L'algoritmo di Edsger Dijkstra

Luca Cataldo

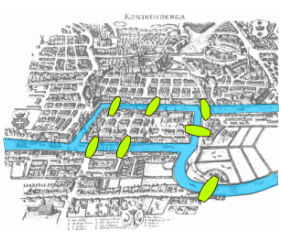
28 Luglio 2020

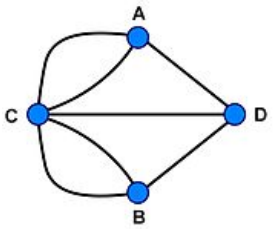
Indice

1 Introduzione

L'algoritmo di Dijkstra o comunemente chiamato anche Dijkstra Shortest Path First, è un algoritmo che viene usato per trovare i cammini minimi tra due nodi all'interno di un grafo. La teoria dei grafi trova moltissime applicazioni. Basti pensare ai Social Network come Facebook oppure Twitter.

La teoria dei grafi possiamo dire che ha una sua data di nascita ben precisa. Correva l'anno 1736, in quel periodo il matematico svizzero **Leonhard Euler**, noto a noi italiani come **Eulero**, risolse il problema dei **sette ponti di Königsberg**. In cosa consisteva questo problema? Tale quesito riguardava la possibilità di compiere un percorso attraversando tutti i ponti della città, passando per ognuno di essi una e una sola volta. Cosi a primo impatto qualcuno potrebbe domandarsi: ma qual'è la relazione tra questa domanda e la teoria dei grafi? La risposta è molto semplice e può essere visualizzata molto facilemente con la disposizione fatta da Eulero. Egli prese la mappa della città, da essa eliminò tutte le parti non indispensabili. In un secondo momento rimpiazzò ogni area urbana con un punto detto nodo, e ogni ponte con un arco.





Con questa visualizzazione Eulero nel 1736 affrontò il problema e dimostrò che una cosa del genere non sarebbe stata possibile ovvero, non era possibile fare questo percorso, passando da tutti i ponti una e una sola volta. Spostando la lancetta del tempo in avanti di circa 220 anni, arriviamo al 1956 ovvero l'anno in cui Dijkstra ideò il suo algoritmo, che venne pubblicato tre anni dopo nel 1959.

2 L'algoritmo di Dijkstra

Gli algoritmi di ricerca dei percorsi sono veramente molto importanti. Li utilizziamo in svariati campi della ricerca, basti pensare per esempio a Google Maps. Capire dove ci troviamo in un determianto istante di tempo è la prima parte del nostro problema.

La seconda domanda a cui dobbiamo trovare risposta è la seguente: qual'è la strada migliore per raggiungere la mia destinazione? A tal proposito ci viene in aiuto l'algoritmo di Dijkstra. In verità esiste una intera classe di algoritmi appositi per la risoluzione di questo problema, come per esempio l'algoritmo A* che però non tratteremo in questa sezione. L'algoritmo di Dijkstra ha molteplici

applicazioni, e si pone il problema di cercare i cammini minimi all'interno di un grafo pesato, nel caso in cui tutti i pesi degli archi non siano negativi.

2.1 Tempo di esecuzione

La complessità dell'algoritmo di Dijkstra la possiamo esprimere in funzione di |V| ed |E|, ovvero il numero di nodi e il numero degli archi. Questo algoritmo implementa una coda di priorità, la cui struttura dati utilizzata per implementarla, andrà a determinare la complessità delle 3 operazioni che verranno eseguite su di essa, e la complesità dell'algoritmo di conseguenza. Le operazioni sono le seguenti:

1. La costruzione della coda.
2. L'estrazione dell'elemento minimo.
3. E il decremento del valore di un elemento

Nella sua implementazione più semplice l'algoritmo di Dijkstra avrà una certa complessità. In informatica, per indicare la complessità spaziale o temporale di un algoritmo al caso pessimo, utilizziamo la notazione matematica degli O-grande $\mathcal{O}$.

