GRAFI: MINIMO ALBERO RICOPRENTE ALGORITMO DI PRIM

[Deme, seconda edizione] cap. 13

Sezione 13.3

Robert C. Prim

Da Wikipedia.

Robert Clay Prim (Sweetwater (Texas), 1921) è un matematico e informatico statunitense.

Nel 1941 si è diplomato in ingegneria elettronica presso l'Università di Princeton; nel 1949 ha conseguito il dottorato di ricerca in matematica. Ha lavorato all'Università di Princeton dal 1948 al 1949 come ricercatore associato.

Durante la seconda guerra mondiale, tra il 1941 e il 1944, ha lavorato come ingegnere per la General Electric; poi è stato assunto allo United States Naval Ordnance Lab come ingegnere, lavorandovi fino al 1949.

È stato direttore della ricerca matematica dei Bell Labs tra il 1958 e il 1961; in seguito è stato vicepresidente della ricerca presso i Sandia National Labs.

Scoperta dell’algoritmo di Prim

Durante la sua carriera presso i Bell Labs, **Robert Prim**, insieme a **Joseph Kruskal**, ha sviluppato due diversi algoritmi greedy per trovare un albero di copertura minimo di un grafo: **l'algoritmo di Kruskal** e **l'algoritmo di Prim**.

Quest'ultimo era stato originariamente scoperto nel 1930 dal matematico **Vojtech Jarník**, e solo nel 1957, in modo indipendente, da Prim (e nel 1959 fu riscoperto da Edsger Dijkstra), ed è a volte indicato anche come algoritmo di Jarník-Prim, o di Dijkstra-Jarník-Prim.

Osservazioni

Un **albero ricoprente** è un possibile **albero di visita**, quindi per trovare un minimo albero ricoprente si può adattare un **algoritmo di visita**.

Il problema è un problema di **minimo** (vogliamo trovare un algoritmo di visita tale che sia minima una caratteristica dell'albero di visita)

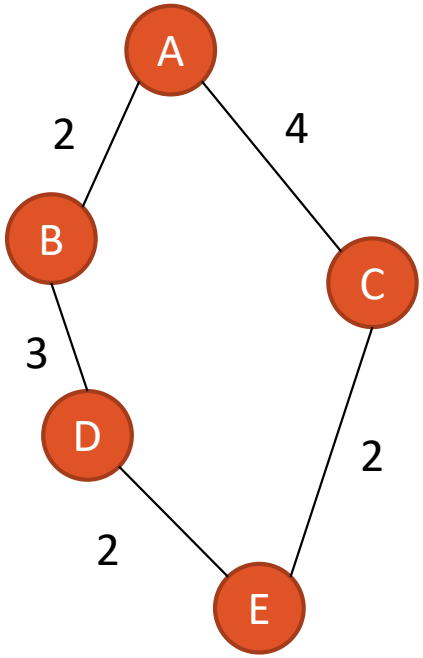
IDEA: anche **Dijkstra** affronta un problema di minimo, possiamo usarlo?

La risposta, in generale, è **no**. L'albero dei cammini minimi che viene generato dall'algoritmo di Dijkstra in generale **non è un MAR** perché nell'albero dei cammini minimi sono **minimi i cammini dal nodo di partenza a ciascun nodo**, cioè sono minimi i cammini dell'albero dalla radice a ciascun nodo, mentre nel MAR deve essere **minima la somma di tutti gli archi dell'albero**.

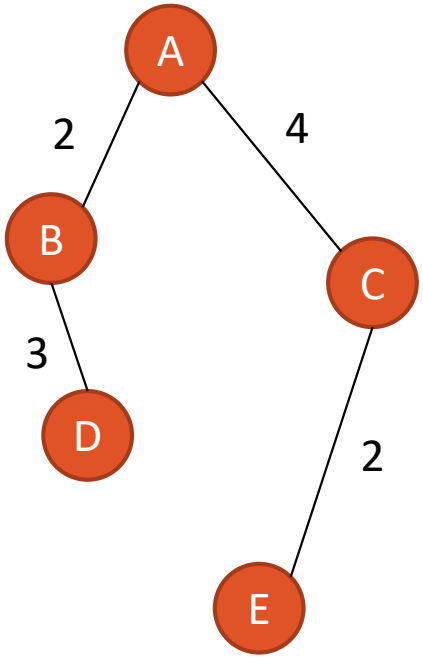
Tuttavia, possiamo **partire** da Dijkstra per costruire un algoritmo.

