1. Strutture di dati e progettazione di algoritmi

Dato un problema, non ci sono "ricette generali" per progettare algoritmi efficienti che lo risolvano. Al contrario, accade molto spesso di ripartire praticamente da zero ogniqualvolta si debba progettare un nuovo algoritmo!

Nel progettare algoritmi, comunque, possono essere evidenziate quattro "fasi" comuni attraverso le quali occorre usualmente passare:

(1) *Classificazione del problema*. Innanzitutto, si cerca di stabilire se il problema da risolvere fa parte di una classe più ampia di problemi aventi caratteristiche comuni. Di solito, si usa distinguere tre classi di problemi:

Problemi decisionali la cui risposta è "sì" o "no" (o vero o falso) a seconda che il dato d'ingresso soddisfi o no una certa proprietà (per esempio: verificare se un grafo non orientato è connesso, o decidere se esiste una foglia di un albero radicato il cui livello sia maggiore di una certa costante k);

Problemi di ricerca dove, tra tutte le possibili soluzioni, si vuole trovarne una, detta *soluzione ammissibile*, che soddisfi una certa condizione (per esempio: ordinare una sequenza di numeri in senso crescente, o trovare una foglia di un albero radicato il cui livello sia maggiore di una certa costante k);

Problemi di ottimizzazione nei quali alle soluzioni ammissibili è associata una misura (o costo, o obiettivo) e si vuole trovare una *soluzione ottima*, cioè una soluzione ammissibile la cui misura sia minima o massima (per esempio: trovare una foglia di un albero radicato il cui livello sia massimo).

(2) *Caratterizzazione della soluzione*. In secondo luogo, si prova a caratterizzare matematicamente la soluzione del problema. Nella maggior parte dei casi, la matematica usata è "elementare", ma ciò non significa che i risultati siano banali da dimostrare. Tale caratterizzazione di solito suggerisce un algoritmo di risoluzione semplice ma non molto efficiente (per esempio: in una sequenza ordinata di numeri, il primo è minore degli altri $n-1$, il secondo è minore dei rimanenti $n-2$, e così via fino all'ultimo che è il massimo; questa caratterizzazione suggerisce un algoritmo di ordinamento basato sulla ricerca iterata del minimo, come illustrato nel Capitolo 2);

(3) *Tecnica di progettazione*. Successivamente, si cerca di applicare certe tecniche di progettazione di algoritmi per rendere gli algoritmi più veloci. Le tecniche principali sono: *divide-et-impera*, programmazione dinamica, *greedy*, ricerca locale, *backtrack*, algoritmi probabilistici. La prima di queste tecniche è già stata implicitamente introdotta, e consiste nel partizionare il problema in sottoproblemi più piccoli, risolubili indipendentemente, le cui soluzioni possono essere ricombinate per ottenere la soluzione del problema di partenza (ad esempio: la procedura di ricerca binaria vista nel Capitolo 1);

(4) *Strutture di dati*. Infine, si vede se c'è un modo astuto di strutturare i dati elaborati dall'algoritmo che permetta di eseguire velocemente le operazioni utilizzate nell'algoritmo (ad esempio: l'algoritmo di ordinamento basato sulla iterata ricerca del minimo è reso più veloce utilizzando uno *heap*).

Di solito, si passa per le fasi (2)-(4) nell'ordine citato, ma questo non sempre è vero. In alcuni casi, se ne possono considerare solo alcune e in un ordine qualsiasi, mentre in altri casi possono essere riconsiderate più di una volta per raffinare e rendere più efficienti diverse versioni successive dell'algoritmo.

Nel seguito di questo capitolo sono esemplificate le fasi di caratterizzazione matematica della soluzione e di utilizzo di strutture di dati nella progettazione di algoritmi efficienti. Per far questo, si considera un noto problema di ottimizzazione su grafi orientati pesati, quello dei cammini minimi, e si mostra come la caratterizzazione della soluzione permetta di definire un algoritmo "prototipo", a partire dal quale si possono derivare diversi algoritmi, più o meno veloci, a seconda della struttura di dati impiegata. Le tecniche di progettazione, invece, saranno l'argomento dei prossimi capitoli.

1.1 Cammini minimi

Dato un grafo orientato $G = (V, E)$, assumiamo che ogni arco $(u, v) \in E$ sia associato ad un *costo* $w(u, v)$ (o *peso*, o *lunghezza*). Dato un cammino $c = v_1, \ldots, v_k$ (con $k > 1$), il *costo del cammino* è definito da:

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

Si consideri il seguente problema di ottimizzazione su grafi:

CAMMINI MINIMI (SHORTEST PATHS). *Dato un grafo orientato $G = (V, E)$, una funzione di costo $w : E \to \mathbf{R}$ e un nodo r, trovare un cammino da r ad u, per ogni nodo $u \in V$, il cui costo sia minimo, ovvero più piccolo o uguale del costo di qualunque altro cammino da r a u.*

Esempio (Cammini minimi) Un proprietario di TIR è libero di trasportare container da una città ad un'altra. Un viaggio dalla città u alla città v porta un profitto di p_{uv} euro se c'è un container da trasportare da u a v, ma provoca una perdita di p_{uv} euro se il TIR viaggia scarico. Assumendo profitti $p_{uv} > 0$ e perdite $p_{uv} < 0$, il percorso più vantaggioso tra due città corrisponde ad un cammino minimo, in cui le lunghezze degli archi sono $w(u, v) = -p_{uv}$. ■

Una soluzione ammissibile di questo problema è data da un insieme di $n - 1$ cammini, ognuno dei quali parte dal medesimo nodo r, ma giunge ad uno e uno solo dei rimanenti nodi. In generale, non è detto che un cammino da r ad un qualsiasi altro nodo u esista, poiché u potrebbe non essere raggiungibile da r. Perché esista almeno una soluzione ammissibile, pertanto, occorre che ogni nodo sia raggiungibile da r con un cammino.

