

UNIVERSITÀ DEGLI STUDI DI ROMA “LA SAPIENZA”



FACOLTÀ DI SCIENZE MATEMATICHE FISICHE E NATURALI

CORSO DI LAUREA IN INFORMATICA

# SULLA RAPPRESENTAZIONE DI ALBERI TRAMITE STRINGHE

RELATORE

Prof.<sup>ssa</sup> Rossella Petreschi

LAUREANDO

Saverio Caminiti

Matricola 11107282

CORRELATORE

Dott.<sup>ssa</sup> Irene Finocchi

Anno Accademico 2002/2003

Alla mia famiglia

# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
1.1	Presentazione della tesi . . . . .	3
1.2	Nozioni preliminari . . . . .	4
1.2.1	Teorema di Cayley . . . . .	7
1.3	Tecniche parallele di base . . . . .	7
1.3.1	Tour di Eulero . . . . .	10
<b>2</b>	<b>Codificare alberi attraverso stringhe</b>	<b>12</b>
2.1	Schema generale . . . . .	12
2.2	Codici di Prüfer . . . . .	18
2.2.1	Algoritmo di codifica . . . . .	19
2.2.2	Algoritmo di decodifica . . . . .	19
2.3	Secondo codice di Neville . . . . .	20
2.3.1	Algoritmo di codifica . . . . .	21
2.3.2	Algoritmo di decodifica . . . . .	22
2.4	Terzo codice di Neville . . . . .	23

2.4.1	Algoritmo di codifica . . . . .	24
2.4.2	Algoritmo di decodifica . . . . .	25
2.5	Codici di Moon . . . . .	26
2.5.1	Algoritmi di codifica . . . . .	26
2.5.2	Algoritmi di decodifica . . . . .	27
2.6	Codici Stack-Queue . . . . .	27
2.6.1	Algoritmo di codifica . . . . .	28
2.6.2	Algoritmo di decodifica . . . . .	30
2.7	Conclusioni . . . . .	31
<b>3</b>	<b>Algoritmi sequenziali ottimi</b>	<b>33</b>
3.1	Codici di Prüfer . . . . .	33
3.1.1	Algoritmo di codifica . . . . .	33
3.1.2	Algoritmo di decodifica . . . . .	36
3.2	Secondo codice di Neville . . . . .	39
3.2.1	Algoritmo di codifica . . . . .	39
3.2.2	Algoritmo di decodifica . . . . .	41
3.3	Terzo codice di Neville . . . . .	43
3.3.1	Algoritmo di codifica . . . . .	43
3.3.2	Algoritmo di decodifica . . . . .	45
3.4	Codici Stack-Queue . . . . .	46
3.4.1	Algoritmo di codifica . . . . .	46

3.4.2	Algoritmo di decodifica . . . . .	48
3.5	Conclusioni . . . . .	49
<b>4</b>	<b>Algoritmi paralleli</b>	<b>51</b>
4.1	Algoritmi di codifica . . . . .	51
4.1.1	Codici di Prüfer . . . . .	52
4.1.2	Terzo codice di Neville . . . . .	55
4.1.3	Secondo codice di Neville . . . . .	57
4.1.4	Codici Stack-Queue . . . . .	59
4.2	Algoritmi di decodifica . . . . .	63
4.2.1	Calcolo dell'ultima occorrenza . . . . .	64
4.2.2	Terzo codice di Neville . . . . .	65
4.2.3	Codici Stack-Queue . . . . .	67
4.2.4	Secondo codice di Neville . . . . .	69
4.2.5	Codici di Prüfer . . . . .	71
4.3	Conclusioni . . . . .	73
<b>5</b>	<b>Conclusioni e sviluppi futuri</b>	<b>76</b>
	<b>Notazione</b>	<b>79</b>

# Capitolo 1

## Introduzione

Un albero è un grafo connesso aciclico privo di cappi o archi paralleli. Molte strutture dati esistono per rappresentare alberi nelle memorie di un calcolatore: da generiche liste e matrici di adiacenza a complesse strutture idonee a rendere efficiente l'implementazione di specifiche operazioni sui dati rappresentati.

Un'interessante alternativa alle comuni rappresentazioni consiste nel poter codificare alberi tramite stringhe, il più noto contributo in tal senso è costituito senza dubbio dai codici di Prüfer.

Introdotti dall'omonimo matematico tedesco in un articolo del 1918 [26], tali codici furono definiti nell'ambito della dimostrazione del teorema di Cayley (Teorema 1.2.4), per mostrare una corrispondenza uno ad uno tra alberi etichettati non radicati di  $n$  nodi e stringhe di  $n-2$  elementi appartenenti ad un insieme di cardinalità  $n$ .

In seguito, nel 1953, Neville [25] propose tre codifiche, la prima delle quali uguale a quella proposta da Prüfer. Ulteriori contributi si devono a Moon [24] che generalizzò i codici dei suoi predecessori per codificare alberi

radicati, ed a Deo e Micikevicius [12] che hanno recentemente introdotto una nuova codifica.

Oltre all'utilizzo squisitamente matematico fattone da Prüfer, la codifica di alberi tramite stringhe vede diverse applicazioni. Prima tra tutte la generazione di alberi casuali in tempo lineare: generare una stringa random e applicare un algoritmo di decodifica lineare è sicuramente più efficiente che creare l'albero inserendo archi casuali dovendo badare a non introdurre cicli. Inoltre, il fatto che sussista una relazione uno ad uno tra alberi e stringhe permette di ottenere una distribuzione di probabilità uniforme. Tale tecnica di generazione risulta particolarmente importante nel contesto del calcolo parallelo dove permette di risolvere il problema della creazione di alberi casuali con un costo lineare [10].

Alcuni di questi codici hanno proprietà che rendono possibile, ad esempio, determinare il diametro o il centro dell'albero direttamente dalla stringa, tali caratteristiche possono ovviamente essere sfruttate per generare alberi che soddisfino determinati vincoli.

È noto, inoltre, l'utilizzo della codifica di alberi in ambito di algoritmi genetici: le stringhe vengono utilizzate per descrivere i cromosomi della popolazione. Tali algoritmi vedono applicazione in euristiche per la ricerca del minor albero ricoprente che soddisfi vincoli aggiuntivi quali, ad esempio, il numero delle foglie, il grado massimo o il diametro dell'albero [13, 15].

## 1.1 Presentazione della tesi

Studiando i codici presenti in letteratura, si è notato come tutti procedano, nel calcolare la stringa corrispondente ad un dato albero, selezionando delle foglie, eliminandole dall'albero ed inserendo nella stringa i loro nodi adiacenti in un qualche ordine; tale processo viene iterato fino alla completa codifica dell'albero.

Da questa osservazione è nata l'idea di definire uno schema generale di codifica, parametrizzato da una funzione  $f$ , per la scelta del sottoinsieme delle foglie da eliminare ad ogni iterazione, e da una relazione d'ordine  $\prec$ , utilizzata per inserire nella stringa i nodi adiacenti ad ogni foglia eliminata.

Oltre a ricondurre tutti i codici noti a tale schema, nel Capitolo 2, si dimostra che per ogni funzione deterministica  $f$  e per ogni relazione d'ordine  $\prec$  è possibile ricostruire l'albero dalla stringa, la codifica è quindi sempre reversibile.

Nel Capitolo 3, vengono analizzati i costi degli algoritmi ottenuti adattando lo schema generale alle funzioni e alle relazioni d'ordine di ciascun codice. Si forniscono poi gli algoritmi ottimi per la codifica e la decodifica di tutti i codici, mentre in letteratura ne erano stati proposti soltanto per i codici di Prüfer [16, 8] e per quelli definiti da Deo e Micikevicius [12]. Gli algoritmi qui presentati tuttavia si differenziano da quelli noti in quanto si è voluto mantenere un approccio uniforme per tutti i codici: per gli algoritmi di codifica ci si riconduce sempre ad un problema di ordinamento lessicografico di coppie, mentre quelli di decodifica si basano sulla possibilità di dedurre il grado di ciascun nodo dell'albero scorrendo il codice.

La possibilità di codificare e decodificare tutti i codici presenti in letteratura con il medesimo approccio e con identiche prestazioni asintotiche, evidenzia come, almeno nel contesto del calcolo seriale, tali codici possano essere considerati sostanzialmente equivalenti.

Anche nel contesto del calcolo parallelo (Capitolo 4) gli algoritmi di codifica si possono ricondurre tutti ad un ordinamento di coppie, tuttavia soltanto per i codici di Prüfer e per il terzo codice di Neville si riesce ad ottenere un costo computazionale asintotico ottimo.

Per decodificare, invece, si ricorre al calcolo dell'ultima occorrenza di ciascun nodo nel codice, tale operazione però non ha un costo lineare e di conseguenza nessuno dei codici viene decodificato in maniera ottima; notiamo comunque che in letteratura soltanto per i codici di Prüfer [7] era stato fornito un algoritmo di decodifica, per altro sostanzialmente equivalente a quello riportato in questa tesi e quindi non ottimo. Questa tesi si conclude con alcune considerazioni relative all'efficienza delle reali implementazioni degli algoritmi paralleli presentati e ai possibili sviluppi futuri della ricerca su questi argomenti.

## 1.2 Nozioni preliminari

Segue ora un breve richiamo di alcune delle nozioni delle quali si farà uso nei capitoli successivi.

**Proposizione 1.2.1.** *Dato un albero  $T$ , sia  $F$  l'insieme delle sue foglie, per ogni  $A \subseteq F$  il grafo  $T-A$ , ottenuto eliminando da  $T$  i nodi in  $A$  e gli archi su essi incidenti, è ancora un albero.*

*Dimostrazione.* Per ogni arco  $e$  eliminato in  $T$  viene eliminato anche l'unico nodo che  $e$  connetteva al resto dell'albero ne risulta che  $T-A$  deve essere connesso. È inoltre privo di cappi ed archi paralleli dal momento che nessun arco viene aggiunto a  $T$ .  $T-A$  è quindi un albero.  $\square$

**Definizione 1.2.2 (Centro di un grafo).** Si dice centro di un grafo un vertice per il quale la massima distanza da ogni altro nodo sia minima.

Dato un albero non radicato di  $n$  nodi, è possibile computare in tempo lineare l'insieme dei suoi centri.

#### ALGORITMO RICERCA DEI CENTRI

**INPUT:** L'albero  $T = (V, E)$  di  $n$  nodi etichettati su  $[1..n]$ .

**OUTPUT:** L'insieme dei centri di  $T$ .

1. FOR  $v = 1$  TO  $n$  DO
2.     IF ( $v$  è foglia) THEN `leaves.add(v)`
3. WHILE ( $|V(T)| > 2$ ) DO
4.     sia `newleaves` una nuova lista vuota
5.     FOREACH  $v$  IN `leaves` DO
6.         sia  $u$  l'unico nodo adiacente a  $v$
7.         elimina da  $T$  il nodo  $v$  e l'arco  $(v, u)$
8.         IF ( $u$  è foglia) THEN `newleaves.add(u)`
9.     `leaves = newleaves`
10. RETURN  $V(T)$

**Teorema 1.2.3.** *L'algoritmo RICERCA DEI CENTRI calcola esattamente l'insieme dei centri dell'albero  $T$  in tempo lineare.*

*Dimostrazione.* Sia  $d(v, u)$  la distanza tra due nodi nell'albero ovvero il numero di archi del cammino che li congiunge, sia inoltre  $D(v)$  la distanza massima che intercorre tra  $v$  e uno qualsiasi degli altri nodi:

$$D(v) = \max_{u \in V(T)} \{d(v, u)\}$$

Mostriamo ora che nessuno dei nodi eliminati ad ogni iterazione è un centro, ricordando che se  $v$  è un centro allora  $D(v) = \min_{u \in V(T)} \{D(u)\}$ . Sia  $v$  una delle foglie eliminate in un generico passo e sia  $u$  l'unico nodo ad essa adiacente, sia, inoltre,  $x$  un nodo a distanza massima da  $u$  ovvero  $d(u, x) = D(u)$ . L'unico cammino che congiunge  $v$  ed  $x$  deve passare necessariamente da  $u$ , ciò implica  $d(v, x) = 1 + d(u, x)$  e quindi  $D(v) > D(u)$ , di conseguenza  $v$  non è un centro.

Il ciclo di riga 3 termina non appena l'albero si compone di sole foglie, infatti in ogni albero con 3 o più nodi almeno uno di essi non è una foglia.

Dal momento che non vi è alcuna necessità di gestire le liste in maniera ordinata, il ciclo di riga 1 risulta avere un costo lineare. Le istruzioni all'interno del ciclo di riga 5 vengono eseguite al più una volta per ciascun nodo richiedono quindi  $O(n)$  tempo. L'intero l'algoritmo è in definitiva lineare.  $\square$

Si noti che i centri restituiti dall'algoritmo sono al più due quindi, se tra i nodi dell'albero è definito un ordinamento, è possibile determinarne il massimo con un banale confronto.

### 1.2.1 Teorema di Cayley

**Teorema 1.2.4 (Cayley).** *Esistono  $n^{n-2}$  differenti alberi non radicati etichettati su  $n$  nodi.*

Per la dimostrazione di questo teorema, la cui presentazione esula dagli scopi di questa tesi, si rimanda all'opera dell'autore [6].

**Corollario 1.2.5 (Cayley).** *Esistono  $n^{n-1}$  differenti alberi radicati etichettati su  $n$  nodi.*

*Dimostrazione.* Ogni albero non radicato di  $n$  nodi può essere radicato in  $n$  modi differenti, uno per ciascun nodo. Per il teorema 1.2.4 esistono  $n^{n-2}$  alberi non radicati etichettati su  $n$  nodi, e per ognuno di essi esistono  $n$  differenti alberi radicati, in totale  $n^{n-1}$  differenti alberi radicati etichettati su  $n$  nodi. □

## 1.3 Tecniche parallele di base

La maggior parte delle nozioni preliminari necessarie alla comprensione degli algoritmi di calcolo parallelo presentati nel Capitolo 4 possono essere reperite in [19, 20]. Il modello cui faremo riferimento è la PRAM a lettura e scrittura esclusive (EREW).

**Teorema 1.3.1 (Principio di Scheduling di Brent [5, 20]).** *Sia  $n \in \mathbb{N}$  la dimensione dell'input e sia  $A$  un algoritmo che richiede  $w(n)$  operazioni e  $t(n)$  tempo su una PRAM EREW. Sia  $p(n)$  una funzione che limita il numero di processori. Se ciascuno dei  $p(n)$  processori è in grado di determinare*

in tempo  $O(t(n))$  quale delle operazioni di  $A$  esso debba simulare, allora l'algoritmo  $A$  può essere eseguito in tempo  $O(w(n)/p(n) + t(n))$  su una PRAM EREW con  $p(n)$  processori.

Sia  $*$  un'operazione binaria associativa su un dominio  $\mathcal{D}$ . Il problema del calcolo delle somme prefisse consiste nel calcolare le somme:

$$\sum_{i=1}^j x_i$$

per ogni  $j \in [1..n]$ , dove la sommatoria è da intendersi relativa all'operazione  $*$  e gli elementi  $x_i$  appartengono tutti al dominio  $\mathcal{D}$ .

**Teorema 1.3.2 (Somme prefisse [14]).** *Il problema della computazione prefissa può essere risolto in  $O(\log n)$  tempo con  $n/\log n$  processori su una PRAM EREW.*

**Teorema 1.3.3 (List Ranking [2]).** *Data una lista di  $n$  elementi la distanza di ciascun elemento dalla testa della lista, può essere determinata in  $O(\log n)$  tempo su una PRAM EREW con  $n/\log n$  processori.*

**Teorema 1.3.4 (Broadcast [19]).** *Data una PRAM EREW con  $p$  processori, è possibile protrarre un valore a tutti i processori in tempo  $O(\log p)$ .*

Un albero binario regolare è un albero in cui ciascun nodo interno ha esattamente due figli. Dato un generico albero  $T$  è possibile trasformarlo in un albero binario regolare  $T_R$  aggiungendo dei nodi: per ogni nodo  $v$ , se  $v$  ha un solo figlio si aggiunge un nodo come secondo figlio, se  $v$  ha più di due figli si applica la sostituzione schematizzata in Figura 1.1. L'albero così ottenuto ha un numero di nodi pari ad  $O(n)$ , inoltre un nodo  $v$  non dista più di  $\log k$  dal suo padre originario  $u$ , dove  $k$  è il numero di figli di  $u$  in  $T$ .

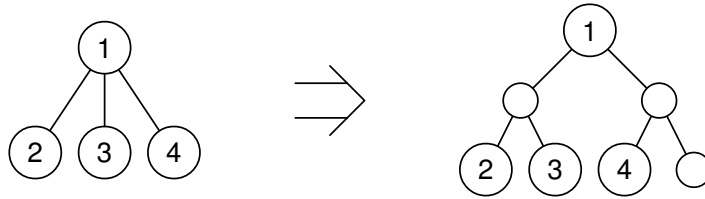


Figura 1.1: Sostituzione di un nodo con più di due figli nella trasformazione di un albero in albero binario regolare.

**Teorema 1.3.5 (Albero binario regolare [16, p. 9]).** *Dato un albero  $T$  è possibile trasformarlo in un albero binario regolare  $T_R$  in  $O(\log n)$  tempo con  $n/\log n$  processori su una PRAM EREW.*

Sia  $*$  un'operazione binaria associativa su un dominio  $\mathcal{D}$ . Sia  $T$  un albero etichettato su  $\mathcal{D}$ , il *problema della contrazione parallela di alberi* (Rake) consiste nel determinare, per ogni nodo  $v$ , il risultato dell'applicazione di  $*$  alle etichette dei nodi del sottoalbero di  $v$ .