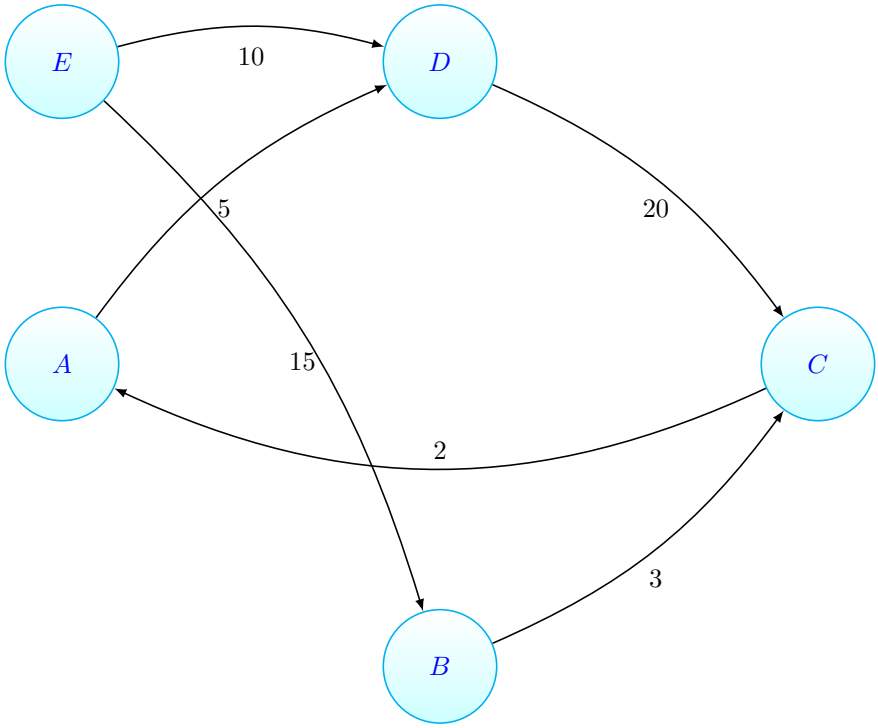
L'algoritmo di Dijkstra nella sua forma più semplice ha una complessità spaziale pari a: $\mathcal{O}(|V|)$, ovvero uguale alla cardinalità di V, cioè i nostri vertici, i nodi del grafo.

Invece dal punto di vista del tempo di esecuzione dell'intero algoritmo, la sua complessità è: $\mathcal{O}(|E| + |V|^2)$, dove |E| è il nostro insieme degli archi.

Invece se l'algoritmo fosse implementato facendo uso di struttre dati differenti come **Binary Heap** o **alberi di ricerca**, le due formule appena citate, andrebbero a cambiare ulteriormente.

3 Esercizio

Esempio 1.0. In questo esempio possiamo notare un grafo avente il seguente insieme dei nodi U=(A,B,C,D,E) e i rispettivi collegamenti:(A,D),(B,C),(C,A),(D,C),(E,B),(E,D).



Applichiamo adesso l'algoritmo di Dijkstra per trovare l'albero dei cammini minimi.

Per prima cosa andiamo a scegliere i due nodi di cui vogliamo calcolare il percorso minimo.

Un nodo **Sorgente**, per esempio il nodo E, e un nodo **Destinazione**, per esempio il nodo A, e vediamo quale è il cammino minimo che collega questi due nodi.

Per capire il funzionamento useremo una tabella nella quale saranno indicati tutti i valori che ci serviranno per trovare il cammino minimo. La tabella sarà composta da 3 campi:

1. Il campo **Nodi**, che contiene tutti i nodi che fanno parte del grafo
2. Il campo **Distanza**, il quale inizialmente sarà settato a 0 per la sorgente e a infinito per tutti gli atri nodi.
3. Il campo **Predecessori**, dove saranno salvati i predecessori dei nodi.

Nello stato iniziale la tabella dei valori si presenta cosi:

Nodi	Distanza	Predecessori
E	0	-
B	∞	-
D	∞	-
C	∞	-
A	∞	-

Il primo step è quello di andare a visitare i vicini del nodo che stiamo processando sul momento, in questo caso il nodo in questione è il nodo E, e quindi andiamo dal primo vicino di E ovvero B. Notiamo grazie al grafico **dell'esempio 1.0** che andare da E a B ha costo 15. Essendo che il costo della sorgente è 0, il costo complessivo per andare da E a B ci costa in totale 15, cioè costo di E + costo di B, ergo 0+15 = 15, e di conseguenza andiamo ad aggiornare la tabella. Continuando con la stessa idea, andiamo a vedere adesso l'altro vicino di E, ovvero D, che ha costo 10, quindi per andare da E a D il costo complessivo è 10, ovvero 0+10 =10.

