Rappresentazione implicita di alberi etichettati

Appunti delle lezioni di Algoritmi e Strutture Dati – a.a. 2010/11
Prof. Rossella Petreschi

a cura di Saverio Caminiti

È note che un albero binario ammette una rappresentazione implicita vettoriale nella 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 binari completi e quasi completi, ma comporta un'eccessiva occupazione di memoria per alberi qualunque.

Nel seguito si mostra come sia possibile, sfruttando i codici di Prüfer, rappresentare in forma implicita alberi qualunque di n nodi etichettati con valori interi in $[1, n]$. Ci limiteremo a discutere gli aspetti algoritmici del problema lasciando alla letteratura specialistica gli aspetti più teorici.

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.

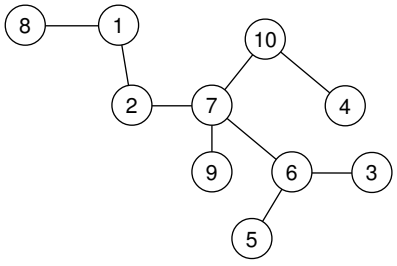


Figura 1: Esempio di albero non radicato di 10 nodi identificati con indici distinti da 1 a 10.

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 via via eliminate sono state messe in basso e costituiscono la sequenza S. Il codice vero e proprio C compare nella parte superiore e contiene, per ogni foglia f in S l'etichetta dell'unico nodo adiacente ad f al momento della sua rimozione. Sia C che S hanno lunghezza $n-1$ pari al numero di spigoli presenti nell'albero T. L'ultimo elemento della sequenza C sarà sempre uguale ad n, indice dell'unico nodo rimasto alla fine della codifica: infatti il codice di Prüfer garantisce che il nodo di etichetta massima non verrà mai scelto per l'eliminazione. Ne consegue che 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.

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

Il nostro obiettivo è quello di realizzare degli algoritmi lineari per codificare e decodificare un albero.

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, estraendo le foglie non appena si incontrano. Se nell'eliminare una foglia di indice v l'unico nodo ad essa adiacente diventa a sua volta foglia ed ha indice minore di v allora è necessario eliminarlo prima di procedere alla ricerca di foglie maggiori di v. Questo controllo va continuato fintanto che si creano nuove foglie minori di v.

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 e ponilo nel vettore d
segna ogni nodo con non marcato
for $v = 1$ **to** $n-1$ **do**
    if $d[v] = 1$ **do** *// se v è una foglia*
        sia u l'unico adiacente di v non ancora marcato
        aggiungi u alla sequenza C
        marca v e decrementa $d[u]$ di 1
        while $(d[u] = 1)$ **and** $(u < v)$ *// se u è divenuto foglia ed è minore di v*
            sia z l'unico adiacente di u non ancora marcato
            aggiungi z alla sequenza C
            marca u e decrementa $d[z]$ di 1
            $u = z$ *// verifichia z*
return primi $n-2$ elementi di C

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 corpo dell'**if** e del ciclo **while** vengono eseguiti 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. Tutte le operazioni interne nel corpo dell'**if** e del 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 quando il nodo viene estratto nel corso di tutta l'esecuzione dell'algoritmo. 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
marca tutti i nodi v con $occ[v] = 0$
for $v = 1$ **to** $n-1$ **do**
    if v è marcato **do**
        sia u il j-esimo valore nella sequenza C
        aggiungi v alla sequenza S in posizione j e incrementa j di 1
        decrementa $occ(u)$ di 1 e se diventa 0 marca u
        while (u è marcato) **and** ($u < v$)
            sia z il j-esimo valore nella sequenza C
            aggiungi u alla sequenza S in posizione j e incrementa j di 1
            decrementa $occ(z)$ di 1 e se diventa 0 marca z
            $u = z$
return l'albero i cui archi sono $\{(C_i, S_i)$ per $i \in [1, n-1]\}$

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.