Esempio

Grafo originale

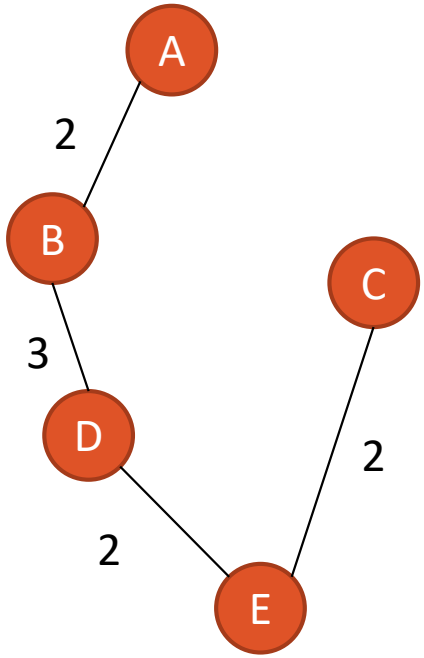


Albero T di visita
Dijkstra (s = A)



$W(A \to C) = 4 = \delta(A, C)$
$W(T) = 11$

Albero T' di visita
Prim (s = A)



$W(A \to C) = 9 \neq \delta(A, C)$
$W(T') = 9$

Idea di base

In **Dijkstra**, per trovare i **cammini minimi** dal nodo s a ciascun altro nodo, si prende ogni volta, fra tutti i cammini a nodi adiacenti a quelli già definitivi (cioè neri), **il cammino minimo da s**, e si aggiornano conseguentemente le altre distanze.

Nel caso dei MAR, poiché si cerca **la somma minima** dei (pesi dei) cammini e non i cammini minimi ai singoli nodi, si prende ogni volta **il cammino minimo non da s ma da un qualsiasi nodo nero**.

Cioè si cerca ogni volta **non il nodo più vicino alla radice**, ma **il nodo più vicino all'albero già costruito.** Quindi l'appetibilità (di un nodo grigio) è rappresentata dalla **stima della distanza del nodo dall'albero di visita**.

Una volta **aggiunto** un vertice, però, (come in Dijkstra) **l'appetibilità dei nodi grigi va aggiornata**.

Partiamo dall'algoritmo di Dijkstra

***Dijkstra* (G, W, s)**

```
INIZIALIZZA (G)
color [s] <- gray
d[s] <- 0
while esistono vertici grigi do begin
    u <- vertice grigio con d[u] minore
    S <- S ∪ {u}//aggiungo u all'albero definitivamente
    for ogni v adj ad u then
        if color[v] ≠ black then
            color [v] <- gray
            if d[v] > d[u] + W(u,v) then
                π[v] <- u
                d[v] <- d[u] + W(u,v)
    end for
    color [u] <- black
end
```

***INIZIALIZZA* (G)**

```
...
for ogni vertice u ∈ V[G] do
        color [u] <- white
        π[u] <- NULL
        d[u] <- +∞
```

Cosa cambia?

Partiamo dall'algoritmo di Dijkstra -II

***~~Dijkstra~~MinimoAlberoRicoprente* (G, W, s)**
INIZIALIZZA (G)
color [s] <- gray
d[s] <- 0
while esistono vertici grigi **do begin**
u <- vertice grigio con d[u] minore
S <- S ∪ {u}//aggiungo u all'albero definitivamente
for ogni v adj ad u **then**
if color[v] ≠ black **then**
color [v] <- gray
if d[v] > ~~d[u] +~~ W(u,v) then
π[v] <- u
d[v] <- ~~d[u] +~~ W(u,v)
end for
color [u] <- black
end

***INIZIALIZZA* (G)**
…
for ogni vertice u ∈ V[G] **do**
color [u] <- white
π[u] <- NULL
d[u] <- +∞

Di base, il nostro algoritmo non mantiene la «storia» passata: non considero il cammino da s al padre di v, ma solo la distanza tra v e suo padre, che è un vertice nero

Visto che sono simili, posso applicare le ottimizzazioni viste per Dijkstra?

MAR con priority queue

```
MinimoAlberoRicoprente (G, W, s)
   INIZIALIZZA (G)
   D <- empty_priority_queue()
   color [s] <- gray
   d[s] <- 0
   enqueue(D,s,d[s])
   while NotEmpty(D) do begin
      u <- dequeue_min(D)
      S <- S ∪ {u}//aggiungo u all'albero definitivamente
      for ogni v adj ad u then
         if color[v] ≠ black then
            if color[v] = white then
               color [v] <- gray
               enqueue(D,v,d[v])
            if d[v] > W(u,v) then
               π[v] <- u
               d[v] <- W(u,v)
               decrease_key(D,v,d[v])
      end for
      color [u] <- black
   end
```

Gestiamo, con una coda con priorità, i nodi grigi.