Nodi	Distanza	Predecessori
E	0	-
B	15	-
D	10	-
C	∞	-
A	∞	-

Notiamo adesso che la distanza di B è maggiore della distanza di D, nel nostro caso andiamo a disporre i nodi in ordine di distanza, e quindi andremo a cambiare l'ordine dei due nodi B e D. Un'altra cosa importantissima che non dobbiamo trascurare è il campo dei predecessori, perchè grazie ad essi possiamo ricostruire tutto il percorso a ritroso partendo dal nodo destinazione, e ottenere cosi il percorso minimo. Andiamo quindi ad aggiornare il campo predecessori. In questo caso entrambi i nodi B e D avranno lo stesso predecessore E, aggiorniamo di conseguenza tale campo.

Arrivati alla fine di questo primo step, notiamo che dal nodo E non possiamo andare più da nessun'altra parte, e quindi con E abbiamo finito e possiamo quindi toglierlo dalla tabella, e metterlo nella **coda di scarto**. Tale coda è una lista di coppie, dove ogni coppia è formata da (Nodo,Predecessore).

Nodi	Distanza	Predecessori
D	10	E
B	15	E
C	∞	-
A	∞	-

```
Coda di scarto = [(E,-)]
```

Adesso continuando cosi passiamo al secondo step ovvero, andiamo a vedere tutti i nodi che possiamo raggiungere da D, ovvero il nodo che sto processando al momento. Sempre prendendo come riferimento il grafo **dell'esempio 1.0** notiamo che da D possiamo raggiungere il nodo C. Il costo per raggiungere il nodo C è: costo di D + costo di C, ergo avremmo 10+20=30. E il predecessore di C è a sua volta D. Aggiorniamo adesso la tabella.

A questo punto dal nodo D non possiamo raggiungere nessun'altro nodo, quindi con D abbiamo finito e lo andiamo a togliere dalla tabella e lo inseriamo nella **coda di scarto**.

Nodi	Distanza	Predecessori
B	15	E
C	30	D
A	∞	-

`Coda di scarto = [(E,-),(D,E)]`

Come possiamo notare la distanza di B è minore della distanza di C, quindi non abbiamo bisogno di riordinare la tabella e procediamo di conseguenza. Andiamo a valutare il nodo B. L'unico nodo che B può raggiungere è C, e la disatzna per farlo è 15+3=18. A questo punto notiamo che però C ha gia un suo valore all'interno della tabella, ma visto che ho trovato una distanza più piccola per raggiungere il nodo C vado ad aggiornare tale valore nella tabella. Vado anche ad aggiornare il predecessore di C, perchè visto che il nuovo cammino per C è cambiato, cambia a sua volta anche il predecessore. Il nuovo predecessore di C è B. A questo punto non ho altri nodi che posso raggiungere da B, ergo con B abbiamo finito, lo tolgo dalla tabella e lo inserisco nella **coda di scarto**.

Nodi	Distanza	Predecessori
C	18	B
A	∞	-

`Coda di scarto = [(E,-),(D,E),(B,E)]`

Passiamo al prossimo nodo ovvero C. Da esso possiamo raggiungere il nodo A. Il costo per farlo sarà: costo di C + costo di A, cioè 18+2=20. Il predecessore di A in questo caso sarà C. Da C non possiamo più raggiungere nessun'altro nodo, quindi lo togliamo dalla tabella e lo inseriamo nella **coda di scarto**. Aggiorniamo di conseguenza la tabella.

Nodi	Distanza	Predecessori
A	20	C

`Coda di scarto = [(E,-),(D,E),(B,E),(C,B)]`

Siamo arrivati allo step finale. Notiamo che abbiamo raggiunto il nodo A ovvero il nodo destinazione. A questo punto per visualizzare il camminimo minimo tra E e A, usiamo la **coda di scarto**, la quale avrà un ruolo fondamentale in questa fase finale.

Nodi	Distanza	Predecessori

`Coda di scarto = [(E,-),(D,E),(B,E),(C,B),(A,C)]`

Per ricostruire il percorso a ritroso, usiamo la **coda di scarto**. Partiamo da A. Avremmo una lista del tipo: `Percorso minimo[A]`.

Il predecessore di A è C, e andiamo ad inserirlo nella lista: `Percorso minimo[A,C]`.

Il predecessore di C a sua volta è B, e andiamo ad inserire pure lui nella lista: `Percorso minimo[A,C,B]`.

