Corso di laurea in Informatica
Progettazione d'algoritmi
Didattica blended

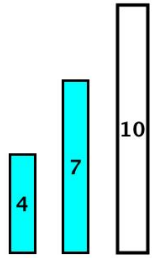
I grafi pesati

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Abbiamo tre contenitori di capienza 4, 7 e 10 litri. Inizialmente i contenitori da 4 e 7 litri sono pieni d'acqua e quello da 10 è vuoto.

Possiamo effettuare un solo tipo di operazione: versare acqua da un contenitore ad un altro fermandoci quando il contenitore sorgente è vuoto o quello destinazione pieno.

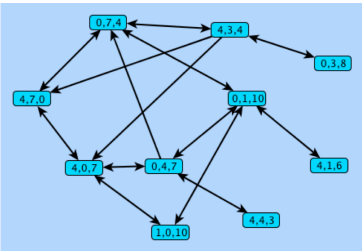


Problema:

Esiste una sequenza di operazioni di versamento che termina lasciando esattamente due litri d'acqua nel contenitore da 4 o nel contenitore da 7?

Posso modellare il problema con un grafo diretto G, i nodi di G sono i possibili stati di riempimento dei 3 contenitori (ogni nodo rappresenta una configurazione (a,b,c) dove a è il numero di litri nel contenitore da 4, b il numero di litri nel contenitore da 7 e c il numero di litri nel contenitore da 10). Metto un arco dal nodo (a,b,c) al nodo (a',b',c') se dallo stato (a,b,c) è possibile passare allo stato (a',b',c') con un versamento lecito.

Di seguito un frammento del grafo G:



n=440 infatti:

un contenitore con capienza z può contenere $0, 1, 2, \ldots z$ litri d'acqua, esso può assumere $z+1$ stati diversi. Quindi, il numero dei nodi è $5 \times 8 \times 11 = 440$ (questi comprendono anche nodi che dalla configurazione iniziale non possono essere raggiunti).

Per risolvere il nostro problema basterà chiedersi se nel grafo diretto G almeno uno dei nodi $(2,?,?)$ o $(?,2,?)$ è raggiungibile a partire dal nodo $(4,7,0)$

Abbiamo quindi ridotto il nostro problema ad un problema di raggiungibilità di nodi su grafi, problema che siamo in grado di risolvere in tempo $O(n+m)$ con una visita DFS o una visita BFS.

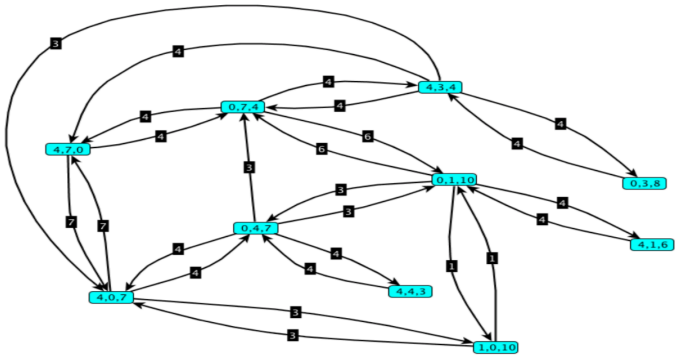
Se siamo interessati a raggiungere una delle configurazioni target $(2,?,?)$ o $(?,2,?)$ con il minor numero di travasi possiamo ancora risolvere il problema in $O(n+m)$ con una visita BFS per la ricerca dei cammini minimi calcolando le distanze minime a partire dal nodo $(4,7,0)$.

Consideriamo ora questa **variante del problema**:

Una sequenza di operazioni di versamento è buona se termina lasciando esattamente 2 litri o nel contenitore da 4 o nel contenitore da 7. Inoltre, diciamo che una sequenza buona è parsimoniosa se il totale dei litri versati in tutti i versamenti della sequenza è minimo rispetto a tutte le sequenze buone.

Vogliamo trovare una sequenza che sia buona e parsimoniosa.

