Metodi divide-et-impera, programmazione dinamica e algoritmi greedy

La *programmazione dinamica*, come il metodo *divide-et-impera*, risolve un problema mettendo insieme le soluzioni di un certo numero di sottoproblemi. In particolare i metodi divide-et-impera suddividono il problema in sottoproblemi indipendenti, li risolve ricorsivamente e la soluzione del problema originale viene determinata fondendo assieme le soluzioni dei sottoproblemi. La programmazione dinamica, invece, si può applicare anche quando i sottoproblemi non sono indipendenti: la risoluzione di alcuni sottoproblemi richiede di risolvere i medesimi sottoproblemi. In questi casi gli algoritmi del tipo divide-et-impera farebbero più lavoro di quello strettamente necessario dovendo risolvere più volte gli stessi sottoproblemi. Gli algoritmi di programmazione dinamica risolvono ciascun sottoproblema una sola volta e memorizzano la soluzione in una tabella; in questo modo si evita di calcolare nuovamente la soluzione ogni volta che deve essere risolto lo stesso sottoproblema. Per alcuni problemi di ottimizzazione, comunque, la scelta di metodologie di programmazione dinamica non è sempre la migliore in quanto è possibile realizzare algoritmi più semplice ed efficienti. Gli *algoritmi greedy* adottano la strategia di prendere quella decisione che, al momento, appare come la migliore. In altri termini, viene sempre presa quella decisione che, localmente, è la decisione ottima, senza preoccuparsi se tale decisione porterà a una soluzione ottima del problema nella sua globalità. Non è sempre facile o possibile sviluppare un algoritmo greedy ma vi sono particolari classi di problemi che si prestano bene ad approcci greedy. Una di queste classi di problemi è quella della ricerca in grafi pesati di percorsi a minimo costo, anzi in questi casi sono stati sviluppati approcci greedy considerati ad oggi i più efficienti.

I grafi

I grafi sono strutture dati molto diffuse in ottimizzazione e in informatica. Un grafo è rappresentato graficamente tramite una struttura a nodi ed archi. I nodi possono essere visti come eventi dal quale si dipartono differenti alternative (gli archi). Tipicamente i grafi vengono utilizzati per rappresentare in modo biunivoco una rete (di calcolatori, stradale, di servizi, …): i nodi rappresentano i singoli calcolatori, gli incroci stradali o le fermate di autobus e gli archi i collegamenti elettrici o le strade.

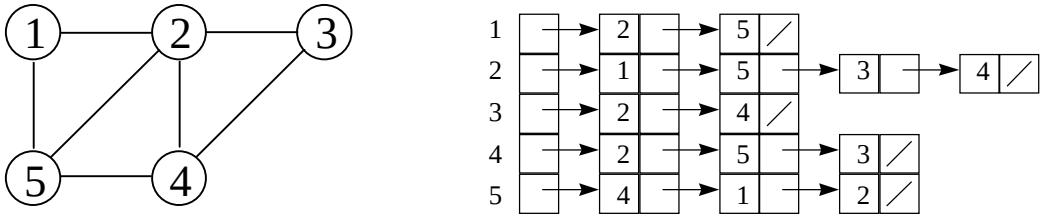
Un grafo viene indicato in modo compatto con $G = (V, E)$, dove V indica l'insieme dei nodi ed E l'insieme degli archi che lo costituiscono, il numero di nodi è $|V|$ e il numero di archi $|E|$.

Vi sono diverse tipologie di grafi: si parla di grafi *non orientati* quelli per i quali gli archi non hanno un verso di percorrenza in contrapposizione a quelli *orientati*. Ad esempio i grafi non orientati

vengono utilizzati per rappresentare reti di calcolatori con collegamenti sincroni per la trasmissione dei dati, i grafi orientati possono rappresentare reti viarie, permettendo la rappresentazione di doppi sensi e sensi unici. Si parla di grafi *connessi* se da un qualunque nodo si può raggiungere tutti gli altri nodi del grafo. Si parla di grafi *pesati* se ad un ogni arco $(u, v) \in E$, con $u, v \in V$, è associato un *peso*, normalmente definito da una *funzione peso* w: $E \rightarrow \Re$. Il peso può essere visto come un costo o la distanza fra i due nodi che l'arco unisce. Il costo può essere dipendente dal flusso che attraversa l'arco tramite una legge. In tal senso la funzione w può essere lineare o meno e dipendere dal flusso che attraversa l'arco (*reti non congestionate*) o anche dal flusso degli archi vicini (*reti congestionate*).