Continuiamo così, il predecessore di B è E, e lo inseriamo nella lista:

`Percorso minimo[A,C,B,E]`.

E infine il predecessore di E non esiste perchè E sarebbe la sorgente, ergo abbiamo ricostruito a ritroso il percorso da A a E, e quindi il nostro percorso minimo dal nodo sorgente al nodo destinazione sarà:

`Percorso minimo[E,B,C,A]`.

Alla fine abbiamo trovato il percorso più breve, l'unico percorso più breve possibile. Grazie alla tabella e alla coda di scarto siamo riusciti a trovare in maniera inversa partendo dal nodo destinazione il percorso con la distanza minima. Come possiamo notare l'utimo percorso che abbiamo trovato aveva una forma diversa dal precedente ovvero:

l'ultimo percorso che abbiamo trovato era:

`Percorso minimo[A,C,B,E]`

ma visto che noi vogliamo un percorso che vada dal nodo sorgente al nodo destinazione dobbiamo leggere questo percorso nella direzione opposta, quindi avremmo che il nostro percoso minimo sarà:

`Percorso minimo[E,B,C,A]`.

4 Fase di Testing

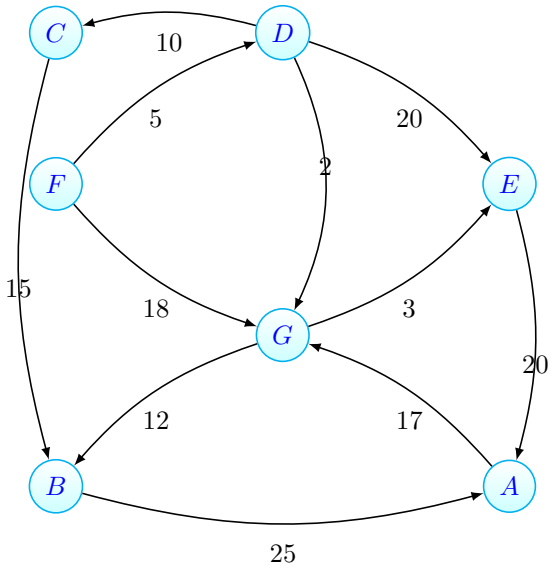
Andiamo a vedere come si comporta l'algoritmo quando si trova davanti vari tipi di grafo.

Andiamo quindi ad analizzare vari casi:

1. I casi tipici
2. I casi atipici
3. I casi limite

4.1 Esempio 1.1.

Prediamo questo primo esempio di grafo con l'insieme dei nodi U=(A,B,C,D,E,F,G,),
e con i seguenti collegamenti (A,G),(B,A),(C,B),(C,D),(D,E),(D,G),(E,A),(F,D),(F,G),(G,E),(G,B)



Questo è un caso tipico di grafo orientato.

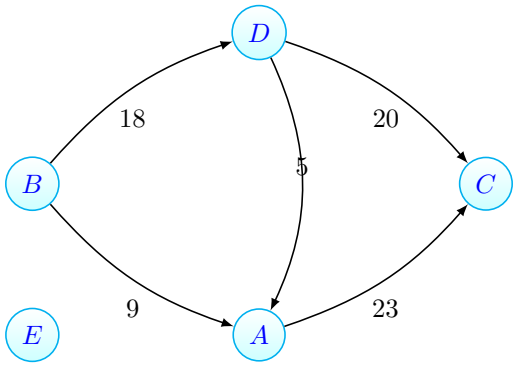
Se andiamo a scegliere come nodo **Sorgente** D e come nodo **Destinazione** A, notiamo che ci sono vari cammini che portano dal nodo D e arrivano al nodo A. In questo caso l'algoritmo deve analizzare bene i cammini e scegliere quello più corto. Quindi dando in input questo grafo al programma, l'ouput che ne deriva è il seguente:

```
Tempo di esecuzione --- 0.000 secondi ---
Il cammino minimo è
 (25, ['D', 'G', 'E', 'A'])
```

Come possiamo notare con una quantità cosi piccola di nodi, l'algoritmo esegue il tutto praticamente in modo instantaneo. Infatti il tempo di esecuzione è 0,000 secondi. Facendo varie esecuzioni ho notato che il tempo di esecuzione oscilla tra i 0,00 - 0,01 secondi. Un tempo relativamente molto basso.