Il verificarsi di questa condizione, da sola, non garantisce l'esistenza di una soluzione ottima. Infatti, poiché sono ammesse lunghezze negative, potrebbe esistere nel grafo un ciclo per il quale la somma delle lunghezze degli archi nel ciclo dia un valore negativo. In tal caso, la lunghezza

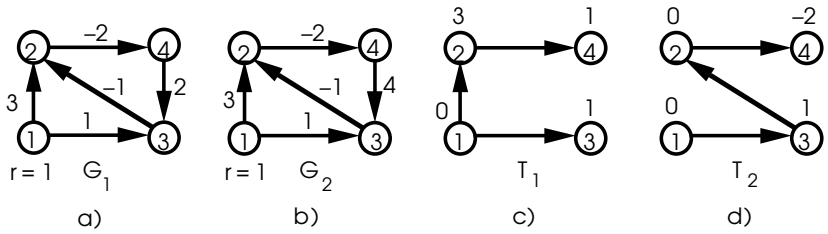


Figura 1.1: a) Un grafo G_1 con un ciclo negativo; b) un grafo G_2 privo di cicli negativi; c) una soluzione ammissibile per G_2; d) una soluzione ottima per G_2.

minima di un qualsiasi cammino che includa un nodo di tale ciclo non sarebbe inferiormente limitata. Occorre quindi che non esistano cicli di lunghezza negativa.

Si osservi che due cammini distinti possono avere un tratto iniziale comune, da r fino ad un certo nodo s, e poi divergere verso le rispettive destinazioni u e v, ma viceversa non possono convergere in un nodo comune s dopo aver percorso due tratti iniziali distinti, perché in tal caso ci sarebbero due cammini distinti da r ad s, mentre il problema ne richiede uno solo. Pertanto, una soluzione ammissibile altro non è che un albero di copertura, radicato in r, che include un cammino da r ad ogni altro nodo.

Esempio (Albero di cammini) La Figura 1.1 mostra un grafo G_1 che contiene un ciclo negativo ed un grafo G_2 privo di cicli negativi. Se $r = 1$, gli alberi T_1 e T_2 sono, rispettivamente, una soluzione ammissibile ed una ottima per G_2. ■

Sia T una soluzione ammissibile. Ogni nodo u del grafo è caratterizzato da un valore reale d_u, che indica la *distanza* di u da r in T, e che è uguale al costo del cammino fra r ed u in T. Quali caratteristiche devono avere le distanze affinché T sia una soluzione ottima?

(1) Sia T una soluzione ottima. Consideriamo un generico arco $(u,v) \in E$ e sia $w(u,v)$ la sua lunghezza. Ovviamente, se $(u,v) \in T$, allora $d_v = d_u + w(u,v)$. Se invece $(u,v) \notin T$, allora, poiché T è ottimo, deve risultare $d_u + w(u,v) \geq d_v$, poiché altrimenti esisterebbe nel grafo G un cammino tra r e v più corto di quello in T, e precisamente il cammino tra r ed u in T seguito dell'arco (u,v).

(2) Viceversa, sia $d_u + w(u,v) = d_v$ per ogni $(u,v) \in T$, e $d_u + w(u,v) \geq d_v$, per ogni arco $(u,v) \notin T$. Se, per assurdo, il cammino da r a u in T non fosse ottimo, allora esisterebbe un altro cammino in G da r a u tale che la distanza d'_u di u da r sarebbe minore di d_u. Sia d'_v la distanza da r ad un generico nodo v che appare in tale cammino. Poiché $d'_r = d_r = 0$, ma $d'_u < d_u$, esiste un arco (h,k) in questo cammino per cui $d'_h \geq d_h$ e $d'_k < d_k$. Per costruzione $d'_h + w(h,k) = d'_k$, mentre per ipotesi $d_h + w(h,k) \geq d_k$. Combinando queste due relazioni, si ottiene: $d'_k = d'_h + w(h,k) \geq d_h + w(h,k) \geq d_k$, e quindi $d'_k \geq d_k$, che contraddice l'ipotesi.

Si è pertanto dimostrato il seguente risultato, dovuto a Bellman (1958):

Teorema 1.1 (Teorema di Bellman) Una soluzione ammissibile T è ottima se e solo se valgono le seguenti condizioni: $d_v = d_u + w(u,v)$ per ogni arco $(u,v) \in T$, e $d_u + w(u,v) \geq d_v$ per ogni arco $(u,v) \in E$.

1.2 Algoritmo prototipo

Questa caratterizzazione della soluzione permette di formulare un algoritmo che si basa sulla verifica, arco per arco, delle condizioni di Bellman, con conseguente sostituzione degli archi che violano tali condizioni:

```
prototipoCamminiMinimi(GRAPH G, NODE r)
% inizializza T ad una foresta di copertura composta da nodi isolati
% inizializza d con una sovrastima della distanza (0 per r, +∞ altrimenti)
while ∃(u,v) : d_u + w(u,v) < d_v do
    d_v = d_u + w(u,v)
    % Sostituisci il padre di v in T con u
```

La foresta di copertura iniziale viene sostituita passo dopo passo dall'albero dei cammini minimi; se al termine dell'esecuzione qualche nodo mantiene ancora la distanza $+\infty$, esso non è raggiungibile dalla radice r.

Il passo cruciale dell'algoritmo consiste nel selezionare un arco (u,v) che violi le condizioni di Bellman. Ciò può essere effettuato mantenendo una struttura di dati S che permetta di aggiungere e togliere nodi tramite due operatori generici aggiungi() e estrai(). Inizialmente viene inserito solo il nodo r. Selezionando un nodo u da S è possibile verificare le condizioni di Bellman su tutti gli archi (u,v) tali che v appartiene all'insieme di adiacenza u.adj(). Se viene modificata la distanza d_v, il nodo v viene inserito nuovamente in S, perché è possibile che v possa ora essere un nodo intermedio più conveniente per raggiungere altri nodi.