Possiamo modellare il nuovo problema come segue:
Dovendo misurare il numero di litri d'acqua versati nelle varie mosse, conviene assegnare un costo ad ogni arco: il numero di litri che vengono versati nella mossa corrispondente. Si ottiene così un grafo di cui quello che segue è una piccola porzione:

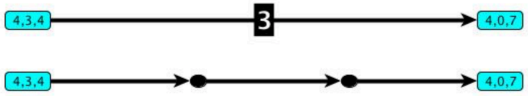


Il problema diventa ora quello di trovare un cammino dal nodo $(4,7,0)$ ad un nodo del tipo $(2,?,?)$ o $(?,2,?)$ che minimizza la somma sugli archi attraversati.

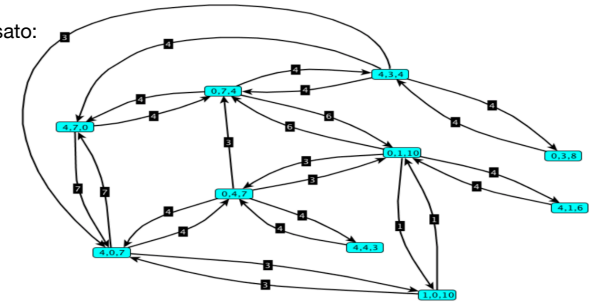
Il problema diventa ora quello di trovare un cammino dal nodo $(4,7,0)$ ad un nodo del tipo $(2,?,?)$ o $(?,2,?)$ che minimizza la somma dei costi degli archi attraversati

Questo problema è una generalizzazione del problema dei cammini di lunghezza minima perché gli archi invece di avere tutti lo stesso valore, cioè 1, hanno valori differenti.

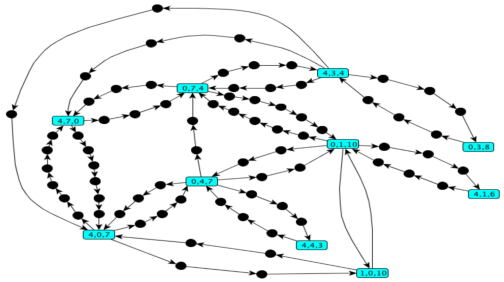
Idea: Possiamo sostituire un arco da x a y di costo C con un cammino con di $C-1$ nuovi nodi.

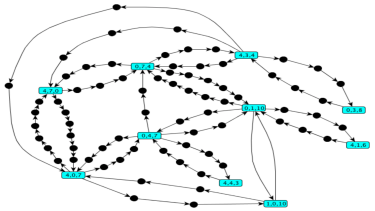


La porzione di grafo pesato:



Ecco come diventa la porzione di grafo dopo le sostituzioni:





Nel nuovo grafo tutti gli archi valgono 1. È come se ogni arco corrisponda al versamento di un litro d'acqua. Così contando semplicemente gli archi di un cammino tra due nodi configurazione contiamo proprio il numero totale di litri versati nelle mosse relative al cammino. Quindi per risolvere il problema possiamo fare una **BFS** nel nuovo grafo a partire dal nodo $(4,7,0)$ che si ferma non appena trova un nodo del tipo $(2,?,?)$ o $(?,2,?)$.

L'algoritmo avrà complessità $O(n'+m')$ dove n' ed m' sono i nodi e gli archi del nuovo grafo, rispettivamente.

Abbiamo quindi ricondotto un problema di cammini minimi su grafi pesati a quello dei cammini minimi in un grafo non pesato.

L' approccio di ricondursi al problema dei cammini minimi in cui tutti gli archi hanno lo stesso valore 1 è possibile quando le etichette degli archi sono **interi** relativamente **piccoli.**

Nel nostro caso il peso degli archi non poteva superare 7 si aveva quindi $n' < 7n$ ed $m' \leq 7m$ dove n ed m sono i nodi e gli archi del grafo pesato, rispettivamente.

L'esempio appena visto ha mostrato un caso in cui un problema viene naturalmente rappresentato tramite un grafo pesato (dove cioè gli archi hanno un valore numerico).