MAR con def/non def

```
MinimoAlberoRicoprente (G, W, s)
  INIZIALIZZA (G)
  D <- empty_priority_queue()
  d[s] <- 0
  for ogni v in V[G]
     enqueue(D,v,d[v])
     def[v] <- false
  while NotEmpty(D) do begin
     u <- dequeue_min(D)
     S <- S U {u} //aggiungo u all'albero definitivamente
     def[u] <- true
      for ogni v adj ad u then
        if def[v] = false and d[v] > W(u,v) then
              π[v] <- u
              d[v] <- W(u,v)
              decrease_key(D,v,d[v])
     end for
  end
```

Eliminiamo la distinzione bianchi/grigi/neri a favore di def/non def

Possiamo eliminare la gestione dei nodi definitivi (neri)?

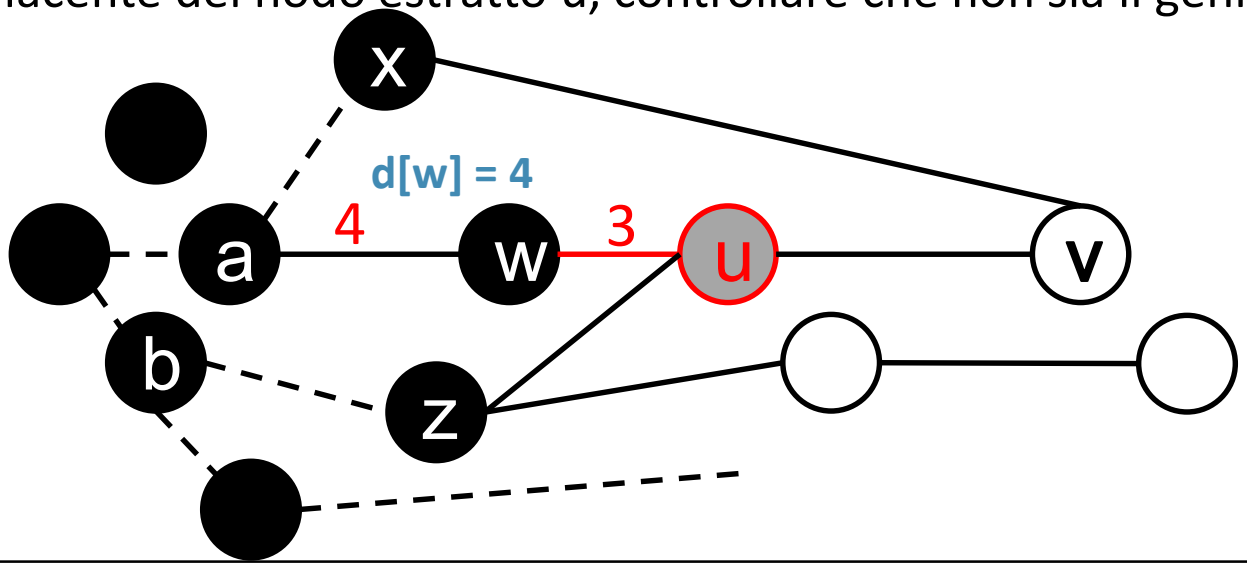
Eliminazione della gestione dei nodi neri

Possiamo eliminare la gestione dei nodi definitivi (neri)?

No. Il grafo è non orientato, quindi fra i nodi **adiacenti a u** vi è **il padre** w **di u**.

Allora può succedere che W(u,w) < d[w].

Ma w è un nodo **nero**, già **definitivamente** nel MAR come figlio di a, non può diventare figlio di u. Per evitarlo bisogna, nell'esaminare ogni adiacente del nodo estratto u, controllare che non sia il genitore di u.



Qual è la differenza con Dijkstra?

Differenza tra Dijkstra e Prim

In Dijkstra, possiamo sfruttare la proprietà per cui **d[v]**, con v nero, è **la distanza di v da s**. Essendo la distanza, **non esisterà nessun cammino il cui peso sarà minore di d[v]**, e quindi **d[v] non verrà mai più modificato**.

In Prim, questa proprietà non vale: **d[v] è il peso dell'arco minimo tra v e l'albero di visita** già costruito **nel momento in cui v viene estratto**, ma questo non vieta che in un secondo momento (aggiungendo un vertice u all'albero) possa esistere un arco da un vertice nero (u) a v di peso minore di d[v].

