Sommario della lezione

- Cammini minimi in grafi
  - Applicazioni
  - Algoritmi
- Alberi ricoprenti minimi in grafi
  - Applicazioni
  - Algoritmi

Primo problema: Cammini minimi in grafi

Input al problema:

- Grafo diretto $G = (V, E)$
- Nodo sorgente s
- Lunghezze $\ell(e) > 0$ per ogni arco $e \in E$

Output al problema: un cammino di *lunghezza minima* da s ad ogni altro nodo t nel grafo (dove la lunghezza di un cammino è pari alla somma delle lunghezze degli archi che lo formano)

Ricordiamo che nel caso in cui le lunghezze $\ell(e)$ degli archi sono tutte uguali tra di loro, allora il problema è risolvibile mediante una visita BFS del grafo.

E perchè mai vogliamo calcolare cammini minimi in grafi?

Anche Google lo fà...

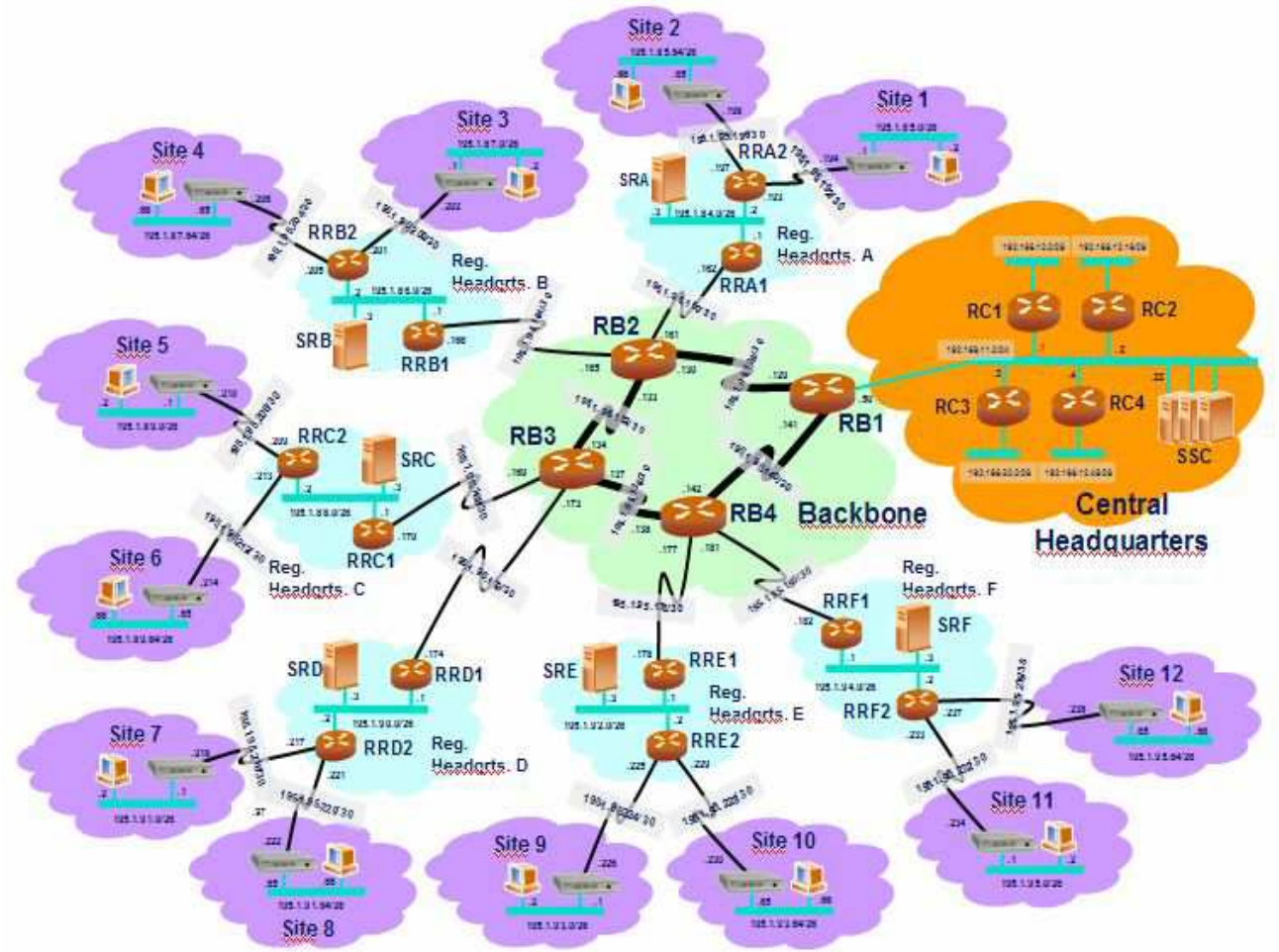
Questo è il calcolo del percorso *più breve* dalla stazione di Salerno all'Università mediante Google Map

Un’altra applicazione:

Di sotto vi è l’output di un algoritmo (che si vende!) per il calcolo del percorso più breve tra due generiche stazioni della metropolitana di Londra.