Problemi con questa caratteristica sono molti, basti ad esempio pensare al seguente:

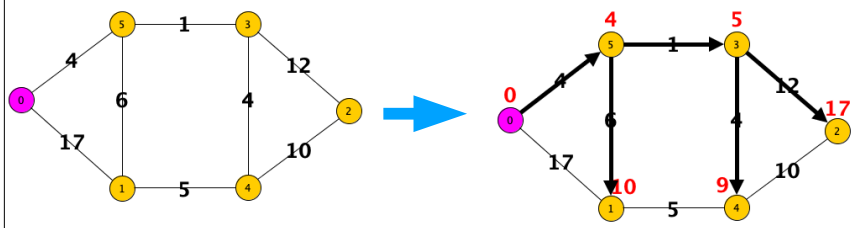
• Abbiamo una mappa stradale, vogliamo determinare il percorso più breve tra due località. I nodi del grafo sono le località e un arco da una località ad un altra avrà un costo pari alla lunghezza della strada che collega i due posti.

Per il problema della mappa stradale la trasformazione del grafo pesato in grafo non pesato potrebbe non essere possibile nel caso le distanze tra le località non fossero numeri interi e comunque sarebbe improponibile perché farebbe esplodere il numero di nodi e di archi del grafo.

Vedremo ora un algoritmo che permette di trovare i cammini minimi lavorando direttamente su grafi pesati (con pesi anche non interi) l'algoritmo è noto come algoritmo di Dijkstra.

Il problema dei cammini minimi in un grafo pesato e l'Algoritmo di Dijkstra

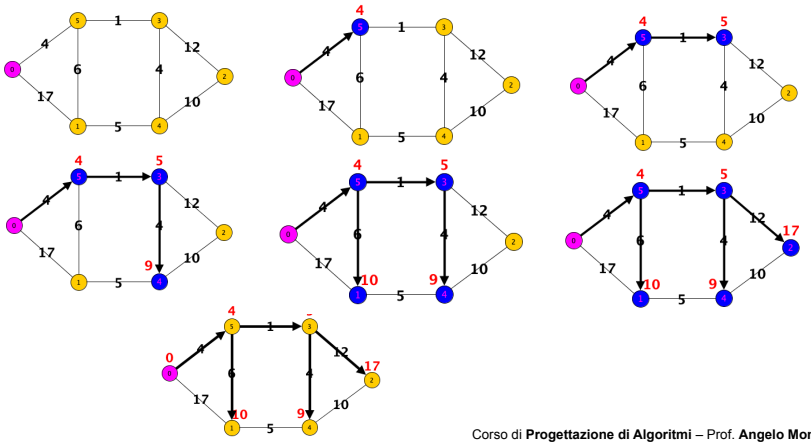
Problema: Dato un grafo pesato vogliamo trovare i cammini minimi e quindi anche le distanze da un certo nodo **s** (detto sorgente) a tutti gli altri nodi del grafo.



I cammini minimi (**in grassetto**) e le distanze (in rosso) quando la sorgente è il nodo 0.

Algoritmo di Dijkstra:

- **costruisci l'albero dei cammini minimi un arco per volta partendo dal nodo sorgente:**
  - **Ad ogni passo aggiungi all'albero l'arco che produce il nuovo cammino più economico.**
  - **Alla nuova destinazione assegna come distanza il costo del cammino.**



Algoritmo di Dijkstra:

- **costruisci l'albero dei cammini minimi un arco per volta partendo dal nodo sorgente:**
  - **Ad ogni passo aggiungi all'albero l'arco che produce il nuovo cammino più economico.**
  - **Alla nuova destinazione assegna come distanza il costo del cammino.**

Nota che l'algoritmo rientra perfettamente nel paradigma della tecnica greedy:

La sequenza di decisioni irrevocabili: decidi ad ogni passo il cammino (e quindi la distanza) dal nodo sorgente ad un nuovo nodo. Una volta deciso non ritornare più su questa decisione.

le decisioni vengono prese in base ad un criterio "locale": tra tutti i nuovi cammini che puoi trovare estendendo i vecchi di un arco prendi quello che costa meno.