**Teorema 1.3.6 (Rake [1, 18, 23, 27]).** *Il problema della contrazione parallela di un albero regolare  $T_R$  può essere risolto in  $O(\log n)$  tempo con  $n/\log n$  processori su una PRAM EREW.*

Si noti in particolare che, essendo il massimo un'operazione binaria associativa, tramite la tecnica del Rake è possibile determinare la massima etichetta nel sottoalbero di ogni nodo in un albero regolare.

**Teorema 1.3.7 (Centri di un albero [21]).** *Dato un albero non radicato  $T$  il suo centro (o uno dei suoi centri se l'albero è bicentrico) può essere determinato in  $O(\log n)$  tempo con  $n/\log n$  processori su una PRAM EREW.*

**Teorema 1.3.8 (Ordinamento [9, 19]).** *Una lista di  $n$  elementi può essere ordinata in  $O(\log n)$  tempo su una PRAM EREW con  $n$  processori.*

**Teorema 1.3.9 (Simulabilità tra modelli di PRAM [19]).** *Una lettura o una scrittura concorrente di una PRAM CRCW con priorità con  $p$  processori, può essere simulata su una PRAM EREW con  $p$  processori in tempo  $O(\log p)$ .*

Dati due insiemi  $S = \{s_1, \dots, s_k\}$  e  $T = \{t_1, \dots, t_l\}$  di punti sul piano, il problema del conteggio della dominanza tra due insiemi consiste nel determinare, per ogni  $t_i$ , quanti punti di  $S$  esso domina. Ricordiamo che  $t = (t_x, t_y)$  domina  $s = (s_x, s_y)$  se e solo se  $t_x > s_x$  e  $t_y > s_y$ .

**Teorema 1.3.10 (Dominanza tra due insiemi [7, 3]).** *Il problema del conteggio della dominanza tra due insiemi può essere risolto su una PRAM EREW con  $n$  processori in tempo  $O(\log n)$ .*

### 1.3.1 Tour di Eulero

Un *Tour di Eulero* è un circuito su un grafo diretto che attraversa ciascun arco esattamente una volta. Al fine di poter determinare il Tour di Eulero di un albero non radicato lo si trasforma in un grafo diretto, sostituendo ciascun arco non orientato  $\{u, v\}$  con la coppia di archi orientati  $(u, v)$  e  $(v, u)$ .

**Teorema 1.3.11 (Tour di Eulero [19, 28]).** *Il Tour di Eulero di un albero non radicato di  $n$  nodi può essere computato su una PRAM EREW in  $O(\log n)$  tempo con  $n/\log n$  processori.*

È importante esplicitare che tale risultato si basa sull'assunzione che l'albero in input sia rappresentato tramite liste di adiacenza circolari dotate di puntatori addizionali: per ogni arco  $(u, v)$ , l'occorrenza di  $u$  nella lista di adiacenza di  $v$  deve puntare all'occorrenza di  $v$  nella lista di adiacenza di

*u.* In [17] Greenlaw e Petreschi mostrano come sia possibile ottenere tale struttura partendo da una lista di archi, fornendo un algoritmo per PRAM EREW che richiede  $n$  processori e  $O(\log n)$  tempo.

Sottolineiamo come da tale struttura dati sia possibile comprendere quali nodi siano foglie in tempo costante e senza letture o scritture concorrenti. Le foglie sono infatti i nodi di grado pari ad 1, ovvero quelli la cui lista di adiacenza si compone di un solo elemento; essendo tali liste circolari è sufficiente verificare se il primo elemento punta a se stesso come successivo.

Presentiamo, infine, alcune applicazioni del Tour di Eulero.

**Teorema 1.3.12 (Radicare un albero [19]).** *Dato il Tour di Eulero di un albero  $T$  e un nodo  $v \in V(T)$ , è possibile radicare  $T$  in  $v$  su una PRAM EREW in  $O(\log n)$  tempo con  $n/\log n$  processori, ottenendo, per ogni nodo  $i$ , l'indice nel nodo padre: `parent[i]`.*

**Teorema 1.3.13 (Distanza dalla radice [19]).** *Dato il Tour di Eulero di un albero  $T$  ed il nodo  $r \in V(T)$  in cui  $T$  è stato radicato, è possibile calcolare la distanza di ogni nodo da  $r$ , su una PRAM EREW in  $O(\log n)$  tempo con  $n/\log n$  processori.*

## Capitolo 2

# Codificare alberi attraverso stringhe

Nel presente capitolo saranno proposte alcune codifiche che permettono di descrivere alberi di  $n$  nodi etichettati su un insieme  $\mathcal{L}$  con sequenze di elementi di  $\mathcal{L}$ .

Prima di introdurre specifiche codifiche verrà presentato uno schema generale di codifica che permette di generare una sequenza ordinata di tutti gli archi di un albero procedendo per selezione iterativa di foglie dell'albero stesso.

Da questo punto in poi assumeremo che l'albero  $T$  abbia  $n$  nodi, che sia radicato in  $r$  e che tutti i suoi nodi abbiano etichette distinte e appartenenti ai primi  $n$  interi:  $\mathcal{L} = [1..n]$ .

### 2.1 Schema generale

Lo schema di codifica fa uso di una funzione  $f : \mathcal{P}(V) \rightarrow \mathcal{P}(V)$  per scegliere ad ogni passo quali foglie selezionare e di una relazione d'ordine totale su

[1..n] indicata con  $\prec$ .

Al primo passo dell'iterazione sia  $T_1 = T$  e sia  $F_1 = F(T_1)$  l'insieme delle sue foglie, sia  $A_1$  un sottoinsieme di esse scelto dalla funzione  $f$ :  $A_1 = f(F_1)$ . Siano  $\{a_1^1, \dots, a_1^{k_1}\}$  gli elementi di  $A_1$  elencati secondo la relazione d'ordine  $\prec$ . Sia, infine,  $b_1^i$  l'unico nodo di  $T_1$  adiacente ad  $a_1^i$ ,  $\forall i \in [1..k_1]$ . Chiameremo  $S_1$  la sequenza di archi incidenti sui nodi di  $A_1$ , ovvero:

$$S_1 = ((a_1^1, b_1^1), (a_1^2, b_1^2), \dots, (a_1^{k_1}, b_1^{k_1}))$$

Poiché per la Proposizione 1.2.1,  $T_2 = T_1 - A_1$  è ancora un albero, è lecito iterare il processo di selezione appena descritto.

In generale all' $i$ -esimo passo dell'iterazione si avrà

- (1)  $T_i = T_{i-1} - A_{i-1}$
- (2)  $F_i = F(T_i)$
- (3)  $A_i = f(F_i) = \{a_i^1, \dots, a_i^{k_i}\}$ , con  $a_i^1 \prec a_i^2 \prec \dots \prec a_i^{k_i}$
- (4)  $(a_i^j, b_i^j) \in E(T_i) \quad \forall j \in [1..k_i]$

L'iterazione si fermerà quando l'albero non avrà più foglie, l'ultimo passo sarà quindi  $m$  tale che  $F(T_m) \neq \emptyset$  e  $F(T_{m+1}) = \emptyset$ .

Ad ogni iterazione si ottiene una sequenza di archi incidenti sui nodi di  $A_i$ :

$$S_i = ((a_i^1, b_i^1), (a_i^2, b_i^2), \dots, (a_i^{k_i}, b_i^{k_i}))$$

Concatenando le sequenze  $S_i$  per  $1 \leq i \leq m$  si ottiene una sequenza di tutti gli archi dell'albero iniziale  $T$ :

$$S = ((a_1^1, b_1^1), \dots, (a_1^{k_1}, b_1^{k_1}), (a_2^1, b_2^1), \dots, (a_2^{k_2}, b_2^{k_2}), \dots, (a_m^1, b_m^1), \dots, (a_m^{k_m}, b_m^{k_m}))$$

Più in generale possiamo dire che concatenando le sequenze da  $S_i$  ad  $S_m$  si ottiene una sequenza di tutti gli archi dell'albero  $T_i$ .

Considerando le sequenze delle sole  $a_i^j$  e delle sole  $b_i^j$  della sequenza  $S$  si ottengono due stringhe di  $n-1$  interi in  $[1..n]$ :  $A = (a_1^1, \dots, a_1^{k_1}, a_2^1, \dots, a_2^{k_2}, \dots, a_m^1, \dots, a_m^{k_m})$  e  $B = (b_1^1, \dots, b_1^{k_1}, b_2^1, \dots, b_2^{k_2}, \dots, b_m^1, \dots, b_m^{k_m})$ .

Vogliamo dimostrare che  $B$  è un codice per  $T$ .

Prima però, è necessario introdurre la seguente proposizione:

**Proposizione 2.1.1.** *Ogni nodo  $v \in V(T)$  compare  $\deg(v) - 1$  volte in  $B$ , ad eccezione della radice  $r$  che vi compare  $\deg(r)$  volte.*

*Dimostrazione.* Nell'intera sequenza  $S$  sono elencati tutti gli archi del grafo, quindi ogni nodo  $v$  appare in  $S$  esattamente  $\deg(v)$  volte. Nella sequenza  $A$  compaiono tutti i nodi che vengono eliminati dall'albero durante la procedura di selezione iterativa, ovvero tutti eccetto la radice, ciascuno una sola volta.

Quindi ogni nodo  $v$  appare una volta in  $A$  e le restanti  $\deg(v) - 1$  volte in  $B$ , la radice invece compare solo in  $B$ ,  $\deg(r)$  volte.  $\square$

Osserviamo che al termine dell'iterazione l'unico nodo rimasto sarà la radice, pertanto tutti i nodi eliminati al passo  $m$  erano adiacenti soltanto ad  $r$ , quindi  $b_m^j = r, \forall j \in [1..k_m]$ .

**Teorema 2.1.2.**  *$B$  è un codice per  $T$ .*

*Dimostrazione.* Sarà sufficiente mostrare come ricostruire  $A$  a partire da  $B$ : avendo  $A$  e  $B$ , infatti, si sarà in grado di costruire un albero di  $n$  nodi con tutti gli archi elencati in  $S$ .

$B$  è una stringa di  $n - 1$  interi in  $[1..n]$  priva di informazioni addizionali sugli indici  $k_i$  generati durante la procedura iterativa, non è quindi lecito scrivere  $B$  come  $(b_1^1, \dots, b_1^{k_1}, b_2^1, \dots, b_2^{k_2}, \dots, b_m^1, \dots, b_m^{k_m})$ . Da qui in poi denoteremo  $B$  come  $(b_1, \dots, b_{n-1})$ .

Le foglie dell'albero iniziale  $T_1 = T$  sono tutti i nodi di grado 1, quindi tutti quelli che per la Proposizione 2.1.1 non compaiono nel codice  $b_1, \dots, b_{n-1}$ :  $F_1 = \{v \in [1..n] : \nexists b_j = v\}$ . Utilizzando la funzione  $f$  e la relazione d'ordine  $\prec$  si ottengono  $A_1 = f(F_1)$ ,  $s_1 = |A_1|$ ,  $A_1 = \{a_1, \dots, a_{s_1}\}$  dove  $a_1 \prec a_2 \prec \dots \prec a_{s_1}$ .

Al secondo passo dell'iterazione, essendo  $T_2 = T_1 - A_1$ , l'insieme dei suoi archi è dato dalla concatenazione delle sequenze  $S_2, \dots, S_m$ ; quindi le sue foglie sono tutti i nodi in  $[1..n] \setminus A_1$  che non compaiono in  $b_{s_1+1}, \dots, b_{n-1}$ :  $F_2 = \{v \in [1..n] \setminus A_1 : \nexists b_j = v \ j > s_1\}$ .

In generale al passo  $i$ -esimo si ha che

$$F_i = \{v \in [1..n] \setminus \bigcup_{k=1}^{i-1} A_k : \nexists b_j = v \ j > s_{i-1}\}$$

inoltre  $A_i = f(F_i)$ ,  $s_i = s_{i-1} + |A_i|$ ,  $A_i = \{a_{s_{i-1}+1}, \dots, a_{s_i}\}$  dove  $a_{s_{i-1}+1} \prec \dots \prec a_{s_i}$ . È quindi possibile ricostruire  $A$  a partire da  $B$ .  $\square$

È evidente che, essendo la procedura di codifica deterministica, ad ogni albero radicato di  $n$  nodi corrisponde una ed una sola stringa in  $[1..n]^{n-1}$ , più interessante è notare che vale anche il contrario:

**Proposizione 2.1.3.** *Per ogni  $(b_1, \dots, b_{n-1}) \in [1..n]^{n-1}$  esiste un albero radicato di  $n$  nodi la cui codifica è  $(b_1, \dots, b_{n-1})$ .*

*Dimostrazione.* Per il Corollario 1.2.5 esistono  $n^{n-1} = |[1..n]^{n-1}|$  alberi di  $n$  nodi radicati etichettati su  $[1..n]$ , quindi se per assurdo esistesse  $(b_1, \dots, b_{n-1})$  in  $[1..n]^{n-1}$  che non è codifica per alcun albero  $T$ , allora almeno due alberi distinti  $T'$  e  $T''$  dovrebbero avere la stessa codifica, da tale stringa almeno uno dei due alberi non potrebbe essere ricostruito, il che contrasta con il Teorema 2.1.2.  $\square$

Saranno di seguito riportati in forma algoritmica gli schemi di codifica e di decodifica sopra esposti.

### SCHEMA GENERALE DI CODIFICA

**INPUT:** Un albero  $T$  di  $n$  nodi radicato in  $r$  etichettato su  $[1..n]$ , una funzione di scelta  $f$ , una relazione d'ordine  $\prec$ .

**OUTPUT:** Un vettore `code` di  $n - 1$  interi in  $[1..n]$  rappresentante il codice per  $T$ .

1. `code` = ()
2.  $i = 1$
3.  $T_1 = T$
4. WHILE  $F(T_i) \neq \emptyset$  DO
5.      $A_i = f(F(T_i))$
6.      $k_i = |A_i|$
7.     ordina  $A_i$  secondo  $\prec$  ottenendo il vettore  $(a_i^1, \dots, a_i^{k_i})$
8.     sia  $(a_i^j, b_i^j)$  l'unico arco di  $T_i$  incidente su  $a_i^j \forall j \in [1..k_i]$
9.     appendi  $(b_i^1, \dots, b_i^{k_i})$  a `code`
10.     $i = i + 1$
11.     $T_i = T_{i-1} - A_{i-1}$

## SCHEMA GENERALE DI DECODIFICA

**INPUT:** Un vettore  $\text{code} = (b_1, \dots, b_{n-1})$  di interi in  $[1..n]$ , una funzione di scelta  $f$ , una relazione d'ordine  $\prec$ .

**OUTPUT:** Un albero  $T$  di  $n$  nodi radicato etichettato su  $[1..n]$ , il cui codice è  $\text{code}$ .

1.  $n = \text{code.length} + 1$
2.  $i = 0$
3.  $s_0 = 0$
4.  $A_0 = \emptyset$
5. WHILE  $s_i < n - 1$  DO
6.      $i = i + 1$
7.      $F_i = \{v \in [1..n] \setminus \bigcup_{k=1}^{i-1} A_k : \nexists b_j = v \text{ per } s_{i-1} < j < n\}$
8.      $A_i = f(F_i)$
9.      $s_i = s_{i-1} + |A_i|$
10.     ordina  $A_i$  secondo  $\prec$  ottenendo il vettore  $(a_{s_{i-1}+1}, \dots, a_{s_i})$
11.  $T = ([1..n], \{(a_i, b_i) : 1 \leq i < n\})$

L'efficienza delle implementazioni degli algoritmi di codifica e di decodifica dipende dalla funzione  $f$ , dalla relazione  $\prec$  e dalle strutture dati da esse utilizzate.

Nei successivi paragrafi saranno presentate alcune delle codifiche più note e significative, si mostreranno inoltre le scelte della funzione  $f$  e della relazione d'ordine  $\prec$  necessarie ad ottenere tali codici dallo schema generale proposto.

Saranno anche riportati esplicitamente gli algoritmi di codifica e di decodifica ottenuti adattando quelli generali alle esigenze dei vari codici. Le discussioni sui costi computazionali e sulle possibili ottimizzazioni sono rimandate al successivo capitolo.

## 2.2 Codici di Prüfer

Partendo da un albero non radicato etichettato su  $[1..n]$ , Prüfer definisce il suo codice tramite la seguente procedura iterativa: si seleziona la foglia di etichetta minima  $u$ , si concatena al codice l'etichetta del suo unico nodo adiacente  $v$  e poi si eliminano dall'albero la foglia  $u$  e l'arco  $(u, v)$ ; fino ad ottenere un codice di lunghezza  $n - 2$ .

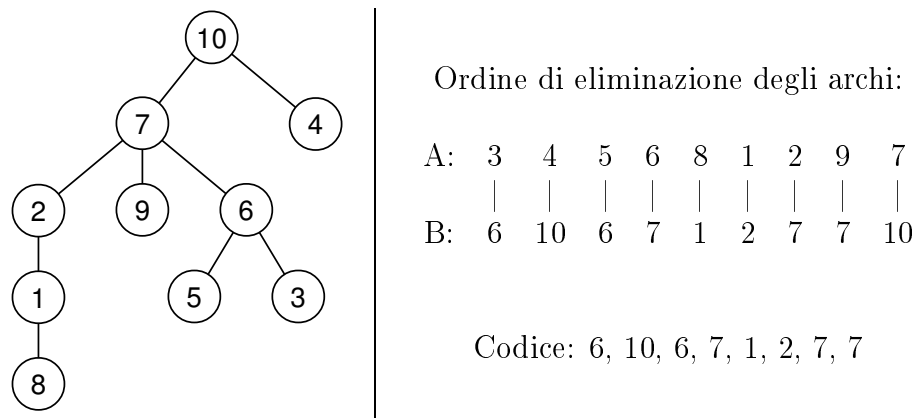


Figura 2.1: Esempio di codice di Prüfer.

La Figura 2.1 mostra un esempio di albero, l'ordine di eliminazione dei suoi archi e il suo codice di Prüfer.