4.2 Esempio 1.2.

Prendiamo invece in consiederazione questo caso descritto dal seguente grafo.



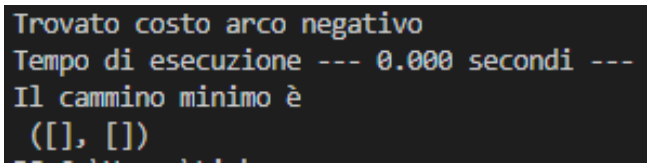
In questo esempio specifico, se andiamo a scegliere come nodo **Sorgente** C, e come nodo **Destinazione** E, notiamo subito che non esiste un cammino che porta da C ad E. Il cammino non esiste perchè E è un nodo isolato dal resto del grafo. In questo caso dando in input tale grafo al programma, l'output risultante sarà il seguente:

```
Tempo di esecuzione --- 0.000 secondi ---
Cammino non presente
Il cammino minimo è
 ([], [])
```

Come possiamo notare anche in questo caso, con un grafo avente solo 5 nodi, il tempo di esecuzione è pressoché istantaneo. Possiamo notare anche come il risultato dell'esecuzione mi dica che il cammino non è presente, infatti le due liste risultanti sono vuote.

4.3 Esempio 1.3.

Un esempio di caso limite invece è quando il grafo ha al suo interno un arco che ha un peso negativo. In questa situazione, l'algoritmo non è sicuro che riesca a trovare un cammino minimo. In questa implementazione infatti se dovesse capitare una cosa del genere,il programma restituirebbe il seguente output:



Come vediamo, l'ouput ci avverte che l'algoritmo ha trovato un arco negativo, ergo ha fermato l'esecuzione del programma, e di conseguenza il cammino minimo tra i due nodi non esiste. Per quanto riguarda il tempo di esecuzione, anche in questo caso è istantaneo, visto la poca quantita di archi che ha il grafo. Facendo varie prove su questo esempio, il tempo di esecuzione oscilla tra i 0.00-0.001 secondi

5 Stress Test

Nella sezione precedente abbiamo visto come si comporta l'algoritmo con vari casi di grafo che si posso presentare.

Adesso invece andiamo a testare l'algoritmo sulla quantità di informazioni che riesce ad elaborare.

5.1 Esempio 1.1.

Come primo esempio andiamo a testare il programma su un grafo che ha 100 nodi, e vediamo come si comporta l'algoritmo.

```
Tempo di esecuzione --- 0.003 secondi ---
Il cammino minimo è
 (11, ['23', '54', '2', '85'])
```

Come possiamo notare il cammino dal nodo 23 al nodo 85 , ha costo 11 è questa volta il tempo col quale l'algoritmo di Dijkstra ha eseguito i calcoli è 0.003 secondi, un tempo relativamente basso. Anche in questo caso facendo vari test, il tempo di esecuzione oscilla tra i 0.03 - 0.05 secondi.

5.2 Esempio 1.2.

In questo secondo esempio aumentiamo i nodi nel grafo, operiamo con un grafo di 500 nodi e vediamo cosa succede.

```
Tempo di esecuzione --- 0.082 secondi ---
Il cammino minimo è
 (61, ['2', '58', '134', '301', '499', '490'])
```

Con questo test notiamo che per arrivare dal nodo 2 al nodo 490, il percorso esiste ed è quello riportato in figura, insieme al costo del percorso, e in questo caso l'algoritmo di Dijkstra ha impiegato 0.082 secondi. Anche in questo caso, eseguendo più volte, il tempo di esecuzione oscilla tra i 0.081-0.083 secondi.

5.3 Esempio 1.3.

In questo ultimo test proviamo ad eseguire il programma su un grafo di 4000 nodi, e vediamo cosa succede.

```
Tempo di esecuzione --- 5.880 secondi ---
Il cammino minimo è
 (7, ['98', '1857', '996', '2333', '1358', '3895'])
```

Come possiamo notare dall'output risultante, il cammino tra il nodo 98 e il nodo 3895 esiste e ha costo 7. Invece il tempo di esecuzione del solo algoritmo di Dijkstra è aumentato notevolmente, arrivando a 5.880 secondi. Con varie esecuzioni il tempo di esecuzione oscilla tra i 5.451-6.232 secondi.

