Ricerca Operativa

Cammini minimi: algoritmi di Bellman-Ford e di Dijkstra

L. De Giovanni

AVVERTENZA: le note presentate di seguito non hanno alcuna pretesa di completezza, né hanno lo scopo di sostituirsi alle spiegazioni del docente. Il loro scopo è quello di fissare alcuni concetti presentati in classe. Le note contengono un numero limitato di esempi ed esercizi svolti. Questi rappresentano una parte fondamentale nella comprensione della materia e sono presentati in classe.

Contents

1 L'algoritmo di Bellman-Ford

L'algoritmo label correcting generico presenta una complessità pseudo-polinomiale. Ciò è dovuto al fatto che non possono essere fatte assunzioni sull'ordine nel quale gli archi vengono controllati e le relative etichette aggiornate. In effetti, dando sistematicità ai controlli e agli aggiornamenti è possibile tenere sotto controllo il numero di iterazioni. Un possibile ordine è dato dal seguente algoritmo.

Algoritmo di *Bellman-Ford*

π_s := 0; set $p(s) = \wedge$;
for each $v \in N - s$ { set π_v:=$+\infty$, set $p(v)$:= $\wedge$ }
for h = 1 to $|N|$ {
    set π' := π; set flag_aggiornato := false;
    for all ($(i,j) \in A : \pi'_j > \pi'_i + c_{ij}$) do {
        set π_j := $\pi'_i + c_{ij}$
        set $p(j)$:= i;
        set flag_aggiornato := false;
    }
    if (not flag_aggiornato) then { STOP: π è ottima }
}
STOP: $\exists$ ciclo di costo negativo.

Si fa notare che si tratta di una particolare implementazione dell'algoritmo label correcting generico: si tratta di eseguire i controlli sugli archi e gli aggiornamenti delle etichette in un particolare ordine, e cioè il controllo h-esimo dello stesso arco può essere effettuato solo se tutti gli altri archi sono stati controllati almeno $h-1$ volte. Infatti, ad ogni iterazione con h fissato, vengono controllati *tutti* gli archi. Inoltre, il passo di inizializzazione è lo stesso, così come il criterio di arresto `not flag_aggiornato`. Si noti che, ad ogni iterazione, viene creata una copia delle etichette dell'iterazione precedente (π') e gli aggiornamenti vengono effettuati sulla base di questi valori. Tale operazione non influenza le caratteristiche della soluzione finale ottenuta: la condizione di uscita con etichette ottime rimane la stessa e, in questo caso, $\pi' = \pi$ (l'ultima iterazione lascia inalterate le etichette). Tale accorgimento sarà però utile, come vedremo, nella discussione della convergenza complessiva dell'algoritmo. In ogni caso, le etichette ottenute con l'algoritmo di Bellman-Ford (almeno se si esce con `not flag_aggiornato`) godono delle stesse proprietà viste in precedenza (ottimalità, possibilità di costruire l'albero e il grafo dei cammini minimi etc.).

Esiste tuttavia una nuova condizione di arresto (che permette all'algoritmo di individuare dei cicli negativi) e il numero di iterazioni è limitato dal ciclo `for` esterno. In effetti, tale limite, vedremo, non influenza la possibilità di ottenere le etichette ottime.

1.1 Convergenza, correttezza e complessità

Per dimostrare la convergenza, la complessità e la correttezza dell'algoritmo in generale ci si basa sulla seguente proprietà.

Lemma 1 *Alla fine dell'iterazione con $h = \bar{h}$, π_j rappresenta il costo di un* cammino minimo *da s a j con* al più $\bar{h}$ archi.

La dimostrazione formale, che qui omettiamo, si ottiene per induzione sul numero di iterazioni, ribadendo che, ad ogni iterazione, gli aggiornamenti vengono effettuati sulla base delle etichette dell'iterazione precedente. Ad esempio, quando $h = 1$, tutti gli archi vengono controllati e si aggiornano le etichette di *tutti* i successori di s nel grafo (e solo quelli). Chiaramente, le etichette così ottenute sono relative ai cammini minimi da s verso tutti gli altri nodi che usano esattamente un arco (le etichette rimangono a $+\infty$ per i nodi non direttamente collegati a s). Con $h = 2$ sono aggiornate le etichette dei nodi collegati ai successori dei nodi aggiornati nella prima iterazione (si considerano i π'). Essendo considerati *tutti* i possibili archi, tutti i possibili miglioramenti delle etichette ottenibili con cammini di al più 2 archi sono considerati, e così via.

Abbiamo visto come, in assenza di cicli di costo negativo, un cammino minimo contiene al più $|N| - 1$ archi. Pertanto, per il Lemma 1, l'ultimo aggiornamento di una etichetta avviene al più all'iterazione $h = |N| - 1$ e, con l'iterazione al più $h = |N|$ si verifica semplicemente che nessuna etichetta è cambiata (`flag_aggiornato = false`) e l'algoritmo termina con le etichette ottime. Se invece esistono cicli negativi, all'iterazione $h = N$ si continuano a migliorare le etichette (`flag_aggiornato = true`) e pertanto si uscirà normalmente dal ciclo `for` e si segnalerà la presenza di un ciclo negativo. In questo caso, un ciclo negativo si ottiene percorrendo a ritroso i puntatori a partire da un nodo la cui etichetta è cambiata nell'iterazione $h = |N|$.

