

L'algoritmo di Dijkstra

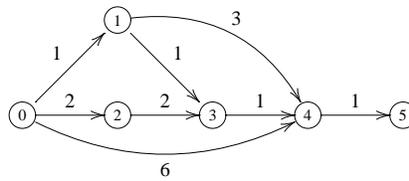
Sebastiano Vigna

22 ottobre 2001

1 Introduzione

Supponiamo di assegnare a ciascuno degli archi a di un grafo orientato G un certo peso intero e positivo p_a . Ai cammini (orientati) nel grafo (cioè sequenze di archi consecutivi) verrà a questo punto assegnato un peso, dato dalla somma dei pesi degli archi che lo compongono. Dati due nodi x e y , il problema del *cammino minimo* consiste nel fornire un cammino da x a y di *peso minimo*.

Se tutti gli archi sono pesati 1 (cioè $p_a = 1$ per ogni arco a) il problema si riduce a quello delle distanze minime, facilmente risolvibile con una visita in ampiezza. Se però ci sono pesi diversi da 1, è facile rendersi conto che una visita in ampiezza non è sufficiente. Ad esempio, se nel grafo seguente vogliamo sapere qual è un cammino minimo dal nodo 0 al nodo 5,



una visita in ampiezza a partire dal nodo 0 assegnerebbe immediatamente al nodo 4 un cammino minimo di peso 6, e (nel caso migliore) al passo successivo un cammino di peso 4 (attraverso il nodo 1), ma, inevitabilmente, al momento di visitare il nodo 4 assegneremo al nodo 5 un cammino di peso 5, mentre “aspettando un po’” se ne sarebbe potuto trovare uno di peso inferiore passante per il nodo 1 e il nodo 3. Per evitare fenomeni di questo tipo, è necessario visitare per primi i nodi che hanno cammini minimi da 0 più brevi.

2 L'algoritmo di Dijkstra

Edsger W. Dijkstra ha proposto nel 1959 [1] un algoritmo molto elegante che risolve questo problema. Vediamo innanzitutto come funziona l'algoritmo in astratto—forniremo poi diverse implementazioni con diversi compromessi tra efficienza, generalità e semplicità.

L'algoritmo di Dijkstra visita i nodi nel grafo, in maniera simile a una ricerca in ampiezza o in profondità. In ogni istante, l'insieme N dei nodi del grafo è diviso in tre parti: l'insieme dei nodi *visitati* V , l'insieme dei nodi di *frontiera* F , che sono successori¹ dei nodi visitati, e i nodi *sconosciuti*, che sono ancora da esaminare. Per ogni nodo z , l'algoritmo tiene traccia di un valore d_z , inizialmente posto uguale a ∞ , e di un nodo u_z , inizialmente indefinito.

¹Un successore di z è un nodo raggiungibile lungo un arco uscente da z .

L'algoritmo consiste semplicemente nel ripetere il seguente passo: si prende dall'insieme F un qualunque nodo z con d_z minimo, si sposta z da F in V , si spostano tutti i successori di z sconosciuti in F , e per ogni successore w di z si aggiornano i valori d_w e u_w . L'aggiornamento viene effettuato con la regola

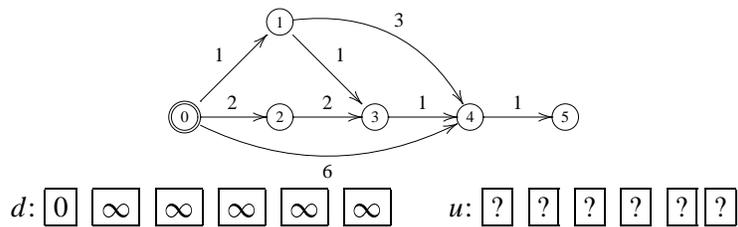
$$d_w \leftarrow \min\{d_w, d_z + p_a\},$$

dove a è l'arco che collega z a w . Se il valore di d_w è stato effettivamente modificato, allora u_w viene posto uguale z .