Vi sono due modi standard di rappresentare un grafo: come una collezione di *liste di adiacenza* o come una *matrice di adiacenza*. Di solito si preferisce la prima perché fornisce un modo compatto di rappresentare grafi *sparsi*, cioè quelli per cui $|E|$ è molto inferiore rispetto $|V|^2$, mentre il secondo tipo di rappresentazione è utilizzato per grafi *densi*, ossia $|E| \cong |V|^2$.

La rappresentazione con liste di adiacenza consiste in un vettore di $|V|$ liste, una per ogni vertice. Per ogni nodo $u \in V$, la lista di adiacenza $Adj[u]$ contiene tutti i vertici v tali che esista un arco $(u, v) \in E$; quindi $Adj[u]$ comprende tutti i vertici adiacenti ad u in G. In ogni lista di adiacenza i vertici vengono di solito memorizzati in ordine arbitrario.



Se G è un grafo orientato, la somma delle lunghezze di tutte le liste di adiacenza è $|E|$, perché un arco della forma (u, v) è rappresentato ponendo v in $Adj[u]$. Se il grafo non è orientato la somma delle lunghezze di tutte le liste di adiacenza è $2|E|$, dato che v appare nella lista di adiacenza di u e viceversa. La rappresentazione con liste di adiacenza è robusta, ossia può essere adattata a molte varianti di grafi, ma qualunque sia il tipo di grafo l'occupazione di memoria massima è $O(V+E)$.

Per la rappresentazione con matrice di adiacenza di un grafo $G = (V,E)$, si assume che i vertici siano numerati $1,2,\ldots,|V|$ in modo arbitrario. La rappresentazione consiste in una matrice $A = (a_{ij})$ di dimensione $|V| \times |V|$ tale che $a_{ij} = 1$ se $(i, j) \in E$, 0 altrimenti. Nel caso del grafo precedente la matrice di adiacenza è:

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

La memoria occupata è $\Theta(V^2)$ indipendentemente dal numero di archi. Con la matrice di incidenza si possono rappresentare autocicli come nelle liste di adiacenza ma non il caso di più archi che collegano due nodi. Si noti che nel caso di grafi non orientati la matrice di adiacenza è simmetrica e quindi basterebbe utilizzare i dati sopra la diagonale principale (diagonale inclusa) riducendo di quasi la metà il numero di dati da memorizzare.

Cammini minimi

Sia dato il grafo pesato $G = (V,E)$. Un cammino da un nodo u ad un nodo v è una sequenza di archi contigui che unisce i due nodi. Il cammino viene rappresentato in modo indicizzato come sequenza di nodi: $p = (v_0, v_1, \ldots, v_i, \ldots, v_k)$ ove $v_i \in V$, $(v_i, v_{i+1}) \in E$, $i = 1,\ldots,k$, $v_0 = u$ e $v_k = v$.

Sia dato il grafo orientato e pesato $G = (V, E)$, con funzione di peso w: $E \rightarrow \Re$.

Il peso di un cammino p è la somma dei pesi degli archi che lo costituiscono:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) .$$

Il *peso di cammino minimo* da u a v è definito come

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \xrightarrow{p} v \} & \text{se esiste un cammino da } u \text{ a } v, \\ \infty & \text{altrimenti.} \end{cases}$$

Un *cammino minimo* dal vertice u al vertice v è definito come un qualunque cammino p con peso $w(p) = \delta(u, v)$. Il peso degli archi può essere inteso come una distanza, ma anche come tempo, costo, penalità, perdita o qualunque altra grandezza che si accumula in modo lineare e che si vuole minimizzare. Vi sono diverse varianti del problema, fra le quali le due più importanti sono: cammino minimo fra una coppia di nodi, e alberi di cammini minimi. Nel primo caso si cerca un cammino minimo fra due nodi del grafo, nel secondo si cercano i cammini minimi fra un nodo e tutti gli altri della rete.

