Capitolo 11

Scelta della struttura dati

Nota. La prima cosa da fare quando si progetta un algoritmo è capire quale sia la struttura dati adatta a risolvere quel particolare problema.

11.1 Cammini minimi, sorgente singola

11.1.1 Problema dei cammini minimi

Definizione 11.1.1 (Costo del cammino)**.** Dato un cammino $p = \langle v_1, v_2, \ldots, v_k \rangle$ con $k > 1$, il *costo del cammino* (*weight of the path*) è dato da

$$w(p) = \sum_{i=2}^{k} w(v_{i-1}, v_i)$$

Ossia dalla somma dei singoli pesi dei lati che compongono il percorso.

Definizione del problema Dati in input un grafo orientato $G = (V, E)$, un nodo sorgente s ed una funzione di peso $w: E \to R$ (che associa ad ogni arco un numero reale che rappresenta il peso).

Trovare un cammino da s ad u, per ogni nodo $u \in V$, il cui costo sia minimo, ovvero più piccolo o uguale al costo di qualunque altro cammino da s a u.

Nota. Non ci limitiamo a trovare un solo percorso, ma tutti i cammini da un nodo a tutti gli altri nodi.

Panoramica sul problema Per risolvere il problema del cammino minimo fra una coppia di vertici, si risolve il problema di cammini minimi da sorgente unica (si trovano tutti i cammini che partono da un nodo) e si estrae il cammino richiesto. Per quanto riguarda il *caso pessimo* non si conoscono algoritmi che abbiamo tempo di esecuzione migliore.

In alcuni casi gli archi possono avere peso negativo. Questo influisce sul problema (se è ben definito oppure no) e sulla soluzione (in assenza di archi negativi si (possono?) devono utilizzare tecniche diverse).

Nell'algoritmo di Dijkstra si suppone che tutti gli archi abbiano peso positivo, mentre nell'algoritmo di Bellman-Ford gli archi possono avere peso negativo, ma non possono esistere cicli di peso negativo.

Nota. In generale possiamo ammettere pesi negativi, ma non cicli negativi.

Considerazioni sui cicli Se esiste un ciclo di peso negativo raggiungibile dalla sorgente, non esistono cammini finiti di peso minimo; per qualunque cammino, basterà passare per un ciclo negativo più volte per ottenere un ciclo di costo inferiore.

Ovviamente, in un cammino minimo *non è possibile la presenza di un ciclo di peso positivo.* Mentre i cicli di peso nullo possono essere banalmente eliminati dal cammino minimo, in quanto inutili e ridondanti.

11.1.2 Sottostruttura ottima

Nota che due cammini minimi possono avere un tratto in comune, ma non possono convergere in un nodo comune B dopo aver percorso un tratto iniziale distinto.

(a) Due cammini minimi che hanno un tratto in comune a partire dal nodo B

(b) Due cammini che convergono in un nodo comune B dopo aver percorso un tratto iniziale distinto.

Figura 11.1: La figura 11.1a è una condizione ammissibile, mentre 11.1b non lo è.

Definizione 11.1.2 (Albero dei cammini minimi)**.** L'albero dei cammini minimi è un albero di copertura radicato in s avente un cammino da s a tutti i nodi raggiungibili da s.

Nota. Non confonderlo con gli alberi di copertura di peso minimo.

Soluzione ammissibile Una soluzione *ammissibile* può essere descritta da un *albero di copertura* T radicato in s e da un *vettore delle distanze* d, i cui valori $d[u]$ rappresentano il costo del cammino da s a u in T.

(a) Soluzione ammissibile (albero di peso minimo)

(b) Soluzione ottima (albero dei cammini minimi)

Figura 11.2: Nota la differenza fra i due alberi: a sinistra l'albero di *peso minimo* che rappresenta una soluzione ammissibile, ma non ottima, mentre a destra l'albero dei cammini minimi, ossia l'insieme dei percorsi che minimizzano il peso fra il nodo sorgente e tutti gli altri nodi, il quale rappresenta la soluzione ottima.

Nota. In questo problema devo trovare i percorsi che minimizzano il peso fra un nodo e tutti gli altri nodi, e **non**, come sembra spontaneo fare, l'albero che ha complessivamente peso minimo.

Rappresentazione dell'albero Utilizziamo la rappresentazione basata su vettore dei padri, così come abbiamo fatto con le visite in ampiezza/profondità.