L'algoritmo di Bellman-Ford, quindi, converge anche in presenza di cicli di costo negativo e il numero massimo di iterazioni è pari al numero di nodi $|N|$. Ad ogni iterazione, esattamente $|A|$ controlli con eventuali aggiornamenti di π e $p(\cdot)$ sono effettuati in tempo costante, per una complessità computazionale di $O(|N| \cdot |A|)$ che è polinomiale. Abbiamo quindi la seguente proprietà:

Proprietà 1 *Dato un grafo pesato $G = (N, A)$ e un nodo origine $s \in N$, l'algoritmo di Bellman-Ford fornisce la soluzione ottima del problema del cammino minimo dal nodo origine s verso tutti gli altri nodi o individua un ciclo di costo negativo in $O(|N| \cdot |A|)$.*

1.2 Esempi e implementazione efficiente

Esempio 1 *Si consideri il grafo in Figura 1 e si calcoli il costo dei cammini minimi da* 1 *verso tutti gli altri nodi applicando l'algoritmo di Bellman-Ford.*

Per seguire il corretto ordine di controllo degli archi e aggiornamento delle etichette (evitare, ad esempio, di aggiornare le etichette sulla base delle π invece che delle π'),

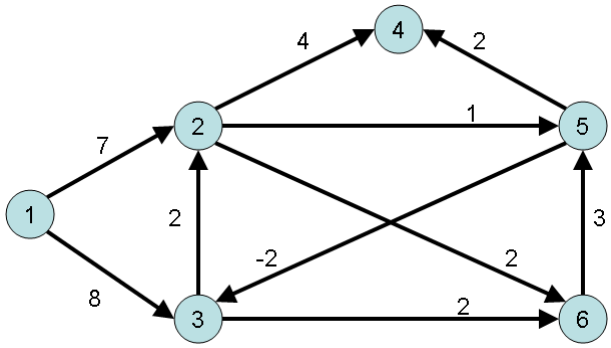

Figure 1: Rete di flusso dell'Esempio 1.

è conveniente organizzare le iterazioni in una tabella. In ogni cella si riportano il valore di π e il nodo predecessore (tra parentesi).

iterazione	nodo 1	nodo 2	nodo 3	nodo 4	nodo 5	nodo 6
$h=0$	$0_{(\wedge)}$	$+\infty_{(\wedge)}$	$+\infty_{(\wedge)\ (\wedge)}$	$+\infty_{(\wedge)}$	$+\infty_{(\wedge)}$	$+\infty_{(\wedge)}$
$h=1$	$0_{(\wedge)}$	$+7_{(1)}$	$+8_{(1)}$	$+\infty_{(\wedge)}$	$+\infty_{(\wedge)}$	$+\infty_{(\wedge)}$
$h=2$	$0_{(\wedge)}$	$+7_{(1)}$	$+8_{(1)}$	$+11_{(2)}$	$+8_{(2)}$	$+9_{(2)}$
$h=3$	$0_{(\wedge)}$	$+7_{(1)}$	$+6_{(5)}$	$+10_{(5)}$	$+8_{(2)}$	$+9_{(2)}$
$h=4$	$0_{(\wedge)}$	$+7_{(1)}$	$+6_{(5)}$	$+10_{(5)}$	$+8_{(2)}$	$+8_{(3)}$
$h=5$	$0_{(\wedge)}$	$+7_{(1)}$	$+6_{(5)}$	$+10_{(5)}$	$+8_{(2)}$	$+8_{(3)}$

Attenzione: ad ogni iterazione i controlli e gli aggiornamenti si devono basare sulle etichette dell'iterazione precedente (riga sopra) e non su quelle dell'iterazione corrente (stessa riga). Ad esempio, all'iterazione $h = 2$, le etichette dei nodi 4, 5 e 6 restano a $+\infty$, anche se gli archi, ad esempio, $(2, 4)$, $(2, 5)$ e $(3, 6)$ vengono controllati dopo l'arco $(1, 2)$, che ha permesso di aggiornare π_2 per l'iterazione corrente, mentre π'_2 è rimasto a $+\infty$. Analogamente, all'iterazione $h = 3$, π_6 resta 9.

L'algoritmo termina con la convergenza delle etichette, e pertanto le etichette nell'ultima (e penultima) riga sono ottime (costi dei cammini minimi). Attraverso i puntatori è inoltre possibile costruire l'albero (o il grafo) dei cammini minimi, mostrato in figura 2.

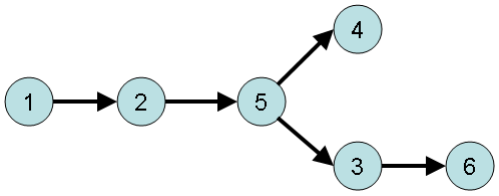

Figure 2: Un albero dei cammini minimi per l'Esempio 1.

Si fa notare come, ad ogni iterazione, sia inutile verificare le condizioni per gli archi uscenti da nodi le cui etichette non vengono aggiornate. Pertanto, è possibile migliorare l'efficienza computazionale media (*ma NON quella del caso peggiore*) andando a memorizzare, ad ogni iterazione, la lista dei nodi le cui etichette vengono aggiornate (Lista `Aggiornati`). All'iterazione successiva basterà controllare gli archi $(i, j) \in A : i \in$ `Aggiornati`.