Il codice di camminiMinimi() definisce meglio il prototipo di procedura appena abbozzato. Sono utilizzati tre vettori, contenenti una posizione per ognuno dei nodi del grafo. Il vettore d contiene le distanze di tutti i nodi dall'origine r; T è un vettore dei padri ricevuto in input e che viene modificato in modo che al termine dell'esecuzione T contenga l'albero dei cammini minimi; infine, b è un vettore booleano tale per cui $b[u] =$ **true** se e solo se u è contenuto nella struttura di dati S. I vettori delle distanze e dei padri sono inizializzati come descritto sopra. La procedura ha accesso ad una funzione di costo w, lasciando alla realizzazione effettiva l'onere di definire dove debbano essere memorizzati tali costi (e.g., nella matrice di adiacenza del grafo oppure nelle celle che formano la lista di adiacenza di un nodo).

Assumendo che la struttura prima o poi restituisca tutto quello che viene inserito, e in assenza di cicli negativi, tale procedura è corretta, ma non molto efficiente. La sua complessità dipende infatti dal numero massimo di inserzioni del medesimo nodo in S e dal numero massimo di miglioramenti delle distanze. In particolare, quest'ultima quantità può non essere polinomialmente correlata alla dimensione n del problema. Nel caso pessimo, le lunghezze $w(u,v)$ possono avere valori che crescono come $\Omega(2^n)$, mentre le distanze possono essere migliorate solo di una unità alla volta, provocando così un tempo complessivo di esecuzione superpolinomiale in n.

Per ottenere una procedura efficiente, non basta utilizzare una generica "struttura di dati" con un'altrettanto generica operazione estrai(), ma occorre specificarla ulteriormente! Scelte ragionevoli per la struttura S possono sembrare la coda con priorità, la coda e la pila. Infatti, tutte e tre adottano criteri diversi ma sistematici di "lettura", al contrario della troppo vaga operazione estrai(). Nell'algoritmo, sono state evidenziate quattro righe (numerate da (1) a (4)) che devono essere specializzate a seconda della struttura di dati utilizzata. Nel seguito vedremo diverse implementazioni, e per ognuna di esse specificheremo come sostituire le quattro righe suddette.

1.3 Algoritmo di Dijkstra

Se la struttura S è una coda con priorità, realizzata con una lista o un vettore non ordinati, si ottiene un algoritmo noto fin dal 1959 e attribuito a Dijkstra. In tal caso, l'operazione estrai() si riduce a deleteMin() [cfr. Capitolo 10.1]. Gli elementi della coda con priorità sono nodi del grafo e le loro priorità corrispondono alla distanza da r. Ad ogni iterazione è estratto da S il nodo u avente distanza $d[u]$ minima. Il codice da sostituire è il seguente:

(1) PRIORITYQUEUE $S =$ PriorityQueue(); S.insert(r,0)

```
camminiMinimi(GRAPH G, NODE r, int[] T)
    int[] d = new int[1...G.n]                    % d[u] è la distanza da r a u
    boolean[] b = new boolean[1...G.n]            % b[u] è true se u è contenuto in S
    foreach u ∈ G.V() − {r} do
        T[u] = nil
        d[u] = +∞
        b[u] = false
    T[r] = nil
    d[r] = 0
    b[r] = true
(1) STRUTTURADATI S = StrutturaDati(); S.aggiungi(r)
    while not S.isEmpty() do
(2)     int u = S.estrai()
        b[u] = false
        foreach v ∈ G.adj(u) do
            if d[u] + w(u,v) < d[v] then
                if not b[v] then
(3)                 S.aggiungi(v)
                    b[v] = true
                else
(4)                 % Azione da intraprendere nel caso v sia già presente in S
                T[v] = u
                d[v] = d[u] + w(u,v)
```

(2) $u = S.\mathsf{deleteMin}()$
(3) $S.\mathsf{insert}(v, d[u] + w(u,v))$
(4) $S.\mathsf{decrease}(v, d[u] + w(u,v))$

Benché la complessità dell'algoritmo resti superpolinomiale qualora il grafo contenga archi di lunghezza negativa, tale complessità si riduce drasticamente se tutte le lunghezze sono non negative. Sotto questa ipotesi, infatti, ogni nodo del grafo viene estratto da S una e una sola volta e, al momento dell'estrazione, la sua distanza da r è minima. Per convincersi di ciò, si proceda per induzione sul numero di nodi estratti. Non essendoci lunghezze negative, l'asserto è chiaramente vero per il primo nodo estratto, cioè r, per il quale $d[r] = 0$. Quando viene estratto il k-esimo nodo, per esempio u, la sua distanza $d[u]$ dipende soltanto dai $k-1$ nodi già estratti, ognuno dei quali ha distanza minima per ipotesi d'induzione, ma non può dipendere dai nodi che si trovano ancora in S, che hanno distanza maggiore o uguale a $d[u]$. Non essendoci lunghezze negative, ed essendo $d[u]$ la distanza più piccola tra tutte quelle dei nodi in S, $d[u]$ è minima ed il nodo u non sarà più reinserito in S.

Esempio (Algoritmo di Dijkstra) La Figura 1.2 mostra un grafo G con lunghezze non negative degli archi e l'esecuzione dell'algoritmo di Dijkstra su tale grafo. L'ordine di estrazione da S è: 1, 2, 3, 5, 4, 6. ■

Poiché ogni nodo è estratto da S una e una sola volta, il ciclo **while** è eseguito esattamente n volte. All'interno del ciclo, l'operazione più costosa è deleteMin() che, utilizzando una struttura non ordinata, richiede tempo $O(n)$, mentre la insert() è $O(1)$ utilizzando una lista non ordinata e già sapendo (grazie al controllo sul vettore booleano) che l'elemento non è presente. Pertanto l'algoritmo di Dijkstra ha complessità $O(n^2)$.

