Cammini di costo minimo: problemi e algoritmi

Marco Liverani*

Dicembre 2017

Individuare su una rete stradale o su una rete di comunicazione la via più breve per andare da un determinato punto ad un altro è senza alcun dubbio un problema concreto assai frequente e studiato da moltissimi anni.

Oggi lo stesso problema può essere applicato alle cosiddette «reti sociali» (o *social network*, utilizzando un gergo ormai molto diffuso) per calcolare il grado di "amicizia" in termini di "distanza" tra due membri della stessa rete sociale: se Mario conosce Gino e Gino è amico di Pina, allora Mario può farsi presentare Pina da Gino; tuttavia se Gino non conosce direttamente Pina, ma conosce una sua cugina Nina, allora per Mario sarà più arduo arrivare a conoscere la bella Pina, perché dovrà farsi presentare Nina da Gino e poi finalmente Pina da Nina. Le cose si complicherebbero però se Mario dovesse venire a sapere che Gino conosce Nina, ma solo superficialmente, perché hanno frequentato insieme un corso di *découpage* molti anni prima; in tal caso Mario, per raggiungere la bella Pina dovrà cercare una strada più "robusta", fatta di relazioni più forti tra le persone.

Per aiutare Mario a coronare il suo sogno d'amore, in queste pagine vengono riassunti alcuni concetti relativi a questo genere di problema, presentando alcuni degli algoritmi classici più famosi per la risoluzione di diversi problemi in questo ambito.

1 Alcuni cenni sulla teoria dei grafi

Il problema che vogliamo affrontare in queste pagine è noto come **problema del cammino minimo**, anzi, più precisamente **problema del cammino di *costo* minimo**. Il problema può essere descritto in modo semplice e privo di imprecisioni, utilizzando un formalismo matematico. Innanzi tutto consideriamo un grafo $G = (V, E)$ con cui viene rappresentato il modello della rete stradale, della rete di comunicazione o della rete sociale su cui vogliamo risolvere il problema.

Un **grafo** è costituito da un insieme finito di vertici $V = \{v_1, v_2, \ldots, v_n\}$ e da un insieme di spigoli, composti da coppie di vertici: $E = \{(u, v) : u, v \in V\}$. Con $n = |V|$ indichiamo il numero di vertici del grafo, mentre con $m = |E|$ indichiamo il numero di spigoli. I vertici del grafo costituiscono i luoghi uniti da strade

*E-mail: liverani@mat.uniroma3.it; ultima modifica: 29 dicembre 2017

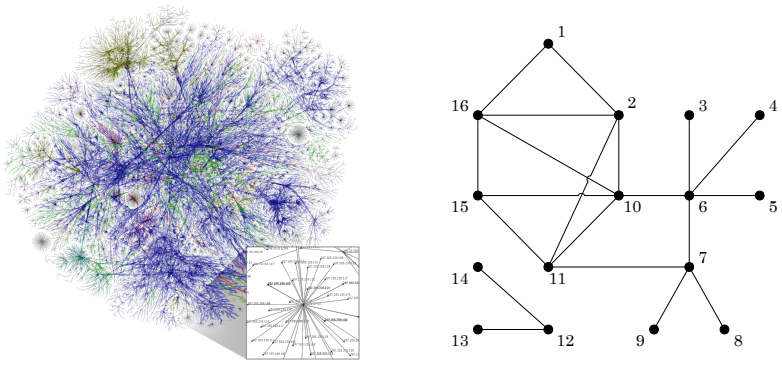


Figura 1: A sinistra il grafo con cui è rappresentata una porzione delle connessioni presenti sulla rete Internet; a destra un semplice grafo con $n = 16$ vertici

su una rete stradale, i computer o le antenne di una rete di telecomunicazione, le persone che fanno parte di una rete sociale; al tempo stesso gli spigoli rappresentano le strade che uniscono i luoghi, le connessione nella rete di telecomunicazione, i legami di amicizia in una rete sociale.

Se il grafo è privo di spigoli, ossia se $E = \emptyset$ e $m = |E| = 0$, allora diremo che il grafo è **nullo**; se al contrario l'insieme E degli spigoli del grafo G è costituito da tutte le possibili coppie di vertici di G (l'insieme degli spigoli corrisponde in questo caso al prodotto cartesiano di V con se stesso: $E = V \times V$), allora diremo che il grafo G è **completo** e lo indicheremo con K_n (grafo completo con n vertici). In questo caso risulta $E = \{(u, v),\ \text{per ogni } u, v \in V\}$.

Grafo nullo e grafo completo

Possiamo disegnare un grafo rappresentando con dei punti i suoi vertici e con delle linee i suoi spigoli. In effetti il disegno di un grafo è utile per comprenderne la struttura se il grafo è piccolo, ossia se ha pochi vertici e pochi spigoli, altrimenti il disegno diventa molto confuso e forse poco utile per ricavare delle informazioni puntuali sul grafo stesso. È bene osservare che il disegno del grafo è una rappresentazione piuttosto arbitraria: nel disegnare un grafo, infatti, non sono importanti la lunghezza degli spigoli o la posizione dei vertici, ma soltanto la connessione dei vertici attraverso gli spigoli. Pertanto uno stesso grafo può essere disegnato su un foglio di carta in modi talmente diversi da renderlo irriconoscibile (vedi ad esempio il grafo rappresentato in Figura 2). Quindi, quando parliamo di un grafo, d'ora in avanti faremo riferimento all'insieme dei suoi vertici V e dei suoi spigoli E, non al suo disegno; il disegno può essere utile per chiarire qualche aspetto o per esemplificare una particolare condizione.

Disegno di un grafo

Un **grafo orientato** è un grafo in cui agli spigoli viene anche assegnato un **verso** per orientarli: in un grafo orientato uno spigolo "esce" da un determinato vertice ed "entra" in un altro vertice. Ad esempio lo spigolo $(u, v) \in E$ esce dal vertice u ed entra nel vertice v. In questo caso, siccome il *verso* dello spigolo è importante, risulta che $(u, v) \neq (v, u)$: ossia lo spigolo che esce da u ed entra in v è differente dallo spigolo che ha il verso opposto e che esce da v ed entra in

Grafo orientato

Figura 2: Due grafi con $n = 5$ vertici: i due grafi sono identici (hanno gli stessi vertici e gli stessi spigoli), ma sembrano differenti solo perché sono disegnati in due modi diversi, disponendo in posizioni diverse i vertici e allungando o accorciando di conseguenza gli spigoli

u. Nel disegnare un grafo orientato per rappresentare gli spigoli si usano delle frecce che consentano anche di evidenziare il verso dello spigolo e non soltanto quali vertici sono gli estremi dello spigolo stesso.

Grafo non orientato

È possibile definire anche **grafi non orientati**, in cui quindi agli spigoli non viene assegnato un verso: in questo caso parlare dello spigolo (u, v) o dello spigolo (v, u) è la stessa cosa; ciò che conta in un grafo *non* orientato sono solo gli estremi di ogni spigolo e non il *verso* dello spigolo stesso. Un grafo non orientato si disegna utilizzando delle linee per unire i vertici estremi di uno spigolo, e non delle frecce.

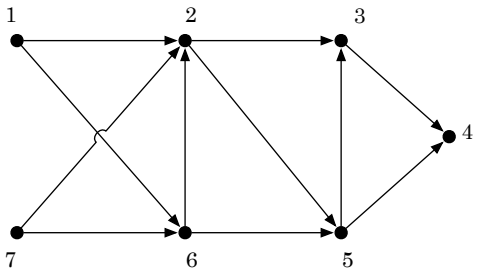
Cammino, lunghezza di un cammino

Un **cammino** p su un grafo $G = (V, E)$ dal vertice u al vertice v è una successione di vertici, uniti a due a due da uno spigolo: $\langle v_1, v_2, \ldots, v_k \rangle$ tale che $u = v_1$, $v = v_k$ e $(v_i, v_{i+1}) \in E$, per ogni $i = 1, 2, \ldots, k-1$. La **lunghezza del cammino** è data dal numero di spigoli che lo compongono, per cui se il cammino passa per k vertici (compresi i due estremi del cammino stesso), allora avrà lunghezza $k-1$. Un cammino di lunghezza 1 è un singolo spigolo che unisce due vertici adiacenti.

Cammino semplice, ciclo, cappio, grafo aciclico

Un cammino che non passa mai due volte per lo stesso vertice è un **cammino semplice**; formalmente possiamo scrivere che il cammino $p : u \rightsquigarrow v$ dal vertice u al vertice v è semplice se $p = \langle v_1, v_2, \ldots, v_k \rangle$ e $v_i \neq v_j$ per ogni $i, j = 1, \ldots, k$ tali che $i \neq j$. Un cammino che inizia e termina nello stesso vertice è un **ciclo**; un ciclo semplice (che non passa due volte per lo stesso vertice intermedio) ha k vertici e k spigoli e viene indicato con C_k. Ad esempio i due grafi in Figura 2 sono due C_5, ossia due cicli con cinque vertici e cinque spigoli. Un ciclo composto da un solo spigolo, che esce e rientra in uno stesso vertice, si chiama **cappio**. Un grafo privo di cicli è un **grafo aciclico**.

Sarà utile più avanti osservare che un cammino semplice da un vertice u ad un altro vertice v in un grafo connesso con n vertici, può essere lungo al massimo $n-1$: se la lunghezza del cammino fosse maggiore o uguale a n, vorrebbe dire che il cammino passa almeno due volte per uno stesso vertice e quindi il cammino non è semplice.



$1 < 7 < 6 < 2 < 5 < 3 < 4$

$7 < 1 < 6 < 2 < 5 < 3 < 4$

Figura 3: Un grafo orientato aciclico: i vertici 1 e 7 sono sorgenti, il vertice 4 è un pozzo; questo grafo ammette due ordinamenti topologici distinti: $1 < 7 < 6 < 2 < 5 < 3 < 4$ e $7 < 1 < 6 < 2 < 5 < 3 < 4$

Grafo connesso, grafo fortemente connesso

Un grafo non orientato è **connesso** se esiste almeno un cammino che unisce ogni coppia di vertici: per ogni $u, v \in V$ esiste almeno un $p : u \rightsquigarrow v$. Naturalmente in un grafo non orientato, se esiste il cammino $p : u \rightsquigarrow v$, allora esiste anche il cammino inverso $p' : v \rightsquigarrow u$. In un grafo orientato, invece, se esiste il cammino $p : u \rightsquigarrow v$ non è detto che esista anche il cammino inverso da v ad u. Un grafo orientato è **fortemente connesso** se per ogni $u, v \in V$ esiste almeno un cammino da u a v (e viceversa). In Figura 1, il grafo rappresentato sulla destra non è connesso: infatti, ad esempio, non esiste nessun cammino per andare dal vertice 12 al vertice 1; il grafo è costituito da due **componenti connesse**, la prima costituita dai vertici $\{12, 13, 14\}$ e la seconda da tutti gli altri: ciascuna delle due componenti costituisce un sottografo connesso.

Grafo orientato aciclico, sorgente, pozzo

Un **grafo** G **orientato e aciclico** (in inglese DAG, *directed acyclic graph*) non è fortemente connesso. Infatti se G non contiene cicli, esiste almeno un vertice del grafo privo di spigoli entranti ed almeno un vertice privo di spigoli uscenti: chiameremo il primo vertice **sorgente** ed il secondo **pozzo** di G. Evidentemente non esiste nessun cammino dal pozzo alla sorgente, per cui il grafo G non è fortemente connesso.

Ordinamento topologico dei vertici

In un grafo orientato aciclico $G = (V, E)$ è possibile definire almeno un **ordinamento topologico** sull'insieme dei vertici V: un ordinamento degli elementi di V tale che se $(u, v) \in E$ allora $u < v$ nell'ordinamento topologico. Questa definizione su un grafo orientato aciclico è consistente: se il grafo non fosse orientato allora per ogni spigolo $(u, v) \in E$ esisterebbe anche lo spigolo di verso opposto $(v, u) \in E$ e quindi non si potrebbe definire una relazione d'ordine tra u e v. Se il grafo contenesse un ciclo, ad esempio il ciclo $v_1 \to v_2 \to v_3 \to v_1$, la relazione d'ordine risulterebbe contraddittoria: $v_1 < v_2$, $v_2 < v_3$, $v_3 < v_1$, quindi $v_1 < v_1$! Su un grafo orientato e aciclico invece la definizione è ben posta e questi casi contraddittori non possono verificarsi.

Albero

Un **albero** è un grafo connesso e aciclico. Se si sceglie un vertice qualsiasi $s \in V$ dell'albero e si impone l'orientazione degli spigoli naturale, in modo che uscendo dal vertice s siano orientati per allontanarsi da tale vertice, otterremo

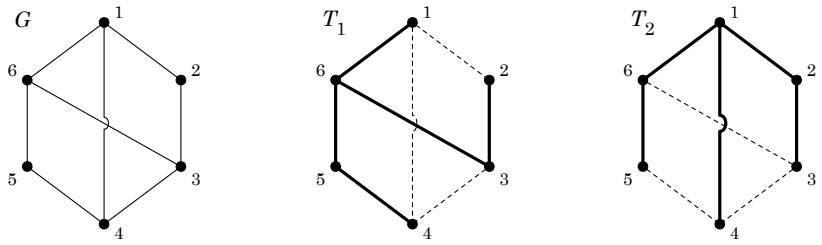


Figura 4: Un grafo G e due suoi alberi ricoprenti T_1 e T_2

un albero orientato con radice in s. In un albero orientato con radice, il solo vertice privo di spigoli entranti è la radice s; i vertici privi di spigoli uscenti si chiamano **foglie** dell'albero.

Sottografo e albero ricoprente

Dato un grafo connesso $G = (V, E)$, un **sottografo** di G è un grafo $G' = (V', E')$ tale che $V' \subseteq V$ ed $E' \subseteq E$. Un **albero ricoprente** (in inglese *spanning tree*) di G è un sottografo $T = (V, E')$ che contiene tutti i vertici di G e che sia un albero, ossia un grafo connesso e privo di cicli. Se G non è connesso non è possibile definire un suo albero ricoprente, ma è possibile definire un albero ricoprente per ogni componente connessa di G: l'unione di tali alberi ricoprenti prende il nome di ***spanning forest***. In generale, dato un grafo G connesso, esistono più alberi ricoprenti per G, a meno che anche G non sia un albero: in tal caso l'albero ricoprente è unico e coincide con lo stesso G.

