Complementi ed Esercizi di Informatica Teorica II

Vincenzo Bonifaci

14 febbraio 2008

8 Algoritmi esatti per problemi NP-ardui

Sotto l'assunzione P $\neq$ NP, se si vuole risolvere un problema NP-arduo è necessario ricorrere ad un algoritmo con tempo di esecuzione super-polinomiale. Ogni problema in NP può essere risolto attraverso una ricerca esaustiva dei suoi possibili certificati: se per un certo problema in NP, $m(x)$ è la lunghezza massima di un certificato per una istanza x, esiste un algoritmo con complessità temporale pari a $O(2^{m(x)} \cdot \text{poly}(x))$: semplicemente generiamo tutte le possibili stringhe binarie di lunghezza $m(x)$ e verifichiamo per ognuna di queste in tempo polinomiale se si tratta di un certificato valido per l'istanza. Ognuno di questi controlli può essere effettuato in tempo polinomiale per l'ipotesi che il problema di partenza è in NP.

Utilizziamo una notazione $O(\cdot)$ modificata per sopprimere fattori polinomiali dato che questi, asintoticamente, non giocano un ruolo di rilievo. In particolare usiamo la notazione $O^*(T(|x|))$ per indicare una complessità temporale pari a $O(T(|x|) \cdot \text{poly}(|x|))$.

Il nostro obiettivo è quello di individuare algoritmi con tempo di esecuzione $O^*(2^{cm(x)})$ per un valore il più piccolo possibile di c. Molti problemi classici di ottimizzazione rientrano in una delle seguenti tre categorie: *problemi di sottoinsieme*, in cui ogni soluzione valida può essere specificata come sottoinsieme di un universo di base; *problemi di permutazione*, in cui ogni soluzione valida può essere specificata come permutazione di un universo di base; *problemi di partizione*, in cui ogni soluzione valida può essere specificata come partizione di un universo di base. Per un universo di base di cardinalità n, un problema di sottoinsieme può essere risolto in modo banale in tempo $O^*(2^n)$ semplicemente enumerando tutti i possibili sottoinsiemi. Allo stesso modo, un problema di permutazione può essere sempre risolto in tempo $O^*(n!)$ e un problema di partizione in tempo $O^*(c^{n \log n})$ per qualche costante $c > 1$.

8.1 Programmazione dinamica

8.1.1 Il commesso viaggiatore

Nel problema del commesso viaggiatore, un venditore deve visitare n città $1, 2, \ldots, n$. Inizia dalla città 1, prosegue attraverso le rimanenti $n - 1$ città

in ordine arbitrario e infine ritorna al punto di partenza, nella città 1. La distanza dalla città i alla città j è dato da $d(i,j)$. L'obiettivo è minimizzare la distanza totale percorsa dal commesso.

Il problema del commesso viaggiatore è un problema di permutazione e quindi può essere chiaramente risolto in tempo $O^*(n!)$. Presentiamo un algoritmo meno banale dovuto a Held e Karp che risolve lo stesso problema in tempo $O^*(2^n)$. L'algoritmo, seguendo il paradigma della programmazione dinamica, sfrutta una formulazione ricorsiva della lunghezza di un cammino minimo che visiti un certo sottoinsieme S di città. In particolare, sia $S \subseteq \{2,\ldots,n\}$. Per ogni città $i \in S$, indichiamo con $\textsc{opt}[S;i]$ la lunghezza del più corto cammino che inizia nella città 1, visita tutte le città in S in un ordine qualsiasi e infine termina nella città i. Abbiamo

$$\begin{aligned} \textsc{opt}[\{i\};i] &= d(1,i) \\ \textsc{opt}[S;i] &= \min\{\textsc{opt}[S-\{i\};j] + d(j,i) : j \in S - \{i\}\}. \end{aligned}$$

La seconda equazione è giustificata dal fatto che ogni cammino che visiti S e termini in i deve necessariamente prima visitare $S - \{i\}$, terminando in una qualche città j, per poi andare da j ad i. Inoltre se il cammino fino a i è ottimo, deve essere ottimo anche il sottocammino fino a j, altrimenti rimpiazzandolo con un cammino migliore si migliorerebbe anche il cammino fino ad i, arrivando così ad una contraddizione.