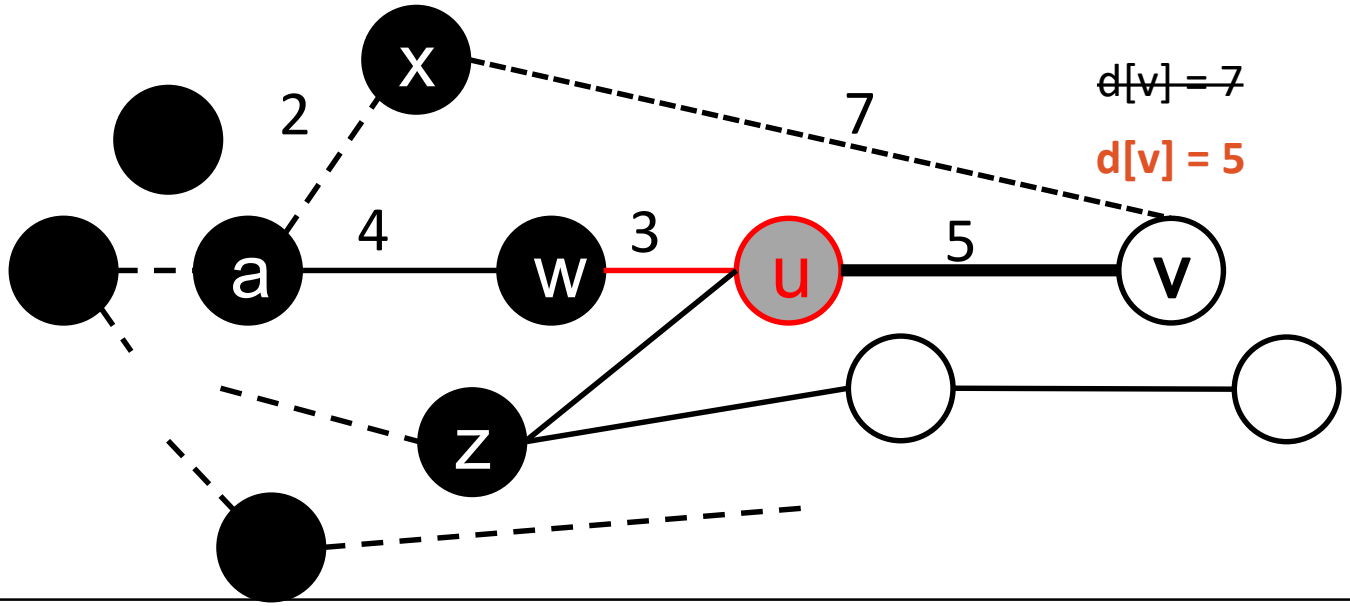
Questa è la motivazione per cui dobbiamo mantenere il controllo dei vertici definitivi.

Aggiornamento nodi non definitivi

L'**aggiornamento dell'appetibilità** dei nodi non definitivi rimane simile a Dijkstra, con l'accortezza che

d[v] <- W(u,v)

e non d[v] <- d[u] + W(u,v). Ad esempio, quando scelgo u, v è non definitivo.



Algoritmo di Prim

***~~MinimoAlberoRicoprente~~Prim* (G, W, s)**
INIZIALIZZA (G)
D <- empty_priority_queue()
d[s] <- 0
for ogni v in V[G]
enqueue(D,v,d[v])
def[v] <- false
while NotEmpty(D) **do begin**
u <- dequeue_min(D)
S <- S ∪ {u} //aggiungo u all'albero definitivamente
def[u] <- true
for ogni v adj ad u **then**
if def[v] = false **and** d[v] > W(u,v) **then**
π[v] <- u
d[v] <- W(u,v)
decrease_key(D,v,d[v])
end for
end

Complessità

La complessità dell'algoritmo di Prim è la stessa di quello di Dijkstra implementato con heap (algoritmo di Johnson). Quindi

Complessità algoritmo di Prim:

t_c è **O(n)**

t_e è **O(log n)**

t_d è **O(log n)**

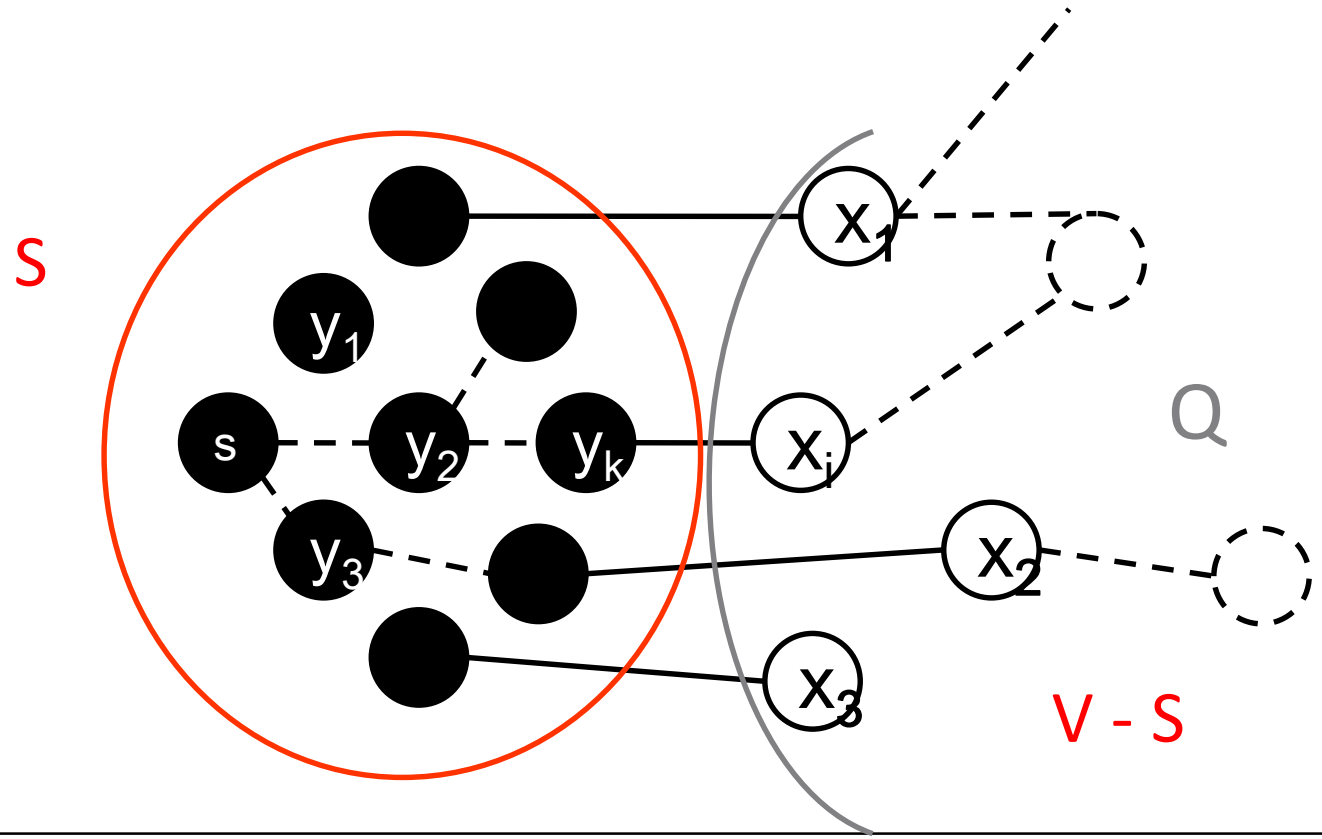
TOT **O((m+n) log n)**

Siccome il grafo è connesso, possiamo raffinare questa formula. Infatti sappiamo che **m ≥ n-1** quindi

O((m+n) log n) = O(m log n)

Correttezza

Sia S l'albero di visita costruito. **S** conterrà i **nodi definitivi** e **V-S** contiene i **nodi non definitivi**.