11.1.3 Teorema di Bellman

Teorema 1 (Teorema di Bellman)**.** *Una soluzione ammissibile T è (anche) ottima se e solo se:*

$$d[v] = d[u] + w(u,v) \text{ per ogni arco } (u,v) \in \boldsymbol{T}$$
$$d[v] \leqslant d[u] + w(u,v) \text{ per ogni arco } (u,v) \in \boldsymbol{E}$$

Dimostrazione per assurdo (parte 1). Sia T una soluzione ottima. Consideriamo un qualunque arco $(u,v) \in E$ e sia $w(u,w)$ la sua lunghezza.

Ovviamente se $(u,v) \in T$, allora $d[v] = d[u] + w(u,v)$. Invece se $(u,v) \notin T$, allora poiché T è ottimo, deve risultare $d[v] \leqslant d[u] + w(u,v)$, altrimenti esisterebbe nel grafo G un cammino da s a v più corto di quello in T, che è *assurdo* perché abbiamo ipotizzato che T fosse ottima. □

Dimostrazione per assurdo (parte 2). Supponiamo per assurdo che il cammino da s a u in T non sia ottimo. Allora esiste un cammino da s a u con distanza $d'[u] < d[u]$. Sia $d'[v]$ la distanza da s ad un generico

nodo v che appare in tale cammino. Poichè $d'[s] = d[s] = 0$, ma $d'[u] < d[u]$, esiste un arco (h,k) per cui $d'[h] \geqslant d'[h]$ e $d'[k] < d[k]$. Per costruzione $d'_h + w(h,k) = d'_k$. Per ipotesi $d_h + w(h,k) \geq d_k$. Combinando queste due relazioni, si ottiene:

$$d'_k = d'_h + w(h,k) \geqslant d_h + w(h,k) \geqslant d_k$$

che contraddice l'ipotesi. □

11.1.4 Verso un algoritmo

Algoritmo 0: Algoritmo prototipo per il calcolo dei cammini minimi

```
// Algoritmo prototipo dei cammini minimi
(int, int) CamminiMinimi(GRAPH G, NODE s)
    // Inizializza T ad una foresta di copertura composta da nodi isolati
    // Inizializza d con una sovrastima della distanza (d[s] = 0, d[x] = +∞)
    while ∃(u,v): d[u] + G.w(u,v) < d[v] do
        // Esiste un arco che mi permette di migliorare la stima
        d[v] = d[u] + w(u,v) // Aggiorno la distanza
        // Sostituisci il padre di v in T con u
    return (T, d)
```

Commento L'algoritmo prende in input un grafo e il nodo sorgente. I pesi vengono estratti dalla struttura dati GRAPH. Inizializziamo d con una sovrastima della distanza; $d[s] = 0$ sta a significare che la sorgente ha distanza da sè stessa pari a 0 (caso base) e con $d[x] = +\infty$ indico che la distanza di tutti gli altri nodi, fintanto che non è nota, è pari a $+\infty$.

Nota. Se al termine dell'esecuzione dell'algoritmo qualche nodo mantiene una distanza infinita, allora esso non è raggiungibile dalla sorgente.

Algoritmo 0: Algoritmo generico per il calcolo dei cammini minimi

```
(int, int) CamminiMinimi(GRAPH G, NODE s)
    // Inizializzazione dei vettori
    int[] d ← new int[1...G.n]                                   // distanze dalla sorgente
    int[] T ← new int[1...G.n]                                   // vettore dei padri
    boolean[] b ← new boolean[1...G.n]                           // per sapere in tempo costante se u ∈ S

    // Inizializzo tutti i nodi tranne la sorgente
    foreach u ∈ G.V − {s} do
        T[u] ← nil  // non hanno padri
        d[u] ← +∞  // non li ho ancora raggiunti
        b[u] ← false  // non appartengono ancora all'insieme

    // Inizializzo la sorgente
    T[s] ← nil // non ha padre
    d[s] ← 0 // per convenzione
    b[s] ← true // appartiene all'insieme
(1) STRUTTURADATI S ← StrutturaDati
    S.aggiungi(s)

    while not S.isEmpty do
(2)     int u ← S.estrai  // estraggo un nodo
        b[]u ← false  // non è più contenuto nella struttura dati

        foreach v ∈ G.adj(u) do // per tutti i vicini
            if d[u] + G.w(u,v) < d[v] then // se migliora la stima
                if not b[v] then // se non fa già parte dell'insieme
(3)                 S.aggiungi(v) // aggiungilo
                    b[v] ← true // fa parte dell'insieme
                else
(4)                 // Azione da intraprendere nel caso v sia già presente in S

                // aggiorno i vettori
                T[v] ← u
                d[v] ← d[u] + G.w(u,v)

    return (T, d)
```

Commento **boolean**[] ci permette di sapere in tempo costante se un certo nodo appartiene ad una struttura dati oppure no, non sarà necessario quando implementeremo realmente il codice.