Esempio 2 *Si risolva il problema dei cammini minimi con origine nel nodo* 1 *per il grafo in Figura 3.*

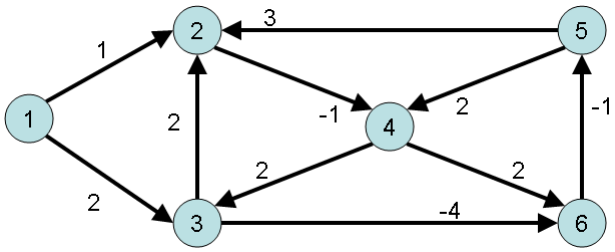

Figure 3: Grafo per l'Esempio 2.

iterazione	nodo 1	nodo 2	nodo 3	nodo 4	nodo 5	nodo 6	`Aggiornati`
$h = 0$	$0_{(\wedge)}$	$+\infty_{(\wedge)}$	$+\infty_{(\wedge)}$	$+\infty_{(\wedge)}$	$+\infty_{(\wedge)}$	$+\infty_{(\wedge)}$	1
$h = 1$	$0_{(\wedge)}$	$+1_{(1)}$	$+2_{(1)}$	$+\infty_{(\wedge)}$	$+\infty_{(\wedge)}$	$+\infty_{(\wedge)}$	2, 3
$h = 2$	$0_{(\wedge)}$	$+1_{(1)}$	$+2_{(1)}$	$0_{(2)}$	$+\infty_{(\wedge)}$	$-2_{(3)}$	4, 6
$h = 3$	$0_{(\wedge)}$	$+1_{(1)}$	$+2_{(1)}$	$0_{(2)}$	$-3_{(6)}$	$-2_{(3)}$	5
$h = 4$	$0_{(\wedge)}$	$0_{(5)}$	$+2_{(1)}$	$-1_{(5)}$	$-3_{(6)}$	$-2_{(3)}$	2, 4
$h = 5$	$0_{(\wedge)}$	$0_{(5)}$	$+1_{(4)}$	$-1_{(5)}$	$-3_{(6)}$	$-2_{(3)}$	3
$h = 6$	$0_{(\wedge)}$	$0_{(5)}$	$+1_{(4)}$	$-1_{(5)}$	$-3_{(6)}$	$-3_{(3)}$	6

L'algoritmo ha terminato con `flag_aggiornato = true` e pertanto esiste un ciclo negativo. Tale ciclo è individuato partendo dal nodo 6, (uno dei nodi) che ha subito un aggiornamento all'ultima iterazione, e procedendo a ritroso attraverso i puntatori $p(\cdot)$: $6 \leftarrow 3 \leftarrow 4 \leftarrow 5 \leftarrow 6$ (di costo $-1 + 2 + 2 - 4 = -1$).

1.3 Cammini minimi con massimo numero di archi

Esistono applicazioni di notevole importanza in cui si è interessati ai cammini minimi con un numero limitato di archi (ad esempio, nelle reti di telecomunicazione, per limitare la probabilità di errore di trasmissione). Si parla in questo caso di cammini minimi con un massimo numero di archi (*max-hop*, per le reti di telecomunicazione). Il Lemma 1 ci permette di risolvere tale problema con l'algoritmo di Bellman-Ford: basta fermarsi all'iterazione $h = \bar{h}$, dove $\bar{h}$ è il limite richiesto sul numero di archi.

Esempio 3 *Si calcolino i cammini minimi con al più 3 archi per il grafo in Figura 1.*

Applicando l'algoritmo di Bellman-Ford limitato a 3 iterazioni si ottiene:

iterazione	nodo 1	nodo 2	nodo 3	nodo 4	nodo 5	nodo 6	`Aggiornati`
$h=0$	$0_{(\wedge)}$	$+\infty_{(\wedge)}$	$+\infty_{(\wedge)\ (\wedge)}$	$+\infty_{(\wedge)}$	$+\infty_{(\wedge)}$	$+\infty_{(\wedge)}$	1
$h=1$	$0_{(\wedge)}$	$+7_{(1)}$	$+8_{(1)}$	$+\infty_{(\wedge)}$	$+\infty_{(\wedge)}$	$+\infty_{(\wedge)}$	$2,3$
$h=2$	$0_{(\wedge)}$	$+7_{(1)}$	$+8_{(1)}$	$+11_{(2)}$	$+8_{(2)}$	$+9_{(2)}$	$4,5,6$
$h=3$	$0_{(\wedge)}$	$+7_{(1)}$	$+6_{(5)}$	$+10_{(5)}$	$+8_{(2)}$	$+9_{(2)}$	$3,4$

Si noti che, anche in presenza di cicli negativi, i cammini minimi con un massimo numero di archi sono ben definiti, visto che non è possibile percorrere i cicli negativi un numero di volte illimitato.

Esempio 4 *Si calcolino i cammini minimi con al più 4 archi per il grafo in Figura 2.*

Applicando l'algoritmo di Bellman-Ford limitato a 4 iterazioni si ottiene:

iterazione	nodo 1	nodo 2	nodo 3	nodo 4	nodo 5	nodo 6	`Aggiornati`
$h=0$	$0_{(\wedge)}$	$+\infty_{(\wedge)}$	$+\infty_{(\wedge)}$	$+\infty_{(\wedge)}$	$+\infty_{(\wedge)}$	$+\infty_{(\wedge)}$	1
$h=1$	$0_{(\wedge)}$	$+1_{(1)}$	$+2_{(1)}$	$+\infty_{(\wedge)}$	$+\infty_{(\wedge)}$	$+\infty_{(\wedge)}$	$2,3$
$h=2$	$0_{(\wedge)}$	$+1_{(1)}$	$+2_{(1)}$	$0_{(2)}$	$+\infty_{(\wedge)}$	$-2_{(3)}$	$4,6$
$h=3$	$0_{(\wedge)}$	$+1_{(1)}$	$+2_{(1)}$	$0_{(2)}$	$-3_{(6)}$	$-2_{(3)}$	5
$h=4$	$0_{(\wedge)}$	$0_{(5)}$	$+2_{(1)}$	$-1_{(5)}$	$-3_{(6)}$	$-2_{(3)}$	$2,4$