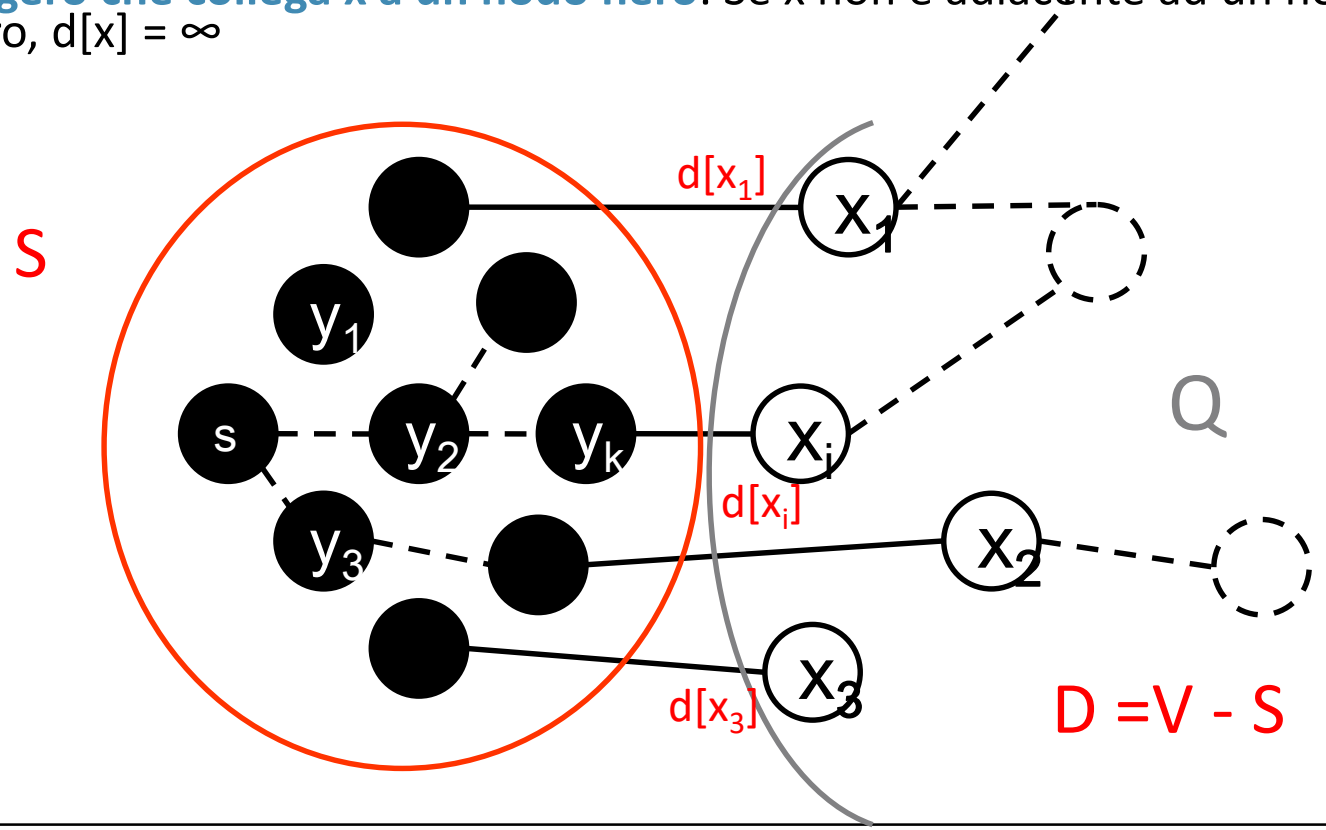


Invarianti

Definiamo gli invarianti

IS) Tutti gli archi dell'albero S **appartengono ad un qualche minimo albero ricoprente** dell'intero grafo G.

ID) Per ogni nodo x non definitivo (in D), d[x] è **il peso dell'arco (più) leggero che collega x a un nodo nero**. Se x non è adiacente ad un nodo nero, d[x] = ∞



Correttezza – base

Dopo la prima iterazione

c'è **un solo nodo nero/definitivo, s**

l'albero **S non contiene nessun arco**

ogni nodo **x adiacente a s** ha distanza **d[x]** uguale al **peso dell'arco (s, x)**

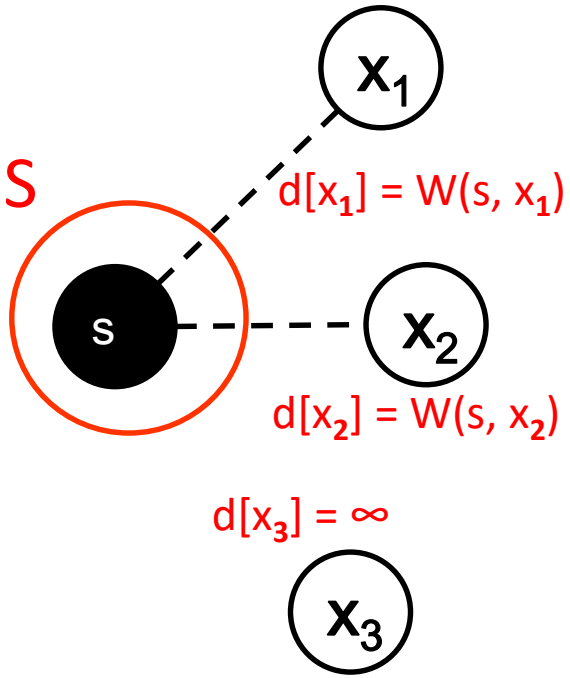
ogni altro nodo ha distanza uguale a **∞**.

Allora:

l'invariante **IS** è soddisfatto (in S **non ci sono archi**)

l'invariante **ID** è soddisfatto, perché essendoci un solo nodo nero, **l'arco che connette s ad un nodo adiacente x è l'unico** che unisce x a un nodo nero (quindi è il **minimo**), e gli altri nodi, non adiacenti ad s, hanno correttamente **d = ∞**.

Esempio base



IS: Il nodo s, **senza archi**, è certamente parte di un albero ricoprente.