Distanza tra due vertici, diametro e centro di un grafo

La **distanza** tra due vertici u e v di G è data dalla lunghezza del cammino più breve che collega u a v; se non esiste nessun cammino che unisce i due vertici, allora diremo che la distanza tra di loro è infinita. Il **diametro di un grafo** è dato dalla massima distanza tra due vertici del grafo stesso; il **centro di un grafo** è dato dal sottoinsieme dei vertici che hanno distanza minima dal vertice più lontano. Ad esempio nel grafo G rappresentato in Figura 4 la distanza tra il vertice 1 e il vertice 5 è 2, mentre la distanza tra 2 e 5 è 3. Anche il diametro di G è 3, visto che tale è la massima distanza tra due vertici del grafo. Il centro del grafo è costituito dal sottoinsieme di vertici $\{1, 3, 4, 6\}$: per questi quattro vertici, infatti, la massima distanza da un altro vertice di G è 2, mentre la massima distanza da un altro vertice per i vertici 2 e 5 è 3, e per questo motivo tali vertici non fanno parte del centro del grafo.

2 Cammini di lunghezza minima

Dato un grafo $G = (V, E)$, il primo problema che ci proponiamo di affrontare è quello di individuare i cammini di lunghezza minima per raggiungere da un determinato vertice $s \in V$ di partenza, tutti gli altri vertici del grafo. Un cammino p di lunghezza minima da u a v è necessariamente un cammino semplice: se non fosse semplice, allora passerebbe almeno due volte per uno stesso vertice

e quindi non sarebbe più un cammino di lunghezza minima, dal momento che, eliminando il ciclo, introdotto passando almeno due volte per un determinato vertice, si otterrebbe un cammino più breve da u a v. Naturalmente, fissato un vertice s di partenza, è possibile raggiungere con un cammino un altro vertice v se questo fa parte della stessa componente connessa a cui appartiene anche s e se il verso degli spigoli (nel caso in cui G sia un grafo orientato) lo consente. Se queste condizioni sono soddisfatte, allora è piuttosto facile calcolare i camini di lunghezza minima (ossia composti dal minor numero di spigoli) per raggiungere ogni altro vertice del grafo a partire da s.

Visita in ampiezza di un grafo, algoritmo BFS

Per risolvere il problema è sufficiente eseguire una **visita in ampiezza** del grafo G, partendo dal vertice s. La visita in ampiezza è un procedimento di esplorazione del grafo eseguito a partire da un determinato vertice iniziale (s nel nostro caso), utilizzando il criterio di allontanarsi sempre poco per volta dalla sorgente della visita: vengono visitati i vertici di distanza k dalla sorgente s solo dopo aver visitato tutti i vertici con distanza minore di k dalla sorgente. Riportiamo nell'Algoritmo 1 la pseudo-codifica dell'algoritmo **BFS** (*breadth-first search*) per la visita in ampiezza di un grafo G.

Algoritmo 1 BFS(G, s)

Input: Il grafo G ed una sorgente $s \in V(G)$
Output: Un albero ricoprente di G con radice in s

```
1: sia Q una coda
2: sia T un albero
3: per ogni v ∈ V(G) ripeti
4:     marca v di colore bianco, poni d(v) := ∞, poni π(v) := null
5: fine-ciclo
6: marca s di colore grigio, poni d(s) := 0, Q := {s}, T := ({s}, ∅)
7: fintanto che Q ≠ ∅ ripeti
8:     estrai un elemento u da Q
9:     per ogni v adiacente ad u ripeti
10:        se v è di colore bianco allora
11:            marca v di colore grigio
12:            accoda v a Q
13:            aggiungi il vertice v e lo spigolo (u, v) all'albero T, π(v) := u
14:            d(v) := d(u) + 1
15:        fine-condizione
16:    fine-ciclo
17:    marca u di colore nero
18: fine-ciclo
```

Fissato il vertice di partenza $s \in V$, la soluzione del problema prodotta dall'algoritmo è un albero ricoprente T di G o della componente connessa di G a cui appartiene il vertice s, con radice in s.

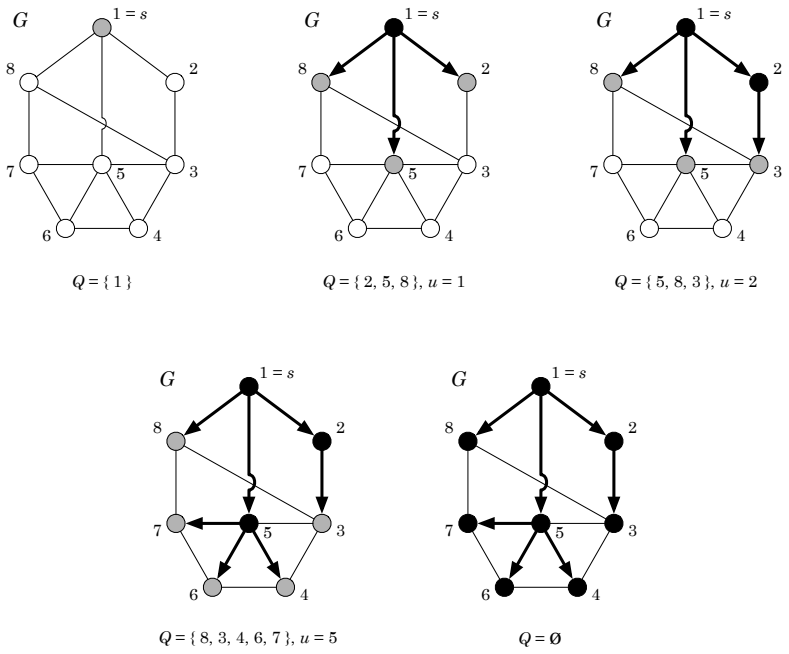


Figura 5: Un esempio di visita in ampiezza del grafo G con l'algoritmo BFS assumendo il vertice $s = 1$ come sorgente della visita

L'algoritmo utilizza alcune semplici strutture dati. La prima è una **coda** Q, usata per tenere memoria dei vertici del grafo nell'ordine in cui questi vengono incontrati nel corso della visita. Ad ogni iterazione del ciclo principale dell'algoritmo viene estratto un vertice u da Q (riga 8); tutti i vertici v adiacenti a u che non siano già stati incontrati nel corso della visita, vengono accodati in Q, per essere presi in esame nei passi successivi dell'algoritmo. Ogni vertice incontrato nel corso della visita entra ed esce esattamente una volta nella coda Q e l'algoritmo termina quando la coda Q è vuota (condizione a riga 7). Per questo motivo il ciclo alle righe 7–18 esegue al massimo n iterazioni: tante quante sono i vertici del grafo.

La coda Q con cui si memorizza l'ordine di visita dei vertici

Per evitare di ripassare più volte per gli stessi vertici e gli stessi spigoli entrando in un *loop* senza fine, l'algoritmo contrassegna i vertici del grafo con un **colore**, che ne caratterizza lo stato di visita: i vertici sono colorati di bianco se non sono ancora stati visitati (inizialmente tutti i vertici del grafo sono bianchi: righe 3–5); vengono colorati di grigio quando vengono incontrati per la prima volta ed inseriti nella coda (righe 11–12); infine i vertici vengono colorati di nero (riga 17) quando la loro visita è conclusa e tutti i vertici adiacenti sono stati inseriti nella coda Q o sono stati visitati. Utilizzando i colori che contrassegnano i vertici del grafo, verranno presi in esame come vertici ancora da visitare solo i vertici bianchi (riga 10), evitando così di visitare più volte lo stesso vertice.

Il colore dei vertici ne caratterizza lo stato di visita

Nel corso della visita del grafo, l'algoritmo raggiungere un vertice generico v individuandolo come vertice non ancora visitato (quindi di colore bianco) adiacente al vertice u su cui sta operando l'algoritmo di visita (righe 9 e 10). In tal caso lo spigolo (u, v) diventa l'ultimo spigolo del cammino più breve da s fino al vertice v e viene quindi aggiunto all'albero T prodotto dall'algoritmo BFS; per tracciare il percorso da s a v, memorizziamo in $\pi(v)$ il **predecessore** di v nel cammino, ossia il **padre** di v nell'albero T (riga 13).

$\pi(v)$ predecessore di v nel cammino più breve da s a v

Siccome per arrivare al generico vertice v si percorre lo spigolo (u, v), la **distanza** di v da s calcolata con l'istruzione a riga 14, sarà uguale alla distanza di u da s maggiorata di un'unità: $d(v) := d(u) + 1$. In questo modo, partendo da $d(s) = 0$ (riga 6: la distanza della sorgente da se stessa è nulla), è possibile calcolare incrementalmente la distanza di ogni vertice dalla sorgente s della visita.

$d(v)$ distanza di v da s

Facendo riferimento all'esempio riportato in Figura 5, possiamo vedere che inizialmente la coda Q contiene il solo vertice s, che viene quindi estratto per primo dalla coda: il primo vertice u preso in esame è proprio il vertice $s = 1$. Da quel vertice vengono presi in esame i suoi adiacenti, che vengono colorati di grigio e inseriti nella coda Q nell'ordine con cui vengono individuati nella lista di adiacenza di u: supponiamo ad esempio che $Q = \{2, 5, 8\}$. Inserendo ciascun vertice nella coda, viene anche calcolato il predecessore di ciascuno ($\pi(2) = 1, \pi(5) = 1, \pi(8) = 1$) e la distanza dei vertici adiacenti ad u, incrementando di 1 la distanza di u ($d(2) = 1, d(5) = 1, d(8) = 1$). Viene così completata la visita del vertice $u = 1$, che viene colorato di nero (riga 17).

Quindi viene eseguita una nuova iterazione del ciclo principale dell'algoritmo (righe 7–18) e viene estratto da Q il vertice u in prima posizione nella coda: nell'esempio risulta $u = 2$. Vengono visitati i vertici adiacenti a $u = 2$ che non siano già stati visitati in precedenza (ciclo 9–16): i vertici adiacenti a $u = 2$ sono $v = 1$ e $v = 3$; il primo è già di colore nero, mentre il secondo è ancora bianco, per cui viene visitato colorandolo di grigio, inserendolo in fondo alla coda Q e calcolando il suo predecessore ($\pi(3) = 2$) e la sua distanza da $s = 1$ ($d(3) = d(2) + 1 = 2$).

Nell'iterazione successiva del ciclo principale dell'algoritmo viene estratto da Q il vertice $u = 5$ e vengono visitati i suoi vertici adiacenti non ancora raggiunti ai passi precedenti dell'algoritmo (ancora bianchi): i vertici $u = 4, u = 6, u = 7$ vengono accodati in Q, colorati di grigio e per tutti e tre viene calcolato il predecessore ($\pi(4) = \pi(6) = \pi(7) = 5$) e la loro distanza dalla sorgente ($d(4) = d(6) = d(7) = d(5) + 1 = 2$).

A questo punto tutti i vertici del grafo sono stati raggiunti dalla visita e per ognuno è stato calcolato il predecessore nell'albero di visita e la distanza da s. L'algoritmo prosegue le sue operazioni fino a quando la coda Q non sarà stata svuotata completamente, tuttavia in queste iterazioni rimanenti non verrà individuato nessun vertice bianco e quindi l'algoritmo termina dopo altre cinque iterazioni senza modificare le distanze o l'albero di visita, fino a quando la coda non risulterà vuota e tutti i vertici saranno stati colorati di nero.

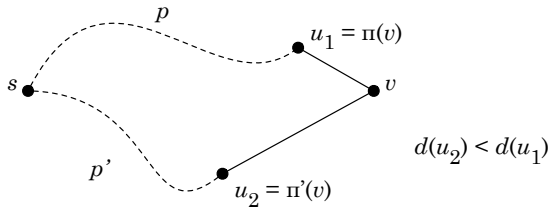


Figura 6: Due cammini p e p' permettono di raggiungere il vertice v da s: se il predecessore di v su p' è più vicino ad s del predecessore di v su p, allora il cammino trovato da BFS è p' e non p

Output prodotto da BFS e visualizzazione del cammino da s a v

Al termine della visita abbiamo ottenuto la distanza $d(v)$ di ogni vertice $v \in V$ dalla sorgente s e il predecessore $\pi(v)$ di ciascun vertice v nel cammino di lunghezza minima da s a v. Con quest'ultima informazione è possibile ricostruire il cammino minimo da s a v, attraverso un semplice procedimento ricorsivo presentato nell'Algoritmo 2.

Algoritmo 2 STAMPACAMMINO(v, π)

Input: Il vertice v e l'insieme dei predecessori $\pi(1), \ldots, \pi(n)$ calcolati con BFS
Output: Il cammino da s a v calcolato da BFS

```
1: se π(v) ≠ null allora
2:     STAMPACAMMINO(π(v), π)
3: fine-condizione
4: scrivi v
```

È interessante osservare che spostando l'istruzione 4 all'inizio dell'algoritmo, prima della condizione, si ottiene la visualizzazione del cammino da v fino ad s, anziché da s fino a v.

Correttezza dell'algoritmo BFS

I cammini prodotti dall'algoritmo BFS sono necessariamente dei cammini minimi: se il cammino $p : s \rightsquigarrow v$ prodotto da BFS non fosse minimo, allora esisterebbe un altro cammino $p' : s \rightsquigarrow v$ di lunghezza inferiore (vedi Figura 6). Ma questo non è possibile, perché significherebbe che il predecessore di v in p, $\pi(v)$, è più distante del predecessore di v in p', $\pi'(v)$, da s. Se questo fosse vero, allora nel passo 9 dell'Algoritmo 1 il vertice v sarebbe stato incontrato come vertice adiacente a $\pi'(v)$ e non come vertice adiacente a $\pi(v)$ e quindi il cammino p non sarebbe stato prodotto dall'algoritmo BFS.