Si fa infine notare che, in presenza del massimo numero di archi, è possibile costruire solo l'albero dei cammini minimi, attraverso i puntatori ai predecessori di ogni nodo. Infatti, il grafo dei cammini minimi, anche se costruito sulla base delle etichette ottenute limitando il numero di iterazioni, NON rappresenta tutti e soli i cammini minimi vincolati: alcuni cammini sul grafo potrebbero superare il limite di archi. Ad esempio, le etichette in Figura 4 sono relative a cammini minimi con al massimo due archi (sono ottenibili con due iterazioni dell'algoritmo di Bellman-Ford), mentre il corrispondente grafo dei cammini minimi contiene un cammino minimo $1 \to 2 \to 3 \to 4$ con 3 archi.

1.4 Cammini minimi e principio di sub-ottimalità (nota storica)

La correttezza degli algoritmi label correcting (tra cui l'algoritmo di Bellman-Ford) si basa sul seguente principio di ottimalità, che abbiamo dimostrato ricorrendo alla teoria della dualità in programmazione lineare:

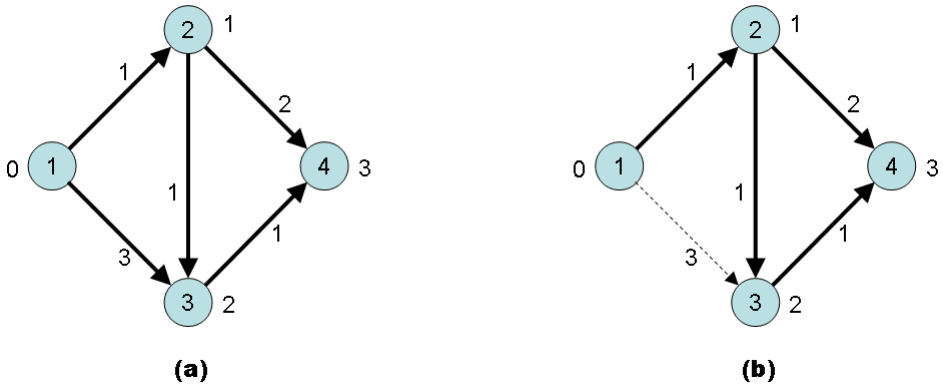

Figure 4: Grafo dei cammini minimi e massimo numero di archi.

Proprietà 2 (Condizioni di ottimalità dei cammini minimi) *Dato un grafo $G = (N, A)$ con costi c sugli archi e un nodo origine $s \in N$, e date delle etichette π_v per ogni nodo $v \in N$, queste rappresentano i costi dei cammini minimi da s verso v* se e solo se *π_v è la lunghezza di un cammino ammissibile da s a v e $\pi_j - \pi_i \leq c_{ij}$, $\forall\, (i, j) \in A$.*

Tali condizioni sono note come *Condizioni di ottimalità di Bellman* e sono le condizioni storicamente sfruttate da Bellman e Ford negli anni '60 per proporre l'algoritmo sopra esposto. Tale algoritmo nasceva come un'applicazione della programmazione dinamica. In effetti, le condizioni di ottimalità possono essere dimostrate del principio di sub-ottimalità dei cammini minimi, alla base dell'approccio di programmazione dinamica.

Proprietà 3 (Principio di sub-ottimalità dei cammini minimi) *P è un cammino minimo da s a d se e solo se, per ogni coppia di nodi i e j nel cammino, il sotto-cammino da i a j $P_{ij} \subseteq P$ è un cammino minimo.*

La proprietà è facilmente dimostrabile, considerando che, se P_{ij} non fosse minimo, allora esisterebbe un altro cammino da i a j con costo inferiore. Sostituendo tale cammino a P_{ij} nel cammino P, si otterrebbe un nuovo cammino da s a d migliore di P, contraddicendo l'ottimalità di P.

2 Algoritmo di Dijkstra per il problema del cammino minimo

L'algoritmo di Dijkstra può essere applicato per calcolare i cammini minimi di un grafo $G = (N, A)$ i cui costi soddisfino la seguente condizione:

$$c_{ij} \geq 0 \; \forall \; (i,j) \in A$$

Si tratta di uno schema che calcola delle etichette π ammissibili (e ottime) per il problema duale con una caratteristica che lo differenzia dagli algoritmi *label correcting* sopra esposti e che gli permette di raggiungere una migliore complessità computazionale. Gli algoritmi label correcting, infatti, mantengono delle etichette π che sono fissate come ottime tutte insieme e solo all'ultima iterazione. L'algoritmo di Dijkstra, invece, sotto l'ipotesi di costi non negativi, è in grado di *fissar*e, ad ogni iterazione, il valore ottimo di un'etichetta e, in questo modo, riduce la complessità computazionale per il calcolo del cammino minimo su grafi con costi tutti ≥ 0. Si tratta pertanto di un algoritmo *label setting*, che termina quando viene fissata l'ultima etichetta.

L'algoritmo di Dijkstra determina i valori ottimi delle etichette π_j, $\forall \; j \in N$ mantenendo, ad ogni iterazione, una partizione dei nodi N in due sottoinsiemi S e $\bar{S}$: i nodi in S sono i nodi con etichetta fissata (già ottima) e i nodi in $\bar{S}$ sono quelli la cui etichetta deve essere ancora fissata. Ad ogni iterazione un nodo in $\bar{S}$ viene trasferito in S (*label setting*) e l'algoritmo termina quando $\bar{S} = \emptyset$ (e $S = N$). L'algoritmo procede per aggiornamenti successivi delle π, con conseguente aggiornamento dei consueti puntatori $p(j)$, $\forall \; j \in N$. Nell'algoritmo considereremo un nodo origine s, e indicheremo con $\Gamma_i \subset N$ l'insieme dei nodi successori di un nodo $i \in N$ nel grafo G, ossia $\Gamma_i = \{j \in N : (i,j) \in A\}$.

