Esercitazione 24 Aprile 2023

1. **Algoritmo di Dijkstra**

Pseudocodice:

```
1  Dijkstra(Grafo G, Funzione Peso ω, Sorgente S)
2  for ogni vertice v ∈ G.V do
3  |  v.d = MAX_INT // .d è la stima del cammino minimo
4  |  v.predecessore = null
5  S.d = 0 // Inizializziamo a 0 la stima del cammino minimo della sorgente S
6  Lista L = New Lista
7  Min-Heap Q = New Min-Heap
8  for ogni vertice v ∈ G.V do
9  |  insert(Q, v) // Metto gli elementi nel Min-Heap
10 while is-empty(Q) == False do
11 |  u = extractMin(Q) // Estrarre la radice dal Min-Heap costa O(log_2 V) per riordinarlo
12 |  // Rilassiamo gli archi connessi ad u
13 |  for ogni vertice v ∈ G.adj[u] do
14 |  |  if v.d > u.d + ω(u, v) then
15 |  |  |  v.d = u.d + ω(u, v)
16 |  |  |  HeapDecreseKey(Q, v) // Aggiorniamo la posizione di v nell'heap, costa O(log_2 V)
17 |  |  |  v.predecessore = u
18 |  append(L, u)
```

Informazioni:

L'algoritmo di Dijkstra serve a trovare Cammini Minimi da Sorgente Singola. Funziona su grafi Diretti, Connessi, che possono contenere Cicli ed i cui archi possono essere Pesati, ma con pesi non negativi (tranne al massimo i vertici uscenti dalla sorgente).

Oss: Se usiamo l'algoritmo di Dijkstra su un grafo con archi di costo negativo, esso troverà dei cammini dalla sorgente agli altri vertici, ma non è assicurato che siano quelli minimi. Il funzionamento dell'algoritmo si basa sul fatto che più ci si allontana dalla sorgente più il cammino minimo **aumenta** di peso; se vi sono archi di peso negativo, oltre a quelli fuoriuscenti dalla sorgente, questo principio non è assicurato.

Funzionamento:
Come sopra affermato, il funzionamento dell'algoritmo si basa sul fatto che più ci si allontana dalla sorgente più il cammino minimo dalla sorgente ai vertici aumenta di peso. Tramite questa proprietà si calcola il cammino minimo, quindi quello di peso minore, dalla sorgente S verso ogni altro vertice $v \in G.V$. Inizialmente l'algoritmo imposta a MAX_INT la stima dei cammini minimi ($.d$) di tutti i nodi ad eccezione della sorgente S, la cui stima è posta a 0. Poi inizializza la Lista L ed il Min-Heap Q. L'algoritmo inserisce tutti i vertici nel Min-Heap Q, dove sono ordinati secondo la loro stima del cammino minimo tra loro e la sorgente S, poi, finché nell'heap ci sono dei vertici, estrae quello con stima minore, rilassa tutti gli archi che fuoriescono da esso, modificandone la stima del cammino minimo $.d$, ed infine lo conserva in una lista L.

Costo Temporale:

Il costo dell'implementazione con un Min-Heap è:

$$T(n)^{Dijkstra} = T(n)^{lines(2-4)} + T(n)^{lines(8-9)} + T(n)^{lines(10-18)} =$$

$$O(V) + O(V) + O((V+E)\log_2 V) = O((V+E)\log_2 V)$$

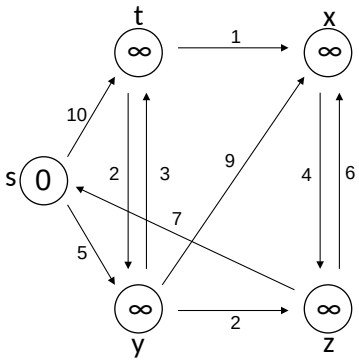
Infatti:

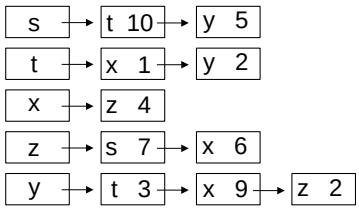
- $T(n)^{lines(2-4)} = V$: Infatti Scorriamo tutti i V vertici del grafo e impostiamo dei parametri.