Naturalmente, come si evince anche osservando attentamente l'esempio proposto in Figura 5, su un grafo G il cammino di lunghezza minima da s a v, se esiste, può non essere unico. Ad esempio, facendo riferimento al grafo in figura, esiste il cammino $p = \langle 1,2,3 \rangle$ da 1 a 3, di lunghezza 2, ma esistono anche i cammini $p' = \langle 1,8,3 \rangle$ e $p'' = \langle 1,5,3 \rangle$, entrambi di lunghezza 2. L'algoritmo BFS individua per ogni vertice del grafo un cammino minimo, tra i tanti che possono esistere nel grafo, in base all'ordine con cui prende in esame i vertici adiacenti al

vertice u nel ciclo a riga 9 dell'Algoritmo 1. In ogni caso la distanza calcolata dall'algoritmo è un *invariante*, una quantità indipendente dal particolare cammino di lunghezza minima.

Complessità computazionale dell'algoritmo BFS

L'algoritmo BFS è molto efficiente ed ha una complessità computazionale lineare nel numero dei vertici e degli spigoli del grafo G: $O(n+m)$. Infatti nella fase di inizializzazione (righe 1–6 dell'Algoritmo 1 di pagina 6) vengono ripetute per n volte alcune operazioni elementari (di complessità $O(1)$): quindi $O(n)$ operazioni. Nel ciclo principale dell'algoritmo (righe 7–18) l'istruzione ripetuta più volte è la condizione di riga 10, che viene eseguita esattamente m volte se il grafo è orientato o $2m$ volte se il grafo non è orientato. Infatti il ciclo esterno (righe 7–18) viene eseguito per ogni vertice u del grafo (ogni vertice entra nella coda al massimo una volta, ad ogni iterazione del ciclo viene estratto un solo vertice dalla coda e il ciclo termina solo quando la coda è completamente vuota) e il ciclo interno (righe 9–16) viene ripetuto per ogni spigolo uscente da u. Quindi vengono ripetute complessivamente tante iterazioni quanti sono gli spigoli uscenti da ogni vertice del grafo, quindi tante iterazioni quanti sono gli spigoli del grafo: $O(m)$. Sommando la complessità delle due fasi dell'algoritmo (inizializzazione e ciclo principale) si ottiene la complessità di BFS: $O(n+m)$.

3 Cammini di costo minimo da una singola sorgente

Quanto visto nelle pagine precedenti è sufficiente per individuare il cammino di lunghezza minima da un vertice s ad ogni altro vertice del grafo. La funzione obiettivo da minimizzare, in questo caso, è la lunghezza del cammino, il numero di spigoli di cui è composto il cammino. Tuttavia, se il problema concreto che dobbiamo affrontare e che possiamo modellizzare con un grafo, richiede di attribuire un *costo* o un *peso* o una *lunghezza* agli spigoli del grafo, la questione cambia radicalmente e un semplice algoritmo di visita in ampiezza non è più sufficiente per risolvere il problema.

Ad esempio, su una rete stradale le strade non hanno certo tutte la stessa lunghezza o il medesimo tempo di percorrenza in una determinata ora del giorno: è possibile attribuire un peso o un costo inferiore alle strade che si percorrono più rapidamente e un peso o un costo maggiore alle strade che si percorrono più lentamente. Anche su una rete di telecomunicazione esistono connessioni più lente o più costose ed altre più veloci o meno costose. Sul grafo $G = (V, E)$ con cui rappresenteremo la rete stradale o la rete di telecomunicazione, assegneremo un **peso** $w(u, v) \in \mathbb{R}$ ad ogni spigolo $(u, v) \in E$, proprio per rappresentare il costo o il tempo di percorrenza di una strada o di un tratto di connessione su una rete informatica. In questo modo è possibile definire il **costo di un cammino** $p : u \rightsquigarrow v$ come somma dei costi degli spigoli che lo compongono: $W(p) = \sum_{(v_i, v_{i+1}) \in p} w(v_i, v_{i+1})$. In questa sezione indicheremo con $\delta(u, v)$ il costo del cammino di costo minimo da u a v, che non è necessariamente il cammino più breve, quello con il minor numero di spigoli, ma il meno costoso. Se

Pesi o costi assegnati agli spigoli del grafo

Costo di un cammino e costo del cammino minimo $\delta(v)$

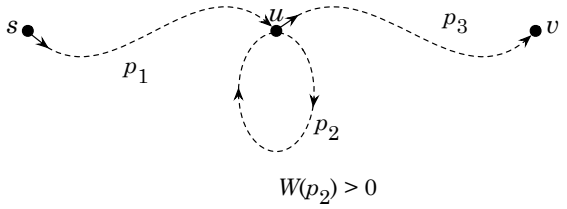


Figura 7: Un cammino $p : s \rightsquigarrow v$ ottenuto componendo il cammino p_1, il ciclo p_2 ed il cammino p_3

la sorgente, il punto di partenza dei cammini, è fissato e non c'è ambiguità nella notazione, indicheremo il costo minimo del cammino dalla sorgente al vertice v con $\delta(v)$.

Problema del cammino di costo minimo da una sorgente singola

Nel problema del **cammino di costo minimo da una singola sorgente**, dato un grafo $G = (V, E)$ con pesi $w(u, v) \in \mathbb{R}$ assegnati agli spigoli (u, v) del grafo ed un vertice sorgente $s \in V$, si chiede di calcolare un cammino $p : s \rightsquigarrow v$ per ogni vertice v del grafo, tale che $W(p) = \delta(v)$, ossia tale che il costo del cammino sia minimo.

Spigoli di costo negativo, cicli di costo negativo

È possibile che alcuni spigoli del grafo abbiano un peso negativo: $w(u, v) < 0$; tuttavia, perché il problema sia ben definito, è necessario che, pur ammettendo spigoli di costo negativo, nel grafo non siano presenti cicli di costo negativo, ossia cicli in cui la somma dei costi degli spigoli sia negativa. Infatti, se esistesse un ciclo di costo negativo, allora percorrendo più volte tale ciclo, si potrebbe ridurre senza fine il costo complessivo di un cammino: il cammino di costo minimo in tal caso non è ben definito. Al contrario, se non si ammettono cicli di costo negativo allora ogni cammino di costo minimo sarà privo di cicli: come abbiamo già osservato nelle pagine precedenti, infatti, passando due volte per uno stesso vertice, percorrendo un ciclo il cui costo è positivo, non si fa altro che aumentare il costo del cammino, mentre escludendo il ciclo si otterrebbe comunque un cammino di costo inferiore che collega gli stessi vertici estremi.

In riferimento al disegno in Figura 7, il cammino non semplice $p : s \rightsquigarrow v$ può essere suddiviso in tre sotto-cammini: $p_1 : s \rightsquigarrow u$, $p_2 : u \rightsquigarrow u$, $p_3 : u \rightsquigarrow v$. Il costo del cammino p è dato dalla somma del costo delle tre componenti: $W(p) = W(p_1) + W(p_2) + W(p_3)$. Se il ciclo p_2 ha un peso negativo, allora percorrendolo più volte si potrà ridurre all'infinito il costo del cammino p, che risulta così indefinito. Se invece il ciclo p_2 ha un costo positivo, $W(p_2) > 0$, allora il cammino $p' : s \rightsquigarrow v$, ottenuto unendo p_1 e p_3 ed escludendo p_2 ha un costo certamente inferiore a quello di p. Per questo motivo i cammini di costo minimo che dobbiamo costruire sono tutti privi di cicli.

Principio di rilassamento

Il problema del cammino di costo minimo su un grafo pesato può essere risolto con algoritmi differenti basati sul medesimo «principio di rilassamento», che possiamo esprimere in questi termini: supponiamo che un cammino

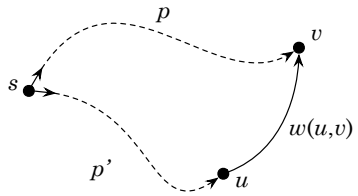


Figura 8: Un cammino $p : s \rightsquigarrow v$ ed un cammino alternativo da s a v ottenuto componendo il cammino $p' : s \rightsquigarrow u$ e lo spigolo (u, v) di costo $w(u, v)$

$p : s \rightsquigarrow v$ individuato durante l'esecuzione di un algoritmo abbia un costo pari a $W(p)$; supponiamo inoltre che esista uno spigolo $(u, v) \in E$ con costo $w(u, v)$ e che esista un cammino $p' : s \rightsquigarrow u$ con costo $W(p')$. Se $W(p') + w(u, v) < W(p)$ allora, per andare da s a v conviene scegliere il cammino che passa per u, piuttosto che il cammino p calcolato in precedenza.

3.1 Algoritmo di Dijkstra

Edsger W. Dijkstra

Edsger Wybe Dijkstra (1930–2002) è stato un celebre informatico olandese, che ha contribuito in modo determinante allo sviluppo della teoria degli algoritmi. Si è laureato in Fisica nel 1956, ma già alcuni anni prima di conseguire la laurea il suo interesse si era spostato sulla programmazione dei computer, che proprio in quegli anni stava nascendo come disciplina autonoma, aprendo in ambito scientifico numerosi interessanti problemi relativi alla calcolabilità. Dopo aver lavorato come ricercatore presso la Burroughs Corporation all'inizio degli anni '70, ha insegnato informatica a lungo all'Università di Eindhoven in Olanda e successivamente all'Università del Texas ad Austin negli Stati Uniti. Nel 1972 gli è stato conferito il Premio Turing, per i suoi importantissimi contributi all'informatica. Notoria è la sua grande abilità didattica, molto apprezzata dai suoi studenti, messa in pratica nei molti anni di insegnamento universitario, di cui oggi abbiamo testimonianza nei suoi numerosissimi scritti, molto curati ed efficaci, di cui esiste on-line un archivio completo.[1]

Algoritmo di Dijkstra

In particolare Dijkstra si rese famoso per i suoi articoli sulla programmazione strutturata, ma il lavoro che lo rese noto fu soprattutto un algoritmo che porta il suo nome, per il calcolo dei cammini di costo minimo da una singola sorgente su un grafo con pesi assegnati agli spigoli. L'algoritmo di Dijkstra richiede che i pesi $w(u, v)$ assegnati agli spigoli del grafo siano positivi; tale vincolo non è particolarmente restrittivo, dal momento che, se dovessero essere presenti spigoli di costo negativo, sarebbe sufficiente sommare al peso di ogni spigolo, il peso minimo assegnato ad uno spigolo del grafo, per ricondursi ad un problema equivalente, privo di pesi negativi sugli spigoli.

[1] L'archivio degli articoli di E.W. Dijkstra si trova on-line sul sito dell'Università del Texas, all'indirizzo «http://www.cs.utexas.edu/users/EWD/».

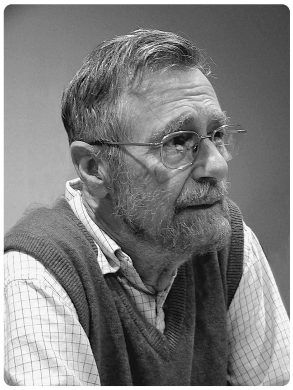


Figura 9: Edsger Wybe Dijkstra (1930–2002)

L'Algoritmo 3 propone una pseudo-codifica dell'algoritmo di Dijkstra. L'algoritmo si basa sul «principio del rilassamento» descritto in precedenza e su una **coda di priorità**, ossia una struttura dati simile ad una coda, ma che per la realizzazione dell'operazione di estrazione di un elemento si basa su un indice di priorità assegnato agli elementi presenti nella coda e non semplicemente sull'ordine di ingresso nella struttura dati. È possibile rappresentare una coda di priorità con una lista, un array o un **heap**. Indicando con $n = |V(G)|$ il numero di vertici del grafo, la rappresentazione più efficiente si ha con quest'ultima struttura dati, che ci permette di implementare le operazioni di inserimento di un elemento, estrazione dell'elemento con la massima priorità e aggiornamento dell'indice di priorità di un elemento, con una complessità di $O(\log_2 n)$. Con un array o una lista, invece, l'operazione di estrazione dell'elemento con la massima priorità ha un costo di $O(n)$, a meno di non mantenere ordinata la struttura dati, nel qual caso, però, è l'operazione di inserimento che ha una complessità lineare di $O(n)$.

Coda di priorità

Strutture dati per la rappresentazione della coda di priorità

Un ***heap*** è un albero binario completo, in cui l'elemento con la priorità più alta è posto come radice dell'albero e, per ogni vertice v dell'heap, risulta che la priorità del padre è maggiore o uguale alla priorità dei figli. Naturalmente è possibile adattare l'indice di priorità al problema su cui si sta impiegando l'heap: nel nostro caso l'indice di priorità è dato dalla stima del costo del cammino minimo da s a v; avremo quindi una coda di priorità con l'indice più basso nella radice dell'albero e i figli con indici di valore crescente. Visto che l'heap è un albero binario completo, è possibile rappresentarlo con un array h, ponendo la radice dell'albero in h_1; se h_i è un elemento generico dell'heap, il figlio sinistro di h_i viene memorizzato in h_{2i}, mentre il figlio destro sarà memorizzato in h_{2i+1}; di conseguenza, il padre di h_i si trova in $h_{i/2}$. Facendo riferimento alla Figura 10, un nuovo elemento verrebbe aggiunto inizialmente come figlio destro del vertice con priorità 44 e quindi collocato nell'array in h_{13}; sulla base dell'indice di priorità del nuovo elemento, questo verrebbe poi confrontato (ed eventualmen-

Operazioni per la gestione di un heap

Algoritmo 3 DIJKSTRA($G = (V, E)$, $w : E \to \mathbb{R}^+$, $s \in V$)

Input: Un grafo pesato (G, w) e un vertice $s \in V$ scelto arbitrariamente come sorgente della visita

Output: Un albero di cammini minimi con radice in s costituito dai cammini di peso minimo che da s consentono di raggiungere ogni vertice $v \in V$

1: **per ogni** $v \in V(G)$ **ripeti**
2: $\delta(v) := \infty$; $\pi(v) :=$ null
3: **fine-ciclo**
4: $\delta(s) := 0$
5: sia Q una coda di priorità; $Q := V(G)$
6: **fintanto che** $Q \neq \emptyset$ **ripeti**
7: sia u il vertice di G estratto da Q con $\delta(u)$ minimo in Q
8: **per ogni** v adiacente a u **ripeti**
9: **se** $\delta(v) > \delta(u) + w(u, v)$ **allora**
10: $\delta(v) := \delta(u) + w(u, v)$, $\pi(v) := u$, aggiorna la posizione di v nella coda di priorità Q
11: **fine-condizione**
12: **fine-ciclo**
13: **fine-ciclo**
14: **restituisci** π

te scambiato, se dovesse risultare un indice più basso) con il padre, il padre del padre, ecc., fino ad arrivare, eventualmente, alla radice dell'heap. Complessivamente, quindi, l'inserimento di un nuovo elemento nell'heap richiede al massimo $O(\log_2 n)$ confronti e scambi. L'estrazione dell'elemento con indice più basso, che si trova necessariamente nella radice dell'heap, comporta la sostituzione di questo elemento con l'ultima foglia a destra, in modo da non corrompere la struttura di albero binario completo; dopo aver spostato l'ultima foglia al posto della radice, questa dovrà essere confrontata ed eventualmente scambiata con il minore dei suoi figli ed i suoi discendenti fino a trovare una collocazione che soddisfa le proprietà dell'heap. Anche in questo caso, quindi, dovranno essere eseguiti al più $O(\log_2 n)$ confronti ed eventuali scambi.