Algoritmo di *Dijkstra*

0) π_s := 0; set $p(1)$:= $\wedge$; set S := $\{s\}$; set $\bar{S}$:= $S \setminus \{s\}$;
   for each $j \in \Gamma_s$ { set π_j := c_{sj}; set $p(j)$:= s }
   for each $v \in N \setminus \Gamma_s \setminus \{s\}$ { set π_v:=$+\infty$, set $p(v)$:= $\wedge$ }
1) set $\hat{v}$:=$\arg\min_{i \in \bar{S}} \{\pi_i\}$
   set S := $S \cup \{\hat{v}\}$; set $\bar{S}$:= $\bar{S} \setminus \{\hat{v}\}$;
   if $\bar{S} = \emptyset$ then STOP: π è ottimo.
2) for all ($j \in \Gamma_{\hat{v}} \cap \bar{S} : \pi_j > \pi_{\hat{v}} + c_{\hat{v}j}$) do {
     set π_j := $\pi_{\hat{v}} + c_{\hat{v}j}$
     set $p(j)$:= $\hat{v}$;
   }
   go to 1)

Prima di discutere le proprietà dell'algoritmo, vediamo un esempio di applicazione.

Esempio 5 *Si applichi l'algoritmo di Dijkstra per il calcolo dei cammini minimi nel grafo in Figura 5 a partire dal nodo 1.*

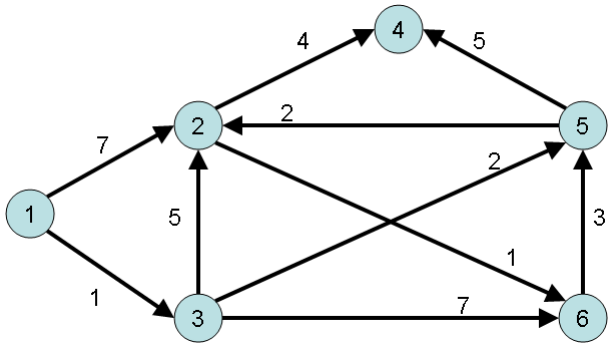

Figure 5: Grafo per l'Esempio 5.

Anche in questo caso possiamo organizzare i calcoli delle varie iterazioni in una tabella.

it.	nodo 1	nodo 2	nodo 3	nodo 4	nodo 5	nodo 6	$\bar{S}$	$\hat{v}$
0	$0^{*}_{(\wedge)}$	$7_{(1)}$	$1_{(1)}$	$+\infty_{(\wedge)}$	$+\infty_{(\wedge)}$	$+\infty_{(\wedge)}$	$2,3,4,5,6$	
1		$6_{(3)}$	*		$3_{(3)}$	$8_{(3)}$	$2,4,5,6$	3
2		$5_{(5)}$		$8_{(5)}$	*		$2,4,6$	5
3		*		$-$		$6_{(2)}$	$4,6$	2
4					$\times$	*	4	6
5				*			$\emptyset$	4

$*$: etichetta fissata.
$-$: etichetta controllata ma non aggiornata.
$\times$: etichetta non controllata perché il nodo è già fissato.

Dopo l'inizializzazione, ad ogni iterazione:

1) si considerano le etichette correnti e si sceglie quella più piccola; il corrispondente nodo $\hat{v}$ viene estratto da $\bar{S}$ e inserito in S;

2) si procede con le verifiche della condizione di Bellman (vincolo duale) sugli archi che escono da $\hat{v}$ e portano a nodi in $\bar{S}$.

Si noti che, al passo `2)`, le etichette dei nodi in S non sono più messe in discussione, e risultano pertanto fissate nel momento in cui un nodo viene inserito in S. Ad esempio, al passo 4, l'unico arco uscente da $\hat{v} = 6$ è $(6, 5)$, ma la verifica non viene effettuata, essendo $6 \in \bar{S}$. Si noti anche che, al passo 4, l'etichetta del nodo 4 viene controllata ($4 \in \Gamma_2$) ma non viene aggiornata ($\pi_4 = 8 < 5 + 4 = \pi_2 + c_{24}$)[1].
L'algoritmo termina quando non ci sono più nodi in $\bar{S}$.

Anche in questo caso, è possibile derivare l'albero (risp. il grafo) dei cammini minimi attraverso i puntatori ai predecessori (risp. la verifica della saturazione dei vincoli duali sugli archi). Nell'esempio, l'albero e il grafo dei cammini minimi coincidono e sono rappresentati in Figura 6.

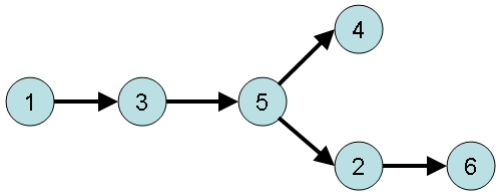

Figure 6: Albero/Grafo dei cammini minimi per l'Esempio 5.

2.1 Convergenza, correttezza e complessità

Per quanto riguarda la convergenza dell'algoritmo di Dijkstra, questa è garantita dal fatto che, ad ogni iterazione, esattamente un nodo lascia l'insieme $\bar{S}$. Saranno pertanto sufficienti $|N|-1 = O(|N|)$ iterazioni per raggiungere la condizione di terminazione $\bar{S} = \emptyset$.