### 2.2.1 Algoritmo di codifica

Per ricondurre tale codice allo schema generale di codifica precedentemente proposto è sufficiente utilizzare come funzione di scelta  $f(F) = \{\min\{v \in F\}\}$  e radicare l'albero nel nodo di etichetta massima  $r = n$ , nodo che per certo non verrà mai eliminato dalla procedura proposta da Prüfer. È da notare che la relazione d'ordine  $\prec$  è irrilevante in quanto gli insiemi  $A_i$  sono tutti di cardinalità 1. Inoltre, essendo  $b_{n-1} = r = n$  sempre, è possibile omettere questa informazione ottenendo così un codice di lunghezza  $n - 2$ .

Di seguito è riportato l'algoritmo per calcolare il codice di Prüfer ottenuto dallo schema generale opportunamente semplificato.

#### ALGORITMO CODIFICA PRÜFER

**INPUT:** Un albero  $T = (V, E)$  non radicato etichettato su  $[1..n]$  rappresentato tramite liste di adiacenza.

**OUTPUT:** Un vettore `pcode` di  $n - 2$  interi in  $[1..n]$  rappresentante il codice di Prüfer di  $T$ .

1. FOR  $i = 1$  TO  $n - 2$  DO
2.      $u = \min\{v \in F(T)\}$
3.     `pcode`[ $i$ ] =  $v : (u, v) \in E(T)$
4.      $T = T - \{u\}$

### 2.2.2 Algoritmo di decodifica

La decodifica proposta originariamente da Prüfer è sostanzialmente identica a quella proposta nello schema generale, con le semplificazioni che derivano dalla particolare funzione  $f$  impiegata.

## ALGORITMO DECODIFICA PRÜFER

**INPUT:** Un vettore `pcode` =  $(b_1, \dots, b_{n-2})$  di interi in  $[1..n]$ .

**OUTPUT:** L'albero  $T = (V, E)$  etichettato su  $[1..n]$  corrispondente a `pcode`.

1. `n = pcode.length + 2`
2. `pcode[n - 1] = n;`
3. FOR  $i = 1$  TO  $n - 1$  DO
4.      $a_i = \min\{u \in [1..n] \setminus \{a_j : 1 \leq j < i\} \setminus \{b_j : i \leq j < n\}\}$
5.  $T = ([1..n], \{(a_i, b_i) : 1 \leq i < n\})$

## 2.3 Secondo codice di Neville

Come detto in precedenza, Neville nel suo articolo del 1953 [25] propose tre codici, il primo dei quali uguale a quello proposto da Prüfer. Il secondo codice proposto si distingue dal primo perché ad ogni passo, invece di eliminare una sola foglia, elimina tutte le foglie dell'albero, in ordine di etichette crescente. Analogamente all'altro ad ogni eliminazione aggiunge al codice l'etichetta dell'unico nodo adiacente alla foglia rimossa.

È interessante notare come l'ultimo nodo che rimarrà nell'albero non sarà più necessariamente quello di etichetta massima, ma il centro dell'albero. Qualora l'albero avesse due centri a rimanere sarebbe quello di etichetta maggiore.

La Figura 2.2 mostra un esempio di albero, l'ordine di eliminazione dei suoi archi e il suo secondo codice di Neville.

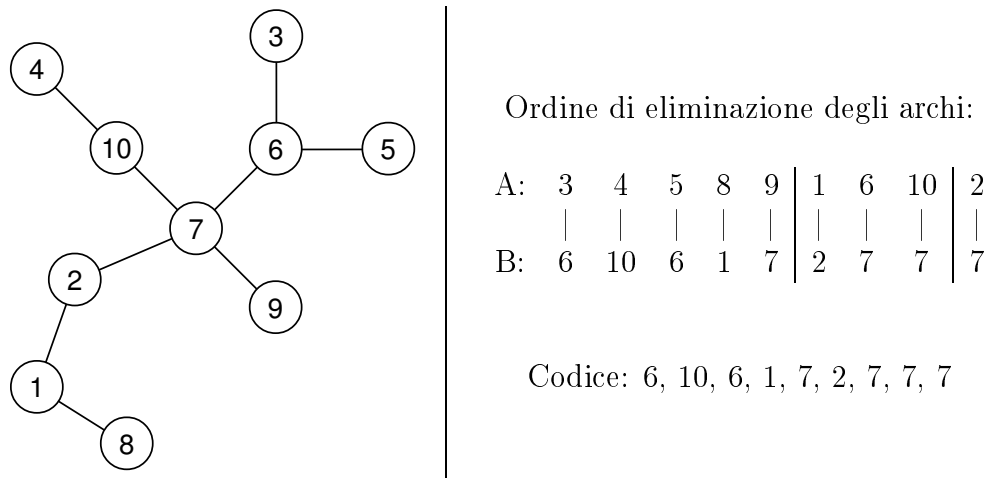


Figura 2.2: Esempio di secondo codice di Neville.

### 2.3.1 Algoritmo di codifica

Per ricondurre tale codice allo schema generale di codifica si deve utilizzare come funzione di scelta la funzione identità dato che tutte le foglie vengono scelte ad ogni passo. La relazione d'ordine  $\prec$  deve essere interpretata con la consueta relazione di minore sui naturali.

Dato che lo schema generale è definito per alberi radicati è inoltre necessario trovare il centro dell'albero di etichetta maggiore ed ivi radicare.

#### ALGORITMO CODIFICA NEVILLE II

**INPUT:** Un albero  $T = (V, E)$  non radicato etichettato su  $[1..n]$  rappresentato tramite liste di adiacenza.

**OUTPUT:** Un vettore `code` di  $n - 1$  interi in  $[1..n]$  rappresentante il secondo codice di Neville di  $T$ .

1.  $r =$  cerca il centro di  $T$  di etichetta maggiore
2. radica  $T$  in  $r$
3. `code` = ()

4.  $i = 1$
5.  $T_1 = T$
6. WHILE  $F(T_i) \neq \emptyset$  DO
7.      $A_i = F(T_i)$
8.      $k_i = |A_i|$
9.     ordina  $A_i$  in ordine crescente ottenendo il vettore  $(a_i^1, \dots, a_i^{k_i})$
10.    sia  $(a_i^j, b_i^j)$  l'unico arco di  $T_i$  incidente su  $a_i^j \forall j \in [1..k_i]$
11.    appendi  $(b_i^1, \dots, b_i^{k_i})$  a `code`
12.     $i = i + 1$
13.     $T_i = T_{i-1} - A_{i-1}$

### 2.3.2 Algoritmo di decodifica

La decodifica procede esattamente come nello schema generale.

#### ALGORITMO DECODIFICA NEVILLE II

**INPUT:** Un vettore `code` =  $(b_1, \dots, b_{n-1})$  di interi in  $[1..n]$ .

**OUTPUT:** L'albero  $T = (V, E)$  etichettato su  $[1..n]$  corrispondente a `code`.

1.  $n = \text{code.length} + 1$
2.  $i = 0$
3.  $s_0 = 0$
4.  $A_0 = \emptyset$
5. WHILE  $s_i < n - 1$  DO
6.      $i = i + 1$
7.      $F_i = \{v \in [1..n] \setminus \{a_j : 1 \leq j < s_{i-1}\} \setminus \{b_j : s_{i-1} \leq j < n\}\}$
8.      $A_i = F_i$

9.  $s_i = s_{i-1} + |A_i|$
10. ordina  $A_i$  in ordine crescente ottenendo il vettore  $(a_{s_{i-1}+1}, \dots, a_{s_i})$
11.  $T = ([1..n], \{(a_i, b_i) : 1 \leq i < n\})$

## 2.4 Terzo codice di Neville

L'ultimo codice proposto da Neville è una variante del primo e impone di eliminare il nodo  $v$  adiacente alla foglia  $u$  al passo subito successivo all'eliminazione di  $u$ , purché  $v$  stesso sia divenuto una foglia. In tutti i casi in cui la regola sopra esposta non può essere applicata si sceglie la foglia di etichetta minima.

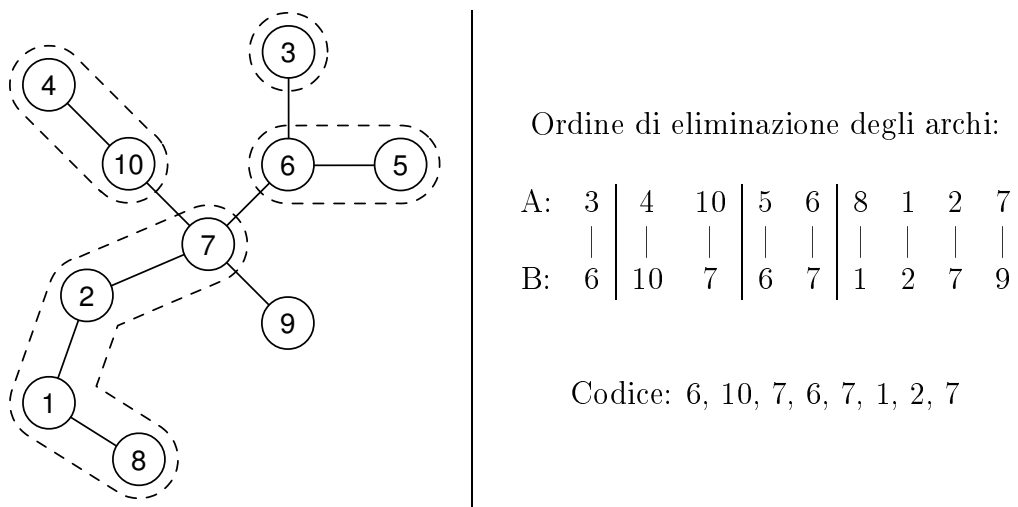


Figura 2.3: Esempio di terzo codice di Neville.

La Figura 2.3 mostra un esempio di albero, l'ordine di eliminazione dei suoi archi e il suo terzo codice di Neville. In figura sono inoltre evidenziati dei blocchi ciascuno corrispondente alla catena che viene eliminata dopo aver scelto una foglia di etichetta minima. Ricordiamo comunque che ad ogni iterazione un solo nodo viene eliminato.

Per comprendere quale sia il nodo che rimarrà al termine dell'esecuzione dell'algoritmo è importante notare che, ogni volta che un nodo inizialmente interno diviene esterno, questo viene eliminato al passo successivo. Nei passi in cui si sceglie una foglia di etichetta minima quindi, lo si fa considerando solo un sottoinsieme delle foglie dell'albero iniziale. In particolare prima di iniziare ad eliminare l'ultima catena si dovrà scegliere tra la foglia di etichetta massima e quella subito minore. Il nodo che rimarrà sarà quindi l'estremo dell'ultima catena che non è stato scelto come foglia di etichetta minore tra queste due, quindi la foglia di etichetta massima tra tutte quelle dell'albero iniziale.

### 2.4.1 Algoritmo di codifica

Usando come funzione di scelta:

$$f(F_i) = \begin{cases} \{b_{i-1}\} & \text{se } b_{i-1} \text{ è ora una foglia;} \\ \{\min\{F_i\}\} & \text{altrimenti.} \end{cases}$$

e radicando in  $r = \max\{F(T)\}$  ci si può ricondurre allo schema generale di codifica. Anche in questo caso la relazione d'ordine  $\prec$  è irrilevante in quanto gli insiemi  $A_i$  sono tutti di cardinalità 1.

Come nel caso dei codici di Prüfer l'algoritmo risulta semplificato rispetto allo schema generale.

### ALGORITMO CODIFICA NEVILLE III

**INPUT:** Un albero  $T = (V, E)$  non radicato etichettato su  $[1..n]$  rappresentato tramite liste di adiacenza.

**OUTPUT:** Un vettore `code` di  $n - 1$  interi in  $[1..n]$  rappresentante il terzo codice di Neville di  $T$ .

1. FOR  $i = 1$  TO  $n - 1$  DO
2.     IF  $(i = 1 \parallel \text{code}[i - 1] \notin F(T))$  THEN
3.          $u = \min\{v \in F(T)\}$
4.     ELSE
5.          $u = \text{code}[i - 1]$
6.      $\text{code}[i] = v : (u, v) \in E(T)$
7.      $T = T - \{u\}$

Si noti che anche in questo caso è possibile ridurre il codice, infatti le foglie dell'albero iniziale sono tutti e soli i nodi che non compaiono nel codice, ad eccezione di  $r$  che ne occuperà l'ultima posizione. Riducendo quindi il codice ai primi  $n - 2$  elementi è possibile ricavare  $b_{n-1}$  come il massimo nodo che non compare nel codice.

### 2.4.2 Algoritmo di decodifica

La decodifica è sostanzialmente identica a quella proposta nello schema generale, con alcune semplificazioni.

#### ALGORITMO DECODIFICA NEVILLE III

**INPUT:** Un vettore  $\text{code} = (b_1, \dots, b_{n-2})$  di interi in  $[1..n]$ .

**OUTPUT:** L'albero  $T = (V, E)$  etichettato su  $[1..n]$  corrispondente a  $\text{code}$ .

1.  $n = \text{code.length} + 2$
2.  $b_{n-1} = \max_{v \in [1..n]} \{v : v \notin \{b_1, \dots, b_{n-2}\}\}$
3.  $b_0 = 0$
4. FOR  $i = 1$  TO  $n - 1$  DO
5.      $F_i = \{v \in [1..n] \setminus \{a_j : 1 \leq j < i\} \setminus \{b_j : i \leq j < n\}\}$

6. IF  $(b_{i-1} \in F_i)$  THEN
7.          $a_i = b_{i-1}$
8. ELSE
9.          $a_i = \min\{F_i\}$
10.  $T = ([1..n], \{(a_i, b_i) : 1 \leq i < n\})$

## 2.5 Codici di Moon

I tre codici appena proposti sono stati tutti originariamente definiti dai loro creatori per alberi non radicati, ma dalla loro funzione di scelta si poteva chiaramente dedurre quale fosse il nodo che mai veniva eliminato; gli alberi vengono quindi implicitamente radicati. Moon [24] definisce delle varianti di tali codici ammettendo che l'albero originario sia radicato in un nodo  $r$  arbitrario, tale nodo non sarà quindi mai considerato nell'insieme delle foglie e quindi mai eliminato.

Si noti che non vale più l'osservazione fatta per i codici di Prüfer che  $b_{n-1} = n$ , e neppure la deduzione che nel terzo codice di Neville  $b_{n-1}$  è la massima foglia. Non sarà quindi possibile omettere tali informazioni: i codici avranno tutti lunghezza  $n - 1$ .

Pochissime modifiche devono essere apportate agli algoritmi presentati per ammettere la variante introdotta da Moon.

### 2.5.1 Algoritmi di codifica

Gli algoritmi proposti per i codici di Prüfer e per il terzo codice di Neville non richiedono alcuna modifica, semplicemente sarà necessario utilizzare strutture

dati tali da assicurare che la radice dell'albero non compaia mai negli insiemi delle foglie.

Per il secondo codice di Neville, oltre a tale accorgimento, si dovranno omettere i passi necessari a cercare il centro di etichetta maggiore e a radicare l'albero.

### 2.5.2 Algoritmi di decodifica

L'unica modifica da apportare, comune a tutti gli algoritmi, è quella di aggiungere al termine un passo per radicare l'albero  $T$  ottenuto nel nodo all'ultima posizione del codice:  $r = b_{n-1}$ .

## 2.6 Codici Stack-Queue

L'ultimo codice che viene presentato è stato proposto da Deo e Micikevicius [12] fornendo direttamente gli algoritmi ottimi per la codifica e la decodifica; il primo fa uso di una coda mentre il secondo di una pila: da cui il nome di codici Stack-Queue.

Partendo da un albero non radicato prima vengono eliminate le foglie dell'albero iniziale in ordine crescente, poi i restanti nodi nell'ordine in cui diventano foglie.

Vedremo in seguito che, come per i codici di Prüfer, è possibile ricostruire l'albero anche se si elimina dal codice l'ultimo elemento, permettendo così di rappresentare l'albero con un codice di lunghezza  $n - 2$ .

La Figura 2.4 mostra un esempio di albero, l'ordine di eliminazione dei suoi archi e il suo codice Stack-Queue.

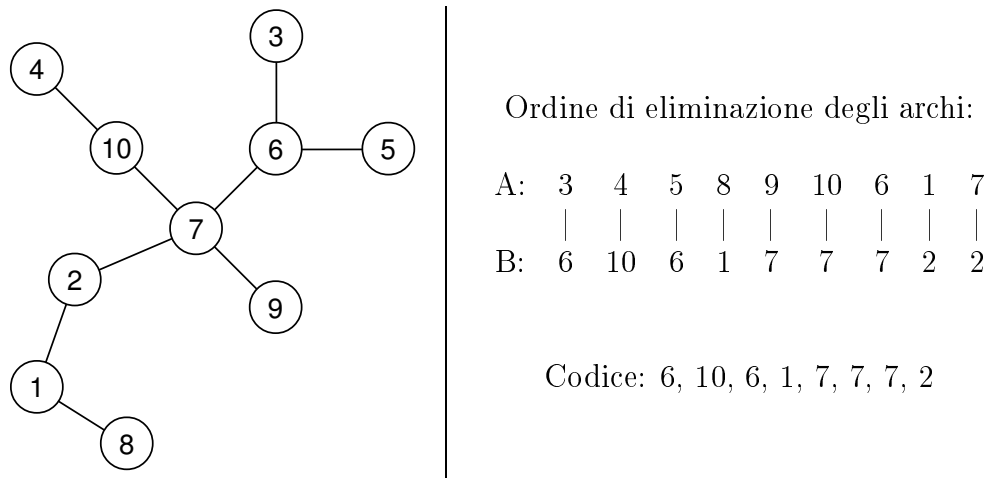


Figura 2.4: Esempio di codice Stack-Queue.

Occupiamoci ora di come ricondurre tale codifica allo schema generale, rimandando al successivo capitolo la presentazione di algoritmi ottimi di codifica e di decodifica.