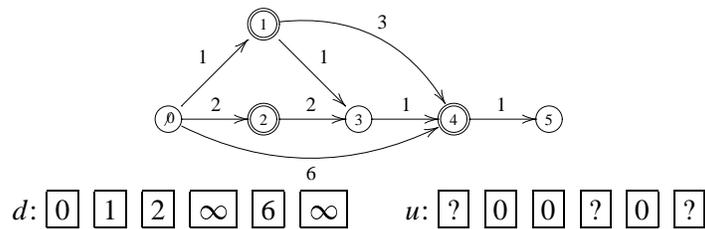
La regola segue un'idea piuttosto naturale: se sappiamo che con peso d_z possiamo arrivare fino a z , allora arrivare a w non può costare più di arrivare a z e spostarsi lungo un arco fino a w . L'aggiornamento di u_w ci permette di ricordare che, al momento, il cammino di peso minimo che conosciamo per arrivare da x in w ha come penultimo nodo z .

L'algoritmo parte con $V = \emptyset$, $F = \{x\}$, $d_x = 0$ e prosegue finché y non viene visitato, o finché $F = \emptyset$: in questo caso, y non è raggiungibile da x lungo un arco orientato. Se usciamo solo nel secondo caso, alla fine dell'algoritmo d_z contiene, per ogni nodo z , il peso di un cammino minimo da x a z ; inoltre, il vettore u permette di ricostruire l'*albero dei cammini minimi con origine in x* .

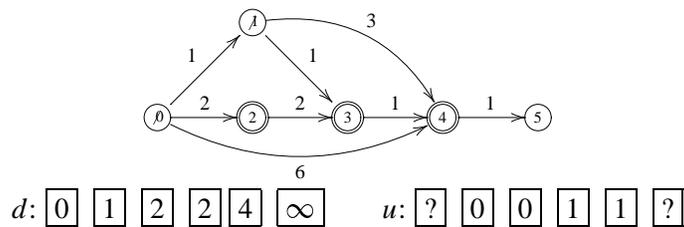
Mostriamo il funzionamento dell'algoritmo sull'esempio che abbiamo visto all'inizio, e supponiamo di voler trovare il cammino minimo da 0 a 3. La configurazione iniziale è quindi



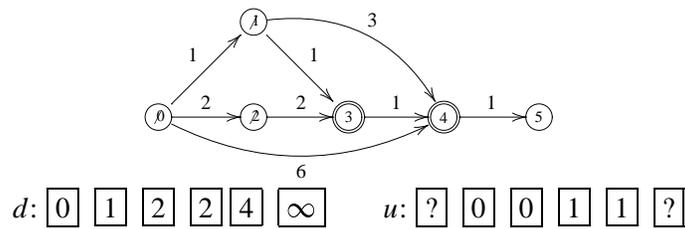
Al primo passo, il nodo 0 viene visitato (nella figura, viene barrato), mentre 1 e 2 passano nella frontiera (denotata da un ulteriore cerchio attorno al nodo).



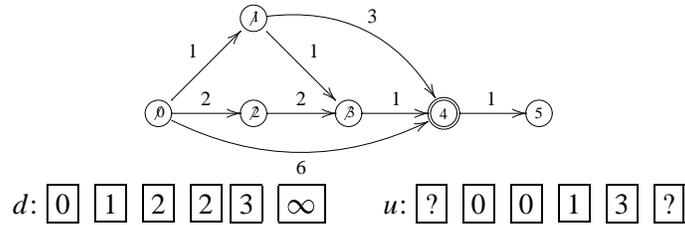
Ora dobbiamo andare a prendere il nodo della frontiera che ha distanza minima da 0, e possiamo vedere come l'algoritmo di Dijkstra risolve i problemi posti da una visita in ampiezza: essendo i nodi 3 e 4 molto distanti da 0 (in termini di cammini di peso minimo), prima visitiamo il nodo 1, il che ha l'effetto di aggiornare d_3 , d_4 , u_3 e u_4 :