1.4 Algoritmo di Johnson

Qualora la struttura S sia una coda con priorità realizzata con uno *heap* binario, si ottiene un algoritmo proposto da D.B. Johnson (1977). In molti testi questa versione è associata a Dijkstra, ma si tenga conto che gli *heap* binari sono stati introdotti solo nel 1964 a opera di Williams, quindi ben cinque anni dopo l'algoritmo di Dijkstra. Ad ogni buon conto, le linee da sostituire sono le stesse che nella sezione precedente.

Qualora le lunghezze degli archi siano non negative, ogni nodo è estratto dallo *heap* S una e una sola volta e il ciclo **while** è eseguito esattamente n volte, come nell'algoritmo di Dijkstra. All'interno del ciclo, però, le operazioni deleteMin() e decrease() richiedono tempo $O(\log n)$. Poiché, nel caso pessimo, decrease() è richiamata ad ogni iterazione su tutti i nodi di $G.\mathsf{adj}(u)$, la complessità totale è

$$O\left(n\log n + \left(\sum_{u\in G.\mathsf{V}()} |G.\mathsf{adj}(u)|\right)\log n\right).$$

Pertanto l'algoritmo di Johnson ha complessità $O(m\log n)$.

Si noti che per grafi sparsi, nei quali il numero m di archi è $O(n)$, la complessità diviene $O(n\log n)$ e l'algoritmo di Johnson risulta più veloce di quello di Dijkstra. Per grafi densi, dove m è $\Omega(n^2)$, la complessità risulta $O(n^2\log n)$ e l'algoritmo di Johnson è più lento di quello di Dijkstra.

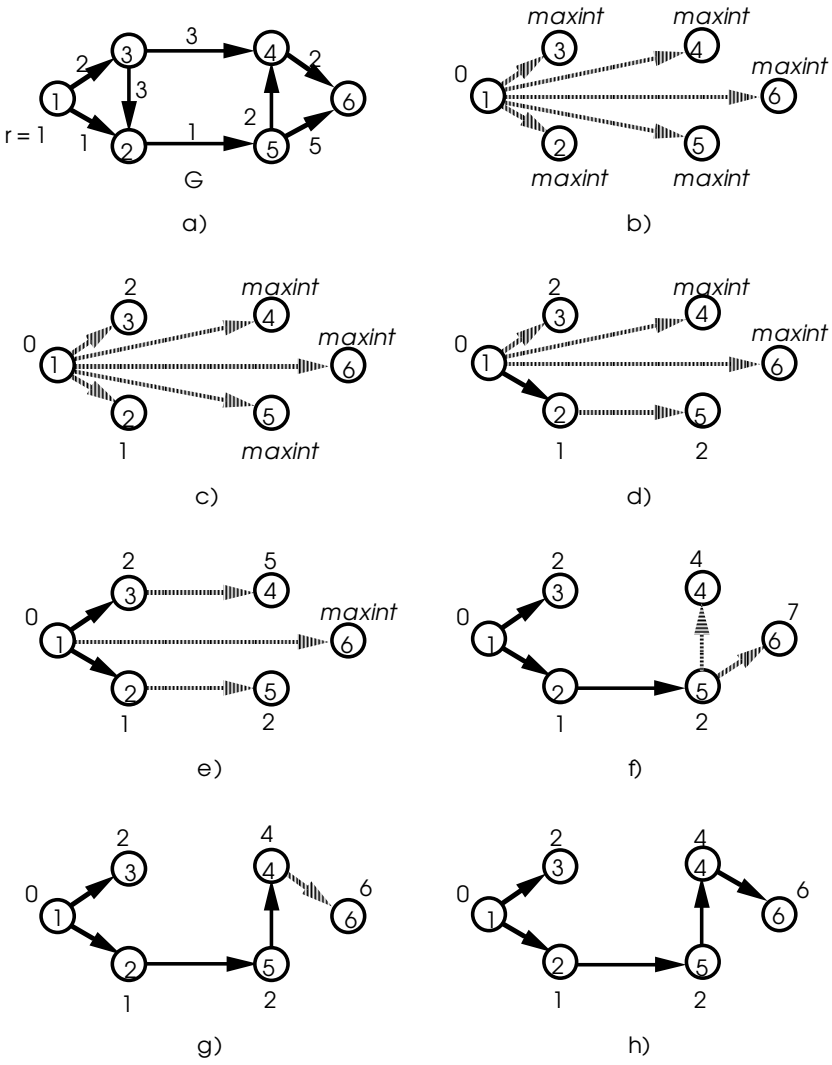


Figura 1.2: Algoritmo di Dijkstra. a) Un grafo G con lunghezze non negative; b) inizializzazione ad un albero fittizio per $r = 1$; c) - h) aggiornamento dell'albero e delle distanze, estraendo i nodi nell'ordine: 1, 2, 3, 5, 4, 6. **TODO**: "maxint" $\rightarrow +\infty$

1.5 Algoritmo di Fredman-Tarjan

La scarsa efficienza dell'algoritmo di Johnson nel caso di grafi densi deriva dal fatto che l'operazione decrease(), che viene eseguita fino a $O(m)$ volte, ha costo $O(\log n)$. Una possibile soluzione a questo problema prevede di adottare una struttura di dati esplicitamente studiata per migliorare le prestazioni del nostro algoritmo generico, gli heap di Fibonacci. Questa struttura di dati, non descritta in questo libro in quanto particolarmente complicata, conserva un costo di $O(\log n)$ per l'operazione deleteMin(), ma richiede un costo ammortizzato $O(1)$ per le operazioni insert() e decrease(). In questo modo, il costo del nostro algoritmo diventa $O(m+n\log n)$. Tuttavia, questo algoritmo ha più interesse teorico che pratico, in quanto gli heap di Fibonacci non sono implementati efficientemente.