### 2.6.1 Algoritmo di codifica

Dovendo eliminare i nodi interni nell'ordine in cui diventano foglie è necessario mantenere tale informazione tramite un vettore ausiliario `ord`: se al passo  $i$ -esimo il nodo  $v$  diventa foglia si assegna a `ord[v]` il valore  $i$ . Tutte le foglie dell'albero iniziale avranno `ord` pari a 0.

La funzione di scelta si limiterà quindi a selezionare la foglia di `ord` minimo e, qualora ci siano ancora due o più foglie di `ord` 0, quella di etichetta minima.

## ALGORITMO CODIFICA STACK-QUEUE

**INPUT:** Un albero  $T = (V, E)$  non radicato etichettato su  $[1..n]$  rappresentato tramite liste di adiacenza.

**OUTPUT:** Un vettore `code` di  $n - 1$  interi in  $[1..n]$  rappresentante il codice Stack-Queue di  $T$ .

1. FOR  $v = 1$  TO  $n$  DO
2.     IF  $(v \in F(T))$  THEN
3.          $\text{ord}[v] = 0$
4.     ELSE
5.          $\text{ord}[v] = \infty$
6. FOR  $i = 1$  TO  $n - 1$  DO
7.     IF  $(\exists v \in F(T) : \text{ord}[v] = 0)$  THEN
8.          $u = \min\{v \in F(T) : \text{ord}[v] = 0\}$
9.     ELSE
10.          $u \in F(T) : \text{ord}[u] = \min\{\text{ord}[v] : v \in F(T)\}$
11.      $\text{code}[i] = v : (u, v) \in E(T)$
12.      $T = T - \{u\}$
13.     IF  $(v \in F(T))$  THEN  $\text{ord}[v] = i$

Come già accennato in precedenza si può abbreviare il codice omettendo l'ultimo elemento: il teorema che segue dà ragione di tale affermazione.

**Teorema 2.6.1.**  $\text{code}[n - 1] = \text{code}[n - 2]$  *sempre*.

*Dimostrazione.* Siano  $u$  e  $v$  i due nodi rimasti nell'albero all' $(n - 1)$ -esimo passo e supponiamo, senza perdita di generalità, che  $\text{code}[n - 1] = v$ , ciò implica che  $u$  è diventato foglia prima di  $v$ :  $\text{ord}[u] < \text{ord}[v]$ . Al passo

$(n - 2)$ -esimo, quindi, il nodo eliminato era di certo adiacente a  $v$  pertanto  $\text{code}[n - 2] = v = \text{code}[n - 1]$ .  $\square$

È quindi sufficiente eseguire il ciclo di riga 6 da 1 a  $n - 2$ .

Concludiamo determinando quale nodo debba essere usato come radice, anche se nell'algoritmo proposto la necessità di radicare l'albero di fatto decade. L'ordine di eliminazione dei nodi che questo codice realizza può essere così reinterpretato: prima vengono tolte tutte le foglie, poi tutti i nodi divenuti foglie a seguito delle eliminazioni precedenti, e così via. La radice sarà quindi uno dei centri (Definizione 1.2.2).

Se l'albero ha due centri questi saranno i nodi di livello massimo, quindi dopo aver eliminato gli altri  $n - 2$  nodi soltanto i due centri saranno rimasti; ma avendo il codice lunghezza  $n - 2$  nessuno dei due verrà eliminato, quindi è di fatto irrilevante quale dei due centri sia stato scelto come radice.

Per questo codice la radice è dunque semplicemente uno dei centri.

### 2.6.2 Algoritmo di decodifica

Al fine di poter applicare la funzione di scelta  $f$  è necessario dedurre dal codice quale sia il passo in cui ciascun nodo interno diventa foglia. È possibile ricostruire il vettore `ord` notando che, quando l'algoritmo di codifica assegna  $\text{ord}[v] = i$ , si ha  $\text{code}[i] = v$ , inoltre,  $v$  non potrà comparire più avanti nel codice. Quindi  $\text{ord}[v]$  è l'indice dell'ultima apparizione di  $v$  nel codice o 0 se  $v$  non compare, cioè se era una foglia dell'albero iniziale.

## ALGORITMO DECODIFICA STACK-QUEUE

**INPUT:** Un vettore  $\text{code} = (b_1, \dots, b_{n-2})$  di interi in  $[1..n]$ .

**OUTPUT:** L'albero  $T = (V, E)$  etichettato su  $[1..n]$  corrispondente a  $\text{code}$ .

1.  $n = \text{code.length} + 2$
2.  $b_{n-1} = b_{n-2}$
3. FOR  $v = 1$  TO  $n$  DO  $\text{ord}[v] = 0$
4. FOR  $i = n - 1$  DOWNTO 1 DO IF  $(\text{ord}[b_i] = 0)$   $\text{ord}[b_i] = i$
5. FOR  $i = 1$  TO  $n - 1$  DO
6.      $F_i = \{v \in [1..n] \setminus \{a_j : 1 \leq j < i\} \setminus \{b_j : i \leq j < n\}\}$
7.     IF  $(\exists v \in F_i : \text{ord}[v] = 0)$  THEN
8.          $a_i = \min\{v \in F_i : \text{ord}[v] = 0\}$
9.     ELSE
10.          $a_i \in F_i : \text{ord}[a_i] = \min\{\text{ord}[v] : v \in F_i\}$
11.  $T = ([1..n], \{(a_i, b_i) : 1 \leq i < n\})$

## 2.7 Conclusioni

A conclusione del capitolo si riportano, nella Tabella 2.1, le informazioni salienti dello schema di codifica e di ciascun codice presentato in questo capitolo: il nome, l'anno in cui è stato introdotto, se l'autore lo ha definito per alberi radicati o non radicati, la lunghezza della stringa generata ed il riferimento bibliografico.

È inoltre interessante riportare la funzione  $f$  e la relazione d'ordine che permettono di ricondurre ciascun codice allo schema generale di codifica (Tabella 2.2), per i codici definiti per alberi non radicati viene inoltre riportato quale sia il nodo nel quale radicare.

Nome	Anno	Per alberi	Lunghezza	Riferimento
Prüfer	1918	non radicati	$n - 2$	[26]
Neville II	1953	non radicati	$n - 1$	[25]
Neville III	1953	non radicati	$n - 2$	[25]
Moon	1970	radicati	$n - 1$	[24]
Stack-Queue	2002	non radicati	$n - 2$	[12]
Schema generale	2003	radicati	$n - 1$	Questa tesi

Tabella 2.1: Riepilogo delle caratteristiche dei codici.

Codice	$f(F_i)$	$\prec$	Radice
Prüfer	$\{\min\{v \in F_i\}\}$	-	$n$
Neville II	$F_i$	$\leq$	il maggiore dei centri
Neville III	$\begin{cases} \{b_{i-1}\} & \text{se } b_{i-1} \text{ è foglia} \\ \{\min\{F_i\}\} & \text{altrimenti} \end{cases}$	-	la massima foglia
Moon	come in uno dei codici precedenti		-
Stack-Queue	$\{\min\{u \in F_i : \text{ord}[u] = \min\{\text{ord}[v] : v \in F_i\}\}\}$	-	uno dei centri

Tabella 2.2: Funzioni di scelta e relazioni d'ordine dei codici.

# Capitolo 3

## Algoritmi sequenziali ottimi

Nel presente capitolo saranno valutati i costi computazionali degli algoritmi proposti nel Capitolo 2, mostrando come, nella maggior parte dei casi, una diretta implementazione delle definizioni dei codici non consente di ottenere algoritmi efficienti.

Analizzando più in dettaglio i codici e l'ordine di eliminazione delle foglie da essi realizzato, si riusciranno ad ottenere, per ciascuno di essi, gli algoritmi di codifica e decodifica ottimi.

### 3.1 Codici di Prüfer

#### 3.1.1 Algoritmo di codifica

Nell'implementazione proposta nel capitolo precedente la ricerca della foglia di etichetta minima, ad ogni passo del ciclo, richiede la scansione di tutti i nodi dell'albero, cioè  $O(n)$  tempo. Per diminuire la complessità temporale si può pensare di mantenere un'opportuna struttura dati per la lista delle foglie, il che permetterebbe di ridurre il tempo necessario per la ricerca del

minimo a  $O(\log n)$ . In definitiva tale algoritmo richiede  $O(n \log n)$  tempo.

È invece possibile ottenere il codice di Prüfer di un albero in tempo  $O(n)$  [16, 8], l'approccio qui riportato segue quello proposto da Chen e Wang in [8] che, anche se più complesso, ha il vantaggio di poter essere adattato per ottenere l'algoritmo parallelo ottimo proposto nel Capitolo 4. Al medesimo scopo anche la codifica dei restanti codici verrà trattata in maniera analoga.

Osserviamo cosa avviene durante l'esecuzione dell'algoritmo riprendendo l'esempio di Figura 2.1 proposto nel capitolo precedente.

È evidente che prima che un nodo  $v$  venga selezionato è necessario che tutto il suo sottoalbero sia già stato eliminato. Inoltre se il nodo di etichetta massima nel sottoalbero di  $v$  è  $u \neq v$ , nel momento in cui  $u$  viene selezionato, di tutto il sottoalbero di  $v$  non sarà rimasto altro che una catena da  $u$  a  $v$ . Tale catena sarà composta da tutti nodi minori di  $u$  e che quindi saranno selezionati nei passi subito successivi alla selezione di  $u$ .

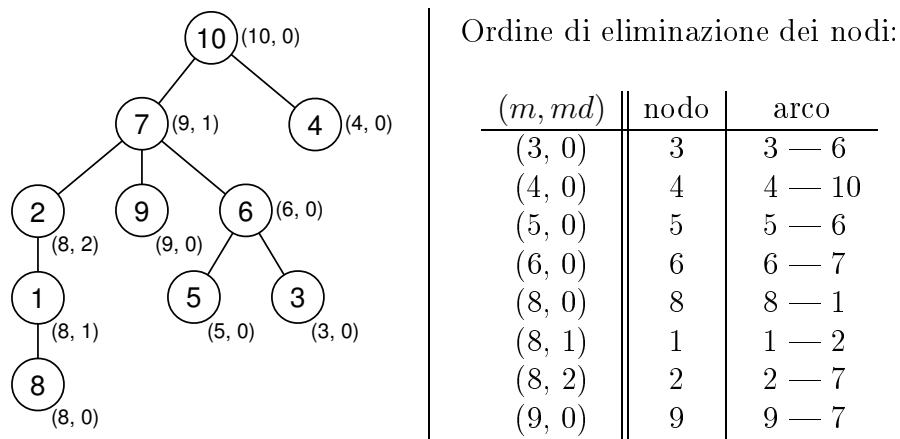


Figura 3.1: Esempio di ordinamento dei nodi per il codice di Prüfer.

Nella Figura 3.1 l'albero dell'esempio del capitolo precedente è stato riportato segnando, vicino ad ogni nodo  $v$ , il nodo di etichetta massima del suo

sottoalbero (denominato con  $m(v)$ ) e la distanza tra  $v$  e  $m(v)$  (denominata con  $dm(v)$ ); risulta evidente che l'ordine di eliminazione dei nodi rispetta l'ordine lessicografico della coppia  $(m(v), dm(v))$ .

Introduciamo infine le liste  $c_j$  per  $j$  da 1 a  $n$ , in ciascuna di esse compaiono tutti i nodi  $v$  per i quali  $m(v) = j$  in ordine crescente rispetto a  $dm(v)$ . Intuitivamente ciascuna di queste liste rappresenta la catena che viene eliminata quando  $j$  diviene la foglia di etichetta minima. Tornando all'esempio precedente si ricavano le liste:

$$\begin{aligned}
 c_3 &= 3 \rightarrow \backslash \\
 c_4 &= 4 \rightarrow \backslash \\
 c_5 &= 5 \rightarrow \backslash \\
 c_6 &= 6 \rightarrow \backslash \\
 c_8 &= 8 \rightarrow 1 \rightarrow 2 \rightarrow \backslash \\
 c_9 &= 9 \rightarrow 7 \rightarrow \backslash \\
 c_{10} &= 10 \rightarrow \backslash
 \end{aligned}$$

Si assumerà di disporre di liste FIFO in grado di realizzare in tempo costante le operazioni di inserimento in coda, estrazione dalla testa e controllo di lista vuota. Chiarita l'idea è ora possibile presentare l'algoritmo:

### ALGORITMO CODIFICA PRÜFER OTTIMO

**INPUT:** Un albero  $T = (V, E)$  non radicato etichettato su  $[1..n]$  rappresentato tramite liste di adiacenza.

**OUTPUT:** Un vettore `pcode` di  $n - 2$  interi in  $[1..n]$  rappresentante il codice di Prüfer di  $T$ .

1. radica  $T$  in  $n$  ottenendo,  $\forall v$ ,  $p(v)$  e  $children(v)$
2. Con una visita in post-order calcola,  $\forall v$ ,  $m(v)$  e  $dm(v)$
3. Costruisci le liste  $c_j$
4. `pcode = ()`

5.  $j = 0$
6. WHILE (`pcode.length < n - 2`) DO
7.      $j++$
8.     WHILE (NOT `cj.empty`) DO
9.          $v = c_j.get()$
10.         appendi  $p(v)$  a `pcode`

I passi 1 e 2 possono essere evidentemente eseguiti in tempo  $O(n)$ . Per costruire le liste  $c_j$  è sufficiente che nel passo 2, quando  $m(v)$  viene impostato a  $j$  si aggiunga  $v$  in coda a  $c_j$ .

Dal momento che nella totalità delle liste compaiono esattamente  $n$  elementi, le istruzioni delle righe 9 e 10 verranno eseguite  $O(n)$  volte: in definitiva l'algoritmo risulta lineare e pertanto ottimo.

### 3.1.2 Algoritmo di decodifica

Anche per l'algoritmo di decodifica proposto nel Capitolo 2, è l'operazione di ricerca del minimo a determinare un costo computazionale non inferiore ad  $O(n \log n)$ , ed anche in questo caso è possibile risolvere il problema in tempo lineare [16].

Si osservi anzitutto che, ricordando la Proposizione 2.1.1, è possibile ottenere il grado di ciascun nodo nell'albero codificato semplicemente scorrendo il vettore `pcode`, ottenendo così il vettore `degree`. Essendo tale passo necessario tanto per questo quanto per i seguenti algoritmi di decodifica, se ne riporta separatamente l'implementazione:

## FUNZIONE CALCOLO DEL GRADO

**INPUT:** Un vettore `code` di  $n - 1$  interi in  $[1..n]$ .

**OUTPUT:** Il vettore `degree` contenente il grado di ciascun nodo nell'albero corrispondente a `code`

```
1. CalculateDegree(code) {
2.     FOR  $v = 1$  TO  $n - 1$  DO
3.         degree[v] = 1
4.     degree[code[n - 1]] = 0
5.     FOR  $i = 1$  TO  $n - 1$  DO
6.         degree[code[i]]++
7. }
```

Per sapere quale sia la foglia di etichetta minima è sufficiente scorrere il vettore `degree` dall'inizio fino a che non si incontra un 1. La foglia trovata sarà  $a_1$  e dovrà essere connessa al nodo  $b_1 = \text{pcode}[1]$ . Se  $b_1$  non aveva figli oltre ad  $a_1$  e se  $b_1 < a_1$  allora al secondo passo l'algoritmo di codifica ha sicuramente eliminato  $b_1$  quindi  $a_2 = b_1$ , altrimenti si deve cercare una nuova foglia di etichetta minima, ma certamente non minore di  $a_1$ , continuando a scorrere il vettore `degree`. Generalizzando quanto detto si ottiene l'algoritmo.

## ALGORITMO DECODIFICA PRÜFER OTTIMO

**INPUT:** Un vettore `pcode` di  $n - 2$  interi in  $[1..n]$ .

**OUTPUT:** L'albero  $T = (V, E)$  etichettato su  $[1..n]$  corrispondente a `pcode`.

```
1.  $n = \text{pcode.length} + 2$ 
2.  $\text{pcode}[n - 1] = n$ 
```

```

3. CalculateDegree(pcode)
4.  $T = ([1..n], \emptyset)$ 
5.  $a_1 = 0$ 
6.  $leafIndex = 1$ 
7. FOR  $i = 1$  TO  $n - 1$  DO
8.     IF ( $a_i = 0$ ) THEN
9.         WHILE ( $degree[leafIndex] \neq 1$ ) DO
10.             $leafIndex++$ 
11.             $a_i = leafIndex$ 
12.             $E(T) = E(T) \cup \{(a_i, pcode[i])\}$ 
13.             $degree[a_i]--$ 
14.             $degree[pcode[i]]--$ 
15.            IF ( $degree[pcode[i]] = 1$ ) AND ( $pcode[i] < a_i$ ) THEN
16.                 $a_{i+1} = pcode[i]$ 
17.            ELSE
18.                 $a_{i+1} = 0$ 

```

Se al passo  $i$ -esimo, inserendo l'arco  $(a_i, pcode[i])$ , il grado di  $pcode[i]$  si riduce ad 1 allora ciò significa che al passo  $i$ -esimo della codifica tale nodo era divenuto una foglia; se inoltre la sua etichetta era inferiore a quella di  $a_i$  allora sicuramente sarà stato eliminato al passo  $(i + 1)$ -esimo. In forza di questa osservazione l'istruzione condizionale di riga 15 decide se scegliere  $pcode[i]$  come foglia per il passo successivo o se far sì che, con il ciclo di riga 9, si cerchi una nuova foglia di etichetta minima. Si osservi, inoltre, che

durante tutta l'esecuzione dell'algoritmo il ciclo di riga 9 scorre il vettore *degree* una sola volta.

Dal momento che tutti gli altri cicli contano  $n - 1$  iterazioni, si può affermare che l'algoritmo è lineare e pertanto ottimo.