Ed un’altra ancora:

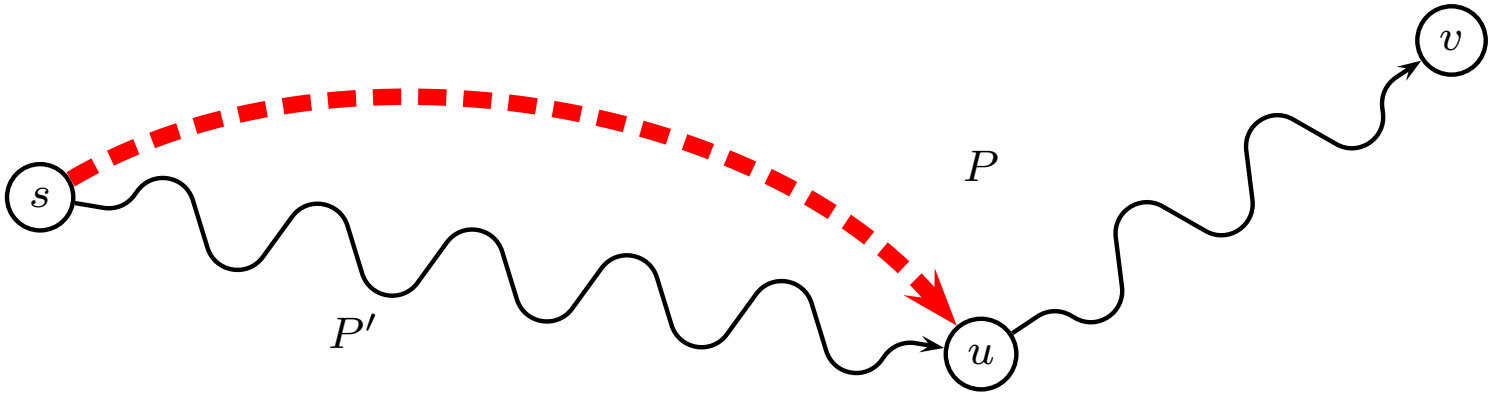
Open Shortest Path First (OSPF) è un protocollo dinamico per l’ instradamento di pacchetti, ampiamente usato in reti basate sull’ Internet Protocol (IP). Esso fà uso dell’algoritmo oggetto di questa lezione.



Cammini minimi via tecnica Greedy

Ricordiamo innanzitutto un fatto che notammo un pò di tempo fà:

- Se un cammino P di lunghezza minima dal nodo s al nodo v passa attraverso il nodo u, allora la parte di cammino P' da s a u è esso stesso un cammino di lunghezza minima da s a u



Perchè? Perchè se P' non lo fosse, allora esisterebbe un cammino differente s a v (linea rossa tratteggiata) di lunghezza inferiore. Ma ciò implicherebbe che si potrebbe sostituire il cammino P' con la linea rossa tratteggiata ed andare da s a v con un cammino di lunghezza totale inferiore a quella di P, contro l'ipotesi dell'ottimalità di P

E perchè vogliamo ricordare questo fatto?

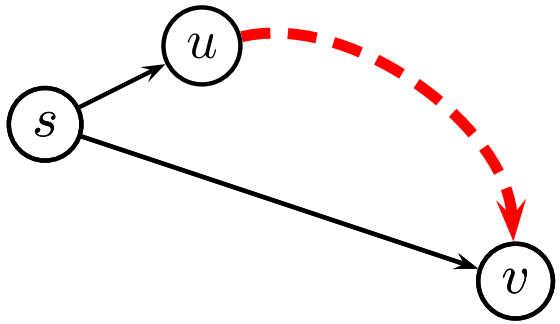
Perchè esso ci dice che un cammino minimo da s ad un nodo generico v non ha una struttura arbitraria (e quindi potrebbe essere difficile da trovare), ma è una *estensione* di cammini minimi da s a nodi u più vicini ad s di quanto lo sia v.

Questa osservazione dovrebbe suggerirci un'idea di tipo Greedy per calcolare cammini di lunghezza minima verso un numero sempre maggiore di nodi: *dato un insieme di cammini minimi che partono da s e raggiungono nodi $v_1, v_2, \ldots, v_k$, determiniamo l'estensione di lunghezza minore (visto che stiamo minimizzando) di uno di questi cammini, che raggiunge un nodo nuovo*, ovverosia tra tutti i vicini dei nodi $v_1, v_2, \ldots, v_k$ trova il nodo x (non ancora considerato) più vicino a s. Poi itera fin quando non hai trovato i cammini di lunghezza minima da s ad ogni nodo del grafo

Di quest'idea almeno il punto di partenza è facile da realizzare: trova il nodo x vicino di s che dista di meno da s e collegalo con il relativo arco.

Giustifichiamo un pò meglio l'idea, a partire dal primo passo

Al primo passo scegliamo un *vicino* v del nodo di partenza s (cioè v è connesso ad s da un arco diretto) *che dista di meno* da s.
E se abbiamo già sbagliato? Ovvero esisteva un altro percorso, di lunghezza totale minore di $\ell(s, v)$, che portava da s a v passando magari per un nodo u vicino di s?

