Rappresentazione vettoriale di alberi m-ari

Appunti delle lezioni di Algoritmi 1 – a.a. 2004/05
Prof. Rossella Petreschi

a cura di Saverio Caminiti

Si è visto come un albero binario ammetta una rappresentazione vettoriale nel quale i figli sinistro e destro del nodo nella posizione i-esima del vettore occupano le posizioni $2i$ e $2i + 1$ rispettivamente; tale rappresentazione è ottima per alberi completi mentre comporta uno spreco di spazio per alberi non completi.

Scopo di questa lezione è mostrare come sia possibile rappresentare in forma vettoriale anche alberi qualunque. Utilizzeremo a tale scopo i codici di Prüfer [2], per una trattazione completa degli aspetti teorici relativi a tali codici rimandiamo alle dispense del corso di Combinatoria del Prof. J. Körner [1], qui ci limiteremo a discutere gli aspetti algoritmici inerenti la codifica e la decodifica.

Codifica

Sia T un albero non radicato di n nodi identificati con i numeri interi da 1 ad n. L'algoritmo di codifica elimina iterativamente dall'albero la foglia di indice minore e l'unico spigolo su essa incidente, fintantoché ci sono spigoli. Ogni spigolo eliminato viene aggiunto ad una sequenza di spigoli detta codice naturale dell'albero T.

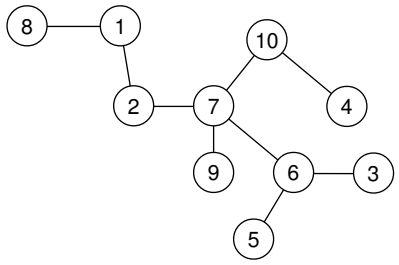


Figura 1: Esempio di albero non radicato di 10 nodi identificati con indici distinti da 1 a 10.

Ad esempio partendo dall'albero in Figura 1 il primo spigolo eliminato sarà quello incidente sulla foglia di indice 3, poi sarà la volta di quelli incidenti sulle foglie di indice 4 e poi 5, subito dopo verrà selezionato lo spigolo $(6, 7)$, infatti il 6 pur non essendo foglia in principio lo sarà diventato dopo l'eliminazione di 3 e 5; la codifica continuerà così finché rimarrà soltanto il nodo 10. Il codice naturale risultante sarà:

C:	6	10	6	7	1	2	7	7	10
	|	|	|	|	|	|	|	|	|
S:	3	4	5	6	8	1	2	9	7

Si noti che nel rappresentare il codice naturale le foglie che venivano via via eliminate sono state messe sempre in basso, chiamiamo la sequenza di tali foglie S, mentre nella parte superiore del codice, in corrispondenza di ogni foglia, compare l'unico nodo ad essa adiacente al momento della rimozione, chiamiamo questa sequenza C. Entrambe queste sequenze hanno lunghezza $n-1$ pari al numero di spigoli presenti nell'albero T. L'ultimo elemento della sequenza C sarà sempre uguale all'indice dell'unico nodo rimasto alla fine della codifica, quindi pari ad n; infatti essendo n il nodo di etichetta massima non verrà mai scelto per l'eliminazione.

Il codice di Prüfer associato ad un albero T è la sequenza dei primi $n-2$ interi che compongono la sequenza C del suo codice naturale; nell'esempio di Figura 1 è quindi 6 10 6 7 1 2 7 7.

Osservazione 1 *Tutti i nodi, tranne n, compaiono in S esattamente una volta e, dal momento che il codice naturale contiene tutti gli spigoli dell'albero, in C ogni nodo, tranne n, compare una volta meno del suo grado nell'albero (n compare in C un numero di volte pari al suo grado in T).*

Non è possibile che due alberi diversi diano luogo alla stessa stringa, dal momento che il procedimento è completamente deterministico, per una prova formale dell'iniettività della codifica si consulti [1].

Un'implementazione diretta del procedimento appena presentato richiederebbe, ad ogni passo, di cercare la foglia di indice minimo nell'albero. Implementando tale operazione con uno scorrimento di tutti i nodi dell'albero ciò richiederebbe $O(n)$ operazioni per ciascuna delle $n-1$ eliminazioni, quindi in totale $O(n^2)$ operazioni. Utilizzando strutture dati più complesse, ad esempio mantenendo l'insieme delle foglie in uno heap ordinato, il numero totale di operazioni può calare ad $O(n \log n)$. Vedremo in seguito come sia possibile aggirare tale problema e realizzare la codifica in tempo lineare.