Rappresentazione dei cammini minimi

Spesso non è importante solo conoscere il peso di cammino minimo, ma anche i vertici di tale cammino. Dato un grafo $G = (V, E)$, si mantenga per ogni vertice $v \in V$ un *predecessore* $\pi[v]$ che è un altro vertice oppure NIL (ossia Nulla). Gli algoritmi di cammino minimo che vedremo utilizzano l'operatore π in modo tale che la catena dei predecessori che parte da un vertice v segua in direzione inversa, un cammino minimo da s a v.

Durante l'esecuzione dell'algoritmo i valori di π, tuttavia, possono non indicare i cammini minimi. In realtà saremo interessati al *sottografo dei predecessori* $G_\pi = (V_\pi, E_\pi)$ indotto dai valori di π dove:

V_π è l'insieme di vertici di G con predecessore diverso da NIL ossia $V_\pi = \{v \in V: \pi[v] \neq \text{NIL}\} \cup \{s\}$

E_π è l'insieme di archi orientati indotto da π per i vertici V_π, cioè $E_\pi = \{(\pi[v], v) \in E: v \in V_\pi - \{s\}\}$.

Gli algoritmi che vedremo avranno la proprietà che alla fine G_π sarà un albero di cammini minimi con radice s.

Sottostruttura ottima di un cammino minimo

Gli algoritmi di cammino minimo normalmente sfruttano la proprietà che un cammino minimo tra due vertici contiene al suo interno altri cammini minimi. Questa proprietà di sottostruttura garantisce l'applicabilità sia della programmazione dinamica che del metodo greedy.

Lemma 1- Sottocammini di cammini minimi sono cammini minimi

Dato un grafo orientato e pesato $G = (V, E)$ con una funzione peso $w: E \rightarrow \Re$, sia $p = (v_1, v_2, \ldots, v_k)$ un cammino minimo dal vertice v_1 al vertice v_k, e per ogni i e j tali che $1 \leq i \leq j \leq k$, sia $p_{ij} = (v_i, v_{i+1}, \ldots, v_j)$ il sottocammino di p dal vertice v_i al vertice v_j. Allora p_{ij} è un cammino minimo da v_i a v_j.

Dimostrazione Se si decompone il cammino p come $p = p_{1i} + p_{ij} + p_{jk}$, allora $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$. Si supponga che esiste un cammino p_{ij}' con peso $w(p_{ij}') \leq w(p_{ij})$. Allora $p' = p_{1i} + p_{ij}' + p_{jk}$ sarebbe un cammino da v_1 a v_k, con peso inferiore di p contraddicendo l'ipotesi.

Corollario 1

Sia $G = (V, E)$ un grafo orientato e pesato con funzione peso $w: E \rightarrow \Re$. Si supponga che un cammino minimo p da una sorgente s ad un vertice v possa essere decomposto in $s \xrightarrow{p'} u \rightarrow v$ per un qualche vertice u e cammino p'. Allora il peso di cammino minimo da s a v è $\delta(s, v) = \delta(s, u) + w(u, v)$.

Dimostrazione Per il lemma 1 il sottocammino p' è un cammino minimo dalla sorgente s al vertice u. Quindi

$$\begin{aligned}\delta(s, v) &= w(p) \\ &= w(p') + w(u, v) \\ &= \delta(s, u) + w(u, v)\end{aligned}$$

Lemma 1

Sia $G = (V, E)$ un grafo orientato e pesato con funzione peso $w: E \rightarrow \Re$ e vertice sorgente s. Allora per ogni arco $(u, v) \in E$ vale $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Rilassamento