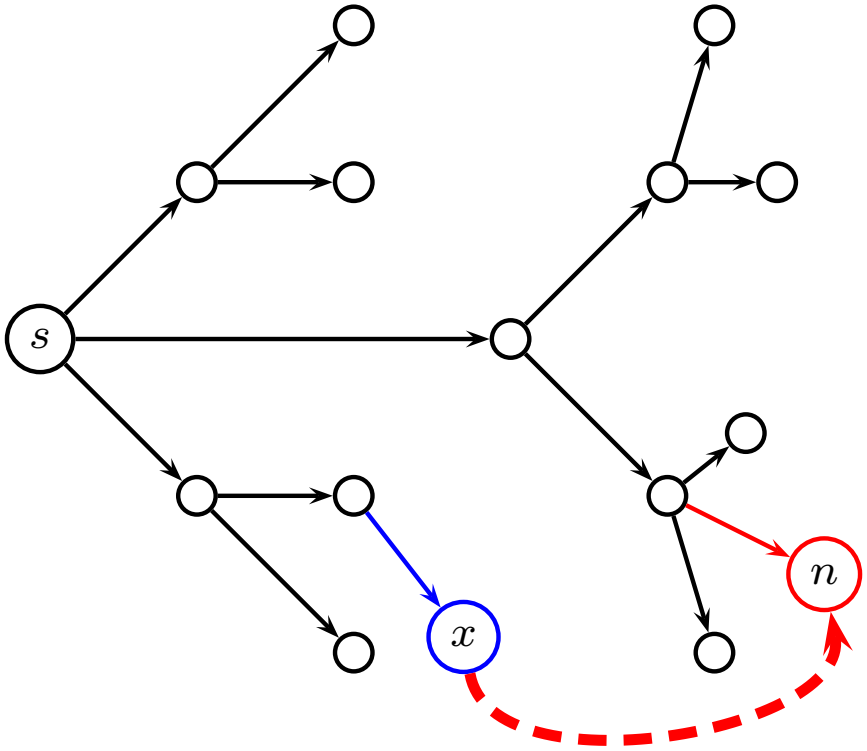


Non è possibile! Infatti ciò contraddirrebbe il fatto che v è il vicino di s *che dista di meno* da s (u disterebbe ancora di meno). Quindi non abbiamo sbagliato già fin dall'inizio...

Potremmo però aver sbagliato nell'iterazione...

...che consiste, ricordiamo, in: *dato un insieme di cammini minimi che partono da s e raggiungono nodi $v_1, v_2, \ldots, v_k$, trova l'estensione di lunghezza minore di uno di questi cammini, che raggiunge un nodo nuvo*.

Ovvero, vorremmo procedere nel modo seguente:



Dati i cammini minimi già costruiti, tra *tutti* i vertici adiacenti dei nodi ○ già considerati, andiamo a sceglierci quello che dista di meno da s, e lo colleghiamo con il rispettivo vicino. Fatto. Potrebbe esistere un percorso più breve da s ad n? No! Perchè se esistesse, non sarebbe n il nodo che, tra i vicini dei dei nodi ○ già considerati, dista di meno da s, in quanto x dista da s ancora di meno.

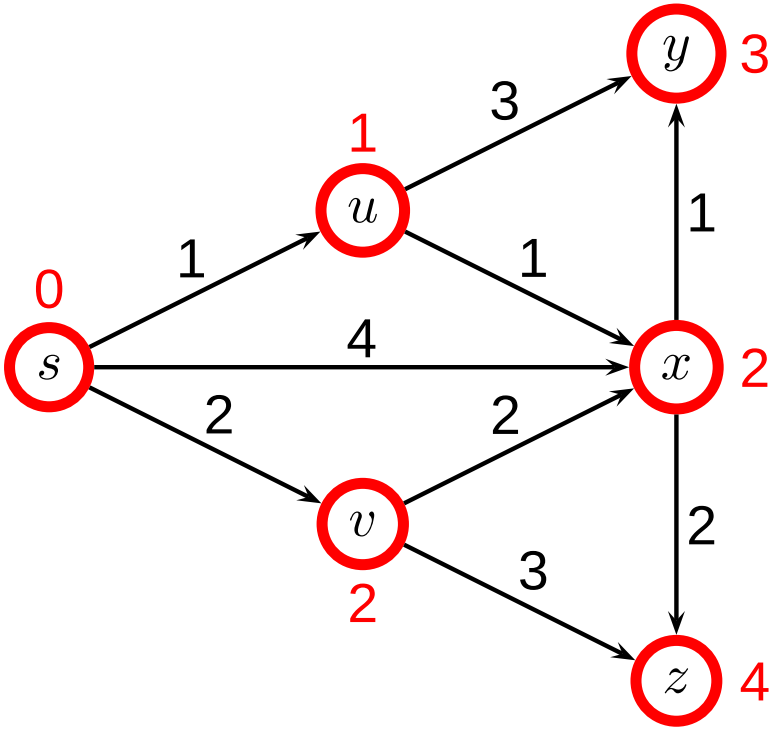
Cammini minimi via tecnica Greedy (Algoritmo di Dijkstra)

Formalizziamo l'intuizione precedentemente guadagnata e, per il momento, preoccupiamoci solo di calcolare la lunghezza $d[u]$ di un cammino di lunghezza minima dal nodo sorgente ad ogni nodo u nel grafo. Chiamiamo $d[u]$ *distanza del nodo* u *da* s.

- Mantieni un insieme S di *nodi esplorati* per cui abbiamo già determinato la distanza $d[u]$ da s.
- Inizializza $S = \{s\}$, $d[s] = 0$
- $\forall v \notin S$, calcola $d'[v] = \min_{e=(u,v):u \in S} d[u] + \ell(e)$
- Sia w il nodo che ha valore $d'[\cdot]$ minimo, aggiungi w in S e poni $d[w] = d'[w]$ (in altre parole w è, tra tutti i nodi $\notin S$ ed adiacenti ai nodi in S, quello più vicino a s).

L'algoritmo termina quando non ci sono più nodi inesplorati, ovvero quando $S = V$

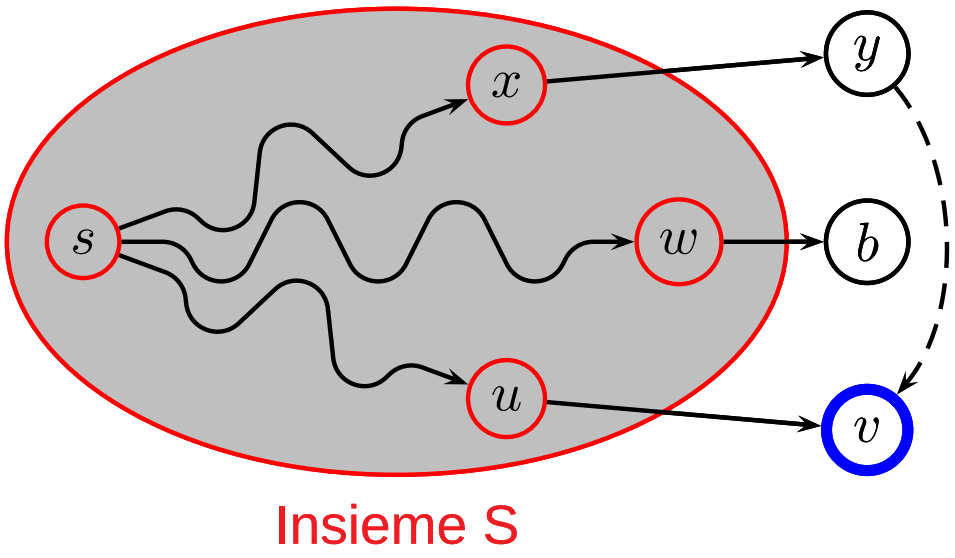
Esempio di esecuzione: i nodi rossi sono quelli esplorati



