Progettazione di algoritmi

Discussione dell'esercizio [resto 2]

Dati n tagli di banconote $1 = v_1 < v_2 < \cdots < v_n$ e un valore R, vogliamo calcolare il numero di modi diversi in cui si può dare il resto R. Guidati dalla tabella usata per risolvere il problema *resto* possiamo definire una tabella simile che cioè considera lo stesso tipo di sotto-problemi:

$$N[k, r] = \text{numero di modi di dare il resto } r \text{ con i primi } k \text{ tagli}$$

I casi base sono

$$\forall k = 1, \ldots, n \quad N[k, 0] = 1 \quad \text{e} \quad \forall r = 0, \ldots, R \quad N[1, r] = 1$$

Il calcolo degli altri elementi della tabella è facile:

$$N[k, r] = \begin{cases} N[k-1, r] + N[k, r - v_k] & \text{se } v_k \leqslant r \\ N[k-1, r] & \text{altrimenti} \end{cases}$$

L'algoritmo che calcola la tabella N è immediato da scrivere ed ha complessità proporzionale alla dimensione della tabella, cioè $O(nR)$. Non è necessario mantenersi l'intera tabella ma sono sufficienti le ultime due righe.

Longest Path in DAG

Dato un DAG pesato G e data una coppia di nodi u, v, vogliamo trovare un cammino da u a v di peso massimo (il cammino sarà necessariamente semplice perchè G non ha cicli). Per prima cosa ordiniamo topologicamente i nodi di G e rinumeriamo i nodi secondo tale ordinamento, cioè per ogni arco (i, j) si ha che $i < j$. Siano h e k, rispettivamente, le nuove numerazioni dei nodi u e v. Assumiamo che $h \leq k$ perchè se $h > k$ non esistono cammini da u a v. Un qualsiasi cammino da u a v può passare solamente per i nodi h, h+1,…,h+Δ, dove $\Delta = k - h$. Per ognuno di questi nodi $h+i$ possiamo calcolarci facilmente il massimo peso di un cammino che che va da h a $h + i$ in base ai relativi massimi per i nodi più vicini a h. Per fare ciò introduciamo la seguente tabella: per ogni i = 0,1, …,Δ,

$$M[i] = \begin{cases} \text{peso massimo di un cammino da } h \text{ a } h+i & \text{se esiste un cammino da } h \text{ a } h+i \\ -1 & \text{altrimenti} \end{cases}$$

Chiaramente, $M[0] = 0$ e il calcolo della tabella procede per j crescenti:

$$M[j] = \begin{cases} \max\{M[i] + p(h+i, h+j) \mid M[i] \neq -1 \wedge (h+i, h+j) \in E\} & \text{se esiste almeno un arco } (h+i, h+j) \\ & \quad \text{con } M[j] \neq -1 \\ -1 & \text{altrimenti} \end{cases}$$

dove E è l'insieme degli archi del DAG G. Infatti, un cammino P di peso massimo da un nodo x a un nodo y o consiste di un solo arco (cioè un arco (x, y)) oppure consiste di due o più archi e sia (w, y) l'ultimo arco di P, per cui w è compreso tra x e y nell'ordinamento topologico. La parte restante del cammino P che va da w a y deve necessariamente essere un cammino di peso massimo da w a y e può passare solamente per nodi compresi tra w e y nell'ordinamento topologico. Adesso possiamo scrivere un algoritmo per calcolare il peso massimo di un cammino da u a v in un DAG:

```
LONGESTPATH(G: DAG pesato con n nodi, u,v: nodi)
    Ordina topologicamente G e rinumera i nodi e gli archi secondo tale ordinamento
    Siano h e k gli indici dei nodi u e v, rispettivamente
    IF k < h THEN RETURN -1
    Δ <- k - h
    M: tabella di dimensione Δ + 1
    M[0] <- 0
    FOR j <- 1 TO Δ DO
```

```
        M[j] <- -1
        FOR ogni arco (h+i, h+j) entrante nel nodo h+j DO
            IF M[i] ≠ -1 THEN
                IF M[i] + p(h+i, h+j) > M[j] THEN
                    M[j] <- M[i] + p(h+i, h+j)
    RETURN M[Δ]
```