Gli algoritmi che vedremo usano la tecnica del *rilassamento*. Per ogni vertice $v \in V$, si mantiene un attributo $d[v]$ che costituisce un limite superiore al peso di un cammino minimo dalla sorgente s a v: chiameremo $d[v]$ una *stima del cammino minimo*. Le stime di cammino minimo e i predecessori vengono inizializzati dalla seguente procedura:

```
Inizializza (G, s)
1    for ogni vertice v ∈ V
2         do   d[v] ← ∞
3              π[v] ← NIL
4    d[s] ← 0
```

Il processo di *rilassare* un arco (u, v) consiste nel verificare se si può migliorare il cammino minimo per v trovato fino a quel momento passando per u, e, in questo caso, nell'aggiornare $d[v]$ e $\pi[v]$.
Il rilassamento sarà la seguente procedura:

```
Rilassa(u, v, w)
1    if d[v] > d[u] + w(u, v)
2         then d[v] ← d[u] + w(u, v)
3              π[v] = u
```

Algoritmo di Dijkstra

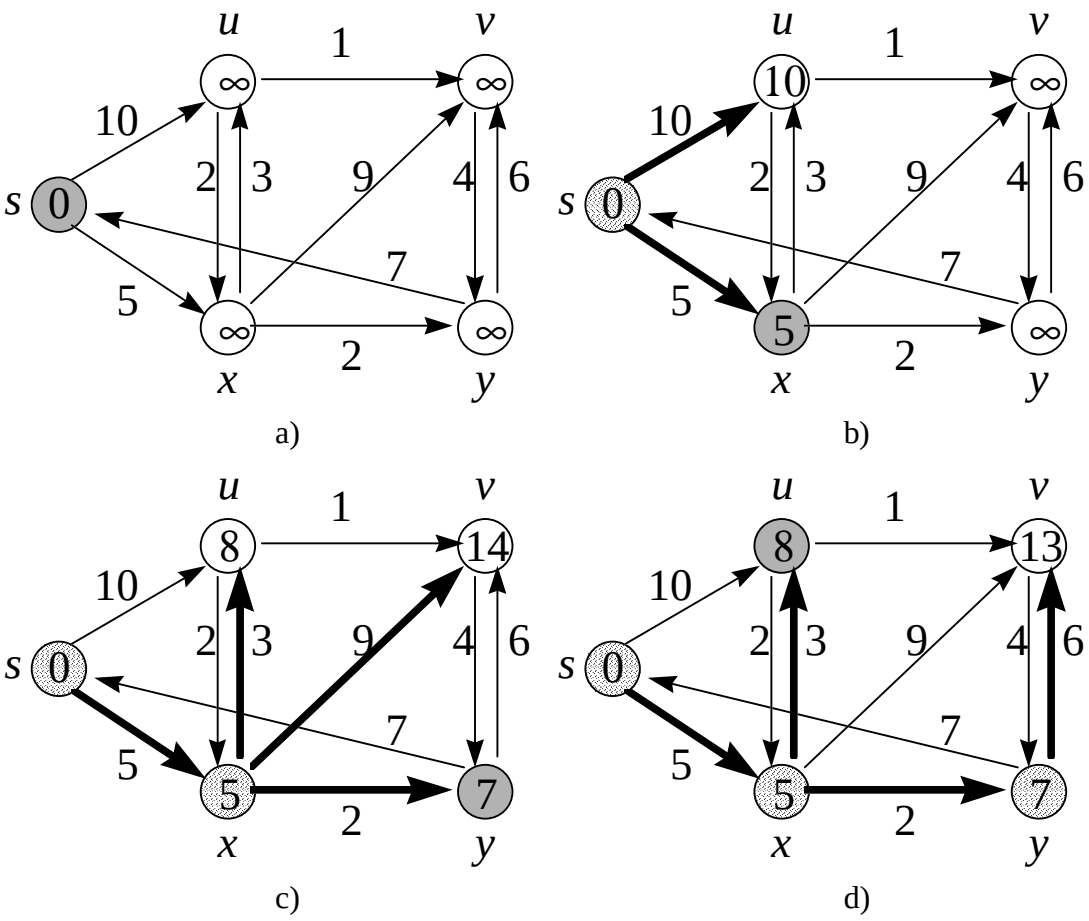
L'algoritmo di Dijkstra utilizza una strategia greedy per risolvere il *problema di cammini minimi con sorgente singola su di un grafo orientato e pesato* $G = (V, E)$ nel caso in cui tutti i *pesi* degli archi siano *non negativi*. L'algoritmo mantiene un insieme S che contiene i vertici il cui peso di cammino minimo dalla sorgente s è già stato determinato, cioè per tutti i vertici $v \in S$ vale $d[v] = \delta(s, v)$. L'algoritmo seleziona ripetutamente i vertici $u \in V - S$ con la minima stima di cammino minimo, inserisce u in S, e rilassa tutti gli archi uscenti da u. Inoltre viene mantenuta una coda con priorità Q che contiene tutti i vertici $V - S$, usando come chiave i rispettivi valori d ; la realizzazione assume che il grafo G sia rappresentato con liste di adiacenza.