Analisi dell'algoritmo di Dijkstra

Ad ogni passo, $\forall v \in S\ d[v] =$distanza del nodo v da s

Vediamo come funziona l'algoritmo quando aggiunge un nodo v ad S:



Ci siamo calcolati $\forall t \notin S$, $d'[t] = \min_{e=(u,t):u\in S} d[u] + \ell(e)$, abbiamo scoperto che v ha il parametro d' minore di tutti, abbiamo posto $d[v] = d'[v] = d[u] + \ell(u, v)$ ed infine messo v in S. Potremmo aver sbagliato? (nel senso che esisteva un cammino da s a v più breve, di lunghezza $< d[v]$, ad es. passando da y)No! Perchè già la lunghezza del cammino da s a y che passa per x è uguale a $d'[y] \geq d'[v]$. Quindi, a maggior ragione, la lunghezza dell'intero cammino fino a v è $>$ di $d'[v]$.

Quindi l'algoritmo di Dijkstra calcola correttamente le distanze dei nodi da s

E se volessimo calcolarci anche i cammini da s di lungheza pari a tali distanze (ovvero i cammini di lunghezza minima)?

Basterà memorizzarci, ogniqualvolta aggiungiamo un nodo v ad S, anche l'arco (u, v) che abbiamo usato, ovvero l'arco per cui

$$d[v] = d[u] + \ell(u, v)$$

In questo modo conosceremo l'ultimo arco del cammino di lunghezza minima da s a v, se lo abbiamo fatto anche per u allora conosciamo anche l'ultimo arco del cammino di lunghezza minima da s a u, *e quindi gli ultimi due archi* del cammino di lunghezza minima da s a v, e così via...

Implementazione dell'algoritmo di Dijkstra

`Inizializza` $S = \{s\}$, $d[s] = 0$, $d'[v] = \infty \ \ \forall v \in V - \{s\}$

`While` $S \neq V$

`seleziona un nodo` $v \in V - S$ `con almeno un arco da` S `per cui` $d'[v] = \min_{e=(u,v):u \in S} d[u] + \ell(e)$ `è più piccolo possibile`

`Aggiungi` v `a` S `e poni` $d[v] = d'[v]$

Ad ogni istante, strutturiamo l'insieme dei nodi in $V - S$ come una MinCoda a Priorità Q, in base ai valori $d'[\cdot]$ a loro associati.
Ad ogni iterazione estraiamo da Q il nodo v con $d'[v]$ minimo, lo mettiamo in S e aggiorniamo Q (come?). Se $w \in Q$ è tale che $(v, w) \notin E$, allora $d'[w] = \min_{e=(u,w):u \in S} d[u] + \ell(e)$ rimane inalterato, se invece $(v, w) \in E$, allora $d'[w] = \min_{e=(u,w):u \in S} d[u] + \ell(e)$ può cambiare (diminuire) e occorrerà con un'operazione di tipo `DecreaseKey` aggiustare la struttura della coda a priorità. Quindi, per ogni generico arco (x, y) chiameremo `DecreaseKey` al più una volta, quando x viene aggiunto a S (il che avviene una volta sola).

Mettendo tutto insieme...

Usando una coda a priorità, l'algoritmo di Dijkstra può essere implementato in un grafo con n nodi ed m archi in modo da richiedere tempo $O(m)$, più il tempo per eseguire n `ExtractMin` ed m `DecreaseKey`.

Se usiamo un min-heap per implementare la coda a priorità in questione, ogni operazione di `ExtractMin` e `DecreaseKey` richiede tempo $O(\log n)$, per un gran totale di $O(m \log n)$ operazioni per implementare l'algoritmo di Dijkstra.

Vediamo un esempio di esecuzione dell'algoritmo.

Secondo problema: Minimo Sottografo Connesso Ricoprente

Supponiamo di avere n postazioni $V = \{v_1, v_2, \ldots, v_n\}$ e vogliamo costruire una rete di comunicazione su di essi, con i seguenti requisiti:

1. La rete deve essere connessa (cosicchè sia possibile andare da ogni nodo ad ogni altro nodo)
2. Vogliamo spendere il meno possibile (assumiamo che stabilire una connessione tra due postazioni ci costi qualcosa)

- È possibile stabilire una connessione tra alcune coppie (v_i, v_j) di locazioni (non è detto che sia possibile stabilire la connessione tra tutte le coppie), ad un costo $c(v_i, v_j) \geq 0$.

Possiamo quindi rappresentare l'insieme delle possibili connessioni mediante un grafo $G = (V, E)$ (dove i vertici rappresentano le locazioni e gli archi le connessioni che si possono stabilire tra connessioni). Inoltre, ciascun arco $e = (u, v)$ ha un costo c_e ad esso associato.

Minimo Sottografo Connesso Ricoprente

Input al problema:

- Grafo $G = (V, E)$, costi $c(e) \geq 0$ per ogni arco $e \in E$

Output al problema: Sottoinsieme di archi $T \subseteq E$ tali che il sottografo (V, T) sia connesso ed il costo totale $\sum_{e \in T} c(e)$ sia il più piccolo possibile.