L'algoritmo di Dijkstra utilizza una coda di priorità per memorizzare i vertici del grafo che devono essere raggiunti dai cammini che vengono costruiti dall'algoritmo stesso. La priorità è costituita dal costo del cammino minimo trovato per raggiungere il vertice partendo dalla sorgente s. Quindi hanno priorità più alta i vertici più "vicini" alla sorgente (collegati ad s da un cammino meno costoso), mentre i vertici più "distanti" si troveranno più in basso nella struttura ad albero dell'heap.

Algoritmo di Dijkstra: fase di inizializzazione

Inizialmente (righe 1–5 dell'Algoritmo 3 a pagina 14) l'algoritmo di Dijkstra inizializza le strutture dati con cui si memorizza la stima del costo del cammino minimo per andare da s ad ogni vertice $v \in V$ e il predecessore di ciascun vertice nel cammino di costo minimo che parte da s: con $\delta(v)$ si indica il costo del cam-

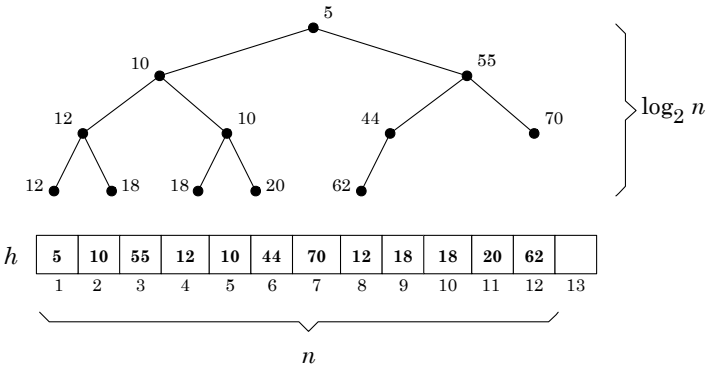


Figura 10: Un heap rappresentato come un albero binario completo e l'array h corrispondente

mino di costo minimo da s a v, mentre con $\pi(v)$ si indica il predecessore di v nel cammino che parte da s. Siccome all'inizio non si conoscono tali cammini, si pone $\delta(v) = \infty$ e $\pi(v) =$ null per ogni $v \in V$, ad eccezione che per il vertice s, per cui si pone $\delta(s) = 0$.

Quindi si inseriscono nella coda di priorità Q tutti i vertici del grafo (riga 5), utilizzando come indice di priorità proprio il valore della stima del costo del cammino minimo per raggiungere ogni vertice: siccome $\delta(v) = \infty$ per ogni vertice $v \neq s$ e $\delta(s) = 0$, si otterrà un heap con radice in s e gli altri vertici disposti arbitrariamente sugli altri nodi dell'albero.

L'algoritmo costruisce i cammini di costo minimo con una dinamica analoga a quella adottata dall'algoritmo di visita in ampiezza di un grafo, salvo che per il fatto che in questo caso, utilizzando una coda di priorità, verranno presi in esame prima i vertici per i quali il cammino trovato è di costo minimo, anche se magari la lunghezza dello stesso cammino non è la più piccola: nel problema del cammino di costo minimo, infatti, si vuole identificare il cammino con il minimo costo complessivo, senza curarsi del numero di spigoli che compongono il cammino stesso. Il ciclo principale dell'algoritmo (righe 6–13) viene iterato fino a quando la coda di priorità Q non sarà completamente vuota. Ad ogni iterazione si estrae da Q il vertice con la massima priorità (quindi il vertice u con $\delta(u)$ minimo). Tale vertice è il più "vicino" alla sorgente s tra tutti i vertici che devono ancora essere elaborati. Pertanto il cammino trovato fino a quel punto per il vertice u è certamente un cammino di costo minimo, dal momento che non potrebbe essere migliorato passando per i vertici ancora da elaborare che hanno un costo di cammino minimo maggiore.

Algoritmo di Dijkstra: ciclo principale

Utilizzando il «principio del rilassamento» vengono presi in esame i vertici v adiacenti a u (riga 8) e viene migliorata la stima del costo del cammino minimo nel caso in cui $\delta(v) > \delta(u) + w(u, v)$. In tal caso (riga 10) viene calcolata la nuova stima di costo del cammino minimo per arrivare in v ponendo $\delta(v) = \delta(u) +$

Figura 11: Richard Bellman (1920–1984) a sinistra e Lester Ford Jr. (1927–2017) a destra

$w(u,v)$; tale costo lo si ottiene raggiungendo il vertice v dal vertice u, per cui si memorizza u come predecessore di v nel cammino di costo minimo: $\pi(v) = u$.

Complessità computazionale dell'algoritmo di Dijkstra

La complessità dell'algoritmo di Dijkstra, utilizzando un heap per rappresentare la coda di priorità Q, è $O(m \log_2 n)$. La fase di inizializzazione (righe 1–5) ha una complessità di $O(n)$. Il ciclo principale (righe 6–13) e il ciclo nidificato al suo interno (righe 8–12) iterano per m volte la condizione a riga 9; se tale condizione dovesse risultare sempre vera, ogni volta verrebbe diminuita la stima del costo del cammino minimo per il vertice v e quindi questo dovrebbe essere riposizionato nell'heap, con un costo $O(\log_2 n)$. Quindi il ciclo principale ha una complessità di $O(m \log_2 n)$ che determina la complessità dell'intero algoritmo.

3.2 Algoritmo di Bellman-Ford

Un procedimento alternativo per il calcolo dei cammini di costo minimo da una singola sorgente viene proposto indipendentemente da Richard Bellman nel 1958 e, precedentemente, da Lester Ford nel 1954, in un famoso algoritmo noto con il nome di entrambi.

Lunghezza massima di un cammino di costo minimo

Come l'algoritmo di Dijkstra, anche l'algoritmo di Bellman-Ford si basa sul «principio del rilassamento». Inoltre, la dinamica dell'algoritmo si basa anche sulla considerazione che su un grafo $G = (V, E)$ con $n = |V|$ vertici, un cammino di costo minimo dal vertice sorgente s ad un qualsiasi vertice $v \in V$ non può essere più lungo di $n-1$ spigoli: infatti, se fosse costituito da n o più spigoli, allora necessariamente il cammino conterrebbe un ciclo e dunque non potrebbe essere un cammino di costo minimo; eliminando il ciclo si otterrebbe un cammino equivalente da s a v, di costo inferiore.

Come vedremo tra breve, l'algoritmo di Bellman-Ford è computazionalmente meno efficiente dell'algoritmo di Dijkstra, ma, a differenza di quest'ultimo, può essere applicato anche su grafi con spigoli di costo negativo. Rimane invece il vincolo che il grafo sia privo di cicli di costo negativo, altrimenti, come abbia-

mo visto nelle pagine precedenti, il problema del cammino di costo minimo non risulta ben definito.

L'Algoritmo 4 presenta una pseudo-codifica dell'algoritmo di Bellman-Ford. Anche in questo caso indichiamo con $\delta(v)$ la stima del costo del cammino di costo minimo da s a v, mentre con $\pi(v)$ indichiamo il predecessore di v su tale cammino. In questo modo, mediante l'insieme dei predecessori $\{\pi(v_1), \ldots, \pi(v_n)\}$, potremo ricostruire l'albero con radice nella sorgente s, ottenuto come unione di tutti gli $n-1$ cammini minimi da s ad ogni altro vertice v del grafo.

Algoritmo 4 BELLMAN-FORD($G = (V, E)$, $w : E \to \mathbb{R}$, $s \in V$)

Input: Un grafo $G = (V, E)$ con pesi $w(u, v)$ assegnati agli spigoli e un vertice $s \in V$ scelto arbitrariamente come sorgente della visita
Output: Un albero con radice in s costituito dai cammini di costo minimo che da s raggiungono ogni vertice $v \in V$

```
1:  per ogni v ∈ V ripeti
2:      δ(v) := ∞; π(v) := null
3:  fine-ciclo
4:  δ(s) := 0
5:  per i := 1, 2, ..., n − 1 ripeti
6:      per ogni (u, v) ∈ E ripeti
7:          se δ(v) > δ(u) + w(u, v) allora
8:              δ(v) := δ(u) + w(u, v), π(v) := u
9:          fine-condizione
10:     fine-ciclo
11: fine-ciclo
12: per ogni (u, v) ∈ E ripeti
13:     se δ(v) > δ(u) + w(u, v) allora
14:         c'è un ciclo negativo!
15:     fine-condizione
16: fine-ciclo
```

Inizialmente (righe 1–4) per ogni vertice $v \in V$ si assegna $\delta(v) = \infty$ e $\pi(v) =$ null, tranne che per la sorgente s, che ha naturalmente $\delta(s) = 0$.

Inizializzazione

Il ciclo principale dell'algoritmo (righe 9–11), ripetuto esattamente $n-1$ volte, applicando il «principio del rilassamento» (righe 7–9) valuta se l'uso di ogni spigolo (u, v) del grafo può essere utile per diminuire il costo del cammino da s a v. Siccome ogni spigolo può comparire al massimo una volta nel cammino di costo minimo da s ad ogni vertice v del grafo, dopo aver valutato il contributo di ogni spigolo per $n-1$ volte, sarà stato certamente individuato il cammino di costo minimo per ciascun vertice del grafo. Se invece impiegando ancora una volta uno degli spigoli (u, v) per raggiungere un vertice generico v, si dovesse ridurre ulteriormente il costo del cammino di costo minimo da s a v, allora questo significherebbe che nel grafo G è presente un ciclo di costo minimo, contrav-

Ciclo principale dell'algoritmo

Individuazione della presenza di cicli di costo negativo

$E = \{e_1=(5,3), e_2=(6,5), e_3=(2,6), e_4=(3,4), e_5=(1,2), e_6=(3,6), e_7=(1,6), e_8=(4,5), e_9=(2,3)\}$

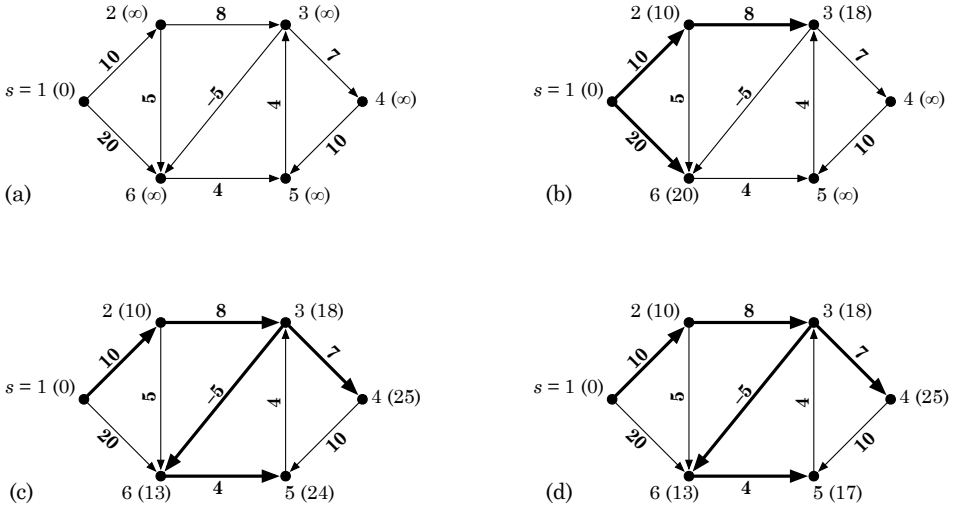


Figura 12: Un esempio di applicazione dell'algoritmo di Bellman-Ford: gli spigoli del grafo nel ciclo 6–10 dell'Algoritmo 4 vengono presi in esame nell'ordine rappresentato in figura

venendo ai vincoli del problema. Questa condizione di "errore" viene rilevata dall'algoritmo di Bellman-Ford con le istruzioni alle righe 12–16.

Complessità computazionale

L'algoritmo di Bellman-Ford su un grafo con n vertici ed m spigoli ha una complessità computazionale di $O(nm)$. Infatti nella fase di inizializzazione (righe 1–4) vengono compiute operazioni elementari di complessità costante $O(1)$ ripetute per n volte. Il corpo principale dell'algoritmo è costituito da due cicli nidificati (righe 5–11 e 6–10) che eseguono rispettivamente n ed m operazioni elementari di confronto e assegnazione, pertanto la complessità di questa parte dell'algoritmo è $O(nm)$, che determina la complessità dell'intero algoritmo.

La complessità di questo algoritmo è certamente superiore a quella dell'algoritmo di Dijkstra, tuttavia l'algoritmo di Bellman-Ford ha almeno due vantaggi: innanzi tutto può essere applicato anche al caso di grafi con pesi negativi assegnati agli spigoli; in secondo luogo è un algoritmo piuttosto semplice e facile da codificare, dal momento che non utilizza nessuna struttura dati particolare o di difficile implementazione.

