u,v)** è l'arco di peso minimo tra quelli che attraversano il taglio [S, V-S] (perché?) e quindi per la proprietà del taglio l'arco **e=(u,v)** è nel minimo albero ricoprente.

Continua nella prossima slide

Correttezza dell'algoritmo di Kruskal

- Ora dimostriamo che T è un albero ricoprente.
- T è un albero ricoprente perchè
  - l'algoritmo non introduce mai cicli in T (ovvio!)
  - connette tutti i vertici
    - Se così non fosse esisterebbe un insieme W non vuoto e di al più n-1 vertici tale che non c'è alcun arco di T che connette un vertice di W ad uno di V-W.
    - Siccome il grafo input G è connesso devono esistere uno o più archi in G che connettono vertici di W a vertici di V-W
    - Dal momento che l'algoritmo di Kruskal esamina tutti gli archi avrebbe selezionato sicuramente uno degli archi che connettono un vertice di W ad uno di V-W
    - Quindi non può esistere alcun insieme W non vuoto e di al più n-1 vertici tale che in T nessun vertice di W è connesso ad un vertice di V-W.

Implementazione dell'algoritmo di Kruskal

- Abbiamo bisogno di rappresentare le componenti connesse (alberi della foresta)
- Ciascuna componente connessa è un insieme di vertici disgiunto da ogni altro insieme.

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

   foreach (u ∈ V) make a set containing singleton u

   for i = 1 to m
      (u,v) = ei
      if (u and v are in different sets) {
         T ← T ∪ {ei}
         merge the sets containing u and v
      }
   return T
}
```

are u and v in different connected components?

merge two components

Implementazione dell'algoritmo di Kruskal

- Ciascun albero della foresta è rappresentato dal suo insieme di vertici
- Per rappresentare questi insiemi di vertici, si utilizza la struttura dati Union-Find per la rappresentazione di insiemi disgiunti
- Operazioni supportate dalla struttura dati Union-Find
- MakeUnionFind(S): crea una collezione di insiemi ognuno dei quali contiene un elemento di S
  - Nella fase di inizializzazione dell'algoritmo di Kruskal viene invocato MakeUnionFind(V): ciascun insieme creato corrisponde ad un albero con un solo vertice.
- Find (x): restituisce l'insieme che contiene x
  - Per ciascun arco esaminato (u,v), l'algoritmo di Kruskal invoca find(u) e find(v). Se entrambe le chiamate restituiscono lo stesso insieme allora vuol dire che u e v sono nello stesso albero e quindi (u,v) crea un ciclio in T.
- Union(x,y): unisce l'insieme contenente x a quello contenente x
  - Se l'arco (u,v) non crea un ciclo in T allora l'algoritmo di Kruskal invoca Union(Find(u),Find(v)) per unire le componenti connesse di u e v in un'unica componente connessa

Implementazione dell'algoritmo di Kruskal con Union-Find

```
Kruskal(G, c) {
   Sort edges weights so that c_1 ≤ c_2 ≤ ... ≤ c_m.
   T ← ϕ

   MakeUnionFind(V)   //create n singletons for the n vertices

   for i = 1 to m
      (u,v) = e_i
      if (Find(u)≠Find(v)) {
         T ← T ∪ {e_i}
         Union(Find(u),Find(v))
      }
   return T
}
```

Implementazione di Union-Find con array

- La struttura dati Union-Find può essere implementata in vari modi
- Implementazione di Union-Find con array
  - Gli elementi sono etichettati con interi consecutivi da 1 ad n e ad ogni elemento è associata una cella dell'array S che contiene il nome del suo insieme di appartenenza.
  - Find(x): $O(1)$. Basta accedere alla cella di indice x dell'array S
  - Union: $O(n)$. Occorre aggiornare le celle associate agli elementi dei due insiemi uniti.
  - MakeUnionFind $O(n)$: Occorre inizializzare tutte le celle.

Analisi dell'algoritmo di Kruskal in questo caso:

- Inizializzazione $O(n)+O(m \log m)=O(m \log n^2)=O(m \log n)$.
  - $O(n)$ creare la struttura Union-Find e $O(m \log m)$ ordinare gli archi

Per ogni arco esaminato: $O(1)$ per le 2 find.

- In totale, 2m find → $O(m)$

Per ogni arco aggiunto a T : $O(n)$ per la union

- In totale n-1 union (perché?) → $O(n^2)$

Algoritmo: $O(m \log n) + O(n^2)$

Implementazione di Union-Find con array e union-by-size

- Implementazione di Union-Find con array ed union-by-size
  - Stessa implementazione della slide precedente ma si usa anche un altro array A per mantenere traccia della cardinalità di ciascun insieme. L'array ha n celle perché inizialmente ci sono n insiemi.
  - La Find(x) è identica a prima
  - MakeUnionFind O(n): occorre inizializzare tutte le celle S e tutte le celle di A. Inizialmente le celle di A sono tutte uguali ad 1.
  - Union: si guarda quali dei due insiemi è più piccolo e si aggiornano solo le celle di S corrispondenti agli elementi di questo insieme. In queste celle viene messo il nome dell'insieme più grande. La cella dell'array A corrispondente all'insieme più piccolo viene posta a 0 mentre quella corrispondente all'insieme più grande viene posta uguale alla somma delle cardinalità dei due insiemi uniti.
    - Corrisponde ad inserire gli elementi dell'insieme più piccolo in quello più grande.

Implementazione di Union-Find con array e union-by-size

- Nell'implementazione con union-by-size la singola operazione di unione richiede ancora O(n) nel caso pessimo perché i due insiemi potrebbero avere entrambi dimensione pari ad n diviso per una certa costante.
- Vediamo però cosa accade quando consideriamo una sequenza arbitrariamente lunga di unioni.
- Notiamo che il tempo richiesto da un'operazione di unione dipende dal numero di celle di S e A che vengono aggiornate.
- Ciascuna cella di S e A viene aggiornata un numero di volte pari al numero di volte in cui l'elemento corrispondente viene spostato in un nuovo insieme.
- Quante volte può essere spostato un elemento?
- Notiamo che ogni volta che facciamo un'unione, un elemento cambia insieme di appartenenza solo se proviene dall'insieme che ha dimensione minore o uguale dell'altro. Ciò vuol dire che l'insieme in cui viene spostato ha dimensione pari almeno al doppio dell'insieme di partenza → ogni elemento viene spostato al più log(n) volte. NB: l'insieme piu` grande in cui l'elemento puo` essere spostato ha cardinalita` n

Continua nella prossima slide

Implementazione di Union-Find con array e union-by-size

Analisi algoritmo di Kruskal in questo caso:

- Inizializzazione $O(n)+O(m \log m)=O(m \log m)$.
  - $O(n)$ creare la struttura Union-Find e
  - $O(m \log m)$ ordinare gli archi

- In totale il numero di find è 2m che in totale richiedono $O(m)$

- Tutte le union, per il risultato dimostrato alla slide precedente, richiedono $O(n \log n)$

Algoritmo: $O(m \log m+n \log n)=O(m \log m)=O(m \log n^2)$
$=O(m \log n)$