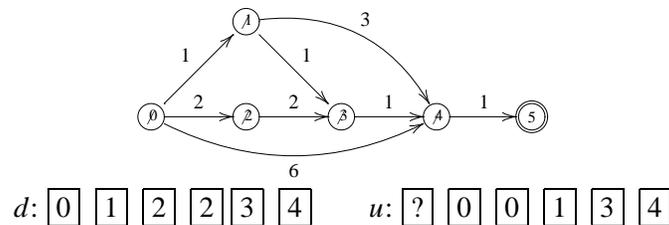
Ancora una volta, invece di scegliere i nodi aggiunti inizialmente alla frontiera, proseguiamo con quello che ha d minimo, cioè 2. L'effetto è solo quello di spostare il nodo tra quelli visitati, dato che per l'unico successore (il nodo 3) conosciamo già un cammino di peso minimo migliore di quello che passerebbe per il nodo 2:



Visitiamo ora il nodo 3, aggiornando nuovamente d_4 e u_4 :



A questo punto visitiamo il nodo 4



e la visita successiva ci dirà che esiste un cammino minimo da 0 e 5 di distanza 4: inseguendo i puntatori all'indietro in u il cammino risulta in effetti essere 0, 1, 3, 4, 5.

Per dimostrare che l'algoritmo funziona correttamente, bisogna dimostrare che a ogni passo vengono mantenute le seguenti proprietà:

1. i nodi in F sono tutti e soli i successori non ancora visitati di nodi visitati (cioè sono i successori di nodi in V , a parte quelli già in V);
2. per ogni nodo z in V , d_z è il peso di un cammino minimo da x in z ; inoltre, uno dei cammini minimi ha come penultimo nodo u_z ;
3. per ogni nodo z in F , d_z è il peso minimo di un cammino di x in z che passa solo per nodi in V , ad esclusione di z stesso; inoltre, un tale cammino ha come penultimo nodo u_z .

Le proprietà sono vere in maniera ovvia al primo passo, ed è anche banale che la proprietà (1) rimanga sempre vera dopo il primo passo.

Dobbiamo innanzitutto mostrare che quando spostiamo un nodo z con d_z minimo da F in V , d_z è il peso di un cammino minimo da x a z . Infatti, se per assurdo ci fosse un cammino di peso inferiore d , esso dovrebbe partire da x , viaggiare entro V , passare per qualche nodo z' in F diverso da z (altrimenti la proprietà (3)

sarebbe violata), e infine arrivare in z . Ma allora il cammino da x a z' avrebbe peso inferiore a d_z , e quindi $d_{z'} < d_z$, contrariamente a quanto assunto. Quindi la proprietà (2) è ancora verificata.

Dobbiamo ora dimostrare che (3) rimane vera. Per i nodi in F che *non* sono successori di z , la proprietà rimane ovviamente vera, perché l'aggiunta di z a V non ha generato nuovi cammini che li raggiungono entro V . Per gli altri nodi, la regola di aggiornamento di d e u fa sì che la proprietà (3) rimanga vera.

Concludiamo che non appena y cade dentro V , possiamo fermare l'algoritmo e restituire d_y come peso di un cammino minimo. Inoltre, per costruire un cammino minimo ci basta inseguire all'indietro i puntatori u_y , u_{u_y} , ecc., fino a x . Dato che qualunque porzione di un cammino di peso minimo è a sua volta un cammino di peso minimo (o potremmo sostituire la porzione con una di peso minore, ottenendo un cammino complessivo più breve), in questo modo ricostruiremo un cammino minimo da x a y .

Infine, se prima che y cada dentro V succede che $F = \emptyset$, allora la proprietà (1) garantisce che non ci sono altri nodi raggiungibili da x , e che, in particolare, y non è raggiungibile da x .

3 Implementazioni

Per semplificare un esempio di implementazione, assumiamo un limite superiore MAXV al numero di vertici, e calcoliamo solo d , tralasciando u (le modifiche per calcolare anche u sono banali).