## 3.2 Secondo codice di Neville

### 3.2.1 Algoritmo di codifica

La complessità dell'algoritmo proposto nel Capitolo 2 è pari alla somma dei tempi impiegati da ogni iterazione del ciclo. L' $i$ -esima iterazione svolge, come compito più oneroso, l'ordinamento di  $k_i$  elementi, quindi richiede  $O(k_i \log k_i)$  tempo. Sommando i contributi di ciascuna delle  $m$  iterazioni si ottiene:

$$\sum_{i=1}^m O(k_i \log k_i) = O\left(\sum_{i=1}^m k_i \log k_i\right) \leq O\left(n \log(\max_{i \in [1..m]} \{k_i\})\right)$$

Nel caso peggiore, quello in cui  $\max\{k_i\} = O(n)$ , l'algoritmo impiega quindi  $O(n \log n)$  tempo.

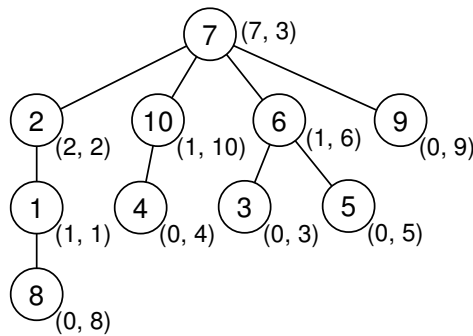
Anche in questo caso si può ottenere un algoritmo lineare ricorrendo ad un ordinamento dei nodi dell'albero. Introduciamo il concetto di livello di un nodo in un albero radicato:

$$l(v) = \begin{cases} 1 & \text{se } v \text{ è una foglia} \\ 1 + \max\{l(x) \mid x \in \text{children}(v)\} & \text{altrimenti} \end{cases}$$

dove  $\text{children}(v)$  è l'insieme dei figli del nodo  $v$ .

Ad ogni iterazione il codice elimina dall'albero tutte le foglie, quindi al primo passo saranno eliminati tutti i nodi di livello 1, al secondo quelli di livello 2 e così via. L'ordine di eliminazione risulta quindi essere equivalente all'ordinamento lessicografico dei nodi rispetto alla coppia  $(l(v), v)$ . La

Figura 3.2 mostra i valori di tale coppia per l'albero dell'esempio di Figura 2.2.



Ordine di eliminazione dei nodi:

$(l, v)$	nodo	arco
(0, 3)	3	3 — 6
(0, 4)	4	4 — 10
(0, 5)	5	5 — 6
(0, 8)	8	8 — 1
(0, 9)	9	9 — 7
(1, 1)	1	1 — 2
(1, 6)	6	6 — 7
(1, 10)	10	10 — 7
(2, 2)	2	2 — 7

Figura 3.2: Esempio di ordinamento dei nodi per il secondo codice di Neville.

Da quanto mostrato deriva il seguente algoritmo, nel quale con  $L_i$  si indica la lista dei nodi di livello  $i$  in ordine di etichetta crescente:

### ALGORITMO CODIFICA NEVILLE II OTTIMO

**INPUT:** Un albero  $T = (V, E)$  non radicato etichettato su  $[1..n]$  rappresentato tramite liste di adiacenza.

**OUTPUT:** Un vettore `code` di  $n - 1$  interi in  $[1..n]$  rappresentante il secondo codice di Neville di  $T$ .

1.  $r =$  cerca il centro di  $T$  di etichetta maggiore
2. radica  $T$  in  $r$  ottenendo,  $\forall v$ ,  $p(v)$  e  $children(v)$
3. Con una visita in post-order calcola,  $\forall v$ ,  $l(v)$
4. FOR  $v = 1$  TO  $n$  DO
5.      $L_{l(v)}.put(v)$
6.  $maxl = \max\{l(v) : v \in V(T)\}$
7. `code` = ()

```

8. FOR  $l = 1$  TO  $maxl - 1$  DO
9.     WHILE (NOT  $L_l.empty$ ) DO
10.          $v = L_l.get()$ 
11.         appendi  $p(v)$  a code

```

Il Teorema 1.2.3 assicura che la ricerca del centro dell'albero di etichetta maggiore può essere realizzata in tempo lineare.

Il ciclo di riga 8 va da 1 a  $maxl - 1$  perché soltanto la radice ha livello pari a  $maxl$  e questa non sarà mai selezionata come foglia da eliminare. Complessivamente le prime  $maxl - 1$  liste conterranno esattamente  $n - 1$  elementi quindi le istruzioni delle righe 10 e 11 verranno eseguite  $O(n)$  volte, l'algoritmo risulta quindi lineare e pertanto ottimo.

### 3.2.2 Algoritmo di decodifica

L'algoritmo di decodifica proposto nel Capitolo precedente ha complessità analoga a quella dell'algoritmo di codifica per via dell'ordinamento presente all'interno di ogni iterazione del ciclo.

Per ottenere un algoritmo lineare è necessario ricreare le liste ordinate  $L_i$  usate nell'algoritmo di codifica, si deve quindi calcolare, per ogni  $v$ ,  $l(v)$ .

Dopo aver calcolato il vettore `degree` come fatto per l'algoritmo di decodifica ottimo dei codici di Prüfer, si può subito sapere quali siano le foglie dell'albero iniziale, ovvero i nodi di livello 1, sia  $n_1$  il loro numero.

I nodi di livello 2 vanno ricercati tra quelli connessi a quelli di livello 1: le  $n_1$  foglie saranno connesse ai primi  $n_1$  nodi elencati nel codice. È sufficiente quindi decrementare di 1 il grado dei primi  $n_1$  nodi elencati nel codice, tutti

quelli il cui grado raggiunge il valore 1 saranno le nuove foglie, quindi i nodi di livello 2.

Generalizzando tale processo per i passi successivi si ottiene l'algoritmo:

### ALGORITMO DECODIFICA NEVILLE II OTTIMO

**INPUT:** Un vettore `code` di  $n - 1$  interi in  $[1..n]$ .

**OUTPUT:** L'albero  $T = (V, E)$  etichettato su  $[1..n]$  corrispondente a `code`.

1.  $n = \text{code.length} + 1$
2. `CalculateDegree(code)`
3.  $n_1 = 0$
4. FOR  $v = 1$  TO  $n$  DO
5.     IF (`degree[v] = 1`) THEN
6.          $l(v) = 1$
7.          $n_1++$
8.  $vcount = n_1$
9.  $l = 1$
10.  $i = 0$
11. WHILE ( $vcount < n$ ) DO
12.     FOR  $j = 1$  TO  $n_l$  DO
13.          $i++$
14.         `degree[code[i]]--`
15.         IF (`degree[code[i]] = 1`) THEN
16.              $l(\text{code}[i]) = l + 1$
17.              $n_{l+1}++$
18.      $vcount = vcount + n_{l+1}$

```

19.      $l++$ 
20. FOR  $v = 1$  TO  $n$  DO
21.      $L_{l(v)}.put(v)$ 
22.  $T = ([1..n], \emptyset)$ 
23.  $i = 0$ 
24.  $maxl = \max\{l(v) : v \in V(T)\}$ 
25. FOR  $l = 1$  TO  $maxl - 1$  DO
26.     WHILE (NOT  $L_l.empty$ ) DO
27.          $i++$ 
28.          $a_i = L_l.get()$ 
29.          $E(T) = E(T) \cup \{(a_i, code[i])\}$ 

```

Il ciclo di riga 11 viene eseguito fintantoché non è noto il livello di ciascun nodo, e il suo ciclo interno scorre, durante tutto l'algoritmo, l'intero vettore del codice, quindi in totale un costo di  $O(n)$ .

Il ciclo di riga 20 costruisce le liste ordinate  $L_i$  e quelli delle righe 25 e 26 aggiungono all'albero i suoi  $n - 1$  archi, tutti eseguono  $O(n)$  iterazioni, in definitiva l'algoritmo risulta lineare e pertanto ottimo.

## 3.3 Terzo codice di Neville

### 3.3.1 Algoritmo di codifica

L'unica operazione onerosa necessaria all'algoritmo proposto nel Capitolo 2 è la ricerca del minimo tra le foglie dell'albero, come già osservato però tale ricerca viene in realtà effettuata soltanto tra le foglie dell'albero iniziale:

quindi è possibile creare una lista ordinata di tali foglie prima di iniziare il ciclo. In tal modo l'algoritmo può essere reso lineare.

Tuttavia un approccio così semplice nel seriale non si presta ad essere parallelizzato in maniera efficiente, vedremo quindi come ottenere un algoritmo ottimo per questo codice con la tecnica dell'ordinamento lessicografico di coppie usata per i due codici precedenti.

Nel caso dei codici di Prüfer ogni nodo  $v$  veniva collocato nella catena che partiva dal nodo di etichetta massima nel suo sottoalbero  $m(v)$ ; per questo codice invece le catene partono sempre dalle foglie, si dovrà quindi calcolare, per ogni nodo  $v$ , la massima foglia nel suo sottoalbero. Riportiamo in Figura 3.3 l'esempio del capitolo precedente segnando, vicino a ciascun nodo  $v$ , i valori di  $f(v)$ , pari all'etichetta della massima foglia nel suo sottoalbero, e  $df(v)$  ovvero la distanza tra  $v$  e  $f(v)$ .

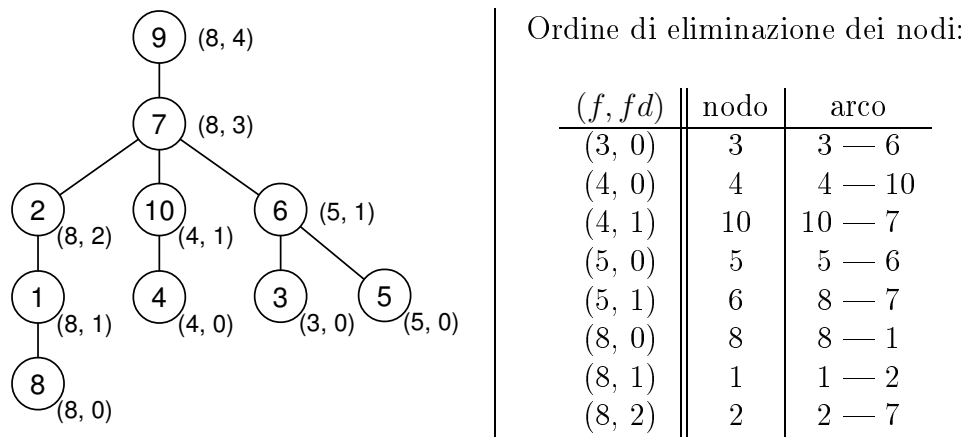


Figura 3.3: Esempio di ordinamento dei nodi per il terzo codice di Neville.

Non è necessario riportare esplicitamente il codice dell'algoritmo ottimo da momento che risulta identico a quello descritto per i codici di Prüfer, con due minime varianti: l'albero viene radicato nella foglia di etichetta massima

e il valore calcolato dalla visita in post-order ora è la massima foglia del sottoalbero di  $v$ .

### 3.3.2 Algoritmo di decodifica

Analogamente a quanto detto per l'algoritmo di codifica, anche in questo caso ordinando a priori le foglie dell'albero finale si può ottenere un algoritmo lineare. Per individuare le foglie si può far ricorso al calcolo del grado di ciascun nodo.

#### ALGORITMO DECODIFICA NEVILLE III OTTIMO

**INPUT:** Un vettore `code` di  $n - 2$  interi in  $[1..n]$ .

**OUTPUT:** L'albero  $T = (V, E)$  etichettato su  $[1..n]$  corrispondente a `code`.

1.  $n = \text{code.length} + 2$
2.  $\text{code}[n - 1] = \max_{v \in [1..n]} \{v : \nexists i \text{ code}[i] = v\}$
3. `CalculateDegree(code)`
4. FOR  $v = 1$  TO  $n$  DO
5.     IF (`degree[v] = 1`) THEN
6.         `L.put(v)`
7.  $T = ([1..n], \emptyset)$
8.  $i = 0$
9.  $a_1 = 0$
10. FOR  $i = 1$  TO  $n - 1$  DO
11.     IF ( $a_i = 0$ ) THEN
12.          $a_i = L.get()$
13.      $E(T) = E(T) \cup \{(a_i, \text{code}[i])\}$

```

14.   degree[ai]-
15.   degree[code[i]]-
16.   IF (degree[code[i]]= 1) THEN
17.       ai+1 = code[i]
18.   ELSE
19.       ai+1 = 0

```

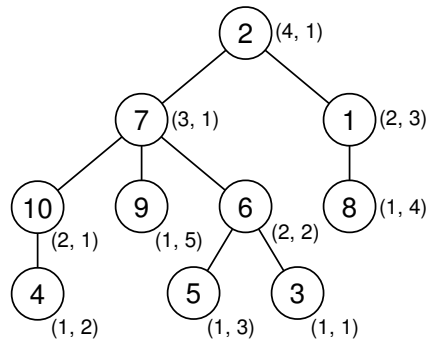
### 3.4 Codici Stack-Queue

Come già detto nel capitolo precedente gli autori di tali codici li hanno proposti congiuntamente a degli algoritmi di codifica e decodifica ottimi [12], mentre quelli presentati nei Capitoli 2.6.1 e 2.6.2 richiedono entrambi  $O(n \log n)$  a causa della ricerca di minimo all'interno del ciclo.

#### 3.4.1 Algoritmo di codifica

Come per il codice precedente anche in questo caso si è scelto di presentare un algoritmo che proceda per ordinamento lessicografico di coppie, nonostante l'approccio proposto dagli autori in [12] nel seriale risulti più semplice. Diversamente dal codice precedente però, non si partirà dall'algoritmo proposto per i codici di Prüfer bensì da quello del secondo codice di Neville.

La sostanziale differenza tra i due codici è che in questo caso i nodi di uno stesso livello, ad eccezione di quelli di livello 1, non devono essere ordinati per etichette crescenti ma secondo l'ordine in cui diventano foglie. Riportiamo in Figura 3.4 l'esempio del capitolo precedente segnando vicino a ciascun nodo il livello e il numero d'ordine all'interno del livello.



Ordine di eliminazione dei nodi:

$(l, ord)$	nodo	arco
(1, 1)	3	3 — 6
(1, 2)	4	4 — 10
(1, 3)	5	5 — 6
(1, 4)	8	8 — 1
(1, 5)	9	9 — 7
(2, 1)	10	10 — 7
(2, 2)	6	6 — 7
(2, 3)	1	1 — 2

Figura 3.4: Esempio di ordinamento dei nodi per il codice Stack-Queue.

Da quanto detto si ottiene il seguente algoritmo, nel quale con  $L_i$  si indica la lista dei nodi di livello  $i$  nell'ordine in cui sono divenuti foglie:

### ALGORITMO CODIFICA STACK-QUEUE OTTIMO

**INPUT:** Un albero  $T = (V, E)$  non radicato etichettato su  $[1..n]$  rappresentato tramite liste di adiacenza.

**OUTPUT:** Un vettore `code` di  $n - 2$  interi in  $[1..n]$  rappresentante il codice Stack-Queue di  $T$ .

1. FOR  $v = 1$  TO  $n$  DO
2.     IF ( $v$  è foglia) THEN
3.          $L_1.put(v)$
4.  $l = 1$
5. `code = ()`
6. WHILE (`code.length` <  $n - 2$ ) DO
7.     IF ( $L_l.empty$ ) THEN
8.          $l++$
9.      $v = L_l.get()$
10.     sia  $u$  l'unico nodo adiacente a  $v$

11.      appendi  $v$  a `code`
12.      rimuovi da  $T$  in nodo  $u$  e l'arco  $(u, v)$
13.      IF( $u$  è foglia) THEN
14.              $L_{l+1}.\text{put}(v)$

L'algoritmo risulta piuttosto semplice in forza del fatto che il ciclo necessario alla creazione delle liste  $L_{l+1}$  a partire da  $L_l$ , può essere utilizzato per generare direttamente il codice; questo perché l'ordine di eliminazione degli elementi è esattamente quello di creazione della lista.

Tutto l'algoritmo è chiaramente lineare e quindi ottimo.

### 3.4.2 Algoritmo di decodifica

L'ultima apparizione di un nodo nel codice corrisponde al passo in cui è divenuto foglia; quindi, scorrendo il vettore `code` dalla fine all'inizio, è possibile ricostruire l'ordine in cui i nodi interni dell'albero sono divenuti foglie semplicemente memorizzando ogni nodo in una pila LIFO  $S$  la prima volta che lo si incontra.

Tutti i nodi che non compaiono nel codice sono le foglie dell'albero iniziale e quindi saranno state eliminate all'inizio della codifica in ordine crescente, vanno quindi impilate in ordine inverso.

#### ALGORITMO DECODIFICA STACK-QUEUE OTTIMO

**INPUT:** Un vettore `code` di  $n - 2$  interi in  $[1..n]$ .

**OUTPUT:** L'albero  $T = (V, E)$  etichettato su  $[1..n]$  corrispondente a `code`.

1.  $n = \text{code.length} + 2$

```

2. code[n - i] = code[n - 2]
3. FOR v = 1 TO n DO
4.     used[v] = false
5. FOR i = n - 1 DOWNT0 1 DO
6.     IF (NOT used[code[i]]) THEN
7.         used[v] = true
8.         S.push(code[i])
9. FOR v = n DOWNT0 1 DO
10.    IF (NOT used[v]) THEN
11.        S.push(v)
12. T = ([1..n], ∅)
13. FOR i = 1 TO n - 1 DO
14.    ai = S.pop()
15.    E(T) = E(T) ∪ {(ai, code[i])}

```

### 3.5 Conclusioni

Riepilogando, gli algoritmi di codifica e decodifica dei vari codici proposti nel capitolo precedente richiedono tutti  $O(n \log n)$  tempo, ad eccezione di quelli per il terzo codice di Neville il cui costo si riduce ad  $O(n)$  semplicemente ordinando a priori le foglie dell'albero iniziale.