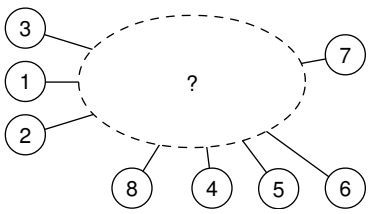
L'analisi del costo dell'algoritmo è analoga a quella dell'algoritmo di codifica. La ricostruzione dell'albero da restituire a partire dall'elenco degli archi può essere facilmente realizzata con $O(n)$ operazioni.

1 Tomografia delle reti

Il problema di stimare la struttura interna di una rete (*topologia*) a partire da misure effettuate tra nodi terminali è noto come *tomografia delle reti*. In [9] il codice di Prüfer è stato sfruttato per dedurre la struttura logica[1] di una rete a partire da una matrice OD di dimensione $r \times r$ che, per ogni coppia Origine-Destinazione (entrambi nodi terminali della rete), contiene la lunghezza del cammino tra i due nodi nella rete (cfr. Figura 2). Si assume che la struttura interna della rete sia un grafo non orientato, connesso e che ci sia un unico cammino tra ogni coppia OD (quindi la rete è un albero). I nodi terminali della rete (in seguito foglie) vengono etichettati con i valori in $[1, r]$, i nodi interni (il cui numero non è noto a priori) vengono invece etichettati con valori consecutivi a partire da $r + 1$.

	1	2	3	4	5	6	7	8
1	0	2	2	4	5	6	6	4
2	2	0	2	4	5	6	6	4
3	2	2	0	4	5	6	6	4
4	4	4	4	0	3	4	4	2
5	5	5	5	3	0	3	3	3
6	6	6	6	4	3	0	2	4
7	6	6	6	4	3	2	0	4
8	4	4	4	2	3	4	4	0

a)



b)

Figura 2: a) Esempio di matrice delle distanze OD. b) La rete la cui topologia interna deve essere dedotta ha 8 nodi terminali.

Nella prima fase l'algoritmo considera le r foglie in ordine crescente e per ciascuna crea un nodo interno. È possibile che un nodo interno creato per una foglia di etichetta i sia adiacente anche ad una foglia di etichetta $j > i$; in tal caso non sarà necessario introdurre un nuovo nodo interno per j. Per capire meglio consideriamo la matrice OD di Figura 2. Inizialmente si considera il nodo 1 e si crea un nuovo nodo con etichetta 9. Tale nodo è a distanza 1 dal nodo 1, mentre è più vicino a ciascun altro nodo di quanto lo sia il nodo 1: $d(9, x) = d(1, x) - 1$. Ciò permette di aggiornare la matrice di adiacenza per rispecchiare l'introduzione del nodo 9 aggiungendo la riga (e la analoga colonna):

	1	2	3	4	5	6	7	8	9
9	1	1	1	3	4	5	5	3	0

Dal momento che il nodo 9 è a distanza 1 anche dalle foglie 2 e 3, per queste foglie non si rende necessario creare altri nodi interni. L'algoritmo procede creando il nodo 10 adiacente al nodo 4 (ed anche al nodo 8), il nodo 11 adiacente al nodo 5 e il nodo 12 adiacente al nodo 6 (ed al 7).

Quando tutte le r foglie iniziali sono connesse ad un nodo interno la prima fase dell'algoritmo è terminata e, se il grafo ottenuto non è ancora connesso inizia una nuova fase che

[1] La struttura fisica corrisponde al livello 1 del modello OSI, mentre la struttura logica riflette il comportamento dei livello 2 e 3.

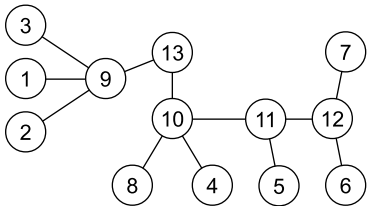
considera come foglie da connettere un sottoinsieme dei nodi creati nella fase appena conclusa. Sia S l'insieme di tali nodi (nell'esempio $S = \{9, 10, 11, 12\}$), vengono esclusi da S tutti i nodi k per cui $\exists i, j \in S$ t.c. $d(i, j) = d(i, k) + d(k, j)$.