3.1 Matrice di adiacenza

Le strutture dati in gioco sono le seguenti:

```
int n;
int G[MAXV][MAXV];
int d[MAXV];
char v[MAXV];
```

dove G è la matrice di adiacenza pesata, n il numero di nodi e v è un vettore di booleani che serve a marcare i nodi visitati (d è il vettore delle distanze).

La prima implementazione è quella immediata: a ogni passo scandiamo il vettore d e cerchiamo, tra i nodi non visitati, quello con d minimo. A questo punto scandiamo la riga corrispondente della matrice di adiacenza e aggiorniamo d per tutti i successori del nodo scelto:

```
for(i=0; i<n; i++) d[i] = INT_MAX;
d[x] = 0;

for(;;) {

    for(m=INT_MAX, i=0; i<n; i++)
        if (!v[i] && d[i] <= m) m = d[j = i];

    v[j] = 1;

    if (j == y || m == INT_MAX) break;

    for(i=0; i<n; i++)
        if (G[j][i] && d[i] > d[j] + G[j][i])
```

```

        d[i] = d[j] + G[j][i];
    }

```

La complessità di questo algoritmo è $O(n^2)$, dato che nel caso peggiore y viene raggiunto per ultimo, e quindi dobbiamo eseguire n passi, ciascuno dei quali richiede $O(n)$ iterazioni.

3.2 Liste di adiacenza

Per migliorare le prestazioni dell'algoritmo su grafi sparsi, cioè in cui il numero di archi a è molto più piccolo di n^2 (per esempio, per i grafi planari è lineare in n) rappresentati tramite liste di adiacenza, è possibile utilizzare una *coda con priorità*. In una coda normale (come quella utilizzata per le visite in ampiezza) il primo nodo inserito è anche il primo a essere rimosso. In una coda con priorità, invece, ogni nodo viene inserito con una priorità associata, e il nodo rimosso è quello di priorità maggiore. A questo punto la ricerca del minimo ha un costo pari all'estrazione dell'elemento di priorità massima dalla coda, e l'aggiornamento dei valori di d , essendo effettuato una sola volta per ogni arco, è dato da a volte il costo di inserimento nella coda o di aggiornamento di priorità. Come vedremo, è possibile implementare una coda con priorità in modo che qualunque operazione sia logaritmico nel numero di elementi della coda. Dato che la coda non contiene mai più di n elementi, l'algoritmo richiede complessivamente $O(a \log n)$ operazioni, e se $a = o(n^2 / \log n)$ risulta vantaggioso.

In questo caso, le strutture dati sono

```

int n;

struct arc {
    struct arc *next;
    int v;
    int w;
};

struct arc *adj[MAXV];

int d[MAXV];
char v[MAXV];

```

La struttura del grafo viene memorizzata costruendo una lista di successori per ogni nodo (la lista indica anche il peso dell'arco di collegamento).

La coda con priorità mette a disposizione le seguenti operazioni:

```

void enqueue(int v, int p); /* Aggiunge v con priorità p */
int dequeue(void);        /* Rimuove e restituisce il
                           nodo con massima priorità */
void update(int v, int p); /* Aggiorna la priorità di v a p;
                           se v non è nella coda, lo
                           inserisce */
int empty(void);         /* Controlla se la coda è vuota */

```

Per semplicità, anche se questo non è completamente "pulito", assumiamo che le operazioni sulla coda manipolino direttamente d . Ovviamente, la priorità è più alta quanto più d è piccolo.

L'algoritmo a questo punto è semplicissimo:

```

for(i=0; i<n; i++) d[i] = INT_MAX;
enqueue(x, 0);

while(!empty()) {
    j = dequeue();

    p = adj[j];

    while(p) {
        update(p->v, d[j] + p->w);
        p = p->next;
    }
}

```

La complessità dell'algorithmo è in effetti scaricata sulla coda con priorità, la cui implementazione (tutt'altro che banale) tramite uno *heap* viene descritta con cura in un'altra dispensa.

Riferimenti bibliografici

- [1] Edsger W. Dijkstra. A note on two problem in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.