- $T(n)^{lines(8-9)} = V$: Infatti inseriamo elementi con valore MAX_INT nell'heap, quindi ogni inserimento costa O(1) poiché essendo tutti uguali non ci sono risalite dei valori nell'heap. L'unica eccezione è nel caso della sorgente, ma essendo un'unica eccezione, di costo $O(\log_2 V)$, il suo costo viene ammortizzato. Il costo $O(\log_2 V)$ si ha perché la sorgente avendo valore della stima del cammino minimo pari a 0 andrà nella cima dell'heap, supponendo che venga inserita come ultimo nodo deve risalire l'intero heap.
- $T(n)^{lines(10-18)} = O((V + E)\log_2 V)$: Infatti abbiamo che il While preleva dall'heap tutti i V nodi del grafo, con un costo totale di $O(V \log_2 V)$, perché estrarre la radice del Min-Heap implica che deve esserne scelta una nuova, operazione di costo $O(\log_2 V)$ nel caso peggiore. Per ogni nodo si va ad analizzare ogni suo arco e si esegue un'operazione di aggiornamemto sulla stima del cammino minimo $.d$ di costo $O(1)$. Poiché non vi è reinserimento dei nodi nell'heap, e poiché per ogni nodo visitiamo tutti i suoi archi una sola volta, abbiamo $O(E)$ operazioni. Ma, anche quando cambiamo la stima dei cammini minimi dobbiamo mantenere le proprietà dello heap tramite la funzione *HeapDecreseKey*. La proprietà consiste nel fatto che in un Min-Heap ogni figlio deve essere maggiore del proprio padre; quando modifichiamo la stima del cammino minimo di un nodo, $.d$, dobbiamo mantenere la proprietà del Min-Heap. Nel caso pessimo in cui il nodo viene spostato dal fondo alla cima dell'heap abbiamo $O(\log_2 V)$ operazioni; infatti le stime dei nodi, per la condizione dell'If nel rilassamento, possono solo diminuire. Quindi il costo finale è di $O(V \log_2 V) + O(E \log_2 V) = O((V + E)\log_2 V)$.

Oss: Se il grafo è denso abbiamo $\#E \approx \#V^2$, quindi il costo diviene: $O(V^2 \log_2 V)$

Esempio di Esecuzione:





Abbiamo qui a sinistra un grafo in cui non vi sono cammini di costo negativo ed a destra la rappresentazione in memoria tramite lista di adiacenza del grafo. Nei nodi del grafo abbiamo scritto la loro stima del cammino minimo, che è già inizializzata (primo ciclo For dell'algoritmo).

Nodo	Padre	Stima Cammino Minimo ($.d$)
s	null	0
t	y	8
x	t	9
z	y	7
y	s	5

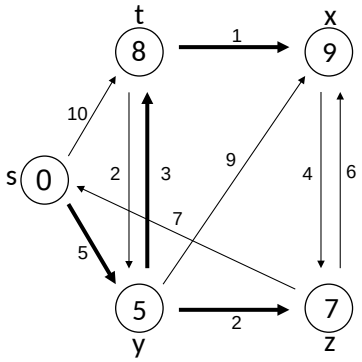
Dopo aver inserito gli elementi nel Min-Heap, tramite il secondo ciclo For, vediamo come si svolge il ciclo While. A destra abbiamo una tabella che riassume i padri e le stime dei cammini minimi ($.d$) di ogni nodo ad algoritmo concluso.
Iterate del Ciclo While:

1: $u = s$: $v = t \rightarrow t.d = 10$, $t.padre = s$ / $v = y \rightarrow y.d = 5$, $y.padre = s$
2: $u = y$: $v = t \rightarrow t.d = 8$, $t.padre = y$ / $v = x \rightarrow x.d = 14$, $x.padre = y$ / $v = z \rightarrow z.d = 7$, $z.padre = y$
3: $u = z$: $v = x \rightarrow x.d = 13$, $x.padre = z$
4: $u = t$: $v = x \rightarrow x.d = 9$, $x.padre = t$
5: $u = x \rightarrow$ Nulla

All'ultima iterazione del ciclo While quando $u = x$, non si esegue nulla, infatti, guardando la lista di adiacenza, l'unico vicino di x è z, ma essendo $z.d = 7 < ((x.d = 9) + (\omega(x, z) = 4))$ non si entra nell'If.