1.6 Algoritmo di Bellman-Ford-Moore

Se la struttura S è una coda si ottiene un algoritmo, noto sin dalla fine degli anni Cinquanta e scoperto indipendentemente da molti ricercatori, che ha complessità polinomiale anche se ci sono archi di lunghezza negativa! L'algoritmo di Bellman-Ford-Moore si ottiene dal prototipo sostituendo le seguenti righe di codice:

(1) QUEUE $S = \mathsf{Queue}(); S.\mathsf{enqueue}(r)$
(2) $u = S.\mathsf{dequeue}()$
(3) $S.\mathsf{enqueue}(v)$
(4) Sezione non necessaria

La struttura dell'algoritmo è in pratica quella di una visita BFS in cui la marcatura di un nodo consiste nel diminuirne la distanza ed il medesimo nodo può essere visitato, cioè estratto dalla coda S, più di una volta, ma non più di $n-1$ volte!

La dimostrazione di questa proprietà è un po' macchinosa e si rifà alla nozione di "passata", che è definita induttivamente come segue:

(1) per $k = 0$, la zeresima passata consiste nell'estrazione del nodo r dalla coda S;
(2) per $k > 0$, la k-esima passata consiste nell'estrazione di tutti i nodi presenti in S al termine della passata $(k-1)$-esima.

Si indichino con $d_{v,k}$ il valore della distanza $d[v]$ all'inizio della passata k-esima, con p_v l'ultima passata in cui il nodo v si trova in S, con S_k l'insieme dei nodi in S nella passata k-esima, con $I(v)$ l'insieme $\{u : (u,v) \in E\}$, e con T^* l'albero ottimo dei cammini minimi, con distanze d^*_u per ogni nodo u.

La proprietà cruciale dell'algoritmo è che, per ogni arco (u,v) che sta nella soluzione ottima T^*, p_v è uguale a p_u oppure a p_u+1. Innanzitutto, è facile verificare che, per ogni v e k, si ha:

$$d_{v,k} = \min\{d_{v,k-1} + w(u,v) : u \in I(v) \cap S_{k-1}\},$$

poiché S è una coda. Da questa proprietà discende che, se $v \in S_{p_v}$, allora deve esistere un nodo $z \in S_{p_v-1}$ che ha provocato l'inserimento di v in S per l'ultima volta. Ma allora la distanza ottima d^*_v è stata determinata esplorando

– l'arco (z,v), e quindi $p_v = p_z + 1$, oppure
– un arco (u,v) con $u \in S_{p_v}$, e allora $p_v = p_z$.

In entrambi i casi, un eventuale reinserimento di u o z causerebbe un'ulteriore diminuzione di p_v e un reinserimento di v in S, il che è impossibile. Poiché il cammino semplice più lungo contiene al più $n-1$ archi, allora $p_v < n$ per ogni nodo v, ed il massimo numero di passate è n. Siccome ad ogni passata la coda S contiene al più tutti i nodi e ad ogni passata si esaminano al più tutti gli archi, l'algoritmo di Bellman-Ford-Moore ha complessità $O(n(m+n)) = O(nm)$, qualsiasi sia il segno delle lunghezze degli archi.

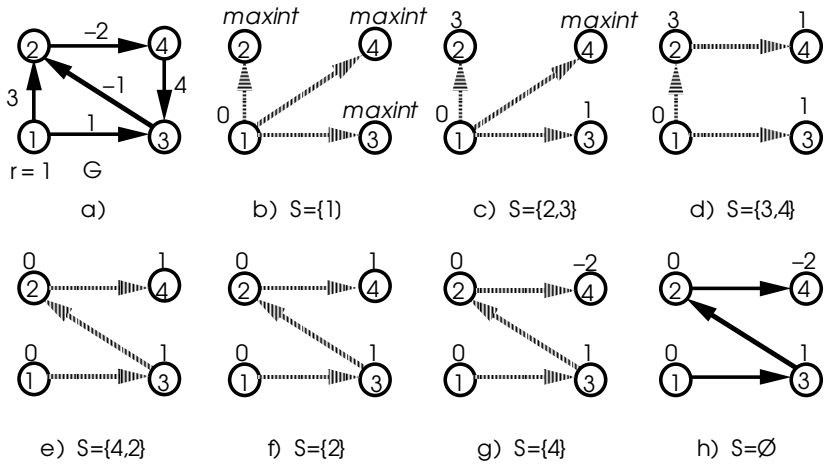


Figura 1.3: Algoritmo di Bellman-Ford-Moore. a) Un grafo G ($r = 1$); b)-g) estrazione dalla coda dei nodi nell'ordine: 1, 2, 3, 4, 2, 4; h) la soluzione ottima. **TODO**: "maxint" $\to +\infty$

Esempio (Algoritmo di Bellman-Ford-Moore) La Figura 1.3 mostra l'esecuzione dell'algoritmo di Bellman-Ford-Moore su un grafo G per $r = 1$. L'ordine di estrazione dei nodi da S è: 1 (passata 0); 2, 3 (passata 1); 4, 2 (passata 2) e 4 (passata 3). ■

1.7 Utilizzo di una pila

Benché venga spontaneo pensare che un algoritmo che impieghi una pila anziché una coda debba avere la stessa complessità dell'algoritmo di Bellman-Ford-Moore, l'utilizzo di una pila per la struttura S rende la procedura camminiMinimi() di complessità superpolinomiale.