Per tutti i codici è possibile ricondurre il problema della codifica ad un ordinamento di coppie, ottenendo in tutti i casi algoritmi dal costo pari ad  $O(n)$  utilizzando delle liste FIFO. Per ottenere medesime prestazioni per la decodifica dei codici ci si può basare sul calcolo del grado, fa eccezione il

codice Stack-Queue il cui algoritmo di decodifica impiega una pila LIFO. Nella Tabella 3.1 sono riportate, per ciascun codice, le definizioni della coppia utilizzata per la codifica e il metodo di decodifica.

Codice	Coppia di codifica	Metodo di decodifica
Prüfer	$(m(v), dm(v))$	Calcolo del grado
Neville II	$(l(v), v)$	Calcolo del grado
Neville III	$(f(v), df(v))$	Calcolo del grado
Stack-Queue	$(l(v), ord(v))$	Pila LIFO

Tabella 3.1: Caratteristiche degli algoritmi di codifica e decodifica dei codici.

# Capitolo 4

## Algoritmi paralleli

In questo capitolo si studierà la possibilità di codificare e decodificare, su calcolatori paralleli, tutti i codici visti nel capitolo precedente. Il modello di macchina per cui gli algoritmi saranno proposti è la PRAM che costituisce una valida astrazione teorica delle reali macchine parallele. In particolare ci si riferirà alla PRAM EREW, tale scelta è dovuta al fatto che questo è il più debole dei modelli comunemente trattati in letteratura.

Verranno proposti dapprima tutti gli algoritmi di codifica, ottenuti adattando quelli presentati nel Capitolo 3. In seguito ci si occuperà della decodifica ma l'approccio utilizzato nel precedente capitolo, sostanzialmente basato sul calcolo del grado di ogni nodo a partire dal codice, sarà sostituito da tecniche che sono fondate sulla determinazione dell'ultima occorrenza di ciascun nodo nel codice.

### 4.1 Algoritmi di codifica

Negli algoritmi che saranno di seguito proposti si assumerà che l'albero in input sia rappresentato tramite una struttura dati idonea al calcolo ef-

ficiente del Tour di Eulero; si veda a tale riguardo la discussione proposta nel paragrafo 1.3.1.

La risoluzione del problema della codifica dei codici di Prüfer nel parallelo si deve a Greenlaw, Halldórsson e Petreschi che in [16] raffinano l'algoritmo presentato da Greenlaw e Petreschi in [17] eliminando il collo di bottiglia rappresentato dall'ordinamento. Con alcune semplici modifiche sarà possibile ottenere da questo un algoritmo ottimo per la codifica del terzo codice di Neville.

Per ciò che riguarda i restanti codici, Deo e Micikevicius [11] hanno proposto degli algoritmi efficienti, ma non ottimi, per ottenere il secondo codice di Neville ed il codice Stack-Queue di un generico albero.

#### 4.1.1 Codici di Prüfer

Dopo aver radicato l'albero, l'algoritmo procede calcolando la massima etichetta nel sottoalbero di ciascun nodo  $i$ ,  $\text{maxnode}[i]$ . Vengono poi create le catene collegando un nodo al padre se condivide con esso il valore di  $\text{maxnode}$ . In fine viene determinato l'ordine di eliminazione di ciascun nodo  $i$ , sommando la sua posizione nella catena con la dimensione delle catene che verranno eliminate prima di quella cui  $i$  appartiene.

#### ALGORITMO CODIFICA PRÜFER PARALLELO OTTIMO

**INPUT:** Un albero  $T = (V, E)$  non radicato etichettato su  $[1..n]$ .

**OUTPUT:** Il vettore `code` di  $n - 2$  interi in  $[1..n]$  rappresentante il codice di Prüfer di  $T$ .

1. *Radica l'albero in  $n$* 
  - 1.1 calcola il Tour di Eulero di  $T$
  - 1.2 radica  $T$  in  $n$  ottenendo,  $\forall i$ , `parent[i]`
2. *Calcola la massima etichetta dei nodi nel sottoalbero di ogni nodo*
  - 2.1 calcola l'albero regolare  $T_R$  corrispondente a  $T$
  - 2.2 per ogni nodo  $i$  calcola `maxnode[i]`, la massima etichetta dei nodi nel sottoalbero di  $T$  radicato in  $i$ , utilizzando il Rake su  $T_R$
3. *Ogni nodo legge il **maxnode** del padre*
  - 3.1 FOR  $i = 1$  TO  $n$  PARDO
  - 3.2     `parentmaxnode[i] = maxnode[parent[i]]`
4. *Crea le catene*
  - 4.1 FOR  $i = 1$  TO  $n$  PARDO
  - 4.2     IF (`parentmaxnode[i] = maxnode[i]`) THEN
  - 4.3         `chain[i] = parent[i]`
  - 4.4     ELSE
  - 4.5         `chain[i] = NULL`
  - 4.6 esegui il List Ranking su `chain[]` ottenendo, per ogni nodo  $i$ , il valore `positionInChain[i]`
5. *Calcola il numero di elementi presenti nelle catene precedenti*
  - 5.1 *Calcola la dimensione di ogni catena, ovvero il rango dell'ultimo nodo*
    - 5.1.1 FOR  $i = 1$  TO  $n$  PARDO
    - 5.1.2     IF (`chain[i] = NULL`) THEN
    - 5.1.3         `chainsize[i] = positionInChain[i]`
    - 5.1.4 propaga in Broadcast la dimensione di ogni catena a tutti i suoi elementi

5.2 *Calcola, per la testa di ogni catena, la somma delle dimensioni delle catene precedenti*

5.2.1 FOR  $i = 1$  TO  $n$  PARDO

5.2.2 IF ( $\text{positionInChain}[i] = 1$ ) THEN

5.2.3      $\text{prevChainSize}[i] = \text{chainSize}[i]$

5.2.4 ELSE  $\text{prevChainSize}[i] = 0$

5.2.5 esegui somme prefisse su  $\text{prevChainSize}[]$

5.2.6 FOR  $i = 1$  TO  $n$  PARDO

5.2.7 IF ( $\text{positionInChain}[i] = 1$ ) THEN

5.2.8      $\text{prevChainSize}[i] = \text{prevChainSize}[i] - \text{chainSize}[i]$

5.2.9 ELSE  $\text{prevChainSize}[i] = 0$

5.3 propaga in Broadcast il valore di  $\text{prevChainSize}$  su ogni catena a partire dalla testa

6. *Calcola l'ordine di rimozione*

6.1 FOR  $i = 1$  TO  $n$  PARDO

6.2      $\text{removal}[i] = \text{prevChainSize}[i] + \text{positionInChain}[i]$

7. *Crea il codice*

7.1 FOR  $i = 1$  TO  $n$  PARDO

7.2 IF ( $\text{removal}[i] \leq n - 2$ ) THEN

7.3      $\text{code}[\text{removal}[i]] = \text{parent}[i]$

Per i Teoremi 1.3.11 e 1.3.12 il passo 1 può essere eseguito su una PRAM EREW in tempo  $O(\log n)$  con  $n/\log n$  processori; analoghe le prestazioni del passo 2 in forza dei teoremi 1.3.5 e 1.3.6.

Nel realizzare il ciclo di cui al passo 3 è necessaria particolare attenzione per evitare letture concorrenti: sfruttando la struttura di  $T_R$  si può far leggere il valore del nodo padre, prima a tutti i figli sinistri e soltanto in seguito a tutti i figli destri. Il numero totale delle operazioni eseguite è quindi  $O(|V(T_R)|) = O(n)$ , e con un attenta applicazione del Principio di Scheduling di Brent

(Teorema 1.3.1) le si può eseguire con  $n/\log n$  processori in tempo  $O(\log n)$ . Una semplice applicazione del medesimo teorema permette di ottenere stesse prestazioni anche per il ciclo presente nel passo 4, inoltre il Teorema 1.3.3 assicura tale efficienza anche per l'operazione di List Ranking.

Nel passo 5 sono presenti, oltre ai tre cicli facilmente trattabili con Principio di Scheduling di Brent (Teorema 1.3.1), operazioni di Broadcast (Teorema 1.3.4) e somme prefisse (Teorema 1.3.2); l'intero passo può essere eseguito in tempo  $O(\log n)$  senza letture o scritture concorrenti con  $n/\log n$  processori.

Applicando ancora il Teorema 1.3.1 ai cicli dei passi 6 e 7 si mostra, infine, che l'intero algoritmo può essere eseguito su PRAM EREW in tempo  $O(\log n)$  con  $n/\log n$  processori, ha quindi un costo computazionale pari a  $O(n)$  ed è pertanto ottimo.

#### 4.1.2 Terzo codice di Neville

Come nel seriale, anche qui è possibile ottenere un algoritmo ottimo per il calcolo del terzo codice di Neville apportando alcune modifiche a quello per il calcolo del codice di Prüfer.

La prima differenza consiste nella necessità di radicare nella foglia di etichetta massima, la seconda differenza sta nel fatto che la ricerca del massimo nel sottoalbero di ciascun nodo si limita all'analisi delle etichette delle foglie.

Per chiarezza è stato omesso il dettaglio dei passi che non subiscono alcuna variazione rispetto all'algoritmo di codifica dei codici di Prüfer.

## ALGORITMO CODIFICA NEVILLE III PARALLELO OTTIMO

**INPUT:** Un albero  $T = (V, E)$  non radicato etichettato su  $[1..n]$ .

**OUTPUT:** Il vettore `code` di  $n - 2$  interi in  $[1..n]$  rappresentante il terzo codice di Neville di  $T$ .

1. *Radica nella foglia di etichetta massima*
  - 1.1 *Ordina le foglie per etichette crescenti*
    - 1.1.1 FOR  $i = 1$  TO  $n$  PARDO
    - 1.1.2 IF ( $i$  è foglia) THEN
    - 1.1.3     `leaf`[ $i$ ] = 1
    - 1.1.4 ELSE `leaf`[ $i$ ] = 0
    - 1.1.5 esegui somme prefisse su `leaf`[]
    - 1.1.6 `leafcount` = `leaf`[ $n$ ]
    - 1.1.7 FOR  $i = 1$  TO  $n$  PARDO
    - 1.1.8 IF ( $i$  è foglia) THEN
    - 1.1.9     `leaevs`[`leaf`[ $i$ ]] =  $i$
  - 1.2 calcola il Tour di Eulero di  $T$
  - 1.3 radica  $T$  in `leaves`[`leafcount`] ottenendo,  $\forall i$ , `parent`[ $i$ ]
2. *Calcola l'etichetta della massima foglia nel sottoalbero di ogni nodo*
  - 2.1 FOR  $i = 1$  TO  $n$  PARDO
  - 2.2 IF ( $i$  è foglia) THEN
  - 2.3     `label`[ $i$ ] =  $i$
  - 2.4 ELSE
  - 2.5     `label`[ $i$ ] = 0
  - 2.6 calcola l'albero regolare  $T_R$  corrispondente a  $T$
  - 2.7 per ogni nodo  $i$  calcola `maxleaf`[ $i$ ], il massimo valore di `label` nel sottoalbero di  $T$  radicato in  $i$ , utilizzando il Rake su  $T_R$
3. *Ogni nodo legge il `maxleaf` del padre*
  - 3.1 FOR  $i = 1$  TO  $n$  PARDO
  - 3.2     `parentmaxleaf`[ $i$ ] = `maxleaf`[`parent`[ $i$ ]]

4. *Crea le catene*

4.1 FOR  $i = 1$  TO  $n$  PARDO

4.2     IF ( $\text{parentmaxleaf}[i] = \text{maxleaf}[i]$ ) THEN

4.3          $\text{chain}[i] = \text{parent}[i]$

4.4     ELSE

4.5          $\text{chain}[i] = \text{NULL}$

4.6 esegue il List Ranking su  $\text{chain}[]$  ottenendo, per ogni nodo  $i$ , il valore  $\text{positionInChain}[i]$

5. *Calcola il numero di elementi presenti nelle catene precedenti*

6. *Calcola l'ordine di rimozione*

7. *Crea il codice*

Le istruzioni dei passi 1 e 2 si differenziano da quelle dell'algoritmo di codifica dei codici di Prüfer per l'introduzione di alcuni cicli e per l'applicazione della tecnica delle somme prefisse. Utilizzando i teoremi 1.3.1 e 1.3.2, tali operazioni possono essere eseguite con  $n/\log n$  processori in tempo  $O(\log n)$  senza letture o scritture concorrenti.

I passi 3 e 4 differiscono da quelli dell'algoritmo precedente soltanto per i nomi di alcune variabili, ed i restanti sono completamente identici; valgono quindi le analisi proposte precedentemente.

In definitiva l'intero algoritmo può essere eseguito su PRAM EREW in tempo  $O(\log n)$  con  $n/\log n$  processori, ha quindi un costo computazionale pari a  $O(n)$  ed è pertanto ottimo.

### 4.1.3 Secondo codice di Neville

Dopo aver radicato l'albero nel centro di etichetta massima l'algoritmo procede calcolando per ciascun nodo  $i$  la distanza dalla radice  $d(i)$ , il massimo

valore di  $d$  nel suo sottoalbero, indicato con  $maxd(i)$  ed infine, il livello  $l(i)$  come differenza tra  $maxd(i)$  e  $d(i)$ .

L'ordine di rimozione dei nodi viene calcolato tramite l'ordinamento lessicografico della coppia  $L(i) = (l(i), i)$ .

### ALGORITMO CODIFICA NEVILLE II PARALLELO

**INPUT:** Un albero  $T = (V, E)$  non radicato etichettato su  $[1..n]$ .

**OUTPUT:** Il vettore `code` di  $n - 1$  interi in  $[1..n]$  rappresentante il secondo codice di Neville di  $T$ .

1. *Radica nel centro di etichetta massima*
  - 1.1 calcola i centri di  $T$
  - 1.2 calcola il Tour di Eulero di  $T$
  - 1.3 radica  $T$  nel centro di etichetta massima ottenendo,  $\forall i$ , `parent[i]`
2. *Calcola il livello di ciascun nodo*
  - 2.1 per ogni nodo  $i$  calcola `d[i]`, la distanza di  $i$  dalla radice, utilizzando il Tour di Eulero precedentemente calcolato
  - 2.2 calcola l'albero regolare  $T_R$  corrispondente a  $T$
  - 2.3 per ogni nodo  $i$  calcola `maxd[i]`, il massimo valore di `d` nel sottoalbero di  $T$  radicato in  $i$ , utilizzando il Rake su  $T_R$
  - 2.4 FOR  $i = 1$  TO  $n$  PARDO
  - 2.5     `l[i] = maxd[i] - d[i]`
3. *Ordina i nodi*
  - 3.1 FOR  $i = 1$  TO  $n$  PARDO
  - 3.2     `L[i] = (l[i], i)`
  - 3.3 crea un vettore `S` delle etichette dei nodi, in ordine crescente rispetto al valore di `L`

#### 4. Crea il codice

```
4.1 FOR  $i = 1$  TO  $n - 1$  PARDO
4.2     code[ $i$ ] = parent[S[ $i$ ]]
```

Il passo 1 può essere eseguito su PRAM EREW in tempo  $O(\log n)$  con  $n/\log n$  processori, dal momento che per i Teoremi 1.3.7, 1.3.11 e 1.3.12 tutte le operazioni in esso coinvolte possono essere eseguite con tali prestazioni.

Applicando i Teoremi 1.3.13, 1.3.5, 1.3.6 e il Principio di Scheduling di Brent (Teorema 1.3.1) al ciclo, anche il passo 2 risulta altrettanto efficiente. Analoghe prestazioni hanno i cicli presenti nei passi 3 e 4, sempre in forza del Teorema 1.3.1.

L'unica operazione che ha un costo più che lineare è l'ordinamento presente all'interno del passo 3 il quale richiede  $O(\log n)$  e  $n$  processori (Teorema 1.3.8). Il costo computazionale è quindi pari a  $O(n \log n)$ .

#### 4.1.4 Codici Stack-Queue

Come fatto nel seriale, anche nel parallelo è possibile ottenere un algoritmo per la codifica dei codici Stack-Queue a partire da quello per il secondo codice di Neville. Diversamente dal seriale, però, in questo contesto non è affatto semplice individuare l'ordine di eliminazione delle foglie, dal momento che tale eliminazione non avviene sequenzialmente ma parallelamente.

Per questo motivo modifichiamo la definizione delle coppie associate ai nodi vista nel capitolo precedente, utilizzando come secondo elemento non più l'ordine di eliminazione ma l'etichetta della foglia più distante. Qualora

le foglie di massima distanza nel sottoalbero di un nodo siano più di una, si sceglie quella di etichetta massima.

Per ogni nodo  $i$  la coppia ad esso associata risulta quindi essere  $M(i) = (l(i), m(i))$  dove  $m(i)$  è appunto la foglia di massima etichetta tra quelle a maggior distanza da  $i$  nel suo sottoalbero. Riportiamo in Figura 4.1 l'esempio del capitolo precedente con i valori della coppia appena definita.

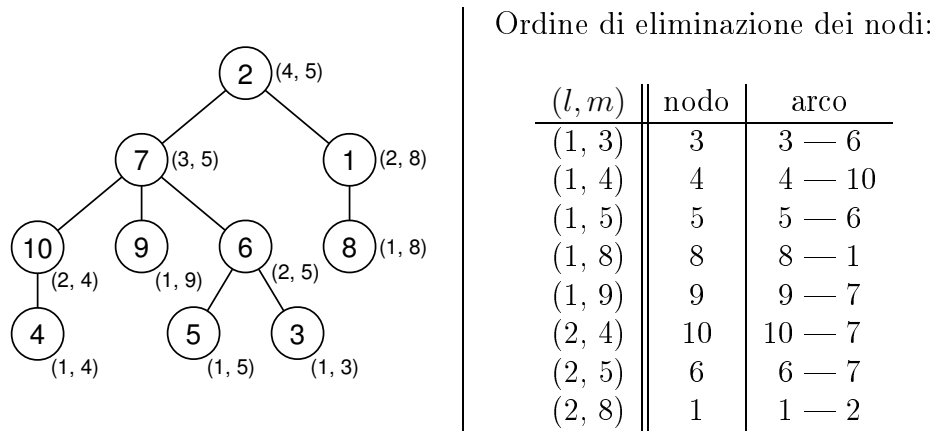


Figura 4.1: Esempio di ordinamento dei nodi per il codice Stack-Queue.