Decodifica

Partendo da una stringa C di lunghezza l ricostruiamo l'albero ad essa associato. Anzitutto aggiungiamo alla fine di C un elemento di valore $n = l + 2$ così da riottenere esattamente la sequenza superiore del codice naturale, e vediamo come sia possibile ricostruirne la parte inferiore, ovvero la sequenza S.

In forza dell'osservazione 1 tutti i nodi che non compaiono in C sono le foglie dell'albero iniziale candidate ad occupare la prima posizione della sequenza S, fra queste verrà scelta quella di indice minore.

Riprendiamo l'esempio precedente partendo dalla sequenza 6 10 6 7 1 2 7 7, di lunghezza $l = 8$. Anzitutto aggiungiamo alla fine della sequenza il valore $10 = 8 + 2$, calcoliamo poi per ogni intero tra 1 e 10 il numero di occorrenze nella sequenza ottenuta 6 10 6 7 1 2 7 7 10:

i:	1	2	3	4	5	6	7	8	9	10
$occ[i]$:	1	1	0	0	0	2	3	0	0	2

Le foglie sono i nodi di grado 1 nell'albero, ovvero quelli che compaiono 0 volte nel codice (potrebbe far eccezione il nodo di indice n che però non comparirà mai in S) quindi 3, 4, 5, 8 e 9, tra queste quella di indice minore è 3, se ne deduce che il primo elemento di S sarà 3.

Prima di iniziare il passo successivo è necessario diminuire di 1 il numero di occorrenze del valore presente nella prima posizione del codice, dal momento che nella parte di codice ancora da analizzare il numero di occorrenze di tale valore è esattamente 1 meno di quanto calcolato in precedenza. Nell'esempio si deve decrementare da 2 ad 1 il numero di occorrenze di $C[1] = 6$, infatti nella parte di codice ancora da analizzare (10 6 7 1 2 7 7 10) il 6 compare una sola volta.

Questo processo viene iterato fino a completare tutta la sequenza S la quale conterrà 1 ed una sola volta tutti gli interi da 1 ad $n-1$ (cfr. Osservazione 1).

Nell'esempio al secondo passo sarà scelto il valore 4 ed al terzo passo il valore 5; a questo punto il numero di occorrenze del valore 6 ha raggiunto 0 e, essendo il più piccolo valore con 0 occorrenze che ancora non è stato inserito in S, sarà scelto al passo successivo. L'algoritmo procede selezionando i valori 8, 2, 1, 9 e 7.

L'accoppiamento tra i valori di C e di S ci fornisce l'elenco di tutti gli archi dell'albero codificato e di conseguenza ci permette di ricostruirlo.

Anche in questo caso un'implementazione diretta del procedimento presentato richiederebbe, un costo più che lineare a causa della ricerca del minimo, vedremo come anche la decodifica possa essere realizzata in tempo $O(n)$.

Algoritmi ottimi

Il nostro obiettivo è di realizzare degli algoritmi per codifica e decodificare un albero, il cui costo sia $O(n)$.

Algoritmo di codifica

L'idea di fondo è quella di evitare la ricerca del minimo e di scorrere i nodi dal più piccolo al più grande, appena si incontra una foglia la si estrae dall'albero. Si rende però necessario un controllo ulteriore: se nell'eliminare una foglia di indice i l'unico nodo ad essa adiacente diventa a sua volta foglia ed ha indice minore di i allora è necessario eliminarlo prima di procedere nel cercare foglie maggiori di i.

ALGORITMO DI CODIFICA
input : un albero di n nodi rappresentato tramite liste di adiacenza
output : una vettore di $n-2$ interi

calcola il grado di ogni nodo nell'albero nel vettore d
for $i = 1$ **to** $n-1$ **do**
    $v = i$
    while $d(v) = 1$ **do** *// se v è una foglia*
        sia x l'unico adiacente di v non ancora marcato
        aggiungi x alla sequenza C
        poiché v è stato estratto, marcalo
        decrementa $d(x)$ di 1
        if $(d(x) = 1)$ **and** $(x < i)$ *// se x è divenuto foglia ed è minore di i*
            then $v = x$ *// itera il ciclo while su x*
            else break *// altrimenti esce dal ciclo while*
return primi $n-2$ elementi di C

Il ciclo **while** interno si occupa dell'effettiva eliminazione delle foglie; per ogni valore di i, in primo luogo si prova ad estrarre i se è una foglia, poi, per ogni foglia estratta, se l'adiacente x è divenuto foglia ed è minore dell'indice i raggiunto, è necessario estrarlo.