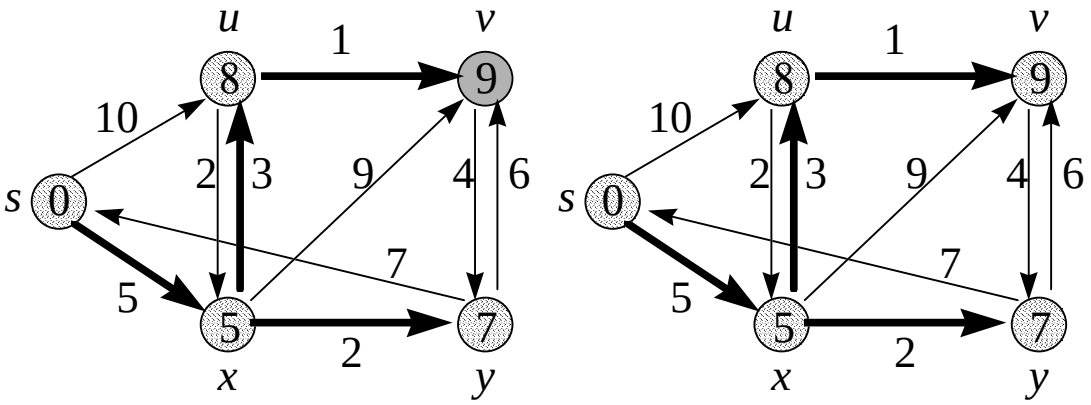
```
Dijkstra(G, w, s)
1    Inizializza(G, s)
2    S ← ∅
3    Q ← V
4    while Q ≠ ∅
5         do   u ← Estrai_Minimo(Q)
6              S ← S ∪ {u}
7              for ogni vertice v ∈ Adj[u]
8                   do   Rilassa(u, v, w)
```

La linea 1 esegue la normale inizializzazione dei valori d e π, e la linea 2 inizializza l'insieme S con l'insieme vuoto. La linea 3 inizializza la coda con priorità Q con tutti i vertici in $V - S = V - \emptyset = V$. Ogni volta che si esegue il ciclo while delle linee 4-8, un vertice u viene estratto da $Q = V - S$ e viene inserito nell'insieme S (la prima volta che viene eseguito il ciclo, $u = s$). Il vertice u ha quindi la minima stima di cammino minimo tra tutti i vertici in $V - S$. Quindi le linee 7-8 rilassano ogni arco (u, v) che esce da u, aggiornando la stima $d[v]$ ed il predecessore $\pi[v]$ se il cammino minimo per v può essere migliorato passando per u. Si noti che nessun vertice viene inserito in Q dopo la linea 3, e che ogni vertice viene estratto da Q ed inserito in S esattamente una volta, quindi il ciclo while delle linee 4-8 viene ripetuto esattamente $|V|$ volte. La convergenza all'ottimo dell'algoritmo di Dijkstra è stata dimostrata. Quanto è veloce l'algoritmo di Dijkstra? Consideriamo il caso in cui mantiene la coda con priorità $Q = V - S$ con un array lineare. Infatti Estrai_Minimo è una funzione che estrae il minimo elemento in un vettore. Estrai_Minimo richiede un tempo $O(V)$, e poiché vi sono $|V|$ operazioni di questo tipo il tempo totale richiesto da Estrai_Minimo è $O(V^2)$. Ogni vertice $v \in V$ viene inserito nell'insieme S esattamente una volta, e quindi ogni arco nella lista di adiacenza $Adj[v]$ viene