Un esempio sull'algoritmo di Bellman-Ford

In Figura 12 rappresentiamo un esempio di applicazione dell'algoritmo di Bellman-Ford su un grafo orientato $G = (V, E)$ con $n = |V| = 6$ vertici ed $m = |E| = 9$ spigoli. In figura sono rappresentati i pesi assegnati agli spigoli del grafo e, tra parentesi, accanto all'etichetta di ogni vertice, viene indicata anche la stima $\delta(v)$ del costo del cammino minimo da $s = 1$ al vertice v. Inizialmente $\delta(v) = \infty$ per ogni $v \in V$, ad eccezione della sorgente $s = 1$, per la quale risulta $\delta(s) = 0$.

Il ciclo principale dell'algoritmo (righe 5–11 dell'Algoritmo 4) esegue esattamente $n-1$ iterazioni (5 iterazioni nel nostro esempio); ciascuna iterazione di questo ciclo considera, con il ciclo interno alle righe 6–10, tutti gli spigoli del grafo in un ordine arbitrario. Nel nostro esempio supponiamo di prendere in esame gli spigoli nell'ordine riportato in figura: $e_1 = (5,3), e_2 = (6,5), e_3 = (2,6), e_4 = (3,4), e_5(1,2), e_6 = (3,6), e_7 = (1,6), e_8 = (4,5), e_9 = (2,3)$.

In figura sono rappresentati i cammini di costo minimo con sorgente $s = 1$ calcolati dopo ogni iterazione del ciclo principale, dopo aver preso in esame tutti gli spigoli del grafo. La quarta e la quinta iterazione del ciclo principale dell'algoritmo non modificano in alcun modo la stima del costo dei cammini minimi, che, nel caso dell'esempio, è stata correttamente identificata al termine della terza iterazione (il grafo (d) della Figura 12).

3.3 Algoritmo di Bellman-Ford per grafi orientati aciclici

Per ridurre il numero di iterazioni del ciclo principale dell'algoritmo di Bellman-Ford si può definire in modo più ragionato l'ordine con cui considerare gli spigoli del grafo nel ciclo 6–10.

Ottimizzazione dell'algoritmo di Bellman-Ford esaminando gli spigoli nell'ordine topologico dei vertici da cui escono gli spigoli del grafo

In particolare è inutile considerare uno spigolo (u,v) se prima non sono stati presi in esame gli spigoli entranti in u: infatti se $\delta(u) = \infty$, prendere in esame lo spigolo (u,v) non migliorerà in alcun modo la stima del costo del cammino minimo per raggiungere il vertice v. Dunque un ordine più utile con cui considerare gli spigoli del grafo è dato dall'ordinamento topologico dei vertici da cui escono gli spigoli del grafo. In questo modo considereremo prima gli spigoli entranti in ciascun vertice del grafo e successivamente gli spigoli uscenti che ci conducono ai vertici adiacenti.

Grafi orientati aciclici e ordinamento topologico dei vertici

L'idea è buona, ma può essere applicata solo quando è possibile definire un ordinamento topologico dei vertici del grafo G, ossia solo quando G è un grafo orientato e aciclico (in inglese un DAG, *directed acyclic graph*). Se G è un grafo orientato e aciclico, allora è possibile produrre un ordinamento topologico dei suoi vertici ed elaborarli secondo tale ordine: se $(u,v) \in E$ allora $u < v$. In questo modo otterremo l'Algoritmo 5, una variante dell'algoritmo di Bellman-Ford.

La complessità dell'algoritmo è $O(n+m)$. La fase di inizializzazione (righe 1–4) ha una complessità di $O(n)$; l'ordinamento topologico del grafo può essere eseguito con una visita in profondità di G: tale operazione ha una complessità di $O(n+m)$. Il ciclo principale dell'algoritmo (righe 6–12) esegue n iterazioni, tante quanti sono i vertici u del grafo. Al suo interno viene eseguito un altro ciclo che svolge tante iterazioni quanti sono gli spigoli uscenti dal vertice u e per ogni iterazione esegue delle operazioni di costo costante $O(1)$. Complessivamente quindi le operazioni di costo costante vengono eseguite tante volte quanti sono gli spigoli del grafo, con una complessità di $O(m)$. Quindi l'algoritmo esegue $O(n+n+m+m) = O(n+m)$ operazioni.

Naturalmente il ciclo delle righe 12-16 dell'Algoritmo 4 in questo caso non è presente, dal momento che tali istruzioni servivano a verificare se nel grafo

Algoritmo 5 BELLMAN-FORD-BIS(G, s)

Input: Un grafo orientato e aciclico $G = (V, E)$ con pesi $w(u, v)$ assegnati agli spigoli e un vertice $s \in V$ scelto arbitrariamente come sorgente della visita

Output: Un albero con radice in s costituito dai cammini di costo minimo che da s raggiungono ogni vertice $v \in V$

```
1: per ogni v ∈ V(G) ripeti
2:     δ(v) := ∞; π(v) := NULL
3: fine-ciclo
4: δ(s) := 0
5: esegui l'ordinamento topologico di V(G)
6: per ogni u ∈ |V(G)| in ordine topologico ripeti
7:     per ogni v ∈ N(u) ripeti
8:         se δ(v) ≥ δ(u) + w(u, v) allora
9:             δ(v) := δ(u) + w(u, v), π(v) := u
10:        fine-condizione
11:    fine-ciclo
12: fine-ciclo
```

fosse presente almeno un ciclo di costo negativo: in questo caso l'algoritmo può essere eseguito solo su grafi orientati e privi di cicli, per cui tale verifica è del tutto inutile.

Algoritmo per l'ordinamento topologico di un grafo orientato aciclico

Come abbiamo detto per calcolare l'ordinamento topologico dei vertici di un grafo orientato aciclico è possibile modificare opportunamente l'algoritmo di visita in profondità del grafo; tale procedimento ha una complessità di $O(n+m)$. In alternativa possiamo utilizzare l'Algoritmo 6, più semplice di una visita in profondità del grafo. L'algoritmo costruisce l'ordinamento topologico partendo dai vertici sorgente (per definizione sono gli elementi minimi dell'ordinamento); rimuovendo virtualmente tali vertici e gli spigoli uscenti dalle sorgenti si otterranno altri vertici privi di spigoli entranti, visto che il grafo è aciclico. Le nuove sorgenti così ottenute vengono accodate nell'ordinamento topologico e quindi rimosse dal grafo. Il procedimento prosegue individuando nuovi vertici privi di spigoli entranti (per questo motivo viene mantenuta aggiornata l'informazione $d(v)$ con cui, per ogni vertice v del grafo, viene indicato il grado entrante di tale vertice) e rimuovendoli, insieme agli spigoli uscenti da essi; l'algoritmo termina quando tutti i vertici sono stati virtualmente rimossi dal grafo e accodati in S, la coda con cui viene tenuta traccia dell'ordinamento topologico dei vertici.

L'algoritmo inoltre utilizza una coda Q per tenere traccia, nell'ordine con cui questi vengono rilevati, dei vertici con grado entrante nullo. Viene utilizzata anche la coda S per memorizzare i vertici in ordine topologico crescente. Come abbiamo già detto, per ogni vertice v del grafo, l'algoritmo utilizza la notazione $d(v)$ per indicare il numero di spigoli entranti nel vertice v (il «grado entrante» di v).

In effetti l'algoritmo non ha bisogno di rimuovere i vertici sorgente e gli spi-

Algoritmo 6 ORDINAMENTOTOPOLOGICO(G)

Input: Un grafo orientato e aciclico $G = (V, E)$
Output: I vertici di V(G) in ordine topologico crescente

```
 1: siano Q := ∅ e S := ∅ due code
 2: per ogni v ∈ V(G) ripeti
 3:     sia d(v) il grado entrante di v
 4:     se d(v) = 0 allora
 5:         accoda v alla coda Q
 6:     fine-condizione
 7: fine-ciclo
 8: fintanto che Q ≠ ∅ ripeti
 9:     estrai il vertice u dalla coda Q
10:     accoda u in S
11:     per ogni v ∈ N(u) ripeti
12:         d(v) := d(v) − 1
13:         se d(v) = 0 allora
14:             accoda v alla coda Q
15:         fine-condizione
16:     fine-ciclo
17: fine-ciclo
18: restituisci S
```

goli da questi uscenti, è sufficiente tenere conto, aggiornando il valore $d(v)$, del numero di spigoli "ancora entranti" in ciascun vertice, diminuendo tale valore ogni volta che viene rimosso virtualmente un vertice sorgente u adiacente a v e lo spigolo (u, v) uscente da u ed entrante in v. Quando (passo 13), dopo aver decrementato il numero di spigoli entranti in un vertice v, dovesse risultare $d(v) = 0$, il vertice v viene inserito nella coda Q, per esserne poi estratto successivamente. La coda Q definisce, di fatto, l'ordinamento topologico dei vertici: il primo vertice estratto da Q è l'elemento minimo, mentre l'ultimo ad essere estratto, sarà l'elemento massimo.

Complessità computazionale dell'algoritmo per l'ordinamento topologico

È facile calcolare la complessità computazionale di questo algoritmo: il ciclo da riga 2 a riga 7 serve a calcolare il grado entrante iniziale per ciascun vertice; questa operazione si esegue in tempo $O(n+m)$, inizializzando $d(v) = 0$ per ogni $v \in V(G)$ e poi scorrendo tutte le liste di adiacenza del grafo ed incrementando il grado entrante $d(v)$ di ogni vertice presente nelle diverse liste di adiacenza. L'operazione di accodamento a riga 5 ha complessità costante $O(1)$.

I due cicli nidificati (8–17 e 11–16) eseguono complessivamente un numero di operazioni pari al numero di spigoli del grafo. Infatti, visto che il grafo è connesso, ogni vertice entra nella coda Q una ed una sola volta; quindi il ciclo 8–17 esegue una iterazione per ogni vertice $v \in V(G)$; per ciascun vertice u, con il ciclo 11–16, si esegue un'iterazione per ciascuno spigolo uscente da u, quindi complessivamente i due cicli eseguono un numero di iterazioni pari al numero

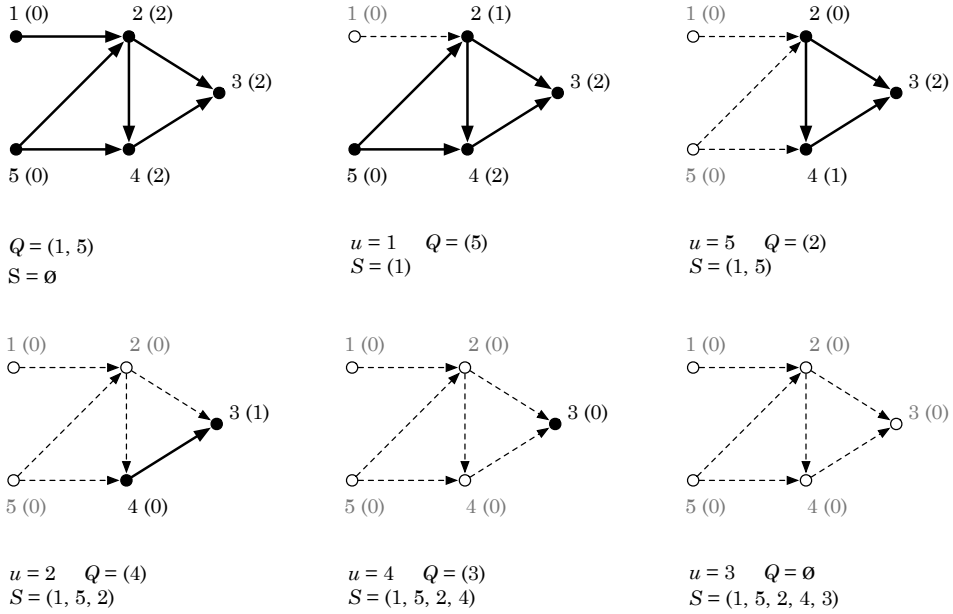


Figura 13: Un esempio di ordinamento topologico dei vertici di un grafo orientato aciclico ottenuto applicando l'Algoritmo 6; accanto all'etichetta di ciascun vertice, tra parentesi, è rappresentato il numero di spigoli entranti "residui"

degli spigoli del grafo, $m = E(G)$. L'operazione interna ai due cicli (le istruzioni 13, 14 e 15) ha complessità costante $O(1)$, pertanto la complessità dell'Algoritmo 6 è $O(n+m)$.

4 Cammini di costo minimo tra tutte le coppie di vertici

Il problema dei cammini di costo minimo per ogni coppia di vertici del grafo

Gli algoritmi visti nelle pagine precedenti permettono di calcolare i cammini di costo minimo per raggiungere ogni vertice del grafo partendo da un singolo vertice prefissato, il punto di origine, la *sorgente* dei cammini. Nella pratica è spesso utile calcolare un cammino di costo minimo per ogni coppia di vertici del grafo: per ogni coppia di vertici $u, v \in V(G)$, con $u \neq v$, si vuole calcolare il cammino di costo minimo $p_{u,v} : u \rightsquigarrow v$.

Per affrontare questo problema è possibile utilizzare uno degli algoritmi visti nelle pagine precedenti, per il calcolo del cammino di costo minimo da una sorgente singola, applicando n volte l'algoritmo utilizzando di volta in volta come sorgente un vertice diverso del grafo G. In alternativa è possibile impiegare un algoritmo specializzato per la soluzione di questo tipo di problema. In particolare, nelle pagine seguenti, ne vedremo alcuni che sfruttano la tecnica generale della **programmazione dinamica**, introdotta negli anni '40 dal matematico

statunitense Richard Ernest Bellman, che abbiamo già incontrato come autore dell'algoritmo di Bellman-Ford (pag. 17).