Fatto 1. Esiste una soluzione T di minimo costo al problema sopra esposto in cui (V, T) è un albero.

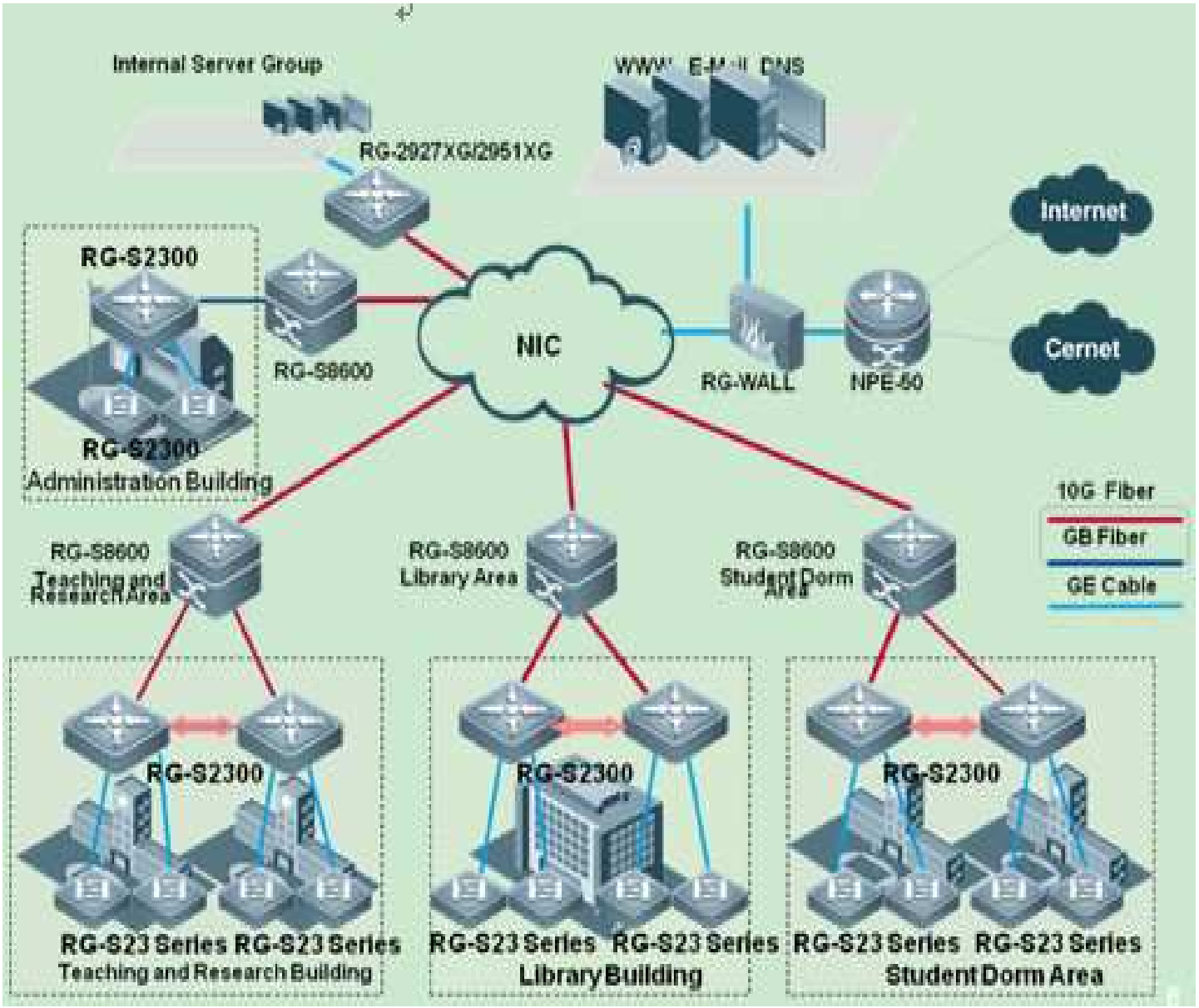
Sia T una soluzione di minimo costo. Per definizione (V, T) è connesso. Se (V, T) è un albero siamo a posto. Se non è un albero, allora contiene un ciclo C. Eliminando un qualsiasi arco (u, v) da C, il sottografo (V, T) rimane connesso in quanto sarà sempre possibile andare da u a v seguendo la "via lunga" rimanente del ciclo C. La soluzione T' così ottenuta ha un costo $\leq$ del costo di T e procedendo per via di eliminazione di cicli otterremo alla fine un albero $(V, \overline{T})$ con costo di $\overline{T} \leq$ costo di T,
$\Rightarrow \overline{T}$ è di costo minimo ma questa volta $(V, \overline{T})$ è *finalmente* un albero.

Quindi in realtà cerchiamo un Minimo *Albero* Ricoprente (MST)

Tuttavia ciò , almeno in linea di principio, *non* rende il problema più semplice. Infatti, il grafo completo K_n (in cui vi è un arco tra ogni coppia di nodi) possiede ben n^{n-2} distinti sottoalberi ricoprenti! (tra cui dovremmo cercarci quello di minimo costo).