L'ordinamento topologico può essere determinato, come sappiamo, in tempo O($n + m$) e la rinumerazione dei nodi secondo un tale ordinamento può essere effettuata anch'essa in O($n + m$). Il calcolo della tabella M esamina al più tutti i nodi e gli archi di G e per ognuno di essi impiega tempo O(1), quindi il tempo richiesto da tale calcolo è limitato da O($n + m$). In totale la complessità dell'algoritmo `LONGESTPATH` è O($n + m$), quindi lineare nella dimensione del grafo. Se si vuole trovare anche il cammino di peso massimo, la cosa più semplice è di modificare l'algoritmo `LONGESTPATH` mantenendo una seconda tabella A tale che $A[j]$ registra l'arco ($h+i$, $h+j$), se esiste, che ha determinato il valore di $M[j]$. Dalla tabella A è immediato ricavare il cammino di peso massimo.

Ritornando al problema dello *Zaino*, sappiamo che possiamo risolvere un'istanza I del problema trovando un cammino di peso massimo dal nodo $<n,C>$ al nodo z nel DAG pesato G_I. Con l'algoritmo `LONGESTPATH` possiamo farlo in tempo lineare nella dimensione del grafo. Il grafo G_I ha $nC + 1$ nodi e da ogni nodo escono al più tre archi, quindi il numero totale di archi è O(Cn). Perciò la complessità rimane (almeno asintoticamente) la stessa di quella dell'algoritmo basato direttamente sul calcolo della tabella Z, cioè O(nC). Si osservi che mentre l'algoritmo `LONGESTPATH` risolve in modo efficiente (avendo complessità lineare) il problema del Longest Path in DAG, quando lo usiamo per risolvere il problema dello Zaino l'algoritmo che otteniamo non può essere considerato efficiente. La ragione è che la dimensione del grafo G_I è, in generale, molto più grande di quella dell'istanza I. Si pensi ad esempio al caso in cui $C = 2^n$ e ogni peso o valore è $\leq 2^n$, la dimensione di I è limitata da O(n^2) perché ogni numero dell'istanza è rappresentabile con O(n) bits. Però il grafo avrà dimensione O($n2^n$), quindi esponenziale nella dimensione di I.

Il Longest Path problem oltre ad essere legato al problema dello *Zaino* ha applicazioni nel campo della ricerca operativa, principalmente nel metodo del percorso critico (CPM) per la pianificazione di un insieme di attività, e nella visualizzazione dei grafi.

Pianificazione di attività (CPM)

La realizzazione di un progetto (in edilizia, sviluppo software, industria, ecc.) consiste nell'esecuzione di un insieme di n attività che identificheremo con gli interi 1,2,…,n. Per ogni attività i si conosce il tempo di esecuzione t_i. Inoltre, ci sono delle dipendenze tra le attività. Ogni dipendenza è relativa a una coppia di attività (i, j) e dice che l'esecuzione dell'attività j non può iniziare prima che sia terminata l'esecuzione dell'attività i, più brevemente i deve precedere j. Ovviamente assumiamo che le dipendenze non creino cicli. Se non ci sono dipendenze dirette o indirette tra i e j, le loro esecuzioni possono avvenire in qualsiasi modo: in sovrapposizione, i prima di j o j prima di i. Date queste informazioni vogliamo determinare qual'è il tempo di completamento del progetto e i tempi d'inizio di ogni attività. Il progetto è completato quando tutte le attività sono state eseguite rispettando le dipendenze. Ad esempio, se non ci fossero dipendenze il tempo di completamento sarebbe uguale al massimo tempo di esecuzione delle attività, cioè $\max\{t_i \mid i = 1,\ldots,n\}$.