6 Conclusioni

Arrivati alla fine dei vari test sull'algoritmo di Dijkstra, abbiamo osservato il suo comportamento, nei casi in cui il grafo aveva una costruzione particolare, oppure valori degli archi negativi. Abbiamo osservato il tempo di esecuzione dell'algoritmo con grafi aventi pochi nodi fino ad arrivare a grafi di 4000 nodi, e di conseguenza abbiamo visto che con grafi di grandi dimensioni, l'algoritmo tende ad avere ovviamente una elaborazione più lunga.

Tabella dei tempi		
Numero di nodi	Soglia 1	Soglia 2
100	0.00	0.02
500	0.62	0.85
1000	0.340	0.350
1500	0.774	0.817
2000	1.434	1.472
2500	2.261	2.252
3000	3.275	3.288
4000	5.451	6.232

In questa tabella sono riportati i tempi con il quale l'algoritmo di Dijkstra lavora sul grafo. I tempi oscillano sostanzialmente dalla Soglia 1 alla Soglia 2. Le tempistiche possono mutare, tutto dipende dal cammino che porta da un nodo all'altro.

7 Appendice

7.1 Creazione e inizializzazione del grafo

```
from collections import defaultdict, deque

class Graph(object):
    def __init__(self):
        self.nodi = []
        self.archi = defaultdict(list)
        self.distanze = {}

    def AggiungiNodi(self, valore):
        self.nodi.append(valore)
        return self.nodi

    def AggiungiArchi(self, NodoA, NodoB, distanza):
        self.archi[NodoA].append(NodoB)
        self.distanze[(NodoA, NodoB)] = distanza
```

7.2 Algoritmo di Dijkstra

```
def dijkstra(graph, initial):
    start_time = time.time()
    nodi_visitati = {initial: 0}
    predecessori = {}

    nodi = set(graph.nodi)

    while nodi:
        min_node = None
        for node in nodi:
            if node in nodi_visitati:
                if nodi_visitati[node]<0:
                    print("Trovato costo arco negativo")
                    print("Tempo di esecuzione --- %.3f secondi ---
    " %(time.time()-start_time))
                    return ([],[])
                if min_node is None:
                    min_node = node
                elif nodi_visitati[node]<nodi_visitati[min_node]:
                    min_node = node
        if min_node is None:
            break

        nodi.remove(min_node)
        Peso_corrente=nodi_visitati[min_node]

        for edge in graph.archi[min_node]:
            try:
                Peso=Peso_corrente+graph.distanze[(min_node,edge)]
            except:
                continue
            if edge not in nodi_visitati or Peso<nodi_visitati[edge
    ]:
                nodi_visitati[edge]=Peso
                predecessori[edge]=min_node
    print("Tempo di esecuzione --- %.3f secondi ---" %(time.time()-
    start_time))
    return nodi_visitati, predecessori
```

7.3 Cammino minimo

```
def Cammino_minimo(graph, sorgente, destinazione):
        nodi_visitati, pred = dijkstra(graph, sorgente)
        tutti_i_predecessori = deque()
        result = 0
        if nodi_visitati==[]:
            return ([],[])
        if(sorgente == destinazione):
            result = 0
        else:
            try:
                nodo_destinazione = pred[destinazione]
            except:
                print("Cammino non presente")
                return ([],[])
            while nodo_destinazione != sorgente:
                tutti_i_predecessori.appendleft(nodo_destinazione)
                nodo_destinazione = pred[nodo_destinazione]


            tutti_i_predecessori.appendleft(sorgente)
            tutti_i_predecessori.append(destinazione)
            result = nodi_visitati[destinazione]
        return result,list(tutti_i_predecessori)
```

7.4 Main

```
if __name__ == '__main__':
    graph = Graph()


    for nodo in ['A', 'B', 'C', 'D', 'E']:
        graph.AggiungiNodo(nodo)

    graph.AggiungiArchi('A', 'D', 5)
    graph.AggiungiArchi('B', 'C', 3)
    graph.AggiungiArchi('C', 'A', 2)
    graph.AggiungiArchi('D', 'C', 20)
    graph.AggiungiArchi('E', 'B', 15)
    graph.AggiungiArchi('E', 'D', 10)

print(Cammino_minimo(graph, 'E', 'A'))
```

8 REFERENCES

- Thomas H. Cormen,Charles E. Leiserson,Ronald L. Rivest,Clifford Stein. Introduzione agli algoritmi e strutture dati.
- Massimo Pappalardo,Mauro Passacantando. Ricerca Operativa