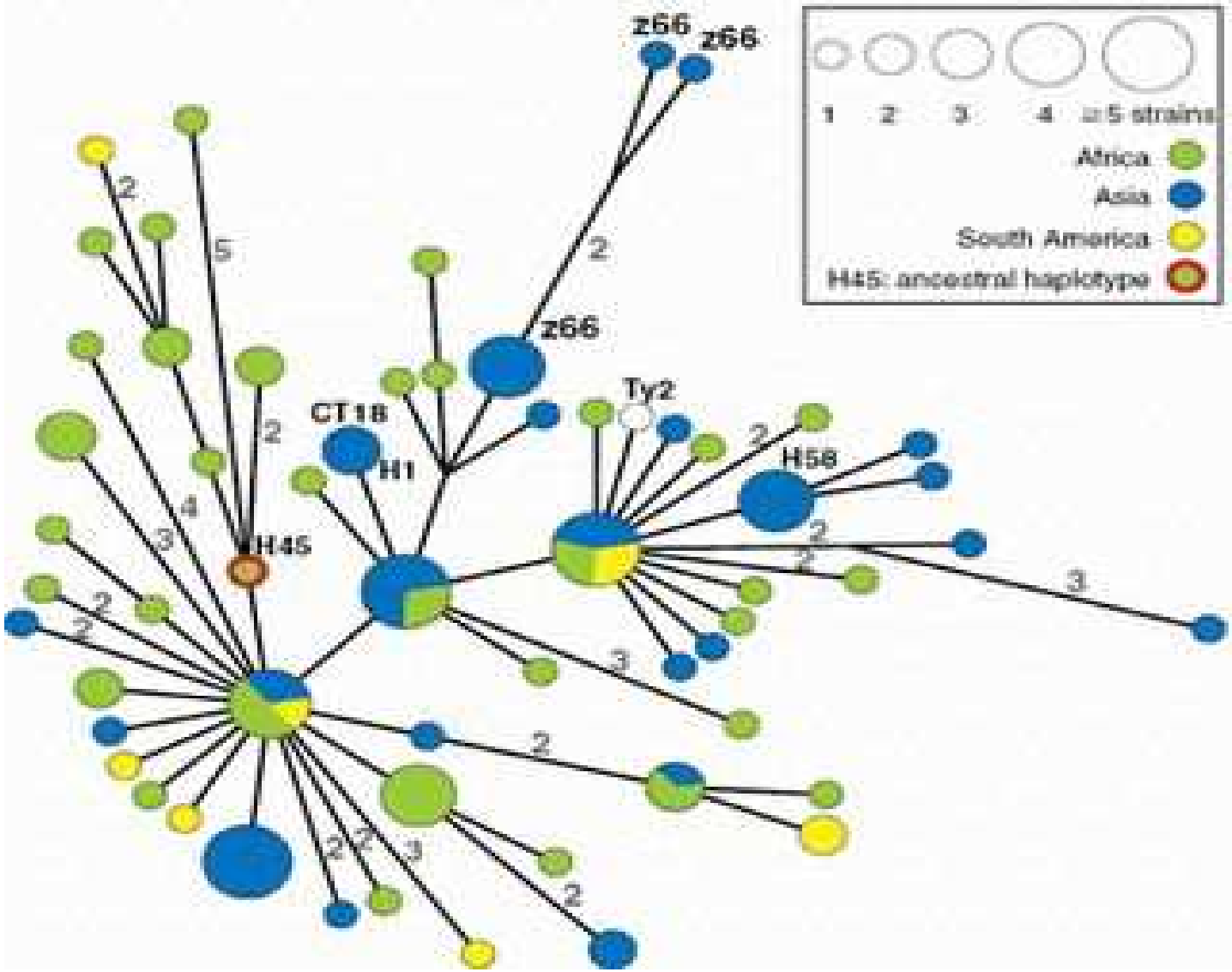
Prima però di procedere oltre, si può sapere a che serve trovare un Minimo Albero Ricoprente di un grafo?

A tante cose, ad es. per reti di comunicazione economiche ed efficienti:



O in studi biologici:

Qui raffigurato vi è il MST costruito sui 59 genotipi (nodi) del *Salmonella Typhi* (batterio del Tifo), con la lunghezza di ogni arco proporzionale alla differenza genomica tra di essi



Intuizione per un possibile algoritmo greedy

Siano $e_1, e_2, \ldots, e_m$ gli archi del grafo, ordinati in modo che $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$

- Un'idea potrebbe essere quella di aggiungere all'insieme $T \subseteq E$ che vogliamo costruire (di minimo costo totale, ricordiamolo), uno ad uno gli archi e_i, iniziando da quello che costa meno (ovvero e_1) e proseguendo via via con quelli di peso maggiore.
- Se l'aggiunta di un arco e_i all'insieme attuale T crea un ciclo, non va bene. Scartiamo l'arco e_i e prendiamo in considerazione l'arco e_{i+1}, iterando il processo.
- Smettiamo di aggiungere archi a T quando abbiamo connesso tutti i vertici di V, ovvero quando (V, T) è un albero.

Potremmo però procedere in maniera analoga all'algoritmo di Dijkstra

Iniziamo con un nodo radice s e tentiamo di costruire, in modo greedy, un albero con radice in s. Ad ogni passo attacchiamo all'albero corrente il nodo per cui ci costa meno farlo.

- Per fare ciò manteniamo ad ogni istante un insieme $S \subseteq V$ su cui un Minimo Albero Ricoprente T è stato costruito fin'ora. Inizialmente $S = \{s\}$.
- Ad ogni iterazione aumentiamo l'albero di un nodo, aggiungendo il nodo $v \notin S$ che minimizza il costo di tale aumento, pari a $\min_{e=(u,v):u \in S} c(e)$, ed includiamo l'arco $e = (u, v)$ che ottiene questo minimo all'albero corrente.
- Come prima, smettiamo di aggiungere archi a T quando abbiamo connesso tutti i vertici di V, ovvero quando (V, T) è un albero.

Quale dei due metodi funziona (ovvero produce un MST)?

Entrambi, per fortuna.

Il primo metodo porta all'*Algoritmo di Kruskal*, il secondo all'*Algoritmo di Prim*.