4.1 Programmazione dinamica

Programmazione dinamica e sottostruttura ottima del problema

La *programmazione dinamica* è una tecnica algoritmica generale, che sfrutta un approccio di carattere *bottom-up* per risolvere un'ampia classe di problemi. Non può essere applicata per la ricerca della soluzione di un problema qualsiasi, ma solo quando il problema gode della proprietà di **sottostruttura ottima**. Ossia quando la soluzione ottima di un sotto-problema è parte della soluzione ottima del problema che "contiene" tale sotto-problema.

Il problema del cammino di costo minimo gode della proprietà di sottostruttura ottima. Infatti, se $p : \alpha \rightsquigarrow \omega$ è un cammino di costo minimo dal vertice α al vertice ω e se β e γ sono due vertici che fanno parte del cammino p, allora il sotto-cammino $p' : \beta \rightsquigarrow \gamma$ è un cammino di costo minimo da β a γ; in altre parole la soluzione del sotto-problema «cammino di costo minimo da β a γ» è contenuta nella soluzione del sovra-problema «cammino di costo minimo da α a ω». Un altro cammino $p'' : \beta \rightsquigarrow \gamma$ di costo inferiore a p' non può esistere, perché se esistesse allora p non sarebbe un cammino di costo minimo da α a ω: infatti il cammino p^* ottenuto come

$$p^* : \alpha \rightsquigarrow \beta \overset{p''}{\rightsquigarrow} \gamma \rightsquigarrow \omega$$

avrebbe un costo più basso e p non sarebbe un cammino di costo minimo, come invece abbiamo supposto per ipotesi.

Procedimento *bottom-up* della programmazione dinamica

La programmazione dinamica si basa su una definizione ricorsiva della struttura ottima delle soluzioni e su un procedimento di calcolo iterativo della soluzione ottima, secondo uno schema "bottom-up". In questo modo di procedere, possiamo dire che la programmazione dinamica adotta la modalità opposta al metodo *divide et impera*, che al contrario utilizza un approccio di tipo "top-down".

Il principio generale su cui si basa la tecnica della programmazione dinamica, che vedremo applicato nelle pagine seguenti al problema del cammino di costo minimo, è quello di calcolare una tabella con la soluzione dei sottoproblemi (nel caso del problema del cammino di costo minimo i sottoproblemi più "piccoli" sono quelli relativi ai cammini di costo minimo tra vertici adiacenti) per poi comporre le soluzioni di problemi di dimensione più grande, riusando, ai passi successivi, le soluzioni trovate per i sotto-problemi, utilizzando un approccio costruttivo di tipo "bottom-up". Di fatto viene calcolata una sequenza di matrici quadrate $n \times n$, $A^{(1)}, A^{(2)}, \ldots, A^{(n)}$, definendo un'espressione ricorsiva del tipo $A^{(k)} = f(A^{(k-1)})$, per $k > 1$, e costruendo a priori $A^{(1)}$ con dei valori costanti. Il significato delle matrici $A^{(k)}$ usate nel procedimento di calcolo, varia da un algoritmo all'altro, così come cambia anche da un algoritmo all'altro la funzione $f(A^{(k-1)})$ utilizzata nel procedimento bottom-up per il calcolo delle matrici.

La forma tabellare si presta molto bene a rappresentare le soluzioni (parziali o finali) del problema del cammino di costo minimo tra tutte le coppie di vertici del grafo. Infatti con una matrice quadrata di ordine n, è possibile rappresentare nell'elemento $a_{i,j}$ della matrice il costo di un cammino tra il vertice v_i e il vertice v_j, oppure il padre del vertice v_j nel cammino di costo minimo dal vertice v_i.

Matrici dei costi e dei predecessori

Come vedremo nelle prossime pagine, i diversi algoritmi che utilizzano questa tecnica per risolvere il problema dei cammini di costo minimo, differiscono tra loro sulla tecnica utilizzata per costruire la soluzione ottima globale, partendo dalle soluzioni ottime dei sotto-problemi.

4.2 Cammini di costo minimo tra tutte le coppie con la programmazione dinamica

Per costruire un procedimento risolutivo basato sulla tecnica della programmazione dinamica dobbiamo fare qualche considerazione preliminare, che ci consentirà di costruire il procedimento "bottom-up" per calcolare la soluzione ottima globale per il problema.

Sia $p : u \rightsquigarrow v$ un cammino di costo minimo da u a v in G. Se $u = v$ allora il costo del cammino p è $w(p) = 0$ e p non ha nessuno spigolo; se invece il costo del cammino $w(p) > 0$ allora possiamo scomporre p in due sotto-cammini, passando per il vertice intermedio z adicante al vertice v:

$$p : u \overset{p'}{\rightsquigarrow} z \rightarrow v$$

Se p è un cammino minimo da u a v, allora, per la proprietà di sottostruttura ottima, il cammino p' è un cammino minimo da u a z. Risulta quindi $W(p) = W(p') + w(z, v)$, ossia, indicando con $\delta(u, v)$ il costo del cammino p, si ha che $\delta(u, v) = \delta(u, z) + w(z, v)$.

Procedimento bottom-up basato sul numero crescente di spigoli accettati per la costruzione dei cammini di costo minimo

Il procedimento alla base dell'algoritmo di programmazione dinamicache intendiamo costruire, si basa sull'idea di rilassare sempre più un vincolo posto sulla lunghezza (numero di spigoli) dei cammini di costo minimo per collegare tra loro le diverse coppie di vertici del grafo. Naturalmente imponendo un limite sul numero di spigoli che possono comporre un cammino non è possibile collegare tra loro tutte le coppie di vertici del grafo: è possibile che due vertici siano tra loro a distanza maggiore del limite fissato e che quindi non possano essere collegati da un cammino di lunghezza massima prefissata. In tal caso la stima del costo del cammino di costo minimo $\delta(u, v)$ sarà posta uguale a infinito: $\delta(u, v) = \infty$.

È bene osservare che porre un limite superiore al numero di spigoli con cui possono essere formati i cammini di costo minimo, non significa che tali cammini dovranno avere necessariamente tale lunghezza massima: il vincolo sulla lunghezza del cammino è un limite superiore, che lascia la possibilità di individuare cammini di costo minimo di lunghezza inferiore.

Indicando con $w(v_i, v_j)$ il costo dello spigolo (v_i, v_j), il dato di input per un algoritmo di questo genere è costituito da una matrice W quadrata di ordine n, definita come segue:

$$W_{i,j} = \begin{cases} 0 & \text{se } v_i = v_j \\ w(v_i, v_j) & \text{se } (v_i, v_j) \in E(G) \\ \infty & \text{se } i \neq j \text{ e } (v_i, v_j) \notin E(G) \end{cases}$$

Sequenza di matrici costruite nel procedimento bottom-up di programmazione dinamica

Il procedimento bottom-up utilizza una sequenza di matrici quadrate di ordine n, $L^{(0)}, L^{(1)}, \ldots, L^{(k)}, \ldots, L^{(n-1)}$, definite assegnando al generico elemento $l_{i,j}^{(k)}$ della matrice $L^{(k)}$ il costo minimo di un cammino da v_i a v_j in G, costituito al massimo da k spigoli. Naturalmente, se $k = 0$ esiste un cammino minimo da u a v se e solo se $u = v$. Quindi la matrice $L^{(0)}$ è definita banalmente come segue:

$$l_{i,j}^{(0)} = \begin{cases} 0 & \text{se } i = j \\ \infty & \text{altrimenti} \end{cases}$$

Anche la matrice $L^{(1)}$ è definita in modo banale: infatti esiste un cammino di lunghezza 1 da v_i a v_j se e solo se $(v_i, v_j) \in E(G)$. Pertanto possiamo definire come segue la matrice $L^{(1)}$ del costo minimo dei cammini composti al massimo da uno spigolo:

$$l_{i,j}^{(1)} = \begin{cases} 0 & \text{se } i = j \\ w(v_i, v_j) & \text{se } (v_i, v_j) \in E(G) \\ \infty & \text{se} i \neq j \text{ e } (v_i, v_j) \notin E(G) \end{cases}$$

In altri termini $L^{(1)} = W$.

Output prodotto dall'algoritmo di programmazione dinamica

Come output l'algoritmo produce due matrici quadrate di ordine n: la matrice $L^{(n-1)}$ e la matrice $\Pi^{(n-1)}$. La prima viene definita assegnando ad ogni elemento $l_{i,j}^{(n-1)}$ il costo del cammino di costo minimo dal vertice v_i al vertice v_j. Indicando con $\pi(v_j)$ il padre di v_j lungo un cammino, si definisce la matrice $\Pi^{(n-1)}$ come segue:

$$\pi_{i,j}^{(n)} = \begin{cases} \text{null} & \text{se non esiste un cammino da } v_i \text{ a } v_j \\ \pi(v_j) & \text{se il cammino esiste} \end{cases}$$

La definizione di $L^{(0)}$ e di $L^{(1)}$ ci permettono di innescare il procedimento bottom-up per il calcolo del costo dei cammini minimi fra tutte le coppie di vertici del grafo. È bene ricordare che un cammino di costo minimo non può avere una lunghezza maggiore di $n-1$: infatti, se un cammino è composto da un numero di spigoli maggiore o uguale a n, allora il cammino non è semplice e contiene dei cicli; pertanto non può essere un cammino di costo minimo. Dunque, se esiste un cammino di costo minimo tra u e v questo dovrà avere una lunghezza massima di $n-1$ spigoli.

Per calcolare il peso del cammino minimo da u a v composto al massimo da k spigoli, calcoliamo il peso minimo di un cammino minimo composto al massimo da $k-1$ spigoli tra u ed un predecessore di v:

Espressione ricorsiva per il calcolo delle matrici $L^{(1)},\ldots,L^{(n-1)}$ utilizzata dall'algoritmo di programmazione dinamica

$$l_{u,v}^{(k)} = \min\left\{ l_{u,v}^{(k-1)}, \min_{1\le z\le n}\left\{ l_{u,z}^{(k-1)} + w(z,v)\right\}\right\}$$

Siccome $w(v,v) = 0$ per ogni vertice $v \in V(G)$, possiamo semplificare come segue l'espressione precedente:

$$l_{u,v}^{(k)} = \min_{1\le z\le n}\left\{ l_{u,z}^{(k-1)} + w(z,v)\right\} \tag{1}$$

L'espressione (1) descrive il procedimento iterativo di tipo bottom-up con cui, utilizzando i risultati parziali calcolati al passo $k-1$, si produce la soluzione intermedia al passo k. Riprendendo la considerazione fatta poc'anzi, in merito al fatto che nessun cammino di costo minimo può avere più di $n-1$ spigoli, possiamo concludere che la corretta stima del costo di un cammino minimo per ogni coppia di vertici u e v, viene individuata dopo $n-1$ iterazioni dell'espressione (1):

$$\delta(u,v) = l_{u,v}^{(n-1)} \tag{2}$$

Osserviamo inoltre che deve risultare $l_{u,v}^{(h)} = l_{u,v}^{(n-1)}$ per ogni $h \ge n$, perché il cammino minimo è semplice (in assenza di cicli di costo negativo, come nel nostro caso). Per le stesse ragioni, se il cammino di costo minimo da u a v ha $h < n-1$ spigoli, allora $l_{u,v}^{(h)}$ è il costo del cammino minimo e risulta $l_{u,v}^{(h)} = l_{u,v}^{(q)}$ per ogni $q > h$.

Per calcolare il costo del cammino minimo tra tutte le coppie di vertici di G si calcolano le matrici $L^{(1)},\ldots,L^{(n-1)}$, con $L^{(k)} = \left(l_{u,v}^{(k)}\right)$, per $k = 1,\ldots,n-1$. Il punto di partenza del procedimento è dato dalla matrice $L^{(1)} = W$, dove la matrice W è calcolata ponendo $W_{u,v} = w(u,v)$.

L'Algoritmo 7 riporta una pseudo-codifica completa di questo procedimento basato sulla tecnica della programmazione dinamica. L'algoritmo restituisce la matrice $L^{(n-1)}$ che rappresenta il costo del cammino di costo minimo per ogni coppia di vertici u, v del grafo G.

Ricostruzione dei cammini di costo minimo

Se volessimo ricostruire i cammini di costo minimo, potremmo modificare l'algoritmo sostituendo alla riga 8 le seguenti istruzioni (o aggiungendole dopo la riga 8) per il calcolo delle matrici $\Pi^{(1)},\ldots,\Pi^{(n-1)}$; la matrice $\Pi^{(n-1)}$ è definita in modo tale che l'elemento generico $\pi_{i,j}^{(n-1)}$ è il padre (il predecessore) di v_j nel cammino da v_i a v_j:

se $l_{ij}^{(k)} > l_{ih}^{(k-1)} + w_{hj}$ **allora**
  $\pi_{ij}^{(k)} := h$
altrimenti
  $\pi_{ij}^{(k)} := \pi_{ij}^{(k-1)}$
fine-condizione

Algoritmo 7 ALLPAIRSSHORTESTPATH(G, w)

Input: Un grafo G orientato con funzione peso $w : E(G) \to \mathbb{R}$
Output: La matrice dei costi dei cammini minimi per ciascuna coppia di vertici in $V(G)$

1: $W = (w_{i,j})$ con $w_{i,j} := w(i, j)$ per ogni $i, j = 1, 2, \ldots, n$
2: $L^{(1)} := W$
3: **per** $k := 2, \ldots, n-1$ **ripeti**
4: **per** $i := 1, 2, \ldots, n$ **ripeti**
5: **per** $j := 1, 2, \ldots, n$ **ripeti**
6: $l^{(k)}_{i,j} := \infty$
7: **per** $h := 1, 2, \ldots, n$ **ripeti**
8: $l^{(k)}_{i,j} := \min\{l^{(k)}_{i,j}, l^{(k-1)}_{i,h} + w_{h,j}\}$
9: **fine-ciclo**
10: **fine-ciclo**
11: **fine-ciclo**
12: **fine-ciclo**
13: **restituisci** $L^{(n-1)}$

Al posto del passo 2 (o in aggiunta al passo 2) è necessario inoltre aggiungere le seguenti istruzioni di inizializzazione degli elementi della matrice $\Pi^{(1)}$:

per ogni $i := 1, 2, \ldots, n$ **ripeti**
per ogni $j := 1, 2, \ldots, n$ **ripeti**
se $(v_i, v_j) \in E(G)$ **allora**
$\pi^{(1)}_{j,i} := i$
altrimenti
$\pi^{(1)}_{i,j} :=$ null
fine-condizione
fine-ciclo
fine-ciclo

In Figura 14 è riportato un esempio di calcolo dei cammini di costo minimo tra tutte le coppie di vertici di un grafo G con $n = 6$ vertici. La matrice $L^{(5)}$ presenta i costi dei cammini minimi calcolati dall'algoritmo.