Rispetto ad un algoritmo label correction, l'algoritmo di Dijkstra limita le verifiche della condizione di Bellman (vincoli duali) ai soli archi uscenti dal nodo correntemente selezionato ed entranti in un nodo correntemente nell'insieme $\bar{S}$. Per dimostrare che ciò è lecito, utilizziamo il seguente risultato.

Lemma 2 *Ad ogni iterazione, per ogni nodo $i \in N$:*

- *se $i \in S$, π_i è il costo di un cammino minimo dal nodo origine s a i, $p(i)$ è il predecessore di i su tale cammino, e tale cammino utilizza solo nodi in S;*
- *se $i \in \bar{S}$, π_i è il costo di un cammino minimo dal nodo origine s a i* che utilizza solo nodi in S *per arrivare a i, e $p(i)$ è il predecessore di i su tale cammino.*

[1]Le condizioni sono segnalate in tabella con i simboli $\times$ e $-$ rispettivamente, in modo da verificare facilmente (in fase di svolgimento di un esercizio) che tutti gli archi uscenti da $\hat{v}$ siano stati correttamente presi in considerazione per effettuare o non effettuare la verifica ed eventualmente aggiornare un'etichetta.

Dimostrazione: Utilizziamo l'induzione sul numero di iterazioni. Alla fine della prima iterazione, S è composto dal nodo s (per il quale la verifica della proprietà è immediata) e dal nodo $\hat{v}$, scelto al passo `1)`. Il cammino $s \to \hat{v}$ è un cammino minimo da s a v a costo $c_{s\hat{v}} = \pi_{\hat{v}}$ e non è possibile migliorarlo: ogni altro percorso dovrebbe passare da un terzo nodo v e sarebbe del tipo $s \to \overbrace{v \rightsquigarrow \hat{v}}^{P}$ il cui costo è $c_{sv} + c(P)$. Ma $c_{sv} \geq c_{s\hat{v}}$ (per scelta di $\hat{v}$) e $c(P) \geq 0$, essendo P composto da *archi* di lunghezza ≥ 0. Abbiamo quindi che $\pi_{\hat{v}}$ è il costo di un cammino minimo da s a $\hat{v}$ e che $s = p(\hat{v})$ è il predecessore di $\hat{v}$ su tale cammino, che utilizza solo nodi in S. Per quanto riguarda i nodi in $j \in \bar{S}$, questi sono di 4 tipi (vedi Figura 7):

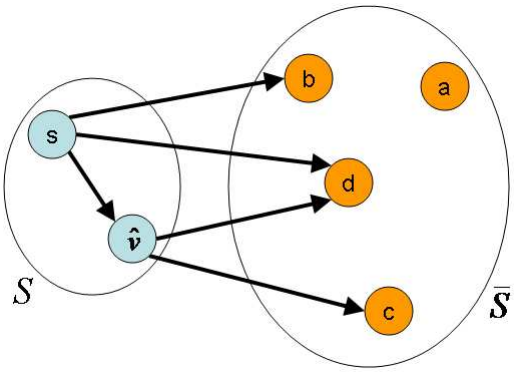

Figure 7: Primo passo induttivo per il Lemma 2.

a) j non è raggiungibile né da s, né da $\hat{v}$: non esistono cammini da s a j con soli nodi in S prima di j; l'etichetta di j non è stata finora aggiornata e rimane $\pi_j = +\infty$ e $p(j) = \wedge$;

b) j è raggiungibile solo da s: l'unico cammino da s a j con soli nodi in S prima di j è $s \to j$ di costo c_{sj}; π_j non viene modificata dall'iterazione e rimane $\pi_j = c_{sj}$ e $p(j) = s$;

c) j è raggiungibile solo da $\hat{v}$: l'unico cammino da s a j con soli nodi in S prima di j è $s \to \hat{v} \to j$ di costo $c_{s\hat{v}} + c_{\hat{v}j}$; l'etichetta π_j era $+\infty$ prima dell'iterazione e diventa $\pi_j = \pi_{\hat{v}} + c_{\hat{v}j}$ con $p(j) = \hat{v}$;

d) j è raggiungibile sia da s che da $\hat{v}$: esistono due cammini da s a j con soli nodi in S prima di j, $P_1 = s \to j$ e $P_2 = s \to \hat{v} \to j$; prima dell'iterazione, π_j e $p(j)$ erano relativi al cammino P_1; dopo l'iterazione, se il costo di P_2 è migliore del costo di P_1, π_j e $p(j)$ vengono modificati di conseguenza.

In ogni caso, l'asserto è verificato per l'iterazione 1.