Lo pseudo-codice dell'algoritmo di Dijkstra:

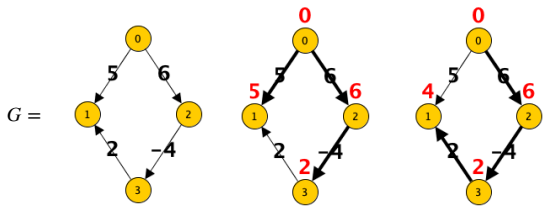
$Dijkstra(s, G)$:

- $P[0 \ldots n-1]$ vettore dei padri inizializzato a -1
- $D[0 \ldots n-1]$ vettore delle distanze inizializzato a $+\infty$
- $D[s],\ \ P[s] = 0,\ \ s$
- *while* esistono archi $\{x, y\}$ con $P[x] != -1$ e $P[y] == -1$:

  sia $\{x, y\}$ quello per cui è minimo $D[x] + peso(x, y)$
- $D[y],\ \ P[y] = D[x] + peso(x, y),\ \ x$

  return P, D

La prima cosa da far notare è che l'algoritmo non è corretto nel caso di grafi con pesi anche negativi:



Al centro la soluzione prodotta da Dijkstra per il grafo G e a destra la soluzione corretta.

$Dijkstra(s, G)$:

. $P[0 \ldots n-1]$ vettore dei padri inizializzato a -1

. $D[0 \ldots n-1]$ vettore delle distanze inizializzato a $+\infty$

. $D[s],\ \ P[s] = 0,\ \ s$

. *while* esistono archi $\{x, y\}$ con $P[x] != -1$ e $P[y] == -1$:

  sia $\{x, y\}$ quello per cui è minimo $D[x] + peso(x, y)$

. $D[y],\ \ P[y] = D[x] + peso(x, y),\ \ x$

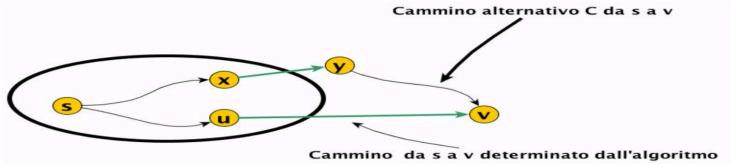
  return P, D

Correttezza nel caso di pesi positivi: Ad ogni iterazione del WHILE viene assegnata una nuova distanza ad un nodo. Per induzione sul numero di iterazioni mostreremo che la distanza assegnata è quella minima.

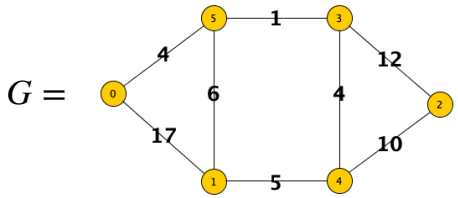
Prova per induzione della correttezza dell'algoritmo: Il caso base è banale (al passo 0 viene assegnata distanza zero alla sorgente e con pesi positivi non può esserci una distanza inferiore).

Sia T_i l'albero dei cammini minimi costruito fino al passo $i > 0$ e (u, v) l'arco aggiunto all'albero al passo $i+1$. Faremo vedere che $D[v]$ è la distanza mimima di v da s. Basterà mostrare che il costo di un eventuale cammino alternativo è sempre superiore o uguale a $D[v]$

- Sia C un qualsiasi cammino da s a v alternativo a quello presente nell'albero e (x, y) il primo arco che incontriamo percorrendo il cammino C all'indietro tale che x è nell'albero T_i e y no (tale arco deve esistere perchè s è in T_i mentre v no.)
- per ipotesi induttiva $costo(C) \geq Dist(x) + peso(x, y)$ (NOTA: quest'affermazione è vera perché i pesi del grafo son tutti non negativi)
- l'algoritmo ha preferito estendere l'albero T_i con l'arco (u, v) anzichè l'arco (x, y) e in base alla regola con cui l'algoritmo sceglie l'arco con cui estendere l'albero deve quindi aversi $D[x] + p(x, y) \geq D[u] + p(u, v)$
- da cui segue: $costo(C) \geq D[x] + peso(x, y) \geq D[u] + p(u, v) = D[v]$
- il cammino alternativo ha un costo superiore a $D[v]$.