Nell'esempio non abbiamo tenuto conto nel del Min-Heap ne della lista, questi sono infatti dettagli implementativi. Il motivo per cui si usa un Min-Heap è che a noi serve il vertice con stima del cammino minimo minore e tenere ordinato un Min-Heap dove nella radice abbiamo sempre il vertice di nostro interesse è meno costoso dell'usare una lista. La lista in cui mettiamo i vertici dopo che

sono stati rilassati la utilizziamo semplicemente come mezzo per memorizzare le loro informazioni (padre e stima del cammino minimo).



Vediamo a sinistra il grafo alla fine dell'algoritmo di Dijkstra. Abbiamo dentro ai nodi la loro stima del cammino minimo e le frecce più spesse rappresentano gli archi dei cammini minimi.

Esercizio di Esempio:

Dato un grafo pesato, con pesi non negativi, ed una destinazione s, e data una costante k, trovare i vertici a distanza $\leq k$ dalla destinazione s.

Soluzione:

Svolgimento:

Notiamo che il problema dice *destinazione* s, ed inoltre pone un limite k alla distanza dei vertici dalla destinazione. Per poter usare Dijkstra a noi serve una sorgente, quindi dobbiamo invertire tutti gli archi del grafo, quindi dobbiamo trovare il Grafo Trasposto di G, chiamato G'; a questo punto possiamo eseguire Dijkstra su G' impedendogli di proseguire oltre la distanza k.

Supponiamo che il grafo sia rappresentato con una lista di adiacenza.

Pseudocodice:

```
1 GrafoTrasposto(Grafo G)
2 Grafo G' = New Grafo // Il nuovo grafo ha E, V ed adj[]
3 G'.V = G.V
4 for ogni arco (u, v) ∈ G.E do
5     G'.E = (v, u)
6 for ogni vertice u ∈ G.V do
7     for ogni vertice v ∈ G.adj[u] do
8         append(G'.adj[v], u) // Mettiamo u nella lista di adiacenza di v
9 return G'
```

```
1  Dijkstra_K(Grafo G, Funzione Peso ω, Sorgente S)
2  for ogni vertice v ∈ G.V do
3  |  v.d = MAX_INT
4  |  v.predecessore = null
5  S.d = 0
6  Lista L = New Lista
7  Min-Heap Q = New Min-Heap
8  for ogni vertice v ∈ G.V do
9  |  insert(Q, v)
10 stop = false // Variabile per fermare il ciclo while
11 while is-empty(Q) == False and stop == true do
12 |  u = extractMin(Q)
13 |  // Rilassiamo gli archi connessi ad u fino a stima k
14 |  if u.d > k then
15 |  |  stop = true
16 |  |  // Usciamo dal ciclo While appena troviamo un vertice a distanza maggiore di k
17 |  else
18 |  |  for ogni vertice v ∈ G.adj[u] do
19 |  |  |  if v.d > u.d + ω(u, v) then
20 |  |  |  |  v.d = u.d + ω(u, v)
21 |  |  |  |  v.predecessore = u
22 |  enqueue(L, u)
```

Spiegazione:

Qui abbiamo creato la funzione che trova il trasposto G' di un grafo G ed una versione modificata dell'algoritmo di Dijkstra che si ferma al primo vertice che estrae a distanza superiore a k. Nella prima abbiamo trovato i tre elementi di G': V, E, $adj[]$. Poiché i vertici sono invariati da un grafo al suo trasposto, $G'.V = G.V$. Per trovare $G'.E$ abbiamo invertito tutte le coppie di vertici in $G.E$. Per trovare la lista di adiacenza trasposta, abbiamo letto tutta la lista di adiacenza di G, e la abbiamo trascritta in $adj'[]$ invertendo le connessioni dei vertici. Nell'algoritmo di Dijkstra modificato abbiamo semplicemente aggiunto dei controlli per fermare l'algoritmo quando la variabile booleana *stop*, inizializzata a *false*, diviene *true*, evento che si verifica qundo si estrae dal Min-Heap un vertice avente stima del cammino minimo maggiore del nostro limite k.

Complessità:

Il costo dell'implementazione con il Min-Heap in Dijkstra_K è:

$$T(n) = T(n)^{Trasposto} + T(n)^{Dijkstra\_K} = O(V+E) + O((V+E)\log_2 V) = O((V+E)\log_2 V)$$

Infatti:

- $T(n)^{Trasposto} = O(V+E)$: Infatti per realizzare il grafo trasposto dobbiamo fare $O(E)$ operazioni per trovare $G'.E$, poiché dobbiamo leggere tutto $G.E$ ed $O(V+E)$ operazioni per trovare $G'.adj[]$, poiché dobbiamo leggere tutta la lista $G.adj[]$; per un costo finale di $O(V+E)$.
- $T(n)^{Dijkstra\_K} = O((V+E)\log_2 V)$: Infatti è il costo del normale algoritmo di Dijkstra, nel caso peggiore $k = maximum\_d$ e quindi si esegue tutto l'algoritmo.

2. **Algoritmo di Bellman-Ford**

Pseudocodice:

```
1  Bellman-Ford(Grafo G, Funzione Peso ω, Sorgente S)
2  for ogni vertice v ∈ G.V do
3  |   v.d = +∞ // .d è la stima del cammino minimo
4  |   v.predecessore = null
5  S.d = 0 // Inizializziamo a 0 la stima del cammino minimo della sorgente S
6  for i to |G.V| − 1 do
7  |   for ogni arco (u, v) ∈ G.E do
8  |   |   if v.d > u.d + ω(u, v) then
9  |   |   |   v.d = u.d + ω(u, v)
10 |   |   |   v.predecessore = u
11 // Controllo per cicli di costo negativo
12 for ogni arco (u, v) ∈ G.E do
13 |   if v.d > u.d + ω(u, v) then
14 |   |   return FALSE
15 return TRUE
```

Informazioni: L'algoritmo di **Bellman-Ford** risolve il problema dei cammini minimi da sorgente unica in cui i pesi degli archi possono essere negativi.

Funzionamento:

Dato un grafo $G = (V, E)$, una sorgente S e una funzione peso $\omega : E \to \mathcal{R}$, l'algoritmo di Bellman-Ford restituisce un booleano che indica la presenza o meno di un ciclo di costo negativo. Se esiste un ciclo di costo negativo l'algoritmo termina segnalando che non esiste soluzione, altrimenti l'algoritmo trova i cammini minimi e i relativi pesi.

L'algoritmo fonda il suo funzionamento sulla funzione RELAX per ridurre progressivamente il valore stimato $v.d$ al fine di ridurre il peso di un cammino minimo dalla sorgente S a ciascun nodo di $v \in V$, fino a raggiungere l'effettivo peso $\delta(S, v)$ di un cammino minimo.

Nel dettaglio quindi, l'algoritmo parte con l'inizializzare i valori di d e π di tutti i vertici (Riga 2-5). Successivamente l'algoritmo effettua $|V| - 1$ passaggi sugli archi del grafo (Riga 6). Durante ogni iterazione del ciclo Riga 6-10 effettuiamo una RELAX per ogni arco in E. Dopo aver effettuato $|V| - 1$ passaggi, nelle Riga 12-14 controlliamo l'esistenza di un ciclo negativo restituendo in caso il valore booleano associato.