Consideriamo ora l'iterazione $k+1$ sotto l'ipotesi induttiva che l'asserto sia vero all'iterazione k. Sia $\hat{w}$ il nodo selezionato al passo `1)` dell'iterazione $k+1$ per il passaggio da $\bar{S}$ a S. Osserviamo che, avendo trasferito $\hat{w}$ in S, adesso $\pi_{\hat{w}}$ è il costo di un cammino tutto appartenente ad S (per ipotesi induttiva tutti i restanti nodi del cammino erano già in S). Bisogna dimostrare che $\pi_{\hat{w}}$ è il costo di un cammino *minimo* da s a $\hat{w}$. Se così non fosse, visto che, per ipotesi induttiva, $\pi_{\hat{w}}$ è il costo di un cammino minimo da s a $\hat{w}$ con soli nodi di S prima di w, dovrebbe esistere un cammino P^* da s a $\hat{w}$ che comprende nodi di $\bar{S}$, cioè, con riferimento alla Figura 8, $P^* = P_1 \cup (p(i), i) \cup P_2 \cup P_3$ di costo $c(P_1) + c_{p(i)i} + c(P_2) + c(P_3) < \pi_{\hat{w}}$. Ora, P_1 utilizza solo archi in S prima di i e, per ipotesi induttiva, $c(P_1) + c_{p(i)i} \geq \pi_i$; per come è stato scelto $\hat{w}$, si ha anche $\pi_i \geq \pi_{\hat{w}}$. Inoltre, essendo i cammini P_2 e P_3 composti da *archi* con costo non negativo, $c(P_2) \geq 0$ e $c(P_3) \geq 0$. Quindi $c(P^*) \geq \pi_{\hat{w}}$, che contraddice l'ipotesi su P^*. Pertanto, $\pi_{\hat{w}}$ è l'etichetta ottima di $\hat{w}$.

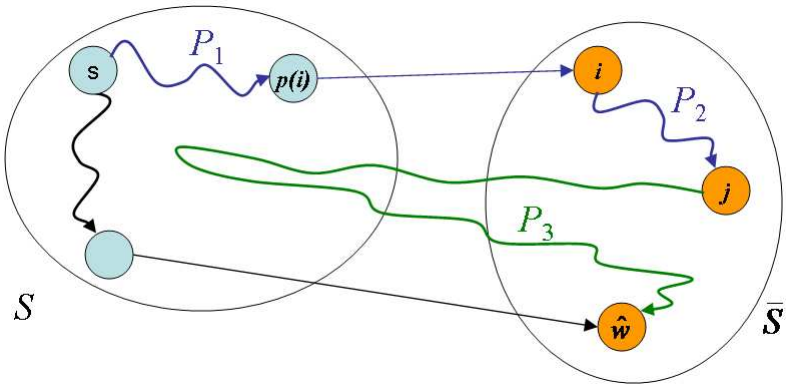

Figure 8: Induzione per il Lemma 2: nodi in S.

Per quanto riguarda i nodi $j \in \bar{S}$, l'inserimento di $\hat{w}$ in S potrebbe portare a ridurre l'etichetta π_j, visto che $\hat{w}$ potrebbe diventare un nodo interno ad un cammino da s a j. Distinguiamo due casi a seconda che j sia o meno successore di $\hat{w}$ (vedi Figura 9). Se $j \notin \Gamma_{\hat{w}}$, allora la sua etichetta potrebbe essere migliorata da un cammino del tipo $P^* = s \rightsquigarrow \hat{w} \rightsquigarrow i \rightarrow j$, con $i \neq \hat{w}$. Osserviamo innanzitutto che, per ipotesi induttiva, non può essere $\pi_j > \pi_i + c_{ij}$, altrimenti avremmo un cammino da s a j con soli nodi in S prima di j a costo inferiore di π_j. Pertanto $\pi_j <= \pi_i + c_{ij}$. Per quanto riguarda il costo di P^*, abbiamo $c(P^*) = c(s \rightsquigarrow \hat{w}) + c(\hat{w} \rightsquigarrow i) + c_{ij}$. Per ipotesi induttiva, π_i è il costo di un cammino minimo da s a i e pertanto, $c(s \rightsquigarrow \hat{w}) + c(\hat{w} \rightsquigarrow i) \geq \pi_i$, da cui $c(P^*) \geq \pi_i + c_{ij} \geq \pi_j$. Quindi, l'etichetta di un nodo $j \notin \Gamma_{\hat{w}}$ non può essere migliorata. Nel caso $j \in \Gamma_{\hat{w}}$, oltre alla possibilità esclusa dalla discussione precedente, potremmo migliorare π_j con un cammino $P^* = s \rightsquigarrow \hat{w} \rightarrow j$, il cui costo è $\pi_{\hat{w}} + c_{\hat{w}j}$. In effetti, il passo `3)` dell'iterazione $k+1$ contempla proprio questa possibilità, aggiornando opportunamente π_j e $p(j)$. ■

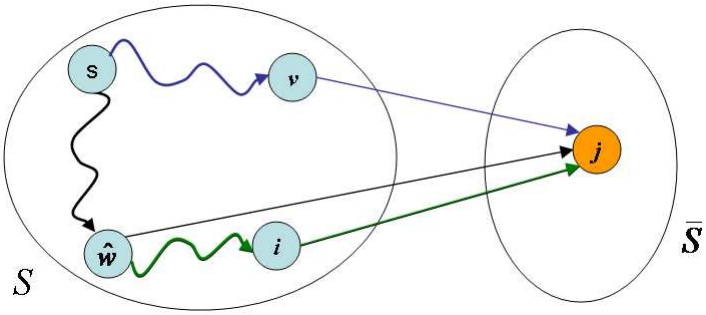

Figure 9: Induzione per il Lemma 2: nodi in $\bar{S}$.

Dal Lemma 2 è immediato ricavare la correttezza dell'algoritmo.