Prima di passare all'implementazione dell'algoritmo di Dijkstra, due parole sulla rappresentazione di grafi pesati (con pesi sugli archi)



In un grafo pesato ogni arco ha associato un peso. Per rappresentare questo tipo di grafi per l'arco (x, y) di peso c nella lista di adiacenza di x invece che il solo nodo destinazione y ci sarà la coppia (y, c) con l'informazione sul nodo destinazione e il peso dell'arco.

Ad esempio il grafo pesato G in figura viene codificato come segue:

$$
\begin{aligned}
G = \{ & \\
& 0 : [(1, 17), (5, 4)], \\
& 1 : [(0, 17), (4, 5), (5, 6)], \\
& 2 : [(3, 12), (4, 10)], \\
& 3 : [(2, 12), (4, 4), (5, 1)], \\
& 4 : [(1, 5), (2, 10), (3, 4)], \\
& 5 : [(0, 4), (1, 6), (3, 1)] \\
\} &
\end{aligned}
$$

$Dijkstra(s, G)$:

. $P[0 \ldots n-1]$ vettore dei padri inizializzato a -1

. $D[0 \ldots n-1]$ vettore delle distanze inizializzato a $+\infty$

. $D[s],\ P[s] = 0,\ s$

. $while$ esistono archi $\{x, y\}$ con $P[x] ! = -1$ e $P[y] = = -1$:

. sia $\{x, y\}$ quello per cui è minimo $D[x] + peso(x, y)$

. $D[y],\ P[y] = D[x] + peso(x, y),\ x$

. $return\ P, D$

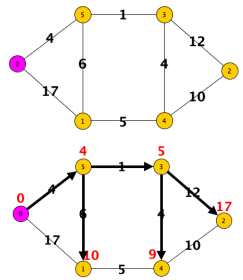
IDEA: nell'implementazione in ogni momento in D[x] è presente la distanza minima se x è già stato inserito nell'albero dei cammini minimi, il costo minimo che si ottiene se si vuole agganciare x all'albero, altrimenti.

```
def dijkstra(s, G):
    '''restituisce il vettore delle distanze ed l'albero dei
    cammini minimi rappresentato tramite vettore dei padri'''
    P=[-1 for  _  in G]
    from math import inf
    D=[inf for  _  in G]
    T=[ 0 for _ in G]
    D[s],P[s] = 0,s
    T[s]=1
    for y, costo in G[s] : D[y],P[y]= costo,s
    while True:
        lista=[(D[i], i) for i in range(len(D)) if T[i] ==0]
        if lista==[ ]: break
        mini, x = min(lista)
        if mini == inf : break
        T[x]=1
        for y, costo in G[x] :
            if D[x] + costo< D[y]:
                D[y]=D[x] + costo
                P[y]=x
    return D,P
```



```
def dijkstra(s, G):
    '''restituisce il vettore delle distanze ed l'albero dei
    cammini minimi rappresentato tramite vettore dei padri'''
    P=[-1 for  _  in G]
    from math import inf
    D=[inf for  _  in G]
    T=[ 0 for _ in G]
    D[s],P[s] = 0,s
    T[s]=1
    for y, costo in G[s] : D[y],P[y]= costo,s
    while True:
        lista=[(D[i], i) for i in range(len(D)) if T[i] ==0]
        if lista==[ ]: break
        mini, x = min(lista)
        if mini == inf : break
        T[x]=1
        for y, costo in G[x] :
            if D[x] + costo< D[y]:
                D[y]=D[x] + costo
                P[y]=x
    return D,P
```