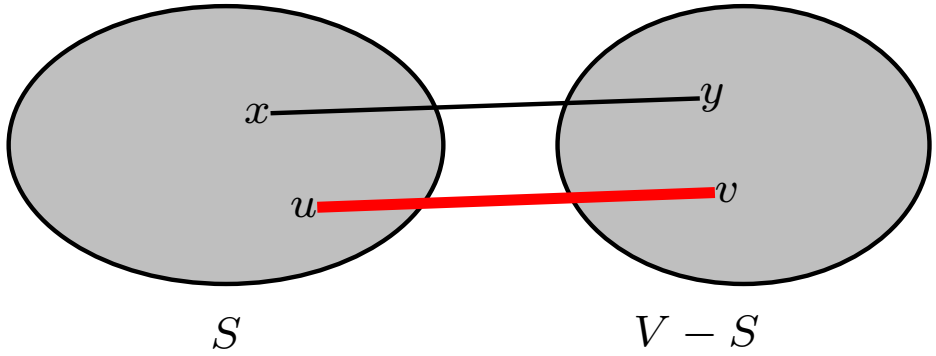
Per provare che i metodi prima esposti producono correttamente un MST, abbiamo bisogno di qualche risultato preliminare.

Innazitutto assumiamo (per il momento) che i costi degli archi siano tutti diversi tra di loro, poi vedremo come gestire la situazione nel caso in cui nel grafo vi sono anche archi di egual costo

Primo risultato preliminare

Sia $\emptyset \neq S \subset V$ un sottoinsieme dei nodi, e sia $e = (u, v)$ l'arco di *costo minimo* con un estremo in S e l'altro in $V - S$. Allora *ogni* MST contiene l'arco e.

Supponiamo che ciò non sia vero, e sia T un MST che *non* contiene e. È ovvio che T dovrà contenere almeno un'arco $a = (x, y) \neq (u, v) = e$ con un'estremo in S e l'altro in $V - S$ (altrimenti come farebbe T a connettere tra di loro tutti i nodi di V?)

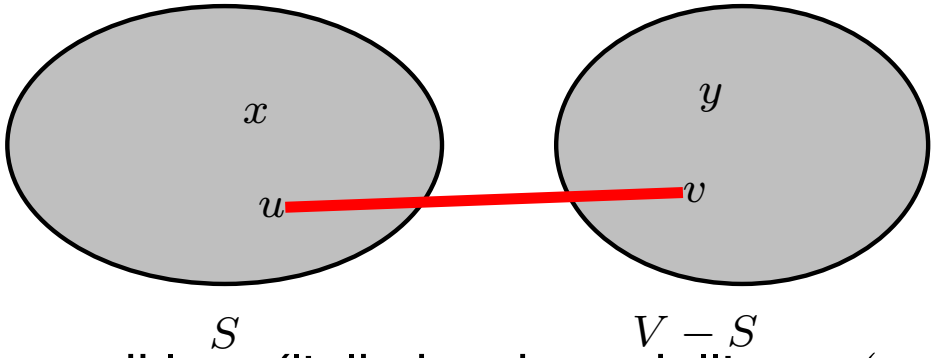


Aggiungiamo a T l'arco $e = (u, v)$, (per ipotesi $c(u, v) < c(x, y)$) che succede nell'albero T? Si crea un ciclo!Eliminiamo allora l'arco (x, y)

Primo risultato preliminare

Sia $\emptyset \neq S \subset V$ un sottoinsieme dei nodi, e sia $e = (u, v)$ l'arco di *costo minimo* con un estremo in S e l'altro in $V - S$. Allora *ogni* MST contiene l'arco e.

Supponiamo che ciò non sia vero, e sia T un MST che *non* contiene e. È ovvio che T dovrà contenere almeno un'arco $a = (x, y) \neq (u, v) = e$ con un'estremo in S e l'altro in $V - S$ (altrimenti come farebbe T a connettere tra di loro tutti i nodi di V?)

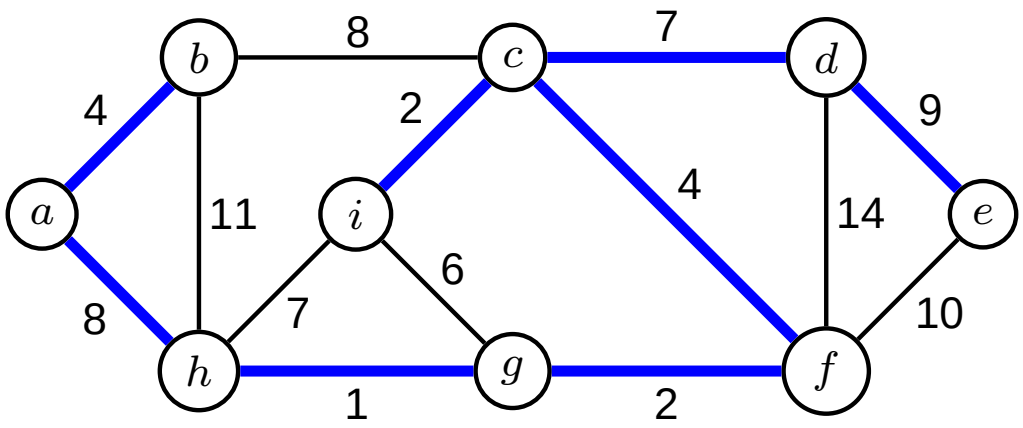


Aggiungiamo a T l'arco $e = (u, v)$, (per ipotesi $c(u, v) < c(x, y)$) che succede nell'albero T? Si crea un ciclo! Eliminiamo allora l'arco (x, y). I nodi in S rimangono connessi tra di loro (l'eliminazione dell'arco (x, y) non influenza i cammini in S), analoga cosa per i nodi in $V - S$ $\Rightarrow$da ogni nodo di S è raggiungibile ogni nodo di $V - S$, attraverso l'arco (u, v) $\Rightarrow \exists$un nuovo albero T' che connette tutti i vertici di V, con costo$(T') <$ costo(T), contro l'ipotesi.