Dimostriamo che l'ordine di eliminazione dei nodi realizzato dal codice rispetta l'ordine lessicografico della coppia appena definita.

**Proposizione 4.1.1.** *Se  $M(i) < M(j)$  allora  $i$  viene eliminato prima di  $j$ .*

*Dimostrazione.* Chiaramente se  $l(i) < l(j)$  nel codice  $i$  viene eliminato prima di  $j$ , dimostriamo che la tesi vale anche quando  $l(i) = l(j)$ , procedendo per induzione sul valore del livello dei due nodi.

Passo base:  $l(i) = l(j) = 1$  allora i nodi sono entrambi foglie quindi  $m(i) = i$  e  $m(j) = j$ . Essendo  $M(i) < M(j)$  deve valere  $i < j$  e quindi  $i$  sarà eliminato prima di  $j$  per definizione del codice.

Induzione: assumendo per ipotesi induttiva che la tesi sia vera per i nodi di livello  $l - 1$ , dimostriamola per  $i$  e  $j$  di livello  $l$ . Siamo  $M(i) = (l, x)$  e  $M(j) = (l, y)$ ,  $x$  ed  $y$  saranno di livello certamente inferiore ad  $i$  e  $j$ . Nel cammino tra  $i$  e  $x$  sicuramente comparirà un nodo di livello  $l - 1$ , denominiamolo  $i'$ ;  $M(i')$  sarà necessariamente  $(l - 1, x)$ , infatti, se per assurdo la foglia di massima etichetta tra quelle a maggior distanza da  $i'$  fosse  $x' \neq x$  allora  $x'$  disterebbe più di  $x$  anche da  $i$  o quantomeno avrebbe etichetta maggiore, quindi  $m(i)$  dovrebbe essere  $x'$  e non  $x$ .

Notiamo inoltre che  $i'$  è certamente l'ultimo figlio ad essere eliminato, infatti, per ogni altro figlio  $i''$  di  $i$  di livello  $l - 1$  si avrà  $M(i'') = (l - 1, x_{i''})$ , con  $x_{i''}$  certamente minore di  $x$ , altrimenti  $m(i)$  sarebbe stato  $x_{i''}$ , quindi  $M(i'') < M(i')$ . Dal momento che per i nodi di livello  $l - 1$  la tesi è valida per ipotesi induttiva,  $i''$  sarà eliminato prima di  $i'$ .

Analogamente per  $j$  esisterà un  $j'$  ultimo figlio ad essere eliminato, tale che  $M(j') = (l - 1, y)$ . Dal momento che

$$M(i) < M(j) \quad \Rightarrow \quad x < y \quad \Rightarrow \quad M(i') < M(j')$$

$i'$  verrà eliminato prima di  $j'$  e quindi  $i$  diverrà foglia prima di  $j$ , il che dimostra la tesi. □

L'algoritmo procede radicando l'albero in uno qualsiasi dei centri dell'albero, calcolando il livello di ogni nodo  $i$ , il valore di  $m(i)$  ed infine ordinando le coppie  $M(i) = (l(i), m(i))$ .

## ALGORITMO CODIFICA STACK-QUEUE PARALLELO

**INPUT:** Un albero  $T = (V, E)$  non radicato etichettato su  $[1..n]$ .

**OUTPUT:** Il vettore `code` di  $n - 2$  interi in  $[1..n]$  rappresentante il codice Stack-Queue di  $T$ .

1. *Radica in un centro*
2. *Calcola il livello di ciascun nodo*
3. *Calcola la foglia a distanza massima da ciascun nodo*
  - 3.1 FOR  $i = 1$  TO  $n$  PARDO
  - 3.2 IF ( $i$  è foglia) THEN
  - 3.3      $D[i] = (d[i], i)$
  - 3.4 ELSE
  - 3.5      $D[i] = (0, 0)$
  - 3.6 per ogni nodo  $i$  calcola `maxdleaf[i]`, il nodo nel sottoalbero di  $T$  radicato in  $i$  per cui è massimo il valore della coppia  $D$ , utilizzando il Rake su  $T_R$
4. *Ordina i nodi*
  - 4.1 FOR  $i = 1$  TO  $n$  PARDO
  - 4.2      $M[i] = (l[i], \text{maxdleaf}[i])$
  - 4.3 crea un vettore  $S$  delle etichette dei nodi, in ordine crescente rispetto al valore di  $M$
5. *Crea il codice*
  - 5.1 FOR  $i = 1$  TO  $n - 2$  PARDO
  - 5.2     `code[i] = parent[S[i]]`

I passi 1 e 2 sono invariati rispetto all'algorithmo per il calcolo del secondo codice di Neville.

Tanto il ciclo (Teorema 1.3.1) quanto il calcolo del massimo (Teorema 1.3.6) del passo 3 possono essere eseguiti su PRAM EREW in tempo  $O(\log n)$

con  $n/\log n$  processori. Stesse prestazioni hanno i cicli presenti nei passi 4 e 5, sempre in forza del Teorema 1.3.1. I valori  $d[i]$  ed  $l[i]$  utilizzati nei passi 3 e 4 sono quelli calcolati all'interno del passo 2.

Anche per questo algoritmo, l'unica operazione ad avere un costo più che lineare è l'ordinamento presente all'interno del passo 4 il quale richiede  $O(\log n)$  e  $n$  processori (Teorema 1.3.8). Il costo computazionale è quindi pari a  $O(n \log n)$ .

## 4.2 Algoritmi di decodifica

Per decodificare si procederà ricostruendo il vettore  $\mathbf{A}$  dei nodi eliminati, l'albero  $T$  sarà ricreato unendo i nodi di tale vettore ai corrispondenti nel codice:  $E(T) = \{(\mathbf{A}[i], \text{code}[i]) : 1 \leq i < n\}$ .

Nel precedente capitolo si faceva uso di una funzione per il calcolo del grado di ciascun nodo nell'albero a partire dal codice, in questo contesto tale informazione non risulta altrettanto interessante dal momento che non è più possibile decrementare i gradi dei nodi passo dopo passo per individuare l'ordine in cui diventano foglie.

Si sostituirà tale approccio con uno basato sul calcolo dell'ultima occorrenza di ciascun nodo nel codice, da quel punto in poi, infatti, il nodo è foglia ed è quindi candidato ad apparire nel vettore  $\mathbf{A}$ .

### 4.2.1 Calcolo dell'ultima occorrenza

Dato un vettore `code` di  $n - 1$  interi in  $[1..n]$  vediamo come sia possibile dedurre l'indice dell'ultima occorrenza di ciascun intero nel vettore:

$$\text{last}[i] = \begin{cases} \max\{j : \text{code}[j] = i\} & \text{se } i \text{ compare in code} \\ 0 & \text{se } i \text{ non compare in code} \end{cases}$$

#### ALGORITMO CALCOLO DELL'ULTIMA OCCORRENZA

**INPUT:** Un vettore `code` di  $n - 1$  interi in  $[1..n]$ .

**OUTPUT:** Il vettore `last` contenente l'indice dell'ultima occorrenza di ogni intero nel vettore `code`

1. FOR  $i = 1$  TO  $n$  PARDO
2.     `last`[ $i$ ] = 0
3. FOR  $j = 1$  TO  $n$  PARDO
4.     `last`[`code`[ $j$ ]] =  $j$

Dopo aver inizializzato il vettore, l'algoritmo, per realizzare  $\text{last}[i] = \max\{j : \text{code}[j] = i\}$  richiede ad ogni processore  $j$  di scrivere parallelamente il valore del suo indice nella posizione  $i = \text{code}[j]$  del vettore `last`; se l'algoritmo venisse eseguito su una PRAM ERCW, che in caso di scrittura concorrente garantisce priorità al processore di indice maggiore, l'algoritmo calcolerebbe esattamente il valore del vettore `last` in tempo  $O(1)$  con  $n$  processori.

Il comportamento di una PRAM ERCW con priorità può essere simulato su una PRAM EREW pagando un costo temporale aggiuntivo pari ad  $O(\log n)$  (Teorema 1.3.9), in definitiva l'algoritmo richiede  $n$  processori e  $O(\log n)$  tempo.

Per completezza precisiamo che il vettore `last` può essere computato, con medesime prestazioni, ricorrendo ad un'operazione di ordinamento sulla coppia  $(\text{code}[i], i)$  [7].

## 4.2.2 Terzo codice di Neville

In questo codice ogni nodo interno viene eliminato non appena diviene foglia, quindi per tutti gli  $i$  per i quali  $\text{last}[i] \neq 0$ , si ha che  $A[\text{last}[i]+1] = i$ , le restanti locazioni di  $A$  saranno occupate dalle foglie ordinate per etichette crescenti.

### ALGORITMO DECODIFICA NEVILLE III PARALLELO

**INPUT:** Un vettore `code` di  $n - 2$  interi in  $[1..n]$ .

**OUTPUT:** L'albero  $T = (V, E)$  etichettato su  $[1..n]$  corrispondente a `code`.

1.  $n = \text{code.length} + 2$
2. *Calcola l'ultima occorrenza di ogni nodo nel codice*
3. *Ordina le foglie per etichette crescenti*
  - 3.1 FOR  $i = 1$  TO  $n$  PARDO
  - 3.2     IF  $(\text{last}[i] = 0)$  THEN
  - 3.3          $\text{leaf}[i] = 1$
  - 3.4     ELSE  $\text{leaf}[i] = 0$
  - 3.5 esegui somme prefisse su `leaf[]`
  - 3.6  $\text{leafcount} = \text{leaf}[n]$
  - 3.7 FOR  $i = 1$  TO  $n$  PARDO
  - 3.8     IF  $(\text{last}[i] = 0)$  THEN
  - 3.9          $\text{leaevs}[\text{leaf}[i]] = i$
4.  $\text{code}[n - 1] = \text{leaevs}[\text{leafcount}]$

5. *Crea il vettore A*

5.1 FOR  $i = 1$  TO  $n - 1$  PARDO

5.2      $A[i] = 0$

5.3 *Assegna le posizioni per i nodi interni*

5.3.1 FOR  $i = 1$  TO  $n$  PARDO

5.3.2     IF ( $\text{last}[i] \neq 0$ ) THEN

5.3.3          $A[\text{last}[i]+1] = i$

5.4 *Numera gli elementi di A non ancora assegnati*

5.4.1 FOR  $i = 1$  TO  $n - 1$  PARDO

5.4.2     IF ( $A[i] = 0$ ) THEN

5.4.3          $\text{hole}[i] = 1$

5.4.4     ELSE  $\text{hole}[i] = 0$

5.4.5 esegui somme prefisse su  $\text{hole}[]$

5.4.6  $\text{holecount} = \text{hole}[n - 1]$

5.4.7 FOR  $i = 1$  TO  $n - 1$  PARDO

5.4.8     IF ( $A[i] = 0$ ) THEN

5.4.9          $\text{holes}[\text{hole}[i]] = i$

5.5 *Completa A inserendo le foglie*

5.5.1 FOR  $i = 1$  TO  $\text{holecount}$  PARDO

5.5.2      $A[\text{holes}[i]] = \text{leaevs}[i]$

6. *Crea l'albero*

6.1  $T = ([1..n], \emptyset)$

6.2 FOR  $i = 1$  TO  $n - 1$  PARDO

6.3      $E(T) = E(T) \cup (A[i], \text{code}[i])$

I passi 1 e 4 possono essere eseguiti da un singolo processore in tempo costante, l'analisi del passo 2 è stata presentata nel paragrafo precedente.

Il passo 3 è identico al passo 1.1 dell'algoritmo parallelo di codifica per il terzo codice di Neville, può quindi essere eseguito in tempo  $O(\log n)$  su una PRAM EREW con  $n/\log n$  processori, e così anche il passo 5.4. Per

i restanti cicli del passo 5 si ottengono medesime prestazioni applicando il Principio di Scheduling di Brent (Teorema 1.3.1).

Anche al ciclo del passo 6 si può applicare il Teorema 1.3.1 riuscendo così a costruire l'albero su una PRAM EREW con  $n/\log n$  processori in tempo  $O(t_{add} \log n)$ , dove  $t_{add}$  è il tempo necessario ad un processore per aggiungere un arco all'albero. Nel caso di una semplice rappresentazione tramite vettore di archi, ogni processore può aggiungere un arco in tempo  $O(1)$  senza scritture concorrenti: in questo caso il tempo necessario per il passo 6 è quindi  $O(\log n)$ .

In definitiva l'unica operazione non ottima presente nell'algoritmo è il calcolo dell'ultima occorrenza di ogni nodo nel codice svolto nel passo 2.

### 4.2.3 Codici Stack-Queue

Per ricostruire l'ordine di eliminazione dei nodi realizzato da questo codice si devono anzitutto inserire in **A** le foglie dell'albero iniziale in ordine crescente, di seguito compariranno gli altri nodi, nell'ordine in cui diventano foglie, ovvero in ordine di ultima apparizione nel codice.

#### ALGORITMO DECODIFICA STACK-QUEUE PARALLELO

**INPUT:** Un vettore `code` di  $n - 2$  interi in  $[1..n]$ .

**OUTPUT:** L'albero  $T = (V, E)$  etichettato su  $[1..n]$  corrispondente a `code`.

1.  $n = \text{code.length} + 2$
2.  $\text{code}[n - 1] = \text{code}[n - 1]$
3. *Calcola l'ultima occorrenza di ogni nodo nel codice*
4. *Ordina le foglie per etichette crescenti inserendole in **A***

- 4.1 FOR  $i = 1$  TO  $n$  PARDO
- 4.2     IF ( $\text{last}[i] = 0$ ) THEN
- 4.3          $\text{leaf}[i] = 1$
- 4.4     ELSE  $\text{leaf}[i] = 0$
- 4.5 esegui somme prefisse su  $\text{leaf}[]$
- 4.6  $\text{leafcount} = \text{leaf}[n]$
- 4.7 FOR  $i = 1$  TO  $n$  PARDO
- 4.8     IF ( $\text{last}[i] = 0$ ) THEN
- 4.9          $A[\text{leaf}[i]] = i$
5. *Marca gli elementi del codice che sono ultime occorrenze di un nodo*
- 5.1 FOR  $i = 1$  TO  $n$  PARDO
- 5.2      $\text{isLast}[i] = 0$
- 5.3 FOR  $i = 1$  TO  $n$  PARDO
- 5.4     IF ( $\text{last}[i] \neq 0$ ) THEN
- 5.5          $\text{isLast}[\text{last}[i]] = 1$
6. *Completa A inserendo i nodi marcati*
- 6.1 esegui somme prefisse su  $\text{isLast}[]$  ottenendo  $\text{lastCount}[]$
- 6.2 FOR  $i = 1$  TO  $n - 1$  PARDO
- 6.3     IF ( $\text{isLast}[i] = 1$ ) THEN
- 6.4          $A[\text{leafcount} + \text{lastCount}[i]] = i$
7. *Crea l'albero*

I passi 1 e 2 richiedono tempo costante ed un singolo processore. Per i passi 3 e 7 valgono le osservazioni fatte nell'analisi dell'algoritmo precedente ed anche il passo 4 non differisce dall'ordinamento delle foglie fatto sopra, se non per il nome di una variabile.

Entrambi i cicli del passo 5 e quello del passo 6 possono essere trattati con il Principio di Scheduling di Brent (Teorema 1.3.1), utilizzando infine il

Teorema 1.3.2 per l'operazione di somme prefisse, si deduce che tutto l'algoritmo può essere implementato su una PRAM EREW con  $n/\log n$  processori in tempo  $O(\log n)$  ad eccezione del calcolo l'ultima occorrenza di ogni nodo nel codice.

#### 4.2.4 Secondo codice di Neville

Anche nel parallelo, come nel seriale, per ricostruire l'ordine di eliminazione dei nodi realizzato da questo codice, è necessario prima comprendere il livello di ogni nodo, per poter poi ordinare rispetto alla coppia  $(l(i), i)$ .

#### ALGORITMO DECODIFICA NEVILLE II PARALLELO

**INPUT:** Un vettore `code` di  $n - 1$  interi in  $[1..n]$ .

**OUTPUT:** L'albero  $T = (V, E)$  etichettato su  $[1..n]$  corrispondente a `code`.

1.  $n = \text{code.length} + 1$
2. *Calcola l'ultima occorrenza di ogni nodo nel codice*
3. *Marca gli elementi del codice che sono ultime occorrenze di un nodo*
4. *Calcola il livello di ogni nodo*
  - 4.1 `actLevel = 1`
  - 4.2 `actNode = 1`
  - 4.3 `levelSize = |\{i : last[i] = 0\}|`
  - 4.4 `WHILE (actNode < n) DO`
  - 4.5     `FOR i = actNode TO actNode+levelSize PARDO`
  - 4.6         `IF (isLast[i] = 1) THEN`
  - 4.7             `l[i] = actLevel`
  - 4.8             `nextLevelSize++`
  - 4.9     `actLevel++`
  - 4.10    `actNode = act + levelSize`

4.11      `levelSize = nextLevelSize`

5. *Ordina i nodi*

5.1 FOR  $i = 1$  TO  $n$  PARDO

5.2       $L[i] = (l[i], i)$

5.3 crea il vettore **A** delle etichette dei nodi, in ordine crescente rispetto al valore di **L**

6. *Crea l'albero*

Per i passi 1, 2, 3 e 6 si rimanda all'analisi dell'algoritmo precedente.