Per risolvere questo problema, che chiameremo *CPM*, dobbiamo determinare il tempo d'inizio di ogni attività in modo da minimizzare il tempo di completamento del progetto. Indichiamo con x_i il tempo d'inizio (incognito) dell'attività i. Una dipendenza (i, j) equivale all'imposizione del vincolo $x_j \geq x_i + t_i$. Quello che vogliamo fare è calcolare il minimo valore di ogni x_i che rispetta tutti i vincoli relativi alle dipendenze. I vincoli possono essere rappresentati tramite un opportuno grafo G in cui ci sono n nodi 1,2,…,n che corrispondono alle attività e un nodo 0. Per ogni dipendenza (i, j) c'è un arco dal nodo i al nodo j di peso t_i e per ogni i c'è un arco di peso 0 dal nodo 0 al nodo i (questo serve a fissare il tempo d'inizio del progetto a 0). Chiaramente G è un DAG pesato perché le dipendenze non creano cicli. Ora è facile vedere che, per ogni i, il massimo peso $M[i]$ di un cammino in G dal nodo 0 al nodo i è un valore ammissibile per x_i. Infatti, ogni dipendenza (i, j), ovvero ogni vincolo $x_j \geq x_i + t_i$, è soddisfatto dato che $M[j] \geq M[i] + t_i$ in quanto in G c'è l'arco (i, j) di peso t_i ed essendo $M[i]$ il peso di un cammino da 0 a i, concatenando tale cammino con l'arco (i, j) si ottiene un cammino da 0 a j di peso $M[i] + t_i$ che deve

essere minore od uguale a *M*[*j*]. I valori non sono solo ammissibili come tempi d'inizio ma sono anche i minimi possibili perchè *M*[*i*], essendo la somma dei pesi degli archi di un cammino da 0 al nodo *i*, è anche la somma delle durate di una catena di attività ognuna delle quali ha una dipendenza con quella che la precede. Quindi il calcolo dei longest path del grafo *G* dal nodo 0 risolve il problema delle pianificazione delle attività. L'algoritmo `LONGESTPATH`, leggermente modificato, calcola i pesi massimi dei cammini dal nodo *0* e risolve il problema in tempo lineare nella dimensione del grafo *G* e quindi in tempo lineare nel numero di attività e dipendenze. Il tempo di completamento *T* del progetto è dato da

$$T = \max\{M[i] + t_i \mid i = 1, \ldots, n\}$$

Un cammino massimo che dà il valore di *T* è chiamato *cammino critico* (*critical path*) da qui il nome *Critical Path Method* (CPM) che è stato dato a questa metodologia per determinare la pianificazione delle attività e la durata minima di un progetto.

Se i pesi degli archi del grafo *G* vengono cambiati di segno, e quindi tutti i pesi positivi diventano negativi, i cammini di peso massimo diventano cammini di peso minimo. Questo non crea problemi perchè *G* non ha cicli negativi essendo aciclico. Quindi l'algoritmo `LONGESTPATH` può calcolare i cammini minimi in un DAG anche se ci sono pesi negativi. Come vedremo tra poco un algoritmo un po' più complicato ma che usa sempre la programmazione dinamica calcola i cammini minimi in grafi pesati con pesi sia positivi che negativi, purchè ovviamente non abbiano cicli di peso negativo.

Sistemi con vincoli di differenza

Le esecuzioni di un insieme di attività potrebbero essere soggette a vincoli un po' più generali di quelli considerati nel problema precedente. Oltre a poter richiedere che l'esecuzione di una attività non possa iniziare prima che un'altra sia terminata, potremmo voler imporre che un'attività debba iniziare entro un certo lasso di tempo dall'inizio di un'altra attività. Ma potremmo volere che il tempo che deve passare tra gli inizi di due attività non sia necessariamente uguale al tempo di esecuzione di una delle due ma possa variare.

Per rappresentare questo tipo di vincoli denotando con $x_1,\ldots,x_n$ i tempi d'inizio (incogniti) delle *n* attività. Per imporre che un'attività *j* inizi solo dopo un certo tempo *t* dall'inizio di un'attività *i*, introduciamo il vincolo:

$$x_j \geqslant x_i + t$$

Per imporre che l'attività *j* inizi entro un certo lasso di tempo *t* dall'inzio di un'attività *i*, introduciamo il vincolo

$$x_j \leqslant x_i + t$$

Entrambi i tipi di vincoli possono essere posti in una forma canonica:

$$x_i - x_j \leqslant b$$

dove *b* può essere positivo, negativo o nullo. Questo tipo di vincoli vengono appunto chiamati *vincoli di differenza* (*difference constraints*). Un insieme di vincoli di differenza è chiamato un *sistema di vincoli di differenza*. Questi hanno molte applicazioni.