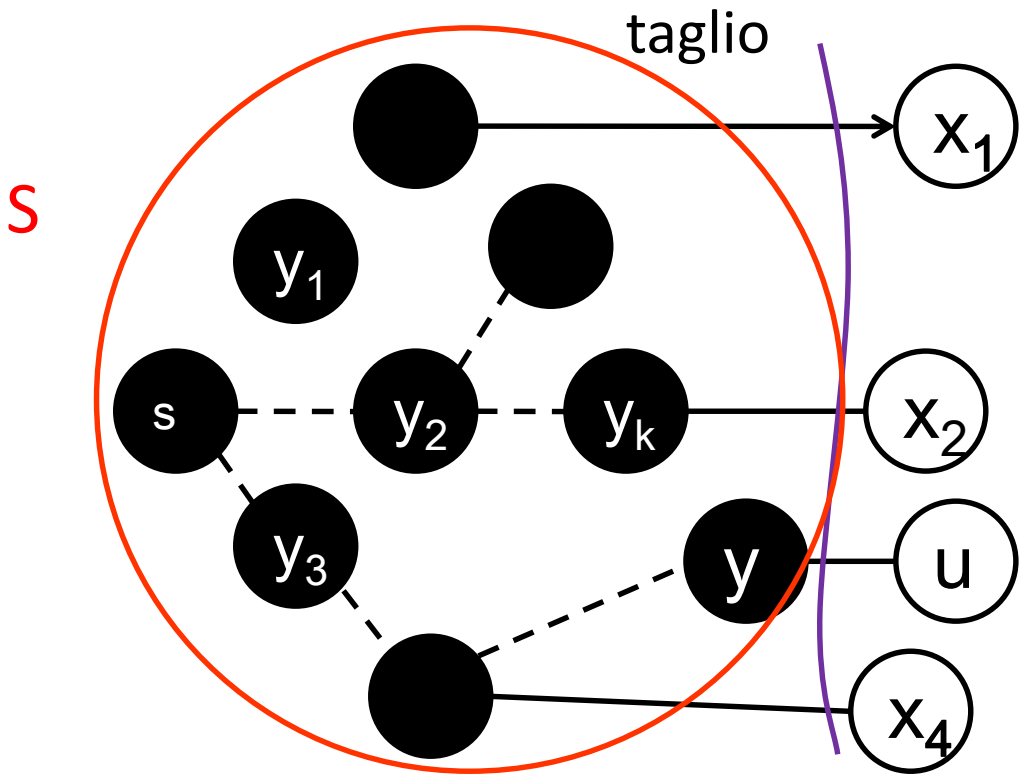
ID: L'arco (s, x_1), **se esiste**, è il più corto (**l'unico**) fra gli archi che connettono x_1 a un nodo nero.

Correttezza – passo

(**IS** e **ID** veri per ip. Induttiva)

Senza perdere di generalità, scegliamo (opportunamente) un taglio che **separi i nodi neri dagli altri**.

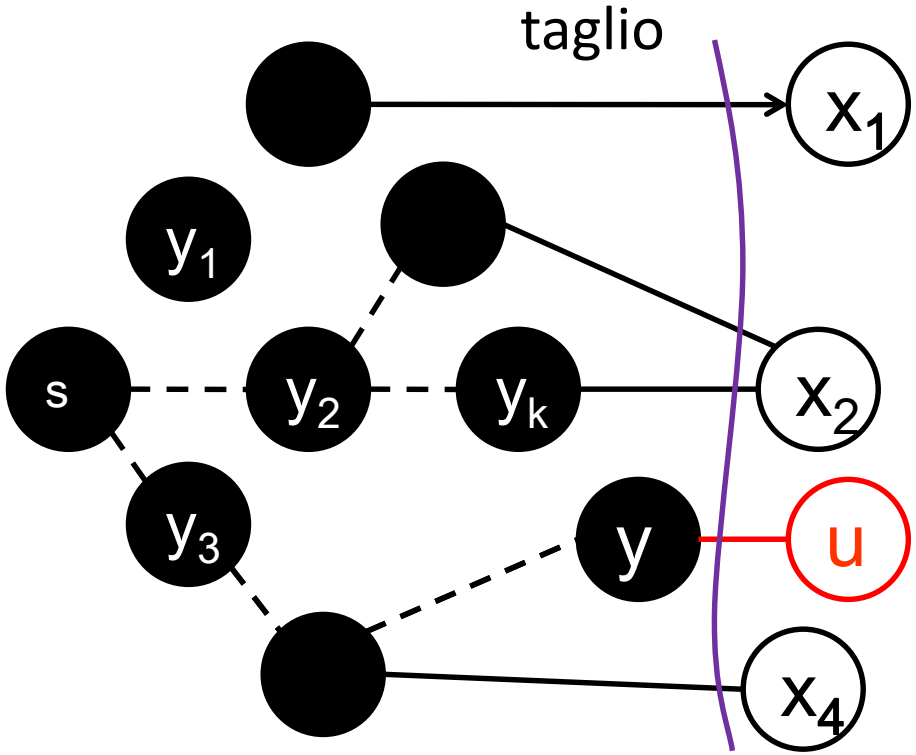
Sicuramente **non taglia nessun arco di S**.