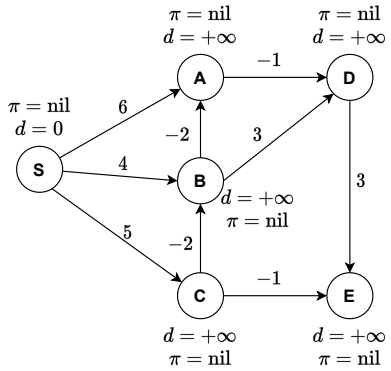
Costo Temporale:

L'algoritmo di Bellman-Ford viene eseguito in tempo $\mathcal{O}(VE)$, perchè l'inizializzazione richiede $\Theta(V)$, ciascuno $|V| - 1$ passaggi sugli archi (Riga 6-10) richiede tempo $\Theta(E)$ e il controllo per eventuali cicli di costo negativo richiede $\mathcal{O}(E)$.

Esempio di Esecuzione:

Dato il seguente grafo calcoliamo nel seguente modo:

$E = \{(S,A),(C,B),(S,C),(D,E),(C,E),(S,B),(B,D),(B,A),(A,D)\}$



a)

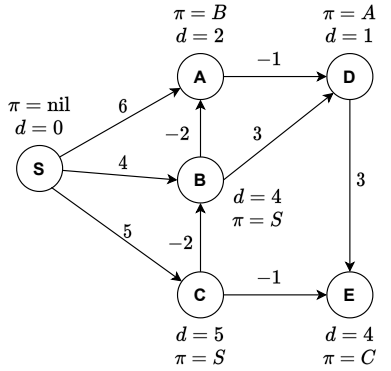
$A.d > S.d + \omega(S,A) = +\infty > 0 + 6$

$\vdots$

$E.d > D.d + \omega(D,E) = +\infty > +\infty + 3$

$\vdots$

$D.d > A.d + \omega(A,D) = +\infty > 2 - 1$



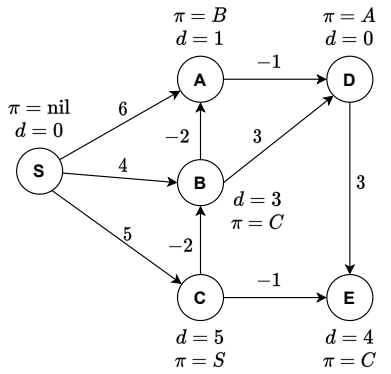
b)

$A.d > S.d + \omega(S,A) = 2 > 0 + 6$

$B.d > C.d + \omega(C,B) = 4 > 5 - 2$

$\vdots$

$D.d > A.d + \omega(A,D) = 1 > 1 - 1$



c)

$A.d > S.d + \omega(S,A) = 2 > 0 + 6$

$\vdots$

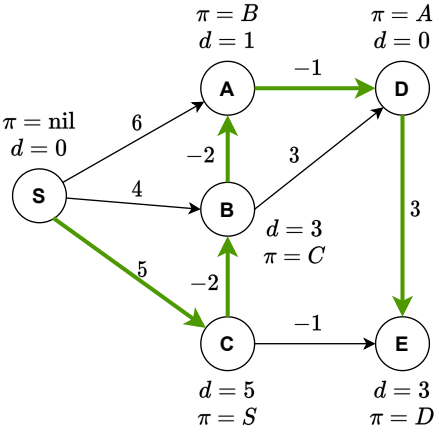
$E.d > D.d + \omega(D,E) = 4 > 0 + 3$

$\vdots$

$D.d > A.d + \omega(A,D) = 0 > 1 - 1$

Pertanto dato che abbiamo trovato i cammini minimi fra S e tutti i nodi $v \in V$, le restanti iterazioni non apporteranno modifiche, e la soluzione sarà la seguente:

Soluzione :



Esercizio 1 di Esempio:

Dato un grafo $G = (V, E)$ pesato con pesi qualsiasi che non ha cicli di lunghezza negativa, modificare Bellman-Ford in modo da fermarsi non appena sono state trovate tutte le distanze minime.