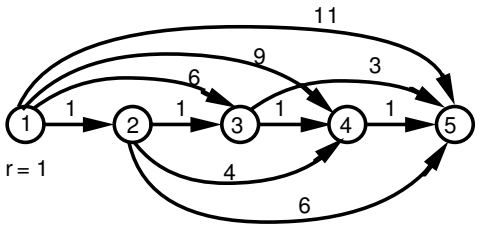
Si consideri il grafo $G_n = (V_n, E_n)$, con $n \geq 2$, così definito: $V_n = \{1, 2, \ldots, n\}$, $E_n = \{(u,v)$: $u < v\}$, con lunghezze $w(u,v) = 2^{n-u-1} - 2^{n-v} + v - u$. Si noti che G_n è un grafo *aciclico* e che le lunghezze sono non negative. Nonostante questa apparente semplicità della struttura di G_n, è possibile dimostrare che, utilizzando una pila, l'estrazione del nodo u da S provoca ogni volta l'inserimento di tutti i nodi k con $k > u$. Pertanto ogni nodo può essere estratto da S per $O(2^n)$ volte. Il fatto più curioso è che l'algoritmo di Bellman-Ford-Moore richiede soltanto un tempo $O(m)$ qualora sia eseguito su G_n!

Esempio (Utilizzo di una pila) La Figura 1.4 mostra il grafo G_n per $n = 5$, le configurazioni successive della pila S (sotto forma di vettore booleano), i nodi estratti da S e i valori delle distanze ad ogni iterazione dell'algoritmo camminiMinimi() eseguito su G_5 con $r = 1$. Si noti che le configurazioni della pila corrispondono a tutti i numeri in base 2 compresi tra $16 = 2^{n-1}$ e 0, e che il nodo $n = 5$ viene estratto da S per $2^{n-2} = 8$ volte. ■

1.8 Algoritmo di Pape-D'Esopo

In pratica, l'algoritmo che risulta più veloce nei problemi reali impiega per S una struttura detta *dequeue* (per *double ended queue*), cioè una sequenza modificabile ad entrambi gli estremi che combina le proprietà di una pila e di una coda [cfr. Esercizio 4.9].

L'algoritmo che usa la dequeue, proposto da Pape (1974), permette di estrarre solo dalla testa di S, ma ammette l'inserzione sia in testa che in fondo. Per la precisione, quando un nodo è inserito



G_5

pila S 1 2 3 4 5	nodo estratto	distanze d_1 d_2 d_3 d_4 d_5
1 0 0 0 0	1	0 ∞ ∞ ∞ ∞
0 1 1 1 1	5	0 1 6 9 11
0 1 1 1 0	4	0 1 6 9 11
0 1 1 0 1	5	0 1 6 9 10
0 1 1 0 0	3	0 1 6 9 10
0 1 0 1 1	5	0 1 6 7 9
0 1 0 1 0	4	0 1 6 7 9
0 1 0 0 1	5	0 1 6 7 8
0 1 0 0 0	2	0 1 6 7 8
0 0 1 1 1	5	0 1 2 5 7
0 0 1 1 0	4	0 1 2 5 7
0 0 1 0 1	5	0 1 2 5 6
0 0 1 0 0	3	0 1 2 5 6
0 0 0 1 1	5	0 1 2 3 5
0 0 0 1 0	4	0 1 2 3 5
0 0 0 0 1	5	0 1 2 3 4
0 0 0 0 0		0 1 2 3 4

Figura 1.4: Esecuzione della procedura camminiMinimi() sul grafo G_5 usando una pila.

in S per la prima volta, allora viene aggiunto in fondo ad S, mentre quando è inserito le volte successive, allora viene aggiunto in testa ad S. In altri termini, quando un nodo v va inserito in S, se $d[v] = +\infty$ allora S è trattata come una coda e v diviene l'ultimo elemento della sequenza, mentre se $d[v] < +\infty$ allora S è trattata come una pila e v diventa il primo elemento della sequenza. La giustificazione di questa modalità di inserimento è la seguente. Ogni volta che la distanza $d[v]$ di un nodo v è decrementata, allora conviene tentare di diminuire anche le distanze dei nodi che seguono v nei cammini che appaiono nell'albero T che rappresenta la soluzione parziale corrente. Questo è lo scopo dell'inserimento in testa quando $d[v] < +\infty$. D'altra parte, per evitare casi patologici con effetto "ping-pong" come si è visto nell'Esempio 1.7, non è conveniente inserire in testa quando la distanza corrisponde ad un cammino nell'albero "fittizio" T dovuto all'inizializzazione. Questo è lo scopo dell'inserimento in fondo quando $d[v] = +\infty$.
L'algoritmo di Pape-D'Esopo si ottiene dal prototipo sostituendo le seguenti righe di codice:

(1) DEQUEUE $S =$ DeQueue(); S.insertHead(r)
(2) $u = S$.removeHead()
(3) **if** $d[v] = +\infty$ **then** S.insertTail(v) **else** S.insertHead(v)
(4) Sezione non necessaria

Esempio (Algoritmo di Pape-D'Esopo) Eseguendo l'algoritmo di Pape-D'Esopo sul grafo G della Figura 1.3, è facile verificare che l'ordine di estrazione dei nodi da S è: 1, 2, 3, 2, 4. Si noti che rispetto all'algoritmo di Bellman-Ford-Moore il nodo 4 è estratto una sola volta e che il numero complessivo di estrazioni è 5 anziché 6. ■

Benché nel caso pessimo l'algoritmo di Pape-D'Esopo richieda tempo superpolinomiale come quello che usa una pila, è stato verificato sperimentalmente che nella pratica esso risulta il più veloce di tutti specie per grafi che rappresentano vere reti di comunicazione stradale. Questi grafi hanno la proprietà di essere sparsi, cioè con m che cresce come $O(n)$, e planari, cioè possono essere disegnati sul piano in modo che le linee corrispondenti a due archi distinti non si sovrappongano mai (dato un grafo G, verificarne la planarità richiede tempo $O(n+m)$ con un algoritmo complicato ideato da Hopcroft e Tarjan (1974); inoltre, se G è planare, è possibile fornirne un disegno planare in tempo polinomiale).