esaminato nel ciclo for delle linee 4-8 esattamente una volta nel corso dell'algoritmo. Poiché il numero totale di archi in tutte le liste di adiacenza è $|E|$, vi sono in totale $|E|$ iterazioni di questo ciclo for, ognuna delle quali richiede tempo $O(1)$. Quindi il tempo totale di esecuzione dell'algoritmo è $O(E + V^2) = O(V^2)$. Nel caso di grafi sparsi se si utilizza un heap binario per rappresentare la coda con priorità Q, si ottiene il cosiddetto *algoritmo di Dijkstra modificato* che richiede un tempo di calcolo totale pari a $O(E\lg V)$ se tutti i vertici sono raggiungibili dalla sorgente. Tale tempo può essere abbassato a $O(V\lg V + E)$ usando heap di Fibonacci.

Esempio





Una esecuzione dell'algoritmo di Dijkstra. La sorgente è il vertice *s*. Le stime di cammino minimo sono indicate all'interno dei vertici, e gli archi in neretto indicano i valori del campo predecessore: se l'arco (*u*, *v*) è in neretto, allora $\pi[v] = u$. I vertici grigi sono nell'insieme *S*, mentre i vertici bianchi sono nella coda *Q*. (a) La situazione subito prima della prima iterazione del ciclo while delle linee 4-8. Il vertice in grigio ha il valore minimo di *d* ed è scelto come vertice *u* nella linea 5. (b)-(f) la situazione dopo ogni iterazione successiva del ciclo while. Il vertice in grigio è il nuovo nodo *u*.

Algoritmo di Bellman-Ford

L'algoritmo di Bellman-Ford risolve il *problema dei cammini minimi con sorgente singola* nel caso più generale in cui i *pesi* degli archi possono essere *negativi*. Dato un grafo orientato e pesato $G = (V, E)$ con sorgente *s* e funzione peso w: $E \rightarrow \Re$, l'algoritmo di Bellman-Ford restituisce un valore booleano che indica se esiste oppure no un ciclo di peso negativo raggiungibile della sorgente. In caso positivo, l'algoritmo indica che non esiste alcuna soluzione ; se invece un tale ciclo non esiste, allora l'algoritmo produce i cammini minimi ed i loro pesi.

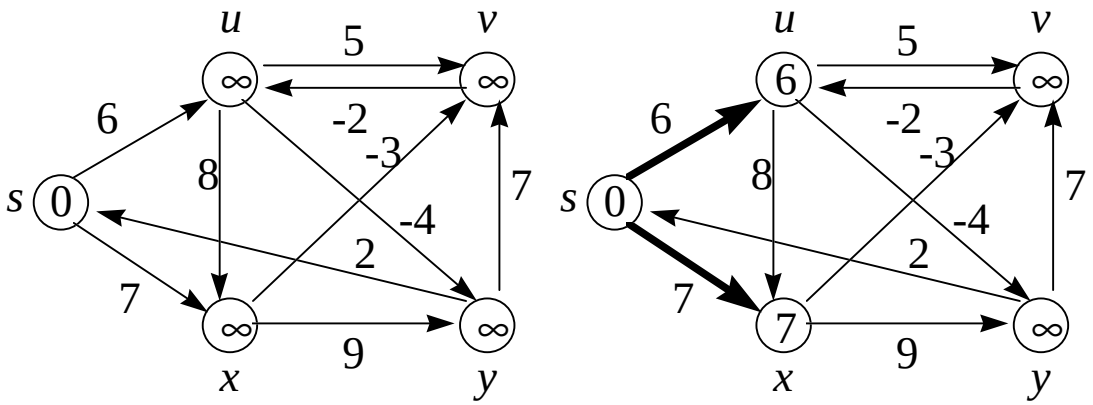
Come il Dijkstra, anche l'algoritmo di Bellman-Ford usa la tecnica del rilassamento, diminuendo progressivamente una stima $d[v]$ del peso di un cammino minimo dalla sorgente *s* ad ogni vertice $v \in V$, finchè non si raggiunge il reale peso di commino minimo $\delta(s, v)$. L'algoritmo restituisce TRUE se e solo se il grafo non contiene un ciclo di peso negativo raggiungibile dalla sorgente.