Soluzione:

Svolgimento:

Per risolvere tale esercizio è necessario sfruttare il fatto che il grafo G non contiene cicli di costo negativo. Infatti, avere tale garanzia ci permette di effettuare fin da subito una modifica al codice di Bellman-Ford, ovvero rimuove il ciclo FOR che controlla la presenza di cicli di costo negativo.

```
1  Bellman-Ford(Grafo G, Funzione Peso ω, Sorgente S)
2  for ogni vertice v ∈ G.V do
3      v.d = +∞ // .d è la stima del cammino minimo
4      v.predecessore = null
5  S.d = 0 // Inizializziamo a 0 la stima del cammino minimo della sorgente S
6  for i to |G.V| − 1 do
7      for ogni arco (u, v) ∈ G.E do
8          if v.d > u.d + ω(u, v) then
9              v.d = u.d + ω(u, v)
10             v.predecessore = u
11 // Controllo per cicli di costo negativo
12 for ogni arco (u, v) ∈ G.E do
13     if v.d > u.d + ω(u, v) then
14         return FALSE
15 return TRUE
```

Tuttavia possiamo fare ancora qualche piccola modifica, infatti non avendo cicli di costo negativo sono garantito che da un certo punto in poi non farò più RELAX. In particolare, se l'ordine con cui il ciclo FOR (Riga 7) visita gli archi è particolarmente fortunato, l'algoritmo di Bellman-Ford potrebbe trovare tutti i cammini minimi con un numero di iterazioni del ciclo FOR (Riga 6) $\leq |V| - 1$. Pertanto conteggiando quante RELAX vengono effettuate ad ogni iterazione del primo ciclo FOR, posso arrestarmi anticipatamente se il numero di RELAX svolte è pari a 0.

Pseudocodice:

```
1  Bellman-Ford-Modificato(Grafo G, Funzione Peso ω, Sorgente S)
2  for ogni vertice v ∈ G.V do
3      v.d = +∞ // .d è la stima del cammino minimo
4      v.predecessore = null
5  S.d = 0 // Inizializziamo a 0 la stima del cammino minimo della sorgente S
6  conta_relax = 0
7  flag_relax = TRUE
8  while i ≥ |G.V| − 1 and flag_relax == TRUE do
9      conta_relax = 0
10     for ogni arco (u, v) ∈ G.E do
11         if v.d > u.d + ω(u, v) then
12             v.d = u.d + ω(u, v)
13             v.predecessore = u
14             conta_relax = conta_relax + 1
15         i = i + 1
16     if conta_relax == 0 then
17         flag_relax = FALSE
```

Spiegazione:

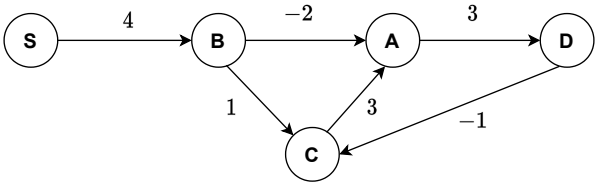
Lo pseudocodice risultante cosi come già discusso non presenta il ciclo FOR che controlla la presenza di cicli di costo negativo, ed inoltre è sono state aggiunte due nuove variabili per effettuare early-stopping una volta raggiunta la soluzione. Nel dettaglio sono state introdotte due variabili, una intera *conta_relax* che come suggerisce il nome ci consente di tenere aggiornato

il numero di rilassamenti che effettuiamo, e una booleana $flag\_relax$ che ci permette di riscontrare la fine dell'algoritmo. Infatti, ad ogni rilassamento effettuato incrementiamo $conta\_relax$ di 1, ma se alla fine del FOR più esterno non abbiamo fatto alcuna RELAX, assegniamo a $flag\_relax$ il valore FALSE, e all'iterazione successiva termineremo.

Per quanto riguarda il primo FOR, è stato fatto un leggero abuso di notazione, in particolare tale ciclo va interpretato nel seguente modo: *"cicla mentre vi è ancora un arco da testare e il flag è uguale* TRUE*"*. PS: è stato messo il connettore logico **and** perchè il ciclo FOR cicla per vero, quindi fintanto che la condizione espressa è verifica. Segue che ne basta almeno una non verificata per uscire.