Considerando tutti i possibili sottoinsiemi S in ordine di cardinalità crescente, possiamo calcolare il valore di ogni $\textsc{opt}[S;i]$ in tempo proporzionale a $|S|$ (che è al più n). Quindi possiamo calcolare tutti i valori $\textsc{opt}[S;i]$ in tempo totale $O(n^2 2^n) = O^*(2^n)$. Per ottenere la lunghezza del tour ottimo, a questo punto è sufficiente calcolare, in tempo lineare, il minimo di $\textsc{opt}[\{2,\ldots,n\};j] + d(j,1)$ per j che varia tra 2 ed n.

Siamo quindi riusciti a trovare un algoritmo con una complessità temporale decisamente migliore di $O^*(n!)$. Il prezzo pagato, purtroppo, è che lo *spazio* usato da questo algoritmo è anch'esso $O^*(2^n)$ (mentre l'algoritmo che tenta tutte le possibili permutazioni usa spazio lineare). Comunque, a tutt'oggi non si conoscono algoritmi più efficienti di questo dal punto di vista della complessità temporale.

8.1.2 Numero cromatico

Il problema del numero cromatico consiste nel trovare, per un grafo dato, il minimo numero di colori necessari per colorare ogni vertice in maniera tale che due vertici adiacenti non abbiano mai lo stesso colore. Si tratta di un problema di partizione (ogni colorazione infatti partiziona i vertici a seconda del loro colore) e quindi può essere risolto banalmente in tempo $O^*(c^{n \log n})$. Per fare di meglio useremo ancora una volta la tecnica della programmazione dinamica. Dovremo anche appoggiarci al seguente risultato: ogni grafo di n nodi ha al più $3^{n/3}$ independent set massimali. Inoltre esiste una procedura che genera tutti questi independent set massimali in un tempo totale pari a $O(n^2 3^{n/3})$.

Sia $S \subseteq V$ un sottoinsieme di vertici del grafo. Indichiamo con $G[S]$ il sottografo di G indotto da S, e indichiamo con $\textsc{opt}[S]$ il numero cromatico di $G[S]$. Ovviamente se S è l'insieme vuoto abbiamo $\textsc{opt}[S] = 0$. Altrimenti, abbiamo la ricorrenza

$$\textsc{opt}[S] = 1 + \min\{\textsc{opt}[S - T] : T \text{ è un independent set massimale in } G[S]\}.$$

La ricorrenza è giustificata dal fatto che se abbiamo una colorazione ottima, per ogni colore l'insieme dei vertici di quel colore forma un independent set. Inoltre senza perdita di generalità ad almeno un colore corrisponde un independent set massimale (perché?). Quindi per colorare ottimamente $G[S]$ possiamo enumerare gli independent set massimali T, colorare $G[S - T]$ con $\textsc{opt}[S - T]$ colori e usare un colore aggiuntivo per i vertici in S.

Possiamo calcolare i valori $\textsc{opt}[S]$ in ordine di cardinalità crescente di S, cosicché la ricorrenza usi sempre dei valori già generati. Il tempo necessario a calcolare $\textsc{opt}[S]$ è dominato del tempo necessario a generare tutti i possibili independent set massimali T di $G[S]$. Da quanto abbiamo detto, ciò può essere fatto in tempo $k^2 3^{k/3}$ dove $k = |S|$. La complessità temporale totale è quindi

$$\sum_{k=0}^{n} \binom{n}{k} k^2 3^{k/3} \leq n^2 \sum_{k=0}^{n} \binom{n}{k} 3^{k/3} = n^2 (1 + 3^{1/3})^n$$

dove per l'ultima uguaglianza abbiamo usato la formula del binomio di Newton: $\sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k} = (a + b)^n$. Dato che $1 + 3^{1/3} \approx 2.4422$, abbiamo quindi una complessità temporale di $O^*(2.4422^n)$. Notiamo che anche in questo caso lo spazio richiesto dall'algoritmo è esponenziale.