L'algoritmo va avanti fintanto che non viene creato un albero. Ogni volta che una foglie viene esaminata l'etichetta del suo nodo adiacente appena creato (o creato per una foglia precedente) viene aggiunta alla sequenza P inizialmente vuota. Al termine dell'algoritmo la stringa P corrisponderà al codice di Prüfer della rete.

In Figura 3 viene mostrata la matrice di adiacenza completa e la rete dedotta dall'algoritmo in 2 fasi. La sequenza P restituita dall'algoritmo sarà 9991011121210131110.

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0	2	2	4	5	6	6	4	1	3	4	5	2
2	2	0	2	4	5	6	6	4	1	3	4	5	2
3	2	2	0	4	5	6	6	4	1	3	4	5	2
4	4	4	4	0	3	4	4	2	3	1	2	3	2
5	5	5	5	3	0	3	3	3	4	2	1	2	3
6	6	6	6	4	3	0	2	4	5	3	2	1	4
7	6	6	6	4	3	2	0	4	5	3	2	1	4
8	4	4	4	2	3	4	4	0	3	1	2	3	2
9	1	1	1	3	4	5	5	3	0	2	3	4	1
10	3	3	3	1	2	3	3	1	2	0	1	2	1
11	4	4	4	2	1	2	2	2	3	1	0	1	2
12	5	5	5	3	2	1	1	3	4	2	1	0	3
13	2	2	2	2	3	4	4	2	1	1	2	3	0

a)



b)

Figura 3: a) La matrice di adiacenza generata al termine dell'algoritmo. b) La rete la dedotta.

Conclusioni

Il codice di Prüfer è stato introdotto in [8]. Nel corso degli anni diversi altri codici analoghi a quello di Prüfer sono stati introdotti in letteratura tra i quali citiamo: i tre codici di Neville [6], la biiezione ϑ_n [5], i codici Blob, Happy e Dandelion [7] e il codice MHappy [3]. Per una trattazione in chiave algoritmica di tali codici si rimanda a [1, 2, 4].

Riferimenti bibliografici

[1] S. Caminiti, N. Deo, and P. Micikevičius. Linear-time Algorithms for Encoding Trees as Sequences of Node Labels. *Congressus Numerantium*, 183:65–75, 2006.

[2] S. Caminiti, I. Finocchi, and R. Petreschi. On Coding Labeled Trees. *Theoretical Computer Science*, 382(2):97–108, 2007. Special issue devoted to the best papers of LATIN'04.

[3] S. Caminiti and R. Petreschi. String Coding of Trees with Locality and Heritability. In *Proceedings of the 11th International Conference on Computing and Combinatorics (COCOON'05), LNCS 3595*, pages 251–262, 2005.

[4] S. Caminiti and R. Petreschi. Unified Parallel Encoding and Decoding Algorithms for Dandelion-Like Codes. *Journal of Parallel and Distributed Computing*, 70(11):1119–1127, 2010.

[5] Ö. Eğecioğlu and J.B. Remmel. Bijections for Cayley Trees, Spanning Trees, and Their q-Analogues. *Journal of Combinatorial Theory*, 42A(1):15–30, 1986.

[6] E.H. Neville. The Codifying of Tree-Structure. In *Proceedings of Cambridge Philosophical Society*, volume 49, pages 381–385, 1953.

[7] S. Picciotto. *How to Encode a Tree*. PhD thesis, University of California, San Diego, 1999.

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

[9] C. Vanniarajan and K. Krithivasan. Network (Tree) Topology Inference Based on Prüfer Sequence. In *Proceedings of the 16th National Conference on Communications (NCC'10)*, 2010.