11.2 Algoritmo di Dijkstra

Il seguente algoritmo è stato sviluppato da Edsger W. Dijkstra nel 1956, pubblicato nel 1959. Nella versione originale veniva utilizzato per trovare la distanza minima fra due nodi sfruttando il concetto di coda con priorità. Tieni conto però che le code di priorità basate sugli heap binari sono state proposte nel '64, infatti l'algoritmo che di solito viene considerato di Dijkstra è in realtà la versione modificata di Johnson.

Implementazione L'algoritmo utilizza una coda con priorità basata su vettore.

Algoritmo 1: Algoritmo di Dijkstra

```
(int[], int[]) CamminiMinimi(GRAPH G, NODE s)
(1)  PRIORITYQUEUE S ← PriorityQueue  // O(n)·1
     S.inserisci(s, 0)
     while not S.isEmpty do // O(n)
(2)      // O(n) vettore ordinato / O(log n) heap binario
         int u ← S.deleteMin
         b[u] ← false
         foreach v ∈ G.adj(u) do
             if d[u] + G.w(u,v) < d[v] then
                 if not b[v] then
                     // O(1)·n vettore ordinato / O(log n)·n heap binario
(3)                  S.inserisci(v, d[u] + G.w(u,v))
                     b[v] ← true
                 else
                     // O(1)·m vettore ordinato / O(log n)·m heap binario
(4)                  S.decrease(v, d[u] + G.w(u,v))
                 // aggiorno i vettori
                 T[v] ← u
                 d[v] ← d[u] + G.w(u,v)
     return (T, d)
```

Analisi della complessità

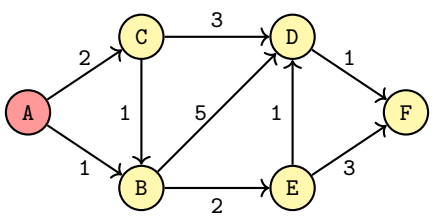
① Viene creato un vettore di dimensione n. Ogni elemento u-esimo rappresenta il nodo u. Le priorità (distanze) vengono inizializzate ad $+\infty$. La priorità di s è posta uguale a 0. Per un costo di $O(n)$;

② Si ricerca il minimo all'interno del vettore, una volta trovato si "cancella" la sua priorità. Per un costo complessivo di $O(n)$ (viene svolto all'interno di un ciclo);

③ Si registra la priorità nella posizione corrispondente all'indice v. Per un costo di $O(1)$;

④ Si aggiorna la priorità nella posizione corrispondente all'indice v. Per un costo di $O(1)$.

Esempio di esecuzione



		A	B	C	D	E	F
A	0	**0**	~~∅~~	~~∅~~	~~∅~~	~~∅~~	~~∅~~
B	∞	1	**1**	~~1~~	~~1~~	~~1~~	~~1~~
C	∞	2	2	**2**	~~2~~	~~2~~	~~2~~
D	∞	∞	6	5	4	**4**	4
E	∞	∞	3	3	**3**	~~3~~	~~3~~
F	∞	∞	∞	∞	6	5	**5**

- ogni colonna contiene lo stato del vettore d all'inizio di ogni ripetizione del ciclo while **not** S.isEmpty

- ogni riga v rappresenta l'evoluzione dello stato dell'elemento $d[v]$;
- la legenda delle colonne rappresenta il nodo che viene estratto.

Correttezza Tutte le volte che estraiamo un nodo, quel nodo ha una distanza (priorità) positiva ed estraiamo nodi a distanza progressivamente crescenti. Se estraggo un nodo dalla coda tutti gli altri nodi hanno distanze più grandi. Tutte le volte che estraggo un nodo la sua distanza non può più essere modificata. Ed è questo il motivo per cui l'algoritmo di Dijkstra funziona (bene) solo con pesi positivi.

Nota. L'algoritmo di Dijkstra funziona correttamente solo con pesi positivi.

11.2.1 Correttezza per pesi positivi

Ogni nodo viene estratto una e una sola volta. Al momento dell'estrazione la sua distanza è minima.