Dato un sistema di vincoli di differenza nelle incognite $x_1,\ldots,x_n$, primariamente vogliamo sapere se ammette soluzioni. Infatti, è facile trovare sistemi che non ammettono soluzioni:

$$\begin{aligned} x_1 - x_2 &\leqslant 1 \\ x_2 - x_1 &\leqslant -2 \end{aligned}$$

Una volta stabilito che un sistema ammette soluzioni, vogliamo trovarne almeno una. In generale ci sono moltissime soluzioni. Si può facilmente vedere che se

$$\bar{x}_1, \ldots, \bar{x}_n$$

è una soluzione, allora anche

$$\bar{x}_1 + \delta, \ldots, \bar{x}_n + \delta$$

per un qualsiasi valore δ, è anch'essa una soluzione. Infatti, per ogni vincolo

$$\bar{x}_i - \bar{x}_j \leqslant b \quad \Longrightarrow \quad (\bar{x}_i + \delta) - (\bar{x}_j + \delta) \leqslant b$$

Così come abbiamo fatto per il problema CPM, vogliamo ridurre il problema della soluzione di un sistema di vincoli di differenza a un problema di cammini su un opportuno grafo. Per fare ciò estendiamo il grafo relativo a quel problema. Definiamo un grafo diretto e pesato G_d che ha un nodo 0 e per ogni incognita x_i un nodo i. Per ogni vincolo $x_i - x_j \leq b$, c'è un arco da j a i di peso b. Infine, c'è un arco di peso 0 dal nodo 0 verso ogni nodo i. In generale, il grafo G_d ha anche pesi negativi. In effetti ha pesi negativi proprio in corrispondenza a vincoli del tipo $x_i - x_j \leq -t$, che sono vincoli dello stesso tipo di quelli del problema CPM.

Se G_d ha un ciclo di peso negativo allora il sistema non ha soluzione.

Dimostrazione. Siano $i_1,\ldots,i_k$ i nodi di un ciclo di peso negativo. I k archi del ciclo corrispondono ai vincoli:

$$\begin{aligned} x_{i_2} - x_{i_1} &\leqslant b_1 \\ x_{i_3} - x_{i_2} &\leqslant b_2 \\ &\cdots \\ x_{i_k} - x_{i_{k-1}} &\leqslant b_k \\ x_{i_1} - x_{i_k} &\leqslant b_k \end{aligned}$$

Siccome una qualsiasi soluzione deve soddisfare tutte queste disuguaglianze, dovrà anche soddisfare la somma di esse. Sommando i membri a sinistra si ottiene che tutte le incognite si cancellano e quindi a sinistra rimane 0. A destra abbiamo invece la somma $b_1+\cdots+b_k$ che corrisponde al peso del ciclo. Quindi il soddisfacimento dei vincoli implica che il peso del ciclo deve essere non negativo, in contraddizione con l'ipotesi che il peso del ciclo è negativo.

Se G_d non ha cicli di peso negativo allora i pesi minimi $M[1],\ldots,M[n]$ dei cammini dal nodo 0 sono una soluzione del sistema (cioè, $x_i = M[i]$ per ogni i).

Dimostrazione. Per ogni vincolo $x_i - x_j \leq b$ c'è un arco da j a i di peso b in G_d. Quindi $M[i] \leq M[j] + b$ perché se così non fosse ci sarebbe un cammino da 0 a i, formato da un cammino da 0 a j e l'arco (j, i), di peso inferiore a $M[i]$. Perciò il vincolo è soddisfatto: $M[i] - M[j] \leq b$.

Perciò per sapere se un sistema di vincoli di differenza ammette soluzioni e in tal caso trovarne una, basta risolvere il problema dei cammini minimi in un grafo pesato con pesi anche negativi.

Esercizio [viaggio]

Bisogna affrontare un viaggio in auto che prevede n tappe e gli interi d_i, $1 \leq i < n$ rappresentano il numero di litri di benzina necessari per spostarsi dalla località i alla successiva. Sappiamo che il serbatoio dell'auto ha una capacità C e che un litro di benzina se acquistato nella località i ha un costo p_i, $1 \leq i < n$. Descrivere un algoritmo che calcola il costo minimo per portare a termine il viaggio, in $O(nC^2)$ tempo.