Si noti che l'algoritmo proposto non elimina effettivamente i nodi dall'albero, ma simula tale estrazione marcando opportunamente i nodi e decrementando il loro grado nel vettore d.

Analizziamo ora nel dettaglio il costo dell'algoritmo. Il calcolo del grado di ogni nodo richiede semplicemente lo scorrimento della lista di adiacenza di ogni nodo quindi in tutto $O(n)$. Il ciclo **for** principale viene eseguito $n-1$ volte. Il ciclo **while** interno viene eseguito ogni volta che viene estratto un nodo, quindi in totale esattamente $n-1$ volte, ciò indipendentemente dal ciclo **for** ; infatti in alcune iterazioni del ciclo **for** il ciclo **while** interno sarà eseguito una o più volte, ma in altre iterazioni non sarà eseguito affatto, quindi nel complesso dell'intera esecuzione dell'algoritmo il corpo del ciclo **while** sarà eseguito esattamente $n-1$ volte. Tutte le operazioni interne al ciclo **while** hanno costo $O(1)$ tranne la ricerca dell'unico adiacente non estratto, per la quale è necessario scorrere la lista di adiacenza. Tuttavia la lista di adiacenza di ogni nodo verrà esaminata soltanto una volta nel corso di tutta l'esecuzione dell'algoritmo, esattamente quando il nodo viene estratto. In definitiva la ricerca dei nodi adiacenti ai nodi estratti richiede complessivamente di scorrere le liste di adiacenze di tutti i nodi e quindi in tutto $O(n)$. Se ne deduce che l'intero algoritmo richiede $O(n)$ operazioni.

Algoritmo di decodifica

Il nostro scopo è ora quello di ricostruire la sequenza S partendo da C e dal vettore delle occorrenze di ogni valore in C; anche in questo caso è necessario evitare la ricerca del minimo ed anche in questo caso riusciremo a farlo scorrendo i valori da 1 ad $n-1$ e considerando al momento opportuno valori di indice più piccolo precedentemente saltati.

Algoritmo di Decodifica
input : un vettore C di interi di lunghezza l
output : un albero di n nodi

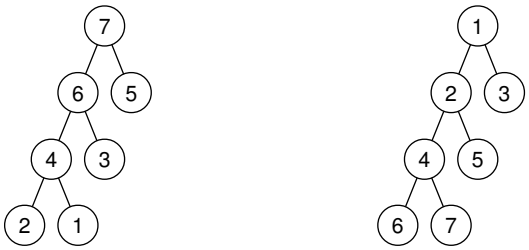
$n = l + 2$
$j = 0$
aggiungi n alla fine di C
calcola nel vettore *occ* il numero di occorrenze di ogni intero in C
for $i = 1$ **to** $n-1$ **do**
    $v = i$
    while $occ(v) = 0$ **do** *// se v ha 0 occorrenze*
        aggiungi v alla sequenza S in posizione j
        si x il j-esimo valore nella sequenza C
        decrementa $occ(x)$ di 1
        incrementa j di 1
        if $(occ(x) = 0)$ **and** $(x < i)$ *// se x ha ora 0 occorrenze ed è minore di i*
            then $v = x$ *// itera il ciclo while su x*
            else break *// altrimenti esce dal ciclo while*
return l'albero i cui archi sono $\{(C_i, S_i)$ t.c. $i \in [1, n-1]\}$

L'algoritmo è esattamente analogo a quello per la codifica, soltanto che non si ha la necessità di scorrere alcuna lista di adiacenza mentre serve la variabile j per tenere traccia di quale si la posizione del vettore S che si sta calcolando.

L'analisi del costo dell'algoritmo è analoga a quella dell'algoritmo di codifica. La ricostruzione dell'albero da restituire a partire dall'elenco degli archi può essere facilmente realizzata con $O(n)$ operazioni.

Esercizi

Provare a codificare e poi a decodificare, utilizzando prima gli algoritmi semplici e poi quelli ottimi, i seguenti alberi:



Implementare in C gli algoritmi di codifica e decodifica ottimi utilizzando come rappresentazione per gli alberi le liste di adiacenza e per le sequenze i vettori di interi.

Riferimenti bibliografici

[1] Körner, J., Malvenuto. C.: dispense del corso di Combinatoria. Dipartimento di Informatica, Università "La Sapienza".

[2] Prüfer, H.: Neuer Beweis eines Satzes über Permutationen. *Archiv für Mathematik und Physik*, 27, pp. 142–144, 1918.