G=



```
G={
0:[(1,17),(5,4)],
1:[(0,17),(4,5),(5,6)],
2:[(3,12),(4,10)],
3:[(2,12),(4,4),(5,1)],
4:[(1,5),(2,10),(3,4)],
5:[(0,4),(1,6),(3,1)],
}
>>> dijkstra(0,G)
([0, 10, 17, 5, 9, 4], [0, 5, 3, 5, 3, 0])
```

$$\begin{array}{lcccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ D = & [0, & 10, & 17, & 5, & 9, & 4] \\ P = & [0, & 5, & 3, & 5, & 3, & 0] \end{array}$$


```
def dijkstra(s, G):
    '''restituisce il vettore delle distanze ed l'albero dei
    cammini minimi rappresentato tramite vettore dei padri'''
    P=[-1 for  _  in G]
    from math import inf
    D=[inf for  _  in G]
    T=[ 0 for _ in G]
    D[s],P[s] = 0,s
    T[s]=1
    for y, costo in G[s] : D[y],P[y]= costo,s
    while True:
        lista=[(D[i], i) for i in range(len(D)) if T[i] ==0]
        if lista==[ ]: break
        mini, x = min(lista)
        if mini == inf : break
        T[x]=1
        for y, costo in G[x] :
            if D[x] + costo< D[y]:
                D[y]=D[x] + costo
                P[y]=x
    return D,P
```

Complessità dell'implementazione:

Il costo totale dell'esecuzione delle istruzioni al di fuori del $while$ è $O(n)$ (nota che i nodi adiacenti a s sono $O(n)$). Abbiamo poi un $while$ con all'interno una $list\ \ comprehention$, una funzione min ed un for. Il contributo del for lo calcoliamo a parte.

- $list\ \ comprehention$ costa $\Theta(n)$, il min costa $O(n)$. Il $while$ viene eseguito $O(n)$ volte. Quindi il costo totale dell'algoritmo (escluso il costo delle varie iterazioni del for è $O(n^2)$.
- ad ogni iterazione del $while$ col for si scorre la lista di adiacenza di un nodo, ogni lista può essere scorsa al più una volta quindi in totale il tempo richiesto dal for sarà $O(m)$.

La complessità di questa implementazione è dunque $O(n) + O(n^2) + O(m) = O\left(n^2\right)$

Nota: questa implementazione è ottima nel caso di grafi densi dove $m = \Theta(n^2)$


```
def dijkstra(s, G):
    '''restituisce il vettore delle distanze ed l'albero dei
    cammini minimi rappresentato tramite vettore dei padri'''
    P=[-1 for  _  in G]
    from math import inf
    D=[inf for  _  in G]
    T=[ 0 for _ in G]
    D[s],P[s] = 0,s
    T[s]=1
    for y, costo in G[s] : D[y],P[y]= costo,s
    while True:
        lista=[(D[i], i) for i in range(len(D)) if T[i] ==0]
        if lista==[ ]: break
        mini, x = min(lista)
        if mini == inf : break
        T[x]=1
        for y, costo in G[x] :
            if D[x] + costo< D[y]:
                D[y]=D[x] + costo
                P[y]=x
    return D,P
```

Nota che se si potesse evitare di scorrere ogni volta il vettore D alla ricerca della posizione in cui è presente il minimo potrei evitare di pagare $\Theta(n)$ ad ogni iterazione del $while$.

IDEA: per ogni nodo x del grafo per cui c'è arco (p, x)con p nell'albero e x non nell'albero, tenere in un heap minimo la tupla $(D[p] + costo(p, x),\ x,\ p)$ (indicizzata rispetto alla prima componente). In questo modo ad ogni step possiamo scoprire in tempo logaritmico nella dimensione dell'heap il nodo x da inserire nell'albero, il costo $D[p] + costo(p, x)$ da assegnargli e il padre p a cui agganciarlo.