Complessità: La complessità della nostra funzione **Bellman-Ford-Modificato** rimane asintoticamente $\mathcal{O}(VE)$. Questo perchè il codice scritto aggiunge soltanto istruzioni con tempo computazionale pari a $\mathcal{O}(1)$, pertanto spendiamo sicuramente al più quanto Bellman-Ford. La modifica fatta ci permette tuttavia di dare un upper-bound più stretto rispetto al cammino minimo più lungo π, i.e., $\mathcal{O}(V\pi)$. Inoltre, in casi particolarmente sfortunati, la nuova condizione di stop non ci garantisce di ottenere con meno passi la soluzione del nostro problema. Ad esempio, avendo il grafo e ordine degli archi in E, riportato come nella figura seguente, sarò costretto ad effettuare $|V| - 1$ iterazioni del ciclo FOR più esterno.

$$E = \{(D, C), (A, D), (C, D), (B, A), (B, C)(S, B)\}$$



Esercizio 2 di Esempio:

Dato un grafo $G = (V, E)$ che rispetta la definizione di Directed Acyclic Graph (DAG), sfruttare le proprietà della struttura data per velocizzare l'algoritmo di Bellman-Ford. Studiare la complessità della soluzione prodotta.

Soluzione:

Svolgimento:

Per svolgere tale esercizio dobbiamo innanzitutto conoscere le proprietà di un DAG. Quando parliamo di un DAG, ci stiamo riferendo ad un grafo diretto senza cicli, i cui vertici possono essere linearizzati, i.e., topologicamente ordinato. Un ordinamento topologico permette quindi di ordinare i vertici lungo una linea orizzontale in modo tale che gli archi siano diretti da sinistra a destra. Pertanto la prima cosa che possiamo osservare è che non dobbiamo preoccuparci di avere cicli di costo negativo, visto che un DAG è aciclico per definizione.

Inoltre sapendo che un DAG può essere ordinato topologicamente, possiamo sfruttare tale peculiarità per risolvere più velocemente il problema dei cammini minimi. Infatti possiamo sfruttare l'ordine dato dal sorting come ordine di rilassamento dei vertici. Pertanto non avendo cicli, e avendo un ordine univoco degli archi possiamo calcolare i cammini minimi usando Bellman-Ford rilassando esattamente una volta ogni arco.

Pseudocodice:

```
1  Bellman-Ford-DAG(Grafo G, Funzione Peso ω, Sorgente S)
2  Topological-Sort(G, s)
3  Ordina E rispetto al tempo di fine del nodo da cui gli archi escono
4  for ogni vertice v ∈ G.V do
5      v.d = +∞ // .d è la stima del cammino minimo
6      v.predecessore = null
7  S.d = 0 // Inizializziamo a 0 la stima del cammino minimo della sorgente S
8  for ogni arco (u, v) ∈ G.E do
9      if v.d > u.d + ω(u, v) then
10         v.d = u.d + ω(u, v)
11         v.predecessore = u
```

Spiegazione:

L'idea chiave è quella di sfruttare i tempi di completamento dei nodi per ordinare gli archi dell'insieme E. Dato che un DAG può essere organizzato in modo tale che i suoi archi siano direzionati da destra verso sinistra e viceversa. Sfruttiamo questo stesso ordine per rilassare gli archi. Garantiti dall'assenza dei cicli, possiamo eliminare il ciclo FOR più esterno presente nella classica implementazione di Bellman-Ford.

Complessità:

Il costo del nostro algoritmo può essere scomposto in:

- Sort Topologico $\mathcal{O}(V + E)$
- Inizializzazione $\mathcal{O}(V)$
- Ciclo FOR in cui scorro gli archi $\mathcal{O}(E)$

Segue quindi che il costo finale del nostro algoritmo è $\mathcal{O}(V + E)$.