Proprietà 4 *Dato un grafo pesato $G = (N, A)$ e un nodo origine $s \in N$, l'algoritmo di Dijkstra risolve il problema del cammino minimo dall'origine s verso tutti gli altri nodi in tempo $O(|N|^2)$.*

Dimostrazione: L'algoritmo parte con $S = \{s\}$ e trasferisce un nodo da $\bar{S}$ in S ad ogni iterazione. Dopo $|N| - 1$ iterazioni quindi, $\bar{S} = \emptyset$ e l'algoritmo termina. All'ultima iterazione, tutti i nodi sono in S e, pertanto, π_i rappresenta il costo del cammino minimo da s verso i, per tutti i nodi $i \in N$ (e $p(i)$ il relativo predecessore). Inoltre, il passo `0)` comprende $O(N)$ operazioni di inizializzazione effettuabili in tempo costante. Il passo `2)` consiste nella ricerca del minimo in un insieme di al più $|N| - 1$ elementi. La ricerca del minimo si può effettuare scandendo i nodi in $\bar{S}$ ed effettuando $|\bar{S}|$ confronti, quindi in tempo $O(|N|)$. Per quanto riguarda il passo `3)`, consideriamo la sua complessità *ammortizzata*, ossia la complessità cumulativa di tutte le iterazioni. Ad ogni iterazione si considerano gli archi uscenti da un solo nodo. Nel corso delle iterazioni si considerano tutti i nodi (esclusa l'origine, considerata però al passo `0)`) e, di conseguenza, si verificano i vincoli duali su tutti gli archi. L'operazione di verifica ed eventuale aggiornamento di un etichetta e di un puntatore prende tempo costante e pertanto, la complessità ammortizzata del passo `3)` è $O(|A|)$. In totale, l'algoritmo converge alla soluzione ottima in tempo $O(|N| + (|N| - 1)|N| + |A|) = O(|N|^2)$. ■

> Si fa notare che, nella dimostrazione di correttezza, abbiamo utilizzato il Lemma 2, che usa l'ipotesi che *tutti i costi* siano non negativi: non basta che non esistano cicli negativi. **Si ribadisce che, per la corretta applicazione dell'algoritmo di Dijkstra, è necessario che *tutti i costi* siano** ≥ 0.

2.2 Implementazioni efficienti

Abbiamo visto come la complessità dell'algoritmo di Dijkstra dipenda dall'operazione di selezione del minimo nell'insieme $\bar{S}$. Sappiamo che la ricerca del minimo in un insieme di

n elementi può essere effettuata in tempi migliori di $O(n)$, al costo di mantenere l'insieme come una struttura dati con proprietà particolari. Per migliorare l'efficienza dell'algoritmo di Dijkstra, quindi, possiamo memorizzare $\bar{S}$ (cioè tutte le etichette π_j per i nodi j da fissare) in una struttura dati a forma di *heap*[2]. Tra le operazioni che possono essere effettuate su un heap, quelle che interessano l'algoritmo di Dijkstra sono le seguenti:

- *create*: crea un heap vuoto, usata in fase di inizializzazione;
- *find-min*: restituisce l'elemento minimo, usata al passo `2)` per selezionare $\hat{v}$;
- *delete-min*: elimina l'elemento minimo dall'heap, usata al passo `2)` per eliminare da $\bar{S}$ l'elemento $\hat{v}$;
- *decrease-key*: diminuisce il valore di un elemento, usata al passo `3)` ogni volta che si migliora un'etichetta.

Ricordiamo che tutte le operazioni devono preservare le caratteristiche dell'heap stesso (relazione tra gli elementi padre-figli e bilanciamento). La complessità delle operazioni dipende dal modo in cui l'heap è implementato. Ad esempio, con un heap binario (ogni nodo ha due figli), e ricordando che l'heap memorizza al massimo $|N|$ etichette, si ha[3]:

- *create*: complessità $O(1)$;
- *find-min*: complessità $O(1)$ (basta leggere la radice dell'albero);
- *delete-min*: complessità $O(\log |N|)$ (una volta eliminata la radice, bisogna selezionare la nuova radice e percorrere l'albero per un numero di passi pari alla profondità dell'albero stesso per ristabilire le proprietà dell'heap);
- *decrease-key*: complessità $O(\log |N|)$ (bisogna ricercare un elemento e ristabilire le proprietà dell'heap, con un numero di passi che dipendono dalla profondità).

Pertanto, al passo `0)` vengono effettuate $O(|N|)$ operazioni di inizializzazione, un'operazione di creazione dell'heap vuoto ($O(1)$) e $|N|-1$ operazioni di inserimento ($O((|N|-1)\log|N|)$); al passo `2)`, un'operazione di ricerca (trova $\hat{v}$) e un'operazione di eliminazione dell'elemento minimo (trasferisci $\hat{v}$ da $\bar{S}$ a S) sono ripetute per $|N|-1$ volte ($O(2(|N|-1)\log|N|)$); al passo `3)`, ammortizzato, si effettuano $|A|$ confronti e al massimo $|A|$ operazioni di diminuzione di un elemento e di aggiornamento del predecessore ($O(|A|(1+\log|N|+1))$. Quindi, in tutto, la complessità dell'algoritmo di Dijkstra implementato con heap binomiale è $O(|A|\log|N|)$.

Citiamo inoltre il fatto che l'implementazione più efficiente dell'algoritmo di Dijkstra fa uso di *heap di Fibonacci* e raggiunge una complessità di $O(|A|+|N|\log|N|)$.

[2]Ricordiamo che un *d-heap* è una struttura dati a forma di albero in cui ogni nodo padre ha d figli; il valore del nodo padre è non maggiore dei nodi figli; i nodi figli (fratelli) sono ordinati in modo che un fratello abbia valore non maggiore dei fratelli a destra. Un heap è sempre *bilanciato*, in modo tale da minimizzare la profondità dell'albero, che così è $\lfloor \log_d n \rfloor + 1$, se n è il numero di elementi da memorizzare.

[3]Non entriamo nei dettagli, si rimanda alle nozioni di algoritmi e strutture dati.