Ovviamente ogni volta che aggiungo un nodo x all'albero dovrò aggiornare anche l'heap inserendovi nuovi valori $(D[x] + costo(x, y),\ y,\ x)$ per ogni vicino y di x(ricorda che ogni inserimento ha costo logaritmico nella dimensione dell' heap)

In questa seconda implementazione utilizzeremo quindi la struttura dati heap

In python abbiamo un modulo *heapq* in cui troviamo le funzioni per creare, inserire un elemento nell'hep, estrarre un elemento dall'heap ecc.ecc., a noi bastano queste due funzioni:

- $heappop(h)$: estrae e restituisce il minimo dall'heap h in tempo $O(\log(len(h)))$
- $heappush(h, x)$: inserisce l'elemento x nell'heap h in tempo $O(\log(len(h)))$

```
def dijkstra1(s, G):
    '''restituisce il vettore delle distanze ed l'albero dei
    cammini minimi rappresentato tramite vettore dei padri'''
    from heapq import heappop, heappush
    P=[-1 for  _  in G]
    from math import inf
    D=[inf for  _  in G]
    D[s],P[s] = 0,s
    H=[]
    for y, costo in G[s] :
        heappush(H, (costo, y, s))
    while H:
        costo,x,padre = heappop(H)
        if P[x]==-1:
            P[x]= padre
            D[x]= costo
            for y, costo1 in G[x]:
                    heappush(H, (D[x] + costo1, y, x))
    return D, P
```



```
def dijkstra1(s, G):
    '''restituisce il vettore delle distanze ed l'albero dei
    cammini minimi rappresentato tramite vettore dei padri'''
    from heapq import heappop, heappush
    P=[-1 for  _  in G]
    from math import inf
    D=[inf for  _  in G]
    D[s],P[s] = 0,s
    H=[]
    for y, costo in G[s] :
        heappush(H, (costo, y, s))
    while H:
        costo,x,padre = heappop(H)
        if P[x]==-1:
            P[x]= padre
            D[x]= costo
            for y, costo1 in G[x]:
                    heappush(H, (D[x] + costo1, y, x))
    return D, P
```


Nota che in H possono esserci anche $O(m)$ elementi e quindi i costi di inserimento ed estrazione saranno $O(\log m) = O(\log n^2) = O(\log n)$.

Complessità dell'implementazione:
Prima dell'accesso al *while* abbiamo l'inizializzazione di D e P (per un costo $O(n)$) e l'inserimento nell' heap H di $O(n)$ elementi (per un costo $O(n \log n)$). Abbiamo un *while* con all'interno un *for*. Il contributo del *for* lo calcoliamo a parte.

- ad ogni iterazione del *while* si elimina un elemento da H e *eventualmente* per mezzo del *for* annidato si scorre la lista di adiacenza di un nodo e vengono aggiunti elementi ad H. Ogni lista di adiacenza può essere scorsa al più una volta quindi ad H in totale possono essere aggiunti al più $O(m)$ elementi. Per quanto detto il numero di iterazioni del *while* è $O(m)$.
- Senza tener conto del *for* annidato (i cui costi verranno calcolati a parte al prossimo punto), il costo di ciascuna iterazione del *while* richiede $O(\log n)$ a causa dell'estrazione da H. Quindi il costo totale del *while*, senza tener conto del *for*, sarà $O(m \log m)$
- Il tempo totale richiesto dalle varie iterazioni del *for* annidato nel *while* è $O(m \log n)$ in quando ad ogni iterazione scorro una lista di adiacenza diversa. In totale scorrerò $O(m)$ archi e per ogni arco pagherò $O(\log n)$ per inserimento in H.

La complessità di questa implementazione è dunque

$$O(n \log n) + O(m \log n) + O(m \log n) = O((n + m) \log n).$$

Abbiamo visto due implementazioni dell'algoritmo di Dijkstra:

- la prima richiede $O(n^2)$ tempo, la seconda $O((n + m) \log\ n)$.
  - La prima è ottimale nel caso di grafi densi.
  - La seconda è da preferirsi nel caso di grafi sparsi, presentando in quel caso una complessità $O(n \log n)$ mentre andrebbe evitata nel caso di grafi densi dove la complessità risultante sarebbe $O(n^2 \log n)$

- Esistono implementazioni più efficienti di quest'algoritmo ottenute utilizzando strutture dati più sofisticate.
  Ad esempio utilizzando gli heap di Fibonacci il tempo d'esecuzione scende a $O(m + n \log n)$.