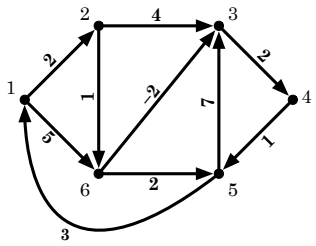
Complessità computazionale di AllPairsShortestPath

Il calcolo della **complessità computazionale** dell'Algoritmo 7 è molto semplice, dal momento che l'algoritmo stesso è composto sostanzialmente da quattro cicli nidificati e l'operazione interna ai quattro cicli (riga 8) è elementare ed ha complessità costante $O(1)$. Pertanto la complessità dell'Algoritmo ALLPAIRSSHORTESTPATH è $O(n^4)$.

Miglioramento dell'efficienza di AllPairsShortestPath

Possiamo rendere più efficiente il procedimento di calcolo dell'Algoritmo 7 procedendo nella sequenza di calcolo *bottom-up* in modo più "spedito", aggregando le soluzioni di due problemi di dimensione pari alla metà della dimensione del problema che stiamo considerando: anziché aumentare solo di 1 la dimensione del problema considerato, passando da cammini con al più $k-1$ spi-

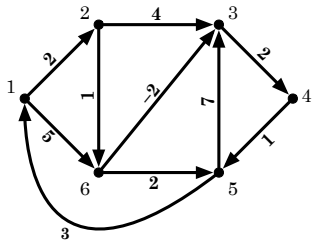
(a)



	1	2	3	4	5	6
1	**0**	**2**	∞	∞	∞	**5**
2	∞	**0**	**4**	∞	∞	**1**
3	∞	∞	**0**	**2**	∞	∞
4	∞	∞	∞	**0**	**1**	∞
5	**3**	∞	**7**	∞	**0**	∞
6	∞	∞	**−2**	∞	**2**	**0**

$L^{(1)}$

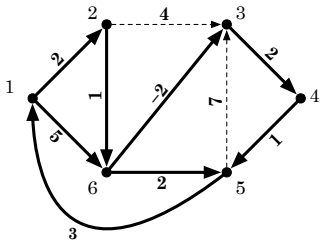
(b)



	1	2	3	4	5	6
1	0	2	**3**	∞	**7**	**3**
2	∞	0	**−1**	**6**	**3**	1
3	∞	∞	0	2	**3**	∞
4	**4**	∞	**8**	0	1	∞
5	3	**5**	7	**9**	0	**8**
6	**5**	∞	−2	**0**	2	0

$L^{(2)}$

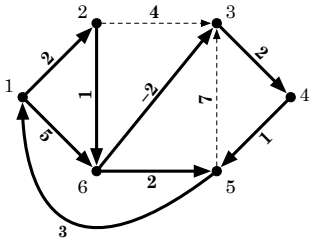
(c)



	1	2	3	4	5	6
1	0	2	**1**	**5**	**5**	3
2	**6**	0	−1	**1**	3	1
3	**6**	∞	0	2	3	∞
4	4	**6**	8	0	1	**9**
5	3	5	**6**	9	0	**6**
6	5	**7**	−2	0	**1**	0

$L^{(3)}$

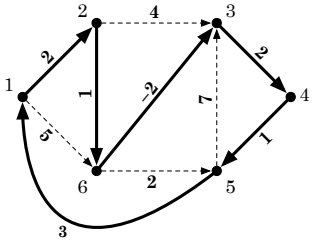
(d)



	1	2	3	4	5	6
1	0	2	1	**3**	5	3
2	6	0	−1	1	**2**	1
3	6	**8**	0	2	3	**11**
4	4	6	**7**	0	1	**7**
5	3	5	6	**8**	0	6
6	**4**	7	−2	0	1	0

$L^{(4)}$

(e)



	1	2	3	4	5	6
1	0	2	1	3	**4**	3
2	**5**	0	−1	1	2	1
3	6	8	0	2	3	**9**
4	4	6	**5**	0	1	7
5	3	5	**4**	**6**	0	6
6	4	**6**	−2	0	1	0

$L^{(5)}$

Figura 14: Un esempio del calcolo dei cammini minimi tra tutte le coppie di vertici del grafo orientato G con l'Algoritmo 7; in grassetto gli spigoli utilizzati nei diversi cammini di costo minimo e, nella matrice, in grassetto le stime di costo minimo modificate nel passo corrispondente dell'algoritmo

goli a cammini con al più k spigoli, possiamo raddoppiare la lunghezza dei cammini ammessi ad ogni passo, passando da cammini di lunghezza $k/2$ a cammini di lunghezza k.

L'espressione (1) di pagina 26 per il calcolo degli elementi della matrice $L^{(k)}$ alla $k-$esima iterazione, viene modificata come segue:

$$l_{u,v}^{(k)} = \min_{1 \le z \le n} \left\{ l_{u,z}^{(k/2)} + l_{z,v}^{(k/2)} \right\} \tag{3}$$

Il procedimento così modificato è riportato nell'Algoritmo 8.

Algoritmo 8 FASTALLPAIRSSHORTESTPATH(G, w)

Input: Un grafo G orientato con funzione peso $w : E(G) \to \mathbb{R}$
Output: La matrice dei costi dei cammini di costo minimo per ciascuna coppia di vertici in $V(G)$

1: $W = (w_{i,j})$ con $w_{i,j} := w(i,j)$ per ogni $i, j = 1, 2, \ldots, n$
2: $L^{(1)} := W$
3: $k := 1$
4: **fintanto che** $k \le n-1$ **ripeti**
5: **per** $i := 1, 2, \ldots, n$ **ripeti**
6: **per** $j := 1, 2, \ldots, n$ **ripeti**
7: $l_{i,j}^{(2k)} := \infty$
8: **per** $h := 1, 2, \ldots, n$ **ripeti**
9: $l_{i,j}^{(2k)} := \min \left\{ l_{i,j}^{(2k)}, l_{i,h}^{(k)} + l_{h,j}^{(k)} \right\}$
10: **fine-ciclo**
11: **fine-ciclo**
12: **fine-ciclo**
13: $k := 2k$
14: **fine-ciclo**
15: **restituisci** $L^{(k)}$

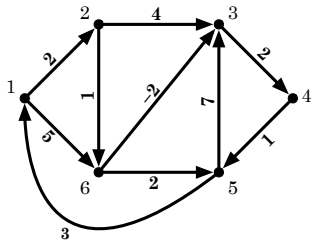
In Figura 15 è riportato un esempio di esecuzione dell'Algoritmo 8 che, effettivamente, esegue un'iterazione in meno rispetto alla stessa istanza del problema risolta con l'Algoritmo 7. Tuttavia il miglioramento della complessità computazionale è ben più rilevante. La complessità computazionale dell'Algoritmo 8 è infatti $O(n^3 \log_2 n)$, quindi quasi un ordine di grandezza inferiore a quella dell'algoritmo precedente. Infatti il ciclo esterno alle righe 4–14 esegue un numero di iterazioni pari a $\log_2 n$, migliorando nettamente la complessità computazionale rispetto a quella del corrispondente ciclo esterno del precedente algoritmo che introduce un fattore lineare alla complessità dell'algoritmo.

Complessità di FastAllPairsShortestPath

4.3 Algoritmo di Floyd-Warshall

Anche l'algoritmo di Floyd e Warshall adotta la tecnica della programmazione dinamica, già vista nelle pagine precedenti, per risolvere il problema del cammino di costo minimo su un grafo orientato con pesi positivi o negativi assegnati

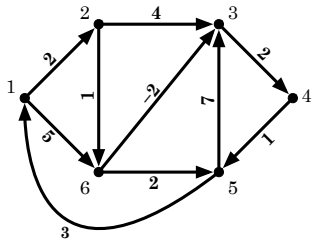
(a)



	1	2	3	4	5	6
1	**0**	**2**	∞	∞	∞	**5**
2	∞	**0**	**4**	∞	∞	**1**
3	∞	∞	**0**	**2**	∞	∞
4	∞	∞	∞	**0**	**1**	∞
5	**3**	∞	**7**	∞	**0**	∞
6	∞	∞	**−2**	∞	**2**	**0**

$L^{(1)}$

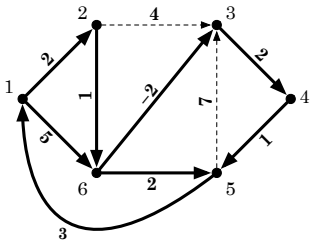
(b)



	1	2	3	4	5	6
1	0	2	**3**	∞	**7**	**3**
2	∞	0	**−1**	**6**	**3**	1
3	∞	∞	0	2	**3**	∞
4	**4**	∞	**8**	0	1	∞
5	3	**5**	7	**9**	0	**8**
6	**5**	∞	−2	**0**	2	0

$L^{(2)}$

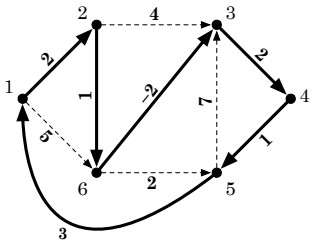
(c)



	1	2	3	4	5	6
1	0	2	**1**	**3**	**5**	3
2	**6**	0	−1	**1**	**2**	1
3	**6**	**8**	0	2	3	**11**
4	4	**6**	**7**	0	1	**7**
5	3	5	**6**	**8**	0	**6**
6	**4**	**7**	−2	0	**1**	0

$L^{(4)}$

(d)



	1	2	3	4	5	6
1	0	2	1	3	**4**	3
2	**5**	0	−1	1	2	1
3	6	8	0	2	3	**9**
4	4	6	**5**	0	1	7
5	3	5	4	**6**	0	6
6	4	**6**	−2	0	1	0

$L^{(8)}$

Figura 15: Un esempio del calcolo dei cammini minimi tra tutte le coppie di vertici del grafo orientato G con l'Algoritmo 8; in grassetto gli spigoli utilizzati nei diversi cammini di costo minimo e, nella matrice, in grassetto le stime di costo minimo modificate nel passo corrispondente dell'algoritmo


Figura 16: Robert Floyd (1936–2001) a sinistra e Stephen Warshall (1935–2006) a destra

agli spigoli. L'algoritmo individua in modo piuttosto efficiente una soluzione ottima, anche nel caso in cui siano presenti spigoli con costi negativi, purché non siano presenti nel grafo cicli di costo negativo. L'algoritmo è stato pubblicato nel 1962 da Robert Floyd sulla rivista scientifica *Communications of the ACM*; tuttavia sostanzialmente lo stesso algoritmo, ma utilizzato, come vedremo in seguito, per risolvere un problema differente del calcolo della *chiusura transitiva* di un grafo, era stato pubblicato nello stesso anno da Stephen Warshall sul *Journal of the ACM*. Per questo motivo oggi l'algoritmo è noto con il nome di entrambi i suoi autori, che però lavorarono separatamente ed autonomamente per poi conseguire il medesimo risultato.

Vertici intermedi in un cammino

Sia $p : u \rightsquigarrow v$ un cammino da u a v sul grafo G, dato da $p = \langle v_1, v_2, \ldots, v_k \rangle$, con $u = v_1$ e $v = v_k$. In questo contesto diremo che i vertici $v_2, \ldots, v_{k-1}$ sono **vertici intermedi** nel cammino p. Consideriamo tutti i cammini da u a v con vertici intermedi in $\{1, 2, \ldots, k\}$; sia p un cammino di costo minimo tra questi. L'algoritmo di Floyd e Warshall si basa su due semplici considerazioni:

- Se k **non è un vertice intermedio di** p allora i vertici intermedi di p sono tutti in $\{1, 2, \ldots, k-1\}$ e quindi un cammino minimo da u a v con vertici intermedi in $\{1, 2, \ldots, k-1\}$ è anche un cammino minimo da u a v con vertici intermedi in $\{1, 2, \ldots, k\}$.

- Se k **è un vertice intermedio di** p allora possiamo spezzare p in due sottocammini: $p : u \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} v$. Per la sottostruttura ottima dei cammini minimi risulta che p_1 è un cammino di costo minimo da u a k e che p_2 è un cammino di costo minimo da k a v; entrambi hanno vertici intermedi in $\{1, 2, \ldots, k-1\}$.

Queste due semplici considerazioni permettono di costruire, con un processo *bottom-up*, il procedimento risolutivo che è alla base dell'algoritmo di Floyd-Warshall. Sia $d_{u,v}^{(k)}$ il costo del cammino da u a v con vertici intermedi in $\{1, \ldots, k\}$.

Vale la seguente definizione ricorsiva:

$$d_{u,v}^{(k)} = \begin{cases} w_{u,v} & \text{se } k = 0 \\ \min\left\{d_{u,v}^{(k-1)}, d_{u,k}^{(k-1)} + d_{k,v}^{(k-1)}\right\} & \text{se } k \geq 1 \end{cases}$$

Visto che un cammino di costo minimo da u a v è semplice (privo di cicli) e che sicuramente tutti i vertici intermedi si trovano in $\{1,\ldots,n\}$, la matrice $D^{(n)} = \left(d_{u,v}^{(n)}\right)$ fornisce la soluzione ottima del problema, ossia risulta $d_{u,v}^{(n)} = \delta(u,v)$ per ogni $u, v \in V(G)$. Formalizziamo questo procedimento nell'Algoritmo 9.