Esempio (Planarità) I grafi illustrati nelle Figure 1.2(a) e 1.3(a) sono planari. Il grafo G_5 della Figura 1.4 invece non lo è, in quanto non è possibile eliminare la sovrapposizione tra gli archi $(1,4)$ e $(3,5)$ senza introdurre ulteriori sovrapposizioni tra altre coppie di archi. ■

1.9 Reality check

I protocolli OSPF (*Open Shortest Path First*) e IS-IS (*Intermediate System to Intermediate System*) sono protocolli di routing utilizzati nella rete Internet. Sono protocolli *link-state*: ogni router costruisce una mappa completa della connettività della rete, un grafo in cui esiste un arco fra due nodi (router) solo se sono connessi direttamente. Ogni nodo calcola in maniera indipendente il cammino minimo per tutte le possibili destinazioni della rete, utilizzando l'algoritmo di Dijkstra basato su heap (proposto da Johnson).

Questi protocolli sono denominati anche *interior gateway protocol*; in altre parole, vengono utilizzati all'interno di *singoli autonomous systems*, ovvero porzioni della rete sotto il controllo di un'unica entità (per esempio un Internet Service Provider). Protocolli diversi vengono utilizzati per instradare il traffico fra autonomous systems distinti.

1.10 Esercizi

Esercizio 1.1 (Algoritmo di Dijkstra)**.** Cosa succede se l'algoritmo di Dijkstra è eseguito su un grafo in cui le lunghezze degli archi possono essere negative?

Esercizio 1.2 (Algoritmo di Bellman-Ford-Moore)**.** Si utilizzi l'algoritmo di Bellman-Ford-Moore per verificare in tempo $O(mn)$ se un grafo possiede cicli negativi.

Esercizio 1.3. 11.3 [Affidabilità] Dato un grafo orientato $G = (V,E)$ nel quale ogni arco $(u,v) \in E$ è associato ad un valore reale $r(u,v)$ preso dall'intervallo $[0,1]$, che rappresenta l'affidabilità del canale di comunicazione dal vertice u al vertice v, ovvero la probabilità che tale canale trasmetta un messaggio. Supponiamo che queste probabilità siano indipendenti.

Progettare un algoritmo efficiente per trovare il cammino più affidabile fra due vertici dati, dove l'affidabilità di un cammino è data dal prodotto delle affidabilità degli archi esistenti.

Esercizio 1.4 (Insieme indipendente di intervalli)**.** Dato un insieme di n intervalli della retta reale, si vuole trovarne un insieme indipendente massimo, cioè un sottoinsieme X di intervalli, avente massima cardinalità, tale che ogni coppia di intervalli in X abbia intersezione vuota. Si fornisca un algoritmo polinomiale, formulando il problema come problema di cammini minimi su un opportuno grafo.

Esercizio 1.5 (Insieme dominante di intervalli)**.** Dato un insieme di n intervalli della retta reale, si vuole trovarne un *insieme dominante*, cioè un sottoinsieme X di intervalli, avente minima cardinalità, tale che ogni intervallo che non sta in X abbia intersezione non vuota con almeno un intervallo che sta in X. Si fornisca un algoritmo polinomiale, formulando il problema come problema di cammini minimi su un opportuno grafo.

Esercizio 1.6 (Insieme dominante totale pesato)**.** Si riconsideri il problema dell'Esercizio 1.5 qualora ad ogni intervallo sia associato un peso intero non negativo e si richieda di trovare un insieme dominante totale di peso minimo, dove ogni intervallo (anche se sta in X) deve avere intersezione non vuota con almeno un intervallo che sta in X.

Esercizio 1.7 (Numero di cammini minimi)**.** Dati un grafo orientato con lunghezze positive sugli archi e due nodi s ed t, possono esserci molti cammini minimi distinti fra essi. Si modifichi l'algoritmo di Dijkstra per calcolare il numero dei cammini minimi distinti da s a t.

Esercizio 1.8 (Cammino con arco più corto)**.** Dati un grafo orientato con lunghezze positive sugli archi e due nodi r ed s, si vuole individuare il cammino da r ad s che contiene l'arco più corto, cioè quello con lunghezza minima tra tutti gli archi che possono apparire nel cammino. Si fornisca un algoritmo per determinare il cammino da r ad s contenente l'arco più corto e se ne analizzi la complessità.

Esercizio 1.9 (Costi sui nodi)**.** Dato un grafo orientato $G = (V,E)$ e una funzione di pesi $w : V \to \mathbf{R}$ che associa pesi ai nodi, invece che agli archi, il peso di un cammino è dato dalla somma dei pesi dei nodi inclusi nel cammino. Progettare un algoritmo per risolvere il problema dei cammini minimi da un nodo r a tutti gli altri nodi su un grafo pesato sui nodi.

Esercizio 1.10 (Pesi quadrati)**.** Dato un grafo orientato $G = (V,E)$ e una funzione di pesi $w : E \to \mathbf{R}$ con pesi positivi, sia p un cammino di costo minimo fra una coppia di nodi s e t. Si consideri ora una funzione di pesi w' tale che $w'(u,v) = w(u,v)^2$ per ogni $(u,v) \in E$. Il cammino p è minimo anche nel grafo G pesato da w'? Se sì, fornire una dimostrazione; se no, fornire un controesempio.

1.11 Soluzioni

Soluzione Esercizio 1.4

Assumiamo senza perdere in generalità che gli n intervalli $[a_1,b_1]$, ..., $[a_n,b_n]$ abbiano tutti i $2n$ estremi distinti e si aggiungano due intervalli fittizi $[a_0,b_0]$ e $[a_{n+1},b_{n+1}]$, tali che $b_0 < a_u$, e $b_u < a_{n+1}$, per $1 \leq u \leq n$. Si costruisce un grafo orientato aciclico G con $n+2$ nodi corrispondenti agli $n+2$ intervalli tale che esiste un arco (u,v) se e solo se $b_u < a_v$. La lunghezza $w(u,v)$ dell'arco (u,v) è -1 se $u \neq 0$, mentre è 0 se $u = 0$. È facile verificare che un cammino minimo tra il nodo 0 ed il nodo $n+1$ corrisponde ad un insieme indipendente massimo. La costruzione del grafo richiede tempo $O(n^2)$ e la risoluzione del problema di cammini minimi richiede lo stesso tempo poiché il grafo è aciclico (DAG).