Correttezza – passo II

Al passo i-esimo, nell'algoritmo, scegliamo **u** con **d[u] minore tra i nodi non neri**.

Allora l'arco **(π[u], u)** (che in questo caso è (y,u)) **esiste** ed **attraversa** il **taglio**, ed è quello di **peso minimo fra tutti gli archi che attraversano il taglio**.

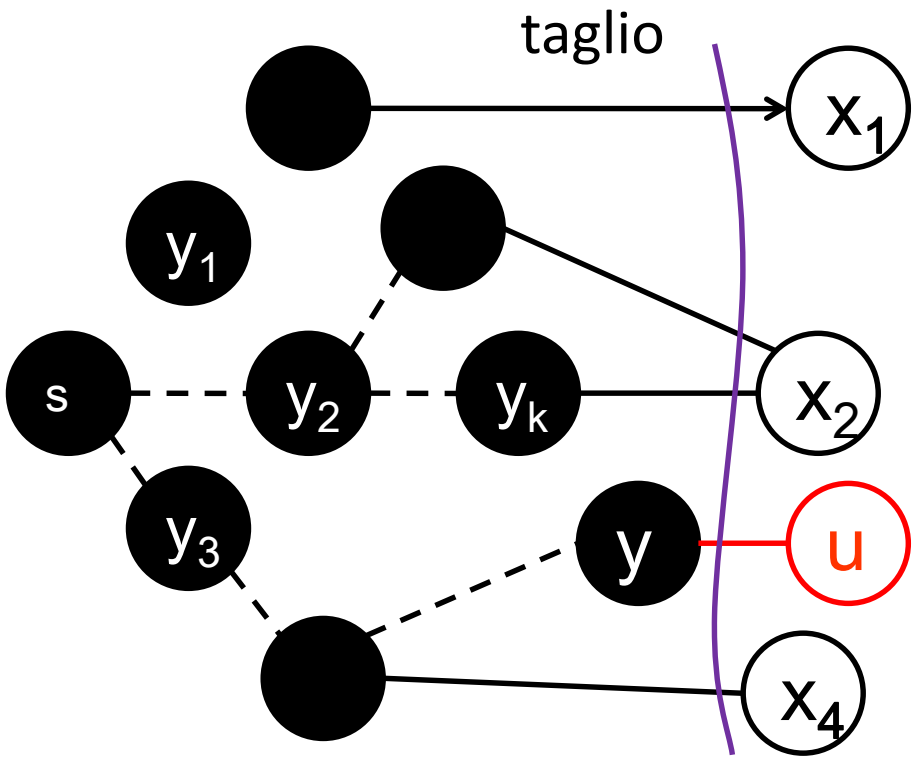


Domanda: come faccio a sapere che esiste sempre un arco (π[u], u) da poter scegliere?

Correttezza – passo III

S è un sottoinsieme di un MAR (per **ip. induttiva**).

Per il **Lemma del Taglio**, **(y,u) appartiene ad un MAR (di G) che estende S**.



Correttezza – passo IV

Quindi, aggiungendo l'arco **(y,u)** all'albero **S**, otteniamo ancora un **sottoinsieme di un qualche MAR di G**. quindi **IS si mantiene**.

Tuttavia, aggiungendo il nodo u ad S, l'invariante **ID non continua ad essere vero**.

Per **ripristinarlo**, devo considerare quali d (distanze minime tra un nodo v appartenente a D ed i nodi appartenenti ad S) possono essere cambiate.

Ma (per **ip. induttiva**) la condizione è già **vera per tutti i nodi neri in {S-u}**

Mi basta quindi guardare se il nuovo nodo **u «avvicina» qualche suo adiacente ad S**. Ma l'algoritmo lo fa, **mantenendo ID**.

```
…
for ogni v adj ad u then
        if def[v] = false and d[v] > W(u,v) then
                π[v] <- u
                d[v] <- W(u,v)
                decrease_key(D,v,d[v])
    end for
…
```

Correttezza – conclusione

Dalla dimostrazione, abbiamo che **IS è un invariante di ciclo**.

Siccome **G è connesso**, alla fine della visita **S conterrà tutti i nodi** (che saranno neri)

Quindi, alla fine della visita, (dato IS e la connessione) **S sarà un «sottoinsieme» di un MAR che contiene tutti i nodi di G**.

Quindi, **S è un Minimo Albero Ricoprente** di G.

Cosa devo aver capito fino ad ora

- Algoritmo di Prim
- Similitudini tra Prim e Dijkstra
- Differenze tra Prim e Dijkstra
- Complessità algoritmo di Prim
- Correttezza algoritmo di Prim

…se non ho capito qualcosa

- Alzo la mano e chiedo
- Ripasso sul libro
- Chiedo aiuto sul forum
- Chiedo o mando una mail al docente