Esempio di esecuzione dell'algoritmo di Kruskal

Archi ordinati per costo:

$(h, g), (i, c), (g, f), (a, b), (c, f), (i, g), (c, d), (i, h), (a, h), (b, c), (d, e), (b, h), (d, f)$



L'algoritmo di Kruskal produce un MST

Aggiungi a T uno ad uno gli archi del grafo, in ordine di costo crescente, saltando gli archi che creano cicli con gli archi già aggiunti.

- Sia $e = (v, w)$ un generico arco inserito in T dall'algoritmo di Kruskal, e sia S l'insieme di tutti i nodi connessi a v attraverso un cammino, al momento appena prima di aggiungere (v, w) a T.
- Ovviamente vale che $v \in S$, mentre $w \notin S$, altrimenti l'arco (v, w) creerebbe un ciclo.
- Inoltre, negli istanti precedenti l'algoritmo non ha incontrato nessun arco da nodi in S a nodi in $V - S$, altrimenti un tale arco sarebbe stato aggiunto, visto che non creava cicli.
- Pertanto l'arco (v, w) è il primo arco da S a $V - S$ che l'algoritmo incontra, ovvero è l'arco di minor costo da S a $V - S$ che, abbiamo visto, appartiene ad ogni MST.
- Ci rimane da mostrare che l'output dell'algoritmo di Kruskal è un albero

Facciamolo:

- Sicuramente, per costruzione, l'output (V, T) non contiene cicli.
- Potrebbe (V, T) non essere connesso? Ovvero potrebbe esistere un $\emptyset \neq S \subset V$ per cui in T non esiste alcun arco da S a $V - S$?
- Sicuramente no! Infatti, poichè il grafo G è connesso, un tale arco e esiste sicuramente in G e poichè l'algoritmo di Kruskal esamina tutti gli archi di G, prima o poi incontrerà tale arco e e lo inserirà, visto che non crea cicli.

L'algoritmo di Prim per MST

- Inizialmente $S = \{s\}$, $T = \emptyset$.
- Ad ogni iterazione aumentiamo l'albero di un nodo, aggiungendo il nodo $v \notin S$ che minimizza il costo di tale aumento, pari a $\min_{e=(u,v):u \in S} c(e)$, ed includiamo l'arco $e = (u, v)$ che ottiene questo minimo all'albero corrente T.
- Terminiamo quando (V, T) è un albero.

Che l'algoritmo di Prim produca un albero è ovvio, visto che aggiunge archi solo da nodi già tra di loro connessi in S a nuovi nodi "fuori" di S (quindi non crea cicli). Inoltre, ad ogni passo aggiunge a T l'arco di minimo costo che ha un estremo u in S (insieme dei nodi su cui un albero ricoprente parziale è stato già costruito) ad un nodo $v \in V - S$.

Dalla proprietà prima vista, tale arco appartiene ad ogni MST del grafo (cioè, di nuovo l'algoritmo non inserisce mai archi che non appartengono a MST, quindi produce effettivamente un MST).

Che succede se esistono archi di costo uguale?

Non cambia nulla. Innanzitutto possiamo “perturbare” i costi degli archi di una piccola quantità in modo che i nuovi costi siano ora tutti diversi tra di loro, ed in modo che l’ordine relativo tra i costi degli archi rimanga inalterato rispetto a prima, cosicchè i due algoritmi prima visti effettuino l’esame degli archi nello stesso ordine, (cioè sia nel caso in cui i costi siano uguali che nel nuovo caso in cui sono diversi).

Inoltre, ogni alberto T che è MST per i nuovi costi è MST anche per i vecchi costi. Infatti, se per assurdo esistesse un T^* per i vecchi costi per cui costo(T^*) <costo(T), allora per una perturbazione sufficientemente piccola che trasforma i costi $c(u,v)$ degli archi in nuovi costi $c'(u,v)$ continuerebbe a valere costo$'(T^*)$ <costo$'(T)$, contro l’ipotesi che T è un MST per i nuovi costi.

Implementazione dell'algoritmo di Prim per MST

L'implementazione è simile a quella dell'algoritmo di Dijkstra: occorre decidere quale nodo aggiungere all'albero T con nodi S che stiamo "crescendo".

Per ogni nodo $v \in V - S$, manteniamo un valore $a(v) = \min_{e=(u,v):u \in S} c(e)$ che rappresenta il costo in cui incorriamo per aggiungere il nodo v all'albero, usando l'arco di minimo costo per tale aggiunta.

Manteniamo i nodi in una coda a priorità, organizzata in base ai valori $a(v)$; selezionamo un nodo con l'operazione `ExtractMin`, e effettuiamo l'aggiornamento dei valori $a(v)$ con l'operazione `DecreaseKey`.

Implementazione ed analisi dell'algoritmo di Prim per MST

```
  MST-PRIM(G = (V, E), w, r)
1 Q ← V
2 For each u ∈ Q
3   a(v) ← ∞
4 a(r) ← 0, parent(r) ← NIL
6 While Q ≠ ∅
7   u ← ExtractMin(Q)
8     For each v ∈ Adj[u]
9       If v ∈ Q and c(u, v) < a(v)
10        Then parent(v) ← u
11             a(v) ← c(u, v); DecreaseKey(Q, v, a(v))
```

Le istruzioni di inizializzazione 1-3 prendono tempo $O(|V|)$. All'interno del **While** vengono esaminati (una sola volta!) tutti i vertici e gli archi incidenti su di essi. Vengono eseguite $|V|$ operazioni di `ExtractMin` (tempo $O(\log |V|)$ ciascuna se usiamo uno heap per implementare la coda a priorità), per un lavoro totale di $O(|V| \log |V|)$. Vengono eseguite $|E|$ operazioni di `DecreaseKey` (tempo $O(\log |V|)$ ciascuna) per un lavoro totale di $O(|E| \log |V|)$. Gran totale=$O(|E| \log |V|)$.

Vediamo un esempio