Dopo aver effettuato la solita inizializzazione, l'algoritmo fa $|V| - 1$ passate sugli archi del grafo: ogni passata è una iterazione del ciclo for delle linee 2-4, e consiste nel rilassare ogni arco del grafo una volta. Dopo le $|V| - 1$ passate, le linee 5-8 controllano l'esistenza di un ciclo di peso negativo e

restituiscono il valore booleano appropriato. L'algoritmo di Bellman-Ford richiede tempo $O(VE)$. Anche in questo caso viene dimostrata la correttezza dell'algoritmo.

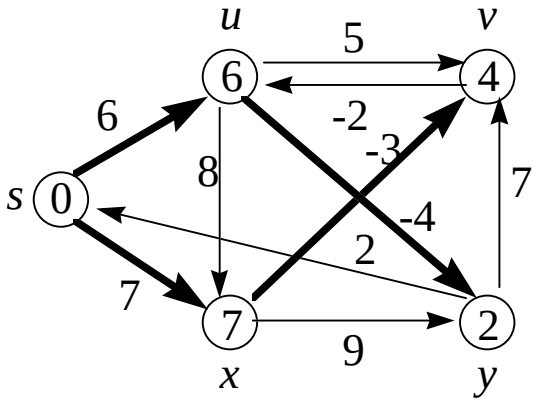
```
Bellman-Ford(G, w, s)
1    Inizializza(G, s)
2    for i ← 1 to |V| - 1
3         do for ogni vertice (u, v) ∈ E
4              do Rilassa(u, v, w)
5    for ogni arco (u, v) ∈ E
6         do if d[v] > d[u] + w(u, v)
7              then return FALSE
8    return TRUE
```

Esempio

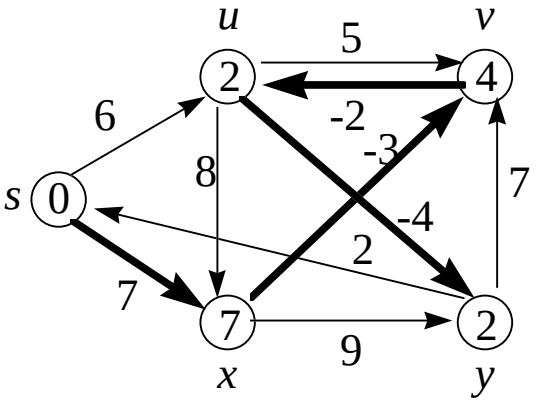


a)

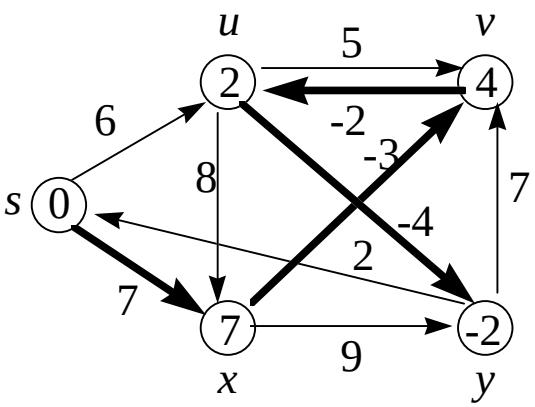
b)



c)



d)



e)

Una esecuzione dell'algoritmo di Bellman-Ford. La sorgente è il vertice *s*. I valori *d* sono mostrati all'interno dei vertici, e gli archi in neretto indicano i valori di π. (a) La situazione subito prima della prima passata sugli archi. (b)-(e) La situazione dopo ogni passata successiva sugli archi. Alla fine l'algoritmo fornisce TRUE.