Dimostrazione per induzione sul numero k di nodi estratti. Per $k = 0$ (caso base) è vero poiché $d[s] = 0$ e non ci sono lunghezze negative. Supponiamo che sia vero per i primi $k-1$ nodi (ipotesi induttiva). Quando viene estratto il k-esimo nodo u, la sua distanza $d[u]$ dipende dai $k-1$ nodi già estratti (passo induttivo). Non può quindi dipendere dai nodi ancora da estrarre, che hanno distanza $\geqslant d[u]$. Di conseguenza $d[u]$ è minimo e u non verrà più re-inserito, perché non ci sono distanze negative. □

11.3 Algoritmo di Johnson

Analisi della complessità Con l'introduzione dell'heap binario nel '64 le operazioni che prima venivano svolte con complessità $\mathcal{O}(n)$ sul vettore ordinato ora hanno complessità $\mathcal{O}(\log n)$. Di conseguenza la complessità totale dell'algoritmo scende da $\mathcal{O}(n^2)$ a $\mathcal{O}(m \log n)$.

Per *grafi densi* non conviene utilizzare uno heap binario in quanto $m = \Theta(n^2)$ e di conseguenza l'algoritmo avrebbe una complessità di $\mathcal{O}(n^2 \log n)$ si preferisce quindi la versione con vettore ordinato per una complessità di $\mathcal{O}(n^2)$, mentre per *grafi sparsi* $m = \Theta(n)$ e l'algoritmo $\mathcal{O}(n \log n)$ che è migliore di $\mathcal{O}(m \log n)$.

11.4 Algoritmo di Fredman-Tarjan

Analisi della complessità Sfruttando un heap di fibonacci l'operazione di decrease ha costo ammortizzato costante; così facendo hanno abbassato la complessità a $\mathcal{O}(m + n \log n)$. Per *grafi sparsi* produce un miglioramento nella complessità.

11.5 Algoritmo di Bellman-Ford-Moore

Nota. É computazionalmente più pesante dell'algoritmo di Dijkstra ma può lavorare anche con archi di peso negativo.

Implementazione Utilizza una coda senza priorità.

Algoritmo 2: Algoritmo di Bellman-Ford-Moore

```
(int[], int[]) CamminiMinimi(GRAPH G, NODE s)
(1)  QUEUE S ← Queue
     S.enqueue(s)  // metto in coda il nodo sorgente
     while not S.isEmpty do // O(n)
(2)      int u ← S.dequeue  // O(1·n)
         b[u] ← false
         foreach v ∈ G.adj(u) do
             if d[u] + G.w(u,v) < d[v] then
                 if not b[v] then
(3)                  // lo metto in cosa quando c'è un miglioramento
                     S.enqueue(v)  // O(m·n)
                     b[v] ← true
     return (T, d)
```

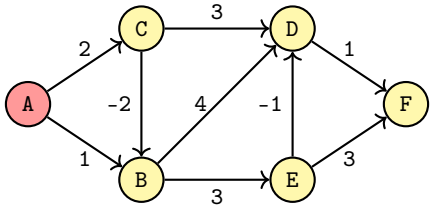
① Viene creata una coda di dimensione n. Per un costo computazionale di $O(n)$;

② Viene estratto il prossimo elemento della coda. Per un costo computazionale di $O(1)$;

③ Si inserisce l'indice v in coda. Per un costo computazionale di $O(1)$.

L'inserimento in coda può essere fatto più di una volta durante il ciclo di esecuzione dell'algoritmo, al contrario di quel che accade nell'algoritmo di Dijkstra. Il passo ④ non è necessario in quanto non c'è una priorità da aggiornare.

Esempio di esecuzione



Come leggere la tabella sottostante
- la prima riga contiene l'elemento estratto dalla coda;
- l'ultima riga contiene lo stato della coda;
- i vettori delle distanze sono rappresentate dalle colonne.

nodo estratto:			A	B	C	D	E	B	F	D	E	D	F
vettori	A	0	0	0	0	0	0	0	0	0	0	0	0
	B	∞	1	1	0	0	0	0	0	0	0	0	0
	C	∞	2	2	2	2	2	2	2	2	2	2	2
	D	∞	5	5	5	3	3	3	3	3	2	2	2
	E	∞	∞	4	4	4	4	3	3	3	3	3	3
	F	∞	∞	∞	∞	6	5	5	5	4	4	3	3
queue:	S	A	BC	CDE	DEB	EBF	BFD	FDE	DE	E	D	F	