Il passo 5 si compone di un ciclo dal costo lineare (Teorema 1.3.1) e di un ordinamento che, in forza del Teorema 1.3.8 richiede  $O(\log n)$  tempo ed  $n$  processori per essere implementato su una PRAM EREW.

Analizziamo ora nel dettaglio il passo 4: le prime due istruzioni possono essere eseguite da un singolo processore in tempo costante; per capire il valore iniziale di `levelSize` è sufficiente contare le foglie dell'albero iniziale, tale operazione è stata fatta più volte nei precedenti algoritmi e richiede  $O(\log n)$  tempo e  $n/\log n$  processori.

Il ciclo `WHILE` è un ciclo seriale il cui numero di iterazioni è pari al numero di livelli dell'albero, sia esso  $H(T)$ . Le tre assegnazioni presenti alla fine del suo corpo possono essere eseguite da un singolo processore in tempo costante, quindi ciò che determina il costo di tale corpo è il ciclo `FOR` parallelo presente al suo interno.

In tale ciclo si identificano i nodi di un livello sapendo quanti erano i nodi di livello precedente; quindi ad ogni iterazione lavoreranno in parallelo al più  $W(T)$  processori, dove  $W(T)$  è il massimo numero di nodi di medesimo livello. Il fatto che all'interno del corpo si eseguano una lettura ed una

scrittura concorrente, fa sì che il tempo necessario ad ogni iterazione sia  $O(\log W(T))$  (Teorema 1.3.9).

In definitiva su una PRAM EREW il ciclo WHILE necessita di un tempo pari a  $O(H(T) \log W(T))$  e di  $W(T)$  processori, mentre le restanti istruzioni dell'algoritmo possono essere tutte eseguite in  $O(\log n)$  tempo con al più  $n$  processori.

#### 4.2.5 Codici di Prüfer

Per riuscire a decodificare i codici di Prüfer non sarà sufficiente conoscere l'ultima apparizione di ogni nodo nel codice, perché in fase di codifica l'eliminazione di un nodo non è in stretta relazione con il momento in cui esso è divenuto foglia.

L'approccio qui proposto si deve a Chan, Liu e Wang [7] e si basa sulla determinazione, per ogni nodo  $i$ , del numero di nodi di etichetta minore di  $i$  che diventano foglie prima che lo diventi  $i$ .

$$\text{prev}[i] = |\{j : j < i \text{ e } \text{last}[j] < \text{last}[i]\}|$$

È immediato notare che il nodo  $i$  verrà eliminato nel passo successivo a quello nel quale diviene foglia, se e solo se non ci sono, in quel momento, foglie di etichetta minore di  $i$ ; ovvero se il numero di nodi minori di  $i$  che diventano foglie prima di lui è minore del passo nel quale si vorrebbe eliminare  $i$ . Formalmente quindi  $A[\text{last}[i]+1] = i$  se e solo se  $\text{last}[i] \geq \text{prev}[i]$ .

Le restanti locazioni di  $A$  saranno occupate dai nodi rimasti non ancora assegnati, in ordine crescente; dal momento che per tali nodi vale  $\text{prev}[i] >$

$\text{last}[i]$  di certo non si rischierà di assegnare  $i$  ad una locazione di  $\mathbf{A}$  precedente al passo in cui  $i$  diventa foglia.

### ALGORITMO DECODIFICA PRÜFER PARALLELO

**INPUT:** Un vettore `code` di  $n - 2$  interi in  $[1..n]$ .

**OUTPUT:** L'albero  $T = (V, E)$  etichettato su  $[1..n]$  corrispondente a `code`.

1.  $n = \text{code.length} + 1$
2.  $\text{code}[n - 1] = n$
3. *Calcola l'ultima occorrenza di ogni nodo nel codice*
4. *Calcola il vettore `prev[]`*
5. *Crea il vettore  $\mathbf{A}$* 
  - 5.1 FOR  $i = 1$  TO  $n - 1$  PARDO
  - 5.2      $\mathbf{A}[i] = 0$
  - 5.3      $\text{used}[i] = 0$
  - 5.4 *Inserisci in  $\mathbf{A}$  i nodi la cui posizione è nota*
    - 5.4.1 FOR  $i = 1$  TO  $n$  PARDO
    - 5.4.2     IF ( $\text{last}[i] \geq \text{prev}[i]$ ) THEN
    - 5.4.3          $\mathbf{A}[\text{last}[i]+1] = i$
    - 5.4.4          $\text{used}[i] = 1$
  - 5.5 *Numera gli elementi di  $\mathbf{A}$  non ancora assegnati ottenendo `holes[]`*
  - 5.6 *Numera i nodi non ancora utilizzati ottenendo `unused[]`*
  - 5.7 *Completa  $\mathbf{A}$  inserendo gli elementi non ancora utilizzati*
    - 5.7.1 FOR  $i = 1$  TO `holecount` PARDO
    - 5.7.2      $\mathbf{A}[\text{holes}[i]] = \text{unused}[i]$
6. *Crea l'albero*

I passi 1, 2, 3 e 6 sono già stati analizzati negli algoritmi precedenti.

Il calcolo del vettore `prev` al passo 4 può essere ricondotto al conteggio della dominanza tra i due insiemi  $S = T = \{(i, \text{last}[i]) : i \in [1..n]\}$ , per il

Teorema 1.3.10 richiede quindi  $O(\log n)$  tempo e  $n$  processori su una PRAM EREW.

I tre cicli presenti nel passo 5 possono, applicando il Teorema 1.3.1, essere eseguiti su una PRAM EREW in tempo  $O(\log n)$  con  $n/\log n$  processori, così anche la numerazione degli elementi di  $A$  non ancora assegnati, per il dettaglio di questa operazione si rimanda all'algoritmo parallelo per la decodifica del terzo codice di Neville. Analogamente si può realizzare la numerazione dei nodi non ancora utilizzati, discriminandoli grazie al vettore `used`.

In definitiva l'implementazione dell'algoritmo su PRAM EREW richiede  $n$  processori e  $O(\log n)$  tempo.

### 4.3 Conclusioni

A conclusione di questo capitolo si può notare che soltanto due degli algoritmi proposti hanno un costo asintotico ottimo, si impongono quindi alcune osservazioni di carattere pratico.

Nel 1998 Deo, Kumar e Kumar [10] hanno proposto un algoritmo di decodifica il cui reale costo temporale, su una macchina MasPar MP-1 [22, 4] con 8192 processori, è pressoché costante per codici di alberi il cui numero di nodi non eccede la metà del numero dei processori:  $n \leq 4096$ .

Tale risultato si deve principalmente al fatto che l'algoritmo proposto coinvolge, come uniche istruzioni dal costo asintotico più che lineare, delle istruzioni di ordinamento e gli autori disponevano di un algoritmo, ottimizzato per la loro macchina, in grado di ordinare, in tempo quasi lineare, vettori di dimensioni non superiori a 8192 elementi.

Anche per molti degli algoritmi non ottimi proposti in questo capitolo l'unico tipo di operazione onerosa è l'ordinamento, valgono dunque i medesimi risultati ottenuti da Deo, Kumar e Kumar su MasPar MP-1. In tal modo per la maggior parte degli algoritmi asintoticamente non ottimi presentati si può comunque ottenere un'implementazione dalle prestazioni pressoché lineari. Si noti in particolare che ciò vale anche per il problema del calcolo dell'ultima occorrenza di ogni nodo nel codice, dal momento che esso può essere ricondotto ad un ordinamento.

Riportiamo, nelle Tabelle 4.1 e 4.2, il dettaglio delle caratteristiche degli algoritmi di codifica e di decodifica: il tempo ed il numero di processori ottenuti dall'analisi asintotica, le prestazioni dell'implementazione reale su MasPar MP-1 e le tecniche di calcolo parallelo utilizzate.

	Prüfer	Neville II	Neville III	Stack-Queue
Tempo	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Processori	$n/\log n$	$n$	$n/\log n$	$n$
Prestazioni	Lineari	Pressoché lineari	Lineari	Pressoché lineari
Scheduling di Brent	✓	✓	✓	✓
Tour di Eulero	✓	✓	✓	✓
Calcolo dei centri	-	✓	-	✓
Radicare	✓	✓	✓	✓
Distanza dalla radice	-	✓	-	✓
Albero regolare	✓	✓	✓	✓
Rake	✓	✓	✓	✓
Somme prefisse	✓	-	✓	-
List Ranking	✓	-	✓	-
Broadcast	✓	-	✓	-
Ordinamento	-	✓	-	✓

Tabella 4.1: Caratteristiche degli algoritmi paralleli di codifica.

	Prüfer	Neville II	Neville III	Stack-Queue
Tempo	$O(\log n)$	$O(\log n + H \log W)$	$O(\log n)$	$O(\log n)$
Processori	$n$	$\max(n, W)$	$n$	$n$
Prestazioni	Non lineari	Non lineari	Pressoché lineari	Pressoché lineari
Scheduling di Brent	✓	✓	✓	✓
Somme prefisse	✓	✓	✓	✓
Ordinamento	-	✓	-	-
Dominanza	-	✓	-	-

Tabella 4.2: Caratteristiche degli algoritmi paralleli di decodifica.

## Capitolo 5

# Conclusioni e sviluppi futuri

Nel corso di questa tesi sono stati studiati diversi codici noti in letteratura, in particolare si è focalizzata l'attenzione sui codici di Prüfer, sul secondo e sul terzo codice di Neville e sui codici Stack-Queue.

Dopo aver definito uno schema generale, idoneo a descrivere tutte le codifiche di alberi che procedono per selezione iterativa di alberi, nel Capitolo 2, si è mostrato come ricondurre a tale schema tutti i codici esaminati. Nel Capitolo 3 si sono analizzati i costi degli algoritmi ottenuti adattando lo schema generale a ciascun codice.

Sempre nello stesso capitolo si sono presentati degli algoritmi ottimi di codifica e decodifica per ognuno dei codici. È importante sottolineare come si sia riusciti a mantenere un approccio uniforme: per gli algoritmi di codifica ci si riconduce sempre ad un problema di ordinamento lessicografico di coppie, mentre per quelli di decodifica ci si basa sulla possibilità di dedurre il grado di ciascun nodo dell'albero scorrendo il codice.

La possibilità di codificare e decodificare tutti i codici presenti in letteratura con il medesimo approccio e con identiche prestazioni asintotiche,

evidenzia come, almeno nel contesto del calcolo seriale, tali codici possano essere considerati sostanzialmente equivalenti.

Nel capitolo 4, sono stati proposti gli algoritmi paralleli di codifica e decodifica dei vari codici, il modello di macchina parallela cui si è fatto riferimento è la PRAM EREW. Dall'analisi asintotica e da alcune considerazioni sulle reali possibilità implementative, si è potuto notare come, nonostante soltanto due degli algoritmi proposti abbiano un costo lineare, la maggior parte dei restanti possa in pratica fornire prestazioni pressoché lineari.

Dal momento che, nel contesto del calcolo parallelo, il terzo codice di Neville risulta essere quello che fornisce le migliori prestazioni, un primo interessante sviluppo consisterebbe nel trovare un algoritmo asintoticamente ottimo per la sua decodifica.

A tale proposito si noti che, il problema della decodifica di questo codice è equivalente al problema della determinazione dell'ultima occorrenza di ogni elemento in un vettore di  $n - 1$  interi in  $[1..n]$ . Sappiamo già che la risoluzione efficiente del secondo problema implica la risoluzione efficiente del primo, osserviamo ora come valga anche il viceversa.

Se si disponesse di un algoritmo efficiente  $\mathcal{A}$  per la decodifica del terzo codice di Neville, dato un vettore di  $n - 1$  interi in  $[1..n]$ , lo si potrebbe decodificare ottenendo un albero. Applicando poi, a tale albero, l'algoritmo di codifica sarebbe possibile, invece che ricreare il vettore di partenza, dedurre, per ogni  $i$ ,  $\text{Last}[i]$  come la posizione nella quale verrà eliminato l'ultimo figlio di  $i$ , ovvero il nodo che lo precede nella sua catena. Per far ciò sarebbe sufficiente sostituire il passo 7 dell'algoritmo parallelo di codifica del

terzo codice di Neville con:

7. *Calcola l'ultima occorrenza di ogni nodo nel codice*

```
7.1 FOR  $i = 1$  TO  $n$  PARDO
7.2     last[ $i$ ] = 0
7.3 FOR  $i = 1$  TO  $n$  PARDO
7.4     IF (chain[ $i$ ]  $\neq$  0) THEN
7.5         last[chain[ $i$ ]] = removal[ $i$ ]
```

L'esistenza di un algoritmo dal costo asintotico lineare per decodificare il terzo codice di Neville, implica quindi la possibilità di calcolare in maniera ottima l'ultima occorrenza degli interi in un vettore.

Un'altra interessante direzione di ricerca consiste nel definire nuove codifiche; così come Deo e Micikevicius hanno introdotto i codici Stack-Queue elaborando direttamente gli algoritmi ottimi di codifica e decodifica nel seriale, si potrebbe cercare di definire nuovi codici per i quali sia possibile esibire gli algoritmi di codifica e decodifica ottimi nel parallelo e studiarne poi il comportamento nel contesto del calcolo seriale.

# Notazione

$[a..b]$  l'insieme dei numeri naturali compresi tra  $a$  e  $b$ .

$\mathcal{P}(A)$  l'insieme di tutti i sottoinsiemi di  $A$ .

$T$  un albero  $(V, E)$  etichettato su  $[1..n]$ , eventualmente radicato in  $r \in [1..n]$ .

$V(T)$  l'insieme dei nodi di  $T$ .

$E(T)$  l'insieme degli archi di  $T$ .

$deg(v)$  grado di  $v \in V(T)$ .

foglia un nodo di grado 1, esclusa l'eventuale radice.

$F(T)$  l'insieme delle foglie di  $T$ .

$T-A$  la foresta ottenuta da  $T$  eliminando i nodi in  $A \subseteq V(T)$  e gli archi su essi incidenti.

$()$  la sequenza vuota o il vettore vuoto.

# Bibliografia

- [1] K. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10(2):287–302, 1989.
- [2] R. J. Anderson and G. L. Miller. Deterministic parallel list ranking. *Algorithmica*, 6:859–868, 1991.
- [3] M. J. Atallah, R. Cole, and M. T. Goodrich. Cascading divide-and-conquer: A technique for designing parallel algorithms. *SIAM Journal of Computing*, 18(3):499–532, 1989.
- [4] T. Blank. The MasPar Mp-1 architecture. In *Proceedings of 35th IEEE Computer Society International Conference*, pages 20–24, San Francisco, California, February 1990.
- [5] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.
- [6] A. Cayley. A theorem on trees. *Quarterly Journal of Mathematics*, 23:376–378, 1889.

- [7] H. C. Chen, W. K. Liu, and Y. L. Wang. A parallel algorithm for constructing a labeled tree. *IEEE Transactions on Parallel and Distributed Systems*, 8(12):1236–1240, December 1997.
- [8] H. C. Chen and Y. L. Wang. An efficient algorithm for generating Prüfer codes from labelled trees. *Theory of Computing Systems*, 33(1):97–105, 2000.
- [9] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.
- [10] N. Deo, N. Kumar, and V. Kumar. Parallel generation of random trees and connected graphs. *Congressus Numerantium*, 130:7–18, 1998.
- [11] N. Deo and P. Micikevicius. Parallel algorithms for computing Prüfer-Like codes of labeled trees. Technical report, CS-TR-01-06, Department of Computer Science, University of Central Florida, Orlando, 2001.
- [12] N. Deo and P. Micikevicius. A new encoding for labeled trees employing a stack and a queue. *Bulletin of ICA*, 34:77–85, 2002.
- [13] W. Edelson and M. L. Gargano. Feasible encodings for GA solutions of constrained minimal spanning tree problems. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, page 754, Las Vegas, Nevada, USA, 2000.
- [14] M. J. Fischer and R. E. Ladner. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

- [15] M. Gen and G. Zhou. A note on genetic algorithms for degree-constrained spanning tree problems. *Networks: an International Journal*, 30:91–95, 1997.
- [16] R. Greenlaw, M. M. Halldórsson, and R. Petreschi. On computing Prüfer codes and their corresponding trees optimally in parallel. In *Proceedings of JIM'2000 - Journées de l'Informatique Messine*, Metz, France, 2000.
- [17] R. Greenlaw and R. Petreschi. Computing Prüfer codes efficiently in parallel. *Discrete Applied Mathematics*, 102(3):205–222, 2000.
- [18] X. He. Efficient parallel algorithms for solving some tree problems. In *Proceeding of 24th Allerton Conference on Communication, Control and Computing*, pages 777–786, Allerton House, Monticello , Illinois, 1986.
- [19] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.
- [20] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 869–941. Amsterdam, Netherlands, 1990.
- [21] W. T. Lo and S. Peng. The optimal location of a structured facility in a tree network. *Journal of Parallel Algorithms and Applications*, 2:43–60, 1994.
- [22] MasPar Computer Corporation. *MasPar Mp-1 Architecture Specifications*, 1991.

- [23] G. L. Miller and S. H. Teng. Systematic method for tree based parallel algorithm development. In *Second International Conference on Supercomputing*, pages 392–403, Santa Clara, California, 1987.
- [24] J. W. Moon. *Counting Labeled Trees*. William Clowes and Sons, London, 1970.
- [25] E. H. Neville. The codifying of tree-structure. In *Proceedings of Cambridge Philosophical Society*, volume 49, pages 381–385, 1953.
- [26] H. Prüfer. Neuer Beweis eines Satzes über Permutationen. *Archiv der Mathematik und Physik*, 27:142–144, 1918.
- [27] T. M. Przytycka. *Parallel Algorithms for Tree Construction and Related Problems*. PhD thesis, The University of British Columbia, November 1990. UBC TR 90-28.
- [28] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal of Computing*, 14(4):862–874, 1985.