Algoritmo 9 FLOYDWARSHALL(G, w)

Input: Un grafo G orientato con funzione peso $w : E(G) \to \mathbb{R}$
Output: La matrice dei pesi dei cammini minimi per ciascuna coppia di vertici in $V(G)$

1: $W := (w_{i,j})$ con $w_{i,j} := \begin{cases} 0 & \text{se } i = j \\ w(i,j) & \text{se } (i,j) \in E(G) \\ \infty & \text{altrimenti} \end{cases}$
2: $D^{(0)} := W$
3: **per** $k := 1,\ldots,n$ **ripeti**
4: **per** $i := 1,\ldots,n$ **ripeti**
5: **per** $j := 1,\ldots,n$ **ripeti**
6: $d_{i,j}^{(k)} := \min\left\{d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right\}$
7: **fine-ciclo**
8: **fine-ciclo**
9: **fine-ciclo**
10: **restituisci** $D^{(n)}$

L'algoritmo è molto semplice ed anche la sua implementazione in un linguaggio di programmazione non presenta particolari difficoltà. In termini di efficienza migliora la complessità computazionale degli algoritmi visti nelle pagine precedenti. Di fatto è composto da tre cicli nidificati che eseguono n iterazioni ciascuno. Pertanto la complessità dell'algoritmo di Floyd-Warshall è $\Theta(n^3)$.

Complessità dell'algoritmo di Floyd e Warshall

È possibile integrare l'algoritmo FLOYD-WARSHALL per calcolare effettivamente i cammini di costo minimo e non soltanto il costo di tali cammini, come avviene nell'Algoritmo 9. Per far questo definiamo la matrice $\Pi^{(n)}$, quadrata di ordine n, in cui il generico elemento $\pi_{u,v}^{(n)}$ rappresenta il predecessore di v nel cammino minimo da u a v. Il calcolo di $\pi_{u,v}^{(n)}$ si basa su $\pi_{u,v}^{(n-1)}$ sostanzialmente con lo stesso schema di programmazione dinamica con cui sono stati calcolati i costi dei cammini minimi; la base del procedimento ricorsivo è la seguente:

Calcolo dei cammini di costo minimo

$$\pi_{u,v}^{(0)} := \begin{cases} \text{null} & \text{se } u = v \text{ o } w(u,v) = \infty \text{ (cioè se } (u,v) \notin E(G)) \\ u & \text{se } u \neq v \text{ e } w(u,v) < \infty \text{ (cioè se } (u,v) \in E(G)) \end{cases}$$

Al passo generico dell'algoritmo possiamo calcolare $\pi_{u,v}^{(k)}$ con la seguente espressione ricorsiva:

$$\pi_{u,v}^{(k)} := \begin{cases} \pi_{u,v}^{(k-1)} & \text{se } d_{u,v}^{(k-1)} \leq d_{u,k}^{(k-1)} + d_{k,v}^{(k-1)} \\ \pi_{k,v}^{(k-1)} & \text{se } d_{u,v}^{(k-1)} > d_{u,k}^{(k-1)} + d_{k,v}^{(k-1)} \end{cases}$$

L'algoritmo di Floyd e Warshall può essere riscritto con lo pseudo-codice presentato nell'Algoritmo 10.

Algoritmo 10 FLOYDWARSHALL(G, w)

Input: Un grafo G orientato con funzione peso $w : E(G) \to \mathbb{R}$
Output: La matrice dei costi dei cammini minimi per ciascuna coppia di vertici in $V(G)$ e la matrice dei predecessori $\Pi^{(n)}$

1: $W := (w_{i,j})$ con $w_{i,j} := w(v_i, v_j)$ per ogni $i, j = 1, 2, \ldots, n$
2: $D^{(0)} := W$
3: **per** $u := 1, 2, \ldots, n$ **ripeti**
4: **per** $v := 1, 2, \ldots, n$ **ripeti**
5: **se** $u = v$ o $w_{u,v} = \infty$ **allora**
6: $\pi_{u,v}^{(0)} := null$
7: **altrimenti**
8: $\pi_{u,v}^{(0)} := u$
9: **fine-condizione**
10: **fine-ciclo**
11: **fine-ciclo**
12: **per** $k := 1, 2, \ldots, n$ **ripeti**
13: **per** $u := 1, 2, \ldots, n$ **ripeti**
14: **per** $v := 1, 2, \ldots, n$ **ripeti**
15: **se** $d_{u,v}^{(k-1)} \leq d_{u,k}^{(k-1)} + d_{k,v}^{(k-1)}$ **allora**
16: $\pi_{u,v}^{(k)} := \pi_{u,v}^{(k-1)}$, $d_{u,v}^{(k)} = d_{u,v}^{(k-1)}$
17: **altrimenti**
18: $\pi_{u,v}^{(k)} := \pi_{k,v}^{(k-1)}$, $d_{u,v}^{(k)} = d_{u,k}^{(k-1)} + d_{k,v}^{(k-1)}$
19: **fine-condizione**
20: **fine-ciclo**
21: **fine-ciclo**
22: **fine-ciclo**
23: **restituisci** $D^{(n)}, \Pi^{(n)}$

La complessità dell'algoritmo naturalmente rimane invariata, dal momento che sono stati aggiunti dei cicli di inizializzazione della matrice $\Pi^{(0)}$ (righe 3–11) che non incrementano la complessità dell'algoritmo, che è $\Theta(n^3)$.

Stampa del cammino di costo minimo da u a v

La matrice $\Pi^{(n)}$ contiene tutte le informazioni necessarie per stampare il cammino di costo minimo calcolato dall'algoritmo di Floyd-Warshall per ciascuna coppia di vertici u e v di G. Per stampare il cammino di costo minimo dal vertice u al vertice v utilizzando la matrice $\Pi^{(n)}$ si può eseguire la procedura

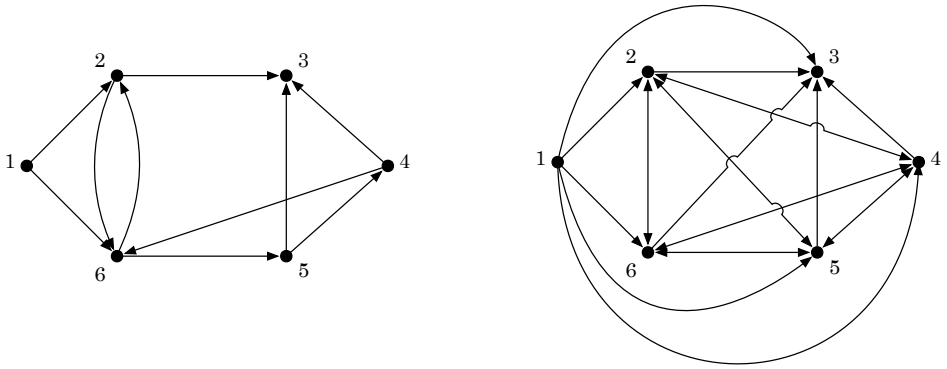


Figura 17: Un grafo G (a sinistra) e la chiusura transitiva di G (a destra)

ricorsiva riportata nell'Algoritmo 11, molto simile a quella presentata in precedenza nell'Algoritmo 2 a pagina 9. È interessante osservare che scambiando il passo 2 con il passo 3 si inverte l'ordine con cui vengono stampati i vertici (da u a v o, al contrario, da v a u).

Algoritmo 11 STAMPACAMMINO$(u, v, \Pi^{(n)})$

Input: La matrice dei predecessori $\Pi^{(n)}$, il vertice iniziale u e quello finale v

Output: La sequenza $\langle v_1, v_2, \ldots, v_k \rangle$ che costituisce il cammino da u a v calcolato dall'algoritmo di FLOYD-WARSHALL

1: **se** $u \neq v$ **allora**
2: STAMPACAMMINO$(u, \pi^{(n)}_{u,v}, \Pi^{(n)})$
3: scrivi v
4: **fine-condizione**

4.4 Calcolo della chiusura transitiva di un grafo

Chiusura transitiva di un grafo

Come abbiamo accennato nelle pagine precedenti, Stephen Warshall arrivò a formulare l'algoritmo noto anche con il suo nome, come procedimento risolutivo per il calcolo della **chiusura transitiva** di un grafo G. Dato un grafo $G = (V, E)$ la *chiusura transitiva* di G è il grafo $G^* = (V, E^*)$, con gli stessi vertici di G e l'insieme degli spigoli definito ponendo $E^* = \{(u, v) : \exists\ p : u \rightsquigarrow v\}$. Quindi in G^* due vertici sono adiacenti se in G sono collegati da un cammino.

Per calcolare il grafo chiusura transitiva G^* utilizziamo un algoritmo di programmazione dinamica molto simile a quello di Floyd-Warshall, in cui vengono sostituite le operazioni di minimo con un "or" logico ($\vee$) e le operazioni di somma con un "and" logico ($\wedge$). L'algoritmo costruisce anche in questo caso una successione di matrici quadrate di ordine n, $T^{(0)}, T^{(1)}, \ldots, T^{(n)}$, ma utilizzando le due espressioni logiche "and" e "or" invece del minimo e delle somme, interpretando i valori degli elementi delle matrici $T^{(k)}$ come valori booleani.

La dinamica dell'algoritmo si basa su un'espressione ricorsiva con cui è possibile calcolare $T^{(k)}$ a partire da $T^{(k-1)}$. La matrice $T^{(0)}$, che è alla base del procedimento bottom-up, è la matrice di adiacenza del grafo G, definita come segue:

Espressioni ricorsive per il procedimento bottom-up di programmazione dinamica per il calcolo della chiusura transitiva di un grafo

$$t_{uv}^{(0)} = \begin{cases} 0 & u \neq v \text{ e } (u,v) \notin E(G) \\ 1 & u = v \text{ o } (u,v) \in E(G) \end{cases}$$

L'espressione ricorsiva per il calcolo della matrice $T^{(k)}$ è la seguente, per $k \geq 1$:

$$t_{uv}^{(k)} = t_{uv}^{(k-1)} \vee \left(t_{uk}^{(k-1)} \wedge t_{kv}^{(k-1)} \right) \tag{4}$$

L'algoritmo termina con il calcolo di $T^{(n)}$, che rappresenta la matrice di adiacenza del grafo chiusura transitiva di G.

Algoritmo 12 CHIUSURATRANSITIVA(G)

Input: Un grafo G orientato
Output: La matrice binaria $T^{(n)}$ di adiacenza per G^*

```
1:  per u := 1,2,...,n ripeti
2:      per v := 1,2,...,n ripeti
3:          se u = v o (u,v) ∈ E(G) allora
4:              t^(0)_{u,v} := 1
5:          altrimenti
6:              t^(0)_{u,v} := 0
7:          fine-condizione
8:      fine-ciclo
9:  fine-ciclo
10: per k := 1,2,...,n ripeti
11:     per u := 1,2,...,n ripeti
12:         per v := 1,2,...,n ripeti
13:             t^(k)_{u,v} := t^(k-1)_{u,v} ∨ ( t^(k-1)_{u,k} ∧ t^(k-1)_{k,v} )
14:         fine-ciclo
15:     fine-ciclo
16: fine-ciclo
17: restituisci T^(n)
```

Interpretando i valori 1 e 0 presenti nelle matrici come i valori logici "vero" e "falso", rispettivamente, il significato dell'espressione (4) sopra riportata risulta piuttosto chiaro: v è raggiungibile da u mediante un cammino con vertici intermedi in $\{1,\ldots,k\}$ se v è raggiungibile da u con un cammino con vertici intermedi $\{1,\ldots,k-1\}$, oppure se esiste un cammino con vertici intermedi in $\{1,\ldots,k-1\}$ dal vertice u al vertice k e, al tempo stesso, esiste un cammino dal vertice k al vertice v con vertici intermedi in $\{1,\ldots,k-1\}$.

L'Algoritmo 12 presenta una pseudo-codifica del procedimento per il calcolo della chiusura transitiva di G, utilizzando la tecnica di programmazione dinamica.

Algoritmo	Sorgente singola	Tutte le coppie
DIJKSTRA	$O(m + n\log_2 n)$	$O(nm + n^2 \log_2 n)$
BELLMAN-FORD (base)	$O(nm)$	$O(n^2 m)$
BELLMAN-FORD (DAG)	$O(n+m)$	$O(n^2 + nm)$
ALLPAIRSSHORTESTPATH		$O(n^4)$
FASTALLPAIRSSHORTESTPATH		$O(n^3 \log_2 n)$
FLOYD-WARSHALL		$O(n^3)$

Tabella 1: Complessità computazionale degli algoritmi per il calcolo dei cammini di costo minimo

La struttura dell'Algoritmo 12 è identica a quella dell'algoritmo di Floyd-Warshall e quindi anche la complessità computazionale è la stessa ed è data dai tre cicli nidificati che eseguono n iterazioni ciascuno: $\Theta(n^3)$.

5 Conclusioni

Nelle pagine precedenti abbiamo visto una panoramica piuttosto sintetica sui principali algoritmi per la risoluzione del problema del cammino di costo minimo su un grafo. Questi algoritmi ci hanno permesso di introdurre dei concetti di base per la risoluzione di questo genere di problema; in particolare il «principio del rilassamento», utilizzato dagli algoritmi di Dijkstra e di Bellman-Ford e la tecnica molto generale della programmazione dinamica utilizzata dagli altri algoritmi.

Nella Tabella 1 riportiamo uno specchietto con l'indicazione della complessità computazionale per ciascun algoritmo, applicato sia alla soluzione del problema del cammino minimo da una sorgente singola che al problema del cammino di costo minimo tra ogni coppia di vertici del grafo.

Riferimenti bibliografici

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, *Data structures and algorithms*, Addison-Wesley, 1987.

[2] A.V. Aho, J.E. Hopcroft, J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduzione agli algoritmi e strutture dati*, terza edizione, McGraw-Hill, 2010.

[4] C. Demetrescu, I. Finocchi, G.F. Italiano, *Algoritmi e strutture dati*, McGraw-Hill, 2004.

[5] M. Liverani, *Programmare in C – Guida al linguaggio attraverso esercizi svolti e commentati*, seconda edizione, Società Editrice Esculapio, 2013.

[6] M. Liverani, *Qual è il problema? Metodi, strategie risolutive, algoritmi*, Mimesis, 2005.

[7] Robert Endre Tarjan, *Data Structures and Network Algorithms*, SIAM, 1983.