Dimostrazione di correttezza Per dimostrare la correttezza dell'algoritmo dobbiamo dare la definizione di "passata". Una passata si definisce ricorsivamente come segue: per $k = 0$, la zeresima passata consiste nell'estrazione del nodo s dalla coda S, per $k > 0$ la k-esima passata consiste nell'estrazione di tutti i nodi presenti in S al termine della passata $k-1$-esima. Viene data solo un'intuizione, non viene fatta una dimostrazione formale. Al termine della passata k, i vettori T e d descrivono i cammini minimi di lunghezza al più k. Al termine della passata $n-1$, i vettori T e d descrivono i cammini (di lunghezza al più $n-1$).

Analisi della complessità L'inserimento in coda dei nodi avviene solo una volta per un costo di $\mathcal{O}(1)$. Un nodo può essere estratto e reinserito al massimo $n-1$ volte, per un costo di $\mathcal{O}(n^2)$. Quando avviene un miglioramento la distanza viene aggiornata, per un costo i $\mathcal{O}(n\dot{m})$. Il costo complessivo dell'algoritmo risulta quindi $\mathcal{O}(n\dot{m})$.

Cammini minimi su DAG I cammini minimi su DAG sono sempre ben definiti; anche in presenza di pesi negativi, in quanto non esistono cicli (nè tantomeno quelli negativi). É possibile rilassare gli archi *in ordine topologico*, *una volta sola*. Non essendoci cicli, non c'è modo di tornare su un nodo già visitato ed abbassare il valore della sua distanza (il suo campo d).

Si utilizza quindi l'ordine topologico.

Algoritmo 3: Algoritmo di Bellman-Ford-Moore applicato su DAG

```
(int[], int[]) CamminiMinimi(GRAPH G, NODE s)
    int[] d = new int[1...G.n]                    // d[u] è la distanza da s a u
    int[] T = new int[1...G.n]                    // T[u] è il padre da u nell'albero T

    // Inizializzo i vettori
    foreach u ∈ G.V − {s} do
        T[u] = nil
        d[u] = +∞

    // Inizializzo la sorgente
    T[s] = nil
    d[u] = 0

    // Effettuo l'ordinamento topologico dei nodi nel DAG
    STACK S = topSort

    // fintanto che la pila non è vuota
    while not S.isEmpty do
        u = S.pop // estraggo un nodo
        foreach v ∈ G.adj(v) do // per ogni nodo adiacente
            if d[u] + G.w(u,v) < d[v] then // se il peso è migliore di quello presente
                // aggiorno il peso
                T[v] = u
                d[v] = d[u] + G.w(u,v)

    // restituisco il vettore dei padri e il vettore delle distanze
    return (T, d)
```

11.5.1 Riassumendo

Tabella 11.1: Quale complessità preferire?

Algoritmo	Complessità	Input
Dijkstra	$\mathcal{O}(n^2)$	Pesi positivi, grafi denso
Johnson	$\mathcal{O}(m \log n)$	Pesi positivi, grafi sparso
Fredman-Tarjan	$\mathcal{O}(m + n \log n)$	Pesi positivi, grafi denso, dimensioni molto grandi
Bellman-Ford	$\mathcal{O}(m \cdot n)$ $\mathcal{O}(m+n)$	Pesi negativi DAG
BFS	$\mathcal{O}(m+n)$	Senza pesi

11.5.2 Cammini minimi, sorgente multipla

Vogliamo cercare i cammini minimi fra tutti i nodi.

Tabella 11.2: Quale complessità preferire?

Algoritmo	Complessità	Input
Pesi positivi, grafo denso	$\mathcal{O}(n \cdot n^2)$	Applicazione ripetuta (n) dell'algoritmo di Dijkstra
Pesi positivi, grafo sparso	$O(n \cdot (m \log n))$	Applicazione ripetuta dell'algoritmo di Johnson
Pesi negativi	$O(n \cdot nm)$	Applicazione ripetuta di Bellman-Ford (sconsigliata)
Pesi negativi, grafo denso	$O(n^3)$	Algoritmo di **Floyd e Warshall**
Pesi negativi, grafo sparso	$O(nm \log n)$	Algoritmo di **Johnson per sorgente multipla**

L'algoritmo di Bellman-Ford è sconsigliato per grafi densi perché può arrivare ad avere una complessità di $\mathcal{O}(n^4)$, mentre l'algoritmo di Floyd e Warshall ha una complessità di $\mathcal{O}(n^3)$ indipendentemente dalla forma del grafo.