Soluzione Esercizio 1.5

Facciamo le stesse assunzioni dell'Esercizio 1.4, ma cambiamo la costruzione del DAG G. Per ciascun intervallo $[a_u, b_u]$, con $u = 0,1,\ldots,n$, definiamo

$$P_u = \{k : a_u < a_k < b_u < b_k\}$$
$$Q_u = \{k : a_k > b_u \text{ e } \nexists\, h \text{ con } b_u < a_h < b_h < a_k\}$$

e introduciamo un arco (u,v) se e solo $v \in P_u \cup Q_u$. La lunghezza $w(u,v)$ dell'arco è 1 se $u > 0$, mentre è 0 se $u = 0$. Un cammino minimo tra il nodo 0 ed il nodo $n+1$ corrisponde ad un insieme dominante minimo. Si noti che la definizione dell'insieme Q_u permette di evitare di "saltare" un eventuale intervallo $[a_h, b_h]$ che altrimenti resterebbe non dominato. Dopo aver preventivamente ordinato gli intervalli per estremi iniziali crescenti, ciascun P_u e ciascun Q_u si può costruire in tempo $O(n)$. Quindi la costruzione di G richiede tempo $O(n^2)$ così come trovare il cammino minimo perché G è aciclico.

Soluzione Esercizio 1.6

Con le medesime ipotesi dell'Esercizio 1.5, definiamo gli insiemi P_u e Q_u nello stesso modo, ed inoltre definiamo

$$C_u = \{k : a_u < a_k < b_k < b_u\}.$$

Il grafo G ora contiene $2n+2$ nodi, poiché ad ogni intervallo originario $[a_u, b_u]$ con $1 \leq i \leq n$ stavolta corrispondono due nodi, indicati con u_{IN} e u_{OUT} (per praticità denotiamo anche il nodo 0 sia con 0_{IN} sia con 0_{OUT}, e lo stesso facciamo col nodo $n+1$). Indichiamo con w_u il peso di un intervallo, dove $w_0 = w_{n+1} = 0$. Gli archi sono definiti nel modo seguente:

(1) Esiste l'arco $(u_{\text{IN}}, v_{\text{IN}})$ con lunghezza w_u se e solo se $v \in P_u$;
(2) Esiste l'arco $(u_{\text{IN}}, v_{\text{OUT}})$ con lunghezza w_u se e solo se $v \in Q_u$;
(3) Esiste l'arco $(u_{\text{OUT}}, v_{\text{IN}})$ con lunghezza w_u se e solo se $v \in P_u$;
(4) Esiste l'arco $(u_{\text{OUT}}, v_{\text{OUT}})$ con lunghezza $w_u + w_k$ se e solo se $v \in Q_u$, $k \in C_u$, e $w_k = \min\{w_h : h \in C_u\}$.

Al solito, un cammino minimo tra il nodo 0 ed il nodo $n+1$ corrisponde ad un insieme dominante totale di peso minimo ed il problema si risolve in tempo $O(n^2)$. Si noti che lo sdoppiamento dei nodi serve ad evitare di includere nel grafo G due archi consecutivi di tipo Q, che porterebbero ad includere nella soluzione X un intervallo isolato, cioè un intervallo che non si interseca con nessun altro intervallo di X. Per costruzione, in un nodo di tipo IN possono entrare solo archi di tipo P (che corrispondono a coppie di intervalli che si intersecano) e quindi possono uscire da tale nodo sia archi di tipo P sia archi di tipo Q. Invece, in un nodo di tipo OUT possono entrare solo archi di tipo Q (che corrispondono a coppie di intervalli che non si intersecano) e quindi, mentre un arco di tipo P può sempre uscire da tale nodo, un arco di tipo Q può uscirvi solo se viene tenuto in considerazione un intervallo propriamente contenuto (per garantire la dominazione totale) e di peso minimo (per garantire l'ottimalità).

Soluzione Esercizio 1.8
È sufficiente modificare l'algoritmo di Dijkstra nel modo seguente:

La riga: **if** $d[u] + w(u,v) < d[v]$ **then**

diventa **if** $\mathsf{min}(w(u,v), d[u]) < d[v]$ **then**

La riga $d[v] = d[u] + w(u,v)$

diventa $d[v] = \mathsf{min}(w(u,v), d[u])$

Soluzione Esercizio 1.9
Invece di progettare un nuovo algoritmo, utilizziamo uno qualunque degli algoritmi proposti, trasformando la funzione di costo $w : V \to \mathbf{R}$ in una funzione di costo $w' : V \times V \to \mathbf{R}$, nel modo seguente: $w'(u,v) = w(v)$. In questo modo, tutte le volte che si attraversa un arco si paga il peso che si pagherebbe ad attraversare il nodo destinazione di quell'arco. Si noti tuttavia che il peso effettivo di tutti i cammini va aumentato di $w(r)$, in quanto il nodo di partenza non viene attraversato da archi entranti.

Soluzione Esercizio 1.10
Si consideri un grafo in cui vi è un arco (s,t) di peso 2 e un cammino p formato da tre archi disgiunti di peso 1. L'arco (s,t) è un cammino minimo da s a t con questi pesi; ma elevando i pesi al quadrato, il cammino p conserva il peso 3, mentre il cammino (s,t) diventa di peso 4 e non è più minimo. Abbiamo quindi trovato un controesempio.