11.6 Algoritmo di Floyd-Warshall

Utilizza la programmazione dinamica. Ci riesce ridefinendo la definione del costo di cammino in modo tale che possa essere calcolato in modo ricorsivo.

Definizione 11.6.1 (Cammini minimi k-vincolati)**.** Sia k un valore in $\{0, \dots, n\}$. Diciamo che un cammino p^k_{xy} è un cammino minimo k-vincolato fra x ed y se esso ha il costo minimo fra tutti i cammini fra x e y che non passano per nessun vertice in $v_{k+1}, \dots, v_n$ (x e y sono esclusi dal vincolo).

Nota. Assumiamo, come abbiamo sempre fatto, che esista un ordinamento fra i nodi del grafo $v_1, v_2, \dots, v_n$.

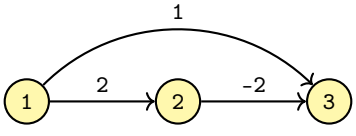
Definizione 11.6.2 (Distanza k-vincolata)**.** Denotiamo con $d^k[x][y]$ il costo totale del cammino minimo k-vincolato fra x e y, se esiste.

$$d^k[x][y] = \begin{cases} w(p^k_{xy}) & \text{se esiste } p^k_{xy} \\ +\infty & \text{altrimenti} \end{cases}$$

La formulazione ricorsiva è la seguente:

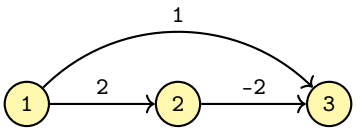
$$d^k[x][y] = \begin{cases} d[x][y] & k = 0 \\ d^{k-1}[x][y] \vee d^{k-1}[x][k] \wedge d^{k-1}[k][y] & k > 0 \end{cases}$$

Ad esempio:



$$\begin{aligned} d^0[1][3] &= 1 \\ d^1[1][3] &= 1 \\ d^2[1][3] &= \min(d^1[1][3], d^1[1][2] + d^2[2][3]) \\ &= \min(1, +2 - 2) \\ &= \min(1, 0) = 0 \end{aligned}$$

Oltre a definire la matrice d, calcoliamo una matrice T dove $T[x][y]$ rappresenta il predecessore di y nel cammino più breve da x a y. Ad esempio:



$T[1][2] = 1$
$T[2][3] = 2$
$T[1][3] = 2$

Algoritmo 4: Algoritmo di Floyd-Warshall

```
(int[], int[]) CamminiMinimi(GRAPH G, NODE s)
    // Credo le matrici
    int[][] d = new int[1...n][1...n]                    // matriche delle distanze
    int[][] T = new int[1...n][1...n]                    // matriche dei padri (predecessori)

    // Inizializzo i vettori
    foreach u, v ∈ G.V do
        d[u][v] = +∞
        T[u][v] = nil

    // Inserisco i valori iniziali
    foreach u ∈ G.V do
        foreach v ∈ G.adj(u) do
            d[u][v] = G.w(u, v)
            T[u][v] = u

    // Aggiorno le distanze
    from k ← 1 until G.n do
        foreach u ∈ G.V do
            foreach v ∈ G.adj(u) do
                if d[u][k] + d[k][v] < d[u][v] then
                    d[u][v] = d[u][k] + d[k][v]
                    T[u][v] = T[k][v]

    return d
```

11.7 Algoritmo di Warshall

Definizione 11.7.1 (Chiusura transitiva)**.** La chiusura transitiva $G^* = (V, E^*)$ di un grafo $G = (V, E)$ è il grafo orientato tale che $(u, v) \in E^*$ se e solo esiste un cammino da u a v in G.

Supponendo di avere il grafo G rappresentato da una matrice di adiacenza M, la matrice M^n rappresenta la matrice di adiacenza di G^*.

La formulazione ricorsiva è la seguente:

$$M^k[x][y] = \begin{cases} M[x][y] & k = 0 \\ M^{k-1}[x][y] \vee M^{k-1}[x][k] \wedge M^{k-1}[k][y] & k > 0 \end{cases}$$

Conclusioni

Abbiamo visto una panoramica dei più importanti algoritmi per la ricerca dei cammini minimi. Esistono anche altri algoritmi, in particolare l'algortimo A^* utilizza